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:

\[
\text{while } x \neq \text{nil} \text{ do } x = x.\text{next} \text{ od;}
\]

Which preconditions guarantee safe termination?
A simple example

Consider the following list traversal program:

\[
\text{while } x \neq \text{nil do } x = x.next \text{ od;}
\]

Which preconditions guarantee safe termination?

\[
x = \text{nil}
\]
A simple example

Consider the following list traversal program:

\[
\text{while } x \neq \text{nil do } x = x.\text{next} \text{ od;}
\]

Which preconditions guarantee safe termination?

\[
x = \text{nil}
\]
\[
x \mapsto \text{nil}
\]
A simple example

Consider the following list traversal program:

\[
\text{while } x \neq \text{nil} \text{ do } x = x.next \text{ od;}
\]

Which preconditions guarantee safe termination?

\[
\begin{align*}
x &= \text{nil} \\
x &\mapsto \text{nil} \\
x &\mapsto x' * x' \mapsto \text{nil} \\
\vdots
\end{align*}
\]
A simple example

Consider the following list traversal program:

\[
\text{while } x \neq \text{nil do } x = x.\text{next} \text{ od;}
\]

Which preconditions guarantee safe termination?

\[
x = \text{nil} \\
x \mapsto \text{nil} \\
x \mapsto x' \ast x' \mapsto \text{nil} \\
: \\
\]

Most general solution is an acyclic linked list, given by

\[
x = \text{nil} \; \Rightarrow \; \text{list}(x) \\
x \neq \text{nil} \ast x \mapsto y \ast \text{list}(y) \; \Rightarrow \; \text{list}(x)
\]
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...
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.
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**:

\( \text{...the hypothesis cannot be admitted, even as a hypothesis, unless it be supposed that it would account for the facts or some of them.} \)
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**:

\[ \ldots \text{the hypothesis cannot be admitted, even as a hypothesis, unless it be supposed that it would account for the facts or some of them. The form of inference, therefore, is this:} \]

\[ \text{The surprising fact, } C, \text{ is observed;} \]

\[ \text{But if } A \text{ were true, } C \text{ would be a matter of course,} \]
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.
Overview of our approach


- 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.

• 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.

• 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.

- 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:
  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.
Expressions are either a variable or nil.

Branching conditions $B$ and commands $C$ are given by

\begin{align*}
B & ::= \star | E = E | E \neq E \\
C & ::= \epsilon | x := E; C | x := E.f; C | E.f := E; C | \\
& \text{free}(E); C | x := \text{new}(); C | \\
& \text{if } B \text{ then } C \text{ fi}; C | \text{while } B \text{ do } C \text{ od}; C
\end{align*}
Syntax of programs

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

$$
B ::= \star | E = E | E \not= E
$$

$$
C ::= \epsilon | x := E; C | x := E.f; C | E.f := E; C |
\text{free}(E); C | x := \text{new}(); C |
\text{if } B \text{ then } C \text{ fi}; C | \text{while } B \text{ do } C \text{ od}; C
$$

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

  \[ B ::= \star \mid E = E \mid E \neq E \]

  \[ C ::= \epsilon \mid x := E; C \mid x := E.f; C \mid E.f := E; C \mid \text{free}(E); C \mid x := \text{new}(); C \mid \text{if } B \text{ then } C\text{ fi}; C \mid \text{while } B \text{ do } C\text{ od}; C \]

  where $E$ ranges over expressions, $x$ over variables, $n$ over **field names** and $j$ over $\mathbb{N}$.
- **A program** is given by **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;
  - \(s : \text{Var} \rightarrow \text{Val}\) is a stack;
A program state is either fault or a triple \((C, s, h)\), where
\begin{itemize}
  \item \(C\) is a command;
  \item \(s : \text{Var} \rightarrow \text{Val}\) is a stack;
  \item \(h : \text{Loc} \rightarrow_{\text{fin}} \text{Val}\) is a heap (we write \(\circ\) for union of disjoint heaps).
\end{itemize}
Semantics of programs

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

• \((C, s, h)\) is called safe if there is no computation sequence \((C, s, h) \Rightarrow^* \text{fault}\). And \((C, s, h)\downarrow\) means there is no infinite computation sequence \((C, s, h) \Rightarrow \ldots\).
Semantics of programs

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

- \((C, s, h)\) is called safe if there is no computation sequence \((C, s, h) \rightsquigarrow^* \text{fault}\). And \((C, s, h) \downarrow\) means there is no infinite computation sequence \((C, s, h) \rightsquigarrow \ldots\).

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.
Formulas $F$ are given by

$$F ::= E = E \mid E \neq E \mid \text{emp} \mid E \leftrightarrow E \mid PE \mid F \ast F$$

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

- **Formulas** $F$ are given by

  $$F ::= E = E \mid E \neq E \mid \text{emp} \mid E \mapsto E \mid PE \mid F \ast F$$

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

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

  $$F \Rightarrow Pt$$
Preconditions

• **Formulas** \( F \) are given by

\[
F ::= E = E \mid E \neq E \mid \text{emp} \mid E \mapsto E \mid P E \mid F * F
\]

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

• An **inductive rule** for predicate \( P \) is a rule of the form

\[
F \Rightarrow P t
\]

• **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:

\[
x = E_f[x'/x] \ast (F \ast E \mapsto E)[x'/x] \vdash C'
\]

\[
\frac{F \ast E \mapsto E \vdash C \quad |E| \geq f}{F \ast E \mapsto E \vdash C}
\]
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:

\[
\begin{align*}
x &= E_{\bar{f}}[x'/x] \ast (F \ast E \mapsto E)[x'/x] \vdash C' \\
F \ast E \mapsto E \vdash C \\
|E| \geq \bar{f}
\end{align*}
\]

(Here, \( \bar{f} \in \mathbb{N} \) and \( E_{\bar{f}} \) is the \( \bar{f} \)th element of \( E \). 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.:

\[
\frac{F \vdash C}{F \ast G \vdash C} \quad \Pi' \subseteq \Pi \quad \text{(Frame)}
\]
Proof rules (contd.)

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

\[
\frac{F \vdash C}{F \ast G \vdash C} \quad \Pi' \subseteq \Pi \quad \text{(Frame)}
\]

• The inductive rules for a predicate \( P \) determine its unfolding rule.
Proof rules (contd.)

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

\[
\frac{F \vdash C}{F \cdot G \vdash C} \quad \Pi' \subseteq \Pi \quad \text{(Frame)}
\]

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

\[
\begin{align*}
x = \text{nil} & \Rightarrow \text{bt}(x) \\
x \neq \text{nil} & \Rightarrow (y, z) \cdot \text{bt}(y) \cdot \text{bt}(z) \Rightarrow \text{bt}(x)
\end{align*}
\]
Proof rules (contd.)

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

\[
F \vdash C \quad \Pi' \subseteq \Pi 
\]

\[
\frac{F \vdash C}{F \ast G \vdash C} \quad (\text{Frame})
\]

- The inductive rules for a predicate \( P \) determine its unfolding rule. E.g., define “binary tree” predicate \( \text{bt} \) by

\[
x = \text{nil} \quad \Rightarrow \quad \text{bt}(x)
\]

\[
x \neq \text{nil} \quad \Rightarrow \quad (y, z) \ast \text{bt}(y) \ast \text{bt}(z) \quad \Rightarrow \quad \text{bt}(x)
\]

This gives the unfolding rule:

\[
F \ast u = \text{nil} \vdash C \quad F \ast u \neq \text{nil} \ast u \leftrightarrow (y, z) \ast \text{bt}(y) \ast \text{bt}(z) \vdash C
\]

\[
\frac{}{F \ast \text{bt}(u) \vdash C}
\]
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:

  ![Diagram of a cyclic pre-proof with back-links](image)

- 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 $\Phi$, and program $C$, and suppose there is a cyclic proof $P$ of $F \vdash C$. Let stack $s$ and heap $h$ satisfy $s, h \models_{\Phi} F$. 
Soundness

Theorem
Fix rule set $\Phi$, 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 $\Phi$, 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 $\Phi$, 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 $x$, find inductive rules $\Phi$ such that

$$P x \vdash C$$

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 $\Phi$ such that

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

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

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

$$F \vdash C \text{ is valid wrt. } \Phi \cup \Phi'.$$
**Problem statement**

- **Initial problem:** Given program $C$ with input variables $x$, find inductive rules $\Phi$ such that

$$Px \vdash C \quad \text{is valid wrt. } \Phi.$$  

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

- **General problem:** Given inductive rules $\Phi$ and subgoal $F \vdash C$, find inductive rules $\Phi'$ 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 $A(P)$ for a combined abduction-and-unfold step.)
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 $P x \Rightarrow P x$:

\[
\frac{P x \vdash 0}{\therefore P x \vdash 0} \ A(P)
\]
**Principia abductica (II)**

When forming back-links, we need to avoid:

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

\[
\begin{array}{c}
\text{\(P x \vdash 0\)}
\end{array}
\]

\[
\frac{P x \vdash 0}{\Rightarrow P x \vdash 0} \quad A(P)
\]

**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;
- abducting trivially inconsistent definitions like $P_x \Rightarrow P_x$:

$$
\begin{array}{c}
P_x \vdash 0 \\
\Rightarrow \\
A(P)
\end{array}
$$

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

\[ P_0(x) \vdash 0 \]
**Worked example: binary tree search**

0: while \((x \neq \text{nil})\) {
1: if(*)
2: \(x := x.l\)
3: else
4: \(x := x.r\)
5: \(\epsilon\)

\(x = \text{nil} \ast P_1(x) \Rightarrow P_0(x)\)
\(x \neq \text{nil} \ast P_2(x) \Rightarrow P_0(x)\)

\(P_0(x) \vdash 0\)
Worked example: binary tree search

0: \textbf{while} \ (x \neq \textit{nil}) \{
1: \textbf{if}(*)
2: x := x.l
3: \textbf{else}
4: x := x.r \}
5: \epsilon

\[
\begin{align*}
& x = \text{nil} * P_1(x) \implies P_0(x) \\
& x \neq \text{nil} * P_2(x) \implies P_0(x)
\end{align*}
\]

\[
\begin{align*}
& x = \text{nil} * P_1(x) \vdash 0 \\
& x \neq \text{nil} * P_2(x) \vdash 0 \\
& P_0(x) \vdash 0
\end{align*}
\]

A(P_0)
Worked example: binary tree search

0: while $(x \neq \text{nil})$
1: if($\ast$)
2: $x := x.l$
3: else
4: $x := x.r$
5: $\varepsilon$

\[
\begin{align*}
&x = \text{nil} \Rightarrow P_1(x) \Rightarrow P_0(x) \\
&x \neq \text{nil} \Rightarrow P_2(x) \Rightarrow P_0(x)
\end{align*}
\]
Worked example: binary tree search

0: \textbf{while} \ (x \neq \text{nil}) \{ \\
1: \quad \textbf{if} (*) \ \\
2: \quad x := x.l \\
3: \quad \textbf{else} \\
4: \quad x := x.r \ \\
5: \quad \epsilon \\
\}

\[
\begin{array}{l}
\text{while} \ (x \neq \text{nil}) \ \\
\quad x = \text{nil} * P_1(x) \quad \Rightarrow \ P_0(x) \\
\quad x \neq \text{nil} * P_2(x) \quad \Rightarrow \ P_0(x)
\end{array}
\]
Worked example: binary tree search

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

x = nil * P_1(x) ⊢ P_0(x)
x ≠ nil * P_2(x) ⇒ P_0(x)

x = nil * P_1(x) ⊢ 0
x ≠ nil * P_2(x) ⊢ 1

while

P_0(x) ⊢ 0
A(P_0)
Worked example: binary tree search

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

\[
\begin{align*}
& x = \text{nil} * P_1(x) \implies P_0(x) \\
& x \neq \text{nil} * P_2(x) \implies P_0(x)
\end{align*}
\]

\[
\begin{align*}
& x = \text{nil} * P_1(x) \downarrow 5 \\
& x = \text{nil} * P_1(x) \downarrow 0 \quad \text{while}
\end{align*}
\]

\[
\begin{align*}
& x \neq \text{nil} * P_2(x) \uparrow 2 \\
& x \neq \text{nil} * P_2(x) \uparrow 4 \quad \text{if}
\end{align*}
\]

\[
\begin{align*}
& x = \text{nil} * P_1(x) \downarrow 5 \quad \text{while}
\end{align*}
\]

\[
\begin{align*}
& x \neq \text{nil} * P_2(x) \uparrow 1 \\
& x \neq \text{nil} * P_2(x) \uparrow 0 \quad \text{while}
\end{align*}
\]

\[
\begin{align*}
& A(P_0) \implies P_0(x) \downarrow 0
\end{align*}
\]
Worked example: binary tree search

0: while (x \neq \text{nil})
   { if(*)
     x := x.l
   else
     x := x.r
   }

1: if(*)
   x = \text{nil} \Rightarrow P_1(x)
   x = \text{nil} \Rightarrow P_0(x)

2: x := x.l
   x \neq \text{nil} \Rightarrow P_2(x)
   x \Rightarrow P_0(x)

3: else
   x \Rightarrow P_2(x)

4: x := x.r

5: \epsilon

x \neq \text{nil} \Rightarrow P_2(x) \Rightarrow 2
x \neq \text{nil} \Rightarrow P_2(x) \Rightarrow 4

\text{if}

x = \text{nil} \Rightarrow P_1(x) \Rightarrow 5
x = \text{nil} \Rightarrow P_0(x) \Rightarrow 1

\text{while}

x = \text{nil} \Rightarrow P_1(x) \Rightarrow 0
x = \text{nil} \Rightarrow P_2(x) \Rightarrow 0

P_0(x) \Rightarrow 0

A(P_0)
Worked example: binary tree search

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

\[
\begin{align*}
  x \neq \text{nil} & \quad \Rightarrow P_0(x) \\
  x \neq \text{nil} & \quad \Rightarrow P_0(x) \\
  x \mapsto (y, z) & \quad \Rightarrow P_2(x)
\end{align*}
\]

\[
\begin{align*}
  x \neq \text{nil} & \quad \Rightarrow P_2(x) \\
  x \neq \text{nil} & \quad \Rightarrow P_2(x) \\
  x \neq \text{nil} & \quad \Rightarrow P_2(x)
\end{align*}
\]

\[
\begin{align*}
  x = \text{nil} & \quad \Rightarrow P_1(x) \\
  x \neq \text{nil} & \quad \Rightarrow P_2(x) \\
  x \neq \text{nil} & \quad \Rightarrow P_2(x)
\end{align*}
\]

\[
\begin{align*}
  x = \text{nil} & \quad \Rightarrow P_1(x) \\
  x \neq \text{nil} & \quad \Rightarrow P_2(x) \\
  x \neq \text{nil} & \quad \Rightarrow P_2(x)
\end{align*}
\]

\[
\begin{align*}
  P_0(x) & \quad \Rightarrow 0 \\
  P_0(x) & \quad \Rightarrow 0
\end{align*}
\]
Worked example: binary tree search

0: while \((x \neq \text{nil})\) {
1: if(\(*\))
2: \(x := x.l\)
3: else
4: \(x := x.r\)  
5: \(\epsilon\)

\[
\begin{align*}
x \neq \text{nil}^* & \\
x \mapsto (y, z) \ast P_3(x, y, z) & \Rightarrow P_2(x)
\end{align*}
\]
**Worked example: binary tree search**

0: while ($x \neq \text{nil}$)

1: if

2: $x := x.l$

3: else

4: $x := x.r$

5: \text{\texttt{epsilon}}

$x' \neq \text{nil*}$

$x' \mapsto (x, z) \ast P_3(x', x, z) \vdash 0$

$x \neq \text{nil*}$

$x \mapsto (y, z) \ast P_3(x, y, z) \vdash 2$

$x \neq \text{nil*} P_2(x) \vdash 2$

$A(P_2)$

$x \neq \text{nil*} P_2(x) \vdash 4$

if

\[ x = \text{nil*} P_1(x) \vdash 5 \]

while

\[ x = \text{nil*} P_1(x) \vdash 0 \]

\[ P_0(x) \vdash 0 \]

\[ x \neq \text{nil*} P_2(x) \vdash 1 \]

while

\[ x \neq \text{nil*} P_2(x) \vdash 0 \]

$A(P_0)$
Worked example: binary tree search

\[\begin{align*}
0: & \text{while } (x \neq \text{nil})\{ \\
1: & \text{if}(\star) \quad x = \text{nil} * P_1(x) \Rightarrow P_0(x) \\
2: & x := x.l \quad x \neq \text{nil} * P_2(x) \Rightarrow P_0(x) \\
3: & \text{else} \quad x \mapsto (y, z) * P_3(x, y, z) \Rightarrow P_2(x) \\
4: & x := x.r \quad P_0(y) * P_4(x, y, z) \Rightarrow P_3(x, y, z) \\
5: & \epsilon
\end{align*}\]
Worked example: binary tree search

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

x' ≠ nil* 
\[ x' \mapsto (x, z) * P_0(x) * P_4(x', x, z) \]
\[ \vdash 0 \]
\[ A(P_3) \]

x' ≠ nil* 
\[ x' \mapsto (x, z) * P_3(x', x, z) \]
\[ \vdash 0 \]
\[ x := x.l \]

x ≠ nil* 
\[ x \mapsto (y, z) * P_3(x, y, z) \]
\[ \vdash 2 \]
\[ A(P_2) \]

x ≠ nil * P_2(x) \vdash 4
if

x = nil * P_1(x) \vdash 5
while

x = nil * P_0(x) \vdash 0

P_0(x) \vdash 0
Worked example: binary tree search

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

\[
P_0(x) \vdash 0
\]

\[
x' \neq \text{nil}*
\]
\[
x' \mapsto (x, z) * P_0(x) * P_4(x', x, z)
\]
\[
A(P_3)
\]

\[
x' \neq \text{nil}*
\]
\[
x' \mapsto (x, z) * P_3(x', x, z)
\]
\[
x := x.l
\]

\[
x \neq \text{nil}*
\]
\[
x \mapsto (y, z) * P_3(x, y, z)
\]
\[
A(P_2)
\]
\[
x \neq \text{nil} * P_2(x) \vdash 2
\]

\[
x = \text{nil} * P_1(x) \vdash 5
\]
\[
x = \text{nil} * P_1(x) \vdash 0
\]

\[
P_0(x) \vdash 0
\]

\[
x \neq \text{nil} * P_2(x) \vdash 4
\]

\[
x \neq \text{nil} * P_2(x) \vdash 1
\]

\[
x \neq \text{nil} * P_2(x) \vdash 0
\]

A(P_0)

x \neq \text{nil} * P_2(x) \vdash 4

if
Worked example: binary tree search

0: while \((x \neq \text{nil})\) {
1: \(\text{if}(*)\)
2: \(x := x.l\)
3: \(\text{else}\)
4: \(x := x.r\)  }
5: \(\varepsilon\)

\(P_0(x) \vdash 0\)

\(x' \neq \text{nil}^*\)
\(x' \mapsto (x, z) * P_0(x) * P_4(x', x, z) \vdash 0\)
\(A(P_3)\)

\(x' \neq \text{nil}^*\)
\(x' \mapsto (x, z) * P_3(x', x, z) \vdash 0\)
\(x := x.l\)
\(x \neq \text{nil}^*\)
\(x \mapsto (y, z) * P_3(x, y, z) \vdash 2\)
\(A(P_2)\)

\(x \neq \text{nil} * P_2(x) \vdash 4\)

\(x = \text{nil} * P_1(x) \vdash 5\)
\(x = \text{nil} * P_1(x) \vdash 0\)
\(P_0(x) \vdash 0\)

\(\text{while } x \neq \text{nil} * P_2(x) \vdash 1\)
\(\text{while } x \neq \text{nil} * P_2(x) \vdash 0\)

\(A(P_0)\)
Worked example: binary tree search

\begin{align*}
0: & \text{while } (x \neq \text{nil}) \\
1: & \text{if } (\star) \quad x = \text{nil} * P_1(x) \quad \Rightarrow P_0(x) \\
2: & x := x.l \quad x \neq \text{nil} * P_2(x) \quad \Rightarrow P_0(x) \\
3: & \text{else} \quad x \mapsto (y, z) * P_3(x, y, z) \quad \Rightarrow P_2(x) \\
4: & x := x.r \quad P_0(y) * P_4(x, y, z) \quad \Rightarrow P_3(x, y, z) \\
5: & \epsilon
\end{align*}

\begin{diagram}
\begin{align*}
\underbrace{x' \neq \text{nil}^*}_{\text{(Frame)}} & \quad \vdash 0 \\
\underbrace{x' \mapsto (x, z) * P_0(x) * P_4(x', x, z)}_{A(P_3)} & \quad \vdash 0 \\
\underbrace{x' \neq \text{nil}^*}_{\quad x := x.l} & \quad \vdash 0 \\
\underbrace{x \neq \text{nil}^*}_{\quad x := x.l} & \quad \vdash 2 \\
\underbrace{x \mapsto (y, z) * P_3(x, y, z)}_{A(P_2)} & \quad \vdash 4 \\
\underbrace{x = \text{nil} * P_1(x) \vdash 5}_{\text{while}} & \quad \vdash 0 \\
\underbrace{x = \text{nil} * P_1(x) \vdash 0}_{\text{while}} & \quad \vdash 0 \\
\underbrace{x \neq \text{nil} * P_2(x) \vdash 1}_{\text{while}} & \quad \vdash 0 \\
& \quad \vdash 0 \\
& \quad \vdash 0 \\
& \quad \vdash 0 \\
\end{align*}
\end{diagram}
Worked example: binary tree search

\[
\begin{align*}
0: & \text{while } (x \neq \text{nil}) \{ \\
1: & \quad \text{if}(\ast) \quad x = \text{nil} \ast P_1(x) \quad \Rightarrow \quad P_0(x) \\
2: & \quad x := x.l \quad x \neq \text{nil} \ast P_2(x) \quad \Rightarrow \quad P_0(x) \\
3: & \quad \text{else} \quad x \mapsto (y, z) \ast P_3(x, y, z) \quad \Rightarrow \quad P_2(x) \\
4: & \quad x := x.r \quad P_0(y) \ast P_4(x, y, z) \quad \Rightarrow \quad P_3(x, y, z) \\
5: & \quad x \mapsto \epsilon \quad x = \text{nil} \ast P_1(x) \quad \Rightarrow \quad P_0(x)
\end{align*}
\]

\[
\begin{align*}
\vdash & P_0(x) \quad (\text{Frame}) \\
\text{if} \quad x' \neq \text{nil} \ast \\
\quad x' \mapsto (x, z) \ast P_0(x) \ast P_4(x', x, z) & \quad \vdash \quad 0 \quad A(P_3) \\
\text{if} \quad x' \neq \text{nil} \ast \\
\quad x' \mapsto (x, z) \ast P_3(x', x, z) & \quad \vdash \quad 0 \quad x := x.l \quad A(P_3) \\
\text{if} \quad x \neq \text{nil} \ast \\
\quad x \mapsto (y, z) \ast P_3(x, y, z) & \quad \vdash \quad 2 \quad x := x.r \quad A(P_2) \\
\text{if} \quad x \neq \text{nil} \ast P_2(x) \vdash \quad 2 \\
\text{if} \quad x = \text{nil} \ast P_1(x) \vdash \quad 5 \\
\text{if} \quad x = \text{nil} \ast P_1(x) \vdash \quad 0 \\
\text{if} \quad x \neq \text{nil} \ast P_2(x) \vdash \quad 1 \\
\text{if} \quad x \neq \text{nil} \ast P_2(x) \vdash \quad 0 \\
\text{if} \quad x \neq \text{nil} \ast P_2(x) \vdash \quad 4 \\
\text{if} \quad x \neq \text{nil} \ast P_2(x) \vdash \quad 2 \\
\text{if} \quad x = \text{nil} \ast P_1(x) \vdash \quad 0 \\
\text{if} \quad x = \text{nil} \ast P_1(x) \vdash \quad 0 \\
\text{if} \quad x \neq \text{nil} \ast P_2(x) \vdash \quad 0 \\
\text{if} \quad x \neq \text{nil} \ast P_2(x) \vdash \quad 0
\end{align*}
\]
Worked example: binary tree search

0: while \((x \neq \text{nil})\) {
  1: \(i(x)\) if \((x \neq \text{nil})\) \(P_1(x) \Rightarrow P_0(x)\)
  2: \(x := x.l\) \(x \neq \text{nil} \Rightarrow P_0(x)\)
  3: else \(P_0(y) * P_4(x, y, z) \Rightarrow P_3(x, y, z)\)
  4: \(x := x.r\) \(P_0(y) * P_4(x, y, z) \Rightarrow P_3(x, y, z)\)
  5: \(x = \text{nil}\)
}

\(P_0(x) \Downarrow 0\) (Frame)

\(x' \neq \text{nil}\)

\(x' \mapsto (x, z) * P_0(x) * P_4(x', x, z) \Downarrow 0\) \(A(P_3)\)

\(x' \neq \text{nil}\)

\(x' \mapsto (x, z) * P_3(x', x, z) \Downarrow 0\)

\(x := x.l\)

\(x \neq \text{nil}\)

\(x \mapsto (y, z) * P_3(x, y, z) \Downarrow 2\) \(A(P_2)\)

\(x = \text{nil} \Rightarrow P_1(x) \Downarrow 5\)

while \(x = \text{nil} * P_1(x) \Downarrow 0\)

\(x = \text{nil} \Rightarrow P_1(x) \Downarrow 5\)

while \(x \neq \text{nil} * P_2(x) \Downarrow 1\)

\(x \neq \text{nil} * P_2(x) \Downarrow 4\) \(A(P_0)\)

\(x \neq \text{nil} \Rightarrow P_2(x) \Downarrow 4\) (P2)

\(x \mapsto (y, z) * P_3(x, y, z) \Downarrow 4\) \(A(P_3)\)

\(x := x.r\)

\(x' \neq \text{nil}\)

\(x' \mapsto (y, x) * P_3(x', y, x) \Downarrow 0\)

\(x = \text{nil} \Rightarrow P_1(x) \Downarrow 5\) while \(A(P_0)\)

\(x = \text{nil} * P_1(x) \Downarrow 0\) while \(A(P_0)\)

\(x = \text{nil} \Rightarrow P_1(x) \Downarrow 5\) while \(A(P_0)\)

\(x \neq \text{nil} * P_2(x) \Downarrow 1\) while \(A(P_0)\)

\(x \neq \text{nil} * P_2(x) \Downarrow 4\) \(A(P_0)\)

\(x = \text{nil} \Rightarrow P_2(x) \Downarrow 4\) (P2)

\(x \mapsto (y, z) * P_3(x, y, z) \Downarrow 4\) \(A(P_3)\)

\(x := x.r\)

\(x \neq \text{nil} \Rightarrow P_2(x) \Downarrow 4\) (P2)

\(x \mapsto (y, x) * P_3(x', y, x) \Downarrow 0\)

\(x' \neq \text{nil}\)

\(x' \mapsto (x, z) * P_0(x) * P_4(x', x, z) \Downarrow 0\) \(A(P_3)\)

\(x' \neq \text{nil}\)

\(x' \mapsto (x, z) * P_3(x', x, z) \Downarrow 0\)

\(x := x.l\)

\(x \neq \text{nil}\)

\(x \mapsto (y, z) * P_3(x, y, z) \Downarrow 2\) \(A(P_2)\)

\(x = \text{nil} \Rightarrow P_1(x) \Downarrow 5\)

while \(x = \text{nil} * P_1(x) \Downarrow 0\)

\(x = \text{nil} \Rightarrow P_1(x) \Downarrow 5\)

while \(x \neq \text{nil} * P_2(x) \Downarrow 1\)

\(x \neq \text{nil} * P_2(x) \Downarrow 4\) \(A(P_0)\)

\(x = \text{nil} \Rightarrow P_2(x) \Downarrow 4\) (P2)

\(x \mapsto (y, z) * P_3(x, y, z) \Downarrow 4\) \(A(P_3)\)

\(x := x.r\)

\(x \neq \text{nil} \Rightarrow P_2(x) \Downarrow 4\) (P2)

\(x \mapsto (y, x) * P_3(x', y, x) \Downarrow 0\)

\(x' \neq \text{nil}\)

\(x' \mapsto (x, z) * P_0(x) * P_4(x', x, z) \Downarrow 0\) \(A(P_3)\)

\(x' \neq \text{nil}\)

\(x' \mapsto (x, z) * P_3(x', x, z) \Downarrow 0\)

\(x := x.l\)

\(x \neq \text{nil}\)

\(x \mapsto (y, z) * P_3(x, y, z) \Downarrow 2\) \(A(P_2)\)

\(x = \text{nil} \Rightarrow P_1(x) \Downarrow 5\)

while \(x = \text{nil} * P_1(x) \Downarrow 0\)

\(x = \text{nil} \Rightarrow P_1(x) \Downarrow 5\)

while \(x \neq \text{nil} * P_2(x) \Downarrow 1\)

\(x \neq \text{nil} * P_2(x) \Downarrow 4\) \(A(P_0)\)

\(x = \text{nil} \Rightarrow P_2(x) \Downarrow 4\) (P2)

\(x \mapsto (y, z) * P_3(x, y, z) \Downarrow 4\) \(A(P_3)\)

\(x := x.r\)

\(x \neq \text{nil} \Rightarrow P_2(x) \Downarrow 4\) (P2)

\(x \mapsto (y, x) * P_3(x', y, x) \Downarrow 0\)
Worked example: binary tree search

0: while \(x \neq \text{nil}\)\
1: \(\text{if}(\ast)\) \\
2: \(x := x.l\) \\
3: \(\text{else}\) \\
4: \(x := x.r\) \\
5: \(\varepsilon\)

\[P_0(x) \vdash 0\] (Frame)

\[x' \neq \text{nil}\star\]
\[x' \mapsto (x, z) * P_0(x) * P_4(x', x, z) \vdash 0\] (Frame)

\[x' \neq \text{nil}\star\]
\[x' \mapsto (y, x) * P_0(y) * P_4(x', y, x) \vdash 0\] (P3)

\[x' \neq \text{nil}\star\]
\[x' \mapsto (x, z) * P_3(x', x, z) \vdash 0\] (Frame)

\[x \neq \text{nil}\star\]
\[x := x.l\]

\[x \neq \text{nil}\star\]
\[x := x.r\]

\[x := x.l\]

\[x \neq \text{nil}\star\]
\[x := x.r\]

\[x = \text{nil} * P_1(x) \vdash 0\] (Frame)

\[x = \text{nil} * P_1(x) \vdash 5\] while

\[x = \text{nil} * P_1(x) \vdash 0\] (Frame)

\[x \neq \text{nil} * P_2(x) \vdash 1\] while

\[x \neq \text{nil} * P_2(x) \vdash 0\] (Frame)

\[x \neq \text{nil} * P_2(x) \vdash 2\] (Frame)

\[x \neq \text{nil} * P_2(x) \vdash 4\] (Frame)

\[x \neq \text{nil} * P_2(x) \vdash 2\] (Frame)

\[P_0(x) \vdash 0\]
Worked example: binary tree search

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

\[\begin{align*}
P_0(x) &\vdash 0 \\
x' \neq \text{nil}^* \\
x' \mapsto (x, z) * P_0(x) * P_4(x', x, z) &\vdash 0 \\
A(P_3) \\
x' \neq \text{nil}^* \\
x' \mapsto (x, z) * P_3(x', x, z) &\vdash 0 \\
x := x.l
A(P_2)
\end{align*}\]

\[\begin{align*}
x = \text{nil} * P_1(x) &\vdash 5 \\
x = \text{nil} * P_1(x) &\vdash 0 \\
\text{while} &\rightarrow P_0(x) \vdash 0 \\
\end{align*}\]

\[\begin{align*}
x = \text{nil} * P_2(x) &\vdash 1 \\
x = \text{nil} * P_2(x) &\vdash 0 \\
\text{while} &\rightarrow P_0(x) \vdash 0 \\
\end{align*}\]

\[\begin{align*}
x = \text{nil} * P_2(x) &\vdash 4 \\
x = \text{nil} * P_2(x) &\vdash 1 \\
\text{while} &\rightarrow P_0(x) \vdash 0 \\
\end{align*}\]
Worked example: binary tree search

0: while \( (x \neq \text{nil}) \) {
    1: if(*)
    2: \( x := x.l \)
    3: else
    4: \( x := x.r \)  
  }
  5: \( x = \text{nil} \)

\( x = \text{nil} * P_1(x) \) \( \Rightarrow P_0(x) \)
\( x \neq \text{nil} * P_2(x) \) \( \Rightarrow P_0(x) \)
\( x \mapsto (y, z) * P_3(x, y, z) \) \( \Rightarrow P_2(x) \)
\( P_0(y) * P_4(x, y, z) \) \( \Rightarrow P_3(x, y, z) \)
\( P_0(z) * P_5(x, y, z) \) \( \Rightarrow P_4(x, y, z) \)

---

\( x' \neq \text{nil} * \)
\( x' \mapsto (y, x) * P_0(y) * P_0(x) * P_5(x', y, x) \) \( \vdash 0 \)
\( (\text{Frame}) \)
\( x' \mapsto (y, x) * P_0(y) * P_4(x', y, x) \) \( \vdash 0 \)
\( A(P_4) \)
\( x' \mapsto (y, x) * P_3(x', y, x) \) \( \vdash 0 \)
\( x' \neq \text{nil} * \)
\( P_0(x) \) \( \vdash 0 \)
\( x := x.l \)
\( x' \neq \text{nil} * \)
\( x' \mapsto (y, z) * P_3(x', x, z) \) \( \vdash 0 \)
\( A(P_3) \)
\( x' \neq \text{nil} * \)
\( x' \mapsto (y, z) * P_0(x) * P_4(x', x, z) \) \( \vdash 0 \)
\( A(P_2) \)
\( x' \neq \text{nil} * \)
\( x' \mapsto (y, z) * P_3(x', x, z) \) \( \vdash 0 \)
\( x := x.r \)
\( x' \neq \text{nil} * \)
\( x := x.l \)
\( x' \neq \text{nil} * \)
\( x' \mapsto (y, z) * P_3(x', y, x) \) \( \vdash 0 \)
\( A(P_2) \)
\( x' \neq \text{nil} * \)
\( x' \mapsto (y, z) * P_3(x', y, x) \) \( \vdash 0 \)
\( (P_2) \)
\( x' \neq \text{nil} * \)
\( x := x.r \)
\( x' \neq \text{nil} * \)
\( x := x.l \)
\( x' \neq \text{nil} * \)
\( x' \mapsto (y, z) * P_3(x', y, x) \) \( \vdash 0 \)
\( A(P_2) \)
\( x' \neq \text{nil} * \)
\( x := x.r \)
\( x' \neq \text{nil} * \)
\( x := x.l \)
\( x' \neq \text{nil} * \)
\( x' \mapsto (y, z) * P_3(x', x, z) \) \( \vdash 0 \)
\( A(P_2) \)
\( x' \neq \text{nil} * \)
\( x := x.r \)
\( x' \neq \text{nil} * \)
\( x := x.l \)
\( x' \neq \text{nil} * \)
\( x := x.r \)
\( x' \neq \text{nil} * \)
\( x := x.l \)
\( x' \neq \text{nil} * \)
\( x := x.r \)
\( x' \neq \text{nil} * \)
\( x := x.l \)
\( x' \neq \text{nil} * \)
\( x := x.r \)
\( x' \neq \text{nil} * \)
\( x := x.l \)
\( x' \neq \text{nil} * \)
\( x := x.r \)

---

\( x = \text{nil} * P_1(x) \) \( \vdash 0 \)
\( x = \text{nil} * P_1(x) \) \( \vdash 5 \)
\( x = \text{nil} * P_1(x) \) \( \vdash 0 \)
\( \text{if} \)
\( x = \text{nil} * P_2(x) \) \( \vdash 2 \)
\( x = \text{nil} * P_2(x) \) \( \vdash 4 \)
\( x = \text{nil} * P_2(x) \) \( \vdash 1 \)
\( \text{while} \)
\( x = \text{nil} * P_2(x) \) \( \vdash 0 \)

\( P_0(x) \) \( \vdash 0 \)
\( P_0(x) \) \( \vdash 0 \)
\( P_0(x) \) \( \vdash 0 \)
\( \text{while} \)

20/27
Worked example: binary tree search

0: \textbf{while } (x \neq \text{nil}) \{ \hspace{1cm} x = \text{nil} \times P_1(x) \Rightarrow P_0(x) \\
1: \textbf{if}(\ast) \hspace{1cm} x \neq \text{nil} \times P_2(x) \Rightarrow P_0(x) \\
2: x := x.l \hspace{1cm} x \mapsto (y, z) \times P_3(x, y, z) \Rightarrow P_3(x, y, z) \\
3: \textbf{else} \hspace{1cm} P_0(y) \times P_4(x, y, z) \Rightarrow P_3(x, y, z) \\
4: x := x.r \} \hspace{1cm} P_0(z) \times P_5(x, y, z) \Rightarrow P_4(x, y, z) \\
5: \epsilon \\

\begin{align*}
\quad P_0(x) \vdash 0 \\
\quad x' \neq \text{nil*} \\
\quad x' \mapsto (x, z) \times P_0(x) \times P_4(x', x, z) \vdash 0
\end{align*}

\begin{align*}
\quad A(P_4) \\
\quad x' \neq \text{nil*} \\
\quad x' \mapsto (y, x) \times P_0(y) \times P_0(x) \times P_5(x', y, x) \vdash 0
\end{align*}

\begin{align*}
\quad A(P_3) \\
\quad x' \neq \text{nil*} \\
\quad x' \mapsto (y, x) \times P_0(y) \times P_4(x', y, x) \vdash 0
\end{align*}

\begin{align*}
\quad (P_3) \\
\quad x' \neq \text{nil*} \\
\quad x' \mapsto (y, x) \times P_3(x', y, x) \vdash 0
\end{align*}

\begin{align*}
\quad x := x.l \\
\quad x \neq \text{nil*} \\
\quad x \mapsto (y, z) \times P_3(x, y, z) \vdash 2
\end{align*}

\begin{align*}
\quad A(P_2) \\
\quad x \neq \text{nil} \times P_2(x) \vdash 2
\end{align*}

\begin{align*}
\quad x = \text{nil} \times P_1(x) \vdash 5 \\
\quad x = \text{nil} \times P_1(x) \vdash 0 \\
\quad x = \text{nil} \times P_1(x) \vdash 0 \\
\quad x \neq \text{nil} \times P_2(x) \vdash 1 \\
\quad x \neq \text{nil} \times P_2(x) \vdash 4
\end{align*}

\begin{align*}
\quad \text{if} \quad x \neq \text{nil} \times P_2(x) \vdash 4
\end{align*}

\begin{align*}
\quad 0 \quad \text{while} \quad x \neq \text{nil} \times P_2(x) \vdash 1 \\
\quad \text{while} \quad x = \text{nil} \times P_1(x) \vdash 5
\end{align*}

\begin{align*}
\quad A(P_0) \\
\quad x = \text{nil} \times P_1(x) \vdash 0
\end{align*}
Worked example: binary tree search

0: while \((x \neq \text{nil})\) 
1: if(*)
2: \(x := x.l\)
3: else
4: \(x := x.r\) 
5: \(\epsilon\)

\[
\begin{align*}
0: \text{while } (x \neq \text{nil}) \{ & x = \text{nil} \ast P_1(x) \Rightarrow P_0(x) \\
1: \text{if}(*): & x \neq \text{nil} \ast P_2(x) \Rightarrow P_0(x) \\
2: x := x.l: & x \mapsto (y, z) \ast P_3(x, y, z) \Rightarrow P_3(x, y, z) \\
3: \text{else}: & P_0(y) \ast P_4(x, y, z) \Rightarrow P_4(x, y, z) \\
4: x := x.r: & \epsilon
\end{align*}
\]
Simplifying inductive rule sets

\[
x = \text{nil} : P_1(x) \Rightarrow P_0(x)
\]

\[
x \neq \text{nil} : P_2(x) \Rightarrow P_0(x)
\]

\[
x \mapsto (y, z) \ast P_3(x, y, z) \Rightarrow P_2(x)
\]

\[
P_0(y) \ast P_4(x, y, z) \Rightarrow P_3(x, y, z)
\]

\[
P_0(z) \ast P_5(x, y, z) \Rightarrow P_4(x, y, z)
\]
Simplifying inductive rule sets

- instantiate undefined predicates to \texttt{emp};

\[
\begin{align*}
    x = \text{nil} : P_1(x) & \Rightarrow P_0(x) \\
    x \neq \text{nil} : P_2(x) & \Rightarrow P_0(x) \\
    x \mapsto (y, z) : P_3(x, y, z) & \Rightarrow P_2(x) \\
    P_0(y) \ast P_4(x, y, z) & \Rightarrow P_3(x, y, z) \\
    P_0(z) \ast P_5(x, y, z) & \Rightarrow P_4(x, y, z)
\end{align*}
\]
Simplifying inductive rule sets

- instantiate undefined predicates to \texttt{emp};

\[
\begin{align*}
    x = \text{nil} : & P_1(x) \Rightarrow P_0(x) \\
    x \neq \text{nil} : & P_2(x) \Rightarrow P_0(x) \\
    x \mapsto (y, z) \ast & P_3(x, y, z) \Rightarrow P_2(x) \\
    P_0(y) \ast & P_4(x, y, z) \Rightarrow P_3(x, y, z) \\
    P_0(z) \ast & P_5(x, y, z) \Rightarrow P_4(x, y, z)
\end{align*}
\]

\[
\begin{align*}
    x = \text{nil} : \texttt{emp} \quad & \Rightarrow P_0(x) \\
    x \neq \text{nil} : & P_2(x) \Rightarrow P_0(x) \\
    x \mapsto (y, z) \ast & P_3(x, y, z) \Rightarrow P_2(x) \\
    P_0(y) \ast & P_4(x, y, z) \Rightarrow P_3(x, y, z) \\
    P_0(z) \Rightarrow & P_4(x, y, z)
\end{align*}
\]
Simplifying inductive rule sets

- instantiate undefined predicates to \( \text{emp} \);
- eliminate redundant parameters;

\[
\begin{align*}
x = \text{nil} : P_1(x) & \implies P_0(x) & \quad x = \text{nil} : \text{emp} & \implies P_0(x) \\
x \neq \text{nil} : P_2(x) & \implies P_0(x) & \quad x \neq \text{nil} : P_2(x) & \implies P_0(x) \\
x \mapsto (y, z) : P_3(x, y, z) & \implies P_2(x) & \quad x \mapsto (y, z) : P_3(x, y, z) & \implies P_2(x) \\
P_0(y) : P_4(x, y, z) & \implies P_3(x, y, z) & \quad P_0(y) : P_4(x, y, z) & \implies P_3(x, y, z) \\
P_0(z) : P_5(x, y, z) & \implies P_4(x, y, z) & \quad P_0(z) & \implies P_4(x, y, z)
\end{align*}
\]
Simplifying inductive rule sets

- instantiate undefined predicates to \( \text{emp} \);
- eliminate redundant parameters;

\[
\begin{align*}
\begin{array}{ll}
x = \text{nil} : P_1(x) & \Rightarrow P_0(x) \\
x \neq \text{nil} : P_2(x) & \Rightarrow P_0(x) \\
x \mapsto (y, z) \ast P_3(x, y, z) & \Rightarrow P_2(x) \\
P_0(y) \ast P_4(x, y, z) & \Rightarrow P_3(x, y, z) \\
P_0(z) \ast P_5(x, y, z) & \Rightarrow P_4(x, y, z)
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{ll}
x = \text{nil} : \text{emp} & \Rightarrow P_0(x) \\
x \neq \text{nil} : P_2(x) & \Rightarrow P_0(x) \\
x \mapsto (y, z) \ast P_3(x, y, z) & \Rightarrow P_2(x) \\
P_0(y) \ast P_4(x, y, z) & \Rightarrow P_3(x, y, z) \\
P_0(z) & \Rightarrow P_4(x, y, z)
\end{array}
\end{align*}
\]

\[
\begin{align*}
\downarrow
\begin{align*}
x = \text{nil} : \text{emp} & \Rightarrow P_0(x) \\
x \neq \text{nil} : P_2(x) & \Rightarrow P_0(x) \\
x \mapsto (y, z) \ast P_3(x, y) & \Rightarrow P_2(x) \\
P_0(y) \ast P_4(z) & \Rightarrow P_3(x, y) \\
P_0(z) & \Rightarrow P_4(z)
\end{align*}
\end{align*}
\]
Simplifying inductive rule sets

- instantiate undefined predicates to \texttt{emp};
- eliminate redundant parameters;
- inline single-clause predicates.

\[
\begin{align*}
  x = \text{nil} : P_1(x) & \Rightarrow P_0(x) \\
  x \neq \text{nil} : P_2(x) & \Rightarrow P_0(x) \\
  x \mapsto (y, z) \ast P_3(x, y, z) & \Rightarrow P_2(x) \\
  P_0(y) \ast P_4(x, y, z) & \Rightarrow P_3(x, y, z) \\
  P_0(z) \ast P_5(x, y, z) & \Rightarrow P_4(x, y, z)
\end{align*}
\]
Simplifying inductive rule sets

- instantiate undefined predicates to \texttt{emp};
- eliminate redundant parameters;
- inline single-clause predicates.

\[
\begin{align*}
  & x = \text{nil} : P_1(x) \Rightarrow P_0(x) \\
  & x \neq \text{nil} : P_2(x) \Rightarrow P_0(x) \\
  & x \mapsto (y, z) \ast P_3(x, y, z) \Rightarrow P_2(x) \\
  & P_0(y) \ast P_4(x, y, z) \Rightarrow P_3(x, y, z) \\
  & P_0(z) \ast P_5(x, y, z) \Rightarrow P_4(x, y, z)
\end{align*}
\]

\[
\begin{align*}
  & x = \text{nil} : \texttt{emp} \Rightarrow P_0(x) \\
  & x \neq \text{nil} : P_2(x) \Rightarrow P_0(x) \\
  & x \mapsto (y, z) \ast P_3(x, y, z) \Rightarrow P_2(x) \\
  & P_0(y) \ast P_4(x, y, z) \Rightarrow P_3(x, y, z) \\
  & P_0(z) \Rightarrow P_4(x, y, z)
\end{align*}
\]

\[
\begin{align*}
  & x = \text{nil} : \texttt{emp} \Rightarrow P_0(x) \\
  & x \neq \text{nil} : x \mapsto (y, z) \ast P_0(y) \ast P_0(z) \Rightarrow P_0(x)
\end{align*}
\]
Simplifying inductive rule sets

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

\[
\begin{align*}
    x = \text{nil} &: P_1(x) &\Rightarrow& P_0(x) \\
    x \neq \text{nil} &: P_2(x) &\Rightarrow& P_0(x) \\
    x \mapsto (y, z) \ast P_3(x, y, z) &\Rightarrow& P_2(x) \\
    P_0(y) \ast P_4(x, y, z) &\Rightarrow& P_3(x, y, z) \\
    P_0(z) \ast P_5(x, y, z) &\Rightarrow& P_4(x, y, z)
\end{align*}
\]

(nil-terminated binary tree)

\[
\begin{align*}
    x = \text{nil} &: \text{emp} &\Rightarrow& P_0(x) \\
    x \neq \text{nil} &: P_2(x) &\Rightarrow& P_0(x) \\
    x \mapsto (y, z) \ast P_3(x, y) &\Rightarrow& P_2(x) \\
    P_0(y) \ast P_4(z) &\Rightarrow& P_3(x, y) \\
    P_0(z) &\Rightarrow& P_4(z)
\end{align*}
\]
Simplifying inductive rule sets

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

\[
\begin{align*}
    x = \text{nil} : P_1(x) & \Rightarrow P_0(x) \\
    x \neq \text{nil} : P_2(x) & \Rightarrow P_0(x) \\
    x \mapsto (y, z) * P_3(x, y, z) & \Rightarrow P_2(x) \\
    P_0(y) * P_4(x, y, z) & \Rightarrow P_3(x, y, z) \\
    P_0(z) * P_5(x, y, z) & \Rightarrow P_4(x, y, z)
\end{align*}
\]

(nil-terminated binary tree)

\[
\begin{align*}
    x = \text{nil} : \text{emp} & \Rightarrow P_0(x) \\
    x \neq \text{nil} : x \mapsto (y, z) * P_0(y) * P_0(z) & \Rightarrow P_0(x)
\end{align*}
\]
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

Evaluating solution quality

- **Backtracking search** can yield different solutions.

- We can **decide** whether a predicate is satisfiable.

  A decision procedure for satisfiability of inductive predicates in separation logic.  
  Submitted.

- Comparing predicates via entailment (\(\models\)) is **not practical**.
Evaluating solution quality

- **Backtracking search** can yield different solutions.

- **We can decide** whether a predicate is satisfiable
  

- 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
  
  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

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Time</th>
<th>Depth</th>
<th>Quality</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>List traverse</td>
<td>3</td>
<td>20</td>
<td>3</td>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>List insert</td>
<td>14</td>
<td>8</td>
<td>7</td>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>List copy</td>
<td>12</td>
<td>0</td>
<td>8</td>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>List append</td>
<td>10</td>
<td>12</td>
<td>5</td>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>Delete last from list</td>
<td>16</td>
<td>12</td>
<td>9</td>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>Filter list</td>
<td>21</td>
<td>48</td>
<td>11</td>
<td>C</td>
<td>✓</td>
</tr>
<tr>
<td>Dispose list</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>Reverse list</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>Cyclic list traverse</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>Binary tree search</td>
<td>7</td>
<td>8</td>
<td>4</td>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>Binary tree insert</td>
<td>18</td>
<td>4</td>
<td>7</td>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>List of lists traverse</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>Traverse even-length list</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>Traverse odd-length list</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>Ternary tree search</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>Conditional diverge</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>B</td>
<td>×</td>
</tr>
<tr>
<td>Traverse list of trees</td>
<td>11</td>
<td>12</td>
<td>6</td>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>Traverse tree of lists</td>
<td>17</td>
<td>68</td>
<td>7</td>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>Traverse list twice</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Problem: initial variable assignment

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

$$
\begin{align*}
    y &= x \cdot P x \vdash 1 \\
    P x \vdash 0
\end{align*}
$$

$y := x$
Problem: initial variable assignment

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

$$y = x \times P_x \vdash 1$$

$$P_x \vdash 0$$

$$\therefore y := x$$

- 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

\[
\frac{y = x \cdot P_x \vdash 1}{P_x \vdash 0} \quad y := x
\]

- 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

\[
y = x * P x \vdash 1
\]

\[
P x \vdash 0
\]

\[
\frac{y := x}{y := x}
\]

- 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

$$
y = x \ast Px \vdash 1
\quad \frac{y := x}{Px \vdash 0}
$$

- 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:

\[
\frac{\Pi : F \ast list(x) \vdash i}{\Pi : F \ast x \mapsto y \vdash i}
\]

\[x \mapsto y \vdash list(x) \quad \text{(Cut)}\]
Problem: abstraction

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

- Here it shows up in the need for lemmas:

\[
\Pi : F \times \text{list}(x) \vdash i \\
\Pi : F \times y \vdash x \mapsto y 
\]

\[x \mapsto y \vdash \text{list}(x) \quad \text{(Cut)}\]

- 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:

\[
\Pi : F \ast \text{list}(x) \vdash i \\
\Pi : F \ast x \mapsto y \vdash i \\
\]

(x \mapsto y \vdash \text{list}(x) \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:

\[
\begin{align*}
\Pi : F \ast \text{list}(x) & \vdash i \\
\Pi : F \ast x \mapsto y & \vdash i \\
x \mapsto y & \vdash \text{list}(x) \quad \text{(Cut)}
\end{align*}
\]

- 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.