Program Verification Using Cyclic Proof

Reuben N. S. Rowe
University College London
Programming Principles, Logic and Verification Research Group (PPLV)

Computer Laboratory Programming Research Group Seminar
Thursday 19th May 2016
• We are all familiar with proofs as finite trees
• We are all familiar with proofs as finite trees
• But what if we allow proofs to be cyclic graphs instead?
• We are all familiar with proofs as finite trees
• But what if we allow proofs to be **cyclic graphs** instead?
• Cyclic proofs must satisfy a **global soundness** property
Our research programme has two broad aims:

- Develop cyclic proof (meta) theory in a verification setting
- Implement the techniques for automatic verification

Why cyclic proof?
- It subsumes standard induction
- It can help discover inductive hypotheses
- Termination arguments can often be extracted from cyclic proofs
• Our research programme has two broad aims:
  • Develop cyclic proof (meta) theory in a verification setting
Our research programme has two broad aims:

- Develop cyclic proof (meta) theory in a verification setting
- Implement the techniques for automatic verification
• Our research programme has two broad aims:
  • Develop cyclic proof (meta) theory in a verification setting
  • Implement the techniques for automatic verification

• Why cyclic proof?
Our research programme has two broad aims:
- Develop cyclic proof (meta) theory in a verification setting
- Implement the techniques for automatic verification

Why cyclic proof?
- It subsumes standard induction
Our research programme has two broad aims:
- Develop cyclic proof (meta) theory in a verification setting
- Implement the techniques for automatic verification

Why cyclic proof?
- It subsumes standard induction
- It can help discover inductive hypotheses
Our research programme has two broad aims:

- Develop cyclic proof (meta) theory in a verification setting
- Implement the techniques for automatic verification

Why cyclic proof?

- It subsumes standard induction
- It can help discover inductive hypotheses
- Termination arguments can often be extracted from cyclic proofs
Example: First Order Logic

- Assume signature with zero, successor, and equality
- Allow inductive predicate definitions, e.g.

\[
\begin{array}{c}
\text{N 0} & \text{N sx} & \text{E 0} & \text{O sx} & \text{E sx} \\
\text{N x} & \text{E x} & \text{O x}
\end{array}
\]
Example: First Order Logic

- Assume signature with zero, successor, and equality
- Allow inductive predicate definitions, e.g.

\[
\begin{array}{cccc}
N x & E x & O x \\
N 0 & N sx & E 0 & O sx \\
N st & & E sx \\
\end{array}
\]

- These induce unfolding rules for the sequent calculus, e.g.

\[
\frac{\Gamma, t \vdash \Delta, N t}{\Gamma, \Delta, N st} \quad (NR_2) \quad \frac{\Gamma, t = 0 \vdash \Delta}{\Gamma, N t \vdash \Delta} \quad \frac{\Gamma, t = sx, N x \vdash \Delta}{\Gamma, N t \vdash \Delta} \quad \text{(Case N)}
\]

where \(x\) is fresh
A Cyclic Proof of $N x \vdash E x, O x$

Suppose $N x \vdash E x, O x$ is not valid:

$\begin{align*}
\text{m}_1 & > & \text{m}_2 = & \text{m}_3 = & \text{m}_4 = & \text{m}_5 = & \text{m}_6 > & \text{m}_7 :& : : \\
\end{align*}$
A Cyclic Proof of \( N x \vdash E x, O x \)

\[
\begin{align*}
  x = 0 & \vdash E x, O x \\
  x = sy, N y & \vdash E x, O x \\
  N x & \vdash E x, O x
\end{align*}
\]

(Case \( N \))
A Cyclic Proof of $N x \vdash E x, O x$

\[
\begin{align*}
\vdash E 0, O 0 \\
x = 0 \vdash E x, O x & \quad \vdash x = sy, N y \vdash E x, O x \\
\hline
N x \vdash E x, O x
\end{align*}
\]
A Cyclic Proof of $N x \vdash E x, O x$

\[
\begin{align*}
\vdash E 0, O 0 \\
\text{(ER}_1\text{)} \\
\vdash E 0, O 0 \\
\text{(=L)} \\
x = 0 \vdash E x, O x \\
x = sy, N y \vdash E x, O x \\
\hline
N x \vdash E x, O x \\
\text{(Case N)}
\end{align*}
\]
A Cyclic Proof of \( N x \vdash E x, O x \)

\[
\begin{align*}
&\vdash E 0, O 0 \quad (ER_1) \\
&x = 0 \vdash E x, O x \quad (=L) \\
&\vdash E x, O x \quad (Case \ N)
\end{align*}
\]
A Cyclic Proof of $N \vdash E x, O x$

\[
\begin{align*}
&\vdash E \ 0, \ O \ 0 \\
&\frac{x = 0 \vdash E \ x, \ O \ x}{=} \text{(L)} \\
&\vdash E \ x, \ O \ x \\
\end{align*}
\]

\[
\begin{align*}
&N \ y \vdash E \ y, \ O \ sy \\
&\frac{N \ y \vdash E \ sy, \ O \ sy}{=} \text{(L)} \\
&\frac{x = sy, \ N \ y \vdash E \ x, \ O \ x}{=} \text{(L)} \\
&\vdash E \ x, \ O \ x \\
\end{align*}
\]

\[
\begin{align*}
&N \ y \vdash E \ y, \ O \ sy \\
&\frac{N \ y \vdash E \ sy, \ O \ sy}{=} \text{(L)} \\
&\frac{x = sy, \ N \ y \vdash E \ x, \ O \ x}{=} \text{(L)} \\
&\vdash E \ x, \ O \ x \\
\end{align*}
\]

Suppose $N \ x \vdash E \ x, \ O \ x$ is not valid:

\[
\begin{align*}
&\exists \ m_1 > \ldots > m_7
\end{align*}
\]
A Cyclic Proof of $N x \vdash E x, O x$

\[
\begin{align*}
&\vdash E 0, O 0 \\
&\vdash x = 0 \vdash E x, O x \\
&\vdash N x \vdash E x, O x \\
&\vdash E y, O y \\
&\vdash N y \vdash E y, O y \\
&\vdash N y \vdash E sy, O sy \\
&\vdash x = sy, N y \vdash E x, O x \\
&\vdash N y \vdash E y, O sy \\
&\vdash N y \vdash E sy, O sy \\
&\vdash x = sy, N y \vdash E x, O x \\
&\vdash N x \vdash E x, O x
\end{align*}
\]

Suppose $N x \vdash E x, O x$ is not valid: 

\[
\begin{align*}
&x = 0 \vdash E x, O x \\
&N x \vdash E x, O x
\end{align*}
\]
A Cyclic Proof of $N x \vdash E x, O x$

\[
\begin{align*}
N x & \vdash E x, O x \\
\text{(Subst)} & \\
N y & \vdash E y, O y \\
\text{(OR\_1)} & \\
N y & \vdash E y, O sy \\
\text{(ER\_2)} & \\
N y & \vdash E sy, O sy \\
= & \\
x & = sy, N y \vdash E x, O x \\
\text{(Case N)} & \\
N x & \vdash E x, O x
\end{align*}
\]
A Cyclic Proof of $N \vdash E \, x, O \, x$

$N \vdash E \, x, O \, x$

$\vdash E \, 0, O \, 0$  \hspace{1cm} (ER$_1$)

$x = 0 \vdash E \, x, O \, x$  \hspace{1cm} (=L)

$N \vdash E \, x, O \, x$  \hspace{1cm} (Case N)

$N \vdash E \, y, O \, y$

$\vdash E \, 0, O \, 0$  \hspace{1cm} (Subst)

$N \vdash E \, y, O \, y$  \hspace{1cm} (OR$_1$)

$N \vdash E \, y, O \, sy$

$x = sy, N \vdash E \, x, O \, x$  \hspace{1cm} (=L)

$N \vdash E \, sy, O \, sy$  \hspace{1cm} (ER$_2$)

$N \vdash E \, sy, O \, sy$

$x = sy, N \vdash E \, x, O \, x$
A Cyclic Proof of $N \vdash E \ x, O \ x$

- Suppose $N \vdash E \ x, O \ x$ is not valid:
  \[
  [X]_{m_1}
  \]
A Cyclic Proof of $N x \vdash E x, O x$

- Suppose $N x \vdash E x, O x$ is not valid:

  $[x]_{m_1} > [y]_{m_2}$
A Cyclic Proof of $N x \vdash E x, O x$

- Suppose $N x \vdash E x, O x$ is not valid:

\[
[x]_{m_1} > [y]_{m_2} = [y]_{m_3}
\]
A Cyclic Proof of $N x \vdash E x, O x$

- Suppose $N x \vdash E x, O x$ is not valid:

$$[x]_{m_1} > [y]_{m_2} = [y]_{m_3} = [y]_{m_4}$$
A Cyclic Proof of $N \vdash E \ x, \ O \ x$

- Suppose $N \vdash E \ x, \ O \ x$ is not valid:
  \[
  [x]_{m_1} > [y]_{m_2} = [y]_{m_3} = [y]_{m_4} = [y]_{m_5}
  \]
A Cyclic Proof of $N x \vdash E x, O x$

- Suppose $N x \vdash E x, O x$ is not valid:

  $[x] m_1 > [y] m_2 = [y] m_3 = [y] m_4 = [y] m_5 = [x] m_6$
A Cyclic Proof of $N x \vdash E x, O x$

\[ \begin{align*}
N x & \vdash E x, O x \\
\hline
N x & \vdash E 0, O 0 \quad (ER_1) \\
\hline
x = 0 & \vdash E x, O x \quad (=L) \\
\hline
\end{align*} \]

\[ \begin{align*}
N y & \vdash E y, O y \quad (O R_1) \\
\hline
N y & \vdash E y, O sy \quad (ER_2) \\
\hline
x = sy, N y & \vdash E x, O x \quad (=L) \\
\hline
\end{align*} \]

- Suppose $N x \vdash E x, O x$ is not valid:

\[ [x]_{m_1} > [y]_{m_2} = [y]_{m_3} = [y]_{m_4} = [y]_{m_5} = [x]_{m_6} > [y]_{m_7} \ldots \]
A Cyclic Proof of $N \vdash E \times, O \times$

1. $\vdash E \times, O \times$ (ER)  
   $x = 0 \vdash E \times, O \times$ (=L)  
2. $N \vdash E \times, O \times$  
   $N y \vdash E \times, O \times$ (Subst)  
   $N y \vdash E y, O y$ (OR1)  
   $N y \vdash E y, O sy$  
3. $N \vdash E \times, O \times$  
   $N y \vdash E \times, O \times$ (Subst)  
   $N y \vdash E sy, O sy$ (ER2)  
   $N y \vdash E sy, O sy$ (=L)  
   $x = sy, N y \vdash E \times, O \times$ (=L)  
4. $N \vdash E \times, O \times$ (Case N)

- Suppose $N x \vdash E \times, O \times$ is not valid:

  $n_1 > n_2 > n_3 > \ldots$
Example: Separation Logic

- Separation Logic incorporates formulas for representing heap memory:
Example: Separation Logic

- Separation Logic incorporates formulas for representing heap memory:
  - `emp` denotes the empty heap
Example: Separation Logic

- Separation Logic incorporates formulas for representing heap memory:
  - `emp` denotes the empty heap
  - `x \mapsto \vec{v}` is the single-cell heap containing values \(\vec{v}\) at memory location \(x\)
Example: Separation Logic

- Separation Logic incorporates formulas for representing heap memory:
  - `emp` denotes the empty heap
  - `x` → `v` is the single-cell heap containing values `v` at memory location `x`
  - `F * G` denotes a heap `h` that can be split into disjoint sub-heaps `h_1` and `h_2` which model `F` and `G` respectively
Example: Separation Logic

- Separation Logic incorporates formulas for representing heap memory:
  - \texttt{emp} denotes the empty heap
  - \( x \mapsto \vec{v} \) is the single-cell heap containing values \( \vec{v} \) at memory location \( x \)
  - \( F \star G \) denotes a heap \( h \) that can be split into disjoint sub-heaps \( h_1 \) and \( h_2 \) which model \( F \) and \( G \) respectively

- Inductive predicates now represent data-structures, e.g. linked-list segments:

\[
\begin{align*}
  x = y \land \texttt{emp} & \quad \Rightarrow \quad \text{ls}(x, y) \\
  x \mapsto z \star \text{ls}(z, y) & \quad \Rightarrow \quad \text{ls}(x, y)
\end{align*}
\]
A Cyclic Proof of List Segment Concatenation

\[ \text{ls}(x, y) \ast \text{ls}(y, z) \vdash \text{ls}(x, z) \]
A Cyclic Proof of List Segment Concatenation

\[(x = y \land \text{emp}) \times \text{ls}(y, z) \vdash \text{ls}(x, z)\]

\[x \leftrightarrow v \times \text{ls}(v, y) \times \text{ls}(y, z) \vdash \text{ls}(x, z)\]  
(Case \text{ls})

\[\text{ls}(x, y) \times \text{ls}(y, z) \vdash \text{ls}(x, z)\]
A Cyclic Proof of List Segment Concatenation

\[\text{emp} \ast \text{ls}(x, z) \vdash \text{ls}(x, z)\]

\[\begin{array}{c}
\vdash \text{ls}(x, z) \\
\end{array}\]

\[\begin{array}{c}
(x = y \land \text{emp}) \ast \text{ls}(y, z) \vdash \text{ls}(x, z) \\
\end{array}\]

\[\vdash \text{ls}(x, z)\]

\[\begin{array}{c}
x \mapsto v \ast \text{ls}(v, y) \ast \text{ls}(y, z) \vdash \text{ls}(x, z) \\
\end{array}\]

\[\vdash \text{ls}(x, z)\]

\[\text{ls}(x, y) \ast \text{ls}(y, z) \vdash \text{ls}(x, z)\]
A Cyclic Proof of List Segment Concatenation

\[
\begin{align*}
\text{ls}(x, z) & \vdash \text{ls}(x, z) \\
\text{emp} \ast \text{ls}(x, z) & \vdash \text{ls}(x, z) \\
(x = y \land \text{emp}) \ast \text{ls}(y, z) & \vdash \text{ls}(x, z)
\end{align*}
\]

(Case \text{ls})

\[
\begin{align*}
\text{ls}(x, y) \ast \text{ls}(y, z) & \vdash \text{ls}(x, z)
\end{align*}
\]
A Cyclic Proof of List Segment Concatenation

\[
\begin{align*}
\text{(Id)} & \quad \text{ls}(x, z) \vdash \text{ls}(x, z) \\
\text{(\equiv)} & \quad \text{emp} * \text{ls}(x, z) \vdash \text{ls}(x, z) \\
\text{ (=L)} & \quad (x = y \land \text{emp}) * \text{ls}(y, z) \vdash \text{ls}(x, z) \\
\vdots & \quad \vdots \\
\text{ (Case ls)} & \quad x \mapsto v * \text{ls}(v, y) * \text{ls}(y, z) \vdash \text{ls}(x, z) \\
\text{(Case ls)} & \quad \text{ls}(x, y) * \text{ls}(y, z) \vdash \text{ls}(x, z)
\end{align*}
\]
A Cyclic Proof of List Segment Concatenation

\[ \text{(Id)} \]
\[
\text{ls}(x, z) \vdash \text{ls}(x, z) \quad \text{(=} \) \\
\text{emp} \ast \text{ls}(x, z) \vdash \text{ls}(x, z) \quad \text{(=} = L) \\
(x = y \land \text{emp}) \ast \text{ls}(y, z) \vdash \text{ls}(x, z) \quad \text{(Case ls)}
\]

\[ \text{ls}(x, y) \ast \text{ls}(y, z) \vdash \text{ls}(x, z) \quad \text{(lsR2)} \]

\[ x \mapsto v \ast \text{ls}(v, y) \ast \text{ls}(y, z) \vdash x \mapsto v \ast \text{ls}(v, z) \]

\[ x \mapsto v \ast \text{ls}(v, y) \ast \text{ls}(y, z) \vdash \text{ls}(x, z) \]

\[ \text{ls}(x, y) \ast \text{ls}(y, z) \vdash \text{ls}(x, z) \]
A Cyclic Proof of List Segment Concatenation

\[ \begin{align*}
\text{(Id)} \\
\text{ls}(x, z) & \vdash \text{ls}(x, z) \\
\hline
\equiv \\
\text{emp} * \text{ls}(x, z) & \vdash \text{ls}(x, z) \\
\equiv \hline
\text{(=L)} \\
(x = y \land \text{emp}) * \text{ls}(y, z) & \vdash \text{ls}(x, z) \\
\end{align*} \]
A Cyclic Proof of List Segment Concatenation

\[
\begin{align*}
\frac{\text{(Id)}}{\text{ls}(x, z) \vdash \text{ls}(x, z)} \\
\frac{\text{(\equiv)}}{\text{emp} \ast \text{ls}(x, z) \vdash \text{ls}(x, z)} \\
\frac{\text{(=L)}}{(x = y \land \text{emp}) \ast \text{ls}(y, z) \vdash \text{ls}(x, z)}
\end{align*}
\]
A Cyclic Proof of List Segment Concatenation

\[
\begin{align*}
\text{(Id)} & \quad \text{ls}(x, z) \vdash \text{ls}(x, z) \\
\text{(≡)} & \quad \text{emp} * \text{ls}(x, z) \vdash \text{ls}(x, z) \\
\text{(≡L)} & \quad (x = y \land \text{emp}) * \text{ls}(y, z) \vdash \text{ls}(x, z)
\end{align*}
\]

\[
\begin{align*}
\text{(Id)} & \quad \text{ls}(x, y) * \text{ls}(y, z) \vdash \text{ls}(x, z) \\
\text{(Subst)} & \quad \text{ls}(v, y) * \text{ls}(y, z) \vdash \text{ls}(v, z) \\
\text{(*)} & \quad x \mapsto v * \text{ls}(v, y) * \text{ls}(y, z) \vdash x \mapsto v * \text{ls}(v, z) \\
\text{(lsR₂)} & \quad x \mapsto v * \text{ls}(v, y) * \text{ls}(y, z) \vdash \text{ls}(x, z) \\
\text{(Case ls)} & \quad \text{ls}(x, y) * \text{ls}(y, z) \vdash \text{ls}(x, z)
\end{align*}
\]
A Cyclic Proof of List Segment Concatenation

\[
\begin{align*}
\text{(Id)} \quad \text{ls}(x, z) &\vdash \text{ls}(x, z) \\
\equiv \quad \text{emp} \ast \text{ls}(x, z) &\vdash \text{ls}(x, z) \\
(x = y \land \text{emp}) \ast \text{ls}(y, z) &\vdash \text{ls}(x, z)
\end{align*}
\]

\[
\begin{align*}
\text{(Id)} \quad x \mapsto v &\vdash x \mapsto v \\
\text{(Subst)} \quad \text{ls}(v, y) \ast \text{ls}(y, z) &\vdash \text{ls}(v, z) \\
\ast \quad x \mapsto v \ast \text{ls}(v, y) \ast \text{ls}(y, z) &\vdash x \mapsto v \ast \text{ls}(v, z) \\
\text{(lsR)} \quad x \mapsto v \ast \text{ls}(v, y) \ast \text{ls}(y, z) &\vdash \text{ls}(x, z) \\
\text{(Case ls)} \quad \text{ls}(x, y) \ast \text{ls}(y, z) &\vdash \text{ls}(x, z)
\end{align*}
\]
A Cyclic Proof of List Segment Concatenation

\[ \begin{align*}
\text{ls}(x, z) & \vdash \text{ls}(x, z) \\
\equiv & \\
\text{emp} \ast \text{ls}(x, z) & \vdash \text{ls}(x, z) \\
\implies & \\
(x = y \land \text{emp}) \ast \text{ls}(y, z) & \vdash \text{ls}(x, z)
\end{align*} \]

\[ \begin{align*}
\text{ls}(v, y) & \ast \text{ls}(y, z) \vdash \text{ls}(v, z) \\
\ast & \\
(x \mapsto v) \ast \text{ls}(v, y) & \ast \text{ls}(y, z) \vdash x \mapsto v \ast \text{ls}(v, z) \\
\text{lsR}_2 & \\
(x \mapsto v) \ast \text{ls}(v, y) & \ast \text{ls}(y, z) \vdash \text{ls}(x, z)
\end{align*} \]

\[ \begin{align*}
\text{ls}(x, y) & \ast \text{ls}(y, z) \vdash \text{ls}(x, z)
\end{align*} \]
Global Soundness for Cyclic Proof: Elements

• Fix some values that we can **trace** along paths in the proof
  • In our examples: inductive predicate instances
Global Soundness for Cyclic Proof: Elements

- Fix some values that we can trace along paths in the proof
  - In our examples: inductive predicate instances
- Map (model, trace-value) pairs to elements of a w.f. set
Global Soundness for Cyclic Proof: Elements

- Fix some values that we can trace along paths in the proof
  - In our examples: inductive predicate instances
- Map \((\text{model}, \text{trace-value})\) pairs to elements of a w.f. set
  - Inductive definitions induce a monotone operator \(\varphi\) on sets of models

\[ \begin{align*}
\text{Interpret the inductive definitions using the lfp} \\
\varphi(X) &\subseteq \varphi(\varphi(X)) \\
\vdots &\vdots \\
\varphi(\varphi(\varphi(X))) &\subseteq X \end{align*} \]
Global Soundness for Cyclic Proof: Elements

- Fix some values that we can trace along paths in the proof
  - In our examples: inductive predicate instances
- Map (model, trace-value) pairs to elements of a w.f. set
  - Inductive definitions induce a monotone operator $\varphi$ on sets of models
  - Interpret the inductive definitions using the lfp

$$\varphi(\bot) \subseteq \varphi(\varphi(\bot)) \subseteq \ldots \subseteq \varphi^\omega(\bot) \subseteq \ldots \subseteq \mu X. \varphi(X)$$
Global Soundness for Cyclic Proof: Elements

- Fix some values that we can trace along paths in the proof
  - In our examples: inductive predicate instances
- Map (model, trace-value) pairs to elements of a w.f. set
  - Inductive definitions induce a monotone operator $\varphi$ on sets of models
  - Interpret the inductive definitions using the lfp
    \[
    \varphi(\bot) \subseteq \varphi(\varphi(\bot)) \subseteq \ldots \subseteq \varphi^\omega(\bot) \subseteq \ldots \subseteq \mu X. \varphi(X)
    \]
- Map $(m, P \bar{t})$ to the least approximation $\varphi^\alpha(\bot)$ of $P$ in which $m$ appears
Global Soundness for Cyclic Proof: Elements

- Fix some values that we can trace along paths in the proof
  - In our examples: inductive predicate instances
- Map (model, trace-value) pairs to elements of a w.f. set
  - Inductive definitions induce a monotone operator $\varphi$ on sets of models
  - Interpret the inductive definitions using the lfp

$$\varphi(\perp) \subseteq \varphi(\varphi(\perp)) \subseteq \ldots \subseteq \varphi^\omega(\perp) \subseteq \ldots \subseteq \mu X. \varphi(X)$$

- Map $(m, P \bar{t})$ to the least approximation $\varphi^\alpha(\perp)$ of $P$ in which $m$ appears
- Identify the progression points of the proof system, e.g.

$$x = y \land \text{emp} \vdash F \quad x \mapsto v \ast \text{ls}(v, y) \vdash F$$

$$\text{ls}(x, y) \vdash F \quad \text{(Case ls)}$$
Global Soundness for Cyclic Proof: General Principle

- Impose global soundness condition on proof graphs:
  - Every infinite path must have an infinitely progressing trace
  - This condition is decidable using Büchi automata
Global Soundness for Cyclic Proof: General Principle

- Impose global soundness condition on proof graphs:
  - Every infinite path must have an infinitely progressing trace
  - This condition is decidable using Büchi automata

- We obtain an infinite descent proof-by-contradiction:
• Impose global soundness condition on proof graphs:
  • Every infinite path must have an infinitely progressing trace
  • This condition is decidable using Büchi automata

• We obtain an infinite descent proof-by-contradiction:
  • Assume the conclusion of the proof is invalid
Global Soundness for Cyclic Proof: General Principle

- Impose global soundness condition on proof graphs:
  - Every infinite path must have an infinitely progressing trace
  - This condition is decidable using Büchi automata

- We obtain an infinite descent proof-by-contradiction:
  - Assume the conclusion of the proof is invalid
  - Local soundness implies an infinite sequence of (counter) models
Global Soundness for Cyclic Proof: General Principle

• Impose global soundness condition on proof graphs:
  • Every infinite path must have an indefinitely progressing trace
  • This condition is decidable using Büchi automata

• We obtain an infinite descent proof-by-contradiction:
  • Assume the conclusion of the proof is invalid
  • Local soundness implies an infinite sequence of (counter) models
  • Global soundness then implies an infinite descending chain in a well-founded set
Cyclic Proof vs Explicit Induction

- Explicit induction requires induction hypothesis $F$ up-front

\[
\begin{array}{c}
N 0 \\
N x
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash F[0] \\
\Gamma, F[x] \vdash F[sx], \Delta \\
\Gamma, F[t] \vdash \Delta
\end{array}
\quad
(\text{Ind } N)
\]

- Cyclic proof enables 'discovery' of induction hypotheses
- Complex induction schemes naturally represented by nested and overlapping cycles
- The explicit induction rules are derivable in the cyclic system (cf. Brotherston & Simpson)
Cyclic Proof vs Explicit Induction

- Explicit induction requires induction hypothesis $F$ up-front

\[
\begin{align*}
\begin{array}{c}
N 0 \\
N x
\end{array} & \quad \begin{array}{c}
N x \\
N sx
\end{array} & \quad \Gamma \vdash F[0] & \quad \Gamma, F[x] \vdash F[sx], \Delta & \quad \Gamma, F[t] \vdash \Delta \\
& & \quad \Gamma, N t \vdash \Delta
\end{align*}
\]

(Ind $N$)

- Cyclic proof enables 'discovery' of induction hypotheses
Cyclic Proof vs Explicit Induction

- Explicit induction requires induction hypothesis $F$ up-front

\[
\begin{align*}
\Gamma &\vdash F[0] & \Gamma, F[x] &\vdash F[sx], \Delta & \Gamma, F[t] &\vdash \Delta \\
\Gamma, N &\vdash \Delta
\end{align*}
\]

- Cyclic proof enables 'discovery' of induction hypotheses

- Complex induction schemes naturally represented by nested and overlapping cycles
Cyclic Proof vs Explicit Induction

- Explicit induction requires induction hypothesis $F$ up-front

$$
\begin{array}{c}
\Gamma \vdash F[0] \\
\Gamma, F[x] \vdash F[sx], \Delta \\
\Gamma, F[t] \vdash \Delta
\end{array}
$$

$(\text{Ind } N)$

- Cyclic proof enables 'discovery' of induction hypotheses

- Complex induction schemes naturally represented by nested and overlapping cycles

- The explicit induction rules are derivable in the cyclic system (cf. Brotherston & Simpson)
Cyclic Proofs vs Infinite Proofs

- Cyclic proofs are the (strict) regular subset of the set of non-well-founded proof trees
Cyclic Proofs vs Infinite Proofs

• Cyclic proofs are the (strict) regular subset of the set of non-well-founded proof trees

• Theorem (Brotherston & Simpson): the full infinite system is cut-free complete
• Cyclic proofs are the (strict) regular subset of the set of non-well-founded proof trees

• Theorem (Brotherston & Simpson): the full infinite system is cut-free complete

• Cut is likely not eliminable in the cyclic sub-system
A Simple Imperative Language

(Terms) \( t ::= \text{nil} \mid x \)

(Boolean Expressions) \( B ::= t=t \mid t!\neq t \)

(Programs) \( C ::= \varepsilon \) (stop)
\( \mid x:=t;C \) (assignment)
\( \mid x:=[y];C \mid [x]:=y;C \) (load/store)
\( \mid \text{free}(x);C \mid x:=\text{new};C \) (de_allocate)
\( \mid \text{if } B \text{ then } C;C \) (conditional)
\( \mid \text{while } B \text{ do } C;C \) (loop)

The following program deallocates a linked list:

\[
\text{while } x!\neq \text{nil} \text{ do } y:=\lfloor x \rfloor;\text{free}(x);x=y;
\]
(Terms) \[ t ::= \text{nil} \mid x \]

(Boolean Expressions) \[ B ::= t = t \mid t \neq t \]

(Programs) \[ C ::= \varepsilon \quad \text{(stop)} \]
\[ \mid x := t; C \quad \text{(assignment)} \]
\[ \mid x := [y]; C \mid [x] := y; C \quad \text{(load/store)} \]
\[ \mid \text{free}(x); C \mid x := \text{new}; C \quad \text{(de_allocate)} \]
\[ \mid \text{if } B \text{ then } C; C \quad \text{(conditional)} \]
\[ \mid \text{while } B \text{ do } C; C \quad \text{(loop)} \]

- The following program deallocates a linked list:

  \[
  \text{while } x \neq \text{nil do } y := [x]; \text{free}(x); x = y;
  \]
• We use Hoare logic for proving triples \{P\} C \{Q\} using Separation Logic as an assertion language
Program Verification by Symbolic Execution

- We use Hoare logic for proving triples $\{P\} C \{Q\}$ using Separation Logic as an assertion language.

- Program commands are executed *symbolically* by the proof rules, e.g.

\[
\begin{align*}
\text{(load):} & \quad \{x = v[x'/x] \land (P \ast y \mapsto v)[x'/x]\} \ C \ \{Q\} \\
& \quad \{P \ast y \mapsto v\} \ x := [y] ; C \ \{Q\}
\end{align*}
\]
Program Verification by Symbolic Execution

• We use Hoare logic for proving triples \( \{P\} C \{Q\} \) using Separation Logic as an assertion language.

• Program commands are executed \textit{symbolically} by the proof rules, e.g.

\[
\begin{align*}
\text{(load):} & \quad \{x = v[x'/x] \land (P \ast y \hookrightarrow v)[x'/x]\} C \{Q\} \\
& \quad (x' \text{ fresh}) \\
& \quad \{P \ast y \hookrightarrow v\} x := [y] ; C \{Q\}
\end{align*}
\]

\[
\begin{align*}
\text{(free):} & \quad \{P\} C \{Q\} \\
& \quad \{P \ast x \hookrightarrow v\} \text{free}(x) ; C \{Q\}
\end{align*}
\]
• The standard Hoare rule for handling *while* loops:

\[
\begin{array}{c}
\{B \land P\} C_1 \{P\} \quad \{\neg B \land P\} C_2 \{Q\} \\
\hline
\{P\while B \text{ do } C_1 ; C_2 \{Q\}
\end{array}
\]
Handling Loops in Cyclic Proofs

- The standard Hoare rule for handling \texttt{while} loops:

\[
\begin{align*}
\{t = z \land B \land P\} C_1 \{t < z \land P\} \quad \{\neg B \land P\} C_2 \{Q\} \\
\{P\} \text{while } B \text{ do } C_1; C_2 \{Q\}
\end{align*}
\]

\(t\) is the loop \texttt{variant}
Handling Loops in Cyclic Proofs

- The standard Hoare rule for handling `while` loops:

\[
\begin{align*}
&\{t = z \land B \land P\} C_1 \{t < z \land P\} \quad \{\neg B \land P\} C_2 \{Q\} \\
\hline
&\{P\} \text{while } B \text{ do } C_1; C_2 \{Q\}
\end{align*}
\]

\(t\) is the loop variant

- With cyclic proof, it is enough just to **unfold** loops

\[
\begin{align*}
&\{B \land P\} C_1 ; \text{while } B \text{ do } C_1; C_2 \{Q\} \quad \{\neg B \land P\} C_2 \{Q\} \\
\hline
&\{P\} \text{while } B \text{ do } C_1; C_2 \{Q\}
\end{align*}
\]
Example: Deallocating the Linked List

```plaintext
while x!=nil do y:=\[x\]; free(x); x=y;
```

Example: Deallocating the Linked List

\{ls(x, nil)\} \textbf{while} x \neq \text{nil} \textbf{do} y := [x]; \text{free}(x); x = y;
Example: Deallocating the Linked List

\{ls(x,nil)\} \textbf{while} x\neq\textit{nil} \textbf{do} y:=\{x\};\textit{free}(x);x=y; \{\textit{emp}\}
Example: Deallocating the Linked List

\[
\{\text{ls}(x, \text{nil})\} \ \text{while} \ x \neq \text{nil} \ \text{do} \ y := \{x\}; \text{free}(x); x := y; \ \{\text{emp}\}
\]
Example: Deallocating the Linked List

\[
\{\text{ls}(x, \text{nil})\} \quad \text{while} \quad x \neq \text{nil} \quad \text{do} \quad y := [x]; \quad \text{free}(x); \quad x = y; \quad \{\text{emp}\}
\]

\[
\begin{align*}
\{ & x \neq \text{nil} \\
& \land \text{ls}(x, \text{nil}) \} \quad y := [x]; \ldots \quad \{\text{emp}\} \\
& \vdots \\
& \vdots \\
\end{align*}
\]

\[
\underbrace{\vdots}_{(\text{while})} \quad \{\text{ls}(x, \text{nil})\} \quad \text{while} \quad \ldots \quad \{\text{emp}\}
\]

\[
\begin{align*}
\{ & x = \text{nil} \\
& \land \text{ls}(x, \text{nil}) \} \quad \in \quad \{\text{emp}\}
\end{align*}
\]
Example: Deallocating the Linked List

\{\text{ls}(x, \text{nil})\} \quad \text{while } x \neq \text{nil} \quad \text{do } y := [x]; \text{free}(x); x = y; \quad \{\text{emp}\}
Example: Deallocating the Linked List

\{
\text{\texttt{ls}}(x, \text{\texttt{nil}})\} \quad \text{while} \ x\neq \text{\texttt{nil}} \quad \text{do} \quad y:= \text{\texttt{[x]}}; \quad \text{\texttt{free}}(x); \quad x=y; \quad \{\text{emp}\}
\}
Example: Deallocating the Linked List

\{ls(x, nil)\} \text{ while } x \neq \text{nild } do \ y := [x]; \text{free}(x); \ x = y; \ \{\text{emp}\}
Example: Deallocating the Linked List

\[ \{\text{ls}(x, \text{nil})\} \text{ while } x \neq \text{nil} \text{ do } y := [x]; \text{free}(x); x = y; \quad \{\text{emp}\} \]
Example: Deallocating the Linked List

\{\text{ls}(x, \text{nil})\} \text{ while } x \neq \text{nil} \text{ do } y := [x]; \text{free}(x); x = y; \{\text{emp}\}

\begin{align*}
\{\text{ls}(y, \text{nil})\} & \quad x = y; \ldots \{\text{emp}\} \\
\quad \text{(free)} & \\
\frac{\{x \mapsto y\}}{\{x \mapsto y\}} & \text{free}(x); \ldots \{\text{emp}\} \\
\quad \text{(load)} & \\
\frac{\{x \mapsto y \quad \text{ls}(y, \text{nil})\}}{\{x \mapsto y \quad \text{ls}(y, \text{nil})\}} & y := [x]; \ldots \{\text{emp}\} \\
\quad \text{(unfold \text{ls})} & \\
\frac{\{x \neq \text{nil} \quad \text{ls}(x, \text{nil})\}}{\{x \neq \text{nil} \quad \text{ls}(x, \text{nil})\}} & y := [x]; \ldots \{\text{emp}\} \\
\quad \text{\ldots} & \\
\quad \text{\ldots} & \\
\quad \text{(unfold \text{ls})} & \\
\frac{\{x = \text{nil} \quad \text{ls}(x, \text{nil})\}}{\{x = \text{nil} \quad \text{ls}(x, \text{nil})\}} & \in \{\text{emp}\} \\
\quad \text{(while)} & \\
\frac{\{\text{ls}(x, \text{nil})\}}{\{\text{ls}(x, \text{nil})\} \text{ while } \ldots \{\text{emp}\}} & \\
\end{align*}
Example: Deallocating the Linked List

{\text{ls}(x,\text{nil})} \quad \text{while} \ x \neq \text{nil} \ \text{do} \quad y := \{x\} \ ; \ \text{free}(x) \ ; \ x := y \quad \{\text{emp}\}

\begin{align*}
\{\text{ls}(x,\text{nil})\} \quad \text{while} \ldots \quad \{\text{emp}\} \\
\{\text{ls}(y,\text{nil})\} \quad x := y \ldots \quad \{\text{emp}\} \\
\{\text{ls}(y,\text{nil})\} \quad x := y \ldots \quad \{\text{emp}\} \\
\end{align*}

\begin{align*}
\{\text{ls}(x,\text{nil})\} \quad \text{while} \ldots \quad \{\text{emp}\} \\
\{\text{ls}(y,\text{nil})\} \quad x := y \ldots \quad \{\text{emp}\} \\
\{\text{ls}(y,\text{nil})\} \quad x := y \ldots \quad \{\text{emp}\} \\
\{\text{ls}(x,\text{nil})\} \quad \text{while} \ldots \quad \{\text{emp}\}
\end{align*}
Example: Deallocating the Linked List

\{\text{ls}(x, \text{nil})\} \text{ while } x \neq \text{nil} \text{ do } y \leftarrow [x]; \text{free}(x); x = y; \{\text{emp}\}

\begin{align*}
\{\text{ls}(x, \text{nil})\} & \text{ while } x \neq \text{nil} \text{ do } y \leftarrow [x]; \text{free}(x); x = y; \{\text{emp}\} \\
& \text{(assign)} \\
\{\text{ls}(y, \text{nil})\} x = y; \ldots \{\text{emp}\} \\
& \text{(free)} \\
\{ x \mapsto y \} & \text{ free}(x); \ldots \{\text{emp}\} \\
& \{ x \mapsto v \} y \leftarrow [x]; \ldots \{\text{emp}\} \\
& \text{(load)} \\
\{ x \neq \text{nil} \} & \text{ unfold \text{ls}} \\
\{ x \neq \text{nil} \} & \{ x \neq \text{nil} \} \text{ y} \leftarrow [x]; \ldots \{\text{emp}\} \\
& \{ x \neq \text{nil} \} \text{ y} \leftarrow [x]; \ldots \{\text{emp}\} \\
& \text{(unfold \text{ls})} \\
\{ x = \text{nil} \} & \{ x = \text{nil} \} \varepsilon \{\text{emp}\} \\
& \{ x = \text{nil} \} \varepsilon \{\text{emp}\} \\
& \{ x = \text{nil} \} \varepsilon \{\text{emp}\} \\
& \{ x = \text{nil} \} \varepsilon \{\text{emp}\} \\
& \text{(while)} \\
\{\text{ls}(x, \text{nil})\} & \text{ while } x \neq \text{nil} \text{ do } y \leftarrow [x]; \text{free}(x); x = y; \{\text{emp}\}
\end{align*}
Example: Deallocating the Linked List

\{\text{ls}(x, \text{nil})\} \text{ while } x \neq \text{nil} \text{ do } y := [x]; \text{free}(x); x := y; \text{ \{emp\} }

\begin{align*}
\{\text{ls}(x, \text{nil})\} \text{ while } \ldots \text{ \{emp\}} & \quad \text{(assign)} \\
\{\text{ls}(y, \text{nil})\} x := y; \ldots \text{ \{emp\}} & \quad \text{(free)} \\
\{ x \leftrightarrow y \} & \quad \text{\{ls(y, nil)\}} \\
\{ x \leftrightarrow v \} & \quad \text{\{ls(v, nil)\}} \\
y := [x]; \ldots \text{ \{emp\}} & \quad \text{(load)} \\
\{ x \neq \text{nil} \} \\
\{ \text{ls}(x, \text{nil}) \} & \quad \text{\{emp\}} \\
\ldots & \quad \text{(unfold \text{ls})} \\
\{ x \neq \text{nil} \} \\
\{ \text{ls}(x, \text{nil}) \} & \quad \text{\{emp\}} \\
\ldots & \quad \text{(while)} \\
\} \quad \text{\{ls(x, nil)\}} \text{ while } \ldots \text{ \{emp\}}
Verifying Recursive Procedures

(Procedures) \( \text{proc } p(\vec{x}) \{ C \} \)
(Problems) \( C ::= \ldots \ | \ p(\vec{t}); C \)
(Procedures) \[
\text{proc } p(\vec{x}) \{ C \}
\]

(Problems) \[
C ::= \ldots \mid p(t); C
\]

The following procedure recursively deallocates a linked list
\[
\text{proc dealloc}(x) \{ 
\text{if } x \neq \text{nil} \text{ then } y := [x] \text{; free}(x) \text{; dealloc}(y) \}
\]

(pro): \[
\begin{align*}
\{P\} & \quad C \{Q\} \\
\{P\} & \quad p(\vec{x}) \{Q\}
\end{align*}
\]

\[
\begin{align*}
\{P\} & \quad C \{Q\} \\
\{P\} & \quad \text{body}(p) = C
\end{align*}
\]
Verifying Recursive Procedures

\[
\text{(Procedures)} \quad \text{proc } p(\vec{x}) \{ C \}
\]

\[
\text{(Programs)} \quad C ::= \ldots \mid p(\vec{t}) ; C
\]

\[
\begin{array}{c}
\text{(proc): } \frac{\{P\} C \{Q\}}{\{P\} p(\vec{x}) \{Q\}} \quad \text{(body}(p) = C) \\
\text{(call): } \frac{\{P\} p(\vec{t}) \{P'\} \quad \{P'\} C \{Q\}}{\{P\} p(\vec{t}) ; C \{Q\}}
\end{array}
\]

The following procedure recursively deallocates a linked list:

\[
\text{proc dealloc}(x) \{ \text{if } x \neq \text{nil then } y := [x] ; \text{free}(x) ; \text{dealloc}(y) \}
\]
Verifying Recursive Procedures

(Procedures) \[ \text{proc } p(\vec{x}) \{ C \} \]
(Progranms) \[ C ::= \ldots | p(\vec{t}); C \]

\[
\begin{align*}
\text{(proc): } & \frac{\{P\} C \{Q\}}{\{P\} p(\vec{x}) \{Q\} \quad \text{body}(p) = C} \\
\text{(call): } & \frac{\{P\} p(\vec{t}) \{P'\} \quad \{P'\} C \{Q\} \quad \{P\} p(\vec{t}); C \{Q\}}{\{P\} p(\vec{t}) \{Q\}} \\
\text{(param): } & \frac{\{P\} p(\vec{t}) \{Q\}}{\{P[t/x]\} p(\vec{t})[t/x] \{Q[t/x]\}}
\end{align*}
\]

The following procedure recursively deallocates a linked list:
\[ \text{proc dealloc}(x) \{ \text{if } x \neq \text{nil then } y := [x]; \text{free}(x); \text{dealloc}(y); \} \]
Verifying Recursive Procedures

\[
\text{(Procedures)} \quad \text{proc } p(\vec{x}) \{ C \}
\]

\[
\text{(Programs)} \quad C ::= \ldots \mid p(\vec{t}); C
\]

\[
\begin{aligned}
\text{(proc): } & \quad \{P\} C \{Q\} \\
& \quad \{P\} p(\vec{x}) \{Q\} \\
& \quad \text{body}(p) = C \\
\text{(call): } & \quad \{P\} p(\vec{t}) \{P'\} \quad \{P'\} C \{Q\} \\
& \quad \{P\} p(\vec{t}); C \{Q\} \\
& \quad \{P\} p(\vec{t}) \{Q\} \\
& \quad \{P[t/x]\} p(\vec{t})[t/x] \{Q[t/x]\}
\end{aligned}
\]

- The following procedure recursively deallocates a linked list

\[
\text{proc dealloc}(x) \{ \text{if } x \neq \text{nil then } y := [x]; \text{free}(x); \text{dealloc}(y); \}
\]
Example: Deallocation of the Linked List (Recursively)

```
proc dealloc(x) { if x!=nil then y:=[x]; free(x); dealloc(y); }
```
Example: Deallocating the Linked List (Recursively)

\[
\text{proc dealloc(x) \{ if } x\neq \text{nil} \text{ then } y := [x]; \text{ free}(x); \text{ dealloc}(y); \}}
\]
Example: Deallocating the Linked List (Recursively)

```plaintext
proc dealloc(x) { if x!=nil then y:=\{x\}; free(x); dealloc(y); }
```
Example: Deallocating the Linked List (Recursively)

```plaintext
proc dealloc(x) { if x!=nil then y:=[x]; free(x); dealloc(y); }
```
A Subtlety with Procedures

```plaintext
proc shuffle(x) {
    if x!=nil then
        y:=[x]; reverse(y); shuffle(y); [x]:=y;
    }
```
proc shuffle(x){
    if x!=nil then
        y:=[x]; reverse(y); shuffle(y); [x]:=y;
    }

{ls(y,nil)} reverse(y); {ls(y,nil)}
----------------------------------------(frame)
{x ↔ y * ls(y,nil)} reverse(y); {x ↔ y * ls(y,nil)}
{x ↔ y * ls(y,nil)} shuffle(y); ... {ls(x,nil)}

{x ↔ y * ls(y,nil)} reverse(y); shuffle(y); ... {ls(x,nil)}
A Subtlety with Procedures

```plaintext
proc shuffle(x) {
    if x!=nil then
        y:=[x]; reverse(y); shuffle(y); [x]:=y;
}
```

\[
\begin{align*}
\{ls(y, nil)\} & \text{ reverse(y); } \{ls(y, nil)\} \\
\{x \mapsto y * ls(y, nil)\} & \text{ reverse(y); } \{x \mapsto y * ls(y, nil)\} \\
\{x \mapsto y * ls(y, nil)\} & \text{ shuffle(y); } ... \{ls(x, nil)\} \\
\{x \mapsto y * ls(y, nil)\} & \text{ reverse(y); shuffle(y); } ... \{ls(x, nil)\}
\end{align*}
\]
proc shuffle(x){
    if x!=nil then
        y:=[x]; reverse(y); shuffle(y); [x]:=y;
    }

{ls(y,nil)} reverse(y); {ls(y,nil})
________________________(frame)
{x → y * ls(y, nil)} reverse(y); {x → y * ls(y, nil})
{x → y * ls(y, nil)} shuffle(y);... {ls(x, nil)}

{x → y * ls(y, nil)} reverse(y); shuffle(y);... {ls(x, nil)}
Solution: Explicit Approximation

• We explicitly label predicate instances, e.g. $\text{ls}_\alpha(x, y)$
  • indicates which approximation to interpret them in
Solution: Explicit Approximation

- We explicitly label predicate instances, e.g. $ls_\alpha(x, y)$
  - indicates which approximation to interpret them in

- We now use these labels as the trace values, e.g.

\[
\begin{align*}
\{ls_\beta(y, nil)\} \text{ reverse}(y); \{ls_\beta(y, nil)\} \\
\{x \mapsto y * ls_\beta(y, nil)\} \text{ reverse}(y); \{x \mapsto y * ls_\beta(y, nil)\} \\
\{x \mapsto y * ls_\beta(y, nil)\} \text{ shuffle}(y); \ldots \{ls_\alpha(x, nil)\}
\end{align*}
\]
Solution: Explicit Approximation

• We explicitly label predicate instances, e.g. $\text{ls}_\alpha(x, y)$
  • indicates which approximation to interpret them in

• We now use these labels as the trace values, e.g.

\[
\begin{array}{c}
\{\text{ls}_\beta(y, \text{nil})\} \text{reverse}(y); \{\text{ls}_\beta(y, \text{nil})\} \\
\{x \mapsto y * \text{ls}_\beta(y, \text{nil})\} \text{reverse}(y); \{x \mapsto y * \text{ls}_\beta(y, \text{nil})\} \\
\{x \mapsto y * \text{ls}_\beta(y, \text{nil})\} \text{shuffle}(y); \ldots \{\text{ls}_\alpha(x, \text{nil})\} \\
\end{array}
\]

• We now need constraints on labels when unfolding, e.g.

\[
\begin{align*}
\Gamma, \beta < \alpha, t = 0 & \vdash \Delta \\
\Gamma, \beta < \alpha, t = sx, N_\beta x & \vdash \Delta \\
\Gamma, N_\alpha t & \vdash \Delta
\end{align*}
\]
(Case N)
The CYCLIST Verification Tool

- Our verification tool, CYCLIST, is implemented in OCaml
- Generic cyclic proof-search procedure using iterated depth-first search
  - Cycles are formed eagerly and discarded if unsound
- The generic proof search is parametric
  - Different proof systems implemented as separate modules
The CYCLIST Verification Tool

- Our verification tool, CYCLIST, is implemented in OCaml
- Generic cyclic proof-search procedure using iterated depth-first search
  - Cycles are formed eagerly and discarded if unsound
- The generic proof search is parametric
  - Different proof systems implemented as separate modules

github.com/ngorogiannis/cyclist
### Performance Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Time (ms)</th>
<th>LOC</th>
<th>Procs</th>
<th>Nodes</th>
<th>Back-links</th>
</tr>
</thead>
<tbody>
<tr>
<td>list traverse</td>
<td>17</td>
<td>6</td>
<td>1</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>tree traverse</td>
<td>24</td>
<td>7</td>
<td>1</td>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>list deallocate</td>
<td>14</td>
<td>7</td>
<td>1</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>tree deallocate</td>
<td>21</td>
<td>8</td>
<td>1</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>tree reflect</td>
<td>20</td>
<td>9</td>
<td>1</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>list rev. deallocate</td>
<td>43</td>
<td>18</td>
<td>1</td>
<td>49</td>
<td>2</td>
</tr>
<tr>
<td>list append</td>
<td>28</td>
<td>21</td>
<td>1</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>list reverse</td>
<td>122</td>
<td>14</td>
<td>1</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>list reverse (tail rec.)</td>
<td>31</td>
<td>18</td>
<td>1</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>list reverse (with append)</td>
<td>47</td>
<td>28</td>
<td>2</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>list filter</td>
<td>27</td>
<td>16</td>
<td>1</td>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>list partition</td>
<td>31</td>
<td>25</td>
<td>1</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>list ackermann</td>
<td>126</td>
<td>17</td>
<td>1</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>queue</td>
<td>894</td>
<td>30</td>
<td>3</td>
<td>119</td>
<td>6</td>
</tr>
<tr>
<td>functional queue</td>
<td>254</td>
<td>28</td>
<td>3</td>
<td>62</td>
<td>1</td>
</tr>
<tr>
<td>shuffle</td>
<td>202</td>
<td>23</td>
<td>2</td>
<td>79</td>
<td>4</td>
</tr>
</tbody>
</table>

Results of Experimental Evaluation on 2.93GHz Intel Core i7-870, 8GB RAM
Concluding Remarks

• Ongoing work: inferring constraints on predicate labels automatically

• Some problems remain hard, of course

  • Generalisation of inductive hypotheses

  • Finding and applying lemmas

  • Synthesizing procedure summaries (see previous point!)
Thank You
Related Work

• Cyclic proofs for FOL with inductive predicates (Brotherston & Simpson, LICS 2007)

• Cyclic proofs for Separation Logic with inductive predicates (Brotherston, SAS 2007)

• Cyclic proofs verifying simple heap-manipulating \textbf{WHILE} language (Brotherston, Bornat & Calcagno, POPL 2008)

• Implementations in Isabelle/HOL, then OCaml (Brotherston, Distefano, Gorogiannis, CADE 2011/APLAS 2012)

• Abduction of inductive predicates using cyclic proof (Brotherston & Gorogiannis, SAS 2014)

• Current Work — cyclic proofs for verifying:
  • \textit{procedural} heap-manipulating language (Rowe, Brotherston)
  • \textit{temporal} properties (Tellez Espinosa, Brotherston)