Logical Relations

Amal Ahmed
Northeastern University

OPLSS Lecture 5, June 26, 2015
Unary Logical Relations

or: Logical Predicates --- can be used to prove:

- strong normalization
- type safety (high-level and low-level languages)
- soundness of logics
- ...

Essential idea:

- A program satisfies a property if, given an input that satisfies the property, it returns an output that satisfies the property
Binary Logical Relations

Proof method that can be used to prove:

• equivalence of modules / representation independence
• noninterference in security-typed languages
• compiler correctness

Essential idea:

• Two programs (same language or different languages) are related if, given related inputs, they return related outputs
Earliest Logical Relations...

• Tait ’67: prove strong normalization for Gödel’s T
• Girard ’72: prove strong normalization for System F (reducibility candidates method)

• Plotkin ’73: *Lambda definability and logical relations*
• Statman ’85: *Logical relations and the typed lambda calculus*

• Reynolds ’83: *Types, Abstraction & Parametric Polymorphism*
• Mitchell ’86: *Representation Independence & Data Abstraction*

Lots of uses through 80’s and 90’s, but ...
L.R. Shortcomings (circa 2000)

Mostly used for “toy” languages

• Lacking support for features found in real languages:
  - recursive types (e.g., lists, objects)
  - mutable references (that can store functions, \( \exists \), \( \forall \))

Complicated, hard to extend

• Denotational vs. operational
<table>
<thead>
<tr>
<th></th>
<th>∀</th>
<th>∃</th>
<th>μ</th>
<th>ref</th>
<th>Simple Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tait’67, Girard’72</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plotkin’73, Statman’85</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reynolds’83, Mitchell’86</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pitts-Stark’93,’98</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Pitts’98,’00</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Birkedal-Harper’99,</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crary-Harper’07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appel-McAllester’01</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Benton-Leperchey’05</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ahmed’06</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Bohr-Birkedal’06</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ahmed-Dreyer-Rossberg’09</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>∀</td>
<td>∃</td>
<td>μ</td>
<td>ref</td>
<td>Simple Model</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>--------------</td>
</tr>
<tr>
<td>Plotkin’73, Statman’85</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Reynolds’83, Mitchell’86</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Pitts-Stark’93,’98 (ref int)</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pitts’98,’00 (recursive functions)</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Appel-McAllester’01</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Benton-Leperchey’05 (ref...ref int)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✗ ✗</td>
</tr>
<tr>
<td>Ahmed’06</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Bohr-Birkedal’06</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✗ ✗</td>
</tr>
<tr>
<td>Ahmed-Dreyer-Rossberg’09</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Logical Relations (highlights, 1967-2009)

<table>
<thead>
<tr>
<th></th>
<th>∀</th>
<th>∃</th>
<th>μ</th>
<th>ref</th>
<th>Simple Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plotkin’73, Statman’85</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Reynolds’83, Mitchell’86</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Pitts-Stark’93,’98 (ref int)</td>
<td></td>
<td></td>
<td></td>
<td>✔–</td>
<td>✔</td>
</tr>
<tr>
<td>Pitts’98,’00 (recursive functions)</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td>✔–</td>
</tr>
<tr>
<td>Appel-McAllester’01</td>
<td></td>
<td></td>
<td>✔–</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Benton-Leperchey’05 (ref...ref int)</td>
<td>✔</td>
<td>✔–</td>
<td></td>
<td></td>
<td>✗ ✗</td>
</tr>
<tr>
<td>Ahmed’06</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Bohr-Birkedal’06</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td>✗ ✗</td>
</tr>
<tr>
<td>Ahmed-Dreyer-Rossberg’09</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>
## Logical Relations (highlights, 1967-2009)

<table>
<thead>
<tr>
<th></th>
<th>∀</th>
<th>∃</th>
<th>μ</th>
<th>ref</th>
<th>Simple Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plotkin’73, Statman’85</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Reynolds’83, Mitchell’86</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Pitts-Stark’93,’98 (ref int)</td>
<td></td>
<td></td>
<td>✓–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pitts’98,’00 (recursive functions)</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓–</td>
<td>✓</td>
</tr>
<tr>
<td>Appel-McAllester’01</td>
<td>✓–</td>
<td></td>
<td></td>
<td>✓–</td>
<td>✓</td>
</tr>
<tr>
<td>Benton-Leperchey’05 (ref...ref int)</td>
<td>✓</td>
<td>✓–</td>
<td></td>
<td></td>
<td>✗ ✗</td>
</tr>
<tr>
<td>Ahmed’06</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓–</td>
<td>✓</td>
</tr>
<tr>
<td>Bohr-Birkedal’06</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✗ ✗</td>
</tr>
<tr>
<td>Ahmed-Dreyer-Rossberg’09</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Mutable References

Reference Types  \( \text{ref}\ \tau \)

Syntax  \( l \mid \text{new } e \mid e_1 := e_2 \mid ! e \)

\[
\begin{align*}
\text{s, new } v & \quad \longmapsto \quad s[l \mapsto v], l \quad \text{where } l \text{ fresh} \\
\text{s, ! } l & \quad \longmapsto \quad s, v \quad \text{where } s(l) = v \\
\text{s, } l := v & \quad \longmapsto \quad s[l \mapsto v], () \quad \text{where } l \in \text{dom}(s)
\end{align*}
\]
Problems in Presence of References

1. Data abstraction via local state

2. Storing functions in references

3. Interaction of $\exists$ and references

[Ahmed-Dreyer-Rossberg, POPL’09] and [Ahmed, PhD’04]
1. Data Abstraction via Local State

\[ e_1 \quad = \quad \text{let } x_1 = \text{new } 0 \text{ in} \]
\[ \lambda z : \text{unit}. \quad x_1 := !x_1 + 1; \quad !x_1 \]

\[ e_2 \quad = \quad \text{let } x_2 = \text{new } 0 \text{ in} \]
\[ \lambda z : \text{unit}. \quad x_2 := !x_2 - 1; \quad -(!x_2) \]
1. Data Abstraction via Local State

\[ e_1 = \quad \text{let } x_1 = \text{new } 0 \text{ in} \]
\[ \lambda z : \text{unit. } x_1 := !x_1 + 1; \;!x_1 \]
\[ \downarrow \quad \lambda z : \text{unit. } l_{x_1} := !l_{x_1} + 1; \;!l_{x_1} \]

\[ e_2 = \quad \text{let } x_2 = \text{new } 0 \text{ in} \]
\[ \lambda z : \text{unit. } x_2 := !x_2 - 1; -(!x_2) \]
\[ \downarrow \quad \lambda z : \text{unit. } l_{x_2} := !l_{x_2} - 1; -(!l_{x_2}) \]
1. Data Abstraction via Local State

\[ e_1 \overset{=} = \text{let } x_1 = \text{new } 0 \text{ in} \]
\[ \lambda z : \text{unit. } x_1 := !x_1 + 1; !x_1 \]
\[ \downarrow \lambda z : \text{unit. } l_{x_1} := !l_{x_1} + 1; !l_{x_1} \]

\[ e_2 \overset{=} = \text{let } x_2 = \text{new } 0 \text{ in} \]
\[ \lambda z : \text{unit. } x_2 := !x_2 - 1; -(!x_2) \]
\[ \downarrow \lambda z : \text{unit. } l_{x_2} := !l_{x_2} - 1; -(!l_{x_2}) \]
1. Data Abstraction via Local State

\[ e_1 = \text{let } x_1 = \text{new } 0 \text{ in } \lambda z : \text{unit. } x_1 := !x_1 + 1; !x_1 \]

\[ \Downarrow \lambda z : \text{unit. } l_{x_1} := !l_{x_1} + 1; !l_{x_1} \]

\[ e_2 = \text{let } x_2 = \text{new } 0 \text{ in } \lambda z : \text{unit. } x_2 := !x_2 - 1; -(!x_2) \]

\[ \Downarrow \lambda z : \text{unit. } l_{x_2} := !l_{x_2} - 1; -(!l_{x_2}) \]

\[ S = \{(s_1, s_2) \mid s_1(l_{x_1}) = -s_2(l_{x_2})\} \]
2. Storing Functions in References

\[ e_1 = \text{let } x_1 = \text{new } 0 \text{ in } \]
\[ \text{let } f_1 = \lambda z : \text{unit. } x_1 := !x_1 + 1; !x_1 \text{ in } \]
\[ \text{new } f_1 \]

\[ e_2 = \text{let } x_2 = \text{new } 0 \text{ in } \]
\[ \text{let } f_2 = \lambda z : \text{unit. } x_2 := !x_2 - 1; -(!x_2) \text{ in } \]
\[ \text{new } f_2 \]
2. Storing Functions in References

\[ e_1 = \text{let } x_1 = \text{new } 0 \text{ in} \]
\[ \text{let } f_1 = \lambda z : \text{unit. } x_1 := !x_1 + 1; !x_1 \text{ in} \]
\[ \text{new } f_1 \]
\[ \Downarrow \]
\[ l_{f_1} \]

\[ e_2 = \text{let } x_2 = \text{new } 0 \text{ in} \]
\[ \text{let } f_2 = \lambda z : \text{unit. } x_2 := !x_2 - 1; -(!x_2) \text{ in} \]
\[ \text{new } f_2 \]
\[ \Downarrow \]
\[ l_{f_2} \]
2. Storing Functions in References

\[ e_1 = \begin{align*}
\text{let } x_1 &= \text{new } 0 \text{ in} \\
\text{let } f_1 &= \lambda z : \text{unit}. \ x_1 := !x_1 + 1; \ !x_1 \text{ in} \\
\text{new } f_1 \\
\downarrow \\
l_{f_1}
\end{align*} \]

\[ e_2 = \begin{align*}
\text{let } x_2 &= \text{new } 0 \text{ in} \\
\text{let } f_2 &= \lambda z : \text{unit}. \ x_2 := !x_2 - 1; -(!x_2) \text{ in} \\
\text{new } f_2 \\
\downarrow \\
l_{f_2}
\end{align*} \]

\[ S = \{(s_1, s_2) \mid s_1(l_{f_1}) = s_2(l_{f_2})\} \]

store relation
2. Storing Functions in References

\[ e_1 = \begin{align*}
& \text{let } x_1 = \text{new } 0 \text{ in} \\
& \text{let } f_1 = \lambda z : \text{unit}. \ x_1 := !x_1 + 1; \ !x_1 \text{ in} \\
& \text{new } f_1 \\
\downarrow \\
& l_{f_1}
\end{align*} \]

\[ e_2 = \begin{align*}
& \text{let } x_2 = \text{new } 0 \text{ in} \\
& \text{let } f_2 = \lambda z : \text{unit}. \ x_2 := !x_2 - 1; \ -(!x_2) \text{ in} \\
& \text{new } f_2 \\
\downarrow \\
& l_{f_2}
\end{align*} \]

\[ S = \left\{ (s_1, s_2) \mid s_1(l_{f_1}) = s_2(l_{f_2}) \right\} \]

store relation

wrong!
\[ e_1 \quad = \quad \text{let } x_1 = \text{new } 0 \text{ in} \\
\quad \quad \text{let } f_1 = \lambda z : \text{unit}. \quad x_1 := !x_1 + 1; \quad !x_1 \text{ in} \\
\quad \quad \text{new } f_1 \\
\quad \quad \downarrow \quad l_{f_1} \\
\]

\[ e_2 \quad = \quad \text{let } x_2 = \text{new } 0 \text{ in} \\
\quad \quad \text{let } f_2 = \lambda z : \text{unit}. \quad x_2 := !x_2 - 1; -(!x_2) \text{ in} \\
\quad \quad \text{new } f_2 \\
\quad \quad \downarrow \quad l_{f_2} \\
\]

\[ S \quad = \quad \{(s_1, s_2) \mid s_1(l_{f_1}) \sim^{k,W} s_2(l_{f_2}) : \text{unit } \rightarrow \text{int}\} \]
2. Storing Functions in References

\[ e_1 = \begin{align*}
    & \text{let } x_1 = \text{new } 0 \text{ in} \\
    & \text{let } f_1 = \lambda z : \text{unit. } x_1 := !x_1 + 1; !x_1 \text{ in} \\
    & \text{new } f_1 \\
    \Downarrow & \quad l_{f_1}
\end{align*} \]

\[ e_2 = \begin{align*}
    & \text{let } x_2 = \text{new } 0 \text{ in} \\
    & \text{let } f_2 = \lambda z : \text{unit. } x_2 := !x_2 - 1; -(!x_2) \text{ in} \\
    & \text{new } f_2 \\
    \Downarrow & \quad l_{f_2}
\end{align*} \]

\[ S = \{(k, W, s_1, s_2) \mid s_1(l_{f_1}) \sim^{k, W} s_2(l_{f_2}) : \text{unit} \rightarrow \text{int}\} \]
2. Storing Functions in References

\[ e_1 = \text{let } x_1 = \text{new 0 in} \]
\[ \quad \text{let } f_1 = \lambda z: \text{unit. } x_1 := !x_1 + 1; !x_1 \text{ in} \]
\[ \quad \text{new } f_1 \]
\[ \Downarrow \quad l_{f_1} \]

\[ e_2 = \text{let } x_2 = \text{new 0 in} \]
\[ \quad \text{let } f_2 = \lambda z: \text{unit. } x_2 := !x_2 - 1; -(!x_2) \text{ in} \]
\[ \quad \text{new } f_2 \]
\[ \Downarrow \quad l_{f_2} \]

\[ S = \{ (k, W, s_1, s_2) \mid s_1(l_{f_1}) \sim^{k, W} s_2(l_{f_2}) : \text{unit} \rightarrow \text{int} \} \]

- Worlds contain store relations
- Store relations contain worlds
2. Storing Functions in References

\[ e_1 = \text{let } x_1 = \text{new } 0 \text{ in} \]
\[ \quad \text{let } f_1 = \lambda z : \text{unit}. \ x_1 := !x_1 + 1; \ !x_1 \text{ in} \]
\[ \quad \text{new } f_1 \]
\[ \Downarrow \ l_{f_1} \]

\[ e_2 = \text{let } x_2 = \text{new } 0 \text{ in} \]
\[ \quad \text{let } f_2 = \lambda z : \text{unit}. \ x_2 := !x_2 - 1; \ -(!x_2) \text{ in} \]
\[ \quad \text{new } f_2 \]
\[ \Downarrow \ l_{f_2} \]

\[ S = \{ (k, W, s_1, s_2) \mid s_1(l_{f_1}) \sim^{k,W} s_2(l_{f_2}) : \text{unit} \to \text{int} \} \]

- Worlds contain store relations
- Store relations contain worlds

Circular!
2. Storing Functions in References

\[ e_1 = \text{let } x_1 = \text{new } 0 \text{ in} \]
\[ \quad \text{let } f_1 = \lambda z : \text{unit. } x_1 := \text{!}x_1 + 1; \text{!}x_1 \text{ in} \]
\[ \quad \text{new } f_1 \]

\[ e_2 = \text{let } x_2 = \text{new } 0 \text{ in} \]
\[ \quad \text{let } f_2 = \lambda z : \text{unit. } x_2 := \text{!}x_2 - 1; -(!x_2) \text{ in} \]
\[ \quad \text{new } f_2 \]

\[ S = \{(k, W, s_1, s_2) \mid s_1(l_{f_1}) \sim_{k-1, [W]^{k-1}} s_2(l_{f_2}) : \text{unit} \rightarrow \text{int}\} \]
\[ e_1 = \text{let } x_1 = \text{new 0 in} \]
\[ \lambda z : \text{unit. } x_1 := !x_1 + 1; !x_1 \]

\[ e_2 = \text{let } x_2 = \text{new 0 in} \]
\[ \text{let } y_2 = \text{new 0 in} \]
\[ \lambda z : \text{unit. } x_2 := !x_2 + 1; y_2 := !y_2 + 1; ( !x_2 + !y_2 ) / 2 \]
1’. Data Abstraction via Local State

\[ e_1 = \begin{array}{l}
\text{let } x_1 = \text{new } 0 \text{ in} \\
\lambda z : \text{unit. } x_1 := !x_1 + 1; \ !x_1 \\
\downarrow \lambda z : \text{unit. } l_{x_1} := !l_{x_1} + 1; \ !l_{x_1}
\end{array} \]

\[ e_2 = \begin{array}{l}
\text{let } x_2 = \text{new } 0 \text{ in} \\
\text{let } y_2 = \text{new } 0 \text{ in} \\
\lambda z : \text{unit. } x_2 := !x_2 + 1; \ y_2 := !y_2 + 1; \ (\!x_2 + \!y_2)/2 \\
\downarrow \lambda z : \text{unit. } l_{x_2} := !l_{x_2} + 1; \ l_{y_2} := !l_{y_2} + 1; \ (\!l_{x_2} + \!l_{y_2})/2
\end{array} \]
1’. Data Abstraction via Local State

\[ e_1 = \text{let } x_1 = \text{new } 0 \text{ in } \lambda z : \text{unit. } x_1 := !x_1 + 1; !x_1 \]

\[ \Downarrow \lambda z : \text{unit. } l_{x_1} := !l_{x_1} + 1; !l_{x_1} \]

\[ e_2 = \text{let } x_2 = \text{new } 0 \text{ in } \text{let } y_2 = \text{new } 0 \text{ in } \lambda z : \text{unit. } x_2 := !x_2 + 1; y_2 := !y_2 + 1; ( !x_2 + !y_2 )/2 \]

\[ \Downarrow \lambda z : \text{unit. } l_{x_2} := !l_{x_2} + 1; l_{y_2} := !l_{y_2} + 1; ( !l_{x_2} + !l_{y_2} )/2 \]
1’. Data Abstraction via Local State

\[ e_1 = \text{let } x_1 = \text{new } 0 \text{ in } \\
\lambda z : \text{unit. } x_1 := !x_1 + 1; !x_1 \]

\[ \Downarrow \quad \lambda z : \text{unit. } l_{x_1} := !l_{x_1} + 1; !l_{x_1} \]

\[ e_2 = \text{let } x_2 = \text{new } 0 \text{ in } \\
\text{let } y_2 = \text{new } 0 \text{ in } \\
\lambda z : \text{unit. } x_2 := !x_2 + 1; y_2 := !y_2 + 1; (!x_2 + !y_2)/2 \]

\[ \Downarrow \quad \lambda z : \text{unit. } l_{x_2} := !l_{x_2} + 1; l_{y_2} := !l_{y_2} + 1; (!l_{x_2} + !l_{y_2})/2 \]

\[ S = \{ (s_1, s_2) \mid s_1(l_{x_1}) = (s_2(l_{x_2}) + s_2(l_{y_2}))/2 \} \]

store relation
3. Data Abstraction via Local State + ∃

Name = ∃α.⟨gen : unit → α, chk : α → bool⟩

\[ e_1 = \begin{align*}
& \text{let } x = \text{new 0 in} \\
& \text{pack int, } \langle \text{gen = } \lambda z : \text{unit. (}x := !x + 1; !x\),} \\
& \text{chk = } \lambda z : \text{int. (}z \leq !x\rangle \text{ as Name} 
\end{align*} \]
3. Data Abstraction via Local State + $\exists$

Name $\equiv \exists \alpha. \langle \text{gen} : \text{unit} \to \alpha, \text{chk} : \alpha \to \text{bool} \rangle$

$$e_1 = \text{let } x = \text{new } 0 \text{ in }$$
$$\text{pack int, } \langle \text{gen} = \lambda z : \text{unit. } (x := !x + 1; !x),$$
$$\text{chk} = \lambda z : \text{int. } (z \leq !x) \rangle \text{ as Name}$$

$$e_2 = \text{let } x = \text{new } 0 \text{ in }$$
$$\text{pack int, } \langle \text{gen} = \lambda z : \text{unit. } (x := !x + 1; !x),$$
$$\text{chk} = \lambda z : \text{int. true} \rangle \text{ as Name}$$
3. Data Abstraction via Local State + ∃

Name = ∃α.〈gen : unit → α, chk : α → bool〉

\[ e_1 = \text{let } x = \text{new } 0 \text{ in } \]
pack int, 〈gen = \lambda z : \text{unit}. (x := !x + 1; !x),
chk = \lambda z : \text{int}. (z \leq !x)〉 as Name

\[ e_2 = \text{let } x = \text{new } 0 \text{ in } \]
pack int, 〈gen = \lambda z : \text{unit}. (x := !x + 1; !x),
chk = \lambda z : \text{int}. \text{true}〉 as Name

Intuitively, we want \( R_\alpha = \{(1, 1), \ldots, (n, n)\} \) where \( n \) is the current value of \( !x \)
3. Data Abstraction via Local State + \exists

Name = \exists \alpha. \langle \text{gen} : \text{unit} \rightarrow \alpha, \text{chk} : \alpha \rightarrow \text{bool} \rangle

\begin{align*}
e_1 & = \text{let } x = \text{new } 0 \text{ in} \\
& \quad \text{pack int, } \langle \text{gen} = \lambda z : \text{unit}. (x := !x + 1; !x), \\
& \quad \quad \text{chk} = \lambda z : \text{int}. (z \leq !x) \rangle \text{ as Name}
\end{align*}

\begin{align*}
e_2 & = \text{let } x = \text{new } 0 \text{ in} \\
& \quad \text{pack int, } \langle \text{gen} = \lambda z : \text{unit}. (x := !x + 1; !x), \\
& \quad \quad \text{chk} = \lambda z : \text{int}. \text{true} \rangle \text{ as Name}
\end{align*}

Intuitively, we want $R_\alpha = \{(1, 1), \ldots, (n, n)\}$ where $n$ is the current value of !x

\textbf{Problem:} How do we express such a dynamic, \textit{state-dependent} representation of $\alpha$?
3. Data Abstraction via Local State + ∃

Name = ∃α.⟨gen : unit → α, chk : α → bool⟩

\[ e_1 = \text{let } x = \text{new 0 in pack int, } \langle \text{gen} = \lambda z : \text{unit.} (x := !x + 1; !x), \text{chk} = \lambda z : \text{int.} (z \leq !x) \rangle \text{ as Name} \]

\[ e_2 = \text{let } x = \text{new 0 in pack int, } \langle \text{gen} = \lambda z : \text{unit.} (x := !x + 1; !x), \text{chk} = \lambda z : \text{int.} \text{true} \rangle \text{ as Name} \]

Intuitively, we want \( R_α = \{(1, 1), \ldots, (n, n)\} \) where \( n \) is the current value of \(!x\)

**Solution:** Permit the property about a piece of local state to evolve over time
# Logical Relations Survey (1967-2009)

<table>
<thead>
<tr>
<th></th>
<th>$\forall$</th>
<th>$\exists$</th>
<th>$\mu$</th>
<th>ref</th>
<th>Simple Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plotkin’73, Statman’85</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Reynolds’83, Mitchell’86</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Pitts-Stark’93,’98 (ref int)</td>
<td></td>
<td>✓ –</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Pitts’98,’00 (recursive functions)</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓ –</td>
<td></td>
</tr>
<tr>
<td>Appel-McAllester’01</td>
<td>✓ –</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Benton-Leperchey’05 (ref...ref int)</td>
<td>✓</td>
<td>✓ –</td>
<td></td>
<td></td>
<td>✓ ✓</td>
</tr>
<tr>
<td>Ahmed’06</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Bohr-Birkedal’06</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓ ✓</td>
</tr>
<tr>
<td>Ahmed-Dreyer-Rossberg’09</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Next...

• Applications

• Step-indexing: hiding the steps

• Open problems, future directions
Applications: Unary Step-Indexed LR

Type Safety

- Foundational Proof-Carrying Code (FPCC) [Appel et al.]
  - recursive types [Appel-McAllester, TOPLAS’01]
  - ... + mutable refs + impredicative $\exists \forall$ [Ahmed, PhD.’04, Chp 2,3; region-based lang. Chp 7]
  - model of LTAL, target lang of ML compiler: Semantic models of Typed Assembly Languages [Ahmed et al., TOPLAS’10]
  - Recommended reading: Section 7 of the TOPLAS’10 paper contains a detailed history of the FPCC project and step-indexed logical relations
Applications: Unary Step-Indexed LR

Type Safety

• L3: Linear Lang. with Locations [Ahmed-Fluet-Morrisett, TLCA’05]
  - alias types revisited, first-class capabilities (linear/unrestricted)

• Substructural State [Ahmed-Fluet-Morrisett, ICFP’05]
  - interaction of linear, affine, relevant, unrestricted references

• Imperative Object Calculus [Hritcu-Schwinghammer, FOOL’08, LMCS]
Applications: Unary Step-Indexed LR

Soundness of Concurrent Separation Logic w.r.t Concurrent C minor operational semantics

• modular semantics to adapt Leroy’s compiler correctness proofs to concurrent setting [Hobor-Appel-Zappa-Nardelli, ESOP’08]

• Oracle Semantics for Concurrent Separation Logic
Applications: Binary Step-Indexed LR

- **Observational Equivalence**
  - System F + recursive types [Ahmed, ESOP’06]; also see *Extended Version with detailed proofs*.
  - ... + mutable references [Ahmed-Dreyer-Rossberg, POPL’09]
  - + first-order store + control [Dreyer-Neis-Birkedal, ICFP’10]
Applications: Binary Step-Indexed LR

- Imperative Self-Adjusting Computation
  - [Acar-Ahmed-Blume, POPL’08]

- Untyped language with dynamically allocated modifiable refs
  - untyped step-indexed LR
Imperative Self-Adjusting Computation

[Acar-Ahmed-Blume, POPL’08]

P(\(v_{\text{orig}}\))  \[\xrightarrow{}\]  \(v_0\)

P(\(v_{\text{changed}}\))  \[\xrightarrow{}\]  \(v_1\)

Idea: update results by reusing those parts of previous computation that are unaffected by the changes.
Imperative Self-Adjusting Computation

[Acar-Ahmed-Blume, POPL’08]

Idea:
update results by reusing those parts of previous computation that are unaffected by the changes
Overview:

- Store all data that may change in modifiable references
- Record a history of all operations on modifials in a trace
- When inputs change, we can selectively re-execute only those parts that depend on the changed data
  - change propagation
- “Imperative”: modifiable refs can be updated
Equivalence of Evaluation Strategies

\[ P(V_{\text{orig}}) \quad \overset{\text{Equivalent}}{\implies} \quad P(V_{\text{changed}}) \]

\[ P(V_0) \quad \overset{\text{Equivalent}}{\implies} \quad P(V_1) \]
Equivalence of Evaluation Strategies

\[ P(v_{\text{orig}}) \xrightarrow{\downarrow} P(v_{\text{changed}}) \xrightarrow{\downarrow} P(v_{\text{changed}}) \xrightarrow{\text{equivalent}} P(v_{\text{changed}}) \xrightarrow{\text{v1'}} \]

\[ v_0 \xrightarrow{\downarrow} v_1 \xrightarrow{\text{equivalent}} v_1' \]
Applications: Binary Step-Indexed LR

- Secure Multi-Language Interoperability (ML / Scheme)
  - Parametricity through run-time sealing [Matthews-Ahmed, ESOP’08] and [Ahmed-Kuper-Matthews]
Secure Multi-Language Interoperability

[Matthews-Ahmed, ESOP’08] and [Ahmed-Kuper-Matthews]

Information hiding:

- typed languages (e.g., ML) : via $\exists \alpha. \tau$
- untyped languages (e.g., Scheme) : via dynamic sealing

A multi-language system in which typed and untyped languages can interoperate ( $SM^T e$, $TMS e$ )

- Parametricity through run-time sealing:
  concrete representations hidden behind an abstract type in ML are hidden using dynamic sealing to avoid discovery by Scheme part of program
Applications: Binary Step-Indexed LR

- Secure Multi-Language Interoperability (ML / Scheme)
  - Parametricity through run-time sealing [Matthews-Ahmed, ESOP’08] and [Ahmed-Kuper-Matthews, 2010]
- Non-Parametric Parametricity
  - Parametricity in a non-parametric language via static sealing [Neis-Dreyer-Rossberg, ICFP’09]
Applications: Binary Step-Indexed LR

- Compiler correctness for components:
  - logical relation between source and target terms $s \sim t : S$
  - System F + recursive functions to SECD [Benton-Hur, ICFP’09]
  - ... + mutable refs [Hur-Dreyer, POPL’11]
  - Theorem: If $s : S$ compiles to $t$, then $s \sim t : S$

- Cross-language LRs: do not scale to multi-pass compilers
- Does not permit linking with code that cannot be written in source

- Recent work: Pilsner compiler [Neis et al., ICFP’15]
Applications: Binary Step-Indexed LR

• Correct component compilation that supports multi-language software:
  - To specify compiler correctness, define multi-language that supports interoperability between source and target
    \[\text{[Perconti-Ahmed, ESOP 2014]}\]
  - Theorem: If \( s : S \) compiles to \( t \), then \( s \approx_{ctx} ST(t) : S \)
  - Scales to multi-pass compilers
  - Allows linking with code that cannot be written in source

• Recommended paper on research program: Verified Compilers for a Multi-Language World \[\text{[Ahmed, SNAPL'15]}\]
Applications: **Binary Step-Indexed LR**

- **Differential Privacy Calculus**
  - Distance Makes the Types Grow Stronger
  - well-typedness guarantees privacy safety [Reed-Pierce, ICFP’10]
  - step-indexed logical relation used to prove “metric preservation” theorem
Applications: Binary Step-Indexed LR

- L.R. for Fine-grained Concurrent Data Structures
  - [Turon, Thamsborg, Ahmed, Birkedal, Dreyer, POPL 2013]
  - step-indexed logical relation for proving correctness (contextual refinement) of many subtle FCDs

\[ e_I \preceq e_S \]

- concurrent implementation
- sequential specification
- contextual refinement
  - every behavior of impl. is a possible behavior of its spec.
Next...

- Applications
- **Step-indexing: hiding the steps**
- Open problems, future directions
Ugly Side of Step-Indexing: the Steps!
Ugly Side of Step-Indexing: the Steps!

Step-index arithmetic pervades proofs:

- Tedious, error-prone, feels *ad-hoc*
- Want to develop clean, abstract, step-free proof principles
Ugly Side of Step-Indexing: the Steps!

Step-index arithmetic pervades proofs:

- Tedious, error-prone, feels *ad-hoc*
- Want to develop clean, abstract, step-free proof principles

We might like to prove:

- **f1 and f2 are infinitely related** (i.e., related for any # of steps) iff for all v1 and v2 that are infinitely related, f1 v1 and f2 v2 are, too.
Ugly Side of Step-Indexing: the Steps!

Step-index arithmetic pervades proofs:

- Tedious, error-prone, feels *ad-hoc*
- Want to develop clean, abstract, step-free proof principles

We might like to prove:

- **f1 and f2 are infinitely related** (i.e., related for any # of steps) *iff* for all v1 and v2 that are infinitely related, f1 v1 and f2 v2 are, too.

Unfortunately, that is false.

- In fact, **f1 and f2 are infinitely related** *iff*, for any step level n, for all v1 and v2 that are related for n steps, f1 v1 and f2 v2 are, too.
Hiding the Steps: Relational Logics

Develop relational modal logic for expressing step-indexed LR without mentioning steps

- System F + recursive types: [Dreyer-Ahmed-Birkedal, LICS’09]
- Start with Plotkin-Abadi logic for relational parametricity [TLCA’93]; extend it with recursively defined relations
- To make sense of circularity, introduce “later” operator \( \triangleright \) from [Appel et al., POPL’07], in turn adapted from Gödel-Löb logic
  - Löb rule: \( (\triangleright A \supset A) \supset A \)
- Using logic, define a step-free logical relation for reasoning about program equivalence
- Show step-free LR is sound w.r.t. contextual equivalence, by defining suitable “step-indexed” model of the logic
- ... + mutable references: [Dreyer-Neis-Rossberg-Birkedal, POPL’10]
Hiding the Steps: Relational Logics

Develop *relational modal logic* for expressing step-indexed LR without mentioning steps

- Using logic, define a *step-free logical relation* for reasoning about program equivalence

- Makes proof method *easier to use*
Hiding the Steps: Indirection Theory

- Step-indexing machinery gets quite tricky in languages with state (e.g., circularity between worlds & store relations)

- **Indirection theory** is a framework that makes it easier to build such models
  - makes world-stratification conceptually simpler, and makes such models easier to mechanize
  - [Hobor-Dockins-Appel, POPL’10]
Next...

• Applications

• Step-indexing: hiding the steps

• Open problems, future directions
1. Other Language Features...

**Dependent types**

- depends on the dependent type theory!
- Coq / ECC / Hoare Type Theory (HTT): higher-order logic
  - would like an operational model of propositional equality; how to deal with impredicativity of h.o.l. (no notion of consuming steps at logical level)
- parametricity for HTT: (extends Coq with type \{P\}x:A\{Q\})
  - invariants about state are part of types; will be able to prove “free theorems” in presence of state!
2. Other Applications...

• Secure Compilation
  - Fully-Abstract Compilation: equivalence-preserving and reflecting
Equivalence-Preserving Compilation

- Semantics-preserving compilation

\[ P_S \xrightarrow{\text{compile}} P_T \]

\[ V_S \xrightarrow{\text{equivalent}} V_T \]

1. ...

2. "equivalent"
Equivalence-Preserving Compilation

• Semantics-preserving compilation

\[ e_S \overset{\text{compile}}{\rightarrow} e_T \]

\[ v_S \overset{\text{“equivalent”}}{\sim} v_T \]

• Equivalence-preserving compilation

\[
\text{If } e_{S_1} \overset{\text{compile}}{\rightarrow} e_{T_1} \text{ and } e_{S_2} \overset{\text{compile}}{\rightarrow} e_{T_2} \\
\quad \text{then } e_{S_1} \overset{\text{ctx}}{\sim}_{S} e_{S_2} \implies e_{T_1} \overset{\text{ctx}}{\sim}_{T} e_{T_2}
\]
Why Should We Care?

**Security issue**: If compilation is not equivalence-preserving then there exist contexts (i.e., attackers!) at target that can distinguish program fragments that cannot be distinguished by source contexts

- **C# to Microsoft .NET IL [Kennedy’06]**: compiler’s failure to preserve equivalence can lead to security exploits

- Programmers think about behavior of their programs by considering only source-level contexts (i.e., other components written in source language)

- **ADTs**: replacing one implementation with another that’s “functionally” equivalent should not lead to problems
Secure Compilation
Secure Compilation

Typed Closure Conversion is Equivalence-Preserving

- Closure conversion: collect free variables of a function in a closure environment & pass environment as an additional argument to the function; (typed c.c. [Minamide+'96], [Morrisett+'98]

- System F + ∃ + recursive types [Ahmed-Blume, ICFP’08]

- Step-indexed logical relations, sound+complete w.r.t. ctx-equiv
Secure Compilation

Typed Closure Conversion is Equivalence-Preserving

• Closure conversion: collect free variables of a function in a closure environment & pass environment as an additional argument to the function; (typed c.c. [Minamide+’96], [Morrisett+’98]
• System F + ∃ + recursive types [Ahmed-Blume, ICFP’08]
• Step-indexed logical relations, sound+complete w.r.t. ctx-equiv
Secure Compilation

Typed Closure Conversion is Equivalence-Preserving

- Closure conversion: collect free variables of a function in a closure environment & pass environment as an additional argument to the function; (typed c.c. [Minamide+’96], [Morrisett+’98]

- System F + ∃ + recursive types [Ahmed-Blume, ICFP’08]

- Step-indexed logical relations, sound+complete w.r.t. ctx-equiv

An Equivalence-Preserving CPS Translation via Multi-Language Semantics [Ahmed-Blume, ICFP’11]

- CPS: names all intermediate computations and makes control flow explicit

- Works for target lang. more expressive than source
Secure Compilation
Secure Compilation

Noninterference for Free

- Dependency Core Calculus (DCC) \([\text{Abadi}^+’99]\) can encode secure information flow
- Noninterference-preserving translation from DCC to System \(F_\omega\) \([\text{Bowman-Ahmed, ICFP’15}]\)
- Includes:
  - “open” logical relation for \(F_\omega\) (based on \([\text{Zhao}+, \text{APLAS’10}]\))
  - cross-language logical relation between DCC and \(F_\omega\)
  - unary logical relation to show that back-translation from \(F_\omega\) to DCC is well-founded
Conclusions...

- **Logical relations**
  - formalize intuitions about abstraction, modularity, information hiding
  - beautiful, elegant, and powerful technique

- Many cool, challenging problems demand reasoning about relational properties

- We are in an exciting **golden age of logical relations**: recent developments enable reasoning about complex languages (mutable memory, concurrency, etc.), compiler correctness, secure compilation, ...
Conclusions

Step-indexed logical relations

• Scale well to linguistic features found in real languages
  - mutable references, recursive types, interfaces, generics

• Elementary (no domain/category theory, just sets & relations)

• Many important applications
  - same intuition works well in a wide variety of contexts; allows us to focus on interesting aspects of problem at hand

• Critical tool for proving reliability of programming languages and compilers
Questions?