Logical Relations

Amal Ahmed
Northeastern University

OPLSS Lecture 6, July 30, 2013
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 math
• Denotational vs. operational

Not easy to do mechanized proofs
• Proof mechanization is important for practical applications
## Logical Relations Survey (1967-2009)

<table>
<thead>
<tr>
<th></th>
<th>∀</th>
<th>∃</th>
<th>μ</th>
<th>ref</th>
<th>Simple Math / Easy Mech</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>x</td>
</tr>
<tr>
<td>Reynolds’83, Mitchell’86</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>x</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>Birkedal-Harper’99, Crary-Harper’07</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>x</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>x</td>
<td>xx</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>xx</td>
</tr>
<tr>
<td>Ahmed-Dreyer-Rossberg’09</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Logical Relations Survey (1967-2009)

<table>
<thead>
<tr>
<th></th>
<th>∀</th>
<th>∃</th>
<th>μ</th>
<th>ref</th>
<th>Simple Math / Easy Mech</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</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>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>
</tbody>
</table>
# Logical Relations Survey (1967-2009)

<table>
<thead>
<tr>
<th></th>
<th>∀</th>
<th>∃</th>
<th>μ</th>
<th>ref</th>
<th>Simple Math / Easy Mech</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</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>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>XX</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>XX</td>
</tr>
<tr>
<td>Ahmed-Dreyer-Rossberg’09</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
## Logical Relations Survey (1967-2009)

<table>
<thead>
<tr>
<th>Authors</th>
<th>∀</th>
<th>∃</th>
<th>μ</th>
<th>ref</th>
<th>Simple Math / Easy Mech</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

\[ l \mid \text{new } e \mid e_1 := e_2 \mid !e \]

\[ s, \text{new } v \quad \mapsto \quad s[l \mapsto v], l \quad \text{where } l \text{ fresh} \]

\[ s, !l \quad \mapsto \quad s, v \quad \text{where } s(l) = v \]

\[ s, l := v \quad \mapsto \quad s[l \mapsto v], () \quad \text{where } l \in \text{dom}(s) \]
Problems in Presence of References

1. Data abstraction via local state

2. Storing functions in references

3. Interaction of ∃ and references

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

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

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

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

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

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

\[ \Downarrow \quad \lambda z : \text{unit}. \ l_{x_2} := !l_{x_2} - 1; \ -(!l_{x_2}) \]
1. Data Abstraction via Local State

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

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

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

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

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

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

\[ S = \{ (s_1, s_2) | s_1(l_{x_1}) = -s_2(l_{x_2}) \} \]

store relation
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

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

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

\[ S = \{ (s_1, s_2) | s_1(l_{f_1}) = s_2(l_{f_2}) \} \]
2. Storing Functions in References

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

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

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

\text{store relation}

\text{wrong!}
2. Storing Functions in References

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

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

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

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

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

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

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

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

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

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

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

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

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

- Worlds contain store relations
- Store relations contain worlds

Circular!
2. Storing Functions in References

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

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

\[ S = \{(k, W, s_1, s_2) | s_1(l_{f_1}) \sim^{k-1, [W]_{k-1}} s_2(l_{f_2}) : \text{unit } \rightarrow \text{ int}\} \]
1’. Data Abstraction via Local State

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

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

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

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

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

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

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

store relation
3. Data Abstraction via Local State + \( \exists \)

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

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

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

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

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

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

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

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

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

**Problem:** How do we express such a dynamic, state-dependent representation of \( \alpha \)?
3. Data Abstraction via Local State + ∃

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

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

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

Intuitively, we want \( R_\alpha = \{(1, 1), \ldots, (n, n)\} \) 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>∀</th>
<th>∃</th>
<th>µ</th>
<th>ref</th>
<th>Simple Math / Easy Mech</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
- Ugly side of step-indexing (and how to fix it)
- 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 (instead of higher-order) and control [Dreyer-Neis-Birkedal, ICFP’10]
Applications: Binary Step-Indexed LR

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

[Acar-Ahmed-Blume, POPL’08]

\[ P(\nu_{\text{orig}}) \quad \text{and} \quad P(\nu_{\text{changed}}) \]

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

\[ P(v_{\text{orig}}) \quad P(v_{\text{changed}}) \]

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

Overview:

• Store all data that may change in modifiable references
• Record a history of all operations on modifiables 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

\[ P(\mathbf{v}_{\text{orig}}) \]

\[ \downarrow \]

\[ \downarrow \]

\[ \downarrow \]

\[ \downarrow \]

\[ \mathbf{v}_0 \]

\[ P(\mathbf{v}_{\text{changed}}) \]

\[ \downarrow \]

\[ \downarrow \]

\[ \downarrow \]

\[ \downarrow \]

\[ \mathbf{v}_1 \]
Equivalence of Evaluation Strategies

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

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

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

\[ v_0 \]

\[ v_1 \]

equivalent

\[ v_1' \]
Imperative Self-Adjusting Computation

Untyped language with dynamically allocated modifiable refs
• *untyped step-indexed LR*
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 ( \( \text{SM}^T \ e, \text{TMS} \ e \) )

• 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 \cite{Matthews-Ahmed, ESOP’08} and \cite{Ahmed-Kuper-Matthews, 2010}

- Non-Parametric Parametricity
  - Parametricity in a non-parametric language via static sealing \cite{Neis-Dreyer-Rossberg, ICFP’09}
Applications: Binary Step-Indexed LR

• Compiler Correctness for “open” programs:
  - 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]
  - Currently does not scale to multi-pass compilers
  - Does not permit linking with code that cannot be written in source

• Theorem: If $s : S$ compiles to $t$, then $s \sim t : S$
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 \hspace{2cm} contextual refinement \hspace{2cm} sequential specification

every behavior of impl. is a possible behavior of its spec.
Next...

• Applications

• Ugly side of step-indexing (and how to fix it)

• 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 \textit{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 $\triangleright A$ 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
- Ugly side of step-indexing (and how to fix it)
- Open problems, future directions
1. Connections with...

How exactly does step-indexing relate to:

• **Denotational models**
  - understanding such connections could help us translate insights
  - recent work by Birkedal et al. on ultra-metric spaces

• **Bisimulation**
  - steps provide an induction metric, while bisimulation relies on coinduction
  - The marriage of Bisimulations and Kripke Logical Relations
    [Hur-Dreyer-Neis-Vafeiadis, POPL’12]
    *(warning: problem with eta rule! See paper: Parametric Bisimulations)*
2. Other Language Features...

Exceptions (should be easy)

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!
3. Other Applications...

- Equivalence-Preserving Compilation
  - Fully-Abstract Compilation
Equivalence-Preserving Compilation

• Semantics-preserving compilation

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

“equivalent”

\[ V_S \xrightarrow{①} V_T \]
Equivalence-Preserving Compilation

- **Semantics-preserving compilation**

\[ e_S \xrightarrow{\text{compile}} e_T \]

\[ \forall_S \ "\text{equivalent}\" \]

- **Equivalence-preserving compilation**

If \( e_{S1} \xrightarrow{\text{compile}} e_{T1} \) and \( e_{S2} \xrightarrow{\text{compile}} e_{T2} \)

then \( e_{S1} \approx^{ctx}_S e_{S2} \implies e_{T1} \approx^{ctx}_T e_{T2} \)
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
Equivalence-Preserving Compilation
Equivalence-Preserving 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
Equivalence-Preserving 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
Equivalence-Preserving 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
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, security-preserving 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)
- Easy to mechanize proofs
- 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?