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, ∃, ∀)

Complicated, hard to extend

- Denotational vs. operational

Logical Relations (highlights,1967-2009)

	∀ ∃	μ	ref	Simple Model
Tait'67, Girard'72				
Plotkin'73, Statman'85				✗
Reynolds'83, Mitchell'86	✓			✗
Pitts-Stark'93,'98 (ref int)			✓–	✓
Pitts'98,'00 (recursive functions)	✓			✓–
Birkedal-Harper'99, Crary-Harper'07	✓	✓		✗
Appel-McAllester'01		✓–		✓
Benton-Leperchey'05 (ref...ref int)		✓	✓–	✗ ✗
Ahmed'06	✓	✓		✓
Bohr-Birkedal'06		✓	✓	✗ ✗
Ahmed-Dreyer-Rossberg'09	✓	✓	✓	✓

Logical Relations (highlights,1967-2009)

	∀ ∃	μ	ref	Simple Model
Plotkin'73, Statman'85				✗
Reynolds'83, Mitchell'86	✓			✗
Pitts-Stark'93,'98 (ref int)			✓−	✓
Pitts'98,'00 (recursive functions)	✓			✓−
Birkedal-Harper'99, Crary-Harper'07	✓	✓		✗
Appel-McAllester'01		✓−		✓
Benton-Leperchey'05 (ref...ref int)		✓	✓−	✗ ✗
Ahmed'06	✓	✓		✓
Bohr-Birkedal'06		✓	✓	✗ ✗
Ahmed-Dreyer-Rossberg'09	✓	✓	✓	✓

Logical Relations (highlights,1967-2009)

	∀ ∃	μ	ref	Simple Model
Plotkin'73, Statman'85				✗
Reynolds'83, Mitchell'86	✓			✗
Pitts-Stark'93,'98 (ref int)			✓−	✓
Pitts'98,'00 (recursive functions)	✓			✓−
Birkedal-Harper'99, Crary-Harper'07	✓	✓		✗
Appel-McAllester'01		✓−		✓
Benton-Leperchey'05 (ref...ref int)		✓	✓−	✗ ✗
Ahmed'06	✓	✓		✓
Bohr-Birkedal'06		✓	✓	✗ ✗
Ahmed-Dreyer-Rossberg'09	✓	✓	✓	✓

Logical Relations (highlights,1967-2009)

	∀ ∃	μ	ref	Simple Model
Plotkin'73, Statman'85				✗
Reynolds'83, Mitchell'86	✓			✗
Pitts-Stark'93,'98 (ref int)			✓−	✓
Pitts'98,'00 (recursive functions)	✓			✓−
Birkedal-Harper'99, Crary-Harper'07	✓	✓		✗
Appel-McAllester'01		✓−		✓
Benton-Leperchey'05 (ref...ref int)		✓	✓−	✗ ✗
Ahmed'06	✓	✓		✓
Bohr-Birkedal'06		✓	✓	✗ ✗
Ahmed-Dreyer-Rossberg'09	✓	✓	✓	✓

Mutable References

Reference Types ref τ

Syntax $l \mid \texttt{new}\, e \mid e_1 \texttt{:=} e_2 \mid \texttt{!}e$

$$
\begin{array}{rcll}
s, \texttt{new}\, v & \longmapsto & s[l \mapsto v], l & \text{where } l \text{ fresh} \\
s, \texttt{!}l & \longmapsto & s, v & \text{where } s(l) = v \\
s, l \texttt{:=} v & \longmapsto & s[l \mapsto v], () & \text{where } l \in \mathrm{dom}(s)
\end{array}
$$

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

$$
\begin{aligned}
e_1 &= \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
&\quad \lambda z : \mathsf{unit}.\;\; x_1 \,\texttt{:=}\, \texttt{!}x_1 + 1;\;\; \texttt{!}x_1
\end{aligned}
$$

$$
\begin{aligned}
e_2 &= \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
&\quad \lambda z : \mathsf{unit}.\;\; x_2 \,\texttt{:=}\, \texttt{!}x_2 - 1; -(\texttt{!}x_2)
\end{aligned}
$$

1. Data Abstraction via Local State

$$
\begin{array}{rcl}
e_1 & = & \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
 & & \lambda z : \mathsf{unit}.\;\; x_1 \;\texttt{:=}\; \texttt{!}x_1 + 1;\; \texttt{!}x_1 \\
 & \Downarrow & \lambda z : \mathsf{unit}.\;\; l_{x1} \;\texttt{:=}\; \texttt{!}l_{x1} + 1;\; \texttt{!}l_{x1}
\end{array}
$$

$$
\begin{array}{rcl}
e_2 & = & \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
 & & \lambda z : \mathsf{unit}.\;\; x_2 \;\texttt{:=}\; \texttt{!}x_2 - 1; -(\texttt{!}x_2) \\
 & \Downarrow & \lambda z : \mathsf{unit}.\;\; l_{x2} \;\texttt{:=}\; \texttt{!}l_{x2} - 1;\; -(\texttt{!}l_{x2})
\end{array}
$$

1. Data Abstraction via Local State

$$e_1 \quad = \quad \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in}$$
$$\lambda z : \mathsf{unit}.\ \ x_1 \texttt{ := } \texttt{!}x_1 + 1;\ \texttt{!}x_1$$

$$\Downarrow \quad \lambda z : \mathsf{unit}.\ \ l_{x1} \texttt{ := } \texttt{!}l_{x1} + 1;\ \texttt{!}l_{x1}$$

$$e_2 \quad = \quad \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in}$$
$$\lambda z : \mathsf{unit}.\ \ x_2 \texttt{ := } \texttt{!}x_2 - 1; -(\texttt{!}x_2)$$

$$\Downarrow \quad \lambda z : \mathsf{unit}.\ \ l_{x2} \texttt{ := } \texttt{!}l_{x2} - 1;\ -(\texttt{!}l_{x2})$$

1. Data Abstraction via Local State

$$e_1 \quad = \quad \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in}$$
$$\lambda z : \mathsf{unit}.\ \ x_1 \mathrel{\texttt{:=}} \texttt{!}x_1 + 1;\ \texttt{!}x_1$$

$$\Downarrow \quad \lambda z : \mathsf{unit}.\ \ l_{x1} \mathrel{\texttt{:=}} \texttt{!}l_{x1} + 1;\ \texttt{!}l_{x1}$$

$$e_2 \quad = \quad \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in}$$
$$\lambda z : \mathsf{unit}.\ \ x_2 \mathrel{\texttt{:=}} \texttt{!}x_2 - 1; -(\texttt{!}x_2)$$

$$\Downarrow \quad \lambda z : \mathsf{unit}.\ \ l_{x2} \mathrel{\texttt{:=}} \texttt{!}l_{x2} - 1;\ -(\texttt{!}l_{x2})$$

$$S \quad = \quad \{\, (s_1, s_2) \mid s_1(l_{x_1}) \ = \ -\, s_2(l_{x2})\, \}$$

store relation

2. Storing Functions in References

$$
\begin{aligned}
e_1 \quad = \quad & \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
& \texttt{let}\, f_1 = \lambda z : \mathsf{unit}.\;\; x_1 \texttt{ := } \texttt{!}x_1 + 1;\; \texttt{!}x_1\, \texttt{in} \\
& \texttt{new}\, f_1
\end{aligned}
$$

$$
\begin{aligned}
e_2 \quad = \quad & \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
& \texttt{let}\, f_2 = \lambda z : \mathsf{unit}.\;\; x_2 \texttt{ := } \texttt{!}x_2 - 1; -(\texttt{!}x_2)\, \texttt{in} \\
& \texttt{new}\, f_2
\end{aligned}
$$

2. Storing Functions in References

$$
\begin{aligned}
e_1 \quad &= \quad \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
&\qquad \texttt{let}\, f_1 = \lambda z : \mathsf{unit}.\;\; x_1 \texttt{ := } !x_1 + 1;\; !x_1\, \texttt{in} \\
&\qquad \texttt{new}\, f_1 \\
&\Downarrow \quad l_{f_1}
\end{aligned}
$$

$$
\begin{aligned}
e_2 \quad &= \quad \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
&\qquad \texttt{let}\, f_2 = \lambda z : \mathsf{unit}.\;\; x_2 \texttt{ := } !x_2 - 1; -(!x_2)\, \texttt{in} \\
&\qquad \texttt{new}\, f_2 \\
&\Downarrow \quad l_{f_2}
\end{aligned}
$$

2. Storing Functions in References

$$
\begin{array}{lll}
e_1 & = & \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
 & & \texttt{let}\, f_1 = \lambda z : \mathsf{unit}.\;\; x_1 \texttt{ := } \texttt{!}x_1 + 1;\; \texttt{!}x_1\, \texttt{in} \\
 & & \texttt{new}\, f_1 \\
 & \Downarrow & l_{f_1}
\end{array}
$$

$$
\begin{array}{lll}
e_2 & = & \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
 & & \texttt{let}\, f_2 = \lambda z : \mathsf{unit}.\;\; x_2 \texttt{ := } \texttt{!}x_2 - 1; -(\texttt{!}x_2)\, \texttt{in} \\
 & & \texttt{new}\, f_2 \\
 & \Downarrow & l_{f_2}
\end{array}
$$

$$S \quad = \quad \{\, (s_1, s_2) \mid s_1(l_{f1}) = s_2(l_{f2})\, \}$$

store relation

2. Storing Functions in References

$$
\begin{array}{lcl}
e_1 & = & \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
 & & \texttt{let}\, f_1 = \lambda z : \mathsf{unit}.\ \ x_1 \,\texttt{:=}\, \texttt{!}x_1 + 1;\ \texttt{!}x_1\, \texttt{in} \\
 & & \texttt{new}\, f_1 \\
 & \Downarrow & l_{f_1}
\end{array}
$$

$$
\begin{array}{lcl}
e_2 & = & \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
 & & \texttt{let}\, f_2 = \lambda z : \mathsf{unit}.\ \ x_2 \,\texttt{:=}\, \texttt{!}x_2 - 1; -(\texttt{!}x_2)\, \texttt{in} \\
 & & \texttt{new}\, f_2 \\
 & \Downarrow & l_{f_2}
\end{array}
$$

$$
S \quad = \quad \{\, (s_1, s_2) \mid s_1(l_{f1}) = s_2(l_{f2}) \,\}
$$

store relation

$\boxed{\text{wrong!}}$

2. Storing Functions in References

$$
\begin{array}{lcl}
e_1 & = & \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
 & & \texttt{let}\, f_1 = \lambda z : \mathsf{unit}.\;\; x_1 \texttt{ := !} x_1 + 1;\; \texttt{!} x_1\, \texttt{in} \\
 & & \texttt{new}\, f_1 \\
 & \Downarrow & l_{f_1}
\end{array}
$$

$$
\begin{array}{lcl}
e_2 & = & \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
 & & \texttt{let}\, f_2 = \lambda z : \mathsf{unit}.\;\; x_2 \texttt{ := !} x_2 - 1; -(\texttt{!} x_2)\, \texttt{in} \\
 & & \texttt{new}\, f_2 \\
 & \Downarrow & l_{f_2}
\end{array}
$$

$$
S \quad = \quad \{\, (s_1, s_2) \mid s_1(l_{f1}) \sim^{k,W} s_2(l_{f2}) : \mathsf{unit} \rightarrow \mathsf{int} \,\}
$$

2. Storing Functions in References

$$\begin{aligned} e_1 &= \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\ &\quad\ \texttt{let}\, f_1 = \lambda z : \mathsf{unit}.\ x_1 \texttt{ := !} x_1 + 1;\ \texttt{!} x_1\, \texttt{in} \\ &\quad\ \texttt{new}\, f_1 \\ &\Downarrow\ l_{f_1} \end{aligned}$$

$$\begin{aligned} e_2 &= \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\ &\quad\ \texttt{let}\, f_2 = \lambda z : \mathsf{unit}.\ x_2 \texttt{ := !} x_2 - 1; -(\texttt{!} x_2)\, \texttt{in} \\ &\quad\ \texttt{new}\, f_2 \\ &\Downarrow\ l_{f_2} \end{aligned}$$

$$S = \{ (k, W, s_1, s_2) \mid s_1(l_{f1}) \sim^{k,W} s_2(l_{f2}) : \mathsf{unit} \longrightarrow \mathsf{int} \}$$

2. Storing Functions in References

$$
\begin{array}{rcl}
e_1 & = & \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
 & & \texttt{let}\, f_1 = \lambda z : \mathsf{unit}.\;\; x_1 \;\texttt{:=}\; \texttt{!}x_1 + 1;\;\; \texttt{!}x_1\, \texttt{in} \\
 & & \texttt{new}\, f_1 \\
 & \Downarrow & l_{f_1}
\end{array}
$$

$$
\begin{array}{rcl}
e_2 & = & \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
 & & \texttt{let}\, f_2 = \lambda z : \mathsf{unit}.\;\; x_2 \;\texttt{:=}\; \texttt{!}x_2 - 1; -(\texttt{!}x_2)\, \texttt{in} \\
 & & \texttt{new}\, f_2 \\
 & \Downarrow & l_{f_2}
\end{array}
$$

$$S \;=\; \{\, (k, W, s_1, s_2) \mid s_1(l_{f1}) \sim^{k,W} s_2(l_{f2}) : \mathsf{unit} \longrightarrow \mathsf{int} \,\}$$

- Worlds contain store relations
- Store relations contain worlds

2. Storing Functions in References

$$
\begin{array}{lcl}
e_1 & = & \texttt{let}\; x_1 = \texttt{new}\; 0\; \texttt{in} \\
 & & \texttt{let}\; f_1 = \lambda z:\mathsf{unit}.\;\; x_1 \mathrel{\texttt{:=}} \texttt{!}x_1 + 1;\; \texttt{!}x_1\; \texttt{in} \\
 & & \texttt{new}\; f_1 \\
 & \Downarrow & l_{f_1}
\end{array}
$$

$$
\begin{array}{lcl}
e_2 & = & \texttt{let}\; x_2 = \texttt{new}\; 0\; \texttt{in} \\
 & & \texttt{let}\; f_2 = \lambda z:\mathsf{unit}.\;\; x_2 \mathrel{\texttt{:=}} \texttt{!}x_2 - 1; -(\texttt{!}x_2)\; \texttt{in} \\
 & & \texttt{new}\; f_2 \\
 & \Downarrow & l_{f_2}
\end{array}
$$

$$S \;=\; \{\, (k, W, s_1, s_2) \mid s_1(l_{f1}) \sim^{k,W} s_2(l_{f2}) : \mathsf{unit} \longrightarrow \mathsf{int}\,\}$$

- Worlds contain store relations
- Store relations contain worlds

$\boxed{\text{Circular!}}$

2. Storing Functions in References

$$
\begin{aligned}
e_1 \quad &= \quad \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
& \texttt{let}\, f_1 = \lambda z : \mathsf{unit}.\;\; x_1 \,\texttt{:=}\, \texttt{!}x_1 + 1;\; \texttt{!}x_1\, \texttt{in} \\
& \texttt{new}\, f_1
\end{aligned}
$$

$$
\begin{aligned}
e_2 \quad &= \quad \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
& \texttt{let}\, f_2 = \lambda z : \mathsf{unit}.\;\; x_2 \,\texttt{:=}\, \texttt{!}x_2 - 1; -(\texttt{!}x_2)\, \texttt{in} \\
& \texttt{new}\, f_2
\end{aligned}
$$

$$
S \quad = \quad \{\, (k, W, s_1, s_2) \mid s_1(l_{f1}) \sim^{k-1, \lfloor W \rfloor_{k-1}} s_2(l_{f2}) : \mathsf{unit} \rightarrow \mathsf{int} \,\}
$$

1'. Data Abstraction via Local State

$$
\begin{aligned}
e_1 \quad &= \quad \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
&\qquad \lambda z : \mathsf{unit}.\ \ x_1 \texttt{ := } \texttt{!}x_1 + 1;\ \ \texttt{!}x_1
\end{aligned}
$$

$$
\begin{aligned}
e_2 \quad &= \quad \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
&\qquad \texttt{let}\, y_2 = \texttt{new}\, 0\, \texttt{in} \\
&\qquad \lambda z : \mathsf{unit}.\ \ x_2 \texttt{ := } \texttt{!}x_2 + 1;\ \ y_2 \texttt{ := } \texttt{!}y_2 + 1;\ \ (\texttt{!}x_2 + \texttt{!}y_2)/2
\end{aligned}
$$

1'. Data Abstraction via Local State

$$
\begin{aligned}
e_1 \quad = \quad & \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
& \lambda z : \mathsf{unit}.\ \ x_1 \texttt{ := } \texttt{!} x_1 + 1;\ \ \texttt{!} x_1 \\
\Downarrow \quad & \lambda z : \mathsf{unit}.\ \ l_{x1} \texttt{ := } \texttt{!} l_{x1} + 1;\ \ \texttt{!} l_{x1}
\end{aligned}
$$

$$
\begin{aligned}
e_2 \quad = \quad & \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
& \texttt{let}\, y_2 = \texttt{new}\, 0\, \texttt{in} \\
& \lambda z : \mathsf{unit}.\ \ x_2 \texttt{ := } \texttt{!} x_2 + 1;\ \ y_2 \texttt{ := } \texttt{!} y_2 + 1;\ \ (\texttt{!} x_2 + \texttt{!} y_2)/2 \\
\Downarrow \quad & \lambda z : \mathsf{unit}.\ \ l_{x2} \texttt{ := } \texttt{!} l_{x2} + 1; l_{y2} \texttt{ := } \texttt{!} l_{y2} + 1;\ \ (\texttt{!} l_{x2} + \texttt{!} l_{y2})/2
\end{aligned}
$$

1’. Data Abstraction via Local State

$$
\begin{aligned}
e_1 \quad &= \quad \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in} \\
&\qquad \lambda z : \mathsf{unit}.\ \ x_1 \texttt{ := } \texttt{!}x_1 + 1;\ \ \texttt{!}x_1 \\
&\Downarrow \quad \lambda z : \mathsf{unit}.\ \ l_{x1} \texttt{ := } \texttt{!}l_{x1} + 1;\ \ \texttt{!}l_{x1}
\end{aligned}
$$

$$
\begin{aligned}
e_2 \quad &= \quad \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in} \\
&\qquad \texttt{let}\, y_2 = \texttt{new}\, 0\, \texttt{in} \\
&\qquad \lambda z : \mathsf{unit}.\ \ x_2 \texttt{ := } \texttt{!}x_2 + 1;\ \ y_2 \texttt{ := } \texttt{!}y_2 + 1;\ \ (\texttt{!}x_2 + \texttt{!}y_2)/2 \\
&\Downarrow \quad \lambda z : \mathsf{unit}.\ \ l_{x2} \texttt{ := } \texttt{!}l_{x2} + 1; l_{y2} \texttt{ := } \texttt{!}l_{y2} + 1;\ (\texttt{!}l_{x2} + \texttt{!}l_{y2})/2
\end{aligned}
$$

1'. Data Abstraction via Local State

$$e_1 \quad = \quad \texttt{let}\, x_1 = \texttt{new}\, 0\, \texttt{in}$$
$$\lambda z : \mathsf{unit}.\;\; x_1 := !x_1 + 1;\;\; !x_1$$

$$\Downarrow \quad \lambda z : \mathsf{unit}.\;\; l_{x1} := !l_{x1} + 1;\;\; !l_{x1}$$

$$e_2 \quad = \quad \texttt{let}\, x_2 = \texttt{new}\, 0\, \texttt{in}$$
$$\texttt{let}\, y_2 = \texttt{new}\, 0\, \texttt{in}$$
$$\lambda z : \mathsf{unit}.\;\; x_2 := !x_2 + 1;\;\; y_2 := !y_2 + 1;\;\; (!x_2 + !y_2)/2$$

$$\Downarrow \quad \lambda z : \mathsf{unit}.\;\; l_{x2} := !l_{x2} + 1; l_{y2} := !l_{y2} + 1;\;\; (!l_{x2} + !l_{y2})/2$$

$$S \quad = \quad \{\, (s_1, s_2) \mid s_1(l_{x_1}) \;=\; (s_2(l_{x2}) \;+\; s_2(l_{y2}))/2 \,\}$$

store relation

3. Data Abstraction via Local State + ∃

$$\mathsf{Name} \quad = \quad \exists\alpha.\,\langle \mathtt{gen} : \mathsf{unit} \to \alpha,\ \mathtt{chk} : \alpha \to \mathsf{bool} \rangle$$

$$\begin{aligned} e_1 \quad = \quad & \mathtt{let}\, x = \mathtt{new}\, 0\, \mathtt{in} \\ & \mathtt{pack}\ \mathsf{int}, \langle \mathtt{gen} = \lambda z : \mathsf{unit}.\,(x \mathrel{\mathtt{:=}} \mathtt{!}x + 1; \mathtt{!}x), \\ & \qquad\qquad \mathtt{chk} = \lambda z : \mathsf{int}.\,(z \leq \mathtt{!}x) \rangle\ \mathtt{as}\ \mathsf{Name} \end{aligned}$$

3. Data Abstraction via Local State + ∃

$$\mathsf{Name} \;=\; \exists\alpha.\,\langle \mathtt{gen} : \mathsf{unit} \to \alpha,\ \mathtt{chk} : \alpha \to \mathsf{bool}\rangle$$

$$\begin{aligned} e_1 \;=\;& \mathtt{let}\, x = \mathtt{new}\, 0\, \mathtt{in} \\ & \mathsf{pack}\ \mathsf{int}, \langle \mathtt{gen} = \lambda z : \mathsf{unit}.\,(x \mathrel{\mathtt{:=}} \mathtt{!}x + 1;\, \mathtt{!}x), \\ & \qquad\qquad \mathtt{chk} = \lambda z : \mathsf{int}.\,(z \le \mathtt{!}x)\rangle\ \mathsf{as}\ \mathsf{Name} \end{aligned}$$

$$\begin{aligned} e_2 \;=\;& \mathtt{let}\, x = \mathtt{new}\, 0\, \mathtt{in} \\ & \mathsf{pack}\ \mathsf{int}, \langle \mathtt{gen} = \lambda z : \mathsf{unit}.\,(x \mathrel{\mathtt{:=}} \mathtt{!}x + 1;\, \mathtt{!}x), \\ & \qquad\qquad \mathtt{chk} = \lambda z : \mathsf{int}.\,\mathtt{true}\rangle\ \mathsf{as}\ \mathsf{Name} \end{aligned}$$

3. Data Abstraction via Local State + ∃

$$\mathsf{Name} \quad = \quad \exists\alpha.\, \langle \mathtt{gen} : \mathsf{unit} \rightarrow \alpha,\ \mathtt{chk} : \alpha \rightarrow \mathsf{bool} \rangle$$

$$\begin{array}{rcl} e_1 & = & \mathtt{let}\, x = \mathtt{new}\, 0\, \mathtt{in} \\ & & \mathsf{pack\ int}, \langle \mathtt{gen} = \lambda z : \mathsf{unit}.\, (x := !x + 1; !x), \\ & & \qquad \mathtt{chk} = \lambda z : \mathsf{int}.\, (z \leq !x) \rangle \ \mathsf{as\ Name} \end{array}$$

$$\begin{array}{rcl} e_2 & = & \mathtt{let}\, x = \mathtt{new}\, 0\, \mathtt{in} \\ & & \mathsf{pack\ int}, \langle \mathtt{gen} = \lambda z : \mathsf{unit}.\, (x := !x + 1; !x), \\ & & \qquad \mathtt{chk} = \lambda z : \mathsf{int}.\, \mathtt{true} \rangle \ \mathsf{as\ Name} \end{array}$$

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

3. Data Abstraction via Local State + ∃

$$\mathsf{Name} \quad = \quad \exists\alpha.\, \langle \mathtt{gen} : \mathsf{unit} \to \alpha,\ \mathtt{chk} : \alpha \to \mathsf{bool} \rangle$$

$$\begin{array}{lcl} e_1 & = & \mathtt{let}\, x = \mathtt{new}\, 0\, \mathtt{in} \\ & & \mathsf{pack\ int}, \langle \mathtt{gen} = \lambda z : \mathsf{unit}.\, (x := !x + 1; !x), \\ & & \qquad\qquad \mathtt{chk} = \lambda z : \mathsf{int}.\, (z \leq\ !x) \rangle \ \mathsf{as\ Name} \end{array}$$

$$\begin{array}{lcl} e_2 & = & \mathtt{let}\, x = \mathtt{new}\, 0\, \mathtt{in} \\ & & \mathsf{pack\ int}, \langle \mathtt{gen} = \lambda z : \mathsf{unit}.\, (x := !x + 1; !x), \\ & & \qquad\qquad \mathtt{chk} = \lambda z : \mathsf{int}.\, \mathtt{true} \rangle \ \mathsf{as\ Name} \end{array}$$

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 α ?

3. Data Abstraction via Local State + ∃

$$\mathsf{Name} \quad = \quad \exists\alpha.\,\langle \mathtt{gen} : \mathsf{unit} \to \alpha,\ \mathtt{chk} : \alpha \to \mathsf{bool}\rangle$$

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

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

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)

	∀ ∃	μ	ref	Simple Model
Plotkin'73, Statman'85				✗
Reynolds'83, Mitchell'86	✓			✗
Pitts-Stark'93,'98 (ref int)			✓–	✓
Pitts'98,'00 (recursive functions)	✓			✓–
Birkedal-Harper'99, Crary-Harper'07	✓	✓		✗
Appel-McAllester'01		✓–		✓
Benton-Leperchey'05 (ref...ref int)		✓	✓–	✗ ✗
Ahmed'06	✓	✓		✓
Bohr-Birkedal'06		✓	✓	✗ ✗
Ahmed-Dreyer-Rossberg'09	✓	✓	✓	✓

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]

$P(v_{orig})$ $P(v_{changed})$

v_0 v_1

Imperative Self-Adjusting Computation

[Acar-Ahmed-Blume, POPL'08]

$P(v_{orig})$ $\Downarrow$ v_0

$P(v_{changed})$ $\Downarrow$ v_1

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

Imperative Self-Adjusting Computation

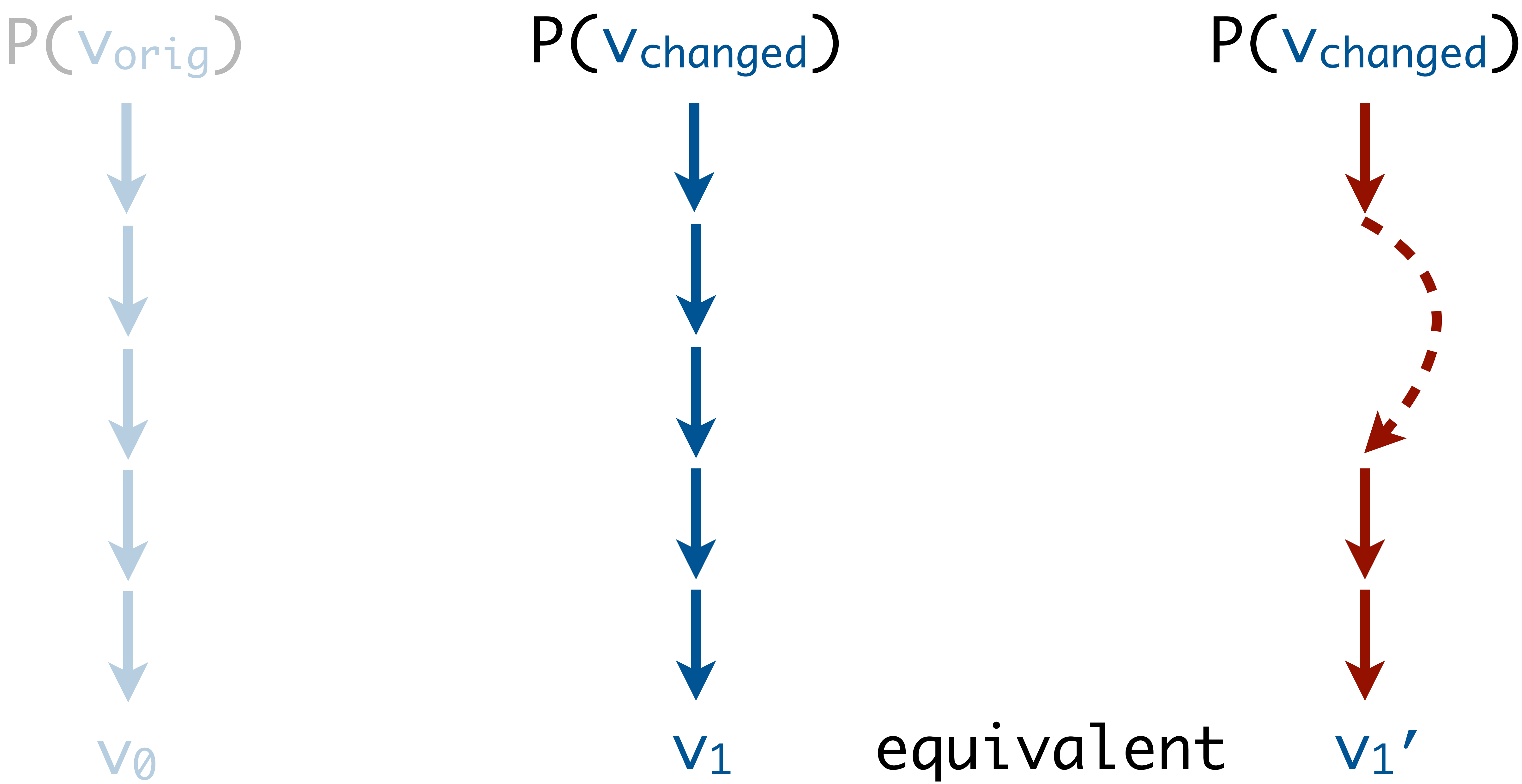
Overview:

- 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

$$
\begin{tikzcd}
P(v_{orig}) \arrow[d] & P(v_{changed}) \arrow[d] \\
\cdot \arrow[d] & \cdot \arrow[d] \\
\cdot \arrow[d] & \cdot \arrow[d] \\
\cdot \arrow[d] & \cdot \arrow[d] \\
\cdot \arrow[d] & \cdot \arrow[d] \\
v_0 & v_1
\end{tikzcd}
$$

Equivalence of Evaluation Strategies



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 ($\mathrm{SM}^{\tau}\,e$, $^{\tau}\mathrm{MS}\;e$)

- 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 *[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 *[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 $\triangleright A$ 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
- Step-indexing: hiding the steps
- Open problems, future directions

1. Other Language Features...

Dependent types

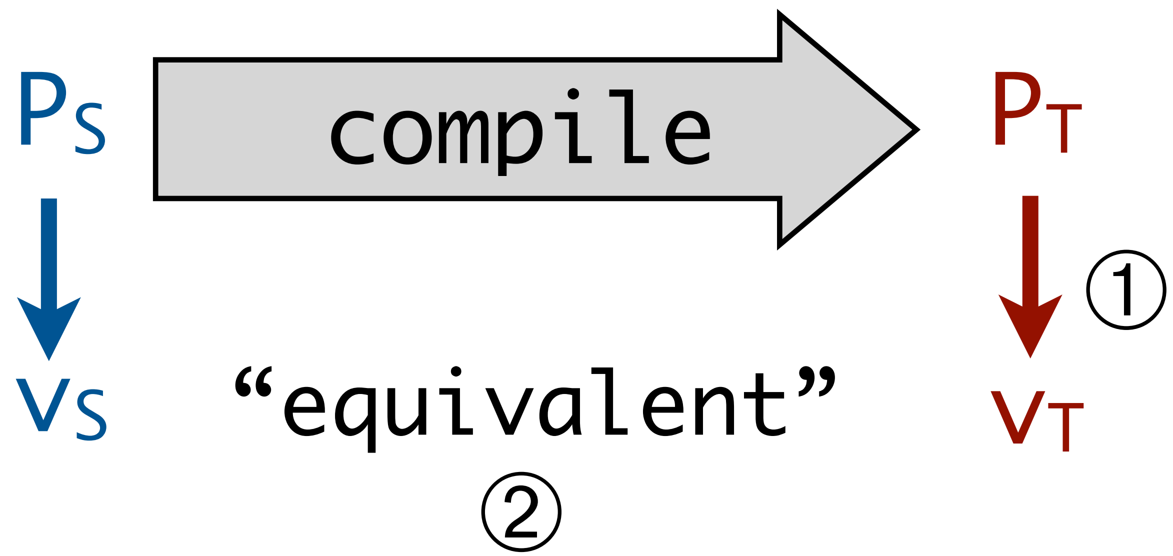
- depends on the dependent type theory!
- Coq / ECC / Hoare Type Theory (HTT): higher-order logic
  - would like an operational model of propositional equality; how to deal with impredicativity of h.o.l. (no notion of consuming steps at logical level)
- parametricity for HTT: (extends Coq with type {P}x:A{Q})
  - 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



Equivalence-Preserving Compilation

- Semantics-preserving compilation

$$\begin{tikzcd} e_S \arrow[r, "\texttt{compile}"] \arrow[d] & e_T \arrow[d, "①"] \\ v_S \arrow[r, phantom, "\text{“}\texttt{equivalent}\text{”}\ ②"] & v_T \end{tikzcd}$$

- Equivalence-preserving compilation

If $e_{S1} \xrightarrow{\texttt{compile}} e_{T1}$ and $e_{S2} \xrightarrow{\texttt{compile}} e_{T2}$

then $e_{S1} \approx_S^{ctx} e_{S2} \implies e_{T1} \approx_T^{ctx} e_{T2}$

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

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) *[Abadi+'99]* can encode secure information flow
- Noninterference-preserving translation from DCC to System Fω *[Bowman-Ahmed, ICFP'15]*
- Includes:
  - "open" logical relation for Fω (based on *[Zhao+, APLAS'10]*)
  - cross-language logical relation between DCC and Fω
  - unary logical relation to show that back-translation from Fω 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?