Contextual Equivalence in Higher-Order Typed Languages

Andrew Pitts

UNIVERSITY OF CAMBRIDGE
Computer Laboratory
When are two programs equal?
Plan

What are they and why study them?
Contextual equivalence without contexts
Techniques for HOT languages
Conclusions
Contextual Equivalences for HOT Programming Languages

Plan

What are they and why study them?
Contextual equivalence without contexts
Techniques for HOT languages
Conclusions
Contextual Equivalences for HOT Programming Languages

Plan

What are they and why study them?
Contextual equivalence without contexts
Techniques for HOT languages
Conclusions
Contextual Equivalences for HOT Programming Languages

What are they and why study them?
Contextual equivalence without contexts
Techniques for HOT languages
Conclusions

First-class functions.
Types: higher-order, polymorphic, recursive.
+ local mutable state, modules, objects, concurrency, proof search, ...
What are they and why study them?
Contextual equivalence without contexts
Techniques for HOT languages
Conclusions
Contextual Equivalences for HOT Programming Languages

Plan

What are they and why study them?
Contextual equivalence without contexts
Techniques for HOT languages
Conclusions
Contextual Equivalences for HOT Programming Languages

Plan

What are they and why study them?
Contextual equivalence without contexts
Techniques for HOT languages
Conclusions
Two phrases of a programming language are ("Morris style") contextually equivalent ($\approx_{\text{ctx}}$) if occurrences of the first phrase in any program can be replaced by the second phrase without affecting the observable results of executing the program.
Are these OCaml expressions contextually equivalent?

\[
\begin{array}{c|c}
\text{z : int} & H : \text{int} \rightarrow \text{int} \\
\hline
\hline
H \triangleq & \text{let } a = \text{ref } z \text{ in} \\
& \text{fun } x \rightarrow a := !a + x ; \ !a \\
\end{array}
\begin{array}{c|c}
\text{z : int} & K : \text{int} \rightarrow \text{int} \\
\hline
\hline
K \triangleq & \text{let } b = \text{ref}(\neg z) \text{ in} \\
& \text{fun } y \rightarrow b := !b - y ; \ \neg(!b)
\end{array}
\]
Are these OCaml expressions contextually equivalent?

\[
\begin{array}{c|c}
\text{z : int} & \text{H : int \rightarrow int} \\
\hline
\text{H} \triangleq & \text{K} \triangleq \\
\text{let } a = \text{ref } z \text{ in} & \text{let } b = \text{ref } (-z) \text{ in} \\
\text{fun } x \rightarrow a := !a + x ; & \text{fun } y \rightarrow b := !b - y ; \\
!a & -(!b)
\end{array}
\]

Yes, \( z : \text{int} \vdash H \equiv_{\text{ctx}} K : \text{int} \rightarrow \text{int} \), in the sense that for all states \( s \) and all well-typed, closing contexts \( C[\_\_] \),

\[
\exists s'. \langle s, C[H] \rangle \rightarrow^* \langle s', \text{true} \rangle \\
\iff \exists s''. \langle s, C[K] \rangle \rightarrow^* \langle s'', \text{true} \rangle
\]
Are these OCaml expressions contextually equivalent?

\[
\begin{align*}
  &\text{z : int } \vdash \, H : \text{int } \to \text{int } \mid \, \text{z : int } \vdash \, K : \text{int } \to \text{int} \\
  &H \triangleq \\
  &\text{let } a = \text{ref z in} \\
  &\text{fun } x \to a := !a + x ; \\
  &\text{!a}
\end{align*}
\]

Yes, \( z : \text{int } \vdash H \cong_{\text{ctx}} K : \text{int } \to \text{int} \), in the sense that for all states \( s \) and all well-typed, closing contexts \( C[\_\_] \),

\[
\exists s'. \langle s, C[H] \rangle \to^* \langle s', \text{true} \rangle \quad \Leftrightarrow \quad \exists s''. \langle s, C[K] \rangle \to^* \langle s'', \text{true} \rangle
\]
Are these OCaml expressions contextually equivalent?

<table>
<thead>
<tr>
<th>$z : \text{int} \vdash H : \text{int} \rightarrow \text{int}$</th>
<th>$z : \text{int} \vdash K : \text{int} \rightarrow \text{int}$</th>
</tr>
</thead>
</table>
| $H \triangleq \begin{array}{l}
\text{let } a = \text{ref } z \text{ in } \\
\text{fun } x \rightarrow a := !a + x ; \\
!a
\end{array}$ | $K \triangleq \begin{array}{l}
\text{let } b = \text{ref } (-z) \text{ in } \\
\text{fun } y \rightarrow b := !b - y ; \\
-(!b)
\end{array}$ |

Yes, $z : \text{int} \vdash H \equiv_{\text{ctx}} K : \text{int} \rightarrow \text{int}$, in the sense that for all states $s$ and all well-typed, closing contexts $C[\_\_]$, 

$$\exists s'. \langle s, C[H] \rangle \rightarrow^* \langle s', \text{true} \rangle$$

$$\iff \exists s''. \langle s, C[K] \rangle \rightarrow^* \langle s'', \text{true} \rangle$$

OCaml structural operational semantics (mythical!): transition relation between $\langle \text{state}, \text{expression} \rangle$-pairs.
Two phrases of a programming language are ("Morris style") contextually equivalent ($\simeq_{\text{ctx}}$) if occurrences of the first phrase in any program can be replaced by the second phrase without affecting the observable results of executing the program.

Different choices lead to possibly different notions of contextual equivalence.
Two phrases of a programming language are ("Morris style") contextually equivalent ($\simeq_{ctx}$) if occurrences of the first phrase in any program can be replaced by the second phrase without affecting the observable results of executing the program.

Gottfried Wilhelm Leibniz (1646–1716): two mathematical objects are equal if there is no test to distinguish them.
Two phrases of a programming language are ("Morris style") contextually equivalent ($\equiv_{ctx}$) if occurrences of the first phrase in any program can be replaced by the second phrase without affecting the observable results of executing the program.

Gottfried Wilhelm Leibniz (1646–1716): two mathematical objects are equal if there is no test to distinguish them.

Two phrases of a programming language are ("Morris style") contextually equivalent ($\simeq_{\text{ctx}}$) if occurrences of the first phrase in any program can be replaced by the second phrase without affecting the observable results of executing the program.

We assume the programming language comes with a structural operational semantics (SOS) as part of its definition!

Contextual equivalence is part of operationally-based reasoning about programming languages...
“But once feasibility has been checked by an operational model, operational reasoning should be immediately abandoned; it is essential that all subsequent reasoning, calculation and design should be conducted in each case at the highest possible level of abstraction.”

"But once feasibility has been checked by an operational model, operational reasoning should be immediately abandoned; it is essential that all subsequent reasoning, calculation and design should be conducted in each case at the highest possible level of abstraction."

Why contextual equivalence matters

- Philosophically important:
  operational behaviour is a characteristic feature of programming language theory that distinguishes it from related areas of logic.
  (Proof Theory, Model Theory, Recursion Theory)

- Pragmatically important:
  Contextual equivalence is used in verification of many programming language correctness properties.
  (E.g. compiler optimisations, correctness of ADTs, information hiding and security properties, . . . )
Why contextual equivalence matters

What is special about HOT languages?

- type-directed “laws” for contextual equivalence :-(

- higher-order types ⇒ programs can make use of constituent phrases in dynamically complicated ways :-(

Why contextual equivalence matters

What is special about HOT languages?

- type-directed “laws” for contextual equivalence

- higher-order types $\Rightarrow$ programs can make use of constituent phrases in dynamically complicated ways

  e.g. Extensionality property for function types...
Are these OCaml expressions contextually equivalent?

<table>
<thead>
<tr>
<th>$F: \text{unit ref} \rightarrow \text{unit ref}$</th>
<th>$G: \text{unit ref} \rightarrow \text{unit ref}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F \triangleq$</td>
<td>$G \triangleq$</td>
</tr>
<tr>
<td>let $a = \text{ref}()$ in</td>
<td>let $c = \text{ref}()$ in</td>
</tr>
<tr>
<td>let $b = \text{ref}()$ in</td>
<td>let $d = \text{ref}()$ in</td>
</tr>
<tr>
<td>fun $x \rightarrow \text{if } x == a \text{ then } b$</td>
<td>fun $y \rightarrow \text{if } y == d \text{ then } d$</td>
</tr>
<tr>
<td></td>
<td>else $a$</td>
</tr>
<tr>
<td></td>
<td>else $c$</td>
</tr>
</tbody>
</table>

```
public
x       a
\rightarrow
b
\rightarrow
private
```
Are these OCaml expressions contextually equivalent?

\[
\begin{array}{ll}
\text{F : unit ref} & \rightarrow \text{unit ref} \\
\text{G : unit ref} & \rightarrow \text{unit ref} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{F} \triangleq \\
\text{let } a = \text{ref()} \text{in} \\
\text{let } b = \text{ref()} \text{in} \\
\text{fun } x \rightarrow \text{if } x == a \text{ then } b \\
\text{else } a \\
\end{array}
\]

\[
\begin{array}{ll}
\text{G} \triangleq \\
\text{let } c = \text{ref()} \text{in} \\
\text{let } d = \text{ref()} \text{in} \\
\text{fun } y \rightarrow \text{if } y == d \text{ then } d \\
\text{else } c \\
\end{array}
\]

No

For \( T \triangleq \text{fun } f \rightarrow \text{let } x = \text{ref()} \text{in} f(f(x)) == f(x), \)

\( TF \) has value \text{false}, whereas \( TG \) has value \text{true},

so \( F \not\sim_{\text{ctx}} G. \)
Are these OCaml expressions contextually equivalent?

\[
\begin{align*}
  z & : \text{int} \vdash H : \text{int} \rightarrow \text{int} \quad | \quad z & : \text{int} \vdash K : \text{int} \rightarrow \text{int} \\
  H & \triangleq \text{let } a = \text{ref } z \text{ in} \\
  & \quad \text{fun } x \rightarrow a := !a + x ; \\
  & \quad !a \\
  K & \triangleq \text{let } b = \text{ref}(\neg z) \text{ in} \\
  & \quad \text{fun } y \rightarrow b := !b - y ; \\
  & \quad -(!b)
\end{align*}
\]

Yes, \( z : \text{int} \vdash H \equiv_{\text{ctx}} K : \text{int} \rightarrow \text{int} \), in the sense that for all states \( s \) and all well-typed, closing contexts \( C[\_\_] \),

\[
\exists s'. \langle s, C[H] \rangle \rightarrow^* \langle s', \text{true} \rangle \\
\iff \exists s''. \langle s, C[K] \rangle \rightarrow^* \langle s'', \text{true} \rangle
\]

How does one prove such statements?
Are these OCaml expressions contextually equivalent?

\[
\begin{align*}
z &: \text{int} \vdash H &: \text{int} \rightarrow \text{int} \\
H &\triangleq \\
\text{let } a &= \text{ref } z \text{ in} \\
\text{fun } x &\rightarrow a := !a + x ; \\
!a \\
\end{align*}
\]

\[
\begin{align*}
z &: \text{int} \vdash K &: \text{int} \rightarrow \text{int} \\
K &\triangleq \\
\text{let } b &= \text{ref } (-z) \text{ in} \\
\text{fun } y &\rightarrow b := !b - y ; \\
-(!b) \\
\end{align*}
\]

Yes, \(z : \text{int} \vdash H \cong_{\text{ctx}} K : \text{int} \rightarrow \text{int}\), in the sense that for all states \(s\) and all well-typed, closing contexts \(C[\_]\),

\[
\exists s'. \langle s, C[H] \rangle \rightarrow^* \langle s', \text{true} \rangle \\
\Leftrightarrow \exists s''. \langle s, C[K] \rangle \rightarrow^* \langle s'', \text{true} \rangle
\]

How does one prove such statements?

these cause difficulty
Formalizing $\cong_{\text{ctx}}$ without contexts
Contexts are too concrete

The semantics of programs only depends on their abstract syntax (parse trees)

\[
\left( \begin{array}{c}
\text{let } a = \text{ref } 0 \text{ in } \\
\text{fun } x \rightarrow \\
\quad a := \! a + x \\
\quad \! a
\end{array} \right) = \\
\left( \begin{array}{c}
\text{let } a = \text{ref } 0 \\
in \\
\text{fun } x \rightarrow \\
\quad a := \! a + x \\
\quad \! a
\end{array} \right)
\]
Contexts are too concrete

The semantics of programs only depends on their abstract syntax (parse trees) modulo renaming of bound identifiers ($\alpha$-equivalence, $\equiv_\alpha$).

\[
\begin{align*}
\text{let } a = \text{ref } 0 \text{ in } \\
\text{fun } x \to \\
a := !a + x ; \\
!a
\end{align*}
\]  \quad =_\alpha  \quad
\begin{align*}
\text{let } b = \text{ref } 0 \\
in \\
\text{fun } y \to \\
b := !b + y ; \\
!b
\end{align*}

E.g. definition & properties of OCaml typing relation $\Gamma \vdash M : \tau$ are simpler if we identify $M$ up to $\equiv_\alpha$. 
Contexts are too concrete

The semantics of programs only depends on their abstract syntax (parse trees) modulo renaming of bound identifiers ($\alpha$-equivalence, $\equiv_{\alpha}$).

So it pays to formulate program equivalences using mathematical notions that respect $\alpha$-equivalence.

But filling holes in contexts does not respect $\equiv_{\alpha}$:
Contexts are too concrete

The semantics of programs only depends on their abstract syntax (parse trees) modulo renaming of bound identifiers ($\alpha$-equivalence, $\equiv_\alpha$).

So it pays to formulate program equivalences using mathematical notions that respect $\alpha$-equivalence.

But filling holes in contexts does not respect $\equiv_\alpha$:

\[
\text{fun } x \rightarrow (-) \equiv_\alpha \text{ fun } y \rightarrow (-)
\]

and

\[
x \equiv_\alpha x
\]

but

\[
\text{fun } x \rightarrow x \not\equiv_\alpha \text{ fun } y \rightarrow x
\]
Expression relations

Language’s typing relation

\[ \Gamma \vdash M : \tau \]

dictates the form of relations like contextual equivalence:
Expression relations

Language’s typing relation

\[ \Gamma \vdash M : \tau \]

dictates the form of relations like contextual equivalence:

Define an expression relation to be any set \( \mathcal{E} \) of tuples \((\Gamma, M, M', \tau)\) satisfying:

\[ (\Gamma \vdash M \mathcal{E} M' : \tau) \Rightarrow (\Gamma \vdash M : \tau) \& (\Gamma \vdash M' : \tau) \]
Operations on expression relations

Composition $\mathcal{E}_1, \mathcal{E}_2 \leftrightarrow \mathcal{E}_1; \mathcal{E}_2$:

\[
\frac{\Gamma \vdash M \mathcal{E}_1 M' : \tau \quad \Gamma \vdash M' \mathcal{E}_2 M'' : \tau}{\Gamma \vdash M (\mathcal{E}_1; \mathcal{E}_2) M'' : \tau}
\]

Reciprocation $\mathcal{E} \leftrightarrow \mathcal{E}^\circ$:

\[
\frac{\Gamma \vdash M \mathcal{E} M' : \tau}{\Gamma \vdash M' \mathcal{E}^\circ M : \tau}
\]

Identity $\text{Id}$:

\[
\frac{\Gamma \vdash M : \tau}{\Gamma \vdash M \text{Id} M : \tau}
\]
Compatible refinement $\mathcal{E} \leftrightarrow \hat{\mathcal{E}}$: 

$$
\begin{align*}
\Gamma \vdash M_1 : \tau \rightarrow \tau' & \quad M_2 : \tau \\
\therefore \quad \Gamma \vdash M_1 \ M_2 : \tau'
\end{align*}
$$
Compatible refinement $\mathcal{E} \leftrightarrow \mathcal{E}^*$:

$$
\Gamma \vdash M_1 \mathcal{E} M'_1 : \tau \rightarrow \tau' \quad \Gamma \vdash M_2 \mathcal{E} M'_2 : \tau \\
\Gamma \vdash M_1 M_2 \mathcal{E} M'_1 M'_2 : \tau'
$$
Operations on expression relations

Compatible refinement $\mathcal{E} \leftrightarrow \mathcal{\hat{E}}$: 

\[
\begin{align*}
\Gamma \vdash M_1 \mathcal{E} M_1' : \tau \to \tau' & \quad \Gamma \vdash M_2 \mathcal{E} M_2' : \tau \\
\Gamma \vdash M_1 M_2 \mathcal{\hat{E}} M_1' M_2' : \tau' \\
\Gamma, x : \tau \vdash M : \tau' \\
\Gamma \vdash (\text{fun } x \to M) : \tau \to \tau'
\end{align*}
\]
Compatible refinement $\mathcal{E} \mapsto \hat{\mathcal{E}}$:

\[
\begin{align*}
\Gamma \vdash M_1 \mathcal{E} M_1' : \tau \rightarrow \tau' \\
\Gamma \vdash M_2 \mathcal{E} M_2' : \tau \\
\hline
\Gamma \vdash M_1 M_2 \hat{\mathcal{E}} M_1' M_2' : \tau'
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : \tau \vdash M \mathcal{E} M' : \tau' \\
\hline
\Gamma \vdash (\text{fun } x \rightarrow M) \hat{\mathcal{E}} (\text{fun } x \rightarrow M') : \tau \rightarrow \tau'
\end{align*}
\]
Operations on expression relations

Compatible refinement $\mathcal{E} \leftrightarrow \hat{\mathcal{E}}$:

\[
\begin{align*}
\Gamma \vdash M_1 \mathcal{E} M'_1 : \tau & \rightarrow \tau' & \Gamma \vdash M_2 \mathcal{E} M'_2 : \tau \\
\Gamma \vdash M_1 M_2 \hat{\mathcal{E}} M'_1 M'_2 : \tau' \\
\Gamma, x : \tau \vdash M \mathcal{E} M' : \tau' \\
\Gamma \vdash (\text{fun} \ x \rightarrow M) \hat{\mathcal{E}} (\text{fun} \ x \rightarrow M') : \tau \rightarrow \tau' \\
\Gamma \vdash M : \tau \\
\Gamma \vdash \text{ref} M : \tau \text{ ref}
\end{align*}
\]
Compatible refinement $\mathcal{E} \leftrightarrow \hat{\mathcal{E}}$:

\[
\begin{align*}
\Gamma \vdash M_1 \mathcal{E} M'_1 : \tau \rightarrow \tau' \quad &\quad \Gamma \vdash M_2 \mathcal{E} M'_2 : \tau \\
\Gamma \vdash M_1 M_2 \hat{\mathcal{E}} M'_1 M'_2 : \tau' \\
\Gamma, x : \tau \vdash M \mathcal{E} M' : \tau' \\
\Gamma \vdash \text{(fun } x \rightarrow M) \hat{\mathcal{E}} \text{(fun } x \rightarrow M') : \tau \rightarrow \tau' \\
\Gamma \vdash M \mathcal{E} M' : \tau \\
\Gamma \vdash \text{ref } M \hat{\mathcal{E}} \text{ ref } M' : \tau \text{ ref}
\end{align*}
\]

etc, etc (one rule for each typing rule)
Contextual equiv. without contexts

**Theorem** [Gordon, Lassen (1998)]

$\equiv_{\text{ctx}}$ (defined conventionally, using contexts) is the greatest **compatible** & adequate expression relation.

where an expression relation $\mathcal{E}$ is

- compatible if $\hat{\mathcal{E}} \subseteq \mathcal{E}$
Theorem [Gordon, Lassen (1998)]

$\equiv_{ctx}$ (defined conventionally, using contexts) is the greatest compatible & adequate expression relation.

where an expression relation $E$ is

- compatible if $\hat{E} \subseteq E$
- adequate if $\emptyset \vdash M E M' : \text{bool} \Rightarrow \forall s. \\ (\exists s'. \langle s, M \rangle \rightarrow^* \langle s', \text{true} \rangle) \iff (\exists s'' . \langle s, M' \rangle \rightarrow^* \langle s'', \text{true} \rangle)$

Precise definition varies according to the observational scenario. E.g. use “bisimulation” rather than “trace” based adequacy in presence of concurrency features.
Contextual equiv. without contexts

**Definition**

\[ \approx_{\text{ctx}} \] is the union of all expression relations that are compatible and adequate.

where an expression relation \( \mathcal{E} \) is

- compatible if \( \hat{\mathcal{E}} \subseteq \mathcal{E} \)
- adequate if \( \emptyset \vdash M \mathcal{E} M' : \text{bool} \Rightarrow \forall s. (\exists s'. \langle s, M \rangle \rightarrow^* \langle s', \text{true} \rangle ) \Leftrightarrow (\exists s''. \langle s, M' \rangle \rightarrow^* \langle s'', \text{true} \rangle ) \)

So defined, \( \approx_{\text{ctx}} \) is also reflexive \( (\text{Id} \subseteq \mathcal{E}) \), symmetric \( (\mathcal{E}^\circ \subseteq \mathcal{E}) \) and transitive \( (\mathcal{E} ; \mathcal{E} \subseteq \mathcal{E}) \).
Techniques
<table>
<thead>
<tr>
<th>Method</th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domains</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Games</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical relns</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bisimulations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program logics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**sound**: determines a compatible and adequate expression relation

**complete**: characterises $\equiv_{\text{ctx}}$

**useful**: for proving programming “laws” & PL correctness properties

**general**: what PL features can be dealt with?
<table>
<thead>
<tr>
<th>Method</th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>(+)</td>
</tr>
<tr>
<td>CIU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domains</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Games</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical relns</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bisimulations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program logics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Brute force**: sometimes compatible closure of \( \{(M, M')\} \) is adequate, and hence \( M \cong_{\text{ctx}} M' \).

(E.g. [AMP, POPL 2007].)
<table>
<thead>
<tr>
<th>Method</th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>(++)</td>
</tr>
<tr>
<td>CIU</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Domains</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Games</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical relns</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bisimulations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program logics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CIU:** “Uses of Closed Instantiations” [Mason-Talcott et al].

Equates open expressions if their closures w.r.t. substitutions have same reduction behaviour w.r.t. any frame stack.
<table>
<thead>
<tr>
<th></th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Brute force</strong></td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>(+)</td>
</tr>
<tr>
<td><strong>CIU</strong></td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td><strong>Domains</strong></td>
<td>+</td>
<td>−</td>
<td>±</td>
<td>+</td>
</tr>
<tr>
<td><strong>Games</strong></td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>−</td>
</tr>
<tr>
<td><strong>Logical relns</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bisimulations</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Program logics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Domains**: traditional denotational semantics.

**Games**: game semantics [Abramsky, Malacaria, Hyland, Ong, ...]
<table>
<thead>
<tr>
<th>Method</th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>(+)</td>
</tr>
<tr>
<td>CIU</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Domains</td>
<td>+</td>
<td>−</td>
<td>±</td>
<td>+</td>
</tr>
<tr>
<td>Games</td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>−</td>
</tr>
<tr>
<td>Logical relns</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Bisimulations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program logics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Logical relations**: type-directed analysis of $\equiv_{ctx}$. At function types: relate functions if they send related arguments to related results.

Initially denotational [Plotkin, . . .], but now also operational [AMP, Birkedal-Harper-Crary, Ahmed, Johann-Voigtlaender, . . .].
<table>
<thead>
<tr>
<th>Method</th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>(+)</td>
</tr>
<tr>
<td>CIU</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Domains</td>
<td>+</td>
<td>−</td>
<td>±</td>
<td>+</td>
</tr>
<tr>
<td>Games</td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>−</td>
</tr>
<tr>
<td>Logical relns</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Bisimulations</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Program logics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bisimulations—the legacy of concurrency theory:

\[ M_1 \sim M_2 \]

\[ T \downarrow \]

\[ M'_1 \]
<table>
<thead>
<tr>
<th></th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>(++)</td>
</tr>
<tr>
<td>CIU</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Domains</td>
<td>+</td>
<td>-</td>
<td>±</td>
<td>+</td>
</tr>
<tr>
<td>Games</td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>-</td>
</tr>
<tr>
<td>Logical relns</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Bisimulations</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Program logics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bisimulations—the legacy of concurrency theory:

\[ M_1 \sim M_2 \] (and symmetrically)

\[ M_1' \sim M_2' \]
<table>
<thead>
<tr>
<th></th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Brute force</strong></td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>(+)</td>
</tr>
<tr>
<td><strong>CIU</strong></td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td><strong>Domains</strong></td>
<td>+</td>
<td>−</td>
<td>±</td>
<td>+</td>
</tr>
<tr>
<td><strong>Games</strong></td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>−</td>
</tr>
<tr>
<td><strong>Logical relns</strong></td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td><strong>Bisimulations</strong></td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Program logics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bisimulations**—the legacy of concurrency theory:

- applicative  [Abramsky, Gordon, AMP]
- environmental  [Pierce-Sumii-Koutavas-Wand]
- “up-to” techniques  [Sangiorgi, Lassen]
<table>
<thead>
<tr>
<th></th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>(+)</td>
</tr>
<tr>
<td>CIU</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Domains</td>
<td>+</td>
<td>−</td>
<td>±</td>
<td>+</td>
</tr>
<tr>
<td>Games</td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>−</td>
</tr>
<tr>
<td>Logical relns</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Bisimulations</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Program logics</td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>−</td>
</tr>
</tbody>
</table>

**Program logics**—e.g. higher-order Hoare logic

[Berger-Honda-Yoshida]

Beyond universal identities.
<table>
<thead>
<tr>
<th>Technique</th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>(+)</td>
</tr>
<tr>
<td>CIU</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Domains</td>
<td>+</td>
<td>−</td>
<td>±</td>
<td>+</td>
</tr>
<tr>
<td>Games</td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>−</td>
</tr>
<tr>
<td>Logical relns</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Bisimulations</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Program logics</td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>−</td>
</tr>
</tbody>
</table>

Q: How do we make sense of all these techniques and results?
<table>
<thead>
<tr>
<th>Method</th>
<th>sound</th>
<th>complete</th>
<th>useful</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>(+)</td>
</tr>
<tr>
<td>CIU</td>
<td>+</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Domains</td>
<td>+</td>
<td>−</td>
<td>±</td>
<td>+</td>
</tr>
<tr>
<td>Games</td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>−</td>
</tr>
<tr>
<td>Logical relns</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Bisimulations</td>
<td>+</td>
<td>±</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Program logics</td>
<td>+</td>
<td>+</td>
<td>±</td>
<td>−</td>
</tr>
</tbody>
</table>

Q: How do we make sense of all these techniques and results?

A: Category Theory can help!

For example...
Example

Relational parametricity is a tool for proving contextual equivalences between polymorphic programs:

\[ \emptyset \vdash \Lambda \alpha . e_1 \simeq_{\text{ctx}} \Lambda \alpha . e_2 : \forall \alpha . \tau \]

if and only if

for all \( \tau_1, \tau_2 \) and all “good” relations \( \tau_1 \leftrightarrow \tau_2 \),

\( e_1[\tau_1/\alpha] \) and \( e_2[\tau_2/\alpha] \) are related by \( \tau[\tau_1/\alpha] \leftrightarrow \tau[\tau_2/\alpha] \)

Category theory guides us to

- “free theorems” via natural transformations [Wadler];
- universal properties of recursive datatypes: initial algebras / final coalgebras / Freyd’s free dialgebras [Hasagawa et al].
“But once feasibility has been checked by an operational model, operational reasoning should be immediately abandoned; it is essential that all subsequent reasoning, calculation and design should be conducted in each case at the highest possible level of abstraction.”

Conclusions

- Operational models can support “reasoning, calculation and design” at a high level of abstraction—especially if we let Category Theory be our guide.

- Calculus of expression relations provides a useful setting for developing the properties of contextual equivalence.
Research opportunities

- The development of programming language theory based on contextual equivalences lags far behind the development of programming language design.

Type soundness results are two a penny, but correctness properties up to $\sim_{ctx}$ are scarce (because they are hard!).

E.g. FP community is enthusiastically designing languages combining (higher rank) polymorphic types/kinds with recursively defined functions, datatypes, local state, subtyping, ... In many cases the relational parametricity properties of $\sim_{ctx}$ are unknown.
Research opportunities

- The development of programming language theory based on contextual equivalences lags far behind the development of programming language design.

- Operationally-based work on programming language theory badly needs better tools for computer-aided proof.

"You want proof? I'll give you proof!"
O fim