Contextual Equivalence in Higher-Order Typed Languages

Andrew Pitts

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

Plan

What are they and why study them?

Contextual equivalence without contexts

Techniques for HOT languages

Conclusions

Contextual Equivalences for HOT Programming Languages

First-class functions.
Types: higher-order, polymorphic, recursive.
$+$ local mutable state, modules, objects, concurrency, proof search, ...

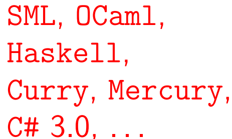
What are they and why study them?

Contextual equivalence without contexts

Techniques for HOT languages

Conclusions

Contextual Equivalences for HOT Programming Languages



SML, OCaml,
Haskell,
Curry, Mercury,
C# 3.0, ...

What are they and why study them?

Contextual equivalence without contexts

Techniques for HOT languages

Conclusions

Plan

What are they and why study them?

Contextual equivalence without contexts

Techniques for HOT languages

Conclusions

Plan

What are they and why study them?

Contextual equivalence without contexts

Techniques for HOT languages

Conclusions

Contextual equivalence

Two phrases of a programming language are ("Morris style") contextually equivalent ($\cong_{\mathrm{ctx}}$) 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?

$z : \mathtt{int} \vdash H : \mathtt{int} \longrightarrow \mathtt{int}$	$z : \mathtt{int} \vdash K : \mathtt{int} \longrightarrow \mathtt{int}$
$H \triangleq$	$K \triangleq$
$\mathtt{let}\ a = \mathtt{ref}\ z\ \mathtt{in}$	$\mathtt{let}\ b = \mathtt{ref}\,(-z)\ \mathtt{in}$
$\mathtt{fun}\ x \longrightarrow a := !a + x\ ;$	$\mathtt{fun}\ y \longrightarrow b := !b - y\ ;$
$!a$	$-(!b)$

Are these OCaml expressions contextually equivalent?

$z : \mathtt{int} \vdash H : \mathtt{int} \longrightarrow \mathtt{int}$	$z : \mathtt{int} \vdash K : \mathtt{int} \longrightarrow \mathtt{int}$
$H \triangleq$ $\mathtt{let}\ a = \mathtt{ref}\ z\ \mathtt{in}$ $\mathtt{fun}\ x \longrightarrow a := !a + x\ ;$ $!a$	$K \triangleq$ $\mathtt{let}\ b = \mathtt{ref}(-z)\ \mathtt{in}$ $\mathtt{fun}\ y \longrightarrow b := !b - y\ ;$ $-(!b)$

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

$$\begin{aligned} & \exists s'.\ \langle s, C[H] \rangle \ \longrightarrow^* \ \langle s', \mathtt{true} \rangle \\ \Leftrightarrow\ & \exists s''.\ \langle s, C[K] \rangle \ \longrightarrow^* \ \langle s'', \mathtt{true} \rangle \end{aligned}$$

Are these OCaml expressions contextually equivalent?

$$\frac{z : \mathtt{int} \vdash H : \mathtt{int} \longrightarrow \mathtt{int} \quad | \quad z : \mathtt{int} \vdash K : \mathtt{int} \longrightarrow \mathtt{int}}{}$$

$H \triangleq$

$\mathtt{let}\ a = \mathtt{ref}\ z\ \mathtt{in}$

$\mathtt{fun}\ x \longrightarrow a := !a + x\ ;$

$!a$

an OCaml syntax tree with some subtrees *within the scope of a binding for z*, replaced by the placeholder $-$

$(!b)$

Yes, $z : \mathtt{int} \vdash H \cong_{\mathrm{ctx}} K : \mathtt{int} \longrightarrow \mathtt{int}$, in the sense that for all states s and all well-typed, closing contexts $C[-]$,

$$\begin{aligned} & \exists s'.\ \langle s, C[H] \rangle \longrightarrow^* \langle s', \mathtt{true} \rangle \\ \Leftrightarrow\ & \exists s''.\ \langle s, C[K] \rangle \longrightarrow^* \langle s'', \mathtt{true} \rangle \end{aligned}$$

Are these OCaml expressions contextually equivalent?

$z : \mathtt{int} \vdash H : \mathtt{int} \longrightarrow \mathtt{int}$	$z : \mathtt{int} \vdash K : \mathtt{int} \longrightarrow \mathtt{int}$
$H \triangleq$ $\mathtt{let}\ a = \mathtt{ref}\ z\ \mathtt{in}$ $\mathtt{fun}\ x \longrightarrow a := !a + x\ ;$ $!a$	$K \triangleq$ $\mathtt{let}\ b = \mathtt{ref}(-z)\ \mathtt{in}$ $\mathtt{fun}\ y \longrightarrow b := !b - y\ ;$ $-(!b)$

Yes, $z : \mathtt{int} \vdash H \cong_{\mathrm{ctx}} K : \mathtt{int} \longrightarrow \mathtt{int}$, in the sense that

for all states s and all well-typed, closing contexts $C[-]$,

$$\begin{array}{rl} & \exists s'.\ \langle s, C[H] \rangle \longrightarrow^* \langle s', \mathtt{true} \rangle \\ \Leftrightarrow & \exists s''.\ \langle s, C[K] \rangle \longrightarrow^* \langle s'', \mathtt{true} \rangle \end{array}$$

OCaml structural operational semantics (mythical!):
transition relation between ⟨state, expression⟩-pairs.

Contextual equivalences

Two phrases of a programming language are ("Morris style") contextually equivalent ($\cong_{\mathbf{ctx}}$) 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.

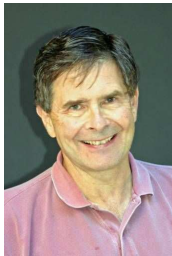
Contextual equivalence

Two phrases of a programming language are ("Morris style") contextually equivalent ($\cong_{\mathbf{ctx}}$) 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.

Contextual equivalence

Two phrases of a programming language are ("Morris style") contextually equivalent ($\cong_{\mathrm{ctx}}$) 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.



first known CS occurrence of this notion in Jim Morris' PhD thesis, *Lambda Calculus Models of Programming Languages* (MIT, 1969)

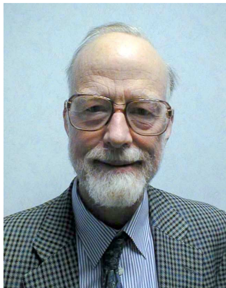
Contextual equivalence

Two phrases of a programming language are ("Morris style") contextually equivalent ($\cong_{\mathbf{ctx}}$) 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...

Wise words



"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."

Tony Hoare, Algebra and models. In *Computing Tomorrow. Future research directions in computer science*, Chapter 9, pp 158–187. (Cambridge University Press, 1996).

Wise words

IGNORE!

“But once feasibility has been
checked by an operational model,
operational reasoning should be
[illegible]mmediately abandoned; it is
[illegible]ntial that all subsequent
[illegible]ing, calculation and design
[illegible] conducted in each case
[illegible]t possible level of
[illegible]

Tony Hoare, Algebra and models. In *Comp*[illegible]*. Future research directions in computer science*, Chapt[illegible]58–187. (Cambridge University Press, 1996).

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 $\Rightarrow$ programs can make use of constituent phrases in dynamically complicated ways
  `:-(`

Why contextual equivalence matters

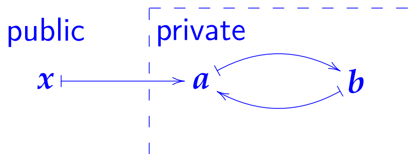
What is special about HOT languages?

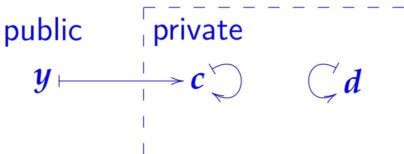
- type-directed "laws" for contextual equivalence
  :-)
- higher-order types ⇒ programs can make use of constituent phrases in dynamically complicated ways
  :-(

e.g. Extensionality property for function types...

Are these OCaml expressions contextually equivalent?

$F : \texttt{unit ref} \longrightarrow \texttt{unit ref}$	$G : \texttt{unit ref} \longrightarrow \texttt{unit ref}$
$F \triangleq$	$G \triangleq$
$\texttt{let}\ a = \texttt{ref ()}\ \texttt{in}$	$\texttt{let}\ c = \texttt{ref ()}\ \texttt{in}$
$\texttt{let}\ b = \texttt{ref ()}\ \texttt{in}$	$\texttt{let}\ d = \texttt{ref ()}\ \texttt{in}$
$\texttt{fun}\ x \longrightarrow \texttt{if}\ x == a\ \texttt{then}\ b$	$\texttt{fun}\ y \longrightarrow \texttt{if}\ y == d\ \texttt{then}\ d$
$\texttt{else}\ a$	$\texttt{else}\ c$





Are these OCaml expressions contextually equivalent?

$F : \texttt{unit ref} \longrightarrow \texttt{unit ref}$	$G : \texttt{unit ref} \longrightarrow \texttt{unit ref}$
$F \triangleq$ $\texttt{let}\ a = \texttt{ref}\,()\ \texttt{in}$ $\texttt{let}\ b = \texttt{ref}\,()\ \texttt{in}$ $\texttt{fun}\ x \longrightarrow \texttt{if}\ x == a\ \texttt{then}\ b$ $\texttt{else}\ a$	$G \triangleq$ $\texttt{let}\ c = \texttt{ref}\,()\ \texttt{in}$ $\texttt{let}\ d = \texttt{ref}\,()\ \texttt{in}$ $\texttt{fun}\ y \longrightarrow \texttt{if}\ y == d\ \texttt{then}\ d$ $\texttt{else}\ c$

No

For $T \triangleq \texttt{fun}\ f \longrightarrow \texttt{let}\ x = \texttt{ref}\,()\ \texttt{in}\ f(f\,x) == f\,x$,

$T\,F$ has value $\texttt{false}$, whereas $T\,G$ has value $\texttt{true}$,

so $F \not\cong_{\text{ctx}} G$.

Are these OCaml expressions contextually equivalent?

$z : \texttt{int} \vdash H : \texttt{int} \longrightarrow \texttt{int}$	$z : \texttt{int} \vdash K : \texttt{int} \longrightarrow \texttt{int}$
$H \triangleq$	$K \triangleq$
$\texttt{let}\ a = \texttt{ref}\ z\ \texttt{in}$	$\texttt{let}\ b = \texttt{ref}(-z)\ \texttt{in}$
$\texttt{fun}\ x \longrightarrow a := !a + x\ ;$	$\texttt{fun}\ y \longrightarrow b := !b - y\ ;$
$!a$	$-(!b)$

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

$$\exists s'.\ \langle s, C[H] \rangle \ \longrightarrow^* \langle s', \texttt{true} \rangle$$
$$\Leftrightarrow \exists s''.\ \langle s, C[K] \rangle \ \longrightarrow^* \langle s'', \texttt{true} \rangle$$

How does one prove such statements?

Are these OCaml expressions contextually equivalent?

$z : \mathtt{int} \vdash H : \mathtt{int} \longrightarrow \mathtt{int}$	$z : \mathtt{int} \vdash K : \mathtt{int} \longrightarrow \mathtt{int}$
$H \triangleq$	$K \triangleq$
$\mathtt{let}\ a = \mathtt{ref}\ z\ \mathtt{in}$	$\mathtt{let}\ b = \mathtt{ref}(-z)\ \mathtt{in}$
$\mathtt{fun}\ x \longrightarrow a := !a + x\ ;$	$\mathtt{fun}\ y \longrightarrow b := !b - y\ ;$
$!a$	$-(!b)$

Yes, $z : \mathtt{int} \vdash H \cong_{\mathrm{ctx}} K : \mathtt{int} \longrightarrow \mathtt{int}$, in the sense that

for all states s and all well-typed, closing contexts $C[-]$,

$$\begin{aligned} & \exists s'.\ \langle s, C[H] \rangle \ \longrightarrow^* \ \langle s', \mathtt{true} \rangle \\ \Leftrightarrow\ & \exists s''.\ \langle s, C[K] \rangle \ \longrightarrow^* \ \langle s'', \mathtt{true} \rangle \end{aligned}$$

How does one prove such statements?

these cause difficulty

Formalizing $\cong_{\mathbf{ctx}}$ without contexts

Contexts are too concrete

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

$$\left(\begin{array}{l}\texttt{let}\ \ a = \texttt{ref}\ \mathbf{0}\ \texttt{in} \\ \texttt{fun}\ x \longrightarrow \\ \quad a := !a + x\ ; \\ \quad !a\end{array}\right) = \left(\begin{array}{l}\texttt{let} \\ \quad a = \texttt{ref}\ \mathbf{0} \\ \texttt{in} \\ \quad \texttt{fun}\ x \longrightarrow \\ \qquad a := !a + x\ ; \\ \qquad !a\end{array}\right)$$

Contexts are too concrete

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

$$
\left(\begin{array}{l}
\texttt{let}\ \ a = \texttt{ref}\ \mathbf{0}\ \texttt{in} \\
\texttt{fun}\ x \longrightarrow \\
\quad a := !a + x\ ; \\
\quad !a
\end{array}\right)
=_\alpha
\left(\begin{array}{l}
\texttt{let} \\
\quad b = \texttt{ref}\ \mathbf{0} \\
\texttt{in} \\
\quad \texttt{fun}\ y \longrightarrow \\
\qquad b := !b + y\ ; \\
\qquad !b
\end{array}\right)
$$

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

Contexts are too concrete

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

So it pays to formulate program equivalences using mathematical notions that respect α-equivalence.

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

Contexts are too concrete

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

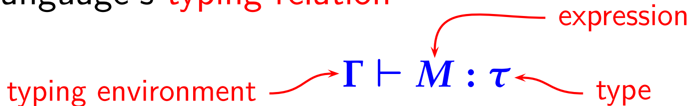
So it pays to formulate program equivalences using mathematical notions that respect α-equivalence.

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

$$
\begin{array}{lrcl}
 & \texttt{fun}\ x \longrightarrow (-) & =_\alpha & \texttt{fun}\ y \longrightarrow (-) \\
\text{and} & x & =_\alpha & x \\
\text{but} & \texttt{fun}\ x \longrightarrow x & \neq_\alpha & \texttt{fun}\ y \longrightarrow x
\end{array}
$$

Expression relations

Language's typing relation



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 (Γ, M, M', τ) satisfying:

$$(\Gamma \vdash M \mathrel{\mathcal{E}} M' : \tau) \;\Rightarrow\; (\Gamma \vdash M : \tau) \;\&\; (\Gamma \vdash M' : \tau)$$

Operations on expression relations

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

$$\frac{\Gamma \vdash M \; \mathcal{E}_1 \; M' : \tau \qquad \Gamma \vdash M' \; \mathcal{E}_2 \; M'' : \tau}{\Gamma \vdash M \; (\mathcal{E}_1 ; \mathcal{E}_2) \; M'' : \tau}$$

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

$$\frac{\Gamma \vdash M \; \mathcal{E} \; M' : \tau}{\Gamma \vdash M' \; \mathcal{E}^\circ \; M : \tau}$$

Identity $\mathcal{I}d$:

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash M \; \mathcal{I}d \; M : \tau}$$

Operations on expression relations

Compatible refinement $\mathcal{E} \mapsto \widehat{\mathcal{E}}$:

$$\frac{\Gamma \vdash M_1 : \tau \longrightarrow \tau' \qquad M_2 : \tau}{\Gamma \vdash M_1\, M_2 : \tau'}$$

Operations on expression relations

Compatible refinement $\mathcal{E} \mapsto \widehat{\mathcal{E}}$:

$$\frac{\Gamma \vdash M_1 \ \mathcal{E} \ M_1' : \tau \to \tau' \qquad \Gamma \vdash M_2 \ \mathcal{E} \ M_2' : \tau}{\Gamma \vdash M_1 \, M_2 \ \widehat{\mathcal{E}} \ M_1' M_2' : \tau'}$$

Operations on expression relations

Compatible refinement $\mathcal{E} \mapsto \widehat{\mathcal{E}}$:

$$\frac{\Gamma \vdash M_1 \; \mathcal{E} \; M_1' : \tau \to \tau' \qquad \Gamma \vdash M_2 \; \mathcal{E} \; M_2' : \tau}{\Gamma \vdash M_1 \, M_2 \; \widehat{\mathcal{E}} \; M_1' M_2' : \tau'}$$

$$\frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash (\mathtt{fun}\, x \to M) : \tau \to \tau'}$$

Operations on expression relations

Compatible refinement $\mathcal{E} \mapsto \widehat{\mathcal{E}}$:

$$\frac{\Gamma \vdash M_1 \; \mathcal{E} \; M_1' : \tau \to \tau' \qquad \Gamma \vdash M_2 \; \mathcal{E} \; M_2' : \tau}{\Gamma \vdash M_1 \, M_2 \; \widehat{\mathcal{E}} \; M_1' M_2' : \tau'}$$

$$\frac{\Gamma, x : \tau \vdash M \; \mathcal{E} \; M' : \tau'}{\Gamma \vdash (\mathtt{fun}\, x \to M) \; \widehat{\mathcal{E}} \; (\mathtt{fun}\, x \to M') : \tau \to \tau'}$$

Operations on expression relations

Compatible refinement $\mathcal{E} \mapsto \widehat{\mathcal{E}}$:

$$\frac{\Gamma \vdash M_1 \; \mathcal{E} \; M_1' : \tau \longrightarrow \tau' \qquad \Gamma \vdash M_2 \; \mathcal{E} \; M_2' : \tau}{\Gamma \vdash M_1 \, M_2 \; \widehat{\mathcal{E}} \; M_1' M_2' : \tau'}$$

$$\frac{\Gamma, x : \tau \vdash M \; \mathcal{E} \; M' : \tau'}{\Gamma \vdash (\mathtt{fun}\, x \longrightarrow M) \; \widehat{\mathcal{E}} \; (\mathtt{fun}\, x \longrightarrow M') : \tau \longrightarrow \tau'}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathtt{ref}\, M : \tau \, \mathtt{ref}}$$

Operations on expression relations

Compatible refinement $\mathcal{E} \mapsto \widehat{\mathcal{E}}$:

$$\frac{\Gamma \vdash M_1 \; \mathcal{E} \; M_1' : \tau \longrightarrow \tau' \qquad \Gamma \vdash M_2 \; \mathcal{E} \; M_2' : \tau}{\Gamma \vdash M_1 \, M_2 \; \widehat{\mathcal{E}} \; M_1' M_2' : \tau'}$$

$$\frac{\Gamma, x : \tau \vdash M \; \mathcal{E} \; M' : \tau'}{\Gamma \vdash (\mathtt{fun}\, x \longrightarrow M) \; \widehat{\mathcal{E}} \; (\mathtt{fun}\, x \longrightarrow M') : \tau \longrightarrow \tau'}$$

$$\frac{\Gamma \vdash M \; \mathcal{E} \; M' : \tau}{\Gamma \vdash \mathtt{ref}\, M \; \widehat{\mathcal{E}} \; \mathtt{ref}\, M' : \tau \, \mathtt{ref}}$$

etc, etc (one rule for each typing rule)

Contextual equiv. without contexts

Theorem [Gordon, Lassen (1998)]
$\cong_{\mathbf{ctx}}$ (defined conventionally, using contexts) is the greatest compatible & adequate expression relation.

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

- compatible if $\widehat{\mathcal{E}} \subseteq \mathcal{E}$

Contextual equiv. without contexts

Theorem [Gordon, Lassen (1998)]
$\cong_{\mathrm{ctx}}$ (defined conventionally, using contexts) is the greatest compatible & adequate expression relation.

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

- compatible if $\widehat{\mathcal{E}} \subseteq \mathcal{E}$
- adequate if $\emptyset \vdash M \mathrel{\mathcal{E}} M' : \mathtt{bool} \;\Rightarrow\; \forall s.$
  $$(\exists s'. \langle s, M \rangle \longrightarrow^* \langle s', \mathtt{true} \rangle) \Leftrightarrow (\exists s''. \langle s, M' \rangle \longrightarrow^* \langle s'', \mathtt{true} \rangle)$$

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
$\cong_{\mathrm{ctx}}$ is the union of all expression relations that are compatible and adequate.

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

- compatible if $\widehat{\mathcal{E}} \subseteq \mathcal{E}$
- adequate if $\emptyset \vdash M \; \mathcal{E} \; M' : \mathtt{bool} \;\Rightarrow\; \forall s.$
$$(\exists s'.\langle s, M\rangle \longrightarrow^{*} \langle s', \mathtt{true}\rangle) \Leftrightarrow (\exists s''.\langle s, M'\rangle \longrightarrow^{*} \langle s'', \mathtt{true}\rangle)$$

So defined, $\cong_{\mathrm{ctx}}$ is also reflexive ($\mathcal{I}d \subseteq \mathcal{E}$), symmetric ($\mathcal{E}^{\circ} \subseteq \mathcal{E}$) and transitive ($\mathcal{E} \,;\, \mathcal{E} \subseteq \mathcal{E}$).

Techniques

	sound	complete	useful	general
Brute force				
CIU				
Domains				
Games				
Logical relns				
Bisimulations				
Program logics				

sound: determines a compatible and adequate expression relation

complete: characterises $\cong_{\mathbf{ctx}}$

useful: for proving programming "laws" & PL correctness properties

general: what PL features can be dealt with?

	sound	complete	useful	general
Brute force	$+$	$+$	$-$	$(+)$
CIU				
Domains				
Games				
Logical relns				
Bisimulations				
Program logics				

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

(E.g. [AMP, POPL 2007].)

	sound	complete	useful	general
Brute force	+	+	−	(+)
CIU	+	+	−	+
Domains				
Games				
Logical relns				
Bisimulations				
Program logics				

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.

	sound	complete	useful	general
Brute force	$+$	$+$	$-$	$(+)$
CIU	$+$	$+$	$-$	$+$
Domains	$+$	$-$	$\pm$	$+$
Games	$+$	$+$	$\pm$	$-$
Logical relns				
Bisimulations				
Program logics				

Domains: traditional denotational semantics.

Games: game semantics [Abramsky, Malacaria, Hyland, Ong,...]

	sound	complete	useful	general
Brute force	$+$	$+$	$-$	$(+)$
CIU	$+$	$+$	$-$	$+$
Domains	$+$	$-$	$\pm$	$+$
Games	$+$	$+$	$\pm$	$-$
Logical relns	$+$	$\pm$	$+$	$-$
Bisimulations				
Program logics				

Logical relations: type-directed analysis of $\cong_{\mathbf{ctx}}$. 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,...].

	sound	complete	useful	general
Brute force	$+$	$+$	$-$	$(+)$
CIU	$+$	$+$	$-$	$+$
Domains	$+$	$-$	$\pm$	$+$
Games	$+$	$+$	$\pm$	$-$
Logical relns	$+$	$\pm$	$+$	$-$
Bisimulations	$+$	$\pm$	$+$	$+$
Program logics				

Bisimulations—the legacy of concurrency theory:

$$M_1 \quad \sim \quad M_2$$

$$M_1 \xrightarrow{T} M_1'$$

	sound	complete	useful	general
Brute force	+	+	−	(+)
CIU	+	+	−	+
Domains	+	−	±	+
Games	+	+	±	−
Logical relns	+	±	+	−
Bisimulations	+	±	+	+
Program logics				

Bisimulations—the legacy of concurrency theory:

$$\begin{array}{ccc} M_1 & \sim & M_2 \\ {\scriptstyle T}\downarrow & & {\scriptstyle T}\downarrow \\ M_1' & \sim & M_2' \end{array} \quad \text{(and symmetrically)}$$

	sound	complete	useful	general
Brute force	+	+	−	(+)
CIU	+	+	−	+
Domains	+	−	±	+
Games	+	+	±	−
Logical relns	+	±	+	−
Bisimulations	+	±	+	+
Program logics				

Bisimulations—the legacy of concurrency theory:

- applicative [Abramsky, Gordon, AMP]
- environmental [Pierce-Sumii-Koutavas-Wand]
- "up-to" techniques [Sangiorgi, Lassen]

	sound	complete	useful	general
Brute force	$+$	$+$	$-$	$(+)$
CIU	$+$	$+$	$-$	$+$
Domains	$+$	$-$	$\pm$	$+$
Games	$+$	$+$	$\pm$	$-$
Logical relns	$+$	$\pm$	$+$	$-$
Bisimulations	$+$	$\pm$	$+$	$+$
Program logics	$+$	$+$	$\pm$	$-$

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

[Berger-Honda-Yoshida]

Beyond universal identities.

	sound	complete	useful	general
Brute force	$+$	$+$	$-$	$(+)$
CIU	$+$	$+$	$-$	$+$
Domains	$+$	$-$	$\pm$	$+$
Games	$+$	$+$	$\pm$	$-$
Logical relns	$+$	$\pm$	$+$	$-$
Bisimulations	$+$	$\pm$	$+$	$+$
Program logics	$+$	$+$	$\pm$	$-$

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

	sound	complete	useful	general
Brute force	$+$	$+$	$-$	$(+)$
CIU	$+$	$+$	$-$	$+$
Domains	$+$	$-$	$\pm$	$+$
Games	$+$	$+$	$\pm$	$-$
Logical relns	$+$	$\pm$	$+$	$-$
Bisimulations	$+$	$\pm$	$+$	$+$
Program logics	$+$	$+$	$\pm$	$-$

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:

$$\emptyset \vdash \Lambda\alpha.e_1 \cong_{\mathrm{ctx}} \Lambda\alpha.e_2 : \forall\alpha.\tau$$

if and only if

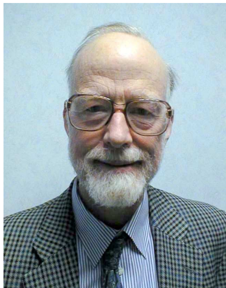
for all τ_1, τ_2 and all "good" relations $\tau_1 \overset{r}{\longleftrightarrow} \tau_2$,

$e_1[\tau_1/\alpha]$ and $e_2[\tau_2/\alpha]$ are related by $\tau[\tau_1/\alpha] \overset{\tau[r/\alpha]}{\longleftrightarrow} \tau[\tau_2/\alpha]$

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].

Wise words



"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."

Tony Hoare, Algebra and models. In *Computing Tomorrow. Future research directions in computer science*, Chapter 9, pp 158–187. (Cambridge University Press, 1996).

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

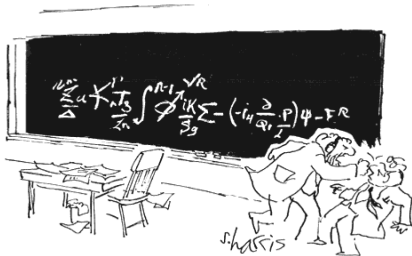
- The development of programming language theory based on contextual equivalences lags far behind the development of programming language design.

  Type soundness results are two a penny, but correctness properties up to $\cong_{\mathbf{ctx}}$ 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 $\cong_{\mathbf{ctx}}$ 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