Techniques

for proving 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.

Contexts are too concrete

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

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

Contexts are too concrete

The semantics of programs only depends on their abstract syntax (parse trees) modulo renaming of bound identifiers (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 $=_\alpha$:

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

Expression relations

Language's typing relation

expression

typing environment → $\boldsymbol{\Gamma \vdash M : \tau}$ ← type

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:

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

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

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

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

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

Operations on expression relations

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

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

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

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

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 ($\mathcal{E} \,;\mathcal{E} \subseteq \mathcal{E}$).

	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 $M \cong_{\mathrm{ctx}} M'$.

(E.g. [AMP + Shinwell, LMCS 4(1:4) 2008].)

	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	+	−	±	+
Games	+	+	±	−
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	+	−	±	+
Games	+	+	±	−
Logical relns	+	±	+	−
Bisimulations	+	±	+	+
Program logics				

Bisimulations—the legacy of concurrency theory:

$$\begin{array}{ccc} M_1 & \sim & M_2 \\ \tau \big\downarrow & & \\ M_1' & & \end{array}$$

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

Bisimulations—the legacy of concurrency theory:

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

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

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

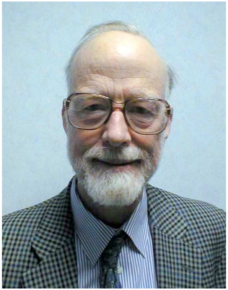
for all τ_1, τ_2 and all "good" relations $\tau_1 \overset{r}{\longleftrightarrow} \tau_2$,

$e_1[\tau_1/\alpha]$ 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).

Conclusion

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

Research opportunities

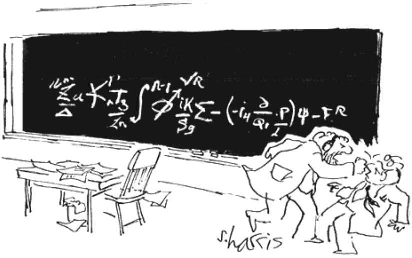
- The development of programming language theory based on contextual equivalences lags far behind the development of programming language design.

  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!"