# Introduction:
# Contextual Equivalence

# Styles of PL semantics

- Program logics
  - basis for verification
  - validated by operational/denotational semantics
- Denotational semantics
  - foundations and structure
  - involves sophisticated mathematics
- Operational semantics
  - basis for implementation
  - involves (deceptively) simple mathematics

The three approaches are inter-linked.

# Styles of semantics

- Program logics
- Denotational semantics
- Operational semantics

Let's compare their answers to a fundamental question:

When are two program phrases equal?

# When are two program phrases semantically equal?

# When are two program phrases semantically equal?

**Program Logic:**

when they satisfy the same logical assertions.

E.g. $C \cong C'$ iff for all pre-, post-conditions $P$, $Q$

$$\{P\}\, C\, \{Q\} \Leftrightarrow \{P\}\, C'\, \{Q\}$$

# When are two program phrases semantically equal?

**Program Logic:**

when they satisfy the same logical assertions.

**Denotational semantics:**

when they have equal denotations.

# When are two program phrases semantically equal?

**Program Logic:**

when they satisfy the same logical assertions.

**Denotational semantics:**

when they have equal denotations.

**Operational semantics:**

when they are contextually equivalent.

# 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 an operational semantics as part of its definition

# Contextual equivalences

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.

Different choices lead to possibly different notions of contextual equivalence.

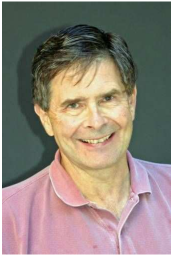
# Contextual equivalence

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.

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 ($\cong_{\mathbf{ctx}}$) 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.



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

# Contextual Equivalence for HOT Programming Languages

# Contextual Equivalence for HOT Programming Languages

First-class functions.
Types: higher-order, polymorphic, recursive.

+ local mutable state, modules, objects, concurrency, proof search, . . .

# Contextual Equivalence for HOT Programming Languages

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

# Are these OCaml expressions contextually equivalent?

$$\begin{array}{l}\texttt{let}\ a = \texttt{ref}\ n\ \texttt{in} \\ \texttt{fun}\ x \rightarrow a := !a + x\,; \\ \qquad\qquad !a\end{array}$$

$$\begin{array}{l}\texttt{let}\ b = \texttt{ref}\,(-n)\ \texttt{in} \\ \texttt{fun}\ y \rightarrow b := !b - y\,; \\ \qquad\qquad -(!b)\end{array}$$

Are these OCaml expressions contextually equivalent?

$\texttt{let}\ a = \texttt{ref}\ n\ \texttt{in}$
$\texttt{fun}\ x \to a := !a + x\,;$
$\quad !a$

$\texttt{let}\ b = \texttt{ref}(-n)\ \texttt{in}$
$\texttt{fun}\ y \to b := !b - y\,;$
$\quad -(!b)$

Etymology of "OCaml" (`caml.inria.fr/ocaml`):

ML = "Meta-language" (for LCF prover)

## Are these OCaml expressions contextually equivalent?

$$\begin{array}{c}\texttt{let}\ a = \texttt{ref}\ n\ \texttt{in} \\ \texttt{fun}\ x \rightarrow a := !a + x\,; \\ !a\end{array}$$

$$\begin{array}{c}\texttt{let}\ b = \texttt{ref}\,(-n)\ \texttt{in} \\ \texttt{fun}\ y \rightarrow b := !b - y\,; \\ -(!b)\end{array}$$

Etymology of "OCaml" (`caml.inria.fr/ocaml`):

CAM = "Categorial Abstract Machine"

Are these OCaml expressions contextually equivalent?

$$\begin{array}{l}\texttt{let}\ a = \texttt{ref}\ n\ \texttt{in} \\ \texttt{fun}\ x \longrightarrow a := !a + x\,; \\ \qquad\qquad !a\end{array}$$

$$\begin{array}{l}\texttt{let}\ b = \texttt{ref}\,(-n)\ \texttt{in} \\ \texttt{fun}\ y \longrightarrow b := !b - y\,; \\ \qquad\qquad -(!b)\end{array}$$

Etymology of "OCaml" (`caml.inria.fr/ocaml`):

O = "Objective"

# Are these OCaml expressions contextually equivalent?

$$\texttt{let } a = \texttt{ref } n \texttt{ in}$$
$$\texttt{fun } x \to a := !a + x\,;\ !a$$

$$\texttt{let } b = \texttt{ref}(-n) \texttt{ in}$$
$$\texttt{fun } y \to b := !b - y\,;\ -(!b)$$

expressions of type
int → int

Are these OCaml expressions contextually equivalent?

$$H \triangleq$$
$$\texttt{let}\ a = \texttt{ref}\ n\ \texttt{in}$$
$$\texttt{fun}\ x \rightarrow a := !a + x\,;$$
$$!a$$

$$K \triangleq$$
$$\texttt{let}\ b = \texttt{ref}\,(-n)\ \texttt{in}$$
$$\texttt{fun}\ y \rightarrow b := !b - y\,;$$
$$-(!b)$$

Yes, $H \cong_{\mathrm{ctx}} K$, in the sense that

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

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

Are these OCaml expressions contextually equivalent?

$$H \triangleq \texttt{let}\ a = \texttt{ref}\ n\ \texttt{in}\ \texttt{fun}\ x \to a := !a + x\,;\ !a$$

$$K \triangleq \texttt{let}\ b = \texttt{ref}(-n)\ \texttt{in}\ \texttt{fun}\ y \to b := !b - y\,;\ -(!b)$$

Yes, $H \cong_{\text{ctx}} K$, in the sense that

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

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

an OCaml syntax tree with some subtrees within the scope of a binding for n replaced by the placeholder "−"

Are these OCaml expressions contextually equivalent?

$$H \triangleq$$
$$\texttt{let}\ a = \texttt{ref}\ n\ \texttt{in}$$
$$\texttt{fun}\ x \to a := !a + x\,;$$
$$!a$$

$$K \triangleq$$
$$\texttt{let}\ b = \texttt{ref}(-n)\ \texttt{in}$$
$$\texttt{fun}\ y \to b := !b - y\,;$$
$$-(!b)$$

Yes, $H \cong_{\mathrm{ctx}} K$, in the sense that

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

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

Ocaml operational semantics

# Why contextual equivalence matters

- Philosophically important:
  operational behaviour is a characteristic feature of programming language semantics 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?

> "When one attempts to combine language concepts, unexpected and counterintuitive interactions arise. At this point, even the most experienced designer's intuition must be butressed by a rigorous definition of what the language means."
>
> *John Reynolds, 1990*

# 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
  :-(

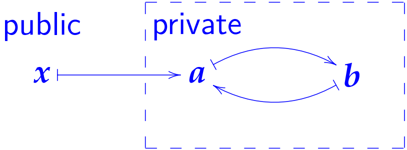
e.g. Extensionality property for function types:

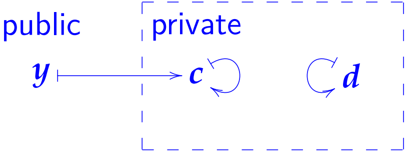
$$e_1 \cong_{\mathrm{ctx}} e_2 : \tau \to \tau' \;\Leftrightarrow$$
$$(\forall e : \tau)\; e_1\, e \cong_{\mathrm{ctx}} e_2\, e : \tau'$$

# Are these OCaml expressions contextually equivalent?

```
let a = ref () in
let b = ref () in
fun x →
if  x == a then b
else a
```

```
let c = ref () in
let d = ref () in
fun y →
if  y == d then d
else c
```



# Are these OCaml expressions contextually equivalent?

$F \triangleq$
let $a$ = ref () in
let $b$ = ref () in
fun $x \rightarrow$
if $x == a$ then $b$
else $a$

$G \triangleq$
let $c$ = ref () in
let $d$ = ref () in
fun $y \rightarrow$
if $y == d$ then $d$
else $c$

No!

For $T \triangleq$ fun $f \rightarrow$ let $x$ = ref () in $f(f\,x) == f\,x$,

$T\,F$ has value false, whereas $T\,G$ has value true,

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

Are these OCaml expressions contextually equivalent?

$$H \triangleq \texttt{let}\ a = \texttt{ref}\ n\ \texttt{in}\ \texttt{fun}\ x \to a := !a + x\,;\ !a$$

$$K \triangleq \texttt{let}\ b = \texttt{ref}\,(-n)\ \texttt{in}\ \texttt{fun}\ y \to b := !b - y\,;\ -(!b)$$

Yes, $H \cong_{\mathrm{ctx}} K$, in the sense that

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

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

How does one prove such statements?

Are these OCaml expressions contextually equivalent?

$$H \triangleq \texttt{let}\ a = \texttt{ref}\ n\ \texttt{in}\ \texttt{fun}\ x \rightarrow a := {!a} + x\,;\ {!a}$$

$$K \triangleq \texttt{let}\ b = \texttt{ref}(-n)\ \texttt{in}\ \texttt{fun}\ y \rightarrow b := {!b} - y\,;\ -({!b})$$

Yes, $H \cong_{\text{ctx}} K$, in the sense that

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

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

How does one prove such statements?

these cause difficulty