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.

Gottfried Wilhelm Leibniz (1646–1716): two mathematical objects are equal if there is no test to distinguish them.
• program \( \triangleq \) well-typed expression with no free identifiers

• executing program \( e \) in a given state \( s \) \( \triangleq \) finding \( (v,s) \) such that \( s,e \Rightarrow v,s \)

• observable results of execution, \( \text{obs}(v,s) \):
  \( \text{obs}(c,s) \triangleq c \) if \( c = \text{true}, \text{false}, \text{n}, () \)
  \( \text{obs}(v_1,v_2,s) \triangleq \text{obs}(v_1,s), \text{obs}(v_2,s) \)
  \( \text{obs}(\text{fun}(a:\text{ty}) \to e),s) \triangleq <\text{fun}> \)
  \( \text{obs}(\text{fun } f = (a:\text{ty}) \to e),s) \triangleq <\text{fun}> \)
  \( \text{obs}(l,s) \triangleq \{\text{contents} = n\} \) if \( (l \to n) \in s \)

• occurrence of an expression in a program...

---

**ML Contexts \( C[e] \)**

• ML syntax trees with a single sub-tree replaced by "hole", \(-\). E.g.
  \( \text{fun } (x:\text{int}) \to x+(-) \)

• \( C[e] \triangleq \) expression resulting from replacing hole \(-\) by \( e \) in context \( C \)
  E.g. When \( C[E] \) is \( \text{fun } (x:\text{int}) \to x+(-) \)
  then \( C[\text{ci}] \) is \( \text{fun } (x:\text{int}) \to x+x \)
ML Contexts \( C[e] \)

- ML syntax trees with a single sub-tree replaced by “hole”. E.g.
  \( \text{fun } (x : \text{int}) \rightarrow x + (-) \)

- \( C[e] \triangleq \) expression resulting from replacing hole - by \( e \) in context \( C \)
  E.g. When \( C[E] \) is \( \text{fun } (x : \text{int}) \rightarrow x + (-) \) then \( C[\alpha] \) is \( \text{fun } (x : \text{int}) \rightarrow x + x \) capture!

- so can’t identify contexts up to \( \alpha \)-equiv.
- complicates type assignment for contexts

ML Contextual Equivalence \( \Gamma \vdash e_1 =c_e e_2 : ty \)
is defined to hold if:

- \( \Gamma \vdash e_1 : ty \) and \( \Gamma \vdash e_2 : ty \)

- for all contexts \( C[-] \) such that \( C[e_1] & C[e_2] \) are programs, and for all states \( s \)
  if \( s, C[e_1] \Rightarrow v_1, s_1 \)
  then \( s, C[e_2] \Rightarrow v_2, s_2 \) with \( \text{obs}(v_1, s_1) = \text{obs}(v_2, s_2) \)
  and vice versa.
ML Contextual Equivalence $\Gamma \vdash e_1 =_{ctx} e_2 : ty$
is defined to hold if:

- $\Gamma \vdash e_1 : ty$ and $\Gamma \vdash e_2 : ty$
- for all contexts $C[-]$ such that $C[e_1]$ & $C[e_2]$ are programs, and for all states $s$
  - if $s, C[e_1] \Rightarrow v_1, s_1$
  - then $s, C[e_2] \Rightarrow v_2, s_2$ with $\text{obs}(v_1, s_1) = \text{obs}(v_2, s_2)$
  - and vice versa.

Simplifying assumptions:
- only consider closed expressions (can use $e[-/x]$) as contexts
- only observe termination (doesn’t change $=_{ctx}$)

Contextual preorder / equivalence

Given $e_1, e_2 \in \text{Prog}_{ty}$, define

\[
e_1 =_{ctx} e_2 : ty \triangleq e_1 \leq_{ctx} e_2 : ty \& e_2 \leq_{ctx} e_1 : ty
\]

\[
e_1 \leq_{ctx} e_2 : ty \triangleq \forall x, e, ty', s. \ (x : ty \vdash e : ty') \&
\]

\[
\quad s, e[e_1/x] \downarrow \supset s, e[e_2/x] \downarrow
\]

where $s, e \downarrow$ indicates termination:

\[
s, e \downarrow \triangleq \exists s', v. (s, e \Rightarrow v, s')
\]

Other natural choices of what to observe apart from termination do not change $=_{ctx}$.

(see Exercise B.3)
Definition of $\Downarrow$ is not syntax-directed

$\frac{s', e_2[v_1/x] \Downarrow}{s, \text{let } x = e_1 \text{ in } e_2 \Downarrow}$ if $s, e_1 \Rightarrow v_1, s'$

but $e_2[v_1/x]$ is not built from subphrases of $\text{let } x = e_1 \text{ in } e_2$.

Simple example of the difficulty this causes: consider a divergent integer expression $\bot \triangleq (\text{fun } f = (x : \text{int}) \Rightarrow f x) \ 0$.

It satisfies $\bot \leq_{\text{ctx}} n: \text{int}$, for any $n \in \text{Prog}_{\text{int}}$

Obvious strategy for proving this is to try to show

$$s, e \Downarrow \supset \forall x, e'. e = e'[\bot/x] \supset s, e'[n/x] \Downarrow$$

by induction on the derivation of $s, e \Downarrow$. But the induction steps are hard to carry out because of the above problem.

Felleisen-style presentation of $\rightarrow$

Lemma. $(s, e) \rightarrow (s', e')$ holds iff $e = \mathcal{E}[r]$ and $e' = \mathcal{E}[r']$ for some evaluation context $\mathcal{E}$ and basic reduction $(s, r) \rightarrow (s', r')$.

Evaluation contexts are closed contexts that want to evaluate their hole ($\mathcal{E} ::= - \mid \mathcal{E} e \mid v \mathcal{E} \mid \text{let } x = \mathcal{E} \text{ in } e \mid \cdots$).

$\mathcal{E}[r]$ denotes the expression resulting from replacing the ‘hole’ $[-]$ in $\mathcal{E}$ by the expression $r$.

Basic reductions $(s, r) \rightarrow (s', r')$ are the axioms in the inductive definition of $\rightarrow$ à la Plotkin—see Sect. A.5.

see (7) on p387 for full definition
**Fact.** Every closed expression not in canonical form is uniquely of the form $E[r]$ for some evaluation context $E$ and redex $r$.

**Fact.** Every evaluation context $E$ is a composition $\mathcal{F}_1[\mathcal{F}_2[\cdots \mathcal{F}_n[-] \cdots ]]$ of basic evaluation contexts, or evaluation frames.

Hence can reformulate transitions between configurations $(s, e) = (s, E_1[E_2[\cdots E_n[r] \cdot \cdot \cdot ]])$ in terms of transitions between configurations of the form

\[
\langle s, \mathcal{F}s, r \rangle
\]

where $\mathcal{F}s$ is a list of evaluation frames—the frame stack.

---

**An ML abstract machine**

Transitions

\[
\langle s, \mathcal{F}s, e \rangle \rightarrow \langle s', \mathcal{F}s', e' \rangle
\]

defined by cases (i.e. no induction), according to the structure of $e$ and (then) $\mathcal{F}s$, for example:

\[
\langle s, \mathcal{F}s, \text{let } x = e_1 \text{ in } e_2 \rangle \rightarrow \\
\langle s, \mathcal{F}s \circ (\text{let } x = [-] \text{ in } e_2), e_1 \rangle
\]

\[
\langle s, \mathcal{F}s \circ (\text{let } x = [-] \text{ in } e), v \rangle \rightarrow \langle s, \mathcal{F}s, e[v/x] \rangle
\]

(See Sect. A.6 for the full definition.)

Initial configurations: $\langle s, Id, e \rangle$

terminal configurations: $\langle s, Id, v \rangle$

($Id$ the empty frame stack, $v$ a closed canonical form).
Theorem. \( \langle s, \mathcal{F}s, e \rangle \rightarrow^* \langle s', \mathcal{I}d, v \rangle \) iff \( s, \mathcal{F}s[e] \Rightarrow v, s' \).

where
\[
\begin{align*}
\mathcal{I}d[e] & \triangleq e \\
(\mathcal{F}s \circ \mathcal{F})[e] & \triangleq \mathcal{F}s[\mathcal{F}[e]].
\end{align*}
\]

Hence:
\( s, e \downarrow \) iff \( \exists s', v \left( \langle s, \mathcal{I}d, e \rangle \rightarrow^* \langle s', \mathcal{I}d, v \rangle \right) \).

So we can express termination of evaluation in terms of termination of the abstract machine. The gain is the following simple, but key, observation:
\[
\nabla \triangleq \{ \langle s, \mathcal{F}s, e \rangle | \exists s', v \left( \langle s, \mathcal{F}s, e \rangle \rightarrow^* \langle s', \mathcal{I}d, v \rangle \right) \}
\]

has a direct, inductive definition following the structure of \( e \) and \( \mathcal{F}s \)—see Sect. A.7.
The relation we are interested in is a retract of a larger one with better structural properties.

\[
\downarrow \quad \leftarrow \quad \rightarrow \quad \leftarrow \quad \downarrow
\]

\[
\text{States} \times \text{Programs} \quad \leftrightarrow \quad \text{Frame Stacks} \times \text{Programs}
\]

\[
(s, e) \quad \rightarrow \quad \langle s, \operatorname{Id}, e \rangle
\]

\[
(s, \mathcal{F}s[e]) \quad \rightarrow \quad \langle s, \mathcal{F}s, e \rangle
\]