Overview

- *Contextual equivalence* of ML expressions in general, and of functions involving local state in particular.
- A brief tour of *structural operational semantics*, culminating in a structural definition of termination via an abstract machine using ‘frame stacks’.
- Applications to reasoning about contextual equivalence.
- Some things we do not know how to do yet.

Main point: A particular style of operational semantics enables a ‘syntax-directed’ inductive definition of termination that is very useful for reasoning about operational equivalence of programs.

$p \triangleq$

```
let a = ref 0 in
fun(x : int) -> (a := !a + x ; !a)
```

$m \triangleq$

```
let b = ref 0 in
fun(y : int) -> (b := !b - y ; 0 - !b)
```

Are these Caml expressions (of type `int -> int`) contextually equivalent?

Contextual equivalence (in general)

Two phrases of a programming language are contextually equivalent ($=_{\mathbf{ctx}}$) if any occurrences of the first phrase in a complete program can be replaced by the second phrase without affecting the observable results of executing the program.

Not a single notion: different choices can be made for the definitions of the underlined phrases, leading to possibly different notions of contextual equivalence.

Also known as *operational*, or *observational* equivalence.

$f \triangleq$

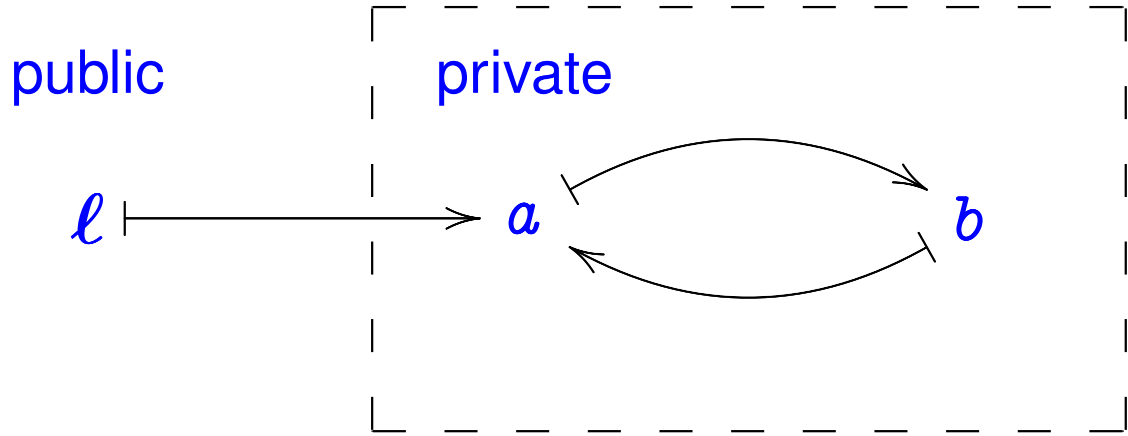
$$\begin{array}{l}\texttt{let } a = \texttt{ref } 0 \texttt{ in} \\ \texttt{let } b = \texttt{ref } 0 \texttt{ in} \\ \texttt{fun(}x : \texttt{int ref) -> if } x == a \texttt{ then } b \texttt{ else } a\end{array}$$

$g \triangleq$

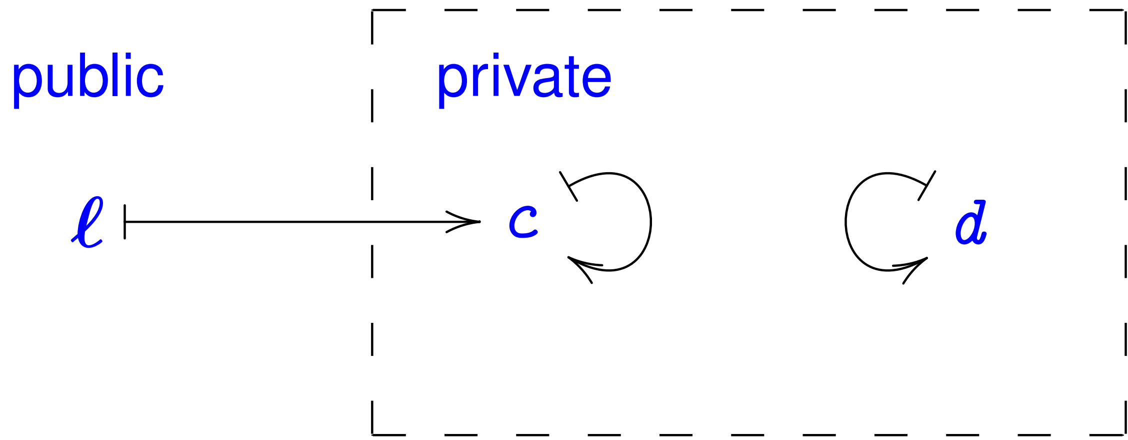
$$\begin{array}{l}\texttt{let } c = \texttt{ref } 0 \texttt{ in} \\ \texttt{let } d = \texttt{ref } 0 \texttt{ in} \\ \texttt{fun(}y : \texttt{int ref) -> if } y == d \texttt{ then } d \texttt{ else } c\end{array}$$

Are these Caml expressions (of type `int ref -> int ref`) contextually equivalent?

Picture for f:



Picture for g:



Function Extensionality Principle. Two functions (defined on the same set of arguments) are equal if they give equal results for each possible argument.

- True of mathematical functions (e.g. in set theory).
- False for ML function expressions in general.
- True for ML function expressions in canonical form (i.e. lambda abstractions), if we take ‘equal’ to mean contextually equivalent.
- True for pure functional programming languages (see Pitts 1997a); also true for languages with ‘block-structured’ local state à l’Algol (see Pitts 1997b).

Distinguishing F and G

```
# let f = ··· (as on Slide 4) ··· ;;
  val f : int ref -> int ref = <fun>
# let g = ··· (ditto) ··· ;;
  val g : int ref -> int ref = <fun>
# let t = fun(h : int ref -> int ref) ->
              let z = ref 0 in h (h z) == h z ;;
  val t : (int ref -> int ref) -> bool = <fun>
# t f ;;
- : bool = false
# t g ;;
- : bool = true
```

ML Evaluation Semantics (simplified, environment-free form)

Evaluation relation

$$\boxed{s, e \Rightarrow v, s'}$$

$$\begin{cases} s & = \text{initial state} \\ e & = \text{closed expression to be evaluated} \\ v & = \text{resulting closed canonical form} \\ s' & = \text{final state} \end{cases}$$

is inductively generated by rules *following the structure* of e, for example:

$$\frac{s, e_1 \Rightarrow v_1, s' \quad s', e_2[v_1/x] \Rightarrow v_2, s''}{s, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Rightarrow v_2, s''}$$

Evaluation semantics is also known as *big-step* (anon), *natural* (Kahn 1987), or *relational* (Milner) semantics.

ML programs are typed

Programs of type ty: $\boxed{\mathbf{Prog}_{ty}} \triangleq \{\, e \mid \emptyset \vdash e : ty \,\}$

where

Type assignment relation $\boxed{\Gamma \vdash e : ty}$ $\begin{cases} \Gamma & = \text{typing context} \\ e & = \text{expression to be typed} \\ ty & = \text{type} \end{cases}$

is inductively generated by axioms and rules *following the structure* of e, for example:

$$\frac{\Gamma \vdash e_1 : ty_1 \quad \Gamma[x \mapsto ty_1] \vdash e_2 : ty_2 \quad x \notin dom(\Gamma)}{\Gamma \vdash (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) : ty_2}$$

Theorem (Type Soundness). If $e, s \Rightarrow v, s'$ and $e \in \mathbf{Prog}_{ty}$, then $v \in \mathbf{Prog}_{ty}$.

Contextual preorder / equivalence

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

$$e_1 =_{\mathrm{ctx}} e_2 : ty \triangleq e_1 \leq_{\mathrm{ctx}} e_2 : ty \ \ \& \ \ e_2 \leq_{\mathrm{ctx}} e_1 : ty$$

$$\begin{aligned} e_1 \leq_{\mathrm{ctx}} e_2 : ty \triangleq\ & \forall x, e, ty', s \,.\, (x : ty \vdash e : ty') \ \ \& \\ & s, e[e_1/x]\Downarrow \ \supset \ s, e[e_2/x]\Downarrow \end{aligned}$$

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 $=_{\mathrm{ctx}}$.

Definition of $\Downarrow$ is not syntax-directed

E.g. $\dfrac{s', e_2[v_1/x] \Downarrow}{s, \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow}$ if $s, e_1 \Rightarrow v_1, s'$

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

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

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

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.

ML transition relation $\boxed{(s\ ,\ e) \rightarrow (s'\ ,\ e')}$

is inductively generated by rules following the structure of e—e.g.

a simplification step

$$\frac{(s\ ,\ e_1) \rightarrow (s'\ ,\ e'_1)}{(s\ ,\ \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightarrow (s'\ ,\ \mathtt{let}\ x = e'_1\ \mathtt{in}\ e_2)}$$

a basic reduction

$$\frac{v \text{ a canonical form}}{(s\ ,\ \mathtt{let}\ x = v\ \mathtt{in}\ e) \rightarrow (s\ ,\ e[v/x])}$$

(see Sect. A.5 for the full definition).

$\boxed{\textbf{Theorem.}\quad s, e \Rightarrow v, s' \quad \textit{iff} \quad (s\ ,\ e) \rightarrow^* (s'\ ,\ v).}$

($\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$.)

Felleisen-style presentation of $\longrightarrow$

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

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

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

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

Fact. Every closed expression not in canonical form is uniquely of the form $\mathcal{E}[r]$ for some evaluation context $\mathcal{E}$ and redex r.

Fact. Every evaluation context $\mathcal{E}$ 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 $(s\ ,\ e) = (s\ ,\ \mathcal{F}_1[\mathcal{F}_2[\cdots\mathcal{F}_n[r]\cdots]])$ in terms of transitions between configurations of the form

$$\boxed{\langle s\ ,\ \mathcal{F}s\ ,\ r\rangle}$$

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

An ML abstract machine

Transitions

$$\boxed{\langle s \;,\, \mathcal{F}s \;,\, e \rangle \longrightarrow \langle s' \;,\, \mathcal{F}s' \;,\, e' \rangle}$$

$$\begin{cases} s, s' & = \text{states} \\ \mathcal{F}s, \mathcal{F}s' & = \text{frame stacks} \\ e, e' & = \text{closed expressions} \end{cases}$$

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

$$\langle s \;,\, \mathcal{F}s \;,\, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \rangle \longrightarrow \langle s \;,\, \mathcal{F}s \circ (\mathtt{let}\ x = [-]\ \mathtt{in}\ e_2) \;,\, e_1 \rangle$$

$$\langle s \;,\, \mathcal{F}s \circ (\mathtt{let}\ x = [-]\ \mathtt{in}\ e) \;,\, v \rangle \longrightarrow \langle s \;,\, \mathcal{F}s \;,\, e[v/x] \rangle$$

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

Initial configurations: $\langle s \;,\, \mathcal{I}d \;,\, e \rangle$

terminal configurations: $\langle s \;,\, \mathcal{I}d \;,\, v \rangle$

($\mathcal{I}d$ the empty frame stack, v a closed canonical form).

$$\boxed{\textbf{Theorem. } \langle s\ ,\ \mathcal{F}s\ ,\ e\rangle \longrightarrow^{*} \langle s'\ ,\ \mathcal{I}d\ ,\ v\rangle \textit{ iff } s, \mathcal{F}s[e] \Rightarrow v, s'.}$$

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

Hence: $\boxed{s, e \Downarrow \textit{ iff } \exists s', v\ (\langle s\ ,\ \mathcal{I}d\ ,\ e\rangle \longrightarrow^{*} \langle s'\ ,\ \mathcal{I}d\ ,\ v\rangle).}$

So we can express termination of evaluation in terms of termination of the abstract machine. The gain is the following **simple, but key, observation:**

$$\searrow\ \triangleq \{\ \langle s\ ,\ \mathcal{F}s\ ,\ e\rangle \mid \exists s', v\ \big(\langle s\ ,\ \mathcal{F}s\ ,\ e\rangle \longrightarrow^{*} \langle s'\ ,\ \mathcal{I}d\ ,\ v\rangle\big)\ \}$$

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.

$$\begin{tikzcd} \Downarrow \arrow[r, hook, shift left] \arrow[d, hook] & \searrow \arrow[l, shift left] \arrow[d, hook] \\ \text{States} \times \text{Programs} \arrow[r, hook, shift left] & \text{States} \times \text{Frame Stacks} \times \text{Programs} \arrow[l, shift left] \end{tikzcd}$$

$$(s\ ,\ e) \longmapsto \langle s\ ,\ \mathcal{I}d\ ,\ e \rangle$$

$$(s\ ,\ \mathcal{F}s[e]) \longmapsfrom \langle s\ ,\ \mathcal{F}s\ ,\ e \rangle$$

‘Logical’ simulation relation between ML programs, parameterised by state-relations

For each state-relation $r \in \mathbf{Rel}(w_1, w_2)$ we can define relations

$$\boxed{e_1 \leq_r e_2 : ty} \qquad (e_1 \in \mathbf{Prog}_{ty}(w_1), e_2 \in \mathbf{Prog}_{ty}(w_2))$$

(for each type ty), with the properties stated on Slides 19–21.

Kripke-style worlds: $w_1, w_2, \ldots$ are finite sets of locations.

States in world w: $\boxed{\mathbf{St}(w)} \triangleq \mathbb{Z}^w$. Programs in world w:

$\boxed{\mathbf{Prog}_{ty}(w)} \triangleq \{ e \in \mathbf{Prog}_{ty} \mid loc(e) \subseteq w \}$.

State-relations: $r, r', \ldots \in \boxed{\mathbf{Rel}(w_1, w_2)}$ are subsets of $\mathbf{St}(w_1) \times \mathbf{St}(w_2)$.

The simulation property of $\leq_r$

To prove $e_1 \leq_r e_2 : ty$, it suffices to show that whenever

$$\begin{cases} (s_1, s_2) \in r \\ s_1, e_1 \Rightarrow v_1, s'_1 \end{cases}$$

then there exists $r' \rhd r$ and v_2, s'_2 such that

$$\begin{cases} s_2, e_2 \Rightarrow v_2, s'_2 \\ (s'_1, s'_2) \in r' \end{cases}$$

and $v_1 \leq_{r'} v_2 : ty$.

This uses the notion of extension of state-relations:

$\boxed{r' \rhd r}$ holds iff $r' = r \otimes r''$ for some r''—see Definition 5.1.

The extensionality properties of $\leq_r$ on canonical forms

- For $ty \in \{\texttt{bool}, \texttt{int}, \texttt{unit}\}$, $v_1 \leq_r v_2 : ty$ iff $v_1 = v_2$.
- $v_1 \leq_r v_2 : \texttt{int ref}$ iff $!v_1 \leq_r \, !v_2 : \texttt{int}$ and for all $\texttt{n} \in \mathbb{Z}$, $(v_1 := \texttt{n}) \leq_r (v_2 := \texttt{n}) : \texttt{unit}$.
- $v_1 \leq_r v_2 : ty_1 * ty_2$ iff $\texttt{fst}\, v_1 \leq_r \texttt{fst}\, v_2 : ty_1$ and $\texttt{snd}\, v_1 \leq_r \texttt{snd}\, v_2 : ty_2$.
- $v_1 \leq_r v_2 : ty_1 \texttt{->} ty_2$ iff for all $r' \rhd r$ and all v_1', v_2'

$$v_1' \leq_{r'} v_2' : ty_1 \;\supset\; v_1\, v_1' \leq_{r'} v_2\, v_2' : ty_2$$

The last property is characteristic of (Kripke) logical relations (Plotkin 1973; O'Hearn and Riecke 1995).

The relationship between $\leq_r$ and contextual equivalence

For all types ty, finite sets w of locations, and programs $e_1, e_2 \in \mathbf{Prog}_{ty}(w)$

$$e_1 \leq_{\mathrm{ctx}} e_2 : ty \quad \text{iff} \quad e_1 \leq_{id_w} e_2 : ty$$

where $id_w \in \mathbf{Rel}(w, w)$ is the identity state-relation for w:

$$id_w \triangleq \{ (s, s) \mid s \in \mathbf{St}(w) \}.$$

Hence e_1 and e_2 are contextually equivalent iff both $e_1 \leq_{id_w} e_2 : ty$ and $e_2 \leq_{id_w} e_1 : ty$.

Outline of the proof of $p =_{\mathrm{ctx}} m : \mathtt{int\ \text{->}\ int}$ (cf. Slide 2)

$$\emptyset, p \Rightarrow (\mathtt{fun(}x\ \mathtt{:\ int)\ \text{->}}\ \ell_1\ \mathtt{:=\ !}\ell_1\ \mathtt{+}\ x\ \mathtt{;\ !}\ell_1), \{\ell_1 \mapsto \mathtt{0}\}$$

$$\emptyset, m \Rightarrow (\mathtt{fun(}y\ \mathtt{:\ int)\ \text{->}}\ \ell_2\ \mathtt{:=\ !}\ell_2\ \mathtt{-}\ x\ \mathtt{;\ 0\ -\ !}\ell_2), \{\ell_2 \mapsto \mathtt{0}\}$$

Define

$$\boxed{r \triangleq \{\ (s_1, s_2) \mid s_1(\ell_1) = -s_2(\ell_2)\ \}} \in \mathbf{Rel}(\{\ell_1\}, \{\ell_2\}).$$

Then $r \rhd id_\emptyset$, $(\{\ell_1 \mapsto \mathtt{0}\}, \{\ell_2 \mapsto \mathtt{0}\}) \in r$, and from Slide 20

$$(\mathtt{fun(}x\ \mathtt{:\ int)\ \text{->}}\ \ell_1\ \mathtt{:=\ !}\ell_1\ \mathtt{+}\ x\ \mathtt{;\ !}\ell_1)\ \leq_r$$
$$(\mathtt{fun(}y\ \mathtt{:\ int)\ \text{->}}\ \ell_2\ \mathtt{:=\ !}\ell_2\ \mathtt{-}\ x\ \mathtt{;\ 0\ -\ !}\ell_2) : \mathtt{int\ \text{->}\ int.}$$

So by Slide 19, $p \leq_{id_\emptyset} m : \mathtt{int\ \text{->}\ int}$.

Hence by Slide 21, $p \leq_{\mathrm{ctx}} m : \mathtt{int\ \text{->}\ int}$.

Similarly $m \leq_{\mathrm{ctx}} p : \mathtt{int\ \text{->}\ int}$.

An unwinding theorem

Given $f : ty_1 \texttt{->} ty_2, x : ty_1 \vdash e_2 : ty_2$,
for each $0 \leq n \leq \omega$ define $f_n \in \mathbf{Prog}_{ty_1 \texttt{->} ty_2}$ by:

$$\begin{cases} f_0 & \triangleq \texttt{fun}\ f\ \texttt{=}\ \texttt{(}x\ \texttt{:}\ ty_1\texttt{)}\ \texttt{->}\ f\ x \\ f_{n+1} & \triangleq \texttt{fun(}x\ \texttt{:}\ ty_1\texttt{)}\ \texttt{->}\ e_2[f_n/f] \\ f_\omega & \triangleq \texttt{fun}\ f\ \texttt{=}\ \texttt{(}x\ \texttt{:}\ ty_1\texttt{)}\ \texttt{->}\ e_2. \end{cases}$$

Then for all $f : ty_1 \texttt{->} ty_2 \vdash e : ty$ and all states s

$$s, e[f_\omega/f] \Downarrow \quad \text{iff} \quad \exists n \geq 0 \,.\, s, e[f_n/f] \Downarrow.$$

Definition of the logical simulation relation

$$
\begin{aligned}
e_1 \leq_r e_2 : ty \;\triangleq\; & \\
& \forall r' \rhd r, (s'_1, s'_2) \in r', (\mathcal{F}s_1, \mathcal{F}s_2) \in \mathbf{Stack}_{ty}(r'). \\
& \qquad \langle s'_1 \,,\, \mathcal{F}s_1 \,,\, e_1 \rangle\searrow \;\supset\; \langle s'_2 \,,\, \mathcal{F}s_2 \,,\, e_2 \rangle\searrow
\end{aligned}
$$

where

$$
\begin{aligned}
(\mathcal{F}s_1, \mathcal{F}s_2) \in \mathbf{Stack}_{ty}(r') \;\triangleq\; & \\
& \forall r'' \rhd r', (s''_1, s''_2) \in r'', (v_1, v_2) \in \mathbf{Val}_{ty}(r''). \\
& \qquad \langle s''_1 \,,\, \mathcal{F}s_1 \,,\, v_1 \rangle\searrow \;\supset\; \langle s''_2 \,,\, \mathcal{F}s_2 \,,\, v_2 \rangle\searrow
\end{aligned}
$$

and where $\mathbf{Val}_{ty}(r'')$ is defined in terms of $- \leq_{r''} - : ty$ by induction on the structure of ty using the extensionality properties on Slide 20.

Some things we do not know how to do yet

Can the method of proving contextual equivalences outlined here be extended to larger fragments of ML with:

- structures and signatures (abstract data types)
- functions with local references to values of arbitrary types (and ditto for exception packets)
- recursively defined, mutable data structures
- objects and classes à la Objective Caml?

The simulation property of the logical relation (Slide 19) is only a sufficient, but not a necessary condition for $e_1 \leq_{\mathbf{ctx}} e_2 : ty$ to hold. Are there other forms of logical relation, useful for proving contextual equivalences?