CS 476 – Programming Language Design

William Mansky
From Typed Lambda Calculus to OCaml

• User-friendly syntax
• Basic types, tuples, records
• Inductive datatypes and pattern-matching
➢ Local declarations
• References
• Type inference
• Generics/polymorphism
OCaml: Local Declarations

• OCaml has two kinds of local declarations:

let x = 3;;
let y = x + 1;;

let p = next_ref s in (update r x p, update s p (Obj o));;
OCaml: Let-In

\[ L ::= \ldots | \text{let } <\text{ident}> = L \text{ in } L \]

\[
\begin{align*}
\Gamma \vdash l_1 : \tau_1 & \quad \Gamma[x \mapsto \tau_1] \vdash l_2 : \tau \\
\hline
\Gamma \vdash \text{let } x = l_1 \text{ in } l_2 : \tau
\end{align*}
\]

\[
\begin{align*}
l' & \to l_1' \\
\hline
\text{let } x = l_1 \text{ in } l_2 & \to \text{let } x = l_1' \text{ in } l_2 & \text{let } x = v \text{ in } l_2 & \to [x \mapsto v]l_2
\end{align*}
\]
OCaml: Local Declarations

let x = 3;;
let y = x + 1;;
=>
let x = 3 in let y = x + 1;;
→
let y = 3 + 1;;
OCaml: Local Declarations

let x = 3;;
let y = x + 1;;
→
let y = x + 1;; (* {x = 3} *)
OCaml: Local Declarations

\[ L ::= \ldots \mid \text{let } \text{<ident> } = L \text{ in } L \]

\[ TD ::= \ldots \]

\[ T ::= \ldots \]

\[ C ::= \text{let } \text{<ident> } = L \mid C;\ C \mid \text{skip} \]

\[ P ::= TD;; \ldots TD;; C;; \]
Local Declarations: Types

\[
\Gamma \vdash l : \tau \\
\Gamma \vdash \text{let } x = l : \Gamma[x \mapsto \tau]
\]

\[
\begin{align*}
\Gamma \vdash c_1 : \Gamma_1 & \quad \Gamma_1 \vdash c_2 : \Gamma_2 \\
\Gamma \vdash c_1 ; ; c_2 : \Gamma_2 & \quad \Gamma \vdash \text{skip} : \Gamma
\end{align*}
\]
Local Declarations: Semantics

• As before, our configurations are now pairs of expression/command and environment

\[
\frac{\rho(x) = v}{(x, \rho) \rightarrow (v, \rho)} \quad \frac{(l, \rho) \rightarrow (l', \rho)}{(\text{let } x = l, \rho) \rightarrow (\text{let } x = l', \rho)} \quad \frac{(\text{let } x = v, \rho)}{(\text{let } x = v, \rho) \rightarrow (\text{skip}, \rho[x \mapsto v])}
\]
Local Declarations: Semantics

As before, our configurations are now pairs of expression/command and environment

\[
\rho(x) = v \quad \frac{}{\langle x, \rho \rangle \Downarrow v}
\]

\[
(l, \rho) \Downarrow v 
\quad \frac{(\text{let } x = l, \rho) \rightarrow (\text{skip}, \rho[x \mapsto v])}{(let x = l, \rho) \rightarrow (\text{skip}, \rho[x \mapsto v])}
\]

\[
(c_1, \rho) \rightarrow (c_1', \rho') 
\quad \frac{(c_1, \rho) \rightarrow (c_1', \rho')}{(c_1; ; c_2, \rho) \rightarrow (c_1'; ; c_2, \rho')}
\]

\[
\text{(skip;} ; c_2, \rho) \rightarrow (c_2, \rho)
\]
OCaml: Declarations and Scope

• At the top level, this gives us mutable variables!

```
let x = 3;;
let x = 4;;
→
let x = 4;; (* {x = 3} *)
→
(* skip *) (* {x = 4} *)
```
• At the top level, this gives us mutable variables!

let x = 3;;
let y = x + 2;;
let x = 4;;
→
let y = x + 2;; (* {x = 3} *)
let x = 4;;
OCaml: Declarations and Scope

• At the top level, this gives us mutable variables!

```ocaml
let y = x + 2;; (* {x = 3} *)
let x = 4;;
→
let x = 4;; (* {x = 3, y = 5} *)
```
OCaml: Declarations and Scope

• At the top level, this gives us mutable variables!

let x = 3;;
let f = fun y -> x + 2;;
(* {x = 3; f = ?} *)
OCaml: Declarations and Scope

• At the top level, this gives us mutable variables!

```ocaml
let x = 3;;
let f = fun y -> x + y;;
(* {x = 3; f = fun y -> x + y} *)
```
OCaml: Declarations and Scope

• At the top level, this gives us mutable variables!

let x = 3;;  x is a free variable in f
let f = fun y -> x + y;;  the function value doesn’t include the value of x
(* {x = 3; f = fun y -> x + y} *)
let x = 4;;
let z = f 2;; (z = x + 2 = 6) (* !! *)
OCaml: Declarations and Scope

• We need function values to remember the values of variables!

```ocaml
let x = 3;;
let f = fun y -> x + y;;  (* {x = 3; f = fun y -> x + y} *)
let x = 4;;
let z = f 2;;  (* z = x + 2 = 6 *)  (* !! *)
```

- `x` is a free variable in `f`
- The function value doesn’t include the value of `x`
OCaml: Declarations and Scope

• We need function values to remember the values of variables!

```
let x = 3;;
let f = fun y -> x + y;;
(* {x = 3; f = <fun y -> x + y; {x = 3}>} *)
let x = 4;;
let z = f 2;;
```

“close” the function’s free variables

“closure”
OCaml: Declarations and Scope

• We need function values to remember the values of variables!

let x = 3;;
let f = fun y -> x + y;;
(* {x = 3; f = <fun y -> x + y; {x = 3}>} *)
let x = 4;;
(let x = 4;; *)
let z = f 2;;  (* z = (x + 2 with {x = 3}) = 5 *)
Local Declarations: Semantics

\[
(l, \rho) \downarrow v \\
(let \ x = l, \rho) \to (skip, \rho[x \mapsto v]) \\
(fun \ x \to l, \rho) \downarrow fun \ x \to l
\]
Local Declarations: Semantics

• Functions are no longer values, closures are

\[
\begin{align*}
(l, \rho) & \Downarrow v \\
(\text{let } x = l, \rho) & \rightarrow (\text{skip}, \rho[x \mapsto v]) \\
\text{(fun } x \rightarrow l, \rho) & \Downarrow (\text{fun } x \rightarrow l, \rho) \\
(l_1, \rho) & \Downarrow (\text{fun } x \rightarrow l, \rho_1) \quad (l_2, \rho) \Downarrow v_2 \quad ([x \mapsto v_2]l, \rho_1) \Downarrow v \\
(l_1 \; l_2, \rho) & \Downarrow v
\end{align*}
\]
Local Declarations: Semantics

• Functions are no longer values, *closures* are

\[
(l, \rho) \downarrow v \\
\text{(let } x = l, \rho \text{)} \rightarrow (\text{skip, } \rho[x \mapsto v]) \\
\]

\[
\text{(fun } x \rightarrow l, \rho \text{)} \downarrow \langle \text{fun } x \rightarrow l, \rho \rangle \\
\]

\[
(l_1, \rho) \downarrow \langle \text{fun } x \rightarrow l, \rho_1 \rangle \quad (l_2, \rho) \downarrow v_2 \quad (l, \rho_1[x \mapsto v_2]) \downarrow v \\
\]

\[
(l_1 l_2, \rho) \downarrow v
\]
Local Declarations: Semantics

• Functions are no longer values, *closures* are
• Substitution semantics:

\[(\text{fun } x \to \text{fun } y \to x + y) \; 2 \Downarrow \; \text{fun } y \to 2 + y\]

• Closure semantics:

\[
\begin{align*}
(l_1, \rho) \Downarrow (\text{fun } x \to l, \rho_1) & \quad (l_2, \rho) \Downarrow v_2 & \quad (l, \rho_1[x \mapsto v_2]) \Downarrow v \\
(l_1, l_2, \rho) \Downarrow v
\end{align*}
\]
Local Declarations: Semantics

• Functions are no longer values, *closures* are

• Substitution semantics:

  \[
  (\text{fun } x \to \text{fun } y \to x + y) \ 2 \Downarrow \text{fun } y \to 2 + y
  \]

• Closure semantics:

  \[
  (\text{fun } x \to \text{fun } y \to x + y) \ 2 \Downarrow \langle \text{fun } y \to x + y, \{x = 2}\rangle
  \]