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: References

• Variables in OCaml are immutable (though hideable)
• What if we want mutable variables?
  1. Don’t! Just make a new thing instead of changing the old one.

  let h = empty_table;;
  let item = Add (Int 1, Int 3);;
  let h1 = table_add h item;;

• But what if we want to change item later, and have it change in the table?
OCaml: References

• Variables in OCaml are immutable (though hideable)
• What if we want mutable variables?
  2. Use references

```
let h = empty_table;;
let item = ref (Add (Int 1, Int 3));; (* item : exp ref *)
let h1 = table_add h item;;
item := Add (Int 2, Int 3);; (* item now points to a new value *)
```

• Just like references in our OO language, or pointers in C
OCaml: References

$L ::= \ldots \mid \text{ref } L \mid L ::= L \mid !L$

$T ::= \ldots \mid T \text{ ref}$

- \text{ref } l \text{ makes a new reference whose initial value is } l$
- $x ::= l$ \text{ changes the value at } x$
- $!l$ \text{ looks up the value at } l
OCaml: References

\[ L ::= \ldots \mid \text{ref } L \mid L := L \mid !L \]

\[ T ::= \ldots \mid T \text{ ref} \]

• \( x := l \) is an expression! What does it return?
  — A special “nothing” value: () of type \text{unit}

• OCaml expressions now have \textit{side effects}
OCaml: References

\[ L ::= \ldots \mid \text{ref } L \mid L := L \mid !L \mid () \mid L; L \]

\[ T ::= \ldots \mid T \text{ ref} \mid \text{unit} \]

• \( x := l \) is an expression, returning () of type unit
• OCaml expressions now have side effects
• \( l_1; l_2 \) throws away the result of \( l_1 \)
\[
\begin{align*}
\Gamma \vdash l : \tau & \quad \frac{\Gamma \vdash l : \tau \text{ ref}}{\Gamma \vdash \text{ref}l : \tau \text{ ref}} & \quad \frac{\Gamma \vdash l : \tau \text{ ref}}{\Gamma \vdash !l : \tau} & \quad \frac{\Gamma \vdash l_1 : \tau \text{ ref} \quad \Gamma \vdash l_2 : \tau}{\Gamma \vdash l_1 := l_2 : \text{unit}} \\
\Gamma \vdash () : \text{unit} & \quad \frac{\Gamma \vdash l_1 : \text{unit} \quad \Gamma \vdash l_2 : \tau}{\Gamma \vdash l_1; l_2 : \tau}
\end{align*}
\]
References: Types

\[ \Gamma \vdash l : \tau \quad \Gamma \vdash l : \tau \text{ref} \quad \Gamma \vdash l_1 : \tau \text{ref} \quad \Gamma \vdash l_2 : \tau \]
\[ \Gamma \vdash \text{ref} \ l : \tau \text{ref} \quad \Gamma \vdash !l : \tau \quad \Gamma \vdash l_1 := l_2 : \text{unit} \]
\[ \Gamma \vdash () : \text{unit} \quad \Gamma \vdash l_1 : \tau_0 \quad \Gamma \vdash l_2 : \tau \quad \Gamma \vdash l_1; l_2 : \tau \]
References: Semantics

• References have aliasing:

let x = ref 1;;  (* {x = r}, {r ↦ 1} *)
let y = x;;      (* {x = r, y = r}, {r ↦ 1} *)
y := 2; !x;;     (* {x = r, y = r}, {r ↦ 2} *)

(* result: 2 *)

• So we need to use a two-level state again
References and $\emptyset$ are values

\[
(l, \rho, \sigma) \downarrow (v, \sigma') \quad r \notin \text{dom}(\sigma) \\
\hline
\text{(ref } l, \rho, \sigma) \downarrow (r, \rho, \sigma'[r \mapsto v])
\]

\[
(l_1, \rho, \sigma) \downarrow (r, \sigma) \quad (l_2, \rho, \sigma) \downarrow (v, \sigma) \\
\hline
(l_1 := l_2, \rho, \sigma) \downarrow (\emptyset, \rho, \sigma'[r \mapsto v])
\]

\[
(l, \rho, \sigma) \downarrow (r, \sigma') \quad \sigma'(r) = v \\
\hline
(! l, \rho, \sigma) \downarrow (v, \rho, \sigma')
\]
References: Evaluation Order

• If any expression can change the state, the order of evaluation matters!

• As designers, how should we decide on the order?

\[
(l_1, \rho, \sigma?) \downarrow (r, \sigma?) \quad (l_2, \rho, \sigma?) \downarrow (v, \sigma?) \\
(l_1 := l_2, \rho, \sigma) \downarrow ((), \rho, \sigma?)
\]
References: Evaluation Order

• If any expression can change the state, the order of evaluation matters!

• Approach 1: do something logical

\[
(l_1, \rho, \sigma) \downarrow (r, \sigma_1) \quad (l_2, \rho, \sigma_1) \downarrow (v, \sigma_2)
\]

\[
(l_1 := l_2, \rho, \sigma) \downarrow ((), \rho, \sigma_2)
\]
References: Evaluation Order

• If any expression can change the state, the order of evaluation matters!

• Approach 2: do what the implementation does

\[
\begin{align*}
(l_1, \rho, \sigma_1) \downarrow (r, \sigma_2) & \quad (l_2, \rho, \sigma) \downarrow (v, \sigma_1) \\
\hline
(l_1 := l_2, \rho, \sigma) \downarrow ((\), \rho, \sigma_2)
\end{align*}
\]
References: Evaluation Order

• If any expression can change the state, the order of evaluation matters!

• Approach 3: leave it unspecified

\[
\begin{align*}
(l_1, \rho, \sigma) & \downarrow (r, \sigma_1) & (l_2, \rho, \sigma_1) & \downarrow (v, \sigma_2) \\
(l_1 := l_2, \rho, \sigma) & \downarrow ((\), \rho, \sigma_2)
\end{align*}
\]

\[
\begin{align*}
(l_1, \rho, \sigma_1) & \downarrow (r, \sigma_2) & (l_2, \rho, \sigma) & \downarrow (v, \sigma_1) \\
(l_1 := l_2, \rho, \sigma) & \downarrow ((\), \rho, \sigma_2)
\end{align*}
\]
References: Evaluation Order

• If any expression can change the state, the order of evaluation matters!
• Approach 4: “undefined behavior”

\[
\begin{align*}
(l_1, \rho, \sigma) \downarrow (r, \sigma_1) & \quad (l_2, \rho, \sigma_1) \downarrow (v, \sigma_1) \\
(l_1 := l_2, \rho, \sigma) \downarrow ((), \rho, \sigma_1)
\end{align*}
\]

\[
\begin{align*}
(l_1, \rho, \sigma_1) \downarrow (r, \sigma_1) & \quad (l_2, \rho, \sigma) \downarrow (v, \sigma_1) \\
(l_1 := l_2, \rho, \sigma) \downarrow ((), \rho, \sigma_1)
\end{align*}
\]
References: Evaluation Order

• If any expression can change the state, the order of evaluation matters!

• Whichever we choose, we have to apply it to every operator with more than one argument:

\[
(l_1, \rho, \sigma) \downarrow (v_1, \sigma_1) \quad (l_2, \rho, \sigma_1) \downarrow (v_2, \sigma_2) \quad (v_1 + v_2 = v)
\]

since \((x := 1; 3) + (x := 2; 4)\) is now a valid expression
OCaml: Evaluation Order

• Evaluation order matters whenever expressions have side effects
  — References/mutable variables
  — I/O (console, file, network, graphics, etc.)

• Side effects raise important questions:
  — Which subexpression gets evaluated first?
  — When does an expression get evaluated?
  — How many times is it evaluated?

• Avoiding them makes code simpler, but they’re hard to avoid!