CS 476 – Programming Language Design

William Mansky
Questions?
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_map;;
  let item = Add (Int 1, Int 3);;
  let h1 = map_insert h item;;
OCaml: References

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

  let h = ref empty_map;; (* h : map ref *)
  let item = Add (Int 1, Int 3);;
  h := map_insert h “e” item;; (* item has now been added to h *)

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

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

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

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

• We can have a ref type for any type (like int*, etc. in C)
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 \text{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 \( l_1 \)
new_id : unit -> int  (* want to return a new int every time *)

let next_id = ref 0 (* next_id : int ref *)

let new_id () : int = let v = !next_id in next_id := !next_id + 1; v
Questions?

Top
OCaml: References

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

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

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

- We can have a ref type for any type (like \text{int*}, etc. in C)
References: Types

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

• References have aliasing:

```ml
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: Semantics

References and \( () \) are values

\[
\begin{align*}
(l, \rho, \sigma) \downarrow (v, \sigma') & \quad r \not\in \text{dom}(\sigma) \\
\text{(ref l, } \rho, \sigma) \downarrow (r, \rho, \sigma'[r \mapsto v])
\end{align*}
\]

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

\[
\begin{align*}
(l, \rho, \sigma) \downarrow (r, \sigma') & \quad \sigma'(r) = v \\
\text{(! l, } \rho, \sigma) \downarrow (v, \rho, \sigma')
\end{align*}
\]
Questions?
If any expression can change the state, the order of evaluation matters!

As designers, how should we decide on the order?

\[
(l_1, \rho, \sigma?) \Downarrow (r, \sigma?) \quad (l_2, \rho, \sigma?) \Downarrow (v, \sigma?)
\]

\[
(l_1 := l_2, \rho, \sigma) \Downarrow ((), \rho, \sigma?)
\]

Exercise: What should our language design say about the order in which \(l_1\) and \(l_2\) evaluate?
References: Evaluation Order

• If any expression can change the state, the order of evaluation matters!
• Approach 1: do something logical, like left-to-right

\[
(l_1, \rho, \sigma) \downarrow (r, \sigma_1) \\
(l_2, \rho, \sigma_1) \downarrow (v, \sigma_2) \\
(l_1 := l_2, \rho, \sigma) \downarrow ((), \rho, \sigma_2)
\]
References: Evaluation Order

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

• Approach 2: do what the implementation does, right-to-left

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

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

• Approach 3: either way is allowed

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

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

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

• Approach 4: “undefined behavior”: if both change the state, the program is wrong

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

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

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

\[
(l_1 + l_2, \rho, \sigma) \downarrow (v, \rho, \sigma_2)
\]

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!