CS 476 – Programming Language Design

William Mansky
Adding Variables

• Next target language: expressions + variables
Adding Variables: Syntax

\[ E ::= \langle\#\rangle \]

\[ | E + E | E - E | E \times E \]

\[ | \langle\text{bool}\rangle \]

\[ | E \text{ and } E | E \text{ or } E \]

\[ | \text{not } E \]

\[ | E = E \]

\[ | \text{if } E \text{ then } E \text{ else } E \]
Adding Variables: Syntax

\[ E ::= <\#> \mid <\text{ident}> \]
\[ \mid E + E \mid E - E \mid E * E \]
\[ \mid \text{bool} \]
\[ \mid E \text{ and } E \mid E \text{ or } E \]
\[ \mid \text{not } E \]
\[ \mid E = E \]
\[ \mid \text{if } E \text{ then } E \text{ else } E \]

Example terms:
x + 5
if y then 3 else z
Adding Variables

• How do variables get their values?

• Option 1: local binding
  — fun x -> x + 5
  — let x = 2 + 3 in if x = 5 then true else false

• Option 2: assignment
  — x = 3; y = x + 5;
  — x = 4; z = x + 5;       // y != z
• Option 2: assignment
  —  \( x = 3; y = x + 5; \)
  —  \( x = 4; z = x + 5; \quad // \ y \neq z \)

• Some terms now don’t just compute values: they have *side effects* that change the meaning of later terms

• We call terms that compute values *expressions*, and terms with side effects *commands*
Adding Variables: Syntax

\[ E ::= \texttt{\#} | \texttt{ident} | E + E | E - E | E \ast E | <\text{bool}> | E \text{ and } E | E \text{ or } E | \text{not } E | E = E | \text{if } E \text{ then } E \text{ else } E \]

\[ C ::= \texttt{ident} ::= E | C; C | \text{skip} \]

Example terms:
- \( x := 3; y := x = 2; z := \text{true} \&\& y \)
- \( x := 0; x := x + 1; y := x; x := x + 1 \)
Adding Variables: Types

• How do we type variables?

\[
\begin{align*}
(n \text{ is a number}) & \quad \frac{e_1 : \text{int}}{n : \text{int}} \\
(x \text{ is an identifier}) & \quad \frac{x : ?}{e_1 + e_2 : \text{int}}
\end{align*}
\]
Typing Variables, Approach #1

• How do we type variables?

\[
\begin{align*}
\text{(n is a number)} & \quad n : \text{int} \\
\text{(x is an identifier)} & \quad x : \text{int} \\
\text{e}_1 : \text{int} & \quad \text{e}_2 : \text{int} \\
\text{(x is an identifier)} & \quad x : \text{bool}
\end{align*}
\]
Typing Variables, Approach #1

• How do we type variables?

\[
\frac{(n \text{ is a number})}{n : \text{int}} \quad \frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}
\]

\[
\frac{(x \text{ is an identifier})}{x : \tau}
\]
Typing Variables, Approach #2

• How do we type variables?

\[ E ::= \langle\#\rangle \mid (\text{ident} : T) \]
\[ \mid E + E \mid E - E \mid E \times E \]
\[ \mid \langle\text{bool}\rangle \]
\[ \mid E \text{ and } E \mid E \text{ or } E \]
\[ \mid \text{not } E \]
\[ \mid E = E \]
\[ \mid \text{if } E \text{ then } E \text{ else } E \]

\[ T ::= \text{INT} \mid \text{BOOL} \]
\[ \]
\[ (x : \text{INT}) : \text{int} \]
\[ (x : \text{BOOL}) : \text{bool} \]

• Each tag has its own namespace
• add INT/BOOL tags in preprocessing
HW2 Questions
Adding Variables: Syntax

\[ E ::= \langle\#\rangle \mid \text{ident} \mid E + E \mid E - E \mid E \times E \mid \text{bool} \mid E \text{ and } E \mid E \text{ or } E \mid \text{not } E \mid E = E \mid \text{if } E \text{ then } E \text{ else } E \]

\[ C ::= \text{ident} := E \mid C; C \mid \text{skip} \]

Example terms:
\[
x := 3; \ y := x = 2; \ z := \text{true} \&\& y \\
x := 0; \ x := x + 1; \ y := x; \ x := x + 1
\]
Typing Variables, Approach #3

• How do we type variables?
• Instead of putting tags in the syntax, we could store them separately, in a type context
• New typing judgment: $\Gamma \vdash e : \tau$, “$e$ has type $\tau$ in context $\Gamma$”
• $\Gamma$ is a partial function from identifiers to types

$$
\Gamma(x) = \tau \quad \text{(n is a number)} \quad \frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}
$$
Adding Variables: Types for Commands

• What types can a command have?
• Only one: “correct”, or “ok”

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad \Gamma(x) = \tau \\
\therefore & \quad \Gamma \vdash x := e : \text{ok} \\
\Gamma \vdash c_1 : \text{ok} & \quad \Gamma \vdash c_2 : \text{ok} \\
\therefore & \quad \Gamma \vdash c_1; c_2 : \text{ok} \\
\Gamma \vdash \text{skip} : \text{ok}
\end{align*}
\]
Adding Variables: Semantics

• Our programs now have *state*!

```
x := 3 + 4;
y := x + 5;
b := if y = 12 then true else false;
x := 2;
z := x + 5;  // What are the values of the variables at this point?
{b = true, x = 2, y = 12}
c := b && true;
```
Adding Variables: Semantics

• Our programs now have state!

• Small-step relation is \((t, \sigma) \rightarrow (t', \sigma')\), where state \(\sigma\) is a map from variables to values

\[
\frac{\sigma(x) = v}{(x, \sigma) \rightarrow (v, \sigma)} \quad \frac{(e_1, \sigma) \rightarrow (e_1', \sigma')}{(e_1 + e_2, \sigma) \rightarrow (e_1' + e_2, \sigma')} \quad \frac{(v_1 + v_2 = v)}{(v_1 + v_2, \sigma) \rightarrow (v, \sigma)}
\]
Textbook Notation Warning

- *The Formal Semantics of Programming Languages*, Winskel

- Where we write $e \Downarrow v$, Winskel writes $e \rightarrow v$

- Where we write $e \rightarrow e'$, Winskel writes $e \rightarrow_1 e'$
Adding Variables: Semantics

• Our programs now have *state*!

• Small-step relation is \((t, \sigma) \rightarrow (t', \sigma')\), where state \(\sigma\) is a map from variables to values

\[
\sigma(x) = v \\
(x, \sigma) \rightarrow (v, \sigma)
\]

\[
(e_1, \sigma) \rightarrow (e'_1, \sigma) \\
(e_1 + e_2, \sigma) \rightarrow (e'_1 + e_2, \sigma)
\]

\[
(v_1 + v_2 = v) \\
(v_1 + v_2, \sigma) \rightarrow (v, \sigma)
\]
Adding Variables: Semantics

• Our programs now have state!

• Small-step relation is \((t, \sigma) \rightarrow (t', \sigma')\), where state \(\sigma\) is a map from variables to values

\[
\begin{align*}
(e, \sigma) &\rightarrow (e', \sigma) \\
(x := e, \sigma) &\rightarrow (x := e', \sigma) \\
(x := v, \sigma) &\rightarrow (\text{skip}, \sigma[x \mapsto v]) \\
(c_1, \sigma) &\rightarrow (c_1', \sigma') \\
(c_1; c_2, \sigma) &\rightarrow (c_1'; c_2, \sigma') \\
(\text{skip}; c_2, \sigma) &\rightarrow (c_2, \sigma)
\end{align*}
\]
Structural Rule for Sequencing

• What if we had

\[
(x := 3; x := x + 1; x := 4) 
\]

$$\rightarrow x := x + 1; x := 4, \{x = 3\} \quad \text{(by rule 1)}$$

$$\rightarrow x := x + 1, \{x = 4\} \quad \text{(by rule 2)}$$

$$\rightarrow \text{skip, } \{x = 5\}$$
Adding Variables: Big-Step Semantics

• What does an expression evaluate to? \((e, \sigma) \downarrow v\)
• What does a command evaluate to? \((c, \sigma) \downarrow \sigma'\)

\[
\frac{\sigma(x) = v}{(x, \sigma) \downarrow v}
\]

\[
\frac{(e_1, \sigma) \downarrow v_1}{(e_1 + e_2, \sigma) \downarrow v}
\]

\[
\frac{(e_2, \sigma) \downarrow v_2}{v_1 + v_2 = v}
\]

\[
\frac{(x := e, \sigma) \downarrow \sigma[x \mapsto v]}{(c_1, \sigma) \downarrow \sigma' (c_2, \sigma') \downarrow \sigma''}
\]

\[
\frac{(c_1; c_2, \sigma) \downarrow \sigma''}{(skip, \sigma) \downarrow \sigma}
\]
Adding Variables: Hybrid Semantics

• Expressions don’t change the state, commands do

• So we might only care about intermediate steps for commands

\[
\begin{align*}
\sigma(x) &= v \\
(x, \sigma) &\Downarrow v \\
\end{align*}
\]

\[
\begin{align*}
(e_1, \sigma) &\Downarrow v_1 \\
(e_2, \sigma) &\Downarrow v_2 \\
(v_1 + v_2 &= v) \\
(e_1 + e_2, \sigma) &\Downarrow v \\
\end{align*}
\]

\[
\begin{align*}
\sigma(x) &= v \\
(x := e, \sigma) &\rightarrow (\text{skip}, \sigma[x \mapsto v]) \\
\end{align*}
\]

\[
\begin{align*}
(c_1, \sigma) &\rightarrow (c_1', \sigma') \\
(c_1; c_2, \sigma) &\rightarrow (c_1'; c_2, \sigma') \\
\text{(skip; } c_2, \sigma) &\rightarrow (c_2, \sigma) \\
\end{align*}
\]
Adding Variables: Interpreter

• Follow the big-step semantics

```ocaml
let rec eval_exp (e : exp) : value option =
  match e with
  | Id x ->
  | Add (e1, e2) ->
  | ...
Adding Variables: Interpreter

- Follow the big-step semantics

```ocaml
type state = string -> value option

let rec eval_exp (e : exp) (s : state) : value option =
  match e with
  | Id x -> \[\sigma(x) = v\]
  | Add (e1, e2) -> \[\sigma(x) = v\]
  | ... 
```

\[\sigma(x) = v\]
Adding Variables: Interpreter

• Follow the big-step semantics

```
type state = string -> value option

let rec eval_exp (e : exp) (s : state) : value option =
  match e with
  | Id x -> s x
  | Add (e1, e2) ->
  | ...
```

\[
\sigma(x) = v \\
\frac{(x, \sigma) \downarrow v}
\]
Adding Variables: Interpreter

• Follow the big-step semantics

type state = string -> value option

let rec eval_exp (e : exp) (s : state) : value option =

match e with
| Add (e1, e2) ->
  (match eval_exp e1 s, eval_exp e2 s with
   | Some (IntV i1), Some (IntV i2) -> Some (IntV (i1 + i2))
   | __, __ -> None)
Adding Variables: Interpreter

let update (s : state) (x : string) (v : value) : state =
  fun y -> if y = x then Some v else s y

let rec eval_cmd (c : cmd) (s : state) : state option =
  match c with
  | Assign (x, e) -> (e, σ) ↓ v
  | Seq (c1, c2) -> (x := e, σ) ↓ σ[x ↦ v]
  | Skip ->
Adding Variables: Interpreter

let rec eval_cmd (c : cmd) (s : state) : state option =
match c with
| Assign (x, e) ->
  (match eval_exp e s with
  | Some v -> Some (update s x v)
  | None -> None)
| Seq (c1, c2) ->
| Skip ->

\[
\frac{(e, \sigma) \downarrow v}{(x := e, \sigma) \downarrow \sigma[x \mapsto v]}
\]
let rec eval_cmd (c : cmd) (s : state) : state option =
  match c with
  | ... |
  | Seq (c1, c2) -> (c1, σ) ↓ σ’  (c2, σ’) ↓ σ’’  (c1; c2, σ) ↓ σ’’
  | Skip ->
let rec eval_cmd (c : cmd) (s : state) : state option =
  match c with
  | ...                      (\(c_1, \sigma \mapsto \sigma'\) \(c_2, \sigma' \mapsto \sigma''\) \(c_1; c_2, \sigma \mapsto \sigma''\))
  | Seq \((c_1, c_2)\) \rightarrow
    (match eval_cmd c1 s with
      | Some \(s'\) \rightarrow eval_cmd c2 s'
      | None \rightarrow None)
  | Skip \rightarrow
Adding Variables: Interpreter

let rec eval_cmd (c : cmd) (s : state) : state option =
  match c with
  | ... | ...
  | Seq (c1, c2) ->
    (match eval_cmd c1 s with
     | Some s' -> eval_cmd c2 s'
     | None -> None)
  | Skip -> Some s

  (skip, σ) ↓ σ
let empty_state = fun x -> None;;
eval_cmd (Seq (Assign ("x", Int 2),
    Assign ("y", Add (Ident "x", Int 3))))) empty_state;;
Adding Variables: Interpreter

eval_cmd (Seq (Assign ("x", Int 2),
    Assign ("y", Add (Ident "x", Int 3))))) empty_state;;

let rec eval_cmd (c : cmd) (s : state) : state option =
    match c with
    | Seq (c1, c2) ->
      (match eval_cmd c1 s with
       | Some s’ -> eval_cmd c2 s’
       | None -> None)
Adding Variables: Interpreter

eval_cmd (Seq (Assign ("x", Int 2),
    Assign ("y", Add (Ident "x", Int 3)))) empty_state;;

let rec eval_cmd (c : cmd) (s : state) : state option =
  match c with
  | Seq (c1, c2) ->
    (match eval_cmd (Assign ("x", 2)) empty_state with
     | Some s' -> eval_cmd (Assign ("y", Add (Ident "x", Int 3))) s'
     | None -> None)
Adding Variables: Interpreter

eval_cmd (Seq (Assign ("x", Int 2),
    Assign ("y", Add (Ident “x”, Int 3)))) empty_state;;

match eval_cmd (Assign ("x", 2)) empty_state with
  | Some s’ -> eval_cmd (Assign ("y", Add (Ident “x”, Int 3))) s’

  | Assign (x, e) ->
    (match eval_exp e s with
      | Some v -> Some (update s x v)
Adding Variables: Interpreter

eval_cmd (Seq (Assign (“x”, Int 2),
    Assign (“y”, Add (Ident “x”, Int 3)))) empty_state;;

match eval_cmd (Assign (“x”, 2)) empty_state with
  | Some s’ -> eval_cmd (Assign (“y”, Add (Ident “x”, Int 3))) s’

  | Assign (x, e) ->
    (match eval_exp (Int 2) empty_state with
     | Some v -> Some (update empty_state “x” v))
Adding Variables: Interpreter

eval_cmd (Seq (Assign ("x", Int 2),
              Assign ("y", Add (Ident "x", Int 3)))) empty_state;;

match eval_cmd (Assign ("x", 2)) empty_state with
  | Some s' -> eval_cmd (Assign ("y", Add (Ident "x", Int 3))) s'

  | Assign (x, e) ->
    (match Some (IntV 2) with
     | Some v -> Some (update empty_state "x" v)
eval_cmd (Seq (Assign ("x", Int 2),
    Assign ("y", Add (Ident "x", Int 3)))) empty_state;;

match Some {"x" = IntV 2} with
    | Some s' -> eval_cmd (Assign ("y", Add (Ident "x", Int 3))) s'
    | Assign (x, e) ->
        (match Some (IntV 2) with
            | Some v -> Some (update empty_state "x" v)
Adding Variables: Interpreter

eval_cmd (Seq (Assign ("x", Int 2),
                  Assign ("y", Add (Ident "x", Int 3))))) empty_state;;

eval_cmd (Assign ("y", Add (Ident "x", Int 3))) {"x" = IntV 2}

| Assign (x, e) ->
  (match eval_exp (Add (Ident "x", Int 3)) {"x" = IntV 2} with
   | Some v -> Some (update {"x" = IntV 2} "y" v)
Adding Variables: Interpreter

eval_cmd (Seq (Assign ("x", Int 2),
    Assign ("y", Add (Ident "x", Int 3)))) empty_state;;

eval_cmd (Assign ("y", Add (Ident "x", Int 3))) \{"x" = IntV 2\}

| Assign (x, e) ->
  (match Some (IntV 5) with
    | Some v -> Some (update \{"x" = IntV 2\} "y" v)
Adding Variables: Interpreter

```
val eval_cmd = \n  Seq (Assign ("x", Int 2),
        Assign ("y", Add (Ident "x", Int 3))))

val empty_state = ();;

val res = Some {"x" = IntV 2, "y" = IntV 5}

match eval_cmd with |
  | Some res -> res "y" |
  | None -> None;; 

   : value option = Some (IntV 5)
```
Typing Variables, Approach #3

• How do we type variables?
• Instead of putting tags in the syntax, we could store them separately, in a type context
• New typing judgment: $\Gamma \vdash e : \tau$, “$e$ has type $\tau$ in context $\Gamma$”
• $\Gamma$ is a partial function from identifiers to types

\[
\begin{align*}
\Gamma(x) = \tau & \quad (n \text{ is a number}) \\
\Gamma \vdash x : \tau & \quad \Gamma \vdash n : \text{int} \\
\Gamma \vdash e_1 : \text{int} & \quad \Gamma \vdash e_2 : \text{int} \\
& \quad \Gamma \vdash e_1 + e_2 : \text{int}
\end{align*}
\]

• Where does $\Gamma$ come from?
IMP with Declarations: Syntax

\[
E ::= \ldots \\
C ::= \ldots \\
D ::= T <ident> | D; D \\
T ::= \text{int} | \text{bool} \\
P ::= D; C
\]

Example:

\[
\text{int } x; \\
\text{bool } y; \\
\text{int } z; \\
x ::= 3; \\
y ::= (x = 4); \\
\text{if } y \text{ then } z ::= 2 \text{ else } x ::= 4
\]
IMP with Declarations: Types

• $d : \Gamma$ means “$d$ constructs type context $\Gamma$”

\[
\begin{align*}
\text{int} \ x : \{x : \text{int}\} & \quad \text{bool} \ x : \{x : \text{bool}\} \\
\frac{d_1 : \Gamma_1 \quad d_2 : \Gamma_2}{d_1 ; d_2 : \Gamma_1 \cup \Gamma_2} & \quad \frac{d : \Gamma \quad \Gamma \vdash c : \text{ok}}{d ; c : \text{ok}}
\end{align*}
\]
IMP with Declarations: Semantics

eval_prog p =

\[
\begin{align*}
d; c \rightarrow (c, \emptyset) & \quad \quad \quad \quad \quad \quad \quad (c, \emptyset) \downarrow \sigma \\
\hline
d; c \downarrow \sigma
\end{align*}
\]

match p with Prog (d, c) -> eval_cmd c empty_state
IMP with Declarations: Type Checker

type typ =
type decl =
type prog =
type tycon =
let rec process_decls (d : decl) : tycon =

let typecheck_prog p =
  match p with Prog (d, c) -> typecheck_cmd c (process_decls d)
Questions