CS 476 – Programming Language Design

William Mansky

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Language Design: Outline

- So far we've:
  - started from an informal description of a language
  - turned its syntax into a *grammar* and corresponding OCaml datatype
  - written down *typing rules* and translated them into a typechecker
  - written down *semantic rules* and translated them into an interpreter and debugger
- These will be our main tools for the class!
- For the rest of the class, we'll apply these tools to a range of languages and language features

Adding Variables

- Next target language: expressions + variables

- A variable has a name (usually alphanumeric), and holds a value

- Examples: `x + 5` `if cond then 3 else z`

Adding Variables: Syntax

E ::= <#>

| *E* + *E* | *E* – *E* | *E* * *E*

| <bool>

| *E* and *E* | *E* or *E*

| not *E*

| *E* = *E*

| if *E* then *E* else *E*

Adding Variables: Syntax

E ::= <#> | <ident>

| *E* + *E* | *E* – *E* | *E* * *E*

| <bool>

| *E* and *E* | *E* or *E*

| not *E*

| *E* = *E*

| if *E* then *E* else *E*

Example expressions:

```
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`

Adding Variables

- Option 2: assignment
  - `x = 3; y = x + 5;`
  - `x = 4; z = x + 5;          // y != z`
- Some terms 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 ::= <#> | <ident>
  | E + E | E − E | E * E
  | <bool>
  | E and E | E or E
  | not E
  | E = E
  | if E then E else E

C ::= <ident> := E
  | C; C
  | skip

Example terms:

```
x := 3; y := x = 2; z := a && y
x := 0; x := x + 1; y := x
```

Adding Variables: Types

- How do we typecheck variables?

$$\frac{(n \text{ is a number literal})}{n : \text{int}} \qquad \frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

$$\frac{(x \text{ is an identifier})}{x : ?}$$

- Exercise: How do you figure out the type of a variable without running the program?

Typechecking Variables, Approach #1

- How do we typecheck variables?

$$\frac{(n \text{ is a number literal})}{n : \text{int}} \qquad \frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

$$\frac{(x \text{ is an identifier})}{x : \text{int}} \qquad \frac{(x \text{ is an identifier})}{x : \text{bool}}$$

Typechecking Variables, Approach #1

- How do we typecheck variables?

$$\frac{(n \text{ is a number literal})}{n : \text{int}}$$

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

$$\frac{(x \text{ is an identifier})}{x : \tau}$$

Typechecking Variables, Approach #2

- How do we typecheck variables?

E ::= <#> | (<ident> : T)
  | E + E | E – E | E * E
  | <bool>
  | E and E | E or E
  | not E
  | E = E
  | if E then E else E

T ::= INT | BOOL

$$\frac{}{(x : \texttt{INT}) : \text{int}}$$

$$\frac{}{(x : \texttt{BOOL}) : \text{bool}}$$

- Each tag has its own namespace
- add **INT**/**BOOL** tags in preprocessing

Adding Variables: Syntax

E ::= <#> | <ident>
  | E + E | E – E | E * E
  | <bool>
  | E `and` E | E `or` E
  | `not` E
  | E = E
  | `if` E `then` E `else` E

C ::= <ident> `:=` E
  | C; C
  | `skip`

Example terms:

```
x := 3; y := x = 2; z := a && y
x := 0; x := x + 1; y := x
```

Typechecking Variables, Approach #3

- How do we typecheck 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 τ in context Γ"

  Greek letter "gamma"
- Γ is a map from identifiers to types

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{(n \text{ is a number})}{\Gamma \vdash n : \text{int}} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

Adding Variables: Types for Commands

- What types can a command have?
- Only one: "correct", or "ok"

$$\frac{\Gamma \vdash e : \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x := e : \mathrm{ok}}$$

$$\frac{\Gamma \vdash c_1 : \mathrm{ok} \quad \Gamma \vdash c_2 : \mathrm{ok}}{\Gamma \vdash c_1 \,;\, c_2 : \mathrm{ok}}$$

$$\frac{}{\Gamma \vdash \mathtt{skip} : \mathrm{ok}}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Adding Variables: Syntax

E ::= <#> | <ident>
| *E* + *E* | *E* – *E* | *E* * *E*
| <bool>
| *E* and *E* | *E* or *E*
| not *E*
| *E* = *E*
| if *E* then *E* else *E*

C ::= <ident> := *E*
| *C*; *C*
| skip

Example terms:

```
x := 3; y := x = 2; z := a && y
x := 0; x := x + 1; y := x
```

Adding Variables: Semantics

- The behavior of programs now depends on the *environment*!

```
x := 3 + 4;
y := x + 5;
b := if y = 12 then true else false;
x := 2;
z := x + 5;
c := b && true
```

← What are the values of the variables at this point?

{b = true, x = 2, y = 12}

Adding Variables: Semantics

- The behavior of programs now depends on the *environment*!
- Small-step relation is $(t, \rho) \to (t', \rho')$, where the environment ρ is a map from variables to values

Greek letter "rho"

"when we look up x in ρ, we find v"

$$\frac{\rho(x) = v}{(x, \rho) \to (v, \rho)}$$

$$\frac{(e_1, \rho) \to (e_1', \rho)}{(e_1 + e_2, \rho) \to (e_1' + e_2, \rho)}$$

$$\frac{(v_1 + v_2 = v)}{(v_1 + v_2, \rho) \to (v, \rho)}$$

Adding Variables: Semantics

- Our programs now have *state*!
- Small-step relation is $(t, \rho) \to (t', \rho')$, where environment ρ is a map from variables to values

$$\frac{(e, \rho) \to (e', \rho)}{(x := e, \rho) \to (x := e', \rho)}$$

$$\frac{(c_1, \rho) \to (c_1', \rho')}{(c_1 ; c_2, \rho) \to (c_1' ; c_2, \rho')}$$

$$\frac{}{(x := v, \rho) \to (\mathtt{skip}, \rho[x \mapsto v])}$$

"set x to v in ρ"

$$\frac{}{(\mathtt{skip}; c_2, \rho) \to (c_2, \rho)}$$

Structural Rule for Sequencing

- What if we had

$$\frac{(c_2, \rho) \to (c_2', \rho')}{(c_1\,;\, c_2, \rho) \to (c_1\,;\, c_2', \rho')}$$

```
x := 3; x := x + 1; x := 4
```

- Exercise: What should this program evaluate to? What could it evaluate to using this rule?

Structural Rule for Sequencing

- What if we had

$$\frac{(c_2, \rho) \to (c_2', \rho')}{(c_1\,;\, c_2, \rho) \to (c_1\,;\, c_2', \rho')}$$

```
   x := 3; x := x + 1; x := 4
```

→ `x := x + 1; x := 4`, {x = 3} (by rule 1)

→ `x := x + 1`, {x = 4} (by rule 2)

→ `skip`, {x = 5}

Adding Variables: Big-Step Semantics

- What does an expression evaluate to? $(e, \rho) \Downarrow v$
- What does a command evaluate to? $(c, \rho) \Downarrow \rho'$

$$\frac{\rho(x) = v}{(x, \rho) \Downarrow v} \qquad \frac{(e_1, \rho) \Downarrow v_1 \quad (e_2, \rho) \Downarrow v_2 \quad (v_1 + v_2 = v)}{(e_1 \mathbf{+} e_2, \rho) \Downarrow v}$$

$$\frac{(e, \rho) \Downarrow v}{(x := e, \rho) \Downarrow \rho[x \mapsto v]} \qquad \frac{(c_1, \rho) \Downarrow \rho' \quad (c_2, \rho') \Downarrow \rho''}{(c_1 ; c_2, \rho) \Downarrow \rho''}$$

$$\frac{}{(\texttt{skip}, \rho) \Downarrow \rho}$$

Adding Variables: Hybrid Semantics

- Expressions don't change the state, commands do
- So we might only care about intermediate steps for commands

$$\frac{\rho(x) = v}{(x, \rho) \Downarrow v}$$

$$\frac{(e_1, \rho) \Downarrow v_1 \quad (e_2, \rho) \Downarrow v_2 \quad (v_1 + v_2 = v)}{(e_1 \mathbf{+} e_2, \rho) \Downarrow v}$$

$$\frac{(e, \rho) \Downarrow v}{(x := e, \rho) \to (\texttt{skip}, \rho[x \mapsto v])}$$

$$\frac{(c_1, \rho) \to (c_1', \rho')}{(c_1 ;\, c_2, \rho) \to (c_1' ;\, c_2, \rho')}$$

$$\frac{}{(\texttt{skip};\, c_2, \rho) \to (c_2, \rho)}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Adding Variables: Interpreter

- Follow the big-step semantics

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

$$\frac{\rho(x) = v}{(x, \rho) \Downarrow v}$$

Adding Variables: Interpreter

- Follow the big-step semantics

let rec eval_exp (e : exp) (r : env) : value option =

- What is the env type? It's a *map* that supports two operations: *lookup* and *update*

$$\frac{\rho(x) = v}{(x, \rho) \Downarrow v} \qquad \frac{(e, \rho) \Downarrow v}{(x := e, \rho) \Downarrow \rho[x \mapsto v]}$$

Adding Variables: Interpreter

- Follow the big-step semantics

```
let rec eval_exp (e : exp) (r : env) : value option =
  match e with
  | Id x ->
  | Add (e1, e2) ->
  | …
```

$$\frac{\rho(x) = v}{(x, \rho) \Downarrow v}$$

Adding Variables: Interpreter

- Follow the big-step semantics

```
let rec eval_exp (e : exp) (r : env) : value option =
  match e with
  | Id x ->
  | Add (e1, e2) ->
  | …
```

$$\frac{\rho(x) = v}{(x, \rho) \Downarrow v}$$

Adding Variables: Interpreter

- Follow the big-step semantics

```
let rec eval_exp (e : exp) (r : env) : value option =
  match e with
  | Id x -> lookup r x
  | Add (e1, e2) ->
  | …
```

$$\frac{\rho(x) = v}{(x, \rho) \Downarrow v}$$

Adding Variables: Interpreter

- Follow the big-step semantics

$$\frac{(e_1, \rho) \Downarrow v_1 \quad (e_2, \rho) \Downarrow v_2 \quad (v_1 + v_2 = v)}{(e_1 + e_2, \rho) \Downarrow v}$$

```
let rec eval_exp (e : exp) (r : env) : value option =
  match e with
  | Add (e1, e2) ->
     (match eval_exp e1 r, eval_exp e2 r with
      | Some (IntVal i1), Some (IntVal i2) -> Some (IntVal (i1 + i2))
      | _, _ -> None)
```

Adding Variables: Interpreter

```
let rec eval_cmd (c : cmd) (r : env) : env option =
  match c with
  | Assign (x, e) ->
  | Seq (c1, c2) ->
  | Skip ->
```

$$\frac{(e, \rho) \Downarrow v}{(x := e, \rho) \Downarrow \rho[x \mapsto v]}$$

Adding Variables: Interpreter

```
let rec eval_cmd (c : cmd) (r : env) : env option =
  match c with
  | Assign (x, e) ->
     (match eval_exp e r with



  | Seq (c1, c2) ->
  | Skip ->
```

$$\frac{(e, \rho) \Downarrow v}{(x := e, \rho) \Downarrow \rho[x \mapsto v]}$$

Adding Variables: Interpreter

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

$$\frac{(e, \rho) \Downarrow v}{(x := e, \rho) \Downarrow \rho[x \mapsto v]}$$

Adding Variables: Interpreter

```
let rec eval_cmd (c : cmd) (r : env) : env option =
  match c with
  | …
  | Seq (c1, c2) ->
  | Skip ->
```

$$\frac{(c_1, \rho) \Downarrow \rho' \quad (c_2, \rho') \Downarrow \rho''}{(c_1\,;\,c_2, \rho) \Downarrow \rho''}$$

Exercise: How would you implement this rule in OCaml?

Adding Variables: Interpreter

```
let rec eval_cmd (c : cmd) (r : env) : env option =
  match c with
  | …
  | Seq (c1, c2) ->
      match eval_cmd c1 r with
      | Some r’ -> eval_cmd c2 r’

  | Skip ->
```

$$\frac{(c_1, \rho) \Downarrow \rho' \quad (c_2, \rho') \Downarrow \rho''}{(c_1 ; c_2, \rho) \Downarrow \rho''}$$

Exercise: How would you implement this rule in OCaml?

Adding Variables: Interpreter

```
let rec eval_cmd (c : cmd) (r : env) : env option =
  match c with
  | …
  | Seq (c1, c2) ->
      (match eval_cmd c1 r with
      | Some r’ -> eval_cmd c2 r’
      | None -> None)
  | Skip ->
```

$$\frac{(c_1, \rho) \Downarrow \rho' \quad (c_2, \rho') \Downarrow \rho''}{(c_1\,;\, c_2, \rho) \Downarrow \rho''}$$

Adding Variables: Interpreter

```
let rec eval_cmd (c : cmd) (r : env) : env option =
  match c with
  | …
  | Seq (c1, c2) ->
      (match eval_cmd c1 r with
      | Some r’ -> eval_cmd c2 r’
      | None -> None)
  | Skip -> Some r
```

$$\frac{}{(\texttt{skip}, \rho) \Downarrow \rho}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Adding Variables: Interpreter

What does the program `x := 2; y := x + 3` evaluate to?

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

Adding Variables: Interpreter

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

let rec eval_cmd (c : cmd) (r : env) : env option =
  match c with
| Seq (c1, c2) ->
      (match eval_cmd c1 r with
      | Some r’ -> eval_cmd c2 r’
      | None -> None)
```

Adding Variables: Interpreter

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

let rec eval_cmd (c : cmd) (r : env) : env option =
  match c with
| Seq (c1, c2) ->
    (match eval_cmd (Assign ("x", Int 2)) empty_env with
    | Some r' -> eval_cmd (Assign ("y", Add (Ident "x", Int 3))) r'
    | None -> None)
```

Adding Variables: Interpreter

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

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

```
| Assign (x, e) ->
    (match eval_exp e r with
    | Some v -> Some (update r x v)
```

Adding Variables: Interpreter

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

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

| Assign (x, e) ->
    (match eval_exp (Int 2) empty_env with
    | Some v -> Some (update empty_env "x" v)
```

Adding Variables: Interpreter

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

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

| Assign (x, e) ->
    (match **Some (IntVal 2)** with
    | Some v -> Some (update empty_env “x” v)

Adding Variables: Interpreter

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

match **Some {“x” = IntVal 2}** with
    | Some r’ -> eval_cmd (Assign (“y”, Add (Ident “x”, Int 3))) r’

| Assign (x, e) ->
    (match **Some (IntVal 2)** with
    | Some v -> Some (update empty_env “x” v)

Adding Variables: Interpreter

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

eval_cmd (Assign (“y”, Add (Ident “x”, Int 3))) {“x” = IntVal 2}

| Assign (x, e) ->
    (match eval_exp (Add (Ident “x”, Int 3)) {“x” = IntVal 2} with
    | Some v -> Some (update {“x” = IntVal 2} “y” v)
```

Adding Variables: Interpreter

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

eval_cmd (Assign (“y”, Add (Ident “x”, Int 3))) {“x” = IntVal 2}

| Assign (x, e) ->
    (match Some (IntVal 5) with
    | Some v -> Some (update {“x” = IntVal 2} “y” v)
```

Adding Variables: Interpreter

```
let res = eval_cmd (Seq (Assign (“x”, Int 2),
                        Assign (“y”, Add (Ident “x”, Int 3)))) empty_env;;
```

Some {“x” = IntVal 2, “y” = IntVal 5}

```
match res with Some r -> lookup r “y” | None -> None;;
- : value option = Some (IntVal 5)
```

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Building Proof Trees

$$\frac{e \to e'}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$\frac{}{\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \to e_1}$$

$$\frac{}{\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \to e_2}$$

$$\frac{e_1 \to e_1'}{e_1\ \mathbf{op}\ e_2 \to e_1'\ \mathbf{op}\ e_2}$$

$$\frac{e_2 \to e_2'}{e_1\ \mathbf{op}\ e_2 \to e_1\ \mathbf{op}\ e_2'}$$

$$\frac{(v_1 \oplus v_2 = v)}{v_1\ \mathbf{op}\ v_2 \to v}$$

where $\oplus$ implements **op**

- Exercise: Write a proof tree for the first step that the following program takes: `if 1+2=3 then 2*2 else 7`

$$\frac{(v_1 \oplus v_2 = v)}{v_1 \ \textbf{op}\ v_2 \to v}$$

where $\oplus$ implements **op**

$$\cfrac{\cfrac{\cfrac{}{1 + 2 \to 3}}{1 + 2 = 3 \to 3 = 3}}{\texttt{if 1+2=3 then 2*2 else 7} \to \text{if } 3 = 3 \text{ then } 2 * 2 \text{ else } 7}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

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 τ in context Γ"
- Γ is a partial function from identifiers to types

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{(n \text{ is a number})}{\Gamma \vdash n : \text{int}} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

- Where does Γ come from?

IMP with Declarations: Syntax

E ::= …

C ::= …

D ::= T <ident> | D; D

T ::= `int` | `bool`

P ::= D; C

Example:

```
int x;
bool y;
int z;
x := 3;
y := (x = 4);
z := if y then 2 else 4
```

IMP with Declarations: Types

- $d : \Gamma$ means "d constructs type context Γ"

$$\frac{}{\texttt{int}\ x : \{x : \text{int}\}}$$

$$\frac{}{\texttt{bool}\ x : \{x : \text{bool}\}}$$

$$\frac{d_1 : \Gamma_1 \quad d_2 : \Gamma_2}{d_1\,;\ d_2 : \Gamma_1 \uplus \Gamma_2}$$

$$\frac{d : \Gamma \quad \Gamma \vdash c : \text{ok}}{d;\ c : \text{ok}}$$

IMP with Declarations: Semantics

$$\frac{}{d; c \to (c, \emptyset)}$$

$$\frac{(c, \emptyset) \Downarrow \rho}{d; c \Downarrow \rho}$$

```
eval_prog p =
  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)
```

Typing Variables, Approach #3

- New typing judgment: $\Gamma \vdash e : \tau$, "e has type τ in context Γ"
- Γ is a partial function from identifiers to types

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{(n \text{ is a number})}{\Gamma \vdash n : \text{int}} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

- Type context Γ can come from variable declarations
- Or we can build it up as we typecheck assignments, etc. ("type inference")

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.