CS 476 – Programming Language Design

William Mansky

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Adding Functions

- With variables, assignment, declarations, and control flow, we have a simple imperative language
- Last major feature of almost every imperative language: functions

Adding Functions

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x = 5; int z;
      z := add(1, 2);
      return x
}
```

- Declare function, then call it
- Function declaration has argument and return types, can return a value
- Can declare local variables inside function body
- Can call functions inside function body (even recursively!)

Functions: Syntax of Definitions

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x; int z;
      x := add(1, 2);
      return x
}
```

E ::= … C ::= …

T ::= …

F ::=

Functions: Syntax of Definitions

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x; int z;
      x := add(1, 2);
      return x
}
```

E ::= … C ::= …

T ::= …

F ::= *type name* (*args*)
      { *body* }

Functions: Syntax of Definitions

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x; int z;
      x := add(1, 2);
      return x
}
```

$E ::= \ldots$ $C ::= \ldots$

$T ::= \ldots$

$F ::= T\ \mathit{name}\ (\mathit{args})$
$\quad\quad \{\ \mathit{body}\ \}$

Functions: Syntax of Definitions

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x; int z;
      x := add(1, 2);
      return x
}
```

E ::= … C ::= …

T ::= …

F ::= T <id> (*args*)
{ *body* }

Functions: Syntax of Definitions

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x; int z;
      x := add(1, 2);
      return x
}
```

E ::= … C ::= …

T ::= …

F ::= T <id> (T <id>, …, T <id>)
{ *body* }

Functions: Syntax of Definitions

```
int add(int x, int y){
     int r;
     r := x + y;
     return r
}

int main(){
     int x; int z;
     x := add(1, 2);
     return x
}
```

E ::= … C ::= …

T ::= …

F ::= T <id> (T <id>, …, T <id>)
{ D; C }

Functions: Syntax of Definitions

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x; int z;
      x := add(1, 2);
      return x
}
```

E ::= …   *C* ::= …

T ::= …

D ::= *T* <id>; …; *T* <id>

F ::= *T* <id> (*T* <id>, …, *T* <id>)
  { *D*; *C* }

Functions: Syntax of Definitions

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x; int z;
      x := add(1, 2);
      return x
}
```

E ::= … C ::= …

T ::= …

D ::= T <id>; …; T <id>

F ::= T <id> (T <id>, …, T <id>)
  { D; C }

P ::= F … F

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Functions: Syntax of Calls

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x; int z;
      x := add(1, 2);
      return x
}
```

E ::= …

C ::= … | return *E*

T ::= …

D ::= *T* <id>; …; *T* <id>

F ::= *T* <id> (*T* <id>, …, *T* <id>) { *D*; *C* }

P ::= *F* … *F*

Exercise: should function calls be commands or expressions?

Functions: Syntax of Calls

```
int add(int x, int y){
    int r;
    r := x + y;
    return r
}

int main(){
    int x; int z;
    x := add(1, 2);
    return x
}
```

E ::= … | <id> (E, …, E)

C ::= … | return E

D ::= T <id>; …; T <id>

F ::= T <id> (T <id>, …, T <id>) { D; C }

P ::= F … F

Exercise: should function calls be commands or expressions?

Functions: Syntax of Calls

```
int add(int x, int y){
    int r;
    r := x + y;
    return r
}

int main(){
    int x; int z;
    x := add(1, 2);
    return x
}
```

E ::= … C ::= … | **return** E

| <id> := <id> (E, …, E)

D ::= T <id>; …; T <id>

F ::= T <id> (T <id>, …, T <id>) { D; C }

P ::= F … F

Exercise: should function calls be commands or expressions?

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Functions: Types

$$\frac{?}{\Gamma \vdash x := f(e_1, \ldots, e_n) : \text{ok}}$$

- ~~Exercise: What do we need to check to make sure a function call is type-correct?~~

Functions: Types

$$\frac{?}{\Gamma \vdash x := f(e_1, \ldots, e_n) : \mathrm{ok}}$$

- f is a declared function
- Each of $e_1, \ldots e_n$ has the right type
- The return type of f matches the type of x

- We can store information about function signatures in Γ

Functions: Types

```
int f(int x, int y){ return x + y }
```

`f` is a function with return type `int`, arguments `int x` and `int y`

$$f : \mathrm{int}(\mathrm{int}\ x, \mathrm{int}\ y)$$

- We can store this information in Γ

Functions: Types

$$\frac{\Gamma(f) = \tau(\tau_1\ x_1, \ldots, \tau_n\ x_n)}{\Gamma \vdash x := f(e_1, \ldots e_n) : \mathrm{ok}}$$

- f is a declared function
- Each of $e_1, \ldots e_n$ has the right type
- The return type of f matches the type of x

Processing Function Declarations

```
int f(int x, int y){ return x + y }
```

add f : int(int x, int y) to Γ

$$\frac{?}{\underbrace{\Gamma}_{\text{old context}} \vdash \underbrace{\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\{\ c\ \}}_{\text{function declaration}} : \underbrace{\Gamma[f \mapsto \tau(\tau_1\ x_1, \ldots, \tau_n\ x_n)]}_{\text{new context}}}$$

Processing Function Declarations

```
int f(int x, int y){ return x + y }
```

$$\frac{\Gamma[x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n] \vdash c : \mathrm{ok}}{\Gamma \vdash \tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\{\ c\ \} : \Gamma[f \mapsto \tau(\tau_1\ x_1, \ldots, \tau_n\ x_n)]}$$

- The function's parameters should be available in the function body
- And so should previously declared functions and global variables
- (Mutually recursive functions would require an extra step)

Functions: Return

```
int f(int x, int y){ return x + y }
```

$$\frac{\Gamma[x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n] \;\vdash c : \text{ok}}{\Gamma \vdash \tau\, f(\tau_1\, x_1, \dots, \tau_n\, x_n)\{\, c \,\} : \Gamma[f \mapsto \tau(\tau_1\, x_1, \dots, \tau_n\, x_n)]}$$

$$\frac{\Gamma \vdash e : \tau \quad \text{current return type is } \tau}{\Gamma \vdash \texttt{return}\; e : \text{ok}}$$

Functions: Return

```
int f(int x, int y){ return x + y }
```

$$\frac{\Gamma[x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n], f \ \vdash c : \text{ok}}{\Gamma \vdash \tau\, f(\tau_1\, x_1, \dots, \tau_n\, x_n)\{\, c \,\} : \Gamma[f \mapsto \tau(\tau_1\, x_1, \dots, \tau_n\, x_n)]}$$

$$\frac{\Gamma \vdash e : \tau \quad \text{current return type is } \tau}{\Gamma \vdash \texttt{return}\ e : \text{ok}}$$

Functions: Return

```
int f(int x, int y){ return x + y }
```

$$\frac{\Gamma[x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n], f \ \vdash c : \mathrm{ok}}{\Gamma \vdash \tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\{\ c\ \} : \Gamma[f \mapsto \tau(\tau_1\ x_1, \ldots, \tau_n\ x_n)]}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma(f) = \tau\ f(\ldots)}{\Gamma, f\ \vdash \texttt{return}\ e : \mathrm{ok}}$$

- Remember which function we're in, so we know the return type

Functions: Return

```
int f(int x, int y){ return x + y }
```

$$\frac{\Gamma[x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n, \texttt{\_\_ret} \mapsto \tau] \vdash c : \mathrm{ok}}{\Gamma \vdash \tau\, f(\tau_1\, x_1, \ldots, \tau_n\, x_n)\{\, c \,\} : \Gamma[f \mapsto \tau(\tau_1\, x_1, \ldots, \tau_n\, x_n)]}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma(\texttt{\_\_ret}) = \tau}{\Gamma \vdash \texttt{return}\; e : \mathrm{ok}}$$

- Use a fake variable to remember the return type

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Functions: Syntax of Calls

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x; int z;
      x := add(1, 2);
      return x
}
```

E ::= …   *C* ::= … | **return** *E*

| <id> := <id> (*E*, …, *E*)

D ::= *T* <id>; …; *T* <id>

F ::= *T* <id> (*T* <id>, …, *T* <id>)
{ *D*; *C* }

P ::= *F* … *F*

Functions: Initial Program State

```
int add(int x, int y){
      int r;
      r := x + y;
      return r
}

int main(){
      int x; int z;
      x := add(1, 2);
      return x
}
```

$P ::= F \ldots F$

- How do we start running a program P?

- Initial configuration of a program

$f_1(params_1)\{\, body_1 \,\}$

…

$f_n(params_n)\{\, body_n \,\}$

is $(\texttt{main()}, [f_i \mapsto (params_i)\{\, body_i \,\}])$

Functions: Initial Program State

```
int add(int x, int y){
    int r;
    r := x + y;
    return r
}

int main(){
    int x; int z;
    x := add(1, 2);
    return x
}
```

- Initial configuration of a program

$f_1(params_1)\{\ body_1\ \}$

…

$f_n(params_1)\{\ body_n\ \}$

is $(\texttt{main()}, [f_i \mapsto (params_i)\{\ body_i\ \}])$

$(\texttt{main()}, [\text{add} \mapsto (\text{x}, \text{y})\{\ \text{int r; …}\ \},$
$\text{main} \mapsto ()\{\ \text{int x; …}\ \}])$

Functions: Semantics of Calls

$$\frac{}{(x := f(e_1, \ldots, e_n), \rho) \Downarrow ?}$$

- Evaluate the arguments $e_1, \ldots, e_n$
- Look up f in ρ
- Execute the body of f and produce a return value
- Assign the return value to x

Functions: Semantics of Calls

$$\frac{(e_1, \rho) \Downarrow v_1 \quad \dots \quad (e_n, \rho) \Downarrow v_n}{(x := f(e_1, \dots, e_n), \rho) \Downarrow \rho[x \mapsto v]}$$

- Evaluate the arguments $e_1, \dots, e_n$
- Look up f in ρ
- Execute the body of f and produce a return value
- Assign the return value to x

Functions: Semantics of Calls

$$\frac{(e_1,\rho) \Downarrow v_1 \;\; \dots \;\; (e_n,\rho) \Downarrow v_n \quad (\rho(f) = f(x_1,\dots,x_n)\{\, c \,\}) \quad (c,\rho) \Downarrow \mathrm{ret}(v)}{(x := f(e_1,\dots,e_n),\rho) \Downarrow \rho[x \mapsto v]}$$

- Evaluate the arguments $e_1, \dots, e_n$
- Look up f in ρ
- Execute the body of f and produce a return value
- Assign the return value to x

Functions: Semantics of Calls

$$\frac{(e_1, \rho) \Downarrow v_1 \;\; \dots \;\; (e_n, \rho) \Downarrow v_n \quad (\rho(f) = f(x_1, \dots, x_n)\{\ c\ \}) \quad (c, \rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]) \Downarrow \mathrm{ret}(v)}{(x := f(e_1, \dots, e_n), \rho) \Downarrow \rho[x \mapsto v]}$$

- Evaluate the arguments $e_1, \dots, e_n$
- Look up f in ρ
- Execute the body of f and produce a return value
  - with the arguments passed in for the parameters
- Assign the return value to x

Functions: Semantics of Calls

$$\frac{(e_1, \rho) \Downarrow v_1 \quad \dots \quad (e_n, \rho) \Downarrow v_n \qquad (\rho(f) = f(x_1, \dots, x_n)\{\ c\ \}) \qquad (c, \rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]) \Downarrow \mathrm{ret}(v)}{(x := f(e_1, \dots, e_n), \rho) \Downarrow \rho[x \mapsto v]}$$

$$\frac{(e, \rho) \Downarrow v}{(\texttt{return}\ e, \rho) \Downarrow \mathrm{ret}(v)}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Functions: Small-Step Semantics of Calls

$$\overline{(x := f(e_1, \ldots, e_n), \rho) \rightarrow ?}$$

- Evaluate the arguments $e_1, \ldots, e_n$
- Look up f in ρ
- Execute the body of f and produce a return value
  - with the arguments passed in for the parameters
- Assign the return value to x
- Exercise: What's the first step this function call should take?

Functions: Semantics of Calls

$$\frac{(e_1, \rho) \Downarrow v_1 \quad \dots \quad (e_n, \rho) \Downarrow v_n \quad (\rho(f) = f(x_1, \dots, x_n)\{\, c \,\})}{\begin{array}{c}(x := f(e_1, \dots, e_n), \rho) \to \\ (c; x := ?, \rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n])\end{array}}$$

- Evaluate the arguments $e_1, \dots, e_n$
- Look up f in ρ
- Execute the body of f and produce a return value
- Assign the return value to x

Functions: Semantics of Calls

$$\frac{(e_1, \rho) \Downarrow v_1 \quad \dots \quad (e_n, \rho) \Downarrow v_n \qquad (\rho(f) = f(x_1, \dots, x_n)\{\, c \,\})}{\begin{array}{c}(x := f(e_1, \dots, e_n), k, \rho) \rightarrow \\ (c,\ (\rho, x) :: k,\ \rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n])\end{array}}$$

- Small-step relation is now $(c, k, \rho) \rightarrow (c', k', \rho')$, where k is a *stack* of environments
- A stack is a list of *stack frames* (ρ, x), where ρ is the caller's environment and x is the variable that gets the return value
- The :: operator connects the top frame to the rest of the stack

Functions: Semantics of Calls

$$\frac{(e_1, \rho) \Downarrow v_1 \;\; \dots \;\; (e_n, \rho) \Downarrow v_n \quad (\rho(f) = f(x_1, \dots, x_n)\{\, c \,\})}{\begin{array}{c}(x := f(e_1, \dots, e_n), k, \rho) \rightarrow \\ (c,\; (\rho, x) :: k,\; \rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n])\end{array}}$$

$$\frac{(e, \rho) \Downarrow v}{(\texttt{return}\; e,\; (\rho_0, x) :: k,\; \rho) \rightarrow (\texttt{skip},\; k,\; \rho_0[x \mapsto v])}$$

- Small-step relation is now $(c, k, \rho) \rightarrow (c', k', \rho')$, where k is a *stack* of environments

Functions: Semantics of Calls

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

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

- Small-step relation is now $(c, k, \rho) \rightarrow (c', k', \rho')$, where k is a *stack* of environments

Functions: Semantics of Calls

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

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

- Small-step relation is now $(c, k, \rho) \to (c', k', \rho')$, where k is a *stack* of environments

Functions: Semantics of Calls

$(\texttt{y:=f(x-1);\ \ z:=(y=x)}, [], \rho_1) \rightarrow ?$

Exercise: What is the next step this program takes?

$$\frac{(e_1, \rho) \Downarrow v_1 \;\; \dots \;\; (e_n, \rho) \Downarrow v_n \quad (\rho(f) = (x_1, \dots, x_n)\{\, c \,\})}{\begin{array}{c}(x := f(e_1, \dots, e_n), k, \rho) \rightarrow \\ (c,\; (\rho, x) :: k,\; \rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n])\end{array}}$$

where $\rho_1 = \{\texttt{f} \mapsto (\texttt{x})\{\,\texttt{return x+1}\,\}, \texttt{x} \mapsto 3\}$

Functions: Semantics of Calls

$(\texttt{y:=f(x-1); z:=(y=x)}, [], \rho_1) \to$
$(\texttt{return x+1; z:=(y=x)}, [(\rho_1, \texttt{y})], \{\texttt{f} \mapsto \cdots, \texttt{x} \mapsto 2\}) \to$
$(\texttt{skip; z:=(y=x)}, [], \{\texttt{f} \mapsto \cdots, \texttt{x} \mapsto 3, \texttt{y} \mapsto 3\})$

$$\frac{(e_1, \rho) \Downarrow v_1 \quad \dots \quad (e_n, \rho) \Downarrow v_n \quad (\rho(f) = (x_1, \dots, x_n)\{\, c \,\})}{\begin{array}{c}(x := f(e_1, \dots, e_n), k, \rho) \to \\ (c,\ (\rho, x) :: k, \rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n])\end{array}}$$

where $\rho_1 = \{\texttt{f} \mapsto (\texttt{x})\{\, \texttt{return x+1} \,\}, \texttt{x} \mapsto 3\}$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Homework 4 Overview

- Lists in OCaml
- Useful types: entry, env, stack, config
- eval_exp and step_cmd
- run_config and testing

Functions: The Hidden Stack

- Big-step and small-step describe the same behavior
- Small-step semantics now need a whole extra piece of state
- And that state corresponds to a feature of real language implementations!

$$\frac{(e_1, \rho) \Downarrow v_1 \quad \ldots \quad (e_n, \rho) \Downarrow v_n \quad (\rho(f) = (x_1, \ldots, x_n)\{\ c\ \}) \qquad (c, \rho[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]) \Downarrow \mathrm{ret}(v)}{(x := f(e_1, \ldots, e_n), \rho) \Downarrow \rho[x \mapsto v]}$$

Functions: Interpreter vs. Compiled

$$\frac{(e_1, \rho) \Downarrow v_1 \quad \dots \quad (e_n, \rho) \Downarrow v_n \quad \rho(f) = (x_1, \dots, x_n)\{\, c \,\} \qquad (c, \rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]) \Downarrow \mathrm{ret}(v)}{(x := f(e_1, \dots, e_n), \rho) \Downarrow \rho[x \mapsto v]}$$

let rec eval_cmd c r = match c with

| Call (x, f, es) ->

Functions: Interpreter vs. Compiled

$$\frac{(e_1, \rho) \Downarrow v_1 \quad \dots \quad (e_n, \rho) \Downarrow v_n \quad \rho(f) = (x_1, \dots, x_n)\{\, c \,\} \quad (c, \rho[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]) \Downarrow \mathrm{ret}(v)}{(x := f(e_1, \dots, e_n), \rho) \Downarrow \rho[x \mapsto v]}$$

```
let rec eval_cmd c r = match c with
| Call (x, f, es) -> match eval_exps es r with Some vs ->
    match lookup r f with Some (Fun (xs, c)) ->
    match eval_cmd c (add_args r xs vs)  with
    | Some (Ret v) -> Some (State (update r x v))
```

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Imperative Languages

- Arithmetic and boolean expressions
- Variables and assignment
- Control flow (conditionals, loops)
- Variable declarations
- Function declarations and calls
- There’s more, but we’ve covered the essentials!

- Next up: object-oriented languages

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.