Dynamic and Static Semantics

February 7, 2001

CS 131: Programming Languages

Review: A Language with Local Definitions

- Abstract Syntax

v ::= n | **tt** | **ff** (values)

e ::= v | e + e | e < e (expressions)

| **if** e **then** e **else** e

| x

| **let** x **be** e **in** e

Review: Dynamic Semantics

$$\frac{}{n_1 + n_2 \to n_1 \oplus n_2}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \qquad \frac{e_2 \to e_2'}{v + e_2 \to v + e_2'}$$

$$\frac{}{n_1 \le n_2 \to n_1 \textcircled{$\le$} n_2}$$

$$\frac{e_1 \to e_1'}{e_1 \le e_2 \to e_1' \le e_2} \qquad \frac{e_2 \to e_2'}{v \le e_2 \to v \le e_2'}$$

Review: Dynamic Semantics

$$\frac{e_1 \to e_1'}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \to \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3}$$

$$\frac{}{\textbf{if tt then } e_2 \textbf{ else } e_3 \to e_2}$$

$$\frac{}{\textbf{if ff then } e_2 \textbf{ else } e_3 \to e_3}$$

Review: Dynamic Semantics

$$\frac{e_1 \to e_1'}{\texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2 \to \texttt{let } x \texttt{ be } e_1' \texttt{ in } e_2}$$

$$\frac{}{\texttt{let } x \texttt{ be } v_1 \texttt{ in } e_2 \to e_2[x \to v_1]}$$

Examples

- **let** x **be** 3+4 **in** (1+2)+x
  - → **let** x **be** 7 **in** (1+2)+x
  - → (1+2)+7
  - → 3+7
  - → 10
- **let** x **be** 1+1 **in** x+x
  - →
  - →
  - →

Review: Alternative Dynamic Semantics

- What if instead we had this single rule for **let**?

$$\frac{}{\texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2 \to e_2[x \to e_1]}$$

- **let** x **be** 1+1 **in** x+x
  - →
  - →
  - →
  - →

Adding Function Values

- Abstract Syntax

v ::= n | **tt** | **ff** (values)
  | **fix** $f(x)$ **is** e

e ::= v | e + e | $e \leq e$ (expressions)
  | **if** e **then** e **else** e
  | x | **let** x **be** e **in** e
  | e e

Recursive Function Values

- The function value

  fix *f*(*x*) **is** *e*

  corresponds roughly to the SML code

```
let
   fun f(x)=e
in
   f
end
```

- In particular, note that the scope of `f` is e and the scope of `x` is e, and that's it.
- This does not permit other code to refer to this function as `f`!

Conventions

- The body of a function is assumed to extend as far as possible:

```
  fix f(x) is x+x
= fix f(x) is (x+x)
≠ (fix f(x) is x)+x
```

Example Expressions

```
(fix f(x) is x+1) 3

(fix f(x) is f(x+1)) 3

let succ be (fix f(x) is x+1)
in let n be 4 in
   (succ n) + (succ (n+1))

let fib be
     fix g(x) is if x≤0 then 1 else
                    if x≤1 then 1 else
                       g(x+(~1)) + g(x+(~2))
in
    fib (1+1)           (* not g(1+1) ! *)
```

Dynamic Semantics

- Add the rules

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'}$$

$$\frac{}{(\textbf{fix}\ \mathtt{f(x)}\ \textbf{is}\ \mathtt{e})\ v \to e[x \to v][\mathtt{f} \to \mathtt{fix\ f(x)\ is\ e}]}$$

Back to Stuck Programs

- Recall: some programs have not reached a final state, but cannot make progress

```
3 + tt
if tt < ff then 3 else 5
```

- Such programs are "about to go wrong"
  - About to apply an operation using the wrong sort of operands
- We can prevent such problems with a type system
  - Let's start with just the type system for arithmetic and conditionals.

A Simple Static Semantics

- We start with just two types

$$\boxed{t ::= \texttt{int} \mid \texttt{bool}}$$

- We will define a typing relation

$$\boxed{\texttt{e : t}}$$

- A type system is frequently called the "static semantics" of the language.

Static Semantic Rules

$$\frac{}{\texttt{n : int}} \qquad \frac{}{\texttt{tt : bool}} \qquad \frac{}{\texttt{ff : bool}}$$

$$\frac{}{e_1 + e_2 :}$$

$$\frac{}{e_1 < e_2 :}$$

$$\frac{}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 :}$$

Claim: Type Soundness

- Well-typed programs (i.e., programs that have some type) can't get stuck
  - That is, must either eventually reach a value or fail to terminate
- Why is this important? How to prove this?
  - See next lecture

But...

- Some programs wouldn't get stuck but still don't typecheck

```
(if ff then tt else 4) + 1
```

- For any interesting language, a type system preventing all bad programs also rejects programs that would run without problems.
- Research topic: type systems that catch as many errors as possible, but don't reject useful programs

Changes to the Static Semantics

- Well-typedness is now *context-sensitive*
  - What is the type of `x` ?
  - Is `x+3` well-typed?

```
let x be 4 in x+3
```

```
let x be tt in x+3
```

- To determine types we need to know the types of the *free* variables

Conditional Judgments

- The typing judgments are now *conditional*
  "If `x : int` then `x+3 : int`"
  "If `x : bool` then `if x then 3 else 4 : int`"

- Given assumptions about the types of variables, we can conclude code is well-typed

$$\texttt{x:int} \vdash \texttt{x+3 : int}$$

$$\texttt{x:bool} \vdash \texttt{if x then 3 else 4 : int}$$

Typing Environments

- An environment is a lookup table for variables
  - A *type environment* associates variables with types
- Notation
  - We use Γ to denote an arbitrary type environment.
  - Type environments can be written as a list
    - e.g., `x:int, y:bool, z:int`
  - The notation $\Gamma(\texttt{x})$ gives the type of x
  - The notation $\Gamma, \texttt{x:int}$ is the extension of Γ that maps x to `int`

Updated Static Semantics

$$\frac{}{\Gamma \vdash \mathtt{n} : \mathtt{int}} \qquad \frac{}{\Gamma \vdash \mathtt{tt} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{ff} : \mathtt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \qquad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash e_1 + e_2 : \mathtt{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{int} \qquad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash e_1 < e_2 : \mathtt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{bool} \qquad \Gamma \vdash e_2 : \mathtt{t} \qquad \Gamma \vdash e_3 : \mathtt{t}}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \mathtt{t}}$$

The Interesting Rules

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, \mathtt{x}{:}t_1 \vdash e_2 : t_2}{\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2 : t_2}$$

Functions

t ::= int | bool | t -> t

$$\frac{\Gamma \vdash e_1 : t_2\text{->}t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\ e_2 : t}$$

$$\frac{\Gamma, x{:}t_1, f{:}t_1\text{->}t_2 \vdash e : t_2}{\Gamma \vdash \mathbf{fix}\ f(x)\ \mathbf{is}\ e : t_1\text{->}t_2}$$