CSEP 505: Programming Languages

Lecture 8
February 26, 2015

v ::= n | *op* | **λ**(*x*:t).e

e ::= v | **if** e e e | *x* | e e | let x = e in e

t ::= num | bool | t → t | [t] | t x t

$$\Gamma \vdash x : \Gamma(x) \qquad \text{(VAR)}$$

$$\Gamma \vdash n : \text{num} \qquad \text{(NUM)}$$

$$\frac{\Gamma \vdash e_c : \text{bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash (\text{if } e_c \ e_t \ e_f) : \tau} \qquad \text{(IF)}$$

$$\frac{\Gamma \vdash e : \tau_x \qquad \Gamma, x : \tau_x \vdash e_b : \tau}{\Gamma \vdash (\text{let } x = e \text{ in } e_b) : \tau} \qquad \text{(LET)}$$

$$\frac{\Gamma, x : \tau_a \vdash e : \tau}{\Gamma \vdash (\lambda x : \tau_a . e) : \tau_a \to \tau} \qquad (\lambda)$$

$$\frac{\Gamma \vdash e_f : \tau_a \to \tau \qquad \Gamma \vdash e_a : \tau_a}{\Gamma \vdash (e_f \ e_a) : \tau} \qquad \text{(APP)}$$

v ::= n | *op* | **λ**(*x*:t).e

e ::= v | **if** e e e | *x* | e e | let x = e in e

t ::= num | bool | t → t | [t] | t x t

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\text{pair } e_1\ e_2) : \tau_1 \times \tau_2} \quad (\textsc{pair})$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash (\text{fst } e) : \tau_1} \quad (\textsc{fst})$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash (\text{snd } e) : \tau_2} \quad (\textsc{snd})$$

$$\Gamma \vdash \text{empty} : [\tau] \quad (\textsc{empty})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : [\tau]}{\Gamma \vdash (\text{cons } e_1\ e_2) : [\tau]} \quad (\textsc{cons})$$

$$\frac{\Gamma \vdash e : [\tau]}{\Gamma \vdash (\text{first } e) : \tau} \quad (\textsc{first})$$

$$\frac{\Gamma \vdash e : [\tau]}{\Gamma \vdash (\text{rest } e) : [\tau]} \quad (\textsc{rest})$$

$$\frac{\Gamma \vdash e : [\tau]}{\Gamma \vdash (\text{empty? } e) : \text{bool}} \quad (\textsc{empty?})$$

v ::= n | *op* | λ(*x*:t).e

t ::= num | bool | t → t | [t] | t x t

E ::= [ ] | if E e e | E e | v E

e ::= v | **if** e e e | *x* | e e | let x = e in e

$$E[\texttt{if true } e_t\ e_f] \rightarrow E[e_t] \qquad [\text{if-t}]$$

$$E[\texttt{if false } e_t\ e_f] \rightarrow E[e_f] \qquad [\text{if-f}]$$

$$E[\texttt{op v}] \rightarrow E[\delta(\texttt{op, v})] \qquad [\delta]$$

$$E[(\lambda(\texttt{x:t}).\texttt{e})\ \texttt{v}] \rightarrow E[[\texttt{x} \leftarrow \texttt{v}]\texttt{b}] \qquad [\beta_v]$$

$$E[\texttt{fix } (\lambda(\texttt{x:t}).\texttt{e})] \rightarrow E[[\texttt{x} \leftarrow (\texttt{fix } (\lambda(\texttt{x:t}).\texttt{e}))]\texttt{e}] \qquad [\text{fix}]$$

v ::= n | *op* | **λ**(*x*:t).e

t ::= num | bool | t → t | [t] | t x t

e ::= v | **if** e e e | *x* | e e | let x = e in e

Soundness

“Well-typed programs do not get stuck.”

- Progress
  If e : t, then e is a value or e → e’.

- Preservation
  If e : t and e → e’, then e’ : t.

```
(+ 1 (* 2 3)) : num
```

```
(+ 1 (* 2 false)) :
```

```
(if true 3 succ) :
```

```
empty<a> : [a]
```

```
(fun ([x : num])
  (if (zero? x)
      1
      (* 2 x))) : num → num
```

```
x : [num] |-
(if (empty? x)
    x
    (cons 5 empty)) : [num]
```

```
(fun ([x : bool])
  (if x
      succ
      (+ x)))
```

```
(fun ([x : num])
  (if x
      succ
      (+ x)))
```

```
(with* ([id (fun ([x : bool])
                 x)])
  (pair (id true) (id 3)))
```

```
(with* ([id (fun ([x : num])
                 x)])
  (pair (id true) (id 3)))
```

```
(fun ([f : … ]
      [g : (num → bool)]
      [x : num])
  (pair (f (g x))
        ((f g) x)))
```

```
((fix (fun ([map :                                    ]
            [f : (num → num)]
            [lst : [num]])
   (if (empty? lst)
       empty
       (cons (f (first lst))
             (map f (rest lst))))))
 succ
 (cons 1 (cons 2 (cons 3 empty))))
```

```
(with* ([two (fun ([f : (num → num)]
                   [x : num])
               (f (f x)))]
  [mult
   (fun ([n : (num → num) → num → num]
         [m : (num → num) → num → num]
         [f : (num → num)])
     (n (m f)))])
 (mult two two)) : num → num
```

```
(with* ([two (fun ([f : (num → num)]
                   [x : num])
               (f (f x)))]
  [pow
   (fun ([n : (num → num) → num → num]
         [m : (num → num) → num → num])
     (n m))])

 (pow two two))
```

```
(with* ([two (fun ([f : (num → num)]
                   [x : num])
               (f (f x)))]
  [pow
   (fun ([n : (num → num) → num → num]
         [m : (num → num)])
     (n m))])

 (pow two two))
```

```
(fun ([x : … ])
  (x x))
```

```
(with* ([id
         (fun ([x :    ]) x)])
  (pair (id true) (id 3)))
```

```
(fun ([f : ∀a.(a → a)]
      [g : (num → bool)]
      [x : num])
  (pair (f <bool>
           (g x))
        ((f <num → bool> g) x)))
```

```
(fix
 (∀ab.
  (fun ([map : (a → b) → ([a] → [b])]
        [f : (a → b)]
        [lst : [a]])
   (if (empty? x)
       empty
       (cons (f (first lst))
             (map f (rest lst)))))))
```

```
(fun ([x : ∀a.(a → a)])
  (x <∀a.(a → a)> x))
```

v ::= n | *op* | **λ***x*.e e ::= v | **if** e e e | *x* | e e | let x = e in e | ∀α.e | e<t>

t ::= num | bool | t → t | [t] | t x t | α | ∀α.t

v ::= n | *op* | **λ***x*.e e ::= v | **if** e e e | *x* | e e | let x = e in e | ∀α.e | e<t>

t ::= num | bool | t → t | [t] | t x t | α | ∀α.t

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash (\forall \alpha . e) : \forall \alpha . \tau} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash e : \forall \alpha . \tau'}{\Gamma \vdash e\langle \tau \rangle : [\alpha \leftarrow \tau]\tau'} \quad \text{(T-SPEC)}$$

$$\frac{\Gamma \vdash \tau_a \qquad \Gamma, x : \tau_a \vdash e : \tau}{\Gamma \vdash (\lambda x : \tau_a . e) : \tau_a \rightarrow \tau} \quad (\lambda)$$

```
(fun ([x : (∀α.α → α)])
  (x<(∀α.α → α)> x))
```

```
((fun ([x : (∀α.α → α)])
   (x<(∀α.α → α)> x))
 (∀α.(fun ([x : α]) x)))
```

```
((fun ([x : (∀α.α → α)])
    (x<(∀α.α → α)> x))
 (fun ([x : (∀α.α → α)])
    (x<(∀α.α → α)> x)))
```

v ::= n | *op* | **λ***x*.e e ::= v | **if** e e e | *x* | e e | let x = e in e | ∀α.e | e<t>

t ::= num | bool | t → t | [t] | t x t | α | ∀α.t

$$\Gamma \vdash x : \Gamma(x) \qquad \text{(VAR)}$$

$$\Gamma \vdash n : \text{num} \qquad \text{(NUM)}$$

$$\frac{\Gamma \vdash e_c : \text{bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash (\text{if } e_c \ e_t \ e_f) : \tau} \qquad \text{(IF)}$$

$$\frac{\Gamma \vdash e : \tau_x \qquad \Gamma, x : \tau_x \vdash e_b : \tau}{\Gamma \vdash (\text{let } x = e \text{ in } e_b) : \tau} \qquad \text{(LET)}$$

$$\frac{\Gamma, x : \tau_a \vdash e : \tau}{\Gamma \vdash (\lambda x : \tau_a . e) : \tau_a \to \tau} \qquad (\lambda)$$

$$\frac{\Gamma \vdash e_f : \tau_a \to \tau \qquad \Gamma \vdash e_a : \tau_a}{\Gamma \vdash (e_f \ e_a) : \tau} \qquad \text{(APP)}$$

```
type Type = NumT | BoolT | ArrowT Type Type
          | VarT …
unify :: Type → Type → …
unify BoolT BoolT = …
unify BoolT NumT = …
unify tvar NumT = …
unify tvar tvar′ = …
unify (ArrowT tvar BoolT)
      (ArrowT NumT tvar′) = …
unify tvar (ArrowT NumT tvar′) = …
unify tvar (ArrowT tvar tvar′) = …
```

```
type LVar a = Box (Maybe a)

newLVar :: STR (LVar a)
newLVar = makeBox Nothing

bindLVar :: LVar a → a → STR ()
bindLVar var val =
  contents ← unbox var
  case contents of
    Nothing → setBox var (Just val)
    Just x → fail ("already bound: " ++ (show x))
```

```
type Type = NumT | BoolT | ArrowT Type Type
          | VarT (LVar Type)

resolve :: Type → STR Type
```

```
type Type = NumT | BoolT | ArrowT Type Type
          | VarT (LVar Type)

resolve :: Type → STR Type
resolve tvar@(VarT lvar) =
  do contents ← unbox lvar
     case contents of
       Nothing → return tvar
       Just ty →
```

```
type Type = NumT | BoolT | ArrowT Type Type
          | VarT (LVar Type)

unify :: Type → Type → STR ()
```

```
type Type = NumT | BoolT | ArrowT Type Type
          | VarT (LVar Type)

unify :: Type → Type → STR ()
unify NumT NumT = return ()
unify BoolT BoolT = return ()
unify (ArrowT argT1 resT1) (ArrowT argT2 resT2) =
  do unify argT1 argT2
     unify resT1 resT2
```

```
(fun (x)
  (if (zero? x)
      1
      (* 2 x)))
```

```
(fun (map f lst)
  (if (empty? lst)
      empty
      (cons (f (first lst))
            (map f (rest lst)))))
```

$t_{map} = t_f \rightarrow t_{lst} \rightarrow t_r \qquad t_{lst} = [t_0] \qquad t_r = [t_1]$

$t_f = t_0 \rightarrow t_1$

$t_{expr} = t_{map} \rightarrow t_f \rightarrow t_{lst} \rightarrow t_r$

$= ((t_0 \rightarrow t_1) \rightarrow [t_0] \rightarrow [t_1]) \rightarrow$

$((t_0 \rightarrow t_1) \rightarrow [t_0] \rightarrow [t_1])$

```
(fix (fun (map f lst)
   (if (empty? lst)
       empty
       (cons (f (first lst))
             (map f (rest lst))))))
```

$t_{fix} = (a \rightarrow a) \rightarrow a$

$t_{fun} = ((t_0 \rightarrow t_1) \rightarrow [t_0] \rightarrow [t_1]) \rightarrow$
$((t_0 \rightarrow t_1) \rightarrow [t_0] \rightarrow [t_1])$

a unifies with $((t_0 \rightarrow t_1) \rightarrow [t_0] \rightarrow [t_1])$

$t_{res} = ((t_0 \rightarrow t_1) \rightarrow [t_0] \rightarrow [t_1])$

```
(with* ([map (fix (fun (map f lst)
      (if (empty? x)
          empty
          (cons (f (first lst))
                (map f (rest lst))))))])
; t_map = ((t_0 → t_1) → [t_0] → [t_1])
  (map zero?
; zero? unifies (t_0 → t_1) with (num → bool)
    (map succ (cons 1 (cons 2 empty)))))
; succ fails to unify (t_0 → t_1) with (num → num)
```

v ::= n | *op* | λ(*x*:t).e

e ::= v | **if** e e e | *x* | e e | let x = e in e

t ::= num | bool | t → t | [t] | t x t

$$\text{tyclos}(\tau) = \langle \tau, \text{FreeTypeVars}(\tau) \rangle \qquad \text{(TYPE-CLOSURE)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \text{tyclos}(\tau_1) \vdash e : \tau}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e) : \tau} \qquad \text{(PLET)}$$

$$\frac{\Gamma(x) = \langle \tau, \{\alpha_1, \ldots, \alpha_n\} \rangle \qquad \beta_1, \ldots, \beta_n \text{ fresh}}{\Gamma \vdash x : \tau[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]} \qquad \text{(PVAR)}$$

Type Closures (1st attempt, not quite correct)

```
(with* ([g (fun (f) (f 3))])
  (pair (g succ) (g zero?)))
```

```
(fun (x)
  (with* ([g (fun (f) (f x))])
    (pair (g succ) (g zero?))))
```

$\Rightarrow$: $\forall\alpha.(\alpha \rightarrow (\text{num}, \text{bool}))$

v ::= n | *op* | λ(*x*:t).e

e ::= v | **if** e e e | *x* | e e | let x = e in e

t ::= num | bool | t → t | [t] | t x t

$$\overline{\Gamma}(\tau) = \langle \tau, \text{FreeTypeVars}(\tau) \setminus \text{FreeTypeVars}(\Gamma) \rangle \qquad \text{(TYPE-CLOSURE)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \overline{\Gamma}(\tau_1) \vdash e : \tau}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e) : \tau} \qquad \text{(PLET)}$$

$$\frac{\Gamma(x) = \langle \tau, \{\alpha_1, \ldots, \alpha_n\} \rangle \qquad \beta_1, \ldots, \beta_n \text{ fresh}}{\Gamma \vdash x : \tau[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n]} \qquad \text{(PVAR)}$$

```
(with* ([b (box (fun (x) x))])
  ((unbox b) 3)))
```

```
; b : ∀α.(box (α → α))
(with* ([b (box (fun (x) x))])
  ; α unifies with bool
  (seq (set-box! b not)
       ; α unifies with num: boom!
       ((unbox b) 3)))
```

Value restriction:

```
; b : ∀α.(box (α → α))
(with* ([b (box (fun (x) x))])
  ; α unifies with bool
  (seq (set-box! b not)
    ; α doesn't unify with num: type error!
    ((unbox b) 3)))
```

Concepts

- Soundness (briefly)
- Polymorphic lambda-calculus (a.k.a. System F).
- Logic variables and unification
- Type inference (Hindley-Milner algorithm for let-polymorphism)
- Value restriction for polymorphism & side-effects