March 2, 2016 Type Reconstruction

HW

Type these functions:

(define (abss x) (if (< x 0) (- 0 x) x))

Follow the recursive algorithm on pp 17-18 of Types-2 lecture:

(Note: this example is tricky because of the possibility of using coercion on the - operation. Let's assume we only have int as a numerical type here)

Rule 1 Build a new type for abss in type environment E

Extend $E(x) = \gamma$, $E(abss) = \{\gamma \to \delta\}$

Rule 2. Analyze function body of abss: (if (< x 0) (- 0 x) x)

"f" "e1" "e2" "e3"

---

t1: Analyze (e1, E) for e1 = (< x 0)
(Note consider binary operator less than a function of 2 arguments)

per Rule #17, function application

t11: Analyze (x) ⇒ E(x) = γ
t12: Analyze (0) ⇒ E(0) = int
s11: Analyze (<) ⇒ E(<) = {τ * τ → bool}
get new type variable β
add equ: γ * int → β = τ * τ → bool

①

(derived type of fcn < only using info about arguments' types)

(type of fcn < in environment E)

return β as type of fcn application

t2: Analyze (e2, E) for e2 = (- 0 x)

t21: Analyze (0) ≈ E(0) = int

t22: Analyze (x) ≈ E(x) = γ

s11: Analyze (-) ≈ E(-) = int * int → int
(by simplifying assumption above)

get new type var γ

(2) add eqn: int * γ → μ = int * int → int

return μ as type of function application

---

t3: Analyze (e3, E) for e3 = x, E(x) = δ

s: Analyze (if, E) ≈ E(if) = b * Δ * Δ → Δ

get fresh type variable ψ

Add eqn

(3) (γ * β * μ * γ → ψ) = bool * Δ * Δ → Δ

return ψ as type of fun application

Now Solve eqns (1) (2) (3) to find most general types satisfying these equations

(1) $\gamma * int \to \beta = \gamma * \gamma \to bool$

(2) $int * \gamma \to \Gamma = int * int \to int$

(3) $\beta * \Gamma * \gamma \to \psi = bool * int * int \to int$

(1) $\Rightarrow \beta = bool$

(2) $\Rightarrow \Gamma = int$

(3) $\Rightarrow \gamma = int = \psi$

E(abs) : $\gamma \to \delta$

but ψ is return type of function body of abs

therefore $\psi = \delta = int$

$\to$ E(abs) : $\gamma \to \delta \equiv int \to int$.

Note: You could do this more generally if you used a rule for coercion of int into float & had a type that was the union of int or float, both of which are valid to use with arithmetic & comparison operators.

Fundion to type

```
(define (app L1 L2)
    (if (null? L1) L2)
        (cons (car L1) (app (cdr L1) L2 $
```

(where $ means close all open left parentheses)
(Note: This is another recursive fun as the one in the notes

Rule 1. Build new type for app.

Extend $E(L1) = \gamma$, $E(L2) = \beta$,

$E(app) = \gamma * \beta \to \delta$

Rule 2. Analyze fun body of app

```
(if (null? L1) L2
     e1        e2
    (cons (car L1) (app (cdr L1) L2 $
                                    e3
```

t1: Analyze (e1, E) for e1 = (null? L1)
t11 - Analyze (L1) $\simeq$ $E(L1) = \gamma$
s11 - Analyze (null?) $\simeq$ $E(null?) = \{list\ \alpha \to bool\}$
get new type var ξ

(1.) $\gamma \to \xi = list\ \alpha \to bool$
return ξ as type of fun application

t2: Analyze (e2, E) for e2 = L2 (6)

t21: E(L2) = β

t3: Analyze (e3, E) for e3 =

(cons (car L1) (app (cdr L1) L2))

f e1' e2'

t31: Analyze e1' in E, E(car L1)

t311: Analyze E(L1) = γ

S311: Analyze E(car) =

list μ → μ

new type variable α

(2) γ → α = list μ → μ

return α as type of fcn appl

t32: Analyze e2' in E that is

E(app (cdr L1) L2)

t321: Analyze (cdr L1) in E:

t3211: E(L1) = γ

S3211: E(cdr) = list τ → list τ

get new type var Π

(3) γ → Π ≡ list τ → list τ

return type is Π

t322: Analyze E(L2) = β

get a new type variable Δ

(4) $\Gamma * \beta \rightarrow \Delta = \gamma * \beta \rightarrow \delta$

return type is Δ

S32: Analyze E(cons) : $\tau * list\ \tau \rightarrow list\ \tau$

(for this problem we are assuming our lists are homogeneous — although this isn't true in Scheme)

get fresh type variable ρ

(5) $\alpha * \Delta \rightarrow \rho = \tau * list\ \tau \rightarrow list\ \tau$

return type is ρ

Now can type the if-exp level of the recursion —

we have type of e_1, e_2, e_3 of if — $\varphi\ \beta\ \rho$

S1: E(if): $bool * \tau * \tau \rightarrow \tau$

fresh type variable w

(6) $\varphi * \beta * \rho \rightarrow w = bool * \tau * \tau \rightarrow \tau$

(1) $\gamma \to \varphi = list\,\alpha \to bool$

(2) $\gamma \to \alpha = list\,\mu \to \mu$

(3) $\gamma \to \Gamma = list\,\tau \to list\,\tau$

(4) $\Gamma * \beta \to \Delta = \gamma * \beta \to \delta$

(5) $\alpha * \Delta \to \rho = \tau * list\,\tau \to list\,\tau$

(6) $\varphi * \beta * \rho \to w = bool * \tau * \psi \to \tau$

---

(1) $\Rightarrow \gamma = list\,\alpha$ and $\varphi = bool$

then (3) $\Rightarrow \Gamma = list\,\alpha$

(4) $\Rightarrow \delta$ unifies with ρ

(5) $\Rightarrow \Delta = list\,\alpha = \rho$

(5)+(4) $\Rightarrow \delta = list\,\alpha$

(6) $\Rightarrow \beta \subseteq \rho = list\,\alpha$.

$\therefore$ type of app is

$\gamma * \beta \to \delta$ or $list\,\alpha * list\,\alpha \to list\,\alpha$

qed.

(note: $list\,\alpha$ as same type as $list\,\tau$ the type vars are placeholders)