Schedule

SCHEDULE

w	date	topic
~~7~~	~~November 25~~	~~introduction~~
~~8~~	~~December 2~~	~~higher-order functions, lists, trees~~
~~9~~	~~December 9~~	~~graphs, combinatorics~~
~~10~~	~~December 16~~	~~program reasoning~~
~~11~~	~~January 13~~	~~λ and interpreter~~
12	January 20	type system
13	January 27	exam part 2

TOPIC

- exam hints
- type check
- type inference
- advanced topics in type systems

Exam

on Jan 27 written exam: closed book, 50 points

please register online, now (before Jan 25)

- data structures
  lists, trees, graphs
- program analysis
  induction proof, tail-recursion
- interpreters
  evaluator, type system

Type Checking

Type Checking

- types $\tau ::= \alpha \mid \tau \to \tau \mid g(\tau_1, \ldots, \tau_n)$
- typing environments $A ::= \varnothing \mid A, z : \tau$ where $z ::= x \mid c$
- typing judgements $A \vdash e : \tau$
- typing rules

$$\frac{z \in \mathrm{dom}(A)}{A \vdash x : A(z)} \qquad \frac{A, x : \tau_1 \vdash e : \tau_2}{A \vdash (\mathsf{fun}\ x \to e) : \tau_1 \to \tau_2}$$

$$\frac{A \vdash e_1 : \tau_2 \to \tau \quad A \vdash e_2 : \tau_2}{A \vdash e_1 e_2 : \tau} \qquad \frac{A \vdash e_1 : \tau_1 \quad A, x : \tau_1 \vdash e_2 : \tau_2}{A \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

EXERCISES

$A(\mathsf{true}) = \mathsf{bool}$, $A(+) = \mathsf{int} \to \mathsf{int} \to \mathsf{int}$

- $A \vdash (\mathsf{fun}\ x \to x)\ \mathsf{true} : \mathsf{bool}$
- $A \vdash (\mathsf{fun}\ x \to x + x) : \mathsf{int} \to \mathsf{int}$

Type Inference

- typing with unknown variables
- solve by unification

Type Substitutions

type substitution θ is

- function from type variables to types
- $\mathrm{dom}(\theta) = \{\alpha \mid \theta(\alpha) \neq \alpha\}$ is finite

DEFINITIONS

- application to types

$$\alpha\theta = \theta(\alpha)$$
$$(\tau_1 \to \tau_2)\theta = \tau_1\theta \to \tau_2\theta$$
$$g(\tau_1, \ldots, \tau_n)\theta = g(\tau_1\theta, \ldots, \tau_n\theta)$$

- application to type environments

$$(x_1 : \tau_1, \ldots, x_n : \tau_n)\theta = x_1 : \tau_1\theta, \ldots, x_n : \tau_n\theta$$

- composition

$$\theta_1\theta_2 = \{\alpha \mapsto (\alpha\theta_1)\theta_2 \mid \alpha \text{ is type variable}\}$$

Type Inference

PROBLEM

- input: $A \rhd e : \tau$
- output: θ such that $A\theta \vdash e : \tau\theta$

solution θ is principal if every solution is of the form $\theta\theta'$ for some θ'

APPROACH

1. to find τ such that $A \vdash e : \tau$, use fresh variable α to solve $A \rhd e : \alpha$
   $\tau = \alpha\theta$ for its principal solution θ
2. $A \rhd e : \tau \quad \rightsquigarrow^* \quad C$
3. principal solution of $A \rhd e : \tau$ = most general unifier of C

Type Inference

- $\mathrm{ftv}(\tau)$ is set of all free type variables in τ

$$\mathrm{ftv}(A) = \bigcup_{z \in \mathrm{dom}(A)} \mathrm{ftv}(A(z))$$

- simplification of type inference problems

$$\begin{array}{lll}
A \rhd z : \tau & \rightsquigarrow \bot & \text{if } z \notin \mathrm{dom}(A) \\
A \rhd z : \tau & \rightsquigarrow A(z) \doteq \tau & \text{if } z \in \mathrm{dom}(A) \\
A \rhd (\text{fun } x \to e) : \tau & \rightsquigarrow \exists \alpha_1, \alpha_2.(A, x : \alpha_1 \rhd e : \alpha_2 \ \wedge\ \tau \doteq \alpha_1 \to \alpha_2) & \\
A \rhd e_1 e_2 : \tau & \rightsquigarrow \exists \alpha.(A \rhd e_1 : \alpha \to \tau \ \wedge\ A \rhd e_2 : \alpha) & \\
A \rhd \text{let } x = e_1 \text{ in } e_2 : \tau & \rightsquigarrow \exists \alpha.(A \rhd e_1 : \alpha \ \wedge\ A, x : \alpha \rhd e_2 : \tau) &
\end{array}$$

where $\alpha_1, \alpha_2, \alpha \notin \mathrm{ftv}(\tau) \cup \mathrm{ftv}(A)$, i.e., fresh variables

EXERCISES rewrite to unification problems

$A(\text{true}) = \text{bool}$, $A(+) = \text{int} \to \text{int} \to \text{int}$

- $A \rhd (\text{fun } x \to x)\ \text{true} : \alpha$
- $A \rhd (\text{fun } x \to x + x) : \alpha$

Unification Algorithm

$$\{\tau_1 \doteq \tau_1', \ldots, \tau_n \doteq \tau_n'\}\theta = \{\tau_1\theta \doteq \tau_1'\theta, \ldots, \tau_n\theta \doteq \tau_n'\theta\}$$

$$\begin{array}{ll}
U(\theta, \varnothing) & = \theta \\
U(\theta, \{\tau \doteq \tau\} \uplus C) & = U(\theta, C) \\
U(\theta, \{\alpha \doteq \tau\} \uplus C) & = U(\theta\{\alpha \mapsto \tau\}, C\{\alpha \mapsto \tau\}) \\
U(\theta, \{\tau \doteq \alpha\} \uplus C) & = U(\theta\{\alpha \mapsto \tau\}, C\{\alpha \mapsto \tau\}) \\
U(\theta, \{g(\tau_1, \ldots, \tau_n) \doteq g(\tau_1', \ldots, \tau_n')\} \uplus C) & = U(\theta, \{\tau_1 \doteq \tau_1', \ldots, \tau_n \doteq \tau_n'\} \cup C) \\
U(\theta, \{\tau \doteq \tau'\} \uplus C) & = \text{raise } (\text{Unify}(\tau, \tau'))
\end{array}$$

where $\alpha \neq \tau$ and τ does not contain α

Properties

- type system : simple types

THEOREM safety

- (progress) e is typed under initial environment $\implies$ e is value or reducible
- (preservation) reduction preserves typing

let-polymorphism

is not part of exam

Universal Types

EXAMPLE

```
sort : 'a list -> 'a list
```

SYSTEM F

- types $\tau ::= \alpha \mid \tau \rightarrow \tau \mid \forall\alpha.\tau$
- type reconstruction is undecidable

CHALLENGE

find decidable fragment of System F

SOLUTION

let-polymorphism

- restrict polymorphism to top-level let-bindings
- disallow functions that take polymorphic values as arguments

Type Checking

- types $\tau ::= \alpha \mid \tau \to \tau \mid g(\tau_1, \ldots, \tau_n)$
- type schemes $\sigma ::= \forall \alpha_1 \cdots \alpha_n . \tau$ ($\forall .\ \alpha$ is abbreviated to α)
- typing environments $A ::= \varnothing \mid A, z : \sigma$ where $z ::= x \mid c$
- typing judgements $A \vdash e : \tau$
- typing rules

~~$\dfrac{z \in \mathrm{dom}(A)}{A \vdash x : A(z)}$~~ $\qquad \dfrac{A, x : \tau_1 \vdash e : \tau_2}{A \vdash (\mathsf{fun}\ x \to e) : \tau_1 \to \tau_2} \qquad \dfrac{A \vdash e_1 : \tau_2 \to \tau \quad A \vdash e_2 : \tau_2}{A \vdash e_1 e_2 : \tau}$

instantiation $\qquad \dfrac{A(z) = \forall \overline{\alpha}.\ \tau}{A \vdash z : \tau\{\overline{\alpha} \mapsto \overline{\tau'}\}}$

generalization $\qquad \dfrac{A \vdash e_1 : \tau_1 \quad A, x : \forall(\mathrm{ftv}(\tau_1) \setminus \mathrm{ftv}(A)).\tau_1 \vdash e_2 : \tau_2}{A \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$

EXERCISES $\quad A(0) = \mathsf{int}$

- $A \vdash \mathsf{let}\ \mathit{id} = \mathsf{fun}\ x \to x\ \mathsf{in}\ \mathit{id}\ 0 : \mathsf{int}$
- $A \vdash \mathsf{let}\ \mathit{id} = \mathsf{fun}\ x \to x\ \mathsf{in}\ \mathit{id}\ \mathit{id}\ 0 : \mathsf{int}$

Type Inference

simplification of type inference problems

~~$A \rhd z : \tau \quad \rightsquigarrow\ A(z) \doteq \tau \quad$ if $z \in \mathrm{dom}(A)$~~

$A \rhd z : \tau \quad \rightsquigarrow\ \exists \overline{\alpha}.\ \tau \doteq \tau' \quad$ if $A(z) = \forall \overline{\alpha}.\ \tau'$

$A \rhd (\mathsf{fun}\ x \to e) : \tau \quad \rightsquigarrow\ \exists \alpha_1, \alpha_2.(A, x : \alpha_1 \rhd e : \alpha_2\ \wedge\ \tau \doteq \alpha_1 \to \alpha_2)$

$A \rhd e_1 e_2 : \tau \quad \rightsquigarrow\ \exists \alpha.(A \rhd e_1 : \alpha \to \tau\ \wedge\ A \rhd e_2 : \alpha)$

~~$A \rhd \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau \rightsquigarrow \exists \alpha.(A \rhd e_1 : \alpha\ \wedge\ A, x : \alpha \rhd e_2 : \tau)$~~

$A \rhd \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau \rightsquigarrow A, x : \forall \alpha \overline{\beta}.\ \theta(\alpha) \rhd e_2 : \tau$

where

- $\alpha_1, \alpha_2, \alpha \notin \mathrm{ftv}(\tau) \cup \mathrm{ftv}(A)$, $\overline{\alpha} \cap \mathrm{ftv}(\tau) = \varnothing$ (fresh variables)
- $A \rhd e_1 : \alpha \rightsquigarrow \exists \overline{\beta}.C$ and $\theta = U(\varnothing, C)$

EXERCISES $\quad A(0) = \mathsf{int}$

- $A \rhd \mathsf{let}\ \mathit{id} = \mathsf{fun}\ x \to x\ \mathsf{in}\ \mathit{id}\ 0 : \alpha$
- $A \rhd \mathsf{let}\ \mathit{id} = \mathsf{fun}\ x \to x\ \mathsf{in}\ \mathit{id}\ \mathit{id}\ 0 : \alpha$

Advanced Topics in Type Systems

Array, Invariant and Java

$$\text{ArrayJava}\,\frac{\tau_1 <: \tau_2}{\tau_1[\,] <: \tau_2[\,]}$$

EXCERCISE

show that this type system is broken

- in Java we need to check all assignments to any arrays in runtime

Value Restriction

```
# let l = ref [];;
val l : '_a list ref = {contents = []}
# let f = ref (fun x -> x);;
val f : ('_a -> '_a) ref = {contents = <fun>}
```

- monomorphic types

EXERCISE

l and f cannot be polymorphic — why?

History

- restoring type safety of ref was long standing open problem
- value restriction
  let-binding can be generalized only if its rhs is syntactic value

$$\text{let } x = v_1 \text{ in } e_2$$

- many techniques for relaxing restriction were proposed
- Wright (1995) analyzed huge corpus of codes and concluded "almost all codes satisfy value restriction"