CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

Version: 1

April 13, 2020

Plan

- **Announcements**
  - HW8 is due Wednesday April 15th
  - HW9 was posted last Friday and is due Wednesday April 22th
- **Last time**
  - Type inference
  - Solving type inference constraints
- **Today**
  - HW7 review
  - HW8 discussion
  - Moving from type schemes to types (Instantiation)
  - Moving from types to type schemes (Generalization)

HW7: List in Impcore problem

- **Note Make-Array**
  - Takes an expression of the element type to initialize entries

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{INT} \qquad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{MAKE-ARRAY}(e_1, e_2) : \text{ARRAY}(\tau)} \quad (\textsc{MakeArray})$$

- **Empty list introduction rule**
  - Need to indicate what the type of the elements will be
  - Ideas how?

```
'() is enough?
'(int)
Emptylist int
emptyList 42
```

Hints for HW8

- **See piazza post @379 for a lot of code from the book organized in one file**
- **Any questions about that code at this point? Can also post questions about the code to piazza.**
- **You cannot share any of your solve code with anyone but your partner (<30 lines in key)**
- **You can share your constraints that you are putting in "constraints.sml". Do not use anyone else's posted constraints as your constraints. Do use them to test your code.**

What is a substitution?

Formally, θ is a function:

- Replaces a *finite* set of type variables with types
- Apply to type, constraint, type environment, . . .

In code, a data structure:

- "Applied" with `tysubst`, `consubst`
- Made with `idsubst`, `a |--> tau`, `compose`
- Find domain with `dom`

Substitution preserves type structure

Type structure:

```
datatype ty
  = TYVAR  of tyvar
  | TYCON  of name
  | CONAPP of ty * ty list
```

Substitution replaces only type variables:

- Every type constructor is unchanged
- Distributes over type-constructor application

$$\theta(\texttt{TYCON}\ \mu) = \texttt{TYCON}\ \mu$$

$$\theta(\texttt{CONAPP}\ (\tau, [\tau_1, \ldots, \tau_n])) = \texttt{CONAPP}\ (\theta\tau, [\theta_1\tau_1, \ldots \theta_n\tau_n]))$$

We infer stratified "Hindley-Milner" types

Two layers: Monomorphic types τ
Polymorphic type schemes σ

τ	::=	α	type variables
	|	μ	type constructors: `int`, `list`
	|	$(\tau_1, \ldots \tau_n)\ \tau$	constructor application
σ	::=	$\forall \alpha_1, \ldots \alpha_n\ .\ \tau$	type scheme

Each variable in Γ introduced via `LET`, `LETREC`, `VAL`, and `VAL-REC` has a type scheme σ with $\forall$.

Each variable in Γ introduced via `LAMBDA` has a degenerate type scheme $\forall\ .\ \tau$—a type, wrapped.

Type inference Algorithm

Given Γ and e, compute C and τ such that

$$C, \Gamma \vdash e : \tau$$

Extend to list of e_i: $C, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{IF}(e_1, e_2, e_3) : \tau} \quad \textsf{(IF)}$$

becomes (note equality constraints with $\sim$)

$$\frac{C, \Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3}{C \wedge \tau_1 \sim \texttt{bool} \wedge \tau_2 \sim \tau_3, \Gamma \vdash \textsf{IF}(e_1, e_2, e_3) : \tau_3} \quad \textsf{(IF)}$$

Key ideas

Type environment Γ binds var to type scheme σ

- `app2` : $\forall \alpha, \beta \,.\, (\alpha \to \beta) \times \alpha \times \alpha \to \beta$
- `cc` : $\forall \alpha \,.\, \alpha$ `list list` $\to \alpha$
- `car` : $\forall \alpha \,.\, \alpha$ `list` $\to \alpha$
- `n` : $\forall \,.$ `int` (note empty $\forall$)

Judgment $\Gamma \vdash e : \tau$ gives expression e a type τ

(Transitions happen automatically!)

Key ideas

Definitions are polymorphic with type schemes

Each use is monomorphic with a (mono-) type

Transitions:

- **At use, type scheme instantiated automatically**
- **At definition, automatically abstract over tyvars**

Question: Constraints are in terms of type schemes or types?

```
datatype con = ~ of ty * ty
             | /\ of con * con
             | TRIVIAL

datatype ty = TYVAR of tyvar | TYCON of tycon
            | CONAPP of ty * ty list

datatype type_scheme = FORALL of tyvar list * ty
```

Moving between type scheme and type

From σ to τ: instantiate

From τ to σ: generalize

$$
\begin{array}{lcl}
\tau & ::= & \alpha \\
 & | & \mu \\
 & | & (\tau_1, \ldots \tau_n)\tau \\
\sigma & ::= & \forall \alpha_1, \ldots \alpha_n . \tau
\end{array}
$$

From Type Scheme to Type

VAR rule instantiates type schema with fresh and distinct type variables:

$$\frac{\Gamma(x) = \forall \alpha_1, \ldots \alpha_n . \tau \qquad \alpha'_1, \ldots \alpha'_n \text{ are fresh and distinct}}{T, \Gamma \vdash x : ((\alpha_1 \mapsto \alpha'_1) \circ \ldots \circ (\alpha_n \mapsto \alpha'_n))\tau} \quad \text{(VAR)}$$

Why the freshness requirement?

- **Consider**

```
Gamma = {fst : forall 'a 'b. 'a * 'b -> 'a, y : 'ay}

`a1 ~ int /\ `ay1 ~ bool, Gamma |- if (y, fst 2 3, 4): int
```

- **Imagine we ignore the freshness constraint when instantiating fst type**

```
fst : `ay * `b -> `ay
```

- **From `if` we get the following unsatisfiable constraints**

```
`ay ~ bool
`ay ~ int
```

Why the distinctness requirement?

- **Consider**

```
Gamma = {fst : forall 'a 'b. 'a * 'b -> 'a, y : 'ay}

`a1 ~ int /\ `b1 ~ bool, Gamma |- fst 2 #t : int
```

- **Imagine we ignore the distinctness constraint when instantiating fst type**

```
fst : `a1 * `a1 -> `a1
```

- **From `apply` rule, we get the following unsatisfiable constraints**

```
`a1 ~ int
`a1 ~ bool
```

Generalization: From Type to Type Scheme

Goal is to get `forall`:

```
-> (val fst (lambda (x y) x))
fst : (forall ('a 'b) ('a 'b -> 'a))
```

First derive:

$$T, \emptyset \vdash \texttt{(lambda (x y) x)} : \alpha \times \beta \to \alpha$$

Abstract over α, β and add to environment:

$$\texttt{fst} : \forall \alpha, \beta \,.\, \alpha \times \beta \to \alpha$$

Generalize Function

Useful tool for finding quantified type variables:

$$\texttt{generalize}(\tau, A) = \forall \alpha_1, \ldots, \alpha_n . \tau$$

where

$$\{\alpha_1, \ldots \alpha_n\} = \text{ftv}(\tau) - A$$

Examples:

$$\texttt{generalize}(\alpha \times \beta \to \alpha, \emptyset) \quad = \forall \alpha, \beta . \alpha \times \beta \to \alpha$$

$$\texttt{generalize}(\alpha \times \beta \to \alpha, \{\alpha\}) = \forall \beta . \alpha \times \beta \to \alpha$$

- The set A above will be useful when some variables in tau are mentioned in the environment. Can't generalize over them.
- Example `fst: generalize('a * 'b -> 'a, emptyset) = forall 'a, 'b. 'a * 'b -> 'a`

First candidate VAL rule (no constraints)

Empty environment:

$$\frac{T, \emptyset \vdash e : \tau \qquad \sigma = \texttt{generalize}(\tau, \emptyset)}{\langle \textsf{VAL}(x, e), \emptyset \rangle \rightarrow \{x \mapsto \sigma\}} \quad \textsc{(Val with } T\textsc{)}$$

But we need to handle nontrivial constraints

Example with nontrivial constraints

```
(val pick (lambda (x y z) (if x y z)))
```

During inference, we derive the judgment:

$$\alpha_x \sim \texttt{bool} \wedge \alpha_y \sim \alpha_z, \emptyset \vdash$$
$$\texttt{(lambda (x y z) (if x y z))} : \alpha_x \times \alpha_y \times \alpha_z \to \alpha_z$$

Before generalization, solve the constraint:

$$\theta = \{\alpha_x \mapsto \texttt{bool}, \alpha_y \mapsto \alpha_z\}$$

So the type we need to generalize is

$$\theta(\alpha_x \times \alpha_y \times \alpha_z \to \alpha_z) = \texttt{bool} \times \alpha_z \times \alpha_z \to \alpha_z$$

And $\texttt{generalize}(\texttt{bool} \times \alpha_z \times \alpha_z \to \alpha_z, \emptyset)$ **is**

$$\forall \alpha_z . \texttt{bool} \times \alpha_z \times \alpha_z \to \alpha_z$$

2nd candidate VAL rule (no context)

$$C, \emptyset \vdash e : \tau$$

θC **is satisfied**

$$\frac{\sigma = \texttt{generalize}(\theta\tau, \emptyset)}{\langle \textsf{VAL}(x, e), \emptyset \rangle \rightarrow \{x \mapsto \sigma\}}$$

(VAL 2)

But we need to handle nonempty contexts

VAL rule — the full version

$$C, \Gamma \vdash e : \tau$$

$$\frac{\theta C \text{ is satisfied} \qquad \theta\Gamma = \Gamma \qquad \sigma = \texttt{generalize}(\theta\tau, \mathrm{ftv}(\Gamma))}{\langle \textsf{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}} \quad \textsc{(Val)}$$

Let Examples

- **Questions**
  - What are types for cons and pair? cons : forall ‘a. ’a * ‘a list -> ‘a list, pair : forall ‘a, ‘b. ‘a * ‘b -> ’a * ‘b
  - Can the type for ys be of the form forall ’a . ‘a list? No. If not why? 7
  - For each example below, what are the types for s and extend?
  - Which of the below will correctly type check?

```
(lambda (ys)
   (let ([s (lambda (x) (cons x '()))])
      (pair (s 1) (s #t))))
```

```
(lambda (ys)
   (let ([extend (lambda (x) (cons x ys))])
      (pair (extend 1) (extend #t))))
```

```
(lambda (ys)
   (let ([extend (lambda (x) (cons x ys))])
      (extend 1)))
```

Let Examples

- **Question: What are the type constraints for the below?**

```
(lambda (ys)
   (let ([s (lambda (x) (cons x '()))])
      (pair (s 1) (s #t))))
```

```
???
```

```
(lambda (ys)
   (let ([extend (lambda (x) (cons x ys))])
      (pair (extend 1) (extend #t))))
```

```
???
```

Let

$$\frac{\begin{array}{c} C,\Gamma \vdash e_1,\ldots,e_n : \tau_1,\ldots,\tau_n \\ \theta C \text{ is satisfied} \qquad \theta \text{ is idempotent} \\ C' = \bigwedge\{\alpha \sim \theta\alpha \mid \alpha \in (dom\theta \cap \mathrm{ftv}(\Gamma))\} \\ \sigma_i = \mathtt{generalize}(\theta\tau_i, \mathrm{ftv}(\Gamma) \cup \mathrm{ftv}(C')), \quad 1 \le i \le n \\ C_b, \Gamma\{x_1 \mapsto \sigma_1,\ldots,x_n \mapsto \sigma_n\} \vdash e : \tau \end{array}}{C' \wedge C_b, \Gamma \vdash \mathsf{LET}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e) : \tau} \quad (\textsc{Let})$$

- If it's not mentioned in the context, it can be anything: independent
- If it is mentioned in the context, don't mess with it: dependent

Let with constraints

- **Operationally**
  - 1. typesof: returns tau1, …, taun and C
  - 2. val theta = solve C
  - 3. C-prime from map, conjoinConstraints, dom, inter, freetyvarsGamma
  - 4. freetyvarsGamma, union, freetyvarsConstraint
  - 5. Map anonymous lambda using generalize, get all the sigma_i
  - 6. Extend the typing environment Gamma (pairfoldr)
  - 7. Recursive call to type checker, gets C_b, \tau
  - 8. Return (tau, C' /\ C_b)

Idempotence

$$\theta \circ \theta = \theta$$

Implies: Applying once is good enough.

Good	Bad
$\alpha \mapsto \texttt{int}$	$\alpha \mapsto \alpha\ \texttt{list}$
$\alpha \mapsto \beta$	$\alpha \mapsto \beta, \beta \mapsto \gamma$
$\alpha_1 \mapsto \beta_1, \alpha_2 \mapsto \beta_2$	

Implies: If $\alpha \mapsto \tau \in \theta$, then $\theta\alpha = \theta\tau$.

VAL-REC rule

$$\frac{C, \Gamma\{x \mapsto \alpha\} \vdash e : \tau \qquad \alpha \text{ is fresh} \qquad \theta(C \wedge \alpha \sim \tau) \text{ is satisfied} \qquad \theta\Gamma = \Gamma \qquad \sigma = \texttt{generalize}(\theta\alpha, \mathrm{ftv}(\Gamma))}{\langle \textsc{val-rec}(x, e), \Gamma \rangle \to \Gamma\{x \mapsto \sigma\}} \quad (\textsc{ValRec})$$

LetRec

$$\Gamma_e = \Gamma\{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\}, \quad \alpha_i \text{ distinct and fresh}$$

$$C_e, \Gamma_e \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n$$

$$C = C_e \wedge \tau_1 \sim \alpha_1 \wedge \ldots \wedge \tau_n \sim \alpha_n$$

$$\theta C \text{ is satisfied} \qquad \theta \text{ is idempotent}$$

$$C' = \bigwedge\{\alpha \sim \theta\alpha \mid \alpha \in dom\theta \cap \mathrm{ftv}(\Gamma)\}$$

$$\sigma_i = \mathtt{generalize}(\theta\tau_i, \mathrm{ftv}(\Gamma) \cup \mathrm{ftv}(C')), \quad 1 \le i \le n$$

$$C_b, \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau$$

$$C' \wedge C_b, \Gamma \vdash \mathsf{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau$$

(LETREC)

Managing Quantified Types

	val and val-rec	let, letrec, ...	lambda
forall	FORALL contains all variables (because none are free in the context)	FORALL contains variables not free in the context	FORALL is empty
Generalization	Generalize over all variables (because none are free in the context)	Generalize over variables not free in the context	Never generalize