CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

The University of Arizona

Version: 1

April 8, 2020

Plan

THE UNIVERSITY OF ARIZONA
Computer Science

- **Announcements**
  - HW7 is due tonight
  - HW8 was posted last Friday and is due Wednesday April 15th
- **Last time**
  - Polymorphic type systems (typed uScheme)
  - Generic type representations
  - Kinds for classifying types
  - Why we want to do type inference
- **Today**
  - Type inference
  - Solving type inference constraints

Key Ideas for Type Inference

- **Fresh type variables represent unknown types**
- **Example:**

```
(lambda (x) (+ x 3))
```

  - Assign variable x a fresh type variable alpha
  - Constraints record knowledges about type variables (e.g. alpha ~ int)

Type inference example

```
(val-rec twotimes (lambda (x) (+ x x))
```

- **What do we know?**
  - Assume `twotimes` has type ‘a1
  - Assume `x` has type ‘a2
  - + has type int * int -> int
  - (+ x x) is an application, what does it require?
    - ‘a2 ~ int /\ ‘a2 ~ int
  - Are those constraints possible to satisfy?
- **Key idea: Record constraint in a typing judgement**

```
`a2 ~ int /\ `a2 ~ int, {twotimes : `a1, x : `a2} |- (+ x x) : int
```

- **General form of typing judgement**

```
C, Gamma                        |- e          : tau
```

Inferring polymorphic types


```
(val app2 = (lambda (f x y)
                 (begin
                     (f x)
                     (f y))))
```

- Assume
  - `app2 : 'a_a`
  - `f : 'a_f,` `x : 'a_x,` `y : 'a_y`
- (f x) implies `'a_f ~ 'a_x -> 'a1`
- (f y) implies `'a_f ~ 'a_y -> 'a2`
- Together these imply `'a_x ~ 'a_y and 'a1 ~ 'a2`
- Begin implies type of function result is `'a2`
- So, app2 : `('a_x -> 'a1) * 'a_x * 'a_y -> 'a2`
- We can generalize to: `forall 'a_x, 'a1. ('a_x -> 'a1) * 'a_x * 'a_x -> 'a1`
- Which is same as `forall 'a, 'b . ('a -> 'b) * 'a * 'a -> 'b`

Exercise:

```
(val cc (lambda (nss) (car (car (nss)))
```

- **Assume**
  - `nss : 'b`
- **We know** `car : forall 'a. 'a list -> 'a`
- **implies car_1 :** `` `a1 list -> 'a1``
- **implies car_2 :** `` `a2 list -> 'a2``
- **(car_1 nss) implies** `'b ~ 'a3 /\ 'a3 ~ 'a1 list`
- **(car_2 (car_1 nss)) implies** `'a1  ~  'a2 list`
- **(car_2 (car_1 nss)) :** `'a2`
- **nss :** `'b ~ 'a3 ~ 'a1 list ~ 'a2 list list`
- **So, cc:** `'a2 list list -> 'a2`
- **Which generalizes to**

```
forall  'a . 'a list list -> 'a
```

Formalizing Type Inference

- Sad news: Type inference for polymorphism is undecidable
- Solution: Each formal parameter has a monomorphic type
- Consequences
  - Polymorphic functions are not first class
  - The argument to a higher-order function cannot be polymorphic
  - Forall appears only outermost in types

We infer stratified "Hindley-Milner" types

Two layers: Monomorphic types τ

Polymorphic type schemes σ

$$
\begin{array}{lcll}
\tau & ::= & \alpha & \text{type variables} \\
 & | & \mu & \text{type constructors: } \texttt{int}, \texttt{list} \\
 & | & (\tau_1, \ldots \tau_n)\,\tau & \text{constructor application} \\
\sigma & ::= & \forall \alpha_1, \ldots \alpha_n \,.\, \tau & \text{type scheme}
\end{array}
$$

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.

Representing Hindley-Milner types

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

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

Two layers: Monomorphic types τ

Polymorphic type schemes σ

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

Key ideas

Type environment Γ binds var to type scheme σ

- `app2` : $\forall\alpha,\beta\,.\,(\alpha \rightarrow \beta) \times \alpha \times \alpha \rightarrow \beta$
- `cc` : $\forall\alpha\,.\,\alpha$ `list list` $\rightarrow \alpha$
- `car` : $\forall\alpha\,.\,\alpha$ `list` $\rightarrow \alpha$
- `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

Type inference Algorithm

Given Γ and e, compute C and τ such that

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

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

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

becomes (note equality constraints with $\sim$)

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

Apply rule

$$\frac{\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \mathsf{APPLY}(e, e_1, \ldots, e_n) : \tau} \ (\textsc{Apply})$$

becomes

$$\frac{C, \Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \qquad \alpha \text{ is fresh}}{C \wedge \hat{\tau} \sim \tau_1 \times \cdots \times \tau_n \to \alpha, \Gamma \vdash \mathsf{APPLY}(e, e_1, \ldots, e_n) : \alpha} \ (\textsc{Apply})$$

Exercise: Begin Rule

$$\frac{\Gamma \vdash e_i : \tau_i \quad 1 \le i \le n}{\Gamma \vdash \mathsf{BEGIN}(e_1, \ldots, e_n) : \tau_n} \quad (\textsc{Begin})$$

$$\frac{C, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n}{C, \Gamma \vdash \mathsf{BEGIN}(e_1, \ldots, e_n) : \tau_n} \quad (\textsc{Begin})$$

Your skills so far

- **You can complete typeof**
  - Takes e and Gamma, returns tau and C
  - (Except for let forms)
- **Next up: solving constraints!**

Representing Constraints

The University of Arizona
Computer Science

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

infix 4 ~
infix 3 /\
```

- **What does a solution to a set of constraints look like?**
- **A substitution/mapping from type variables to types: Theta**

Solving Constraints

We *solve* a constraint C by finding a substitution θ such that the *constraint* θC *is satisfied*.

A substitution θ that makes all contraints in C true is a solution for C.

Examples

- **Which have solutions?**

```
1.   int ~ bool, No
2.   int list ~ bool list, No
3.   'a ~ int, Theta {‘a |-> int }
4.   'a ~ int list, Theta {‘a |-> int list}
5.   'a ~ int -> int, Theta {‘a |-> int -> int
6.   'a ~ 'a, Theta {‘a |- ‘a }
7.   'a * int ~ bool * 'b, Theta { ‘a |-> bool, ‘b |-
> int }
8.   'a * int ~ bool -> 'b, No
9.   'a ~ ('a, int), No
10. 'a ~ tau (arbitrary tau), tau can’t contain ’a
unless it is ‘a
```

Substitutions over constraints

Substitutions distribute over constraints:

$$
\begin{aligned}
\theta(\tau_1 \sim \tau_2) &= \theta\tau_1 \sim \theta\tau_2 \\
\theta(C_1 \wedge C_2) &= \theta C_1 \wedge \theta C_2 \\
\theta T &= T
\end{aligned}
$$

When is a constraint satisfied?

$$\frac{\tau_1 = \tau_2}{\tau_1 \sim \tau_2 \text{ is satisfied}} \quad \textbf{(EQ)}$$

$$\frac{C_1 \text{ is satisfied} \qquad C_2 \text{ is satisfied}}{C_1 \wedge C_2 \text{ is satisfied}} \quad \textbf{(AND)}$$

$$\frac{}{T \text{ is satisfied}} \quad \textbf{(TRIVIAL)}$$

Solving Constraint Conjunctions

Useless rule:

$$\frac{\theta_1 C_1 \text{ is satisfied} \qquad \tilde{\theta}_2 C_2 \text{ is satisfied}}{(\tilde{\theta}_2 \circ \theta_1) C_1 \wedge C_2 \text{ is or is not satisfied}} \quad (\textsc{UnsolvedConjunction})$$

Useful rule:

$$\frac{\theta_1 C_1 \text{ is satisfied} \qquad \theta_2(\theta_1 C_2) \text{ is satisfied}}{(\theta_2 \circ \theta_1) C_1 \wedge C_2 \text{ is satisfied}} \quad (\textsc{SolvedConjunction})$$

Food for thought (or recitation): Find examples to illustrate that UNSOLVEDCONJUNCTION is bogus.

What you can do after this lecture

THE UNIVERSITY OF ARIZONA
Computer Science

- **After this lecture, you can write ”solve”, a function which given a constraint C has one of three outcomes:**
  - Returns the identity substitution in the case where C is trivially satisfiable
  - Returns a non-trivial substitution if C is satisfiable otherwise
  - Calls unsatisfiableEquality in when C cannot be satisified
- **You could also write a type inference ty for everything except let forms**