

Type Inference:

How does your compiler know the type of an expression?

Stefan Schulze Frielinghaus

stefan@seekline.net

Motivation

C/C++

```
int main(void) {  
    int a = 42;  
    double b = 23.0;  
    return a * (int) b;  
}
```

OCaml

```
let a = 42 in  
    let b = 23.0 in  
        a * int_of_float(b);;
```

Motivation cont'd

How would you (as a human) infer the type of the function `fac`?

```
let rec fac = function
| 0 -> 1
| n -> fac (n-1) * n;;
```

Motivation cont'd

How would you (as a human) infer the type of the function `fac`?

```
let rec fac = function
| 0 -> 1
| n -> fac (n-1) * n;;
```

- ▶ expression `>>1<<` is obviously an integer \implies return type integer

Motivation cont'd

How would you (as a human) infer the type of the function `fac`?

```
let rec fac = function
| 0 -> 1
| n -> fac (n-1) * n;;
```

- ▶ expression `»1«` is obviously an integer \implies return type integer
- ▶ multiplication is only allowed with same types for both operands, i.e. `integer * n` \implies `n` must be an integer

Motivation cont'd

How would you (as a human) infer the type of the function `fac`?

```
let rec fac = function
| 0 -> 1
| n -> fac (n-1) * n;;
```

- ▶ expression `»1«` is obviously an integer \implies return type integer
- ▶ multiplication is only allowed with same types for both operands, i.e. `integer * n` \implies `n` must be an integer
- ▶ `fac : int -> int`

So far ...

- ▶ this works only for type-safe languages
 - ▶ goodbye C, C++, ...
 - ▶ (C++0x Type Inference is nothing else then some type deduction)
 - ▶ welcome (pure) functional programming languages
- ▶ we will use the Hindley-Milner algorithm to infer types [1, 5]
- ▶ and stick to a simple language (lambda-calculus with let-polymorphism):

$$e ::= c \mid x \mid e_1 e_2 \mid \lambda x. e \mid \text{let } x = e_1 \text{ in } e_2$$

- ▶ with the following types:

$$\tau ::= \alpha \mid \text{Int} \mid \text{Bool} \mid \text{String} \mid \tau_1 \rightarrow \tau_2$$

- ▶ and type schemes:

$$\sigma ::= \forall \alpha_1, \dots, \alpha_k. \tau \quad k \geq 0$$

Background: Inference Rules

$$\frac{A \quad B}{C}$$

Read the rule as: If A and B hold, then also C.
Or: In order to prove C, prove A and B.

and more general:

$$\frac{\mathcal{A}_1 \vdash F_1 \quad \dots \quad \mathcal{A}_n \vdash F_n}{\mathcal{B} \vdash F} \quad n \geq 0$$

Background: Inference Rules

$$\frac{A \quad B}{C}$$

Read the rule as: If A and B hold, then also C.
Or: In order to prove C, prove A and B.

and more general:

$$\frac{\mathcal{A}_1 \vdash F_1 \quad \dots \quad \mathcal{A}_n \vdash F_n}{\mathcal{B} \vdash F} \quad n \geq 0$$

Example:

$$\frac{}{\text{even}(0)} \quad \frac{}{\text{odd}(1)} \quad \text{axioms}$$

$$\frac{\text{even}(n-1)}{\text{odd}(n)} \quad \frac{\text{odd}(n-1)}{\text{even}(n)} \quad \text{inference}$$

Preliminaries



$$\Gamma \vdash e : t$$

Should be read as: Expression e has type t under assumptions Γ .

- ▶ Bounded and Free Variables:

$$\lambda x. xy$$

x is bounded by λ whereas y is a **free variable**

- ▶ Substitution:

$$s[t/x]$$

substitution of t for x in s , i.e. $s[t/x] = t$

Preliminaries cont'd

- ▶ Why bother with free and bounded variables?

$$\lambda n.n - m \equiv_{\alpha} \lambda v.v - m$$

- ▶ substitution of a free variable might change the semantics

$$\underbrace{(\lambda x.y)}_{\text{const fct.}}[x/y] = \underbrace{\lambda x.x}_{\text{identity}} \quad \not\equiv$$

- ▶ \implies we have to check for free variables

Inference Rules

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

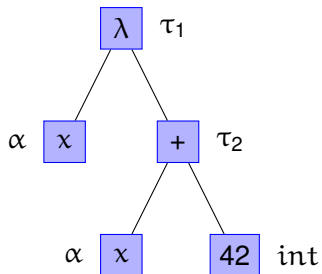
$$\text{APP: } \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1 e_2) : t_2}$$

$$\text{ABS: } \frac{\Gamma[x : t_1] \vdash e : t_2}{\Gamma \vdash \lambda x. e : t_1 \rightarrow t_2}$$

$$\text{LET: } \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma[x : t_1] \vdash e_2 : t_2}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : t_2}$$

Assign type-variables to each expression

Example: $\lambda x.x + 42$



Set-up constraint system and solve it

$$\tau_1 = \alpha \rightarrow \tau_2$$

$$\tau_2 = \text{int}$$

$$\alpha = \text{int}$$

we conclude (find the most general unifier):

$$\tau_1 = \text{int} \rightarrow \text{int}$$

Formalization of the last steps

- ▶ assign a fresh type-variable to each expression e and variable x : $\tau[e]$ and $\alpha[x]$
- ▶ generate constraints according to the previous inference rules:

$$\text{Var: } e \equiv x \quad \Longrightarrow \quad \tau[e] = \alpha[x]$$

$$\text{App: } e \equiv e_1 e_2 \quad \Longrightarrow \quad \tau[e_1] = \tau[e_2] \rightarrow \tau[e]$$

$$\text{Abs: } e \equiv \lambda x. e_1 \quad \Longrightarrow \quad \tau[e] = \alpha[x] \rightarrow \tau[e_1]$$

$$\text{Let: } e \equiv \text{let } x = e_1 \text{ in } e_2 \quad \Longrightarrow \quad \tau[e_1] = \alpha[e_1] \text{ and } \tau[e] = \tau[e_2]$$

Extending our inference rules

$$\Gamma \vdash e : t \mid \mathcal{C}$$

$\mathcal{C} :=$ constraints

$$\text{VAR: } \frac{\Gamma(x) = \forall \alpha_1, \dots, \alpha_k. \tau \quad \beta_i \text{ fresh}}{\Gamma \vdash x : \tau[\beta_1/\alpha_1, \dots, \beta_k/\alpha_k] \mid \emptyset}$$

$$\text{APP: } \frac{\Gamma \vdash e_1 : \tau_1 \mid \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \mid \mathcal{C}_2 \quad \alpha \text{ fresh}}{\Gamma \vdash (e_1 e_2) : \alpha \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 = \tau_2 \rightarrow \alpha\}}$$

Extending our inference rules cont'd

$$\text{ABS: } \frac{\Gamma \oplus [x : \alpha] \vdash e : \tau \mid \mathcal{C} \quad \alpha \text{ fresh}}{\Gamma \vdash \lambda x. e : \alpha \rightarrow \tau \mid \mathcal{C}}$$

$$\text{LET: } \frac{\begin{array}{c} \alpha = \text{gen}(\Gamma, \tau_1, \mathcal{C}_1) \\ \Gamma \vdash e_1 : \tau_1 \mid \mathcal{C}_1 \quad \Gamma \oplus [x : \alpha] \vdash e_2 : \tau_2 \mid \mathcal{C}_2 \end{array}}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau_2 \mid \mathcal{C}_1 \cup \mathcal{C}_2}$$

Lets have a look at an example

Expression: $\lambda f.\lambda x.f x$

$$\begin{array}{c} \text{VAR} \frac{}{[f : \tau_1, x : \tau_3] \vdash f : \tau_5} \quad \text{VAR} \frac{}{[f : \tau_1, x : \tau_3] \vdash x : \tau_6} \\ \text{APP} \frac{}{[f : \tau_1, x : \tau_3] \vdash (f x) : \tau_4 \mid \{\tau_5 = \tau_6 \rightarrow \tau_4\}} \\ \text{ABS} \frac{}{[f : \tau_1] \vdash (\lambda x \rightarrow f x) : \tau_2 \equiv \tau_3 \rightarrow \tau_4} \\ \hline [] \vdash (\lambda f.\lambda x.f x) : \tau_0 \equiv \tau_1 \rightarrow \tau_2 \end{array}$$

This leads to the following constraints:

$$\tau_0 \equiv [\tau_1 \rightarrow \tau_2] \quad \tau_2 \equiv [\tau_3 \rightarrow \tau_4] \quad \tau_5 \equiv [\tau_6 \rightarrow \tau_4] \quad \tau_5 \equiv \tau_1 \quad \tau_6 \equiv \tau_3$$

Solve the constraint system by substitution

This leads to the following constraints:

$$\tau_0 \equiv [\tau_1 \rightarrow \tau_2] \quad \tau_2 \equiv [\tau_3 \rightarrow \tau_4] \quad \tau_5 \equiv [\tau_6 \rightarrow \tau_4] \quad \tau_5 \equiv \tau_1 \quad \tau_6 \equiv \tau_3$$

- ▶ Eliminate τ_6 :

$$\tau_0 \equiv [\tau_1 \rightarrow \tau_2] \quad \tau_2 \equiv [\tau_3 \rightarrow \tau_4] \quad \tau_5 \equiv [\tau_3 \rightarrow \tau_4] \quad \tau_5 \equiv \tau_1$$

- ▶ Eliminate τ_5 :

$$\tau_0 \equiv [\tau_1 \rightarrow \tau_2] \quad \tau_2 \equiv [\tau_3 \rightarrow \tau_4] \quad \tau_1 \equiv [\tau_3 \rightarrow \tau_4]$$

- ▶ Eliminate τ_2 :

$$\tau_0 \equiv [\tau_1 \rightarrow [\tau_3 \rightarrow \tau_4]] \quad \tau_1 \equiv [\tau_3 \rightarrow \tau_4]$$

- ▶ Eliminate τ_1 :

$$\tau_0 \equiv [[\tau_3 \rightarrow \tau_4] \rightarrow [\tau_3 \rightarrow \tau_4]]$$

Finally we have our type: $[[\alpha \rightarrow \beta] \rightarrow [\alpha \rightarrow \beta]]$

Unification

OK, this was nice. But how do you find the most general type automatically?

```
unify( $\mathcal{C}$ ) := if  $\mathcal{C} = \emptyset$ , then []  
                else let  $\mathcal{C} = \mathcal{C}' \cup \{\tau \doteq \tau'\}$  in  
                    if  $\tau = \tau'$ , then unify( $\mathcal{C}'$ )  
                    else if  $\tau = \alpha$  and  $\alpha \notin \text{FV}(\tau')$ , then // occurs chk  
                        unify( $\mathcal{C}'[\tau'/\alpha]$ )  $\oplus$   $[\tau'/\alpha]$   
                    else if  $\tau' = \alpha$  and  $\alpha \notin \text{FV}(\tau)$ , then // occurs chk  
                        unify( $\mathcal{C}'[\tau/\alpha]$ )  $\oplus$   $[\tau/\alpha]$   
                    else if  $\tau = \tau_1 \rightarrow \tau_2$  and  $\tau' = \tau'_1 \rightarrow \tau'_2$ , then  
                        unify( $\mathcal{C}' \cup \{\tau_1 = \tau'_1, \tau_2 = \tau'_2\}$ )  
                    else fail
```

Unification cont'd

- ▶ $\text{unify}(\mathcal{C})$ returns the most general unifier Θ
- ▶ we still have not talked about the $\text{gen}(\Gamma, \tau, \mathcal{C})$ function:

$$\begin{aligned}\text{gen}(\Gamma, \tau, \mathcal{C}) &:= \Theta = \text{unify}(\mathcal{C}) \\ &\quad \{\alpha_1, \dots, \alpha_k\} = \text{FV}(\tau\Theta) \setminus \text{FV}(\Gamma) \\ &\quad \forall \alpha_1, \dots, \alpha_k. \tau\Theta\end{aligned}$$

- ▶ the function makes all type-variables in τ generic, that are not available in Γ
- ▶ \implies classic \mathcal{W} algorithm
- ▶ which has exponential complexity for $\gg\text{let}\ll$ expressions
- ▶ in all other cases we have linear complexity

Complexity

Example¹:

```
let f0 = fun x -> (x,x) in
  let f1 = fun y -> f0 (f0 y) in
    let f2 = fun y -> f1 (f1 y) in
      let f3 = fun y -> f2 (f2 y) in
        let f4 = fun y -> f3 (f3 y) in
          let f5 = fun y -> f4 (f4 y) in
            f5 (fun z -> z);;
```

- ▶ on an Intel Core i5-560M (2.66 – 3.2 GHz) OCaml needs about 1.5 hours to find the type
- ▶ ML is DEXPTIME-complete, i.e. $\text{DTIME}\left(2^{n^{O(1)}}\right)$, see [4, 2, 3] for more

¹see [6, p. 334]

Complexity cont'd

- ▶ Can we do better?
- ▶ Seeing it as a graph theoretical problem we could identify isomorphic sub-graphs
- ▶ ...

Remark on let-polymorphism

- ▶ Operationally $\gg\text{let}\ll$ and $\gg\lambda\ll$ do the same but from a type-systems point of view they differ:

```
let f =  $\lambda x.x$  in (f 42, f 23.0)
```

compare against

```
 $\lambda f.(f\ 42, f\ 23.0)$ 
```

- ▶ let is polymorph while λ is only monomorph
- ▶ for the same expression using only λ we would get a type clash

Outline

- ▶ today, plenty of type inference algorithms exist:
 - ▶ Liquid Types
 - ▶ Intersection Types
 - ▶ Linear Types
 - ▶ Phantom Types
 - ▶ ...

Questions?

Appendix: Free Variables

$$\text{FV}(c) = \emptyset$$

$$\text{FV}(x) = \{x\}$$

$$\text{FV}(e_1 e_2) = \text{FV}(e_1) \cup \text{FV}(e_2)$$

$$\text{FV}(\lambda x. e) = \text{FV}(e) \setminus \{x\}$$

Appendix: Substitution

$$x[t/x] = t$$

$$a[t/x] = a$$

if $a \neq x$

$$(s_1 s_2)[t/x] = (s_1[t/x])([s_2 t/x])$$

$$(\lambda y. s)[t/x] = \lambda y. (s[t/x])$$

if $y \neq x$ and $y \notin \text{FV}(s)$

References I



Roger Hindley.

The Principal Type-Scheme of an Object in Combinatory Logic.
Transactions of the American Mathematical Society, 146:29–60,
December 1969.



A. J. Kfoury, J. Tiuryn, and P. Urzyczyn.

ML typability is dextime-complete.

In *Proceedings of the fifteenth colloquium on CAAP'90*, pages
206–220, New York, NY, USA, 1990. Springer-Verlag New York, Inc.



A. J. Kfoury, J. Tiuryn, and P. Urzyczyn.

An analysis of ml typability.

J. ACM, 41:368–398, March 1994.

References II



Harry G. Mairson.

Deciding ml typability is complete for deterministic exponential time.
In Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '90, pages 382–401, New York, NY, USA, 1990. ACM.



Robin Milner.

A Theory of Type Polymorphism in Programming.
Journal of Computer and System Sciences, 17(3):348–375, December 1978.



Benjamin C. Pierce.

Types and Programming Languages.
MIT Press, 2002.