Hindley-Milner Type Inference

presented by Josh Goldman

HoPL Spring 2021

Areas of Interest

1) One department, two papers
2) Algorithm W? Prove it!
3) Polynomial, how about exponential
4) I can't decide if I should use this

1 One Department, two Papers

1969: Hindley

"The Principal type-scheme of an Object in Combinatory Logic"

1978: Milner

"A theory of type Polymorphism in Programming"

Many Questions

- Can we check types once and be done?

- Can we avoid explicitly mentioning types all together?

- Do polymorphic types (polytypes) need to be specified?

- Maybe

- maybe

- maybe

This concludes the lecture.

That was Easy!

Lets take a closer look!

Source: Milner '78

let $x = e$ in e'

let $f(x_1, \ldots, x_n) = e$ in e'

Ex 1: map

```
letrec map(f, m) =
    if null(m) then nil
    else cons(f(hd(m)),
              map(f, tl(m)))
```

generic types: α, β

map : $((\alpha \to \beta) \times \alpha\ list) \to \beta\ list$

null : $\alpha\ list \to bool$

nil : $\alpha\ list$

hd : $\alpha\ list \to \alpha$

tl : $\alpha\ list \to \alpha\ list$

cons : $(\alpha \times \alpha\ list) \to \alpha\ list$

$$\sigma_{nil} = T_1\ list \to bool$$

$$\sigma_{n,l} = T_2\ list$$

$$\sigma_{hd} = T_3\ list \to T_3$$

$$\sigma_{tl} = T_4\ list \to T_4\ list$$

$$\sigma_{cons} = (T_5 \times T_5\ list) \to T_5\ list$$

map, f, m

$\sigma_{map} = \sigma_f \times \sigma_m \to \rho_1$

$\sigma_{null} = \sigma_m \to bool$

$\sigma_{hd} = \sigma_m \to \rho_2$

$\sigma_{tl} = \sigma_m \to \rho_3$

$\sigma_f = \_\_ \to \rho_4$

$\sigma_{map} = \sigma_f \times \rho_3 \to \rho_5$

$\sigma_{cons} = \rho_4 \times \rho_5 \to \rho_6$

$\rho_1 = \sigma_{null} = \rho_6$

Ex 2: Tagging

$$(b, c) \mapsto ((a, b), (a, c))$$

option 1:

let tagPair(a) =

$\lambda(b, c) \cdot ((a, b), (a, c))$

$$\alpha \to (\beta \times \gamma \to (\alpha \times \beta) \times (\alpha \times \gamma))$$

option 2:

Infixed function

$$\#:(\alpha \to \beta) \times (\gamma \to \delta) \to ((\alpha \times \gamma) \to (\beta \times \delta))$$

Such that

$$(f \# g)(a, c) = (f(a), g(c))$$

Pairing function

$$pair : \alpha \to (\beta \to \alpha \times \beta)$$

```
let tagPair =
    λa.(let tag = pair(a)
        in tag # tag
```

$a : \alpha$

$pair(a) : \delta \to \alpha \times \delta$

$tag \# tag :$

$\beta \to \alpha_1 \times \beta, \quad \gamma \to \alpha_2 \times \gamma$

$\beta \times \gamma \to (\alpha_1 \times \beta) \times (\alpha_2 \times \gamma)$

$tagPair :$

$$\boxed{\alpha \to (\beta \times \gamma \to (\alpha_1 \times \beta) \times (\alpha_2 \times \gamma))}$$

Does that make sense?

option 1:

let tagPair(a) = $\lambda(b,c).((a,b)\times(b,c))$

$\alpha \to (\beta \times \gamma \to (\alpha \times \beta) \times (\alpha \times \gamma))$

option 2:

let tagPair =

$\lambda a.$ (let tag = pair(a) in tag # tag)

$\alpha \Rightarrow (\beta \times \gamma \to (\alpha_1 \times \beta) \times (\alpha_2 \times \gamma))$

(2) Algorithm W?

Prove It!

1982: Damas-Milner

"Principal TYPE-Schemes for Functional Programs"

- The Language: Exp
- Algorithm W
- Proof of concept

Exp

Source: Damas-Milner '82

$x \in Id$

$e ::= x$

$\mid (e_1, e_2)$

$\mid \lambda x \cdot e$

$\mid \text{let } x = e_1 \text{ in } e_2$

Type Scheme of Exp

$$\tau ::= \alpha \mid \iota \mid \tau \to \tau$$

- α: type variables
- ι: primitive types (iota)
- $\tau \to \tau$: function types

$$\sigma ::= \tau \mid \forall \alpha \sigma$$

- σ: type scheme

$$S : [\tau_i / \alpha_i]$$

- S: substitution of types for type variables

Algorithm W

$W(A, e) = (S, \tau)$ where

(A: assumptions, e: expression, S: type scheme, τ: type)

(1) if e is X and there is an assumption $X : \forall \alpha_1 \ldots \alpha_n \tau'$ in A then:

$S = Id$ and

$\tau = [\beta_i / \alpha_i] \tau'$

(β_i: new type variables)

(2) If e is $(e_1 e_2)$ then:

let $W(A, e_2) = (S_1, T_2)$

and $W(S_1 A, e_2) = (S_2, T_2)$

and $U(S_2 T_1, T_2 \to \beta) = V$

then $S = V S_2 S_1$ and $T = V\beta$

U is an algorithm which takes in a pair of types and either returns a substitution V or fails

(3) If e is $\lambda x.e$ then:

$W(A_x \cup \{x:\beta\}, e_1) = (S_1, \tau_1)$

A_x: assumptions without x

then $S = S_1$ and $\tau = S_1\beta \to \tau_1$

(4) If e is let $x = e_1$ in e_2 then:

let $W(A, e_1) = (S_1, \tau_2)$

and $W(S_1 A_x \cup \{x : \overline{S_1 A}(\tau_1)\}, e_2)$

$= (S_2, \tau_2)$

then $S = S_2 S_1$

$\tau = \tau_2$

$\overline{S_1 A}(\tau_1)$ are the type vars that are free in τ_1 but not in A

Proof of Concept

- Soundness of W

If $W(A, e)$ succeeds with (S, τ) then there is a derivation of $SA \vdash e : \tau$

Proof

By induction on e using Proposition 2 [D-M]

- Completeness of W

  If $A \vdash e : \sigma$, for some σ, then W computes a principal type scheme for e under A [D-M]

- Decidability can be derived from the completeness theorem

(3) Polynomial, how about Exponential

1990: Mairson

"Deciding ML typability is Complete for Deterministic Exponential Time"

Folklore

ML expressions can be efficiently typed in Polynomial Time

DEXPTIME-hard to decide ML Typability

The worst case for deciding typability is with nested let bindings

Ex: let $X_1 = \lambda y.\{y, y\}$

in let $X_2 = \lambda y.X_1(X_1(y))$

in ...

in let $X_n = \lambda y.X_{n-1}(X_{n-1}(y))$

in $X_n(\lambda z.z)$

$\{x, y\}$ is an abbreviation for pairing

$\lambda z.zxy$

Upper bound

- Process
  - Simulate a Turing machines transition function in ML
  - The TM
    - marks off exponential amount of tape
    - Write the input
    - return the leftmost end marker
    - Start Simulation

The problem occurs when Unifying the type in the end reject? state

(4) I can't Decide

1993: Henglein

"Type Inference With Polymorphic Recursion"

Reduce to Semi unification

- Milner-Mycroft Calculus
  - Extension of Damas-Milner Calculus

- Milner-Mycroft Calculus is log space equivalent to semi-unification

- Semi-unification is a problem known to be undecidable

Small types

Expression e of size n has a small typing if it has a well-typed version e' of at most size $p(n)$ for a fixed polynomial p

- typability with small types is NP-complete