Algorithm W

Hindley-Milner Inference algorithm

Introduction

- Hindley–Milner (HM) type system is a classical type system for the lambda calculus with parametric polymorphism.
- Does Type Inference without programmers’ annotations
- Rather high complexity

Simply Untyped Lambda Calculus

$$e ::= x \mid \lambda x.\, e \mid e\, e$$

Simply Untyped Lambda Calculus

$$e ::= x \mid \lambda x.\, e \mid e\, e$$

Simply Untyped Lambda Calculus

$$e ::= x \mid \lambda x.\, e \mid e\, e$$

Simply Untyped Lambda Calculus

$$e ::= x \mid \lambda x.\, e \mid e\, e$$

Simply Untyped Lambda Calculus

$$e ::= x \mid \lambda x.\, e \mid e\, e$$

Simply Untyped Lambda Calculus

$$e ::= x \mid \lambda x.\, e \mid e\, e$$

Simply Typed Lambda Calculus

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\, e \mid c$$

Simply Typed Lambda Calculus

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\, e \mid c$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma} \text{ (1)}$$

$$\frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T} \text{ (2)}$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)} \text{ (3)}$$

$$\frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau} \text{ (4)}$$

Simply Typed Lambda Calculus

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\, e \mid c$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma}\ (1) \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T}\ (2)$$

$$\frac{\Gamma, \boxed{x{:}\sigma} \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)}\ (3) \qquad \frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau}\ (4)$$

Simply Typed Lambda Calculus

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\, e \mid c$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma} \; (1)$$

$$\frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T} \; (2)$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)} \; (3)$$

$$\frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau} \; (4)$$

Simply Typed Lambda Calculus

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\, e \mid c$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma} \; (1) \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T} \; (2)$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)} \; (3) \qquad \frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau} \; (4)$$

Simply Typed Lambda Calculus

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\, e \mid c$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma}\ (1) \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T}\ (2)$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)}\ (3) \qquad \frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau}\ (4)$$

Simply Typed Lambda Calculus

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\, e \mid c$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma}\ (1) \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T}\ (2)$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)}\ (3) \qquad \frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau}\ (4)$$

Simply Typed Lambda Calculus

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e\, e \mid c$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma} \; (1) \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T} \; (2)$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)} \; (3) \qquad \frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau} \; (4)$$

Simple Lambda Calculus with Polymorphism

- If we don't have polymorphism...

Simple Lambda Calculus with Polymorphism

- If we don't have polymorphism...
  - (λ id. (foo (id 1) (id 'two'))) ($\lambda\ x.x$)

Simple Lambda Calculus with Polymorphism

- If we don’t have polymorphism...
  - (λ id. (foo (id 1) (id ’two’))) ($\lambda\, x.\, x$)

Simple Lambda Calculus with Polymorphism

- If we don't have polymorphism...
  - (λ id. (foo (id 1) (id 'two'))) $(\lambda\, x.\, x)$

Simple Lambda Calculus with Polymorphism

- If we don't have polymorphism...
  - (λ id. (foo (id 1) (id 'two'))) $(\lambda\, x.\, x)$

Simple Lambda Calculus with Polymorphism

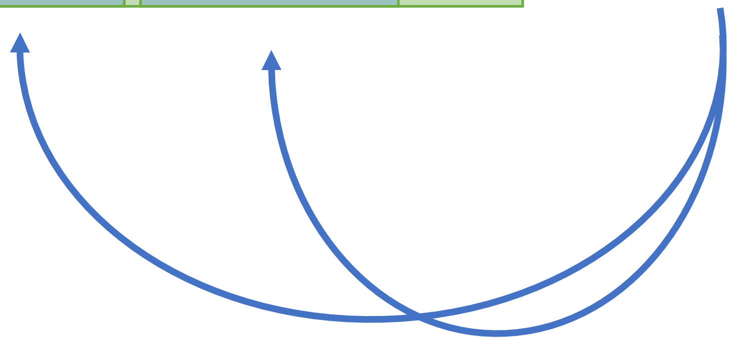
- If we don't have polymorphism...
  - (λ id. (foo (id 1) (id 'two'))) $(\lambda\, x.\, x)$

Simple Lambda Calculus with Polymorphism

- If we don't have polymorphism...
  - (λ id. (foo (id 1) (id 'two'))) $(\lambda\, x.\, x)$

Simple Lambda Calculus with Polymorphism

- If we don't have polymorphism...
  - (λ id. (foo (id 1) (id 'two'))) $(\lambda\, x.\, x)$



Simple Lambda Calculus with Polymorphism

- Motivating example
  - (λ id. (foo (id 1) (id 'two'))) ($\lambda\ x. x$)

Generalize

$\forall x. x \to x$

Simple Lambda Calculus with Polymorphism

- Motivating example
  - (λ id. (foo (id 1) (id 'two'))) ($\lambda\, x. x$)

Generalize

$\forall x. x \rightarrow x$

Simple Lambda Calculus with Polymorphism

- Motivating example
  - (λ id. (foo (id 1) (id ’two’))) ($\lambda\, x.\, x$)

Instantiate

$\lambda\, x\!: int\, .\, x \rightarrow x$

Instantiate

$\lambda\, x\!: char\, .\, x \rightarrow x$

Simple Lambda Calculus with Polymorphism

- Motivating example
  - (λ id. (foo (id 1) (id 'two'))) $(\lambda\, x.\, x)$

Instantiate

Instantiate

$\lambda\, x{:}\, int\, .\, x \rightarrow x$ $\lambda\, x{:}\, char\, .\, x \rightarrow x$

- Let polymorphism
  - let x = e in e'

Simple Lambda Calculus with Polymorphism

- Motivating example
  - (λ id. (foo (id 1) (id 'two'))) ($\lambda\, x.\, x$)

Instantiate

Instantiate

$\lambda\, x\!: int\, .\, x \rightarrow x$

$\lambda\, x\!: char\, .\, x \rightarrow x$

- Let polymorphism
  - let x = e in e'
  - let id = λx.x in foo (id 1) (id 'two')

Syntax of HM-type system

$$
\begin{array}{llll}
e & = & x & \text{variable} \\
  & | & e_1\ e_2 & \text{application} \\
  & | & \lambda\, x\, .\ e & \text{abstraction} \\
  & | & \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 &
\end{array}
$$

```haskell
type Var = String

data Exp =  ExpVar  Var
         |  ExpBool Bool
         |  ExpInt  Integer
         |  ExpChar Char
         |  ExpString String
         |  ExpLam  Var Exp -- if want 2 parameters, write a nested lambda expr
         |  ExpApp  Exp Exp
         |  ExpLet  Var Exp Exp
         -- some binary operations
         |  ExpAdd  Exp Exp
         |  ExpEql  Exp Exp
         |  ExpSub  Exp Exp
         |  ExpMul  Exp Exp
         deriving (Eq, Ord)

data Type = TypeVar Var
          | TypeArr Type Type
          | TypeInt
          | TypeBool
          | TypeChar
          | TypeString
          deriving (Eq, Ord)
```

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*
  - *Infer fun x->x* [Abstraction rule] :: a->a

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*
  - *Infer fun x->x* [Abstraction rule] :: a->a
  - Generalize *fun x->x* [Generalization rule] :: $\forall a. a \rightarrow a$

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*
  - *Infer fun x->x* [Abstraction rule] :: a->a
  - Generalize *fun x->x* [Generalization rule] :: $\forall a. a \rightarrow a$

  - Infer *fun y -> fun z -> y (id true) (id 1)* [Application rule]

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*
  - *Infer fun x->x* [Abstraction rule] :: a->a
  - Generalize *fun x->x* [Generalization rule] :: $\forall a. a \to a$

  - Infer *fun y -> fun z -> y (id true) (id 1)* [Application rule]
    - Infer *fun y -> fun z -> y* [Abstraction rule] :: b->c->b

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*
  - *Infer fun x->x* [Abstraction rule] :: a->a
  - Generalize *fun x->x* [Generalization rule] :: $\forall a. a \rightarrow a$

  - Infer *fun y -> fun z -> y (id true) (id 1)* [Application rule]
    - Infer *fun y -> fun z -> y* [Abstraction rule] :: b->c->b
    - Infer *(id true)* [Application rule]

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*
  - *Infer fun x->x* [Abstraction rule] :: a->a
  - Generalize *fun x->x* [Generalization rule] :: $\forall a. a \rightarrow a$

  - Infer *fun y -> fun z -> y (id true) (id 1)* [Application rule]
    - Infer *fun y -> fun z -> y* [Abstraction rule] :: b->c->b
    - Infer *(id true)* [Application rule]
      - Infer *id* [Var rule & Instantiation rule] :: d -> d

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*
  - *Infer fun x->x* [Abstraction rule] :: a->a
  - Generalize *fun x->x* [Generalization rule] :: $\forall a. a \rightarrow a$

  - Infer *fun y -> fun z -> y (id true) (id 1)* [Application rule]
    - Infer *fun y -> fun z -> y* [Abstraction rule] :: b->c->b
    - Infer *(id true)* [Application rule]
      - Infer *id* [Var rule & Instantiation rule] :: d -> d

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*
  - *Infer fun x->x* [Abstraction rule] :: a->a
  - Generalize *fun x->x* [Generalization rule] :: $\forall a. a \rightarrow a$

  - Infer *fun y -> fun z -> y (id true) (id 1)* [Application rule]
    - Infer *fun y -> fun z -> y* [Abstraction rule] :: b->c->b
    - Infer *(id true)* [Application rule]
      - Infer *id* [Var rule & Instantiation rule] :: d -> d
      - Unification id, bool->returnType :: d->d = bool->returnType

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*
    - *Infer fun x->x* [Abstraction rule] :: a->a
    - Generalize *fun x->x* [Generalization rule] :: $\forall a. a \to a$

    - Infer *fun y -> fun z -> y (id true) (id 1)* [Application rule]
        - Infer *fun y -> fun z -> y* [Abstraction rule] :: b->c->b
        - Infer *(id true)* [Application rule]
            - Infer *id* [Var rule & Instantiation rule] :: d -> d
            - Unification id, bool->returnType :: d->d = bool->returnType

Worked Example

- *let id = fun x -> x in fun y -> fun z -> y (id true) (id 1)*
  - *Infer fun x->x* [Abstraction rule] :: a->a
  - Generalize *fun x->x* [Generalization rule] :: $\forall a. a \rightarrow a$

  - Infer *fun y -> fun z -> y (id true) (id 1)* [Application rule]
    - Infer *fun y -> fun z -> y* [Abstraction rule] :: b->c->b
    - Infer *(id true)* [Application rule]
      - Infer *id* [Var rule & Instantiation rule] :: d -> d
      - Unification id, bool->returnType :: d->d = bool->returnType
    - Same for (id 1)

Type rules

$$\frac{\Gamma \vdash e_1 : \tau_1, S_1 \qquad S_1\Gamma, x : Gen(\tau_1) \vdash e_2 : \tau_2, S_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau, S_2 S_1} \quad (\text{T-L{\scriptsize ET}})$$

Type rules

$$\frac{\Gamma \vdash e_1 : \tau_1, S_1 \qquad S_1\Gamma, x : Gen(\tau_1) \vdash e_2 : \tau_2, S_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau, S_2 S_1} \quad \text{(T-LET)}$$

```
ti env (ExpLet x e1 e2) =
    do (s1, t1) <- ti env e1
       let TypeEnv env' = remove env x
           t' = generalize (substitute s1 env) t1
           env'' = TypeEnv (Map.insert x t' env')
       (s2, t2) <- ti (substitute s1 env'') e2
       return (s1 `mergeSubst` s2, t2)
```

Type rules

$$\frac{\Gamma \vdash e_1 : \tau_1, S_1 \qquad S_1\Gamma, x : Gen(\tau_1) \vdash e_2 : \tau_2, S_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau, S_2 S_1} \quad (\text{T-Let})$$

```
ti env (ExpLet x e1 e2) =
    do (s1, t1) <- ti env e1
       let TypeEnv env' = remove env x
           t' = generalize (substitute s1 env) t1
           env'' = TypeEnv (Map.insert x t' env')
       (s2, t2) <- ti (substitute s1 env'') e2
       return (s1 `mergeSubst` s2, t2)
```

Type rules

$$\frac{\Gamma \vdash e_1 : \tau_1, S_1 \qquad S_1\Gamma, x : Gen(\tau_1) \vdash e_2 : \tau_2, S_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau, S_2 S_1} \quad \text{(T-Let)}$$

```
ti env (ExpLet x e1 e2) =
    do (s1, t1) <- ti env e1
       let TypeEnv env' = remove env x
           t' = generalize (substitute s1 env) t1
           env'' = TypeEnv (Map.insert x t' env')
       (s2, t2) <- ti (substitute s1 env'') e2
       return (s1 `mergeSubst` s2, t2)
```

Type rules

$$\frac{\Gamma \vdash e_1 : \tau_1, S_1 \qquad S_1\Gamma, x : Gen(\tau_1) \vdash e_2 : \tau_2, S_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau, S_2 S_1} \quad \text{(T-LET)}$$

```haskell
ti env (ExpLet x e1 e2) =
    do (s1, t1) <- ti env e1
       let TypeEnv env' = remove env x
           t' = generalize (substitute s1 env) t1
           env'' = TypeEnv (Map.insert x t' env')
       (s2, t2) <- ti (substitute s1 env'') e2
       return (s1 `mergeSubst` s2, t2)
```

Features

- Support 4 basic types, char, string, int, boolean
- Support basic arithmetic operations, comparison, multiply, add.
- Abstraction, Application, Variable, Let-Polymorphism
- Overloading “==” and “+”

Implementation

- Infer monad
  - Error Handling
  - State Tracking

```haskell
-- do error handling and state tracking
type InferMonad a = ExceptT String (StateT TIState IO) a

runInferMonad :: InferMonad a -> IO (Either String a, TIState)
runInferMonad t =
    do (res, st) <- runStateT (runExceptT t) initTIState
       return (res, st)
  where initTIState = TIState{tiSupply = 0}
```

Implementation

- Pretty Print

```haskell
module Pretty ( prExp,
                prParenExp,
                prType,
                prParenType,
                prPoly
              )where

import Syntax
import Text.PrettyPrint
import Prelude hiding ((<>))

instance Show Exp where
    showsPrec _ x = shows (prExp x)

prExp                       ::  Exp -> Doc
prExp (ExpBool b)            =   if b then text "true" else text "false"
prExp (ExpInt i)             =   integer i
prExp (ExpVar name)          =   text name
prExp (ExpString name)       =   text name
prExp (ExpChar name)         =   char name
prExp (ExpLet x b body)      =   text "let" <+>
                               text x <+> text "=" <+>
                               prExp b <+> text "in" $$
                               nest 2 (prExp body)
prExp (ExpApp e1 e2)         =  text "$" <+> prExp e1 <+> prParenExp e2
prExp (ExpLam n e)           =   text "fun" <+> text n <+>
                               text "->" <+>
                               prExp e
prExp (ExpAdd e1 e2)         = prExp e1 <+> char '+' <+> prExp e2
prExp (ExpSub e1 e2)         = prExp e1 <+> char '-' <+> prExp e2
prExp (ExpMul e1 e2)         = prExp e1 <+> char '*' <+> prExp e2
prExp (ExpEql e1 e2)         = prExp e1 <+> text "==" <+> prExp e2
```