Type inference from scratch

Theory and implementation

Christoph Hegemann

Disclaimer

Hi, I’m Christoph.

Hi, I’m Christoph. I love types.

Hi, I’m Christoph. I love types. I hate typing.

Hi, I’m Christoph. I love types. I hate typing. I hate typing types.

So let’s learn about type inference!

What is type inference?

Definition
Type inference refers to the automatic detection of the type of an expression in a programming language. - Wikipedia

What is type inference?

Definition
Type inference refers to the automatic detection of the type of an expression in a programming language. - Wikipedia

What is type inference?

Definition
Type inference refers to the automatic detection of the type of an expression in a programming language. - Wikipedia

What is type inference?

Definition
Type inference refers to the automatic detection of the type of an expression in a programming language. - Wikipedia

Our language

We'll be implementing a type inference algorithm for the Lambda Calculus, extended with let-bindings, integer literals, and boolean literals.

Expressions

1

Expressions

```
1
\x -> x
```

Expressions

```
1
```

```
\x -> x
```

```
let const = \a -> \b -> a
in const 1 true
```

Defining our expression language

```
E ::=
   1                  -- int literal
```

Defining our expression language

```
E ::=
   1                -- int literal
   | true/false     -- boolean literal
```

Defining our expression language

```
E ::=
   1                -- int literal
   | true/false     -- boolean literal
   | x              -- Variable
```

Defining our expression language

```
E ::=
   1                -- int literal
   | true/false     -- boolean literal
   | x              -- Variable
   | E E            -- Application
```

Defining our expression language

```
E ::=
   1                -- int literal
   | true/false     -- boolean literal
   | x              -- Variable
   | E E            -- Application
   | \x -> E        -- Lambda
```

Defining our expression language

```
E ::=
   1                -- int literal
   | true/false     -- boolean literal
   | x              -- Variable
   | E E            -- Application
   | \x -> E        -- Lambda
   | let x = E in E -- Let binding
```

Value AST

```
E ::=
    1
    | true/false
    | x
    | E E
    | \x -> E
    | let x = E in E
```

Value AST

```
data Lit
  = LInt Integer
  | LBool Bool

data Exp
  = ELit Lit
```

```
E ::=
    1
  | true/false
  | x
  | E E
  | \x -> E
  | let x = E in E
```

Value AST

```
data Lit
  = LInt Integer
  | LBool Bool

data Exp
  = ELit Lit
  | EVar Text
```

```
E ::=
   1
   | true/false
   | x
   | E E
   | \x -> E
   | let x = E in E
```

Value AST

```
data Lit
  = LInt Integer
  | LBool Bool

data Exp
  = ELit Lit
  | EVar Text
  | EApp Exp Exp
```

```
E ::=
   1
   | true/false
   | x
   | E E
   | \x -> E
   | let x = E in E
```

Value AST

```haskell
data Lit
  = LInt Integer
  | LBool Bool

data Exp
  = ELit Lit
  | EVar Text
  | EApp Exp Exp
  | ELam Text Exp
```

```
E ::=
    1
  | true/false
  | x
  | E E
  | \x -> E
  | let x = E in E
```

Value AST

```
data Lit
  = LInt Integer
  | LBool Bool

data Exp
  = ELit Lit
  | EVar Text
  | EApp Exp Exp
  | ELam Text Exp
  | ELet Text Exp Exp
```

```
E ::=
   1
   | true/false
   | x
   | E E
   | \x -> E
   | let x = E in E
```

Types

$$Int$$

Types

$$Int$$

$$Int \rightarrow Bool$$

Types

$$Int$$

$$Int \to Bool$$

$$\forall a.\ a \to a$$

Types

$$Int$$

$$Int \to Bool$$

$$\forall a.\ a \to a$$

$$\forall a\ b.\ (a \to b) \to a \to b$$

Defining our type language

$T ::=$

Defining our type language

```
T ::=
      Int    -- primitive
    | Bool   -- primitive
```

Defining our type language

```
T ::=
      Int    -- primitive
    | Bool   -- primitive
    | a      -- type variable
```

Defining our type language

```
T ::=
      Int    -- primitive
    | Bool   -- primitive
    | a      -- type variable
    | T → T -- function type
```

Defining our type language

```
T ::=
      Int    -- primitive
    | Bool   -- primitive
    | a      -- type variable
    | T → T  -- function type

Scheme :=
```

Defining our type language

```
T ::=
      Int    -- primitive
    | Bool   -- primitive
    | a      -- type variable
    | T → T -- function type

Scheme :=
      T
```

Defining our type language

```
T ::=
      Int    -- primitive
    | Bool   -- primitive
    | a      -- type variable
    | T → T  -- function type

Scheme :=
      T
    | ∀ a1 a2 an. T
```

Type AST

```
T ::=
  Int
  | Bool
  | a
  | T → T

Scheme := T
  | ∀ a1 a2 an. T
```

Type AST

```
data Type
  = TInt
  | TBool
```

```
T ::=
  Int
  | Bool
  | a
  | T → T

Scheme := T
  | ∀ a1 a2 an. T
```

Type AST

```
data Type
  = TInt
  | TBool
  | TVar Text
```

```
T ::=
  Int
  | Bool
  | a
  | T → T

Scheme := T
  | ∀ a1 a2 an. T
```

Type AST

```
data Type
  = TInt
  | TBool
  | TVar Text
  | TFun Type Type
```

```
T ::=
  Int
  | Bool
  | a
  | T → T

Scheme := T
  | ∀ a1 a2 an. T
```

Type AST

```haskell
data Type
  = TInt
  | TBool
  | TVar Text
  | TFun Type Type

data Scheme = Scheme [Text] Type
```

```
T ::=
  Int
  | Bool
  | a
  | T → T

Scheme := T
  | ∀ a1 a2 an. T
```

Warmup Quiz

1 : ?

$$1 : Int$$

`true` : ?

$\texttt{true} : \mathit{Bool}$

```
\x -> x : ?
```

`\x -> x` $: \forall a.\, a \to a$

```
\x -> \y -> x : ?
```

`\x -> \y -> x` $: \forall a\, b.\, a \to b \to a$

```
\x -> x 1 : ?
```

`\x -> x 1` $: \forall b.\ (Int \to b) \to b$

```
let x = 5 in x : ?
```

$\texttt{let x = 5 in x} : \mathit{Int}$

```
let id = \x -> x in id true : ?
```

`let id = \x -> x in id true` : *Bool*

```
let id = \x -> x in id : ?
```

`let id = \x -> x in id` $: \forall a.\, a \to a$

Judgements

Definition
A typing judgement describes a relation between a piece of syntax and its type.

Judgements

Definition
A typing judgement describes a relation between a piece of syntax and its type.

Notation

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

In order to show C, one needs to show A and B.

Inferring Lambdas

```
\binder → body
```

Inferring Lambdas

$$\underbrace{\texttt{\textbackslash binder}}_{\textit{tyBinder}} \to \texttt{body}$$

Inferring Lambdas

$$\underbrace{\texttt{\textbackslash binder}}_{\textcolor{blue}{tyBinder}} \rightarrow \underbrace{\texttt{body}}_{\textcolor{red}{tyBody}}$$

Inferring Lambdas

$$\underbrace{\underbrace{\texttt{\textbackslash binder}}_{\color{blue}{tyBinder}} \rightarrow \underbrace{\texttt{body}}_{\color{red}{tyBody}}}_{\color{blue}{tyBinder} \rightarrow \color{red}{tyBody}}$$

$$\frac{\quad}{\quad}$$

Inferring Lambdas

$$\underbrace{\underbrace{\texttt{\textbackslash binder}}_{tyBinder} \to \underbrace{\texttt{body}}_{tyBody}}_{tyBinder \to tyBody}$$

$$\frac{}{\texttt{\textbackslash binder} \to \texttt{body} : tyBinder \to tyBody}$$

Inferring Lambdas

$$\underbrace{\underbrace{\texttt{\textbackslash binder}}_{\color{blue}{tyBinder}} \to \underbrace{\texttt{body}}_{\color{red}{tyBody}}}_{\color{blue}{tyBinder} \to \color{red}{tyBody}}$$

$$\frac{\texttt{body} : \color{red}{tyBody}}{\texttt{\textbackslash binder} \to \texttt{body} : \color{blue}{tyBinder} \to \color{red}{tyBody}}$$

Inferring Lambdas

$$\underbrace{\underbrace{\texttt{\textbackslash binder}}_{\color{blue}{tyBinder}} \to \underbrace{\texttt{body}}_{\color{red}{tyBody}}}_{\color{blue}{tyBinder} \to \color{red}{tyBody}}$$

$$\frac{\texttt{body} : \color{red}{tyBody}}{\texttt{\textbackslash binder} \to \texttt{body} : \color{blue}{tyBinder} \to \color{red}{tyBody}}$$

Problem
We're missing the fact that when inferring we're the type of `body`, there is an additional variable `binder` in scope. We'll fix that by introducing *Contexts*.

Contexts

Definition
A Context is a mapping from (value level) variables to schemes.

Contexts

Definition
A Context is a mapping from (value level) variables to schemes.

Notation
We use capital greek letters to denote contexts, for our type system we'll only need one Γ (Gamma). The $\vdash$ symbol means "in the context"

$$\Gamma \vdash x : ty_1$$

In the context Γ, x has type ty_1.

Contexts

Definition
A Context is a mapping from (value level) variables to schemes.

Notation
We use capital greek letters to denote contexts, for our type system we'll only need one Γ (Gamma). The $\vdash$ symbol means "in the context"

$$\Gamma \vdash x : ty_1$$

In the context Γ, x has type ty_1.

Implementation

```
type Context = Map Text Scheme
```

Inferring Lambdas

$$\underbrace{\underbrace{\texttt{\textbackslash binder}}_{\color{blue}{tyBinder}} \rightarrow \underbrace{\texttt{body}}_{\color{red}{tyBody}}}_{\color{blue}{tyBinder} \rightarrow \color{red}{tyBody}}$$

$$\frac{\texttt{body} : \color{red}{tyBody}}{\texttt{\textbackslash binder} \rightarrow \texttt{body} : \color{blue}{tyBinder} \rightarrow \color{red}{tyBody}}$$

Inferring Lambdas

$$\underbrace{\underbrace{\texttt{\textbackslash binder}}_{\color{blue}{tyBinder}} \rightarrow \underbrace{\texttt{body}}_{\color{red}{tyBody}}}_{\color{blue}{tyBinder} \rightarrow \color{red}{tyBody}}$$

$$\frac{\texttt{body} : \color{red}{tyBody}}{\Gamma \vdash \texttt{\textbackslash binder} \rightarrow \texttt{body} : \color{blue}{tyBinder} \rightarrow \color{red}{tyBody}}$$

Inferring Lambdas

$$\underbrace{\underbrace{\texttt{\textbackslash binder}}_{\mathit{tyBinder}} \rightarrow \underbrace{\texttt{body}}_{\mathit{tyBody}}}_{\mathit{tyBinder} \rightarrow \mathit{tyBody}}$$

$$\frac{\Gamma, \texttt{binder} : \mathit{tyBinder} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{\textbackslash binder} \rightarrow \texttt{body} : \mathit{tyBinder} \rightarrow \mathit{tyBody}}$$

Infer with context

```
infer :: Expr -> Type
```

Infer with context

```
infer :: Context -> Expr -> Type
```

Infer with context

```
infer :: Context -> Expr -> TI Type
```

Fresh type variables

```
type TI a = State Int a

newTyVar :: TI Type
newTyVar = do
  s <- get
  put (s + 1)
  pure (TVar ("u" <> showT s))
```

Don't worry about this too much. Just remember we can call `newTyVar` whenever we want a type variable with a unique name.

Inferring Lambdas - implementation

$$\frac{}{\Gamma \vdash \texttt{\textbackslash binder} \rightarrow \texttt{body} : \mathit{tyBinder} \rightarrow \mathit{tyBody}}$$

Inferring Lambdas - implementation

$$\frac{}{\Gamma \vdash \texttt{\textbackslash binder} \rightarrow \texttt{body} : tyBinder \rightarrow tyBody}$$

```
infer :: Context -> Expr -> TI Type
infer ctx (ELam binder body) = do
```

Inferring Lambdas - implementation

$$\frac{}{\Gamma \vdash \texttt{\textbackslash binder} \rightarrow \texttt{body} : \mathit{tyBinder} \rightarrow \mathit{tyBody}}$$

```
infer :: Context -> Expr -> TI Type
infer ctx (ELam binder body) = do



  pure (TFun tyBinder tyBody)
```

Inferring Lambdas - implementation

$$\frac{}{\Gamma \vdash \texttt{\textbackslash binder} \rightarrow \texttt{body} : tyBinder \rightarrow tyBody}$$

```
infer :: Context -> Expr -> TI Type
infer ctx (ELam binder body) = do
  tyBinder <- newTyVar


  pure (TFun tyBinder tyBody)
```

Inferring Lambdas - implementation

$$\frac{\Gamma, \texttt{binder} : \mathit{tyBinder} \vdash}{\Gamma \vdash \texttt{\textbackslash binder} \rightarrow \texttt{body} : \mathit{tyBinder} \to \mathit{tyBody}}$$

```haskell
infer :: Context -> Expr -> TI Type
infer ctx (ELam binder body) = do
  tyBinder <- newTyVar
  let tmpCtx = Map.insert binder (Scheme [] tyBinder) ctx

  pure (TFun tyBinder tyBody)
```

Inferring Lambdas - implementation

$$\frac{\Gamma, \texttt{binder} : \mathit{tyBinder} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{\textbackslash binder} \rightarrow \texttt{body} : \mathit{tyBinder} \rightarrow \mathit{tyBody}}$$

```
infer :: Context -> Expr -> TI Type
infer ctx (ELam binder body) = do
  tyBinder <- newTyVar
  let tmpCtx = Map.insert binder (Scheme [] tyBinder) ctx
  tyBody <- infer tmpCtx body
  pure (TFun tyBinder tyBody)
```

Propagating information

```
|   \f -> const 1 (f true)
```

Propagating information

```
const 1 (f true) : ?              |  \f -> const 1 (f true)
```

Propagating information

```
const 1 (f true) : Int          |  \f -> const 1 (f true)
```

Propagating information

```
|  \f -> const 1 (f true)
```

Propagating information

Things we learn about f:

```
\f -> const 1 (f true)
```

Propagating information

Things we learn about f:

- f is a function

```
\f -> const 1 (f true)
```

Propagating information

Things we learn about f:

- f is a function
- f accepts a Bool as its first argument

```
\f -> const 1 (f true)
```

```
\f -> const 1 (f true) : ?
```

`\f -> const 1 (f true)` $: \forall a.\, (Bool \to a) \to Int$

Infer with substitutions

We need to propagate additional information when inferring.

```
infer ∷ Context → Expression → TI Type
```

Infer with substitutions

We need to propagate additional information when inferring.

```
infer :: Context → Expression → TI (Substitution, Type)
```

Substitution

Definition
A substitution is a mapping from type variables to types.

Substitution

Definition
A substitution is a mapping from type variables to types.

Notation
We write the substitution S, mapping type variables $var_1, \dots, var_n$ to the types $ty_1, \dots, ty_n$ like so:

$$S = [var_1 \mapsto ty_1,\ var_2 \mapsto ty_2,\ \dots,\ var_n \mapsto ty_n]$$

Substitution

Definition
A substitution is a mapping from type variables to types.

Notation
We write the substitution S, mapping type variables $var_1, \dots, var_n$ to the types $ty_1, \dots, ty_n$ like so:

$$S = [var_1 \mapsto ty_1,\ var_2 \mapsto ty_2,\ \dots,\ var_n \mapsto ty_n]$$

Implementation

```
type Substitution = Map Text Type
```

Applying a substitution

When applying a substitution to a type, we substitute all occurrences of variables in that type that also occur in the substitution.

$$[a \mapsto Int]\, a = ?$$

$$[a \mapsto Int]\, a = Int$$

$$[a \mapsto Int]\ (a \to a) = ?$$

$$[a \mapsto Int]\ (a \to a) = Int \to Int$$

$$[a \mapsto Int]\ (a \to b) = ?$$

$$[a \mapsto Int]\ (a \to b) = Int \to b$$

$$[a \mapsto Int]\ (\forall a.\ a) = ?$$

$$[a \mapsto Int]\ (\forall a.\ a) = \forall a.\ a$$

$$[a \mapsto Int, b \mapsto Bool]\ (\forall a.\ a \to b) = ?$$

$$[a \mapsto Int, b \mapsto Bool]\ (\forall a.\ a \to b) = \forall a.\ a \to Bool$$

Applying a substitution - Implementation

When applying a substitution to a type, we substitute all occurrences of variables in that type that also occur in the substitution.

Applying a substitution - Implementation

When applying a substitution to a type, we substitute all occurrences of variables in that type that also occur in the substitution.

```
applySubst :: Substitution -> Type -> Type
applySubst subst ty = case ty of
```

Applying a substitution - Implementation

When applying a substitution to a type, we substitute all occurrences of variables in that type that also occur in the substitution.

```
applySubst :: Substitution -> Type -> Type
applySubst subst ty = case ty of
  TVar var ->
    fromMaybe (TVar var) (Map.lookup var subst)
```

Applying a substitution - Implementation

When applying a substitution to a type, we substitute all occurrences of variables in that type that also occur in the substitution.

```haskell
applySubst :: Substitution -> Type -> Type
applySubst subst ty = case ty of
  TVar var ->
    fromMaybe (TVar var) (Map.lookup var subst)
  TFun arg res ->
    TFun (applySubst subst arg) (applySubst subst res)
```

Applying a substitution - Implementation

When applying a substitution to a type, we substitute all occurrences of variables in that type that also occur in the substitution.

```
applySubst :: Substitution -> Type -> Type
applySubst subst ty = case ty of
  TVar var ->
    fromMaybe (TVar var) (Map.lookup var subst)
  TFun arg res ->
    TFun (applySubst subst arg) (applySubst subst res)
  TInt -> TInt
  TBool -> TBool
```

Applying a substitution - Implementation

When applying a substitution to a scheme, we substitute all occurrences of *free* variables in that type that also occur in the substitution.

Applying a substitution - Implementation

When applying a substitution to a scheme, we substitute all occurrences of *free* variables in that type that also occur in the substitution.

```
applySubstScheme :: Substitution -> Scheme -> Scheme
applySubstScheme subst (Scheme vars t) =
  Scheme vars (applySubst (foldr Map.delete subst vars) t)
```

Applying a substitution - Implementation

When applying a substitution to a scheme, we substitute all occurrences of *free* variables in that type that also occur in the substitution.

```
applySubstScheme :: Substitution -> Scheme -> Scheme
applySubstScheme subst (Scheme vars t) =
  Scheme vars (applySubst (foldr Map.delete subst vars) t)
```

We delete all variables that are bound by the scheme from the substitution before applying it to the inner type.

Applying a substitution - Implementation

When applying a substitution to a context, we apply the substitution to every scheme in the context.

Applying a substitution - Implementation

When applying a substitution to a context, we apply the substitution to every scheme in the context.

```
applySubstCtx :: Substitution -> Context -> Context
applySubstCtx subst ctx =
  Map.map (applySubstScheme subst) ctx
```

Composing substitutions

Definition
Let S and U be substitutions then their composition for all types t is defined as:

$$(S \circ U)\ t = S\ (U\ t)$$

Composing substitutions

Definition
Let S and U be substitutions then their composition for all types t is defined as:

$$(S \circ U)\ t = S\ (U\ t)$$

Implementation

```
composeSubst :: Substitution -> Substitution -> Substitution
composeSubst s1 s2 = Map.union (Map.map (applySubst s1) s2) s1
```

Inferring Lambdas with Substitutions

```
infer :: Context -> Expr -> TI Type
infer ctx (ELam binder body) = do
  tyBinder <- newTyVar
  let tmpCtx = Map.insert binder (Scheme [] tyBinder) ctx
  tyBody <- infer tmpCtx body
  pure (TFun tyBinder tyBody)
```

Inferring Lambdas with Substitutions

```haskell
infer :: Context -> Expr -> TI (Substitution, Type)
infer ctx (ELam binder body) = do
  tyBinder <- newTyVar
  let tmpCtx = Map.insert binder (Scheme [] tyBinder) ctx
  (s1, tyBody) <- infer tmpCtx body
  pure (s1, TFun (applySubst s1 tyBinder) tyBody)
```

Inferring literals

When inferring literals we don't care about substitution, nor context.

```
infer :: Context -> Expr -> TI (Substitution, Type)
infer _ (ELit (LInt _)) = pure (emptySubst, TInt)
infer _ (ELit (LBool _)) = pure (emptySubst, TBool)
```

Inferring Let

$$\underbrace{\texttt{let x = }\underbrace{\texttt{expr}}_{\color{red}{tyExpr}}\texttt{ in }\underbrace{\texttt{body}}_{\color{blue}{tyBody}}}_{\color{blue}{tyBody}}$$

Inferring Let

$$\underbrace{\texttt{let x = }\underbrace{\texttt{expr}}_{tyExpr}\texttt{ in }\underbrace{\texttt{body}}_{tyBody}}_{tyBody}$$

$$\frac{}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : tyBody}$$

Inferring Let

$$\underbrace{\texttt{let x = } \underbrace{\texttt{expr}}_{tyExpr} \texttt{ in } \underbrace{\texttt{body}}_{tyBody}}_{tyBody}$$

$$\frac{\Gamma \vdash \texttt{expr} : tyExpr}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : tyBody}$$

Inferring Let

$$\underbrace{\texttt{let x = }\underbrace{\texttt{expr}}_{\textit{tyExpr}}\texttt{ in }\underbrace{\texttt{body}}_{\textit{tyBody}}}_{\textit{tyBody}}$$

$$\frac{\Gamma \vdash \texttt{expr} : \textit{tyExpr} \qquad \Gamma, \texttt{x} : \textit{tyExpr} \vdash \texttt{body} : \textit{tyBody}}{\Gamma \vdash \texttt{let x = expr in body} : \textit{tyBody}}$$

Inferring Let - Implementation

$$\frac{\Gamma \vdash \texttt{expr} : \mathit{tyExpr} \qquad \Gamma, \texttt{x} : \mathit{tyExpr} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : \mathit{tyBody}}$$

Inferring Let - Implementation

$$\frac{\Gamma \vdash \texttt{expr} : \mathit{tyExpr} \qquad \Gamma, \texttt{x} : \mathit{tyExpr} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{let x = expr in body} : \mathit{tyBody}}$$

Inferring Let - Implementation

$$\frac{\Gamma \vdash \texttt{expr} : \mathit{tyExpr} \qquad \Gamma, \texttt{x} : \mathit{tyExpr} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \texttt{expr} : \mathit{tyExpr} \qquad \Gamma, \texttt{x} : \mathit{tyExpr} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \mathtt{expr} : \mathit{tyExpr} \qquad \Gamma, \mathtt{x} : \mathit{tyExpr} \vdash \mathtt{body} : \mathit{tyBody}}{\Gamma \vdash \mathtt{let}\ \mathtt{x} = \mathtt{expr}\ \mathtt{in}\ \mathtt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do



  pure tyBody
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \texttt{expr} : \mathit{tyExpr} \qquad \Gamma, \texttt{x} : \mathit{tyExpr} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do



  pure tyBody
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \mathtt{expr} : \mathit{tyExpr} \qquad \Gamma, \mathtt{x} : \mathit{tyExpr} \vdash \mathtt{body} : \mathit{tyBody}}{\Gamma \vdash \mathtt{let}\ \mathtt{x} = \mathtt{expr}\ \mathtt{in}\ \mathtt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do
  tyExpr <- infer ctx expr


  pure tyBody
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \texttt{expr} : \mathit{tyExpr} \qquad \Gamma, \texttt{x} : \mathit{tyExpr} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do
  tyExpr ← infer ctx expr


  pure tyBody
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \mathtt{expr} : \mathit{tyExpr} \qquad \Gamma, \mathtt{x} : \mathit{tyExpr} \vdash \mathtt{body} : \mathit{tyBody}}{\Gamma \vdash \mathtt{let}\ \mathtt{x} = \mathtt{expr}\ \mathtt{in}\ \mathtt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do
  tyExpr <- infer ctx expr
  let tmpCtx = Map.insert x (Scheme [] tyExpr) ctx

  pure tyBody
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \texttt{expr} : \mathit{tyExpr} \qquad \Gamma, \texttt{x} : \mathit{tyExpr} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do
  tyExpr <- infer ctx expr
  let tmpCtx = Map.insert x (Scheme [] tyExpr) ctx

  pure tyBody
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \texttt{expr} : \mathit{tyExpr} \qquad \Gamma, \texttt{x} : \mathit{tyExpr} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do
  tyExpr <- infer ctx expr
  let tmpCtx = Map.insert x (Scheme [] tyExpr) ctx
  tyBody <- infer tmpCtx body
  pure tyBody
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \mathtt{expr} : \mathit{tyExpr} \qquad \Gamma, \mathtt{x} : \mathit{tyExpr} \vdash \mathtt{body} : \mathit{tyBody}}{\Gamma \vdash \mathtt{let}\ \mathtt{x} = \mathtt{expr}\ \mathtt{in}\ \mathtt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do
  (s1, tyExpr) <- infer ctx expr
  let tmpCtx = Map.insert x (Scheme [] tyExpr) ctx
  tyBody <- infer tmpCtx body
  pure tyBody
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \texttt{expr} : \mathit{tyExpr} \qquad \Gamma, \texttt{x} : \mathit{tyExpr} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do
  (s1, tyExpr) <- infer ctx expr
  let tmpCtx = Map.insert x (Scheme [] tyExpr) ctx
  (s2, tyBody) <- infer (applySubstCtx s1 tmpCtx) body
  pure tyBody
```

Inferring Let - Implementation

$$\frac{\Gamma \vdash \texttt{expr} : \mathit{tyExpr} \qquad \Gamma, \texttt{x} : \mathit{tyExpr} \vdash \texttt{body} : \mathit{tyBody}}{\Gamma \vdash \texttt{let}\ \texttt{x} = \texttt{expr}\ \texttt{in}\ \texttt{body} : \mathit{tyBody}}$$

```
infer ctx (Let x expr body) = do
  (s1, tyExpr) <- infer ctx expr
  let tmpCtx = Map.insert x (Scheme [] tyExpr) ctx
  (s2, tyBody) <- infer (applySubstCtx s1 tmpCtx) body
  pure (composeSubst s1 s2, tyBody)
```

Inferring Var

$$\underbrace{\texttt{var}}_{tyVar}$$

$$\frac{}{\Gamma \vdash \texttt{var} : tyVar}$$

Inferring Var

$$\underbrace{\mathtt{var}}_{tyVar}$$

$$\frac{\mathtt{var} : tyVar \in \Gamma}{\Gamma \vdash \mathtt{var} : tyVar}$$

Inferring Var

$$\underbrace{\texttt{var}}_{\color{blue}{tyVar}}$$

$$\frac{\texttt{var} : \color{blue}{tyVar} \color{black}{\in \Gamma}}{\Gamma \vdash \texttt{var} : \color{blue}{tyVar}}$$

```
infer ctx (EVar var) =
```

Inferring Var

$$\underbrace{\texttt{var}}_{tyVar}$$

$$\frac{\texttt{var} : tyVar \in \Gamma}{\Gamma \vdash \texttt{var} : tyVar}$$

```
infer ctx (EVar var) =
  case Map.lookup var ctx of
```

Inferring Var

$$\underbrace{\texttt{var}}_{tyVar}$$

$$\frac{\texttt{var} : tyVar \in \Gamma}{\Gamma \vdash \texttt{var} : tyVar}$$

```
infer ctx (EVar var) =
  case Map.lookup var ctx of
    Nothing ->
      throwError ("unbound variable: " <> showT var)
```

Inferring Var

$$\underbrace{\texttt{var}}_{tyVar}$$

$$\frac{\texttt{var} : tyVar \in \Gamma}{\Gamma \vdash \texttt{var} : tyVar}$$

```
infer ctx (EVar var) =
  case Map.lookup var ctx of
    Nothing ->
      throwError ("unbound variable: " <> showT var)
    Just scheme -> do
      pure (emptySubst, scheme)
```

Inferring Var

$$\underbrace{\mathsf{var}}_{tyVar}$$

$$\frac{\mathsf{var} : tyVar \in \Gamma}{\Gamma \vdash \mathsf{var} : tyVar}$$

```
infer ctx (EVar var) =
  case Map.lookup var ctx of
    Nothing ->
      throwError ("unbound variable: " <> showT var)
    Just scheme -> do
      pure (emptySubst, scheme)
```

Problem
The context contains *schemes* not types. Getting a type from a scheme is called instantiation.

Instantiation

Definition
We instantiate a scheme by replacing all bound variables with fresh type variables.

Example

$$\mathit{instantiate}\ (\forall a\ b.\ a \to b \to a) = u1 \to u2 \to u1$$

Implementing Instantiation

Definition
We instantiate a scheme by replacing all bound variables with fresh type variables.

Implementation

```
instantiate :: Scheme -> TI Type
instantiate (Scheme vars ty) = do
```

Implementing Instantiation

Definition
We instantiate a scheme by replacing all bound variables with fresh type variables.

Implementation

```
instantiate :: Scheme -> TI Type
instantiate (Scheme vars ty) = do
  newVars <- traverse (const newTyVar) vars
```

Implementing Instantiation

Definition
We instantiate a scheme by replacing all bound variables with fresh type variables.

Implementation

```
instantiate :: Scheme -> TI Type
instantiate (Scheme vars ty) = do
  newVars <- traverse (const newTyVar) vars
  let subst = Map.fromList (zip vars newVars)
```

Implementing Instantiation

Definition
We instantiate a scheme by replacing all bound variables with fresh type variables.

Implementation

```haskell
instantiate :: Scheme -> TI Type
instantiate (Scheme vars ty) = do
  newVars <- traverse (const newTyVar) vars
  let subst = Map.fromList (zip vars newVars)
  pure (applySubst subst ty)
```

Inferring Var with instantiaton

$\underbrace{\texttt{var}}_{tyVar}$

$$\frac{\texttt{var} : tyVar \in \Gamma}{\Gamma \vdash \texttt{var} : tyVar}$$

```
infer ctx (EVar var) =
  case Map.lookup var ctx of
    Nothing ->
      throwError ("unbound variable: " <> showT var)
    Just scheme -> do
      pure (emptySubst, scheme)
```

Inferring Var with instantiaton

$$\underbrace{\texttt{var}}_{tyVar}$$

$$\frac{\texttt{var} : tyVar \in \Gamma}{\Gamma \vdash \texttt{var} : tyVar}$$

```
infer ctx (EVar var) =
  case Map.lookup var ctx of
    Nothing ->
      throwError ("unbound variable: " <> showT var)
    Just scheme -> do
      ty <- instantiate scheme
      pure (emptySubst, ty)
```

Inferring applications

$$\underbrace{\underbrace{\texttt{fun}}_{tyFun}\ \underbrace{\texttt{arg}}_{\color{red}{tyArg}}}_{\color{blue}{tyRes}}$$

$$\frac{}{\Gamma \vdash \texttt{fun arg} : \color{blue}{tyRes}}$$

Inferring applications

$$\underbrace{\underbrace{\texttt{fun}}_{tyFun}\ \underbrace{\texttt{arg}}_{tyArg}}_{tyRes}$$

$$\frac{\Gamma \vdash \texttt{fun} : tyFun}{\Gamma \vdash \texttt{fun arg} : tyRes}$$

Inferring applications

$$\underbrace{\underbrace{\texttt{fun}}_{\mathit{tyFun}}\ \underbrace{\texttt{arg}}_{\mathit{tyArg}}}_{\mathit{tyRes}}$$

$$\frac{\Gamma \vdash \texttt{fun} : \mathit{tyFun} \qquad \Gamma \vdash \texttt{arg} : \mathit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \mathit{tyRes}}$$

Inferring applications

$$\underbrace{\underbrace{\texttt{fun}}_{tyFun}\ \underbrace{\texttt{arg}}_{\color{red}{tyArg}}}_{\color{blue}{tyRes}}$$

$$\frac{\Gamma \vdash \texttt{fun} : tyFun \qquad \Gamma \vdash \texttt{arg} : \color{red}{tyArg}}{\Gamma \vdash \texttt{fun arg} : \color{blue}{tyRes}}$$

Relation: $tyFun = \color{red}{tyArg} \color{black}{\rightarrow} \color{blue}{tyRes}$

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \mathit{tyFun} \qquad \Gamma \vdash \texttt{arg} : \mathit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \mathit{tyRes}}$$

$$\mathit{tyFun} = \mathit{tyArg} \to \mathit{tyRes}$$

```
infer ctx (App fun arg) = do
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \mathit{tyFun} \qquad \Gamma \vdash \texttt{arg} : \mathit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \mathit{tyRes}}$$

$$\mathit{tyFun} = \mathit{tyArg} \rightarrow \mathit{tyRes}$$

```
infer ctx (App fun arg) = do




  pure tyRes
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \mathit{tyFun} \qquad \Gamma \vdash \texttt{arg} : \mathit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \mathit{tyRes}}$$

$$\mathit{tyFun} = \mathit{tyArg} \to \mathit{tyRes}$$

```
infer ctx (App fun arg) = do
  tyRes <- newTyVar




  pure tyRes
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \mathit{tyFun} \qquad \Gamma \vdash \texttt{arg} : \mathit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \mathit{tyRes}}$$

$$\mathit{tyFun} = \mathit{tyArg} \rightarrow \mathit{tyRes}$$

```
infer ctx (App fun arg) = do
  tyRes <- newTyVar
  tyFun <- infer ctx fun


  pure tyRes
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \mathit{tyFun} \qquad \Gamma \vdash \texttt{arg} : \mathit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \mathit{tyRes}}$$

$$\mathit{tyFun} = \mathit{tyArg} \to \mathit{tyRes}$$

```
infer ctx (App fun arg) = do
  tyRes <- newTyVar
  tyFun <- infer ctx fun
  tyArg <- infer ctx arg

  pure tyRes
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \mathit{tyFun} \qquad \Gamma \vdash \texttt{arg} : \mathit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \mathit{tyRes}}$$

$$\mathit{tyFun} = \mathit{tyArg} \rightarrow \mathit{tyRes}$$

```
infer ctx (App fun arg) = do
  tyRes <- newTyVar
  tyFun <- infer ctx fun
  tyArg <- infer ctx arg
  ?
  pure tyRes
```

Unification

Definition
Unifying two types yields the most general substitution that when applied to both types makes them equal.

Notation

$$a \sqcup b = S$$

$$a \sqcup Int = ?$$

$$a \sqcup Int = [a \mapsto Int]$$

$$(a \to Int) \sqcup (Bool \to Int) = ?$$

$$(a \to Int) \sqcup (Bool \to Int) = [a \mapsto Bool]$$

$$(Bool \to Int) \sqcup (a \to Int) = ?$$

$$(Bool \to Int) \sqcup (a \to Int) = [a \mapsto Bool]$$

$$(a \to b) \sqcup (b \to a) = ?$$

$$(a \to b) \sqcup (b \to a) = [a \mapsto b]$$

Unification - Implementation

```
varBind :: Text -> Type -> TI Substitution
varBind var ty
```

Unification - Implementation

$$a \sqcup a = []$$

```
varBind :: Text -> Type -> TI Substitution
varBind var ty
```

Unification - Implementation

$$a \sqcup a = []$$

```haskell
varBind :: Text -> Type -> TI Substitution
varBind var ty
  | ty == TVar var = pure emptySubst
```

Unification - Implementation

$$a \sqcup (a \to a) = \text{ERROR}$$

```haskell
varBind :: Text -> Type -> TI Substitution
varBind var ty
  | ty == TVar var = pure emptySubst
```

Unification - Implementation

$$a \sqcup (a \to a) = \text{ERROR}$$

```haskell
varBind :: Text -> Type -> TI Substitution
varBind var ty
  | ty == TVar var = pure emptySubst
  | Set.member var (freeTypeVars ty) =
      throwError "occurs check failed"
```

Unification - Implementation

$$a \sqcup b = [a \mapsto b]$$

```haskell
varBind :: Text -> Type -> TI Substitution
varBind var ty
  | ty == TVar var = pure emptySubst
  | Set.member var (freeTypeVars ty) =
      throwError "occurs check failed"
```

Unification - Implementation

$$a \sqcup b = [a \mapsto b]$$

```
varBind :: Text -> Type -> TI Substitution
varBind var ty
  | ty == TVar var = pure emptySubst
  | Set.member var (freeTypeVars ty) =
      throwError "occurs check failed"
  | otherwise = pure (Map.singleton var ty)
```

Unification - Implementation

```
unify :: Type -> Type -> TI Substitution
```

Unification - Implementation

```
unify :: Type -> Type -> TI Substitution
unify TInt TInt = pure emptySubst
```

Unification - Implementation

```haskell
unify :: Type -> Type -> TI Substitution
unify TInt TInt = pure emptySubst
unify TBool TBool = pure emptySubst
```

Unification - Implementation

```haskell
unify :: Type -> Type -> TI Substitution
unify TInt TInt = pure emptySubst
unify TBool TBool = pure emptySubst
unify (TFun arg1 res1) (TFun arg2 res2) = do
  s1 <- unify arg1 arg2
  s2 <- unify (applySubst s1 res1) (applySubst s1 res2)
  pure (composeSubst s1 s2)
```

Unification - Implementation

```haskell
unify :: Type -> Type -> TI Substitution
unify TInt TInt = pure emptySubst
unify TBool TBool = pure emptySubst
unify (TFun arg1 res1) (TFun arg2 res2) = do
  s1 <- unify arg1 arg2
  s2 <- unify (applySubst s1 res1) (applySubst s1 res2)
  pure (composeSubst s1 s2)
unify (TVar u) t = varBind u t
unify t (TVar u) = varBind u t
```

Unification - Implementation

```haskell
unify :: Type -> Type -> TI Substitution
unify TInt TInt = pure emptySubst
unify TBool TBool = pure emptySubst
unify (TFun arg1 res1) (TFun arg2 res2) = do
  s1 <- unify arg1 arg2
  s2 <- unify (applySubst s1 res1) (applySubst s1 res2)
  pure (composeSubst s1 s2)
unify (TVar u) t = varBind u t
unify t (TVar u) = varBind u t
unify t1 t2 =
  throwError
    ("types do not unify: " <> showT t1 <> " vs. " <> showT t2)
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \textit{tyArg} \rightarrow \textit{tyRes} \qquad \Gamma \vdash \texttt{arg} : \textit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \textit{tyRes}}$$

$$\textit{tyFun} = \textit{tyArg} \rightarrow \textit{tyRes}$$

```
infer ctx (App fun arg) = do
  tyRes ← newTyVar
  tyFun ← infer ctx fun
  tyArg ← infer ctx arg
  ?
  pure tyRes
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \textit{tyArg} \to \textit{tyRes} \qquad \Gamma \vdash \texttt{arg} : \textit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \textit{tyRes}}$$

$$\textit{tyFun} = \textit{tyArg} \to \textit{tyRes}$$

```
infer ctx (App fun arg) = do
  tyRes <- newTyVar
  tyFun <- infer ctx fun
  tyArg <- infer ctx arg
  unify tyFun (TFun tyArg tyRes)
  pure tyRes
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : tyArg \to tyRes \qquad \Gamma \vdash \texttt{arg} : tyArg}{\Gamma \vdash \texttt{fun arg} : tyRes}$$

$$tyFun = tyArg \to tyRes$$

```
infer ctx (App fun arg) = do
  tyRes <- newTyVar
  (s1, tyFun) <- infer ctx fun
  tyArg <- infer ctx arg
  unify tyFun (TFun tyArg tyRes)
  pure tyRes
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \mathit{tyArg} \to \mathit{tyRes} \qquad \Gamma \vdash \texttt{arg} : \mathit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \mathit{tyRes}}$$

$$\mathit{tyFun} = \mathit{tyArg} \to \mathit{tyRes}$$

```
infer ctx (App fun arg) = do
  tyRes <- newTyVar
  (s1, tyFun) <- infer ctx fun
  (s2, tyArg) <- infer (applySubstCtx s1 ctx) arg
  unify tyFun (TFun tyArg tyRes)
  pure tyRes
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \mathit{tyArg} \to \mathit{tyRes} \qquad \Gamma \vdash \texttt{arg} : \mathit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \mathit{tyRes}}$$

$$\mathit{tyFun} = \mathit{tyArg} \to \mathit{tyRes}$$

```
infer ctx (App fun arg) = do
  tyRes <- newTyVar
  (s1, tyFun) <- infer ctx fun
  (s2, tyArg) <- infer (applySubstCtx s1 ctx) arg
  s3 <- unify (applySubst s2 tyFun) (TFun tyArg tyRes)
  pure tyRes
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \textit{tyArg} \rightarrow \textit{tyRes} \qquad \Gamma \vdash \texttt{arg} : \textit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \textit{tyRes}}$$

$$\textit{tyFun} = \textit{tyArg} \rightarrow \textit{tyRes}$$

```
infer ctx (App fun arg) = do
  tyRes <- newTyVar
  (s1, tyFun) <- infer ctx fun
  (s2, tyArg) <- infer (applySubstCtx s1 ctx) arg
  s3 <- unify (applySubst s2 tyFun) (TFun tyArg tyRes)
  let subst = composeSubst s3 (composeSubst s2 s1)
  pure tyRes
```

Inferring applications - Implementation

$$\frac{\Gamma \vdash \texttt{fun} : \mathit{tyArg} \to \mathit{tyRes} \qquad \Gamma \vdash \texttt{arg} : \mathit{tyArg}}{\Gamma \vdash \texttt{fun arg} : \mathit{tyRes}}$$

$$\mathit{tyFun} = \mathit{tyArg} \to \mathit{tyRes}$$

```
infer ctx (App fun arg) = do
  tyRes <- newTyVar
  (s1, tyFun) <- infer ctx fun
  (s2, tyArg) <- infer (applySubstCtx s1 ctx) arg
  s3 <- unify (applySubst s2 tyFun) (TFun tyArg tyRes)
  let subst = composeSubst s3 (composeSubst s2 s1)
  pure (subst, applySubst s3 tyRes)
```

Conclusion

We’ve implemented a fully functional type inferencer for a functional programming language in around 120 lines of Haskell.

This was just an introduction, but after this talk you should have enough context to pick up a paper on type systems on your own and have a go at implementing it.

Demo?

Demo?

All materials can be found at github.com/kRITZCREEK/fby19