Type Inference for OCaml in TypeScript

Gabriel Yeo & Sharan Thangavel

April 8, 2021

Table of Contents

Table of Contents

What is Type Inference?

Type inference refers to the automatic detection of the type of an expression in a formal language. [Wikipedia]

Why is Type Inference useful?

Removing redundancy from the POV of the programmer.
e.g. let a = (1 + 2) > (1 + 1)
let condition = true && false

Table of Contents

What is Hindley-Milner(-Damas) Type Inference?

A Hindley–Milner (HM) type system is a classical type system for the lambda calculus with parametric polymorphism.

Table of Contents

The expression language

Exactly those of lambda calculus + the let-expression

$$
\begin{aligned}
expr ::= {} & x & & & x = \{\text{valid ident}\} \\
\mid \quad & expr\ expr & & [application] \\
\mid \quad & \text{fun } x \rightarrow expr & & [abstraction] \\
\mid \quad & \text{let } x = expr \text{ in } expr & & [let]
\end{aligned}
$$

The expression language

Exactly those of lambda calculus + the let-expression + our global extension.

$$
\begin{array}{rlr}
\mathit{expr} ::= & x & \\
| & \mathit{expr}\ \mathit{expr} & [\mathit{application}] \\
| & \text{fun } x \to \mathit{expr} & [\mathit{abstraction}] \\
| & \text{let } x = \mathit{expr} \text{ in } \mathit{expr} & [\mathit{let}] \\
| & \text{let } x = \mathit{expr} & [\mathit{globalLet}]
\end{array}
\qquad x = \{\text{valid ident}\}
$$

Types

Monomorphic and Polymorphic.

$$\begin{array}{lll} \tau ::= \alpha & & [\mathit{typeVars}] \\ \quad \mid \ \iota & & [\mathit{literals}] \\ \quad \mid \ \tau \to \tau & & [\mathit{funTypes}] \end{array}$$

$$\iota = \{\mathit{int}, \mathit{string}, \mathit{bool}\}$$

$$\begin{array}{l} \sigma ::= \tau \\ \quad \mid \ \forall \alpha\, \sigma \end{array}$$

Types

Monomorphic and Polymorphic.

$$\begin{array}{rll} \tau ::= & \alpha & [\mathit{typeVars}] \\ | & \iota & [\mathit{literals}] \\ | & \tau \to \tau & [\mathit{funTypes}] \end{array}$$

$$\iota = \{\mathit{int},\ \mathit{string},\ \mathit{bool}\}$$

$$\begin{array}{rl} \sigma ::= & \tau \\ | & \forall \alpha\, \sigma \end{array}$$

Polymorphic types σ are also known as Type Schemes, i.e.:

$$\forall \alpha_1 \alpha_2 . \alpha_1 \to \alpha_2$$

Table of Contents

Warm up

$$expr : \sigma$$

Warm up

5:

Warm up

5: *int*

Warm up

let f = fun x → x in f 4:

Warm up

let f = fun x $\rightarrow$ x in f 4: *int*

Warm up

let kestrel = fun x $\rightarrow$ fun y $\rightarrow$ x:

Warm up

let kestrel = fun x $\rightarrow$ fun y $\rightarrow$ x: $\forall \alpha_1 \alpha_2 . \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1$

Warm up

kestrel "hello" :

Warm up

kestrel "hello": $\forall \alpha_2 . \alpha_2 \rightarrow$ *string*

Table of Contents

Instantiation and Generic Instantiation

Given a Type Scheme σ, we instantiate by substitution with some $S = \{a_i \mapsto \tau_i, ...\}$ to obtain $S\sigma$.

Instantiation and Generic Instantiation

Given a Type Scheme σ, we instantiate by substitution with some $S = \{a_i \mapsto \tau_i, ...\}$ to obtain $S\sigma$.

Example:
Let $\sigma = \forall\alpha_1.\alpha_1 \to \alpha_1$,
and $S = \{\alpha_1 \mapsto \mathit{string}\}$
then $S\sigma = \mathit{string} \to \mathit{string}$

Instantiation and Generic Instantiation

$$\frac{\tau' = \{\alpha_i \mapsto \tau_i\}\tau \quad \beta_i \notin \mathit{free}(\forall \alpha_1 \ldots \alpha_n . \tau)}{\forall \alpha_1 \ldots \alpha_n . \tau \sqsubseteq \forall \beta_1 \ldots \beta_m . \tau'} [\mathit{Specialisation}]$$

We define a partial order $\sqsubseteq$ based on the above. In other words $\sigma' \sqsubseteq \sigma$ if σ' is more general than σ.

Instantiation and Generic Instantiation

$$\frac{\tau' = \{\alpha_i \mapsto \tau_i\}\tau \quad \beta_i \notin free(\forall \alpha_1 ... \alpha_n . \tau)}{\forall \alpha_1 ... \alpha_n . \tau \sqsubseteq \forall \beta_1 ... \beta_m . \tau'} [Specialisation]$$

We define a partial order $\sqsubseteq$ based on the above. In other words $\sigma' \sqsubseteq \sigma$ if σ' is more general than σ.

Example:

Let $\sigma = \forall \alpha_1 . \alpha_1 \to \alpha_1$,

and $S = \{\alpha_1 \mapsto (int \to \beta_1)\}$

then $S\sigma = \forall \beta_1 . (int \to \beta_1) \to (int \to \beta_1)$

Table of Contents

Unification

"Unification is a recursive algorithm for determining a substitution of terms for variables (i.e. a variable assignment) that makes two terms equal. For example we can unify $f(a, y)$ with $f(x, f(b, x))$ with the substitution $[a/x, f(b, a)/y]$"
- (Ian Grant, 2011)

Unification

Algorithm (*Unification*)

To find the most general unifier $U = \mathrm{MGU}(t, t')$ of terms t and t'

(i) If $t = x$ and $t' = y$ then $U = [x/y]$.

(iia) If $t = x$ and $t' = f(s_1, s_2, \ldots, s_n)$ and x does not occur in t' then $U = [t'/x]$.

(iib) If $t = f(s_1, s_2, \ldots, s_n)$ and $t' = x$ and x does not occur in t then $U = [t/x]$.

(iii) If $t = a$ and $t' = a$ then $U = []$.

(iv) If $t = f(s_1, s_2, \ldots, s_n)$ and $t' = f(s_1, s_2, \ldots, s_n)$ then

$$U = \mathrm{MGU}(f(U_1 s_2, U_1 s_3, \ldots, s_n), f(U_1 s_2', U_1 s_3', \ldots, s_n)) \circ U_1$$

where $U_1 = \mathrm{MGU}(s_1', s_1')$.

In any other circumstances the algorithm fails. □

Unification

```
function unify(type1: AstType, type2: AstType): undefined {
    let t1 = prune(type1),
        t2 = prune(type2)
    if (t1 instanceof TypeVariable) {
        if (t1 !== t2) {
            if (occursInType(t1, t2))
                throw new UnificationError('Occurs Check')
            t1.instance = t2
        }
    }
    else if (t1 instanceof TypeOperator && t2 instanceof TypeVariable) {
        return unify(t2, t1)
    }
    else if (t1 instanceof TypeOperator && t2 instanceof TypeOperator) {
        if (t1.name !== t2.name || t1.types.length !== t2.types.length) {
            throw new UnificationError('Cannot unify: ' + t1 + ' != ' + t2)
        }
        t1.types.forEach((t, i) => unify((t1 as TypeOperator).types[i],
            (t2 as TypeOperator).types[i]))
    }
    else {
        throw new UnificationError('Failed to unify')
    }
}
```

Table of Contents

Contexts and Types

Recall our definition of a context Γ.

$$\begin{aligned} \Gamma &::= \epsilon \\ &\mid \quad \Gamma, x : \sigma \end{aligned}$$

$$\Gamma \vdash \mathit{expr} : \sigma$$

HM Inference Rules (Syntactic)

We now have all the information to define our rules of inference.

$$\frac{x : \sigma \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau}[Var] \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \text{fun } x \rightarrow e : \tau \rightarrow \tau'}[Abs]$$

$$\frac{\Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 e_1 : \tau'}[App] \qquad \frac{\Gamma \vdash e_0 : \tau \quad \Gamma, x : \bar{\Gamma}(\tau) \vdash e_1 : \tau'}{\Gamma \vdash \text{let } x = e_0 \text{ in } e_1 : \tau'}[Let]$$

$$\bar{\Gamma}(\tau) = \forall \hat{\alpha}.\tau \qquad \hat{\alpha} = free(\tau) - free(\Gamma)$$

Table of Contents

Var

$$\frac{x : \sigma \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau}[Var]$$

```
export function infer(node: AstNode, env: TypeEnv, nonGeneric: Set<AstType>): AstTypeEnvPair {
    if (node instanceof Id) {
        if (node.type === basicType.reference) {
            return new TypeEnvPair(env.get(node.name, nonGeneric), env)
        } else {
            return new TypeEnvPair(env.get(node.type, nonGeneric), env)
        }
    }
```

App

$$\frac{\Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\, e_1 : \tau'}[App]$$

```
else if (node instanceof Apply) {
    let c1 = infer(node.func, env, nonGeneric),
        funcType = c1.type,
        c2 = infer(node.arg, env, nonGeneric),
        argType = c2.type,
        retType = new TypeVariable()
    unify(FunctionType(argType, retType), funcType)
    return new TypeEnvPair(retType, env)
}
```

Abs

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \text{fun } x \rightarrow e : \tau \rightarrow \tau'}[Abs]$$

```
else if (node instanceof Lambda) {
    let argType = new TypeVariable(),
        newEnv = env.extend(node.args, argType),
        newGeneric = new Set(Array.from(nonGeneric).concat(argType)),
        context = infer(node.body, newEnv, newGeneric)
    return new TypeEnvPair(FunctionType(argType, context.type), context.env)
}
```

Let

$$\frac{\Gamma \vdash e_0 : \tau \quad \Gamma, x : \bar{\Gamma}(\tau) \vdash e_1 : \tau'}{\Gamma \vdash \text{let } x = e_0 \text{ in } e_1 : \tau'}[Let]$$

```
else if (node instanceof Let) {
    let newContext = infer(node.value, env, nonGeneric),
        newEnv = env.extend(node.variable, newContext.type),
        resultContext = infer(node.body, newEnv, nonGeneric)
    return new TypeEnvPair(resultContext.type, env)
}
```

Extension - GlobalLet

```
else if (node instanceof GlobalLet) {
    let newContext = infer(node.value, env, nonGeneric),
        newEnv = env.extend(node.variable, newContext.type)
    return new TypeEnvPair(newContext.type, newEnv)
}
```

Extension - BinaryOp

```
else if (node instanceof BinaryOp) {
    let operatorType = env.get(node.operator, nonGeneric),
        leftType = infer(node.left, env, nonGeneric),
        rightType = infer(node.right, env, nonGeneric),
        retType = new TypeVariable()
        unify(operatorType, FunctionType(leftType.type, FunctionType(rightType.type, retType)))
    return new TypeEnvPair(retType, env)
}
```

Table of Contents

Hindley-Milner Exponential Case

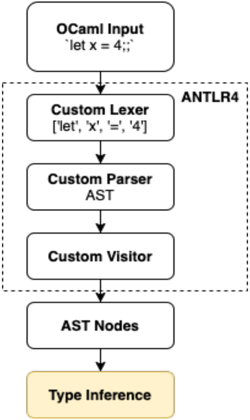
In certain cases like deeply-nested let bindings, the size of the type inferred increases exponentially.

let dup = fun x → (pair x x);;
let dup2 = dup dup;;
let dup3 = dup dup2;;

Table of Contents

Table of Contents

Parsing with ANTLR4TS



1. Establishing the supported OCaml Grammar
2. ANTLR-enabled Lexing
3. ANTLR-enabled Parsing
4. ANTLR-enabled Visiting
5. Custom AST Nodes for Type Inference

Table of Contents

ANTLR4 OCaml Grammar

```
expr:
    value_name                                                                    # valueName
    | constant_integer                                                            # constantInt
    | constant_boolean                                                            # constantBool
    | constant_string                                                             # constantStr
    | '(' inner = expr ')'                                                        # exprInParantheses
    | PREFIX_SYMBOL expr                                                          # exprWithPrefix
    | left = expr operator = infix_op right = expr                                # binaryOp
    | 'if' condition = expr 'then' consequent = expr ('else' alternative = expr)? # conditionalExpr
    | 'while' condition = expr 'do' body = expr 'done'                            # whileLoop
    | 'for' name = value_name '=' binding = expr ('to' | 'downto') end = expr 'do' body = expr 'done' # forLoop
    | first = expr ';' second = expr                                              # exprSemicolonExpr
    | 'fun' (params = parameter)+ '->' body = expr                                # lambda
    | fun = expr argument = expr                                                  # application
    | 'let' name = pattern '=' binding = expr 'in' in_context = expr              # letExpr
    | 'let' name = pattern '=' binding = expr                                     # globalLetExpr
    | 'let' 'rec' name = pattern '=' binding = expr 'in' in_context = expr        # letRecExpr
    | 'let' 'rec' name = pattern '=' binding = expr                               # globalLetRecExpr;
```

* Community Contribution

Custom ASTNodes

```
export class Let implements AstNode {
    constructor(public variable: Id, public value: AstNode, public body: AstNode) { }
    toString() { return `(let ${this.variable} = ${this.value} in ${this.body})` }
}
```

Custom Visitor

```
visitLetExpr(ctx: LetExprContext): AstNode {
  return new Let(this.visitPattern(ctx._name), this.visit(ctx._binding), this.visit(ctx._in_context))
}
```

Table of Contents

Type Checker Demo

`http://tinyurl.com/ocamltypechecker`

http://tinyurl.com/ocamltypechecker

- **‘true == true’** : bool
- **‘1 == true’** : Unification Error
- **‘let s = fun a → fun b → fun c → (a c (b c))’**
- **‘let k = fun x → fun y → x’**
- **‘s k k’** : t5 → t5
- **‘s (k s) k’** : ((t6 → t7) → ((t8 → t6) → (t8 → t7)))
- **‘let p = pair “first” 2’** : (string * int)
- **‘head p’** : string
- **‘tail p’** : int

Table of Contents

Useful links and resources

1. OCaml
2. antlr4ts
3. Principal type-schemes for functional programs, Luis Damas† and Robin Milner (1982)
4. The Hindley-Milner Type Inference Algorithm, Ian Grant (2011)
5. Hindley Milner Type System, Wikipedia
6. CS4215!