PoPL Assignment-5 Type Systems

Aryan Agarwal 19CS30005

1. Type Error: A type is a collection of computational entities, that share some common property (like int, bool, int → int). A type error represents an error when an operation could not be performed because a value is not of the expected type or when a computational entity is used in a manner that is inconsistent with the concept it represents.

   Eg. Adding a string to an integer.

Type System: A type system is a logical system comprising of a set of rules that assigns a property, type, to the various constructs of the program (i.e., variables, expressions, functions etc.). It is a tractable syntactic method for providing the absence of certain programming behaviours by classifying phrases according to the values they compute.

Advantages of a Type System

(i) Abstraction: Enables a programmer to think at a higher level without bothering about low-level implementation.

(ii) Documentation: Types can serve as a documentation, clarifying the intent of the programmer, making the program easy to understand.

(iii) Error detection: Static Type checking in Type Systems allows early detection of some programming errors.

(iv) Safety: A type system enables the compiler to detect meaningless or invalid code.

(v) Efficiency: A Type system eliminates many of the dynamic checks and results in more efficient execution of a program.

(2) Type Inference

Type Inference refers to the automatic detection of the type of an expression in a formal language. It is the process of determining the types of expressions based on the known types of some symbols in the expression.

Example (By Hand - Weaving)

Consider,

```
f x = 5 * x
```

We know

type of * : Int → Int → Int

type of 5 : Int

As we are applying '*' to x, we require –

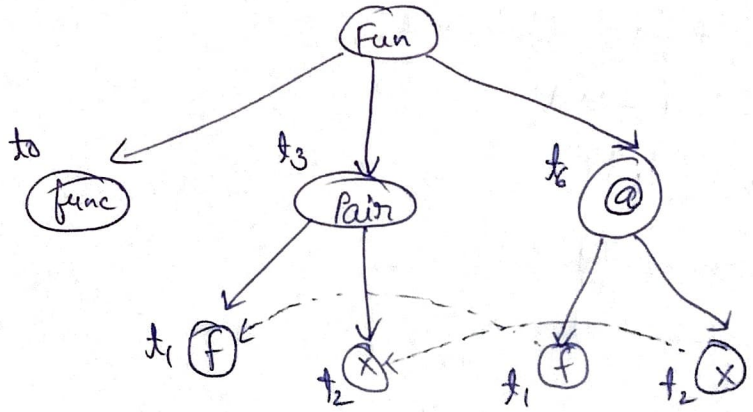
x :: Int

∴ f x = 5 * x, has type Int → Int

Type Inference Algorithm for Polymorphic function:

func (f, x) = f x

Parse tree :



Assigning type variables to the nodes of the parse tree:

func :: t_0

Pair :: t_3

f :: t_1

x :: t_2

@ :: t_6

Constraints

$t_1 = t_2 \rightarrow t_6$

$t_0 = t_3 \rightarrow t_6$

$t_3 = (t_1, t_2)$

Replacing t_3,

$t_0 = (t_1, t_2) \rightarrow t_6$

$t_1 = (t_2 \rightarrow t_6)$

$t_3 = (t_1, t_2)$

Replacing t_1,

$t_0 = (t_2 \rightarrow t_0, t_2) \rightarrow t_6$

$t_1 = t_2 \rightarrow t_6$

$t_3 = (t_1, t_2)$

Now, renaming, $t_2 \to t$ and $t_0 \to t'$

$t_0 : (t \to t', t) \to t'$

$t_1 = t \to t'$

$t_2 = (t, t')$

∴ Determine type

func(f, x) = fx

↳ func :: $(t \to t', t) \to t'$

[Where x :: t, fx :: t']

(3) Type Inference Algorithm

- Assign a type to the expression and each expression (as in the parse tree). For known operations or constants (like 2 or +) use the type already known. For compared expression or variable, use a type variable.
- Generate constraints or types using the parse tree obtained from the expression, using the fact that the ~~argument~~ argument of a function must be the same type as the domain of the function.
- Solve the constraints obtained by unification i.e., a substitution based algorithm for solving a system of equations.

Frame and solve type constraint using Matrix

Find type of $(A+BC)^2$ given:

A: $t_1 \to t_2$

B: $t_3 \to t_4$

C: $t_5 \to t_6$

Sub-expressions

A: $t_1 \to t_2$

B: $t_3 \to t_4$

C: $t_5 \to t_6$

BC: $t_7 \to t_8$

A+BC: $t_9 \to t_{10}$

$(A+BC)^2$: $t_{11} \to t_{12}$

We consider the type of mxn matrix is : m×n

Using the rules,

for BC: $t_4 = t_5$, $t_3 = t_7$, $t_6 = t_8$

for A+BC: $t_1 = t_7 = t_9$, $t_2 = t_8 = t_{10}$

for $(A+BC)^2$: $t_9 = t_{10} = t_{11} = t_{12}$

For multiplying B and C, $t_4 = t_5$ from matrix
For A+BC, dimensions of A and BC must be same
For $(A+BC)^2$, (A+BC) must be square matrix

(4) Overload resolution with one Parameter

Let the list of the function prototype be as follows:-

```
void a();          //F1
int b(int);        //F2
void a(int);       //F3
void a(char);      //F4
void c(double);    //F5
void a(int,double); //F6
void b(char);      //F7
void a(double, double=0.0) //F8
void a(char, char*);       //F9
```

we need to resolve:

a(6.6);

Resolution

↳ By name, Candidate functions are:

F1, F3, F4, F6, F8, F9

↳ By number of parameters, viable functions are:

F3, F4, F8

↳ By exact match (type – double), Best viable function:

F8

(5) Parametric Polymorphism

Parametric Polymorphism is a way to make a programming language more ~~expr~~ expressive, while still maintaining full static type-safety. Using parametric polymorphism, a function or a data type can be written generically so that it can handle values identically without depending on their types.

The main characteristic of parametric polymorphism is that the set of types associated with a function or other value is given by a type expression.

Parametric polymorphism may be implicit or explicit.

In explicit parametric polymorphism, the program text contains type variables that determine the way that a function or other value may be treated polymorphically.

Explicit polymorphism often involves explicit instantiation or type application to indicate how type variables are replaced with specific types in the use of a polymorphic value.

e.g: C++ templates

Implicit parametric polymorphism, also known as Haskell polymorphism, deals with programs that declare and use polymorphic functions but do not need to contain types - the type-inference algorithm computes when a function is polymorphic and computes the instantiation of type variables as needed.

Rvalue reference

For any type T, T&& is called an rvalue reference to T.

A rvalue reference is a type, that behaves similarly to the ordinary reference T&, with several exceptions. In case of function overload resolution, rvalue prefers the new rvalue. They must be initialised and cannot be rebound.

Example
```
void func(X &x);  // lvalue reference
void func(X &&x); // rvalue reference overload

X x;
X try();
func(x); // lvalue argument: func(X&)
func(try()); // rvalue argument: func(X&&)
```