Assignment-5

19CS10060

Sunanda Mandal

1. • A <u>type error</u> occurs where a computational entity such as a function or a data value, is used in a manner that is inconsistent with the concept it represents.

• A <u>type system</u> is a ~~tractable~~ syntactic method for proving the absence of a certain program behaviours by classifying pharases according to the kinds of value they compute.

• <u>Advantage of type systems</u>:

(a) <u>Detecting errors</u>: Static type-checking allow early detection of some programming errors.

(b) <u>Abstraction</u>: Enforce disciplined programming.

(c) <u>Documentation</u>: Types are useful whil reading programs

(d) <u>Language safety</u>: A safe laguage protects its own abstraction, portability.

(e) <u>Efficiency</u>: Distinguish between integer-valued arithmatic expression and real valued one; Eliminate many of dynamic checks, etc.

19CS10060 Sunanda Mandal

② • Type Inference is the process of determining the type of expressions based on the known types of some symbols that appear in them.

• Example by Hand-weaving:

let's : $f\ x = 2 + x$

$\gg f :: Int \rightarrow Int$

Type of f:

$+$ has type : $Int \rightarrow Int \rightarrow Int$

2 has type : Int

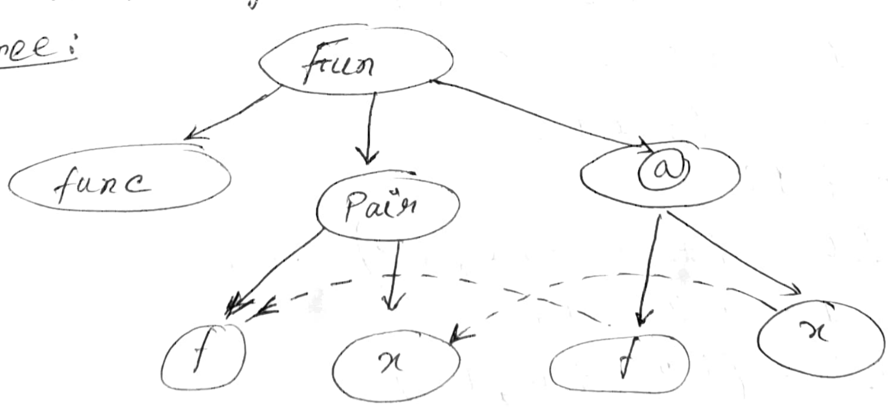
As we are applying $+$ to x we need $x :: Int$

$\Rightarrow f\ x = 2 + x$ has type $Int \rightarrow Int$.

• Type Inference Algo for polymorphic function:

$func\ (f, x) = f\ x$

Parse tree:



Assigning type variables to nodes of the parse tree and add the constraints:

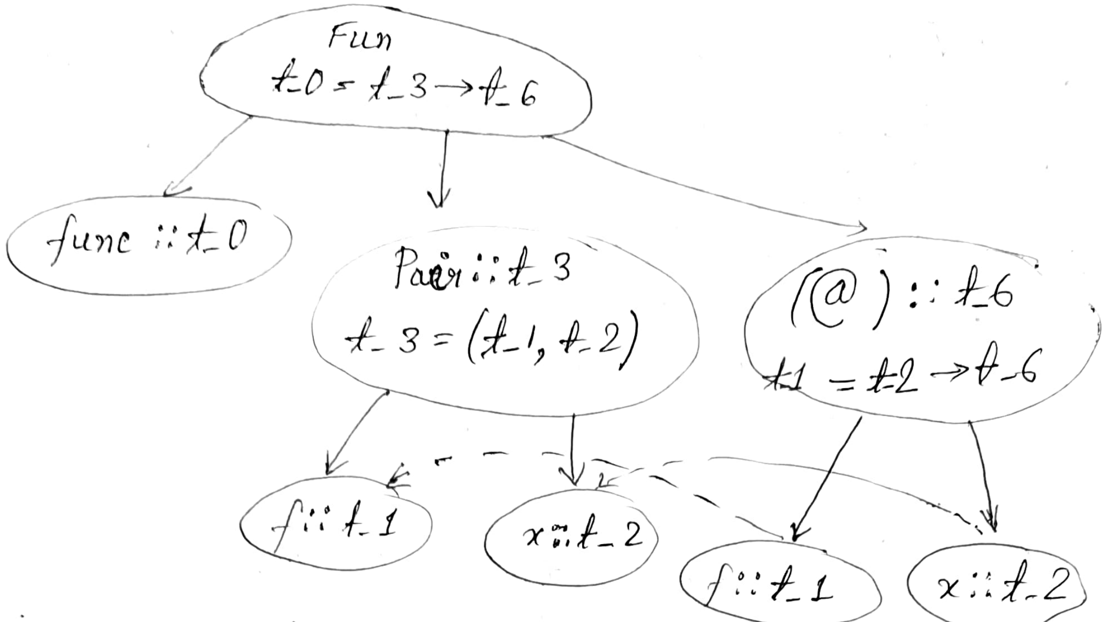
$t\_1 = t\_2 \rightarrow t\_6$

$t\_0 = t\_3 \rightarrow t\_6$

$t\_3 = (t\_1, t\_2)$

19CS10060

Sunanda Mandal



Solving constraints:

$t\_0 = t\_3 \rightarrow t\_6$

$t\_1 = t\_2 \rightarrow t\_6$

$t\_3 = (t\_1, t\_2)$

Replace t_3:

$t\_0 = (t\_1, t\_2) \rightarrow t\_6$

$t\_1 = t\_2 \rightarrow t\_6$

$t\_3 = (t\_1, t\_2)$

Replace t_1:

$t\_0 = (t\_2 \rightarrow t\_6, t\_2) \rightarrow t\_6$

$t\_1 = t\_2 \rightarrow t\_6$

$t\_3 = (t\_2 \rightarrow t\_6, t\_2)$

Replace t_2 with t and t_6 with t1:

$t\_0 = (t \rightarrow t1, t) \rightarrow t1$

$t\_1 = t \rightarrow t1$

$t\_3 = (t \rightarrow t1, t)$

Determining type:

func (f, x) = f x

> func :: (t -> t1, t) -> t1

19CS10060 Sunanda Mandal

(3) • Type Inference Algorithm :

⇒ Assign type to the expression and subexpression.

- for any compound expression or any variable use a type variable
- For known operations or constants, such as + or 3, use the type that is known for this symbol.

⇒ Generate a set of constraints on types, using the parse tree of the expression. These constraints reflect the fact that if a function is applied to an argument, for example, then the type of the argument must equal the type of the domain of the function.

⇒ Solve these constraints by means of unification, it is a substitution-based algorithm for solving systems of equations.

• Framing and Solving Type Constraints (using a Matrix)

Find type of $(AB+C)^2$ given

$A : s \to t$

$B : u \to v$

$C : w \to x$.

Let us consider the type of a $m \times n$ matrix as $m \to n$.

Assigning type variable for the subexpressions of $(AB+C)^2$

$A : s \to t$	$AB : a \to b$
$B : u \to v$	$AB+C : c \to d$
$C : w \to x$	$(AB+C) : e \to f$

Sunanda Mondal 19CS10060

Applying typing rules:

$t = u$, $a = s$, $b = v$ for AB

$a = c = w$, $b = d = x$ for $AB + C$

$c = d = e = f$ for $(AB+C)^2$

Typing rules used are:

Multiplication: $$\frac{E \vdash A : s \to t, \quad E \vdash B : t \to u}{E \vdash AB : s \to u}$$

Addition: $$\frac{E \vdash A : s \to t, \quad E \vdash B : s \to t}{E \vdash A+B : s \to t}$$

Squaring: $$\frac{E \vdash A : s \to s}{E \vdash A^2 : s \to s}$$

Solving the constraints, we get the following two equivalent classes,

$t = u$

$a = s = c = w = d = e = f = b = x = v$

Hence,

$A : a \to t$

$B : t \to a$

$C : a \to a$

is the most general typing

Any value of a, t make the expression $(AB+C)$ well formed.

4. Overload Resolution with One Parameter:

In the context of a list of function prototype:

```
int g(double);              // F1
void f();                   // F2
double h(void);             // F3
void f(int);                // F4
int g(char, int);           // F5
void f(double, double = 3.4);   // F6
void h(int, double);        // F7
void f(char, char*);        // F8
```

The call site to resolve is:

`f(5.6)`

Resolution

- Candidate functions (by name):
  F2, F4, F6, F8
- Viable functions are:
  F4, F6
- Best viable function: F6 (by type double - exact match)

⑤ Parametric Polymorphism:

refers to the ability where a function may be applied to any arguments whose types match a type expression involving type variables. Basically, it opens a way to use the same code for different types.

- In explicit parametric polymorphism, the program text contains type varitables that determine the way that a fuction or other value maybe treated polymorphically.

It often involves explicit instatiation or type application to indicate how type variable are replaced

Sunanda Mandal

19CS10060

with specific types in the use of a polymorphic value. Examples include C++ templates.

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

- rvalue reference:

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

An rvalue reference behaves just like an lvalue references except that it can bind to a temporary (an rvalue), whereas lvalue reference cannot be bound to a rvalue.

It allow programmers to avoid logically unneccesary copying and provide perfect forwarding functions.

E.g.

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

X x;
X foobar();
func(x); // argument is lvalue, calls func(X&)
func(foobar()) // argument is rvalue, calls
                  func(X&&)
```