1/10/24 class - 24 :-)

* type (+)
* foo x y = x + y
  type foo
  foo inherited the type of +
  types are optional, if you don't give anything.

From here have to remember;
you can infer about program just by seeing the program.

* char * → void *
  void * → not compatible to boolean
  Compiler won't throw an error.

(C is weakly typed to Java, Java is strongly typed to C) why.

* dynamic typed → Python.
  gradually typed (Even Java, due to reflection etc, cannot determine type)
* Python support - Duck type
* Scheme also has type checks in runtime (Assembly no type. Some langs that which register specify type.)
* bool / or, and are primitive?
  or & and are not only for bool. In that sense or & add are primitive.

* pg-6 Circle isn't a datatype.
it is a name given to constructor of type Shape.

(we've seen data constructors with arguments)

* pg-7 (Java has option programming)

2 ways of constructing type Maybe.

Maybe very useful for handling failures.

```
V. getval (k key){
    if (this.map.containskey(key))
        return th.map.get(key)
    else
        return NULL
}
```

```
if (getval(..)
    == NULL)
throw ____
print ____
}
```

Intead of throwing an error & exit,
what to do if we just want handling,

(return NULL, return 0 ... hacky ways of what haskel does)

(purpose of "just" is just to convert into type Maybe)

3/10/24 class :- (25)

* Typed lambda calculus:-

$M \rightarrow x$

$\mid \lambda x : T\, M$ — This is the change from usual/untyped lambda calculus

$\mid M_1 M_2$

T → type enviroment (bindings from variables to types)

Inference rule, $\dfrac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2}$ $\dfrac{t_2 \rightarrow t_2'}{v_1 t_2 \rightarrow v_1 t_2'}$ } Possible evaluations $t_i \in$ term of these }

Actually done ⇐

$v_i \in$ Variables.

What is enforced here??

$M \to M_1 M_2$ → first we evaluate operater M_1 & then apply on operand M_2

$t_1 t_2 \longrightarrow t_1' t_2$.

But operater had simplified to a value i.e., $v_1 t_2$

then t_2 has to be simplified $v_1 t_2 \to v_1 t_2'$

i.e., If you are applying a procedure, first simplify procedure.

* If only these 2 rules $t_1 t_2 \to t_1' t_2$; $v_1 t_2 \to v_1 t_2'$ } (understand; left evaluated first)

Withat types,

there is no check whether t_1 simplified into a procedure/Value.

We get ron time error if $M_1 M_2$ applied.

(Since everything is a f^n in lambda calculus, even variables ??)

Understand why we need TYPES

---

* $x : T \in \Gamma$ / $\Gamma \vdash x : T$ (if in envirment Γ, 'looked' up for x, we get type T)

inference (Similiar to logic inference rules)

* consider an expression like

$\Gamma \vdash \lambda \overbrace{x : T_1}^{x \text{ of type } T_1}. t_2 : ?$ → term.

→ To know about type of this f^n, I need to know type of x.

Final rules: $\dfrac{x : T \in \Gamma}{\Gamma \vdash x : T}$, $\dfrac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}$

extending Γ with x is binded with type T_1 } given one fact — 2nd fact

~~Why do you need~~ using these 2 facts we can say type of this fn.

* To determine t_2's type,

type of x is needed (Bcz argument is usually used in body)

* $$\frac{\tau \vdash t_1 : T_1 \to T_2 \quad \tau \vdash t_2 : T_1}{\tau \vdash t_1 t_2 : T_2}$$ (Sir wrote → On board).

* Sound ≡ correct

Type-soundness theorem:-

"Well typed program cannot go wrong".

(Q) what if an unmatched types?

We'll be stuck → then we throw error

Converse of this ⇒ Not well typed → Will go wrong (throw error).

---

* $\tau \vdash$ if t_1 then t_2 else t_3 ?

$$\frac{\tau \vdash t_1 : \text{Bool} \quad \tau \vdash t_2 : T \quad \tau \vdash t_3 : T}{\tau \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3}$$

Where same type returned in then, else branches

if true then 0 else "Hello"

(this doesn't fit into above rule) → (File cannot assign type to this expression, so program is wrong)

Say then branch → T_1

else branch → T_2

What should I do? ($T_1 \cup T_2$) we need union to be defined

We want a universal type (as we have Java.lang.object).

We define

Subtyping :-

$S \leq T$ (Monkey is subtype of Animal)

↑ subtype

(When monkey have more fields than animal,
But why mokey called sub (In some sense smaller)
Bcz all mokeys are animals
But all animals aren't mokeys)

* $\forall S, S \leq U$

↑ universal type (In Java, Java.lang.object).

We maintain U, for unirion to be defined always.

---

* $S \leq S$, $\dfrac{S \leq T \quad T \leq U}{S \leq U}$ } rules.

*

$T_1 \leq S_1 \quad S_2 \leq T_2$

$S_1 \to S_2 \leq T_1 \to T_2$

(Monkey ≤ animal

* If a function expects animal, I can give mokey.
* If a function returns monkey, I can store it in animal)

Today's class:-

We have a fundamental system to represent functional programming.
Extended it into typing & notion of subtyping → both with inheritance etc.

Why rust famous? → Bcz of ownership type.

8/10/24 Lec-27:- Effectful programming in Haskel :-

* IO ( ) → void
  "perform" an IO "action" and "store" nothing.

* :t getLine.
  IO String → perform an IO "action" & store a string.

* "←" extract operator.

(do is a syntactic sugar (in fact more) we'll see what it's for.)

* Scheme isn't pure functional
  But Haskel is pure functional. (So everything is composition (.).)

"do" illustrate
like "C" do line by line.

* (>>) : sequencing operator.

* λ → not possible on keyboard. So haskel have "\" as λ.

* (>>=) sequencing operator → extract the value stored by the previous action.

* Haskel programs has 2 kind of - -
  1. Action (IO is not the only action / only side effect want to perform)
  2. function

>> & >>= } compose actions

"." } compose functions.

do → syntactic sugar of ">>".

disadvantage :-
- Extra names has to be given as input str

have to collect it as a variable & then use it.

* Return is diff from what we've done before. So using word "store"...

* Last statement is type of returned value.

* "Bool" vs "IO Bool" error

So return (c == 'y') . return constructs an action.

takes Bool & make it IO bool.

* Monad is type class (by seeing ⇒)

(+) : Num a ⇒ a → a → a

---

m ≡ IO

(>>) : IO a → IO b → IO b.

(>>=) : IO a → (a → IO b) → IO b

(return) : a → IO a

In basicio2.hs

putStrLn( ) => IO ()
getline => IO String

} return type of (>>) will be IO String.

* The reason why >>= & >> are seperated bcz,
if you want to do math, have to come out of these actions :-

Though >> & >>= are functions, we keep them separately.

* Monads are something in math. But haskel devlopers kept it as type class, so that can be used by usual programmer.

* Int, Nat are instances of type class Num (I think)
operators defined in Num operators has to be overridden.

Similarly IO is an instance of type class Monad.

* class Monad m where
>>= :: ma -> (a -> mb) -> mb
return :: a -> m a

(>> can be derived from >>=)

Instance Monad IO where
>>= :: ——
return :: ——

---

* instance Monad Maybe where
return x = Just x
Nothing >>= f = Nothing
Just x >>= f = fx

(doing pattern matching ?? "f" ?)

* case lookup x a map of
  Nothing → Nothing
  Just y → case lookup y b map of
    Nothing → Nothing
    Just z → lookup z c map

* If Maybe is a monad class, then we can write as

do
  y ← lookup x a map
  z ← lookup y b map
  return (lookup z c map)

Because of these, the error handling is hidden inside
(will not be case of Java).

do is syntax sugar of

return (lookup x a map >>= lookup y b map >>= lookup z c map)

* main is a key word in Haskel.
* you can "create" an IO action in fn
  not "perform" but can force it
* Side effects not there in 7-8 yrs ... → later some monad in math.
  call cout from C program. → (before that only for math)
  List, Maybe, IO ... are monads
* Newer version of Haskel.
  Monadic style of programming (even without knowing monads).