COL765 : ILFP

Principal Typing Schemes

Solution to $\Gamma \vdash e : \tau \rhd_T C$ is

$\langle S, \sigma \rangle$

S : Substitution policy s.t. $S(\tau) = \sigma$

Once the type constraints are generated, solution to the constraints is obtained through an algorithm called Unification.

Unification

- Whenever two Types A & B are required to be "the same", the compiler must "unify" what it knows about A & B to produce a potentially more detailed description of their common type.

Eg: E1 : ('a * int)

E2 : (String * 'b) then

in the Exp. ITE <X, E_1, E_2>

the compiler can infer that

'a = String ^

'b = int

Unification's uses

- Type inference
- Pattern matching

Defn & Prop. of Unifiers

Unification is a process of finding a substitution S s.t. $aS = bS$

aS and bS: result of applying S to type exprs a & b.

Eg:

$E_1 = f \quad x \quad (g \; y)$

$E_2 = f (g \; z) \quad \omega$

then $S = \{ x \mapsto (g \; z), \quad \omega \mapsto (g \; y) \}$

$E_1 S = f \; (g \; z) \; (g \; y)$

$E_2 S = f \; (g \; z) \; (g \; y)$

Note:

- Unifiers donot necessary exist

  Eg: x & $f\,x$ can't be unified

- Neither is the case that unifiers are unique

  Eg: $T = \{ x \mapsto g\,(f\,a\,b),$
  $y \mapsto f\,b\,a,$
  $z \mapsto f\,a\,b,$
  $w \mapsto g\,(f\,b\,a) \}$

Applying T to E_1 & E_2 from the previous Example, we get

$E_1T = E_2T = f\,(g\,(f\,a\,b))\,(g\,(f\,b\,a))$

T is not the "most general unifier"

- Whenever a unifier exists ⇒ there is a mgu (most general unifier)
- Def^n of mgu: A unifier S for a & b is mgu if:
  - Any other unifier T for a & b can be obtained from S by doing further substitutions.

Eg: Consider: S, T, E_1, E_2 from previous examples

Consider $U = \{ z \longmapsto f\ a\ b,\ \ y \longmapsto f\ b\ a \}$

We find that

$T = SU$ s.t.

$E_1 SU = E_2 SU = T$

$E_1 SU = f\ x\ (g\ y)\ SU = f\ (g\ z)\ (g\ y)\ U$

$= f\ (g\ (f\ a\ b))\ (g\ (f\ b\ a))$

$= E_1 T$

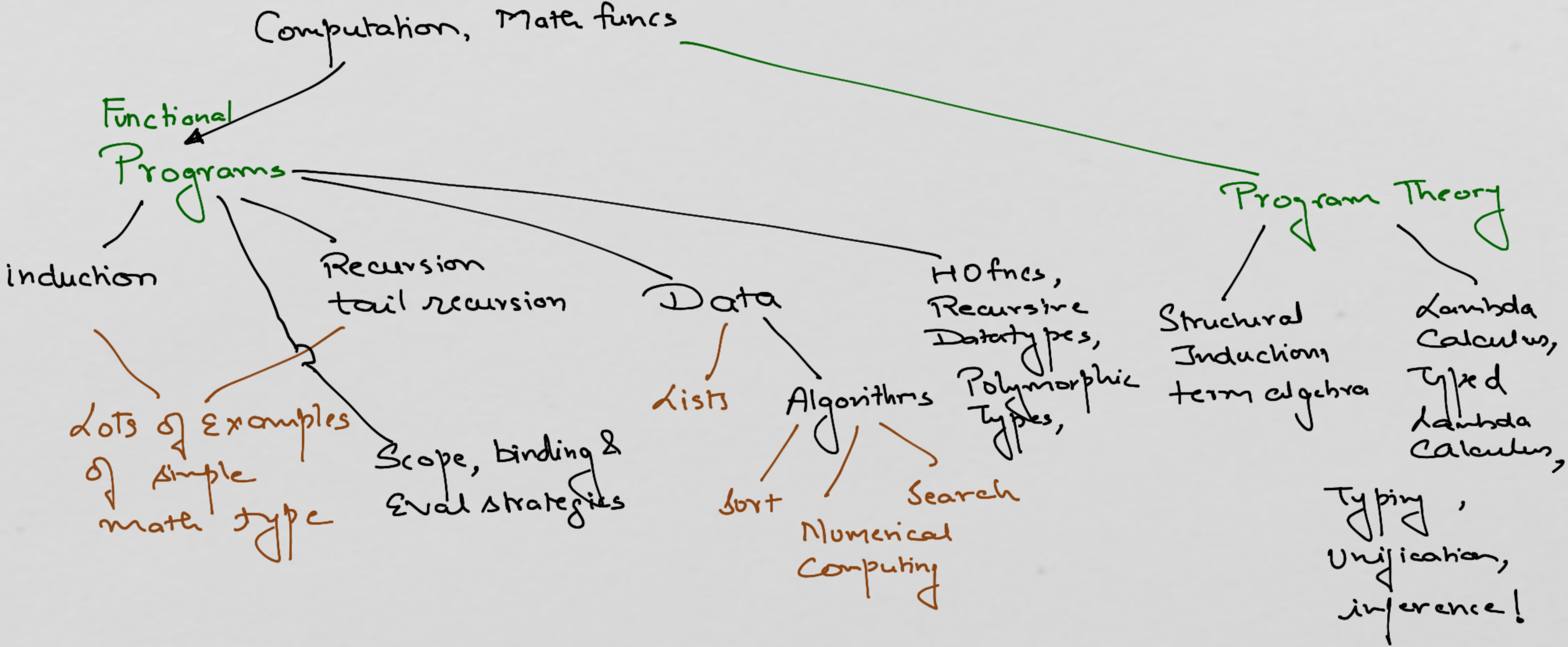
Unification Alg for a not just a pair of terms
but also for a set of pair of terms

```
type  id = String
data  term = Var id | Term (id * Term list)

type  Substitution = (id * term) list

(* check if var x occurs in term t *)
fun occurs (x::id)(t::term):: bool =
    case t of
      Var y  ⟶  (x = y)
    | Term(_, s) ⟶ (occurs x s)
```

```
(* apply a substitution right to left *)
fun apply (s:: substitution) (t::term) :: term =
    foldr (\x u ⇒ subst u x) s t

fun subst (s::term) (t::term) :: term =
  (* subst. term s for all occurrences of x in t*)
    case t of
      Var y → if x=y then s else t
    | Term (i,u) → Term (i, map (subst s x) u)
```

```
fun unify_one (s::term) (t::term):: substitution =
    case (s, t) of
      (Var x, Var y) ⟶ if x=y then [] else [(x, t)]

      Term (f, sl), Term (g, tl)
          ⟶ if f==g && length sl == length tl
            then unify (combine sl tl)
(* Combine :: [a] ⟶ [b] ⟶ [(a,b)] *)
            else error "head symbol conflict"

      Var x, (Term (_,_) as t),
      Term (_,_) as t, Var x ⟶
            if occurs x t
            then error "not unifiable"
            else [(x, t)]
```

```
fun unify (s :: [(term * term)]) :: substitution =
    case s of
      [] ⟶ []
    | (x, y) :: sl ⟶ let t2 = unify sl in
                     let t1 = unify-one
                         (apply t2 x)
                         (apply t2 y) in
                     t1 ++ t2
```


Computation, Math funcs
Functional
Programs
Program Theory
Induction
Recursion
tail recursion
Data
HOfncs,
Recursive
Datatypes,
Polymorphic
Types,
Structural
Induction,
term algebra
Lambda
Calculus,
Typed
lambda
Calculus,
Typing,
Unification,
inference!
Lots of Examples
of simple
math type
Scope, binding &
Eval strategies
Lists
Algorithms
Sort
Numerical
Computing
Search

Why f.P. & PLT

- Great features : Elegance, Crisp & well defined semantics
  HO fncs, Lazy evaluation, Conciseness, forces you
  to think on Essentials of computation
- Big data → Conceptually all functional
- Native concurrency
- PLT – Math + S.E. + Linguistics + CogSc.
  [design, implm., analysis & characterisation
  of formal languages]