# FP4

December 17, 2024

**Introduction**

Ce cours donne une introduction à la "théorie de la démonstration/preuve" en commançant par les logiques des propositions et des prédicats.

Il propose aussi le code d'un démonstrateur de théorème similaire à CoqIDE utilisé plus tard.

On rappel qu'un *langage* (logique ou autre) peut être représenté par un arbre syntaxique et aussi un *type*

## 1 Des types aux propositions

**Questions**

1. Donner le type des fonctions suivantes ?
2. Est-ce qu'il existe une forme "d'équivalence" ? En quoi y a t'il une différence ?
3. Comment représenter sous forme mathématique que:

   `SI ('il fait beau' ET (SI 'il fait beau' ALORS 'la lumière est éteinte'))`
   `ALORS 'la lumière est éteinte'` ?

[1]:
```
let f1 (x,f) = f x
let f2  x f  = f x
```

*Remarque.*

Il existe un opérateur spécifique `(x |> f) = (f x)` et correspondant à la fonction `f2` ci-dessus.

4. Comment spécifier le langage des Types/Propositions ?

[2]:
```
type Prop =
  | Var of string
  | Imp of Prop*Prop
  | And of Prop*Prop

let a,b,c,d = Var "A", Var "B", Var "C", Var "D"
let (=>) p1 p2 = Imp (p1,p2)
let (*)  p1 p2 = And (p1,p2)

let rec str = function
  | Var v       -> v
```

```
  | Imp (p1,p2) -> "("+(str p1)+" => "+(str p2)+")"
  | And (p1,p2) -> "("+(str p1)+" * "+(str p2)+")"

let p1 = (a * (a => b)) => b
let p2 = a => ((a => b) => b)

printf "%s" (str p1)
```

```
((A * (A => B)) => B)
```

5. Rappeler le type `Dictionnaire` et les fonctions `lookup/add`.

[3]:
```
type Dict<'k,'v> = list<'k*'v>

let rec has k = function
  | []        -> false
  | (k2,v)::d -> match k=k2 with
                 | true -> true
                 | _    -> has k d

let rec lookup k = function
  | []        -> failwith "key not found !"
  | (k2,v)::d -> match k=k2 with
                 | true -> v
                 | _    -> lookup k d

let rec add k v = function
  | []         -> [(k,v)]
  | (k2,v2)::d -> match k=k2 with
                  | true -> (k2,v)::d
                  | _    -> (k2,v2)::(add k v d)
```

6. Comment définir la notion de “Séquent” (où “Etape de démonstration”) défini par un ensemble de propositions/hypothèses et une (plusieurs) propositions à démontrer ?

[4]:
```
type Sequent = Dict<string,Prop>*list<Prop>
let proven ((_,ps)) = (ps=[])

let s : Sequent = ([],[p1])
```

7. Comment définir les règles de “déduction naturelle” pour:
   a) introduire une hypothèse, et
   b) appliquer une hypothèse ?

[14]:
```
let proof p = ([],[p])

let show (d,ps) =
```

```
  let c = (List.fold (+) "" (List.map (fun (k,v)->k+" : "+(str v)+"\n")␣
 ↪d))+"---\n"
  let p = string (List.map str ps)
  c+p

let intro n s =
  match s with
  | (d,(Imp (p1,p2))::ps) -> (add n p1 d,p2::ps)
  | _                     -> s

let apply n (d,ps) =
  let q = lookup n d
  match q,ps with
  | x,p::ps          when p=q -> (d,ps)
  | Imp (a,b),p::ps when b=p -> (d,a::ps)
  | _,_                      -> (d,ps)

(proof p2)
|> show
|> printfn "%s"

(proof p2)
|> intro "x"
|> intro "f"
|> apply "f"
|> apply "x"
|> proven
|> printfn "%A"
```

```
---
[(A => ((A => B) => B))]
true
```

*Exercises.* Prouver les propriétés suivantes:

[15]:
```
let p3 = a =>a
let p4 = (a =>b)=>((b=>c)=>(a=>c))

printfn "%s" (str p3)
printfn "%s" (str p4)
```

```
(A => A)
((A => B) => ((B => C) => (A => C)))
```

8. Pour la conjonction, on ajoute deux nouvelles règles spécifiques:

[16]:
```
#nowarn "25"

let split s =
```

```
  match s with
  | (ds,(And (p1,p2))::ps) -> (ds,p1::p2::ps)
  | _                      -> s

let destruct h h1 h2 s =
  match s with
  | (ds,ps) -> match (lookup h ds) with
               | And (p1,p2) -> (add h1 p1 (add h2 p2 ds),ps)
               | _           -> s
  | _       -> s
```

```
input.fsx (13,5)-(13,17) typecheck warning This rule will never be matched
```

*Exercice.* Prouver les propriétés suivantes:

[20]:
```
let p5 = a => (b => (a*b))
let p6 = ((a=>b)*(c=>d)) => ((a*c)=>(b*d))

(proof p5)
|> show
|> printfn "%s"
```

```
---
[(A => (B => (A * B)))]
```

## 2 Passage aux Prédicats

Comment représenter la formule suivante ?

```
(TOUT Homme est Mortel) ET (Socrate est un Homme) DONC (Socrate est Mortel)
```

[21]:
```
let f1  = ["Socrate";"est";"Homme"]
let f2a = ["?x";"est";"Homme"]
let f2b = ["?x";"est";"Mortel"]

let var (s:string) = (s[0]='?')
```

On reprend alors la même démarche que la partie précédente et les propositions.

[22]:
```
type Pred =
  | Fact of list<string>
  | Then of Pred*Pred

let (>>) p1 p2 = Then (p1,p2)

let p7 = Then (Fact ["Socrate";"est";"Homme"],Then (Then (Fact f2a,Fact␣
 ↪f2b),Fact ["Socrate";"est";"Mortel"]))
```

```
let rec shw = function
  | Fact ss      -> List.reduce (fun x y->x+" "+y) ss
  | Then (p1,p2) -> "("+(shw p1)+" => "+(shw p2)+")"

p7
|> shw
|> printfn "%s"
```

```
(Socrate est Homme => ((?x est Homme => ?x est Mortel) => Socrate est Mortel))
```

Avec des nouvelles règles de déduction/raisonnement.

**unification & substitutions**

[23]:
```
#nowarn "25"

let unify0 s1 s2 d =
  match (var s1) with
  | false -> (s1=s2,d)
  | _     -> match (has s1 d) with
             | true -> (s2=(lookup s1 d),d)
             | _    -> (true,add s1 s2 d)

let unify [x1;x2;x3] [y1;y2;y3] d1 =
  match (unify0 x1 y1 d1) with
  | (true,d2) -> match (unify0 x2 y2 d2) with
                 | (true,d3) -> unify0 x3 y3 d3
                 | _         -> (false,d1)
  | _         -> (false,d1)

unify ["?x";"est";"Homme"] ["Socrate";"est";"Homme"] []
|> printfn "%A"

let susbtitute xs d = List.map (fun x->if (var x) then lookup x d else x) xs

let (_,d) = unify ["?x";"est";"Homme"] ["Socrate";"est";"Homme"] []

susbtitute ["?x";"est";"Mortel"] d
|> printfn "%A"
```

```
(true, [("?x", "Socrate")])
["Socrate"; "est"; "Mortel"]
```

[31]:
```
type Seq = Dict<string,Pred>*list<Pred>

let rintro n s =
  match s with
  | (d,(Then (p1,p2))::ps) -> (add n p1 d,p2::ps)
```

```
  | _                        -> s

let rapply h (d,p::ps) =
  let q = lookup h d
  match q,p with
  | Fact f1, Fact f2                -> if (f1=f2) then (d,ps) else (d,p::ps)
  | Then (Fact f1,Fact f2),Fact f3 -> match (unify f2 f3 []) with
                                       | (true,d2) -> (d,(Fact (susbtitute f1␣
 ↪d2))::ps)
                                       | _         -> (d,p::ps)
  | _                               -> (d,p::ps)

let display (d,ps) =
  d
  |> List.map (fun (k,v)->k+": "+(shw v)+"\n")
  |> List.fold (+) ""
  |> fun x->x+"-----------------\n"
  |> fun x-> x+(if (ps=[]) then "Qed" else (shw (List.head ps)))

([],[p7])
|> rintro "x"
|> rintro "f"
|> rapply "f"
|> rapply "x"
|> display
|> printfn "%s"
```

```
x: Socrate est Homme
f: (?x est Homme => ?x est Mortel)
-----------------
Qed
```

Les mêmes éléments peuvent être utilisées sur des *relations* !

[39]:
```
let f1 =  Fact ["A";"related";"B"]
let f2 = (Fact ["?x";"related";"?y"]) >> (Fact ["?y";"related";"?x"])

([],[f1 >> (f2 >> (Fact ["B";"related";"A"]))])
|> rintro "x"
|> rintro "f"
|> rapply "f"
|> rapply "x"
|> display
|> printfn "%s"
```

```
x: A related B
f: (?x related ?y => ?y related ?x)
-----------------
Qed
```

Une relation particulière est *l'galit* utilisée dans la définition des programmes ; ce qui sera la base des “preuves de programmes” développées dans le prochain cours.