Induction sur les arbres

Planning

- Motivations
- Comment définir les arbres ?
- Équations récursives sur les arbres
- Complexité de fonctions sur les arbres
  - Recherche dans un arbre binaire de recherche
  - Recherche dans un arbre 2-3-4
- Preuve de propriétés par récurrence sur les arbres

Les arbres en informatique

- Organiser des données en une structure hiérarchique (les fichiers dans un système d'exploitation, le sommaire d'un livre, les expressions arithmétiques, les phrases du langage naturelle, les arbres de jeu)
- Rechercher/accéder à l'information plus facilement
- Modéliser de nombreux problèmes

Quelques types d'arbres

Arbre binaire (I) : tout nœud interne a exactement deux fils.

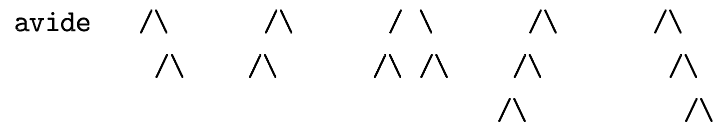
Arbre binaire (II) : tout nœud interne a au plus deux fils.

Arbre n-aire (I) : tout nœud interne a exactement n fils.

Arbre n-aire (II) : tout nœud interne a au plus n fils.

Arbre quelconque : le nombre de fils n'est pas borné.

Les arbres binaires de type I sans étiquette



L'ensemble *AbinSE* est le plus petit ensemble t.q.

- $avide \in AbinSE$
- Si $a_1, a_2 \in AbinSE$, alors $noeud(a_1, a_2) \in AbinSE$.

Un type abstrait AbinSE

Domaine :

`AbinSE`

Opérations de construction :

`avide` : `AbinSE`

`noeud` : `AbinSE` × `AbinSE` → `AbinSE`

Opérations de test :

`est_arbre_vide` : `AbinSE` → `booléen`

Opérations d'accès :

`fils_gauche` : `AbinSE` → `AbinSE`

`fils_droit` : `AbinSE` → `AbinSE`

AbinSE en OCAML

```
# type abinse = Av | N of abinse * abinse;;
type abinse = Av | N of abinse * abinse

(* opération de construction *)
# let cons_ab(x,y) = N(x,y);;
val cons_ab : abinse * abinse -> abinse = <fun>

(* opérations d'accès *)
# let fils_gauche a = match a with
      N(x,y) -> x
    | _      -> failwith "erreur";;
val fils_gauche : abinse -> abinse = <fun>
```



```
# let fils_droite a = match a with
      N(x,y) -> y
    | _      -> failwith "erreur";;
val fils_droite : abinse -> abinse = <fun>

(* opération de test *)
# let est_arbre_vide a = match a with
  Av -> true | _ -> false;;
val est_arbre_vide : abinse -> bool = <fun>

# let arb1 = N(N(Av,Av),Av);;
val arb1 : abinse = N (N (Av, Av), Av)
```

```
# est_arbre_vide arb1;;
- : bool = false

# arbre_gauche arb1 ;;
- : abinse = N (Av, Av)

# arbre_droite arb1 ;;
- : abinse = Av
```


Équations récursives pour AbinSE

Schéma récursif :

$$\texttt{f}(a) = g(a, \texttt{f}(\texttt{fils_gauche}(a)), \texttt{f}(\texttt{fils_droit}(a)))$$

Exemple : nombre de feuilles

Problème : définir une fonction qui calcule le nombre de feuilles d'un arbre binaire de type I.

Déclaration du type :

$$\texttt{nb_f} : \texttt{AbinSE} \to \texttt{entier}$$

Équation récursive :

$$\texttt{nb_f}(a) = g(a, \texttt{nb_f}(\texttt{fils_gauche}(a)), \texttt{nb_f}(\texttt{fils_droit}(a)))$$

Cas particulier :

a=avide

On pose

$$g(a, z_1, z_2) = z_1 + z_2$$

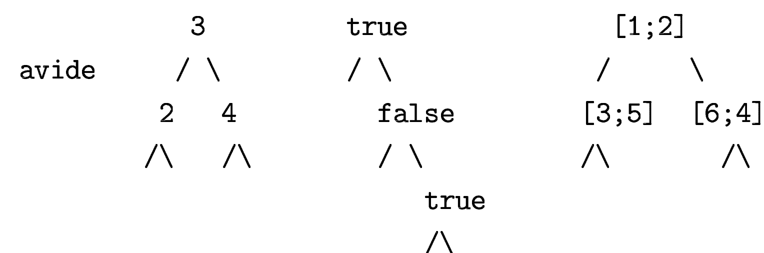
Ceci donne :

$$\begin{array}{lcl} \texttt{nb_f}(a) & = & 1 \\ & & \text{si } \texttt{est_arbre_vide}(a) \\ \texttt{nb_f}(a) & = & \texttt{nb_f}(\texttt{fils_gauche}(a)) + \texttt{nb_f}(\texttt{fils_droit}(a)) \\ & & \text{sinon} \end{array}$$

Le nombre de feuilles en OCAML

```
# let rec nb_feuilles a = match a with
      Av       -> 1
    | N(x,y) -> nb_feuilles x + nb_feuilles y;;
val nb_feuilles : abinse -> int = <fun>
```


Les arbres binaires de type I avec étiquette de type 'a



L'ensemble `'a abin` est le plus petit ensemble t.q.

- $avide \in$ `'a abin`
- Si $a_1, a_2 \in$ `'a abin` et m est de type `'a`, alors $abr(m, a_1, a_2) \in$ `'a abin`.

Un type abstrait pour 'a abin

Domaine :

`'a abin`

Opérations de construction :

avide : `'a abin`

abr : `'a` × `'a abin` × `'a abin` → `'a abin`

Opérations de test :

est_arbre_vide : `'a abin` → booléen

Opérations d'accès :

etiquette : `'a abin` → `'a`

fils_gauche : `'a abin` → `'a abin`

fils_droit : `'a abin` → `'a abin`

'a abin en OCAML

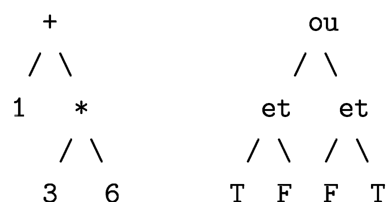
```
# type 'a abin = Av | Arb of 'a * 'a abin * 'a abin;;
type 'a abin = Av | Arb of 'a * 'a abin * 'a abin

(* opération de construction *)
# let cons_ab(e,x,y) = Arb(e,x,y);;
val cons_ab : 'a * 'a abin * 'a abin -> 'a abin = <fun>

(* opérations d'accès *)
# let etiquette a = match a with
      Arb(e,x,y) -> e
    | _       -> failwith "erreur";;
val etiquette : 'a abin -> 'a = <fun>
```

```
# let fils_gauche a = match a with
      Arb(e,x,y) -> x
    | _       -> failwith "erreur";;
val fils_gauche : 'a abin -> 'a abin = <fun>

# let fils_droite a = match a with
      Arb(e,x,y) -> y
    | _       -> failwith "erreur";;
val fils_droite : 'a abin -> 'a abin = <fun>

(* opération de test *)
# let est_arbre_vide a = match a with
  Av -> true | _ -> false;;
```



```
val est_arbre_vide : 'a abin -> bool = <fun>

# let arb1 = Arb(4,Arb(2,Av,Av),Av);;
val arb1 : int abin = Arb (4, Arb (2, Av, Av), Av)

# let arb2 = Arb("bonjour",Arb("madame",Av,Av),Av);;
val arb2 : string abin = Arb ("bonjour", Arb ("madame", Av,

# etiquette arb1;;
- : int = 4

# etiquette arb2;;
- : string = "bonjour"
```


Les arbres binaires de type II avec étiquette



- On les utilise par exemple pour représenter des expressions arithmétiques.
- Pas de notion d'arbre vide : l'expression "la plus petite" est une étiquette.
- Chaque nœud est unaire ou binaire.

Les arbres binaires de type II avec étiquette

C'est un ensemble parametré par deux types `'a` et `'b`, où chaque étiquette d'une feuille est de type `'a` et chaque étiquette d'un nœud interne est de type `'b`.

L'ensemble $('a,'b)$ `AbinII` est le plus petit ensemble t.q.

- Si m est de type `'a`, alors $feuille(m) \in ('a,'b)$ `AbinII`
- Si $a_1, a_2 \in ('a,'b)$ `AbinII` et o est de type `'b`, alors $noeud(o, a_1, a_2) \in ('a,'b)$ `AbinII`.

Un type abstrait pour ('a, 'b) abinII

Domaine :

('a, 'b) abinII

Opérations de construction :

cons_f : 'a → ('a, 'b) abinII

cons_n : 'b × ('a, 'b) abinII × ('a, 'b) abinII → ('a, 'b) abinII

Opérations de test :

est_feuille : ('a, 'b) abinII → booléen

Opérations d'accès :

etiquette_feuille : ('a, 'b) abinII → 'a

etiquette_noeud : ('a, 'b) abinII → 'b

fils_gauche, fils_droit : ('a, 'b) abinII → ('a, 'b) abinII

('a, 'b) abinII OCAML

```
# type ('a, 'b) abinII =
  F of 'a | N of 'b * ('a, 'b) abinII * ('a, 'b) abinII;;

(* opération de construction *)
# let cons_feuille(e) = F(e);;
val cons_feuille : 'a -> ('a, 'b) abinII = <fun>

# let cons_noeud(c,x,y) = N(c,x,y);;
val cons_noeud :
'b * ('a, 'b) abinII * ('a, 'b) abinII -> ('a, 'b) abinII =
```



```
(* opérations d'accès *)
# let eti_feuille a = match a with
       F(e) -> e
    | _     -> failwith "erreur";;
val eti_feuille : ('a, 'b) abinII -> 'a = <fun>

# let eti_noeud a = match a with
       N(e,x,y) -> e
    | _         -> failwith "erreur";;
val eti_noeud : ('a, 'b) abinII -> 'b = <fun>

# let fils_gauche a = match a with
      N(e,x,y) -> x
    | _        -> failwith "erreur";;
```



```
val fils_gauche : ('a, 'b) abinII -> ('a, 'b) abinII = <fun>

# let fils_droite a = match a with
      N(e,x,y) -> y
    | _        -> failwith "erreur";;
val fils_droite : ('a, 'b) abinII -> ('a, 'b) abinII = <fun>

(* opération de test *)
# let est_feuille a = match a with
  F(_) -> true | _ -> false;;
val est_feuille : ('a, 'b) abinII -> bool = <fun>

# type op_arith = Plus | Mul | Sous | Div ;;
type op_arith = Plus | Mul | Sous | Div
```

```
# let arb1 = N(Mul,F(4),F(5));;
val arb1 : (int, op_arith) abinII = N (Mul, F 4, F 5)

# let arb2 = N(Plus,F(4.4),F(5.3));;
val arb2 : (float, op_arith) abinII = N (Plus, F 4.4, F 5.3)
```


Les arbres n-aires

On généralise sans problèmes les arbres binaires de type I ou II au cas n-aire.

Les arbres quelconques

```
avide      3
          / \
             5
           / | \
          1  7
          |  |\
             6  8
             |  |
```


Les arbres quelconques : définition

Un arbre quelconque est
– soit un arbre vide
– soit un nœud interne ayant un nombre arbitraire de fils

L'ensemble `'a arbre` est le plus petit ensemble t.q.
– $avide \in$ `'a arbre`
– Si $m \in$ `'a` et $l \in$ `('a arbre) liste`,
  alors $noeud(m, l) \in$ `'a arbre`.

Un exemple

```
   3            noeud(3,
  / \                 [avide,
    5                  noeud(5,
   / | \                     [noeud(1,[]),
  1   7                       noeud(7,
  |   |\                            [noeud(6,[]),
    6   8                            noeud(8,[])])
    |   |                         avide])])
```


Type abstrait pour un arbre quelconque

Domaine :

`'a arbre`

Opérations de construction :

`avide` : `'a arbre`

`cons_arbre` : `'a` × `('a arbre) liste` → `'a arbre`

Opérations de test :

`est_arbre_vide` : `'a arbre` → `booléen`

Opérations d'accès :

`racine` : `'a arbre` → `'a`

`fils` : `'a arbre` → `('a arbre) liste`

En OCAML

```
# type 'a arbre =  Av | N of 'a * 'a arbre list ;;
type 'a arbre = Av | N of 'a * 'a arbre list

(* opération de construction *)
# let cons_arbre_vide = Av;;
val cons_arbre_vide : 'a arbre = Av

# let cons_noeud(e,l) = N(e,l);;
val cons_noeud: 'a * 'a arbre list -> 'a arbre = <fun>

(* opération de test *)
# let est_arbre_vide a = match a with
```



```
 Av -> true | _ -> false;;
val est_arbre_vide : 'a arbre -> bool = <fun>

(* opérations d'accès *)
# let racine a = match a with
      N(e,l) -> e
    | _      -> failwith "erreur";;
val racine : 'a arbre -> 'a = <fun>

# let fils a = match a with
      N(e,l) -> l
    | _      -> failwith "erreur";;
val fils : 'a arbre -> 'a arbre list = <fun>
```

Équations récursives sur les arbres quelconques

Schéma récursif :

$$\texttt{f}(a) = g_1(a, \texttt{f_F}(\texttt{fils}(a)))$$
$$\texttt{f_F}(s) = g_2(s, \texttt{f}(\texttt{premier}(s)), \texttt{f_F}(\texttt{reste}(s)))$$

Le domaine de `f` est `'a arbre`

Le domaine de `f_F` est `('a arbre) liste`

Cas particuliers :

$a = \texttt{avide}$ ($\texttt{fils}(\texttt{avide})$ n'est pas défini).

$s = \texttt{nil}$ ($\texttt{premier}(\texttt{nil})$ et $\texttt{reste}(\texttt{nil})$ ne sont pas définis).

Exemple I : la taille d'un arbre quelconque

Problème : définir une fonction qui calcule le nombre de nœuds internes d'un arbre quelconque.

Déclaration du type :

`taille` : `'a arbre` → `entier`

`taille_F` : `('a arbre) liste` → `entier`

Équation récursive :

$$\texttt{taille}(a) = g_1(a, \texttt{taille_F}(\texttt{fils}(a)))$$
$$\texttt{taille_F}(s) = g_2(s, \texttt{taille}(\texttt{premier}(s)), \texttt{taille_F}(\texttt{reste}(s)))$$

On pose

$$g_1(a, z) = 1 + z \qquad g_2(a, z_1, z_2) = z_1 + z_2$$

Ceci donne :

$$\texttt{taille}(a) = 0 \quad \text{si } \texttt{est_arbre_vide}(a)$$
$$\texttt{taille}(a) = 1 + \texttt{taille_F}(\texttt{fils}(a)) \quad \text{sinon}$$

$$\texttt{taille_F}(s) = 0 \quad \text{si } \texttt{liste_vide}(s)$$
$$\texttt{taille_F}(s) = \texttt{taille}(\texttt{premier}(s)) + \texttt{taille_F}(\texttt{reste}(s)) \quad \text{sinon}$$

La taille d'un arbre en OCAML

```
# let rec taille a = match a with
      Av     -> 0
    | N(_,l) -> 1 + taille_f(l)
  and
  taille_f l = match l with
      []    -> 0
    | p::r -> taille p + taille_f r;;
val taille : 'a arbre -> int = <fun>
val taille_f : 'a arbre list -> int = <fun>
```


Exemple II : les étiquettes d'un arbre quelconque

Problème : définir une fonction qui calcule l'ensemble des valeurs associées aux nœuds d'un arbre quelconque.

Déclaration du type :

$$\texttt{ensval} : \texttt{'a arbre} \rightarrow \texttt{'a ens}$$

$$\texttt{ensval_F} : \texttt{('a arbre) liste} \rightarrow \texttt{'a ens}$$

Équation récursive :

$$\texttt{ensval}(a) = g_1(a, \texttt{ensval_F}(\texttt{fils}(a)))$$

$$\texttt{ensval_F}(s) = g_2(s, \texttt{ensval}(\texttt{premier}(s)), \texttt{ensval_F}(\texttt{reste}(s)))$$

On pose :

$$g_1(a, z) = \{\texttt{racine}(a)\} \cup z \qquad g_2(a, z_1, z_2) = z_1 \cup z_2$$

Ceci donne :

$$\texttt{ensval}(a) = \emptyset$$

si $\texttt{est_arbre_vide}(a)$

$$\texttt{ensval}(a) = \{\texttt{racine}(a)\} \cup \texttt{ensval_F}(\texttt{fils}(a))$$

sinon

$$\texttt{ensval_F}(s) = \emptyset$$

si $\texttt{liste_vide}(s)$

$$\texttt{ensval_F}(s) = \texttt{ensval}(\texttt{premier}(s)) \cup \texttt{ensval_F}(\texttt{reste}(s))$$

sinon

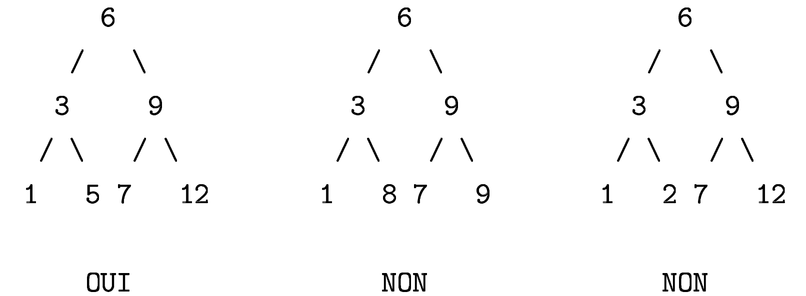
Exercice : implanter la fonction ensval en OCAML.

Les arbres binaires de recherche (ABR)

Un arbre binaire de recherche est un arbre binaire de type I ayant des étiquettes distinctes de type `entier` t.q.

- Le sous-arbre gauche de *tout* nœud `n` ne contient que des entiers inférieurs ou égaux à `n` ;
- Le sous-arbre droit de *tout* nœud `n` ne contient que des entiers strictement supérieurs à `n` ;

Quelques exemples



```
        6                  6                  6
      /   \              /   \              /   \
     3     9            3     9            3     9
    / \   / \          / \   / \          / \   / \
   1   5 7   12       1   8 7   9        1   2 7   12

      OUI                 NON                NON
```


Recherche d'un élément dans un ABR

Pour rechercher un élément dans un ABR a on doit :

1. Comparer avec la racine de a,
2. Rechercher dans le fils droit ou gauche de a.

```
# let rec recherche(e,a) = match a with
      Av           -> false
    | Arb(r,fg,fd) -> if e=r then true
                       else if e<r then recherche(e,fg)
                            else recherche(e,fd);;
val recherche : 'a * 'a abin -> bool = <fun>
```


Complexité de la recherche dans un ABR

Considérons un ABR avec n éléments.

La complexité dans le meilleur des cas est donc $O(1)$.

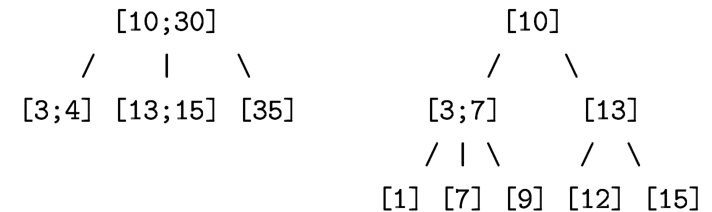
La complexité au pire est donc en $O(n)$.

Les arbres 2-3-4

Un *arbre 2.3.4* est un arbre 4-aire étiqueté de type II t.q.

1. chaque nœud est étiqueté par une liste de $1 \leq k \leq 3$ d'éléments $[x_1, \ldots, x_k]$ t.q. $x_1 \leq \ldots \leq x_k$
2. un nœud étiqueté par $[x_1, \ldots, x_k]$ contient $k+1$ fils $f_1, \ldots, f_{k+1}$ t.q
   - tous les éléments de f_i sont inférieurs ou égaux à x_i (pour $1 \leq i \leq k$)
   - tous les éléments de f_i sont strictement supérieurs à x_{i-1} (pour $2 \leq i \leq k+1$)
3. toutes les feuilles sont au même niveau

Quelques exemples



Recherche d'un élément dans un arbre 2.3.4

Pour rechercher un élélement dans un arbre 2.3.4 a on doit :

1. Comparer avec les éléments de la racine de a,
2. Rechercher dans le "bon" fils selon la première comparaison.

Exercice : implanter la fonction `recherche` en OCAML.

Complexité de la recherche dans un arbre 2.3.4

Considérons un arbre 2.3.4 a avec n éléments et hauteur `h(a)`.

- Une borne inférieur : si l'arbre 2.3.4 a contient uniquement un élément dans chaque noeud, alors $2^{\mathtt{h(a)}} - 1 \leq n$
- Une borne supérieure : si l'arbre 2.3.4 a contient trois éléments dans chaque noeud, alors $n \leq 4^{\mathtt{h(a)}} - 1$

$$2^{\mathtt{h(a)}} - 1 \leq n \leq 4^{\mathtt{h(a)}} - 1$$

$$2^{\mathtt{h(a)}} \leq n + 1 \leq 4^{\mathtt{h(a)}}$$

$$log_4(n+1) \leq \mathtt{h(a)} \leq log_2(n+1)$$

La complexité dans le meilleur des cas est donc $O(1)$.

La complexité au pire est donc en $O(log(n+1))$.

Preuve de propriétés par récurrence sur les arbres

- On utilise la définition inductive de l'ensemble d'arbres
- On utilise la définition récursive d'une ou plusieures fonctions et/ou propriétés.

Exemple I

Propriété à démontrer : $\forall$`'a abin` b, `miroir`$($`miroir`$(b)) = b$

Preuve : Par induction sur b

- Cas de base : $b = avide$.

$$\texttt{miroir}(\texttt{miroir}(avide)) = \texttt{miroir}(avide) = avide$$

- Cas inductif : $b = abr(m, a_1, a_2)$.

$$\begin{aligned}
&\texttt{miroir}(\texttt{miroir}(b)) &=\\
&\texttt{miroir}(\texttt{miroir}(abr(m, a_1, a_2))) &=\\
&\texttt{miroir}(abr(m, \texttt{miroir}(a_2), \texttt{miroir}(a_1))) &=\\
&abr(m, \texttt{miroir}(\texttt{miroir}(a_1), \texttt{miroir}(\texttt{miroir}(a_2)))) &=_{h.r}\\
&abr(m, a_1, a_2) &=\\
&b
\end{aligned}$$

Exemple II

Propriété à démontrer :
$\forall$AbinSE b, peigne$(b) \rightarrow$ prof$(b) =$ nb_internes(b)

Où peigne est la propriété :

$$\texttt{peigne}(b) = \texttt{peigne_gauche}(b) \vee \texttt{peigne_droit}(b)$$

$$\begin{aligned} \texttt{peigne_gauche}(avide) &= true \\ \texttt{peigne_gauche}(noeud(a_1, a_2)) &= \texttt{peigne_gauche}(a_1) \wedge \\ &\quad \texttt{est_arbre_vide}(a_2) \\ \texttt{peigne_droit}(\ldots) &= \ldots \end{aligned}$$

Preuve : Par induction sur b

- Cas de base : $b = avide$.

$$\texttt{prof}(avide) = 0 = \texttt{nb_internes}(avide)$$

- Cas inductif : $b = noeud(a_1, a_2)$.
  $\texttt{peigne}(noeud(a_1, a_2))$ implique

  $\texttt{peigne_gauche}(noeud(a_1, a_2)) \vee \texttt{peigne_droit}(noeud(a_1, a_2))$

  Supposons sans perte de généralité

$$\texttt{peigne_gauche}(noeud(a_1, a_2))$$

C'est à dire : $\texttt{peigne_gauche}(a_1) \wedge a_2 = avide$
Alors : $\texttt{prof}(noeud(a_1, a_2)) = 1 + \texttt{prof}(a_1)$ et
$\texttt{nb_internes}(noeud(a_1, a_2)) = 1 + \texttt{nb_internes}(a_1) + 0$
Comme $\texttt{peigne_gauche}(a_1)$, alors par h.r.
$\texttt{prof}(a_1) = \texttt{prof}(a_1) = \texttt{nb_internes}(a_1)$ et donc
$\texttt{prof}(b) = 1 + \texttt{prof}(a_1) = 1 + \texttt{nb_internes}(a_1) =$
$\texttt{nb_internes}(b)$.

Induction sur les formules propositionnelles

Syntaxe du calcul propositionnel (rappel)

Soit $\mathcal{R}$ en ensemble dénombrable de lettres propositionnelles.

Définition : L'ensemble $\mathcal{F}_{prop}$ de formules du calcul propositionnel, est le plus petit ensemble t.q.

- $\mathcal{R} \subseteq \mathcal{F}_{prop}$
- Si $A \in \mathcal{F}_{prop}$, alors $\neg A \in \mathcal{F}_{prop}$.
- Si $A, B \in \mathcal{F}_{prop}$, alors $\vee(A, B)$, $\wedge(A, B)$, $\rightarrow (A, B) \in \mathcal{F}_{prop}$.

Exemple :	$\neg(p)$	$\vee(p, p)$	$\rightarrow (\wedge(p, q), \neg(r))$
Notation simplifiée :	$\neg p$	$p \vee p$	$(p \wedge q) \rightarrow \neg r$

Le calcul propositionnel en OCAML (rappel)

```
# type fprop = L of int | Neg of fprop |
               Ou of fprop * fprop |
               Et of fprop * fprop |
               Impl of fprop * fprop;;

# let f1 = Et(Ou(L(1),L(2)),Neg(L(3)));;
val f1 : fprop = Et (Ou (L 1, L 2), Neg (L 3))
```


Les formules comme des arbres

Les formules constitue un ensemble parametré par deux types `'a` et `'b`, où chaque étiquette d'une feuille est de type `'a` et chaque étiquette d'un nœud interne unaire ou binaire est de type `'b`.

Ceci donne une nouvelle définition du type `('a,'b) AbinII` :

L'ensemble `('a, 'b) AbinII` est le plus petit ensemble t.q.

- Si m est de type `'a`, alors $feuille(m) \in$ `('a, 'b) AbinII`
- Si $c \in$ `('a, 'b) AbinII` et o est de type `'b`, alors $noeud1(o, c) \in$ `('a, 'b) AbinII`.
- Si $c_1, c_2 \in$ `('a, 'b) AbinII` et o est de type `'b`, alors $noeud2(o, c_1, c_2) \in$ `('a, 'b) AbinII`.

Un type abstrait pour `('a, 'b) abinII`

Domaine :

`('a,'b) abinII`

Opérations de construction :

`cons_f` : `'a` $\rightarrow$ `('a,'b) abinII`

`cons_u` : `'b` $\times$ `('a,'b) abinII` $\rightarrow$ `('a,'b) abinII`

`cons_b` : `'b` $\times$ `('a,'b) abinII` $\times$ `('a,'b) abinII` $\rightarrow$ `('a,'b) abinII`

Opérations de test :

`est_f`, `est_u`,`est_b` : `('a,'b) abinII` $\rightarrow$ booléen

Opérations d'accès :

`etiquette_feuille` : `('a,'b) abinII` $\rightarrow$ `'a`

`etiquette_noeud` : `('a,'b) abinII` $\rightarrow$ `'b`

`fils_u`, `fils_g`, `fils_d` : `('a,'b) abinII` $\rightarrow$ `('a,'b) abinII`

Le calcul propositionnel en OCAML (deuxième possibilité)

```
# type ('a,'b) abinII =
  F of 'a |
  N1 of  'b * ('a,'b) abinII |
  N2 of 'b * ('a,'b) abinII * ('a,'b) abinII;;

# type op = Neg | Et | Ou | Impl;;

# let f1 = N2(Et,N2(Ou,F(1),F(2)),N1(Neg,F(3)));;
val f1 : (int, op) abinII =
    N2 (Et, N2 (Ou, F 1, F 2), N1 (Neg, F 3))
```


Fonctions récursives sur les formules

Formules $\subseteq$ Arbres, donc

pour définir une fonction sur les formules on peut utiliser l'un des schémas récursifs sur les arbres.

Schéma récursif pour les formules

$$\begin{aligned} \texttt{f}(a) &= g_1(a, \texttt{f}(\texttt{fils_u}(a))) \\ & \quad \text{si } \texttt{est_u}(a) \\ \texttt{f}(a) &= g_2(a, \texttt{f}(\texttt{fils_g}(a)), \texttt{f}(\texttt{fils_d}(a))) \\ & \quad \text{si } \texttt{est_b}(a) \end{aligned}$$

Quelques fois g_1 et g_2 se “ressemblent” beaucoup.

Cas particulier : lors que $\texttt{est_f}(a)$

Exemple : nb de lettres d'une formule

```
# let rec nblettres f = match f with
      F(_)      -> 1
    | N1(_,a)   -> nblettres a
    | N2(_,a,b) -> nblettres a + nblettres b ;;
```

Exercice : Qui sont les fonctions g_1 et g_2 ?

Preuve de propriétés par induction sur les formules

Exemple : Montrer que toute formule A est équivalente à une formule A' où le connecteur $\neg$ se trouve uniquement à gauche d'une lettre propositionnelle.

Calculs des séquents

Définition : Un séquent est un couple de la forme $\Delta \triangleright \Gamma$, où Δ et Γ sont de multi-ensembles de formules.

Exemple :

$$p, p, p \rightarrow q \quad \triangleright \quad r, p \vee s$$

$$p \rightarrow q \quad \triangleright$$

$$\triangleright \quad p, s$$

$$\triangleright$$

Définition d'un calcul des séquents

- On fixe des axiomes (des séquents particulièrs)
- On fixe des règles d'inférence de la forme $\dfrac{\Delta_1 \triangleright \Gamma_1 \ldots \Delta_n \triangleright \Gamma_n}{\Delta \triangleright \Gamma}$

Le système $\mathcal{G}$

Axiome : $\Delta, A \triangleright \Gamma, A$

Règles d'inférence logiques :

$$\frac{\Delta \triangleright \Gamma, A}{\Delta, \neg A \triangleright \Gamma}\ (\neg\ g) \qquad \frac{\Delta, A \triangleright \Gamma}{\Delta \triangleright \Gamma, \neg A}\ (\neg\ d)$$

$$\frac{\Delta \triangleright A, \Gamma \quad \Delta, B \triangleright \Gamma}{\Delta, A \rightarrow B \triangleright \Gamma}\ (\rightarrow\ g) \qquad \frac{\Delta, A \triangleright B, \Gamma}{\Delta \triangleright A \rightarrow B, \Gamma}\ (\rightarrow\ d)$$

$$\frac{\Delta, A, B \triangleright \Gamma}{\Delta, A \wedge B \triangleright \Gamma}\ (\wedge\ g) \qquad \frac{\Delta \triangleright A, \Gamma \quad \Delta \triangleright B, \Gamma}{\Delta \triangleright A \wedge B, \Gamma}\ (\wedge\ d)$$

$$\frac{\Delta, A \triangleright \Gamma \quad \Delta, B \triangleright \Gamma}{\Delta, A \vee B \triangleright \Gamma}\ (\vee g) \qquad \frac{\Delta \triangleright A, B, \Gamma}{\Delta \triangleright A \vee B, \Gamma}\ (\vee d)$$

Dérivation d'un séquent

Définition : La dérivation d'un séquent $\Gamma \triangleright \Delta$ à partir d'un ensemble de séquents Φ, notée $\Phi \vdash \Gamma \triangleright \Delta$, est un arbre fini tel que

- les nœuds sont des séquents
- chaque feuille est soit une formule de Φ, soit un axiome
- si S est le père des $S_1 \ldots S_n$, alors S est obtenu par l'application d'une règle d'inférence sur $S_1 \ldots S_n$.
- la racine de l'arbre est le séquent$\Gamma \triangleright \Delta$

Preuves et théorèmes

Définition : La preuve d'un séquent $\Gamma \triangleright \Delta$ est une dérivation de $\Gamma \triangleright \Delta$ à partir de l'ensemble vide de séquents. On dit dans ce cas que $\Gamma \triangleright \Delta$ est un théorème.

Premier exemple de dérivation dans $\mathcal{G}$

$$\dfrac{\dfrac{p \rhd q}{\rhd \neg p, q}(\neg\ d)}{\neg q \rhd \neg p}(\neg\ g)$$

On a une dérivation de $\neg q \rhd \neg p$ à partir de $p \rhd q$: $p \rhd q \vdash \neg q \rhd \neg p$.

Deuxième exemple de dérivation dans $\mathcal{G}$

$$\dfrac{\dfrac{p \rhd \neg p}{\rhd p \rightarrow \neg p}(\rightarrow\ d)}{\neg(p \rightarrow \neg p) \rhd}(\neg\ g)$$

On a $p \rhd \neg p \vdash \neg(p \rightarrow \neg p) \rhd$

Premier exemple de théorème dans $\mathcal{G}$

Modus ponens

$$\dfrac{p \rhd p, q\ (ax) \qquad p, q \rhd q\ (ax)}{p, p \rightarrow q \rhd q}(\rightarrow\ g)$$

On a $\vdash p, p \rightarrow q \rhd q$

Deuxième exemple de théorème dans $\mathcal{G}$

Tiers exclu

$$\dfrac{\dfrac{p \rhd p\ (ax)}{\rhd p, \neg p}(\neg\ d)}{\rhd p \vee \neg p}(\vee\ d)$$

On a $\vdash \rhd p \vee \neg p$

Troisième exemple de théorème dans $\mathcal{G}$

Loi de Pierce

$$\dfrac{\dfrac{\dfrac{p \rhd q, p \ (ax)}{\rhd p \to q, p}(\to\ d) \quad p \rhd p \ (ax)}{(p \to q) \to p \rhd p}(\to\ g)}{\rhd ((p \to q) \to p) \to p}(\to\ d)$$

On a $\vdash \rhd((p \to q) \to p) \to p$

Comment transformer quelques dérivations dans $\mathcal{G}$

Théorème : (Affaiblissement) Si $\Delta \rhd \Gamma$ est dérivable dans le système $\mathcal{G}$, alors $\Delta, A \rhd \Gamma$ et $\Delta \rhd A, \Gamma$ le sont aussi.

Théorème : (Contraction) Si $\Delta, A, A \rhd \Gamma$ est dérivable dans le système $\mathcal{G}$, alors $\Delta, A \rhd \Gamma$ l'est aussi. Si $\Delta \rhd \Gamma, A, A$ est dérivable dans le système $\mathcal{G}$, alors $\Delta \rhd \Gamma, A$ l'est aussi.

Conséquence logique (rappel)

Définition : Un multi-ensemble de formules Γ est conséquence logique d'un multi-ensemble de formules Δ, noté $\Delta \models \Gamma$, si toute interprétation qui satisfait toutes les formules de Δ satisfait au moins une formule de Γ.

Exemple :

$$p, p \to q \models q, r, s$$

Propriétés fondamentales du système $\mathcal{G}$

Théorème :(Correction) Le système $\mathcal{G}$ est correcte, i.e., si $\Delta \rhd \Gamma$ est un théorème, alors $\Delta \models \Gamma$.

Théorème :(Complétude) Le système $\mathcal{G}$ est complet, i.e., si $\Delta \models \Gamma$, alors $\Delta \rhd \Gamma$ est un théorème.