Introduction à la logique

MAM 3 Polytech'Nice, 2010-2011
Loïc Pottier

Ce cours est susceptible d'être modifié en ligne de temps en temps. Je me suis inspiré des cours suivants: E. Beffara, J.Goubault-Larrecq, G.Dowek. Il y aura quelques démonstrations difficiles dans ce cours[1], mais pas toutes; les autres se trouvent dans les cours cités. La liste des TD (13x2h) et devoirs associés se trouve ici. Les TD seront faits sur machines avec le logiciel Coq qui se télécharge ici. Y.Bertot et P. Castéran ont écrit un manuel en français et en anglais (traduit en chinois aussi).

Compléments: vous pouvez regarder cette conférence de J.M.Deshouillers avant, pendant, ou après la lecture de ce cours. Ainsi que cette conférence de J.Y.Girard, à la fois profonde, humoristique et érudite.
Vous pouvez lire les livres de vulgarisation de Gilles Dowek, par exemple La Logique.
Aussi Logicomix, cette récente BD retraçant l'histoire de Bertrand Russell et de la logique (un peu romancée, pourtant) est géniale!
Enfin, plus pointue, cette vidéo de V.Voevodsky à propos de l'utilisation de Coq pour des mathématiques contemporaines.

Pour imprimer ce cours, mieux vaut le télécharger en PDF d'abord (menu File->Download as->PDF).

---

1. par exemple: la complétude du calcul des séquents.

I. Introduction, langages formels.

1. Quelques exemples de phrases où de la logique intervient.

Syllogisme:

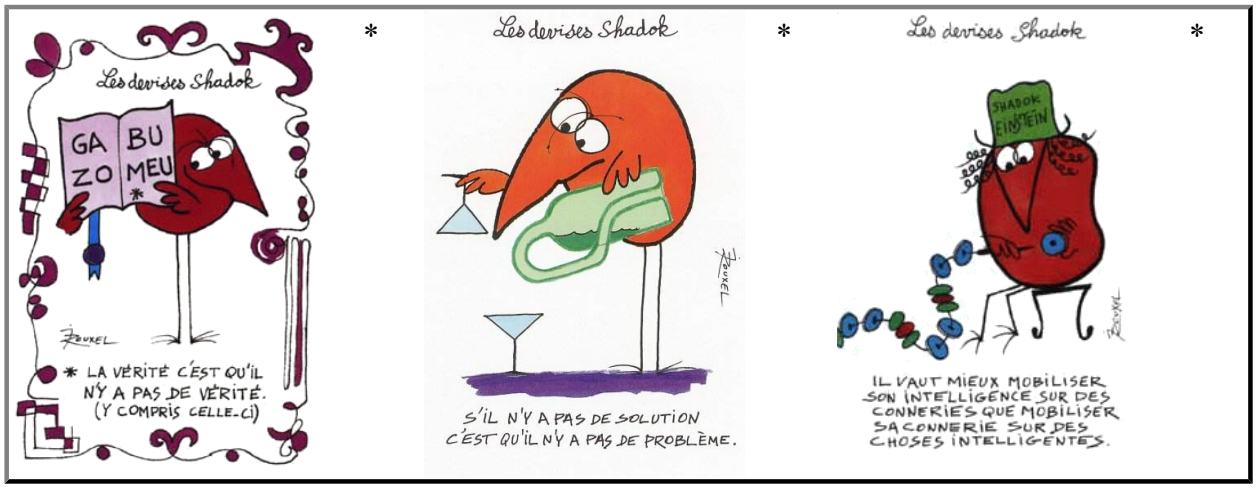
Tous les hommes sont mortels,
Socrate est un homme,
donc Socrate est mortel.

Sophismes:

Tout ce qui est rare est cher,
un iphone bon marché est rare,
donc un iphone bon marché est cher.

Plus il y a de gruyère, plus il y a de trous.
Plus il y a de trous, moins il y a de gruyère.
Donc plus il y a de gruyère, moins il y a de gruyère.

Absurde:



Paradoxes:

Un barbier est un homme qui rase les hommes qui ne se rasent pas eux-mêmes.
Le barbier se rase-t-il?

Un menteur dit "Je mens".

L'ensemble des ensembles. Il appartient à lui-même.
Considérons l'ensemble des ensembles qui n'appartiennent pas à eux- mêmes.
Appartient-il à lui-même?

Bertrand Russell

Déductions.

Il existe en Écosse un club très fermé qui obéit aux règles suivantes :

– Tout membre non écossais porte des chaussettes rouges.
– Tout membre porte un kilt ou ne porte pas de chaussettes rouges.
– Les membres mariés ne sortent pas le dimanche.
– Un membre sort le dimanche si et seulement s'il est écossais.
– Tout membre qui porte un kilt est écossais et marié.
– Tout membre écossais porte un kilt.

Montrer que ce club est si fermé qu'il ne peut accepter personne.

Jack Palmer enquête. Il aboutit aux constations suivantes:

– Si Padmé n'a pas rencontré Dark Vador l'autre nuit, c'est que Dark Vador est le meurtrier ou que Padmé est une menteuse.
– Si Dark Vador n'est pas le meurtrier, alors Padmé n'a pas rencontré Dark Vador l'autre nuit et le crime a eu lieu après minuit.
– Si le crime a eu lieu après minuit, alors Dark Vador est le meurtrier ou Padmé n'est pas une menteuse.

Il en conclut que Dark Vador est le meurtrier.

A-t-il raison?

Mathématiques.

> Démontrons par l'absurde que l'ensemble des nombres premiers est infini. Supposons qu'il est fini et soit N le produit de tous les nombres premiers plus 1. Comme on sait que 2 est premier, N est différent de 1; or tout nombre plus grand que 1 et non premier a au moins un facteur premier plus petit, et N n'est divisible par aucun des nombres premiers, car le reste de la division est 1 dans tous les cas. Donc N est premier. Mais par définition, N est strictement plus grand que tous les nombres premiers. Contradiction.

Informatique.

```
int fact(int n){
  if (n == 0){
    return 1;}
  else {
    return n * fact(n-1);}
}
```

2. Isoler la logique de ces phrases: formaliser.

Pour faciliter la compréhension, on isole les propositions logiques et on les relie entre elles avec des connecteurs logiques. Prenons l'exemple de Jack Palmer:

– Si Padmé n'a pas rencontré Dark Vador l'autre nuit, c'est que Dark Vador est le meurtrier ou que Padmé est une menteuse
– Si Dark Vador n'est pas le meurtrier, alors Padmé n'a pas rencontré Dark Vador l'autre nuit et le crime a eu lieu après minuit
– Si le crime a eu lieu après minuit, alors Dark Vador est le meurtrier ou Padmé n'est pas une menteuse.
Il en conclut que Dark Vador est le meurtrier.

A = "Padmé a rencontré Dark Vador l'autre nuit"
B = "Dark Vador est le meurtrier"
C = "Padmé est une menteuse"
D = "le crime a eu lieu après minuit"

le problème devient

si (non A) est vraie alors B ou C est vraie,
et
si (non B) est vraie alors (non A) et D sont vraies,
et
si D est vraie alors B ou (non C) est vraie,
alors B est vraie

et plus formellement encore:

```
( ((non A) implique (B ou C))
et ((non B) implique ((non A) et D))
et (D implique (B ou (non C)))
) implique B
```

avec des symboles, et en enlevant des parenthèses inélégantes:

```
(non A -> B \/ C) /\ ( non B -> non A /\ D) /\ (D -> B \/ non C) -> B
```

on peut alors même utiliser une machine pour le démontrer: le système Coq

```
Require Export Classical_Prop.
Lemma jack_palmer: forall A B C D:Prop,
(not A -> (B \/ C)) /\ (not B -> (not A /\ D)) /\ (D -> (B \/ not C)) -> B .
```

```
intros.
apply NNPP.
tauto.
Qed.
```

3. Syntaxe et langage formels.

La logique ne souffre pas d'ambiguïté, encore moins que les mathématiques... Il lui faut un langage formel.

Qu'est-ce-qu'un langage formel? Commençons par en donner une définition un peu informelle...et un exemple.

Alphabet.
On se donne d'abord S, un ensemble fini de lettres (ou symboles), qu'on appelle alphabet.

Prenons l'exemple du calcul des propositions.
Dans ce cas, S = V U {non, /\, \/, ->} U { (,), <espace> },
où V = {A,A1,A2,...,B,B1,B2,...} est l'ensemble des "variables propositionnelles",
/\ dénote la conjonction logique: le "et"
\/ dénote la disjonction: le "ou"
et -> dénote l'implication, qu'on note aussi => en mathématiques.

Mots.
Les suites finies de lettres sont appelés les mots. On note S* leur ensemble.

Un langage est par définition un sous-ensemble de S*.

Règles d'inférence.
Un langage est dit "formel" quand il est défini par des règles d'inférence.
Une règle d'inférence décrit comment construire un nouveau mot à partir de mots déjà construits.

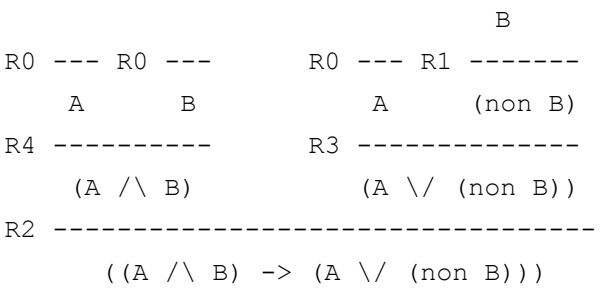
Pour le langage formel du calcul des propositions, qu'on va appeler CP, les règles sont:

```
       f            f    g          f    g          f    g
R1: -------    R2: --------   R3: --------   R4: --------   R0: -----  si x ∈ V
    (non f)        (f -> g)       (f /\ g)       (f \/ g)          x
```

Une règle se lit de la façon suivante: si les mots au-dessus de la barre sont dans le langage formel qu'on définit, alors le mot en-dessous de la barre y est aussi. Il peut y avoir des conditions supplémentaires, qu'on écrit à droite de la règle.

Par exemple, le mot ((A /\ B) -> (A \/ (non B))) appartient à CP. En effet, on peut le construire avec les règles suivantes:

```
                          R0 ---
```



Arbre de preuve.
Notons que ce schéma a la structure d'un arbre. Le tronc est le mot du bas, et, en montant, les branches se divisent ou se poursuivent à chaque règle, et les feuilles sont les règles R0.
C'est aussi une sorte de preuve/démonstration que le mot qui se trouve en bas est bien dans le langage formel CP. On appelle ça un "arbre de preuve".

Les mots de CP seront appelés "formules du calcul des propositions", ou plus brièvement "formules".

Exercice 1: montrer que les formules suivantes sont dans CP:

- (A -> (B -> A))
- ((A /\ (B \/ C)) -> ((A /\ B) \/ (A /\ C)))
- ((non (non A)) -> A)
- ((non (A /\ B)) -> ((non A) \/ (non B)))

Formellement (!), on définit un langage formel ainsi:

Définition (alphabet, règles d'inférence, langage formel):
Soit S un ensemble fini ou infini dénombrable, qu'on appellera "alphabet".
Soit S l'ensemble des suites finies de S, qu'on appellera "mots" de S.*
Une règle d'inférence r d'arité n est une relation (n+1)-aire sur S, i.e. une partie de S* x...x S* (S* apparaît n+1 fois). S*i $(x_1,...,x_{n+1}) \in r$, on notera:

$$r\ \frac{x_1 \ \ldots \ x_n}{x_{n+1}}$$

*Etant donné R un ensemble de règles d'inférence, l*e langage formel F défini par R est par définition le plus petit sous-ensemble de S* stable par R, i.e. tel que
$\forall$ *r* $\in$ *R telle que r est d'arité n,*
$\forall$ $x_1,...,x_{n+1} \in$ S* tels que $(x_1,...,x_{n+1}) \in r$, $x_1,...,x_n \in F \Rightarrow x_{n+1} \in F$

Propriété: *soit F_0 la réunion des règles d'arité 0 de R.*
Soit F_{n+1} la réunion de F_n et des mots de S obtenus par les règles de R à partir des mots de F_n. Alors F est égal à la réunion de tous les F_n*

Définition (profondeur): *étant donnée un mot f de F, le plus petit n tel que f appartient F_n est appelé la profondeur de f.*

Par exemple, la profondeur de la formule ((A /\ B) -> (A \/ (non B))) est 3.

Exercice 2: prenons S = { (,) } l'alphabet constitué uniquement des deux parenthèses,
et les règles d'inférence suivantes:

```
        f             f    g
R1: -------    R2: --------   R3: ---   où o dénote le mot vide
      (f)             fg           o
```

Soit D le langage formel défini par ces règles. Les mots suivants sont-ils dans D?
1. ()(())
2. ((()))(
3. (()())
4.)))(((

Comment caractériser ces mots?

II. Calcul des propositions.

1. Syntaxe.

Résumons pour commencer cette section la définition précédente du langage formel du calcul des propositions:

Définition (calcul des propositions):
Alphabet: S = V U {non, /\, \/, ->} U { (,), <espace> }, où V = {A,A1,A2,...,B,B1,B2,...}
Règles: RCP = {R0, R1, R2, R3, R4}

```
       f             f    g          f    g         f     g
R1: -------    R2: --------   R3: --------   R4: --------   R0: -----  si x ∈ V
    (non f)        (f -> g)       (f /\ g)       (f \/ g)         x
```

Formules: CP

Abus de notations: pour alléger, on évitera d'écrire toutes les parenthèses des formules, comme pour les formules arithmétiques. En effet on écrit a^2/b + c/d et non (((a^2)/b)+(c/d)), car la puissance a priorité sur la division, qui a priorité sur l'addition. De même on donne un ordre de priorité aux connecteurs logiques: non a la plus forte priorité, ensuite /\ , puis \/ et enfin ->.
Ainsi ((A /\ B) -> (A \/ (non B))) sera écrite A /\ B -> A \/ non B. Cool, non?

2. Sémantique classique du calcul des propositions.

Une fois qu'on a formalisé les choses, on n'a pas fini pour autant...

C'est un peu comme lorsque l'on connaît le vocabulaire et la grammaire d'une langue: cela ne suffit pas pour dire des phrases qui ont un sens!
Il nous faut donner un sens aux formules du calcul des propositions. Et en logique, le sens d'une formule c'est sa vérité: est-elle vraie ou non?
Par exemple, la formule A -> A est intuitivement toujours vraie, que A soit vraie ou fausse. Son sens commun est "si A est vraie, alors A est vraie".
La formule A /\ B, elle, n'est pas toujours vraie. Par exemple si A est vraie et si B est fausse, elle est fausse. Elle signifie "A est vraie et B est vraie".
Plus généralement, le sens d'une formule dépend de celui de ses sous-formules.
Par contre, le sens d'une variable propositionnelle (i.e. une formule de V) est à priori libre: elle est vraie ou fausse, selon les cas qui nous intéressent.

2.1. L'algèbre de Boole.

Nous savons tous donner un sens logique à des formules simples:
- non A: est vraie ssi A est fausse
- A /\ B: est vraie ssi A est vraie et B est vraie
- A \/ B: est vraie ssi A est vraie ou B est vraie
- A -> B: est si, lorsque A est vraie, B est vraie.[2]

Le premier à avoir mis cela en équations a été Georges Boole.
L'algèbre de Boole (CP/<=>, \/, /\) est obtenue en quotientant CP par la relation d'équivalence qui identifie les formules équivalentes, i.e. qui sont simultanément vraies ou fausses. Par exemple, A /\ B et B /\ A sont équivalentes. Ensuite, on ajoute à CP/<=> les opérations \/ et /\, qui passent au quotient.

Ce n'est pas une algèbre au sens classique: la loi \/ n'a pas d'opposé. Mais pour le reste, on retrouve par exemple la distributivité:
A /\ (B \/ C) = (A /\ B) \/ (A /\ C)

Nous utiliserons une version encore plus algébrique de son calcul, en remplaçant la disjonction par la disjonction exclusive: A + B = (A \/ B) /\ (non (A /\ B))
On voit que A + B est vraie ssi A est vraie et B est fausse, ou A est fausse et B est vraie.
Maintenant, (CP/<=>, +, /\) est une algèbre commutative au sens classique, et même un anneau.

2.2. Interprétation logique des formules: polynômes booléens.

Considérons que 0 représente la valeur de vérité fausse et 1 la valeur vraie.

Le fait de donner un sens logique à une formule, 0 ou 1, sera appelé "interpréter la formule". Ce sens s'obtient par calcul à partir du sens des sous-formules:

Définition: *une interprétation est une application v de CP dans le* corps à 2 éléments $\mathbb{Z}_2$ = {0,1}, qui vérifie les axiomes suivants:

---

2. Dans le dernier cas, il manque quelque chose... que dire quand A est fausse?

a. *v(non f) = 1 + v(f)*
b. *v(f /\ g) = v(f)v(g)*
c. *v(f \/ g) = v(f)v(g) + v(f) + v(g)*
d. *v(f -> g) = v(f)(v(g) + 1) + 1*

On retrouve le sens commun:
- non f est vraie <=> v(non f) = 1 <=> v(f) + 1 = 1 <=> v(f) = 0 <=> f est fausse
- A /\ B est vraie <=> v(A /\ B) = 1 <=> v(A)v(B) = 1 <=> v(f) = 1 et v(g) = 1 <=> A est vraie et B est vraie.
etc.

On peut aussi construire les "tables de vérité" des connecteurs logiques.
Pour l'implication ->:

v(A -> B)	v(B) = 0	v(B) = 1
v(A) = 0	1	1
v(A) = 1	0	1

On voit que l'implication est fausse uniquement quand l'hypothèse est vraie et la conclusion fausse, ce qui est raisonnable. Notons que l'implication est vraie dès que l'hypothèse est fausse.[3] [4]
Exercice 1: construire les tables de vérité de /\, \/, + et non.
Exercice 2: exprimer le connecteur d'équivalence <-> en fonction des autres connecteurs logiques. Donner une formule algébrique exprimant v(A <-> B) en fonction de v(A) et v(B).

Calcul de v(f).
Par exemple calculons v(A -> A).
D'après d., v(A -> A) = v(A)(v(A) + 1) + 1
En développant on obtient $v(A)^2 + v(A) + 1$.
Mais dans $\mathbb{Z}_2$, $x^2 = x$, donc v(A -> A) = v(A) + v(A) + 1.
En remarquant que dans $\mathbb{Z}_2$, x + x = 0, on obtient finalement v(A -> A) = 1.

---

3. En effet, à partir d'une hypothèse fausse, on peut tout démontrer! Par exemple qu'un iphone bon marché est cher, puisque tout ce qui est rare n'est pas forcément cher, en fait... Le raisonnement dans ce sophisme est correct, comme dans tout syllogisme, mais c'est la première hypothèse qui est fausse, bien que "généralement" vraie...

4. Un certain philosophe fut très choqué quand Bertrand Russell lui apprit qu'une proposition fausse implique n'importe quelle proposition. Il dit à Russell:
- "Voulez-vous dire qu'il résulte de la proposition: "deux et deux font cinq" que vous êtes le Pape ?" et Russell répondit: "oui". Le philosophe lui demanda alors: "pouvez-vous le prouver ?"
Russell répondit: "certainement" et inventa sur le champ la démonstration suivante:
- "Supposons que 2 + 2 = 5, on retranche 2 aux deux membres et on obtient 2 = 3. on transpose pour avoir 3 = 2. Enfin, on retranche 1 aux deux membres et on obtient 2 = 1.
A présent, le Pape et moi nous sommes 2. Mais 2 = 1 et par conséquent, le Pape et moi sommes 1. Donc, je suis le Pape."

Ainsi la formule A -> A est toujours vraie, que A soit vraie ou fausse. On s'y attendait un peu!

Autre exemple: calculons v(f), pour f = A \/ B -> A:
v(f) = v(A \/ B)(v(A) + 1) + 1 par d.
    = (v(A)v(B) + v(A) + v(B)) (v(A) + 1)) + 1 par c.
ensuite, on peut développer et simplifier cette expression algébrique:
v(f) = $v(A)^2v(B) + v(A)v(B) + v(A)^2 + v(A) + v(A)v(B) + v(B) + 1$
    = $v(A)v(B) + v(A)v(B) + v(A) + v(A) + v(A)v(B) + v(B) + 1$ car $\forall x, x^2 = x$
    = v(A)v(B) + v(B) + 1 car $\forall x, x + x = 0$

Par exemple, si v(A) = 1, v(f) = 1 (car v(B)+v(B) = 0). La formule f est donc vraie dès que A est vraie.

Exercice 3: déterminer tous les cas où f est vraie.

2.3. Tautologies.

Définition: une tautologie est une formule f telle que pour toute interprétation v, v(f) = 1.

Exercice 4: montrer que les formules suivantes sont des tautologies:
1. A -> A
2. non (non A) -> A
3. A -> non (non A)
4. non (A /\ B) -> non A \/ non B
5. non (A \/ B) -> non A /\ non B

2.4. Premier algorithme de décision logique: polynômes booléens.

On voit que le calcul de v(f) conduit en général à un polynôme dont les variables sont les valeurs de v pour les variables propositionnelles, et à coefficients dans $\mathbb{Z}_2$. Cette forme est une forme canonique: deux formules sont logiquement équivalentes si leurs formes polynomiales sont égales. Ce qui conduit à un premier algorithme pour savoir si une formule est une tautologie: il suffit de calculer son polynôme. S'il se réduit à 1, c'est une tautologie. Sinon, c'est que la formule peut être fausse pour certaine valeurs des variables. Ceci qui peut se faire à la main...ou avec Maple, ou Coq.

Théorème (et algorithme): *une formule du calcul des propositions est une tautologie si et seulement si le calcul de v(f) par les règles a. b. c. d. et les règles de calcul dans* $\mathbb{Z}_2$ conduit au polynôme 1.

De même, une formule dont le polynôme obtenu par calcul de v(f) est nul, est toujours fausse. On appelle ça une antilogie.

Calculons!
Reprenons la formule qu'on avait obtenu en formalisant l'enquête de Jack Palmer:

f = (non A -> B \/ C) /\ (non B -> non A /\ D) /\ (D -> B \/ non C) -> B

et calculons son polynôme:
v(f) = v(B)(v((non A -> B \/ C) /\ (non B -> non A /\ D) /\ (D -> B \/ non C)) + 1) + 1
donc

$$v(f) = v(B)(v(f1)v(f2)v(f3) + 1) + 1$$

où
f1 = non A -> B \/ C
f2 = non B -> non A /\ D
f3 = D -> B \/ non C

Calculons les polynômes de f1, f2 et f3:

v(f1) = v(non A)((v(B \/ C) + 1) + 1 = (v(A) + 1)((v(B)v(C)+v(B)+v(C)) + 1) + 1
pour alléger, on notera v(A) par A, v(B) par B etc.
ainsi v(f1) = (A + 1)(BC + B + C + 1) + 1 = ABC + AB + AC + A + BC + B + C + 1 + 1
donc

$$v(f1)= ABC + AB + AC + BC + A + B + C$$

v(f2) = v(non B)(v(non A /\ D) + 1) + 1 = (B + 1)((A + 1)D + 1) + 1 = (B + 1)(AD + D + 1) + 1
= ABD + BD + B + AD + D + 1 + 1
et donc

$$v(f2) = ABD + AD + BD + B + D$$

v(f3) = v(D)(v(B \/ non C) + 1) + 1 = D(B(C + 1) + B + (C + 1) + 1) + 1 = D(BC + B + B + C) + 1 = D(BC + C) + 1
donf v(f3) = BCD + CD + 1

Maintenant calculons v(f1)v(f2):
v(f1)v(f2) = (ABC + AB + AC + BC + A + B + C)(ABD + AD + BD + B + D)
= ~~ABCD~~ + ~~ABCD~~ + ~~ABCD~~ + ~~ABC~~ + ~~ABCD~~ (on a simplifié x^2 en x partout)
+ ~~ABD~~ + ~~ABD~~ + ~~ABD~~ + ~~AB~~ + ~~ABD~~
+ ~~ABCD~~ + ~~ACD~~ + ~~ABCD~~ + ~~ABC~~ + ~~ACD~~
+ ~~ABCD~~ + ~~ABCD~~ + ~~BCD~~ + ~~BC~~ + ~~BCD~~
+ ~~ABD~~ + ~~AD~~ + ~~ABD~~ + ~~AB~~ + ~~AD~~
+ ~~ABD~~ + ~~ABD~~ + ~~BD~~ + B + ~~BD~~
+ ABCD + ACD + BCD + ~~BC~~ + CD
= ABCD + ACD + BCD + CD + B
puis v(f1)(v(f2)v(f3)
= (ABCD + ACD + BCD + CD + B)(BCD + CD + 1)
= ~~ABCD + ABCD~~ + ~~ABCD~~
+ ~~ABCD~~ + ~~ACD + ACD~~
+ ~~BCD + BCD~~ + ~~BCD~~
+ ~~BCD~~ + ~~CD + CD~~
+ ~~BCD + BCD~~ + B
= B

enfin v(f) = B(B + 1) + 1 = B + B + 1 = 1

La formule f est donc vraie quelles que soient la vérité des variables propositionnelles:
A = "Padmé a rencontré Dark Vador l'autre nuit"
B = "Dark Vador est le meurtrier"
C = "Padmé est une menteuse"
D = "le crime a eu lieu après minuit"

Ainsi, Dark Vador est bien le meurtrier...on s'en doutait!

Exercice 5: Jack Palmer peut-il conclure que Padmé n'est pas une menteuse? Qu'elle n'a pas rencontré Dark Vador l'autre nuit? ou que le meurtre a eu lieu après minuit?

Exercice 6:
Montrer que le club écossais du début ne peut avoir de membre, en montrant que la conjonction des ses règles a un polynôme nul.

2.5. Second algorithme de décison logique: tables de vérité.

En fait, le premier algorithme qui vient à l'esprit pour savoir si une formule est vraie tout le temps, est de calculer sa valeur de vérité (son interprétation) dans tous les cas possibles des valeurs de vérité de ses variables. Par exemple, pour la formule A /\ B -> A \/ B, on distinguera 2x2 cas, en calculant successivement les valeurs de vérité des sous-formules:

A	1	1	0	0
B	1	0	1	0
A /\ B	1	0	0	0
A \/ B	1	1	1	0
A /\ B -> A \/ B	1	1	1	1

Pour la formule de Jack Palmer,

f = (non A -> B \/ C) /\ (non B -> non A /\ D) /\ (D -> B \/ non C) -> B

on a 4 variables, donc 16 cas, et 16 sous-formules, ce qui fait un tableau avec 16 lignes et 17 colonnes...oups! Complétez-le, j'ai la flemme, là[5]:

A	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
B	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
C	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
D	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
non A	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

5. d'autant plus qu'à la télé, il y a l'épisode II de Star wars ce soir, et ça va bientôt commencer...

B \/ C	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0	0
f1 = non A -> B \/ C	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
non B	0															
non A /\ D	0															
f2 = non B -> non A /\ D	1															
non C	0															
B \/ non C	1															
f3 = D -> B \/ non C	1															
f1 /\ f2	1															
f1 /\ f2 /\ f3	1															
f = f1 /\ f2 /\ f3 -> B	1															

Bon, évidemment, dès qu'on a plus de n=2 variables, mieux vaut écrire un programme (Maple, C, java, ocaml, etc) qui teste les 2^n possibilités!
Quand n devient grand, il faut faire preuve d'astuce, et les agorithmes deviennent plus malins, mais toujours de complexité exponentielle en n, malheureusement.
En fait, c'est le problème central de l'informatique que l'on vient d'aborder:

existe-t-il un algorithme qui dit si une formule du calcul des propositions est une tautologie ou non en un temps non plus exponentiel mais polynomial en la taille de la formule?

Monthy Python Sacré Graal[6]

Les gens raisonnables pensent que non...mais on ne sait jamais!

2.6. Arbres de décision binaire.

Le principe est identique aux tables de vérité, on explore toutes les valeurs de vérité possibles des variables propositionnelles, mais pas globalement. On le fait variable par variable: ayant choisi une variable, on distingue deux cas: soit elle est vraie, soit elle est fausse. Dans chacun des cas, on peut simplifier la formule: par exemple, si A est vraie, A \/ f est vraie quelle que soit f. En pratique, cela simplifie grandement les formules évitant souvent des calculs qu'on aurait fait avec la méthode de tables de vérité.

---

6. Excellent film, beaucoup d'humour fondé sur l'absurde.

3. Preuves.

Bien que la notion de vérité logique d'une formule soit assez intuitive (quelque soit la vérité des variables propositionnelles, la formule est vraie) les méthodes précédentes sont pourtant très éloignées de la pratique. Elles sont bonnes pour les machines, mais convaincront peu de gens (en tout cas pas un juge). Comment fait-on pour savoir ou montrer qu'une formule logique est vraie? On la démontre, on en donne une preuve, c'est-à-dire une succession d'étapes de déduction, qui peut être longue, mais chaque étape est simple, compréhensible. Ainsi, à la fin, on est convaincu par la démonstration, même si on ne l'a plus toute entière en tête.
L'étape de déduction la plus courante est le modus ponens:

si A est vraie et si A implique B, alors B est vraie

Mais il en a d'autres, par exemple le tiers exclus, qui permet de commencer une preuve par cas:

de deux choses l'une: A est vraie ou bien A est fausse

la preuve par l'absurde:

si A est faux, on obtient une contradiction, donc A est vraie

la preuve par contraposée:

pour montrer que A entraîne B, montrons que si B est fausse, alors A aussi.

la double négation:

il est faux que A est fausse, donc A est vraie

etc.

Une preuve part toujours d'axiomes, c'est-à-dire d'assertions qu'on ne démontre pas, qu'on admet, soit car ce sont des données du problème (on les appelle alors plutôt "hypothèses"), ou que ce sont des faits communément admis.
Ce qui fait la valeur d'une preuve est alors non pas ses axiomes ou ses hypothèses, mais la succession correcte des étapes de déduction. Si ses axiomes sont vrais, alors sa conclusion est garantie être vraie. Par exemple, un syllogisme est une preuve correcte. Un sophisme est un syllogisme, donc logiquement correct, mais avec avec au moins une hypothèse fausse (même si ce n'est pas évident, et c'est l'intérêt du sophisme), ce qui explique sa conclusion manifestement fausse.

Avec le développement des sciences en général et des mathématiques en particulier, au XIX siècle, la nécessité de donner un fondement mathématique à la logique elle-même est apparu, et les mathématiciens ont mathématisé les notions de raisonnement et de preuve (Frege, Russell, Hilbert, etc). Le but étant d'exprimer non plus seulement les énoncés dans le langage formel des mathématiques, mais aussi les démonstrations. Cela a donné lieu à quelques difficultés, car il s'agissait d'appliquer une science à elle-même (les mathématiques), avec les risques de raisonnements ou définitions cycliques (qui se mordent la queue) que cela implique. Sont apparus avec Cantor les paradoxes de l'infini[7] et avec Russell, celui de l'ensemble des

ensembles. Les logiciens se sont alors employés à échaffauder des théories résolvant ces problèmes, ce qu'ils ont réussi, mais découvrant du même coup les limites de la méthode (Gödel).

Regardons ce que cela a donné dans le cas du calcul des propositions:

3.1. Un système à la Hilbert: SKC.

David Hilbert a le premier proposé de bâtir les mathématiques sur un système logique formel d'axiomes et de règles de déductions. Etudions un descendant de son système dans le cas des propositions.

Le langage des propositions est restreint ici à l'alphabet S'= V $\bigcup$ {->, F, (,), <espace>}. F représente une formule toujours fausse. Les autres connecteurs sont exprimés comme des abréviations (i.e. des macros) à l'aide de l'implication et de F:

non A = A -> F
A /\ B = non (A -> non B) = (A -> (B -> F)) -> F
A \/ B = non A -> B = (A -> F) -> B

Exercice 1: montrer que du point de vue des interprétations (avec v(F) = 0), ces égalités sont justifiées (i.e. les 2 formules d'une égalité ont même table de vérité).

Le langage des formules CP' est construit donc avec trois règles d'inférence:

```
                 f    f -> g
R1: ---    R2: --------    R0: -----  si x ∈ V
     F          (f -> g)          x
```

Théorèmes.
On définit un sous ensemble de CP' par les règles d'inférences suivantes:

```
K: ---------------          S: -----------------------------------
    f -> (g -> f)               (f -> g -> h) -> (f -> g) -> (f -> h)

                                     f  f -> g
C: ---------------------    MP: -----------
   ((f -> F) -> F) -> f              g
```

---

7. Dans un hôtel qui a une infinité de chambres, et qui est complet, il y a pourtant toujours moyen de caser un client impromptu... Et même une infinité dénombrable.

Les règles S, K, et C n'ont pas d'hypothèse: elles signifient que les formules en bas sont des axiomes, elles n'ont pas besoin de preuve.
Une seule règle permet de construire de nouveaux théorèmes: la règle du modus ponens MP.

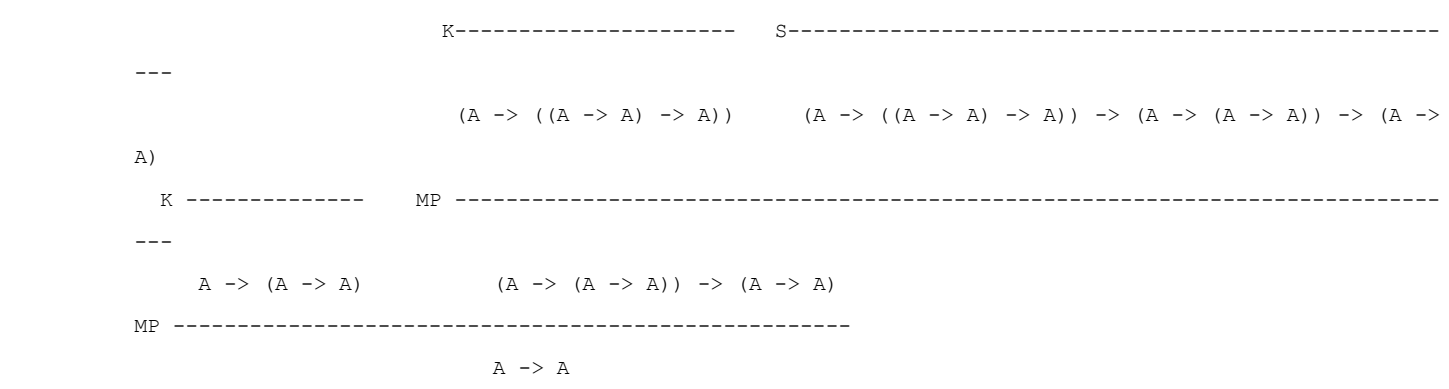
Appelons T_{SKC} l'ensemble des formules de CP' que l'ont peut construire avec ces règles. Ses éléments sont appelés "théorèmes", ou "formules démontrables", du système SKC. En effet, un théorème posséde une démonstration: c'est la succession des règles qui le construisent. Les règles K,S,C et MP sont appelées "règles de déduction".

Théorème: *T_{SKC} est égal à l'ensemble des tautologies, i.e. les formules f telles que v(f) = 1 pour toute interprétation v.*

Ce théorème en contient deux en fait: le premier, le "théorème de consistance", qui dit que tout théorème est une tautologie, et le second, le "théorème de complétude", qui dit que toute tautologie est démontrable, i.e. est un théorème. Le théorème de complétude est dur et long à démontrer...

Exemple: montrons que la formule A -> A est un théorème. Vu sa forme, elle ne peut provenir que de la règle MP. Il faut donc inventer une formule f telle que f et f -> (A -> A) soient des théorèmes... Prenons f = A -> (A -> A). f est facilement démontrée, puisque c'est un axiome (règle K). Reste à montrer f -> (A -> A), c'est à dire (A -> (A -> A)) -> (A -> A). Là encore on a pas le choix: il faut utiliser MP. Prenons alors dans ce cas f = A -> ((A -> A) -> A). C'est un axiome aussi (K), et il nous reste à montrer (A -> ((A -> A) -> A)) -> (A -> (A -> A)) -> (A -> A), qui est un axiome: la règle S avec f = A, g = A -> A, et h = A.

Un dessin de l'arbre de preuve est plus clair:



Voilà une démonstration formelle que A -> A est toujours vraie!
Bon, on a utilisé qu'une seule règle de déduction, le modus ponens, naturelle... mais les axiomes, eux, sont plus obscurs, surtout S...

Exercice 2: montrer que (B->C) -> (A->B) -> (A -> C) est un théorème.
Exercice 3: montrer que les axiomes K, S et C sont des tautologies.
Exercice 4: montrer que si les hypothèses de MP sont des tautologies, sa conclusion en est aussi une. Avec l'exercice 2, en déduire le théorème de consistance.

Le problème avec ce système SKC, c'est que l'usage du modus ponens est sportif: il faut à chaque fois inventer une formule, qui est souvent de plus en plus compliquée... On n'est pas guidé du tout par la formule que l'on veut démontrer.
Le système suivant est plus agréable:

3.2. Déduction naturelle.

Jugements:

Gerhard Gentzen

On définit maintenant le langage formel des "séquents", en ajoutant la virgule et le symbole "|-" à l'alphabet, et l'ensemble des règles d'inférence suivant:

```
S0: -------  si f ∈ CP'
     |- f

      f1 ... fn  f
Sn: ----------------   si f,f1,...,fn ∈ CP'
     f1,...,fn |- f
```

Un séquent H |- f est donc constitué d'un ensemble fini de formules H et d'une formule f. Il signifie "à partir des formules de E, on peut déduire f". Le symbole |- peut se lire "démontre".

Parmi ces jugements, seuls certains sont valides. On les définit avec des règles d'inférence. Les voici (on note f, H la réunion de {f} et de H[8]):

```
                                  H |- f
Ax: ------------          inclusion -------- si H est inclus dans G
     f, H |- f                    G |- f

           f, H |- g                H |- f   H |- f -> g
imp_intro -------------   imp_elim ----------------------
           H |- f -> g                   H |- g

                             f -> F, H |- F
F_intro: ---------   F_elim ----------------
         H, F |- f              H |- f
```

Les théorèmes sont maintenant les formules f telles que le séquent |- f est démontrable avec ces règles. On note T_{DN} leur ensemble.

---

8. attention: {f} et H ne sont pas nécessairement disjoints. C'est juste une façon de distinguer un élément dans H.

Ces règles sont dites "règles de la déduction naturelle", et c'est vrai qu'elles sont plus naturelles que celles du système SKC ... une fois qu'on s'est habitué à la notation des séquents!

Exemple: la formule A -> A est un théorème de T_{DN}. Sa preuve est simple, avec ce système de règles:

```
        Ax --------
            A |- A
imp_intro -----------
           |- A -> A
```

Exercice 5: démontrer que (A -> B) -> (B -> C) -> (A -> C) appartient à T_{DN}.

En utilisant les abréviations

non A = A -> F

A /\ B = (A -> (B -> F)) -> F

A \/ B = (A -> F) -> B

on montre facilement (faites-le en exercice) que l'on peut accélérer les démonstrations à l'aide de règles supplémentaires, conséquences des règles Ax, imp_intro, imp_elim, F_intro et F_elim:

```
            f, H |- F              non f, H |- F
non_intro -----------   non_elim ----------------
            H |- non f               H |- f

           H |- f  H |- g             H | f /\ g              H |- f /\ g
et_intro ----------------   et_elim1 ------------   et_elim2 -------------
            H |- f /\ g                H |- f                    H |- g

             H |- f                      H |- g
ou_intro1 ---------------   ou_intro2 ------------
            H |- f \/ g                 H |- f \/g

         H |- f \/ g   H |- f -> h   H |- g -> h
ou_elim ----------------------------------------
              H |- f \/ g -> h
```

Exercice 6: Démontrer que les formules suivantes sont des théorèmes de T_{DN}:
- non non A -> A
- A -> non non A
- A \/ non A
- non (A -> B) <-> A /\ non B, où f <-> g est une abréviation pour (f -> g) /\ (g -> f)
- (A -> B) <-> non A \/ B

Théorème: $T_{DN} = T_{SKC}$

Ce théorème n'est pas très difficile à démontrer: on montre qu'un théorème d'un système est démontrable dans l'autre en démontrant les règles d'un système dans l'autre.

Voilà donc une autre notion formelle de démonstration qui coïncide avec le sens logique commun, et qui est bien plus pratique à utiliser.

Il y en a un qui est encore plus pratique:

3.3. Calcul des séquents.

Il se trouve que la déduction naturelle fait encore intervenir la règle du modus ponens (imp_elim):

```
            H |- f   H |- f -> g
  imp_elim ----------------------
                   H |- g
```

qui a le gros désavantage d'obliger à inventer une formule f pour continuer la démonstration: la formule f n'est pas facilement déduite de H et g. Les autres règles n'ont pas cet inconvénient. Elles s'appliquent sans douleur...
Le calcul des séquents suivant résoud ce problème, en éliminant cette règle. Il est encore dû à Gentzen.
On étend le langage des séquents précédents en autorisant plus d'une formule à droite:

```
      f1 ... fn  g1 ... gm
Sq: ------------------------   si n,m ≥0, et f1,...,fn,g1,...,gm ∈ CP'
     f1,...,fn |- g1,...,gm
```

En fait, on considère que de part et d'autre du signe |- se trouvent des ensembles finis de formules. On notera H,f ou f,H la réunion de H et {f}.

Les règles du calcul des séquents sont alors les suivantes:

```
AxS -----------------                    F_gauche -----------
     f, H |- f, G                                 F, H |- G

            H |- f, G   g, H |- G                     f, H |- g, G
imp_gauche ----------------------        imp_droite ----------------
                f -> g, H |- G                          H |- f -> g, G
```

Notons T_S l'ensemble des théorèmes de ce système de règles, c'est-à-dire l'ensemble des formules f telles que le séquent |- f est démontrable.

Exercice 7: démontrer que (A -> B) -> (B -> C) -> (A -> C) appartient à T_S.

Soit v: CP' -> $\mathbb{Z}_2$ une interprétation et H |- G un séquent. On définit v(H |- G) ainsi:
- si H est vide, et G = $g_1,...,g_m$, v(H |- G) = v(g_1 \/...\/ g_m)
- si G est vide, et H = $f_1,...,f_n$, v(H |- G) = 1 + v(f_1 /\.../\ f_n)

- sinon, H = $f_1,...,f_n$, G = $g_1,...,g_m$, et v(H |- G) = v(f_1 /\.../\ f_n -> g_1 \/...\/ g_m)

Théorème: *un séquent H |- G est démontrable par les règles du calcul des séquents ssi pour toute interprétation v, v(H |- G) = 1.*

Corollaire: *une formule f est une tautologie ssi le séquent |- f est démontrable.*

Démonstration du théorème:
on va montrer le sens difficile (complétude): ($\forall$v, v(H |-G) = 1) => H |- G est démontrable. L'autre sens (consistance) est sans difficulté: il suffit de vérifier que les règles ne produisent que des séquents vrais pour toute interprétation si leurs hypothèses le sont.

Soient H |- G un séquent tel que $\forall$v, v(H |-G) = 1.

Remarquons que v(g_1 \/...\/ g_m) = 1 + $\prod_{i=1...m}$ (v(g_i)+1)

Ainsi, on a

(1) v(H |- G) = 1 <=> $\prod_{f \in H}$v(f) $\prod_{g \in G}$(v(g)+1) = 0

Notons qu'un produit dont l'ensemble des indices est vide est égal à 1.
Procédons par récurrence sur la taille de H |- G, i.e. la somme des tailles des formules de H et G. La taille d'une formule étant le nombre de symboles de l'alphabet S' qu'elle contient.

Supposons que toutes les formules de H et G sont des variables ou la formule F.
Si H $\cap$ G est non vide, alors il existe f dans H et g dans G tels que f = g. Donc est démontrable par la règle AxS.

Si H et G sont disjoints, alors **(1)** montre que $\forall$v,v(H |- G) = 1 implique que H contient la formule F. Ce qui signifie que le séquent H |- G est démontrable par la règle F_gauche.

Supposons maintenant qu'il existe une formule de la forme a -> b dans G: G = a -> b, G'. Dans ce cas on peut démontrer H |- G par la règle imp_droite. Ce qui conduit au séquent a, H |- b, G'.

On a v(a, H |- b, G') = 1 + $\prod_{f \in H}$v(f). v(a). $\prod_{g \in G'}$(v(g)+1).(v(b)+1)

= 1 + $\prod_{f \in H}$v(f). $\prod_{g \in G'}$(v(g)+1).v(a)(v(b)+1)

= 1 + $\prod_{f \in H}$v(f). $\prod_{g \in G'}$(v(g)+1).(v(a -> b) + 1)

= v(H |- a -> b, G') = v(H |- G) = 1 par hypothèse. Comme le séquent a, H |- b, G' est de taille plus petite que celle de H |- G, par hypothèse de récurrence, il est démontrable, et ainsi H |- G aussi.

Reste le cas où G ne contient que des variables ou la formule F, et il y a une formule a -> b dans H: H = H', a -> b.
Dans ce cas on peut démontrer H |- G par la règle imp_gauche.
Ce qui conduit aux séquents H' |- a, G et b, H' |- G.

On a **(2)** v(H' |- a, G) = 1 + $\prod_{f \in H'}$v(f) $\prod_{g \in G}$(v(g)+1).(v(a)+1),

et **(3)** v(b, H' |- G) = 1 + $\prod_{f \in H'}$v(f).v(b). $\prod_{g \in G}$(v(g)+1).

Or v(a -> b, H' |- G) = 1 par hypothèse.
Donc

(4) 1+$\prod_{f \in H'}$v(f).(v(a)(v(b)+1)+1). $\prod_{g \in G}$(v(g)+1) = 1

Prenons v telle que v(g) = 0 pour toutes les variables de G. Ainsi on a, puisque v(F) = 0 aussi, et que G n'est composé que de variables ou de F:

(5) $\prod_{g \in G}$(v(g)+1) = 1, et donc, avec (4), $\prod_{f \in H'}$v(f).(v(a)(v(b)+1)+1) = 0.

Si v(a) = 0, on en déduit que $\prod_{f \in H'}$v(f) = 0. Donc, par (2) et (5), v(H' |- a, G) = 1 et par (3) et (5), v(b, H' |- G) = 1.

Si v(a) = 1, on en déduit $\prod_{f \in H'}$v(f).v(b) = 0. Donc, par (3) et (5), v(b, H' |- G) = 1. Par (2), on a v(H' |- a, G) = 1.

Prenons maintenant v telle que v(g) = 1 pour une des variables de G.

On a alors $\prod_{g \in G}$(v(g)+1) = 0.

Et donc, par (2), on a v(H' |- a, G) = 1, et par (3), v(b, H' |- G) = 1.

Donc, quelque soit v, on a v(H' |- a, G) = 1, et v(b, H' |- G) = 1.
Ces séquents sont de taille plus petite que celle de H |- G. Ainsi, par hypothèse de récurrence, ils sont démontrables. Et donc H |- G aussi.
CQFD.

Règles dérivées:

Le modus ponens est une conséquence non triviale des ces règles:

```
     H |- f, G    H |- f -> g, G
MPS ----------------------------
             H |- g, G
```

Il est très proche de la "règle de coupure", également superflue:

```
         f, H |- G    H |- f, G
Coupure ------------------------
                H |- G
```

On peut montrer que les règles suivantes sont des conséquences de ces règles:

```
           f, g, H |- G                          H |- f, G   H |- g, G
et_gauche ---------------          et_droite ----------------------
```

```
          f /\ g , H |- G                          H |- f /\ g, G

          f, H |- G   g, H |- G                 H |- f, g, G
ou_gauche --------------------       ou_droite ----------------
             f \/ g, H |- G                     H |- f \/ g, G

             H |- f, G                            f, H |- G
non_gauche --------------            non_droite ---------------
            non f, H |- G                         H |- non f, G
```

Exercice 8: montrer justement que ces règles sont conséquences des règles AxS, F_gauche, impl_droite et impl_gauche.

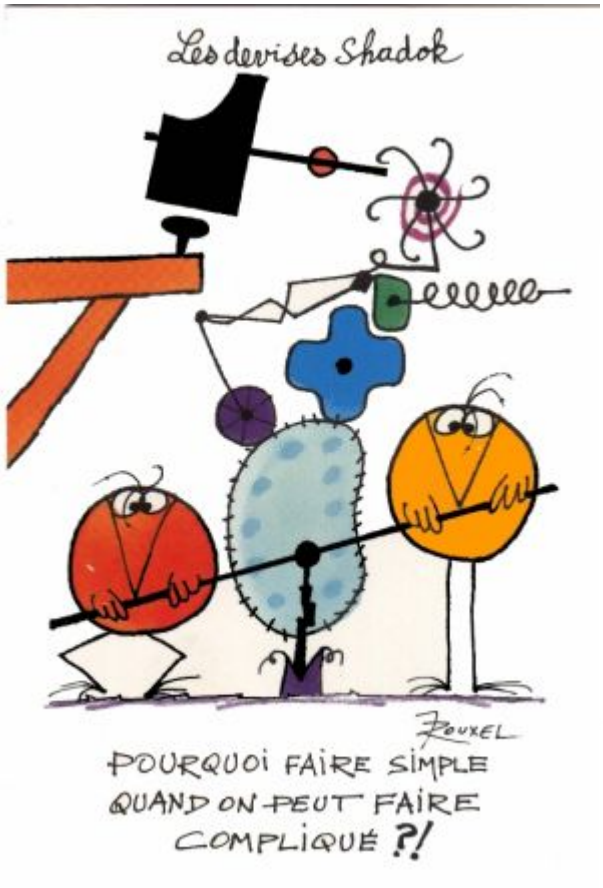
Exercice 9: faire l'exercice 6 en remplaçant T_{DN} par T_S.

Voyons ce que cela donne sur la formule de Jack Palmer:
(non A -> B \/ C) /\ (non B -> non A /\ D) /\ (D -> B \/ non C) -> B

```
                                                                                    ...
                                                                  non A -> B \/ C, non B -> non A /\ D |- C,
B
AxS ----------------------------------------   non_gauche ----------------------------------------
 non A -> B \/ C, non B -> non A /\ D, B |- B          non A -> B \/ C, non B -> non A /\ D, non C |-
B
ou_gauche ----------------------------------------------------------------------------------------
- ...
    non A -> B \/ C, non B -> non A /\ D, B \/ non C |- B      non A -> B \/ C, non B -> non A /\ D |-
D, B
 imp_gauche ---------------------------------------------------------------------------------------
-----
             non A -> B \/ C, non B -> non A /\ D, D -> B \/ non C |- B
  et_gauche ------------------------------------------------------------------
            (non A -> B \/ C), ( non B -> non A /\ D) /\ (D -> B \/ non C) |- B
  et_gauche ------------------------------------------------------------------
            (non A -> B \/ C) /\ ( non B -> non A /\ D) /\ (D -> B \/ non C) |- B
impl_droite ----------------------------------------------------------------------
            |- (non A -> B \/ C) /\ ( non B -> non A /\ D) /\ (D -> B \/ non C) -> B
```

Oups, la feuille est trop petite, continuez de votre coté, sur une feuille A3...

Comme ça faisait longtemps qu'il n'y avait plus d'image dans ce cours, en voilà deux d'un coup!



Contre-exemples.

Le calcul des séquents a un avantage: il donne explicitement les cas où une formule est fausse.

Essayons par exemple de démontrer la formule f = (non A -> non B) -> non (A -> B) :

```
        AxS --------                                          AxS --------
             A |- A          B, A |-                  |- B, A            B |- B
impl_gauche -----------------------    impl_gauche -----------------------------
              A -> B, A |-                              A -> B |- B
non_droite -----------------            non_gauche ------------------
            A -> B |- non A                          non B, A -> B |-
impl_gauche ------------------------------------------------------------------
                  non A -> non B, A -> B |-
  non_droite ---------------------------------
               non A -> non B |- non (A -> B)
impl_droite -------------------------------------
              |- (non A -> non B) -> non (A -> B)
```

On ne peut terminer la preuve, à cause de deux séquents: B, A |- et |- B, A, pour lesquels aucune règle ne s'applique.
En considérant que la preuve ne peut être terminée à cause d'eux, on peut légitimement penser que ces séquents représentent des cas où la formule de départ est fausse. En se souvenant qu'un séquent a le sens d'une implication (voir la définition de v(H |- G) plus haut), on voit qu'un séquent est faux uniquement quand ses formules de gauche sont toutes vraies et celles de droite sont toutes fausses. On en déduit que soit B et A sont vraies, par le séquent B, A |- , soit B et A sont fausses, par le séquent |- B, A. Et, de fait, ce sont exactement les deux cas où la formule (non A -> non B) -> non (A -> B) est fausse.

C'est vrai en général: les feuilles d'un arbre de preuve incomplet donnent exactement les cas où la formule de départ est fausse.

Exercice 10: déterminer les cas où les formules suivantes sont fausses:
- (A \/ B) /\ non (A /\ B)
- A1 -> (A2 -> ... -> (An -> A1) ...)
- A -> ((A -> B) -> ((B -> C) -> C))

III. Calcul des prédicats.

Reprenons le syllogisme de Socrate:

Tous les hommes sont mortels,
Socrate est un homme,
donc Socrate est mortel.

En le formalisant avec le calcul des propositions, on va distinguer 3 propositions:

A = tous les hommes sont mortels,
B = Socrate est un homme,
C = Socrate est mortel.

Le syllogisme devient A /\ B -> C, ce qui n'est évidemment pas toujours vrai...

Le langage du calcul des propositions n'est donc pas assez fin pour toute la logique ...

Le problème dans ce syllogisme, c'est de rendre compte de ce qui est commun entre A et B, c'est à dire le mot "homme", de ce qui est commun à B et C, qui est le mot "Socrate", et de ce qui est commun à A et C, qui est le mot "mortel". Voilà au minimum les atomes, les briques de bases du langage, qu'on doit utiliser.

Gottlob Frege.jpg

Il y a en outre des liens entre "homme", mortel" et "Socrate". C'est le verbe "être", qui les relie, attachant une *propriété* ("homme", "mortel") à un(des) objet(s) ("Socrate", "tous les hommes").

C'est ce langage des propriétés des objets que Gottlob Frege a formalisé, lui donnant le nom de calcul des prédicats.

Un prédicat est une fonction dont les valeurs sont "vrai" ou "faux", ou encore 1 ou 0 dans $\mathbb{Z}_2$. Elle prend en argument un objet, qui est soit une constante, soit un résultat de calcul sur d'autres objets.

Gottlob Frege

Dans le cas de Socrate, cela donne le prédicat "mortel", le prédicat "homme", l'objet "Socrate",
et les propositions B, C deviennent:
B = homme(Socrate)
C = mortel(Socrate)
Il reste un problème avec la proposition A, qui est le mot "tous". Il faut pouvoir parler d'une propriété valable pour tous les objets, ici, le fait qu'un objet qui est un homme est aussi mortel. On le fait en introduisant le quantificateur universel $\forall$[9] et une variable, qui désigne un objet quelconque: A = $\forall$x, si x est un homme, alors x est mortel.
La sous-phrase de A "si x est un homme, alors x est mortel" peut alors être formalisée naturellement avec le calcul des propositions, sous la forme C -> D:
C = x est un homme
D = x est mortel
et enfin, en utilisant de nouveau les prédicats "homme" et "mortel", on obtient le syllogisme complètement formalisé:

($\forall$x, homme(x) -> mortel(x)) $\wedge$ homme(Socrate) -> mortel(Socrate)

En fait, on utilise le langage mathématique moderne, à base de variables, de fonctions, de quantificateurs et d'implications!

Exercice 1: formaliser le sophisme

Tout ce qui est rare est cher,
un iphone bon marché est rare,
donc un iphone bon marché est cher.

Formalisons tout cela, maintenant.

---

9. notation avec un A à l'envers, qui vient de l'allemand "Alle" qui veut dire "tous", justement.

1. Syntaxe.

1.1. Alphabet.

On se donne un alphabet S, constitué de 4 parties:

- V = {x,y,...} un ensemble de variables, infini.
- S_f = {f,g,a,b,c,...} un ensemble de symboles de fonctions. Chaque fonction a une *arité*, qui est le nombre d'arguments qu'elle prend. Une constante est par convention une fonction d'arité 0.
- S_p = {P,Q,...} un ensemble de symboles de prédicats. Comme pour les fonctions, les prédicats ont une arité.
- S = {$\forall$, $\exists$, ",",->, $\wedge$, $\vee$, non,(,)} qui regroupe les quantificateurs universel et existentiel, la virgule, et les connecteurs du calcul des propositions.

Exemple: l'arithmétique
on va prend

$$S_f = \{+,-,*,0,s\}$$

avec + et * des fonctions binaires, - qui représente l'opposé, donc unaire, 0 une constante, et s une fonction unaire qui ajoute 1 à un nombre (i.e. qui rend son successeur). L'idée est de représenter 1 par s(0), 2 par s(s(0)), etc.[10]
Comme prédicats on va prendre

$$S_p = \{=, \leq, \geq\}$$

3 prédicats binaires, qui représentent l'égalité, et les relations d'ordres usuelles.

1.2. Termes.

Les termes sont les mots que l'on peut construire à l'aide des variables (V) et des fonctions (S_f), i.e. à l'aide des règles d'inférence suivantes:

$$Rv\ \frac{}{x}\ \text{si } x \in V \qquad Rf\ \frac{t1\ \ldots\ tn}{f(t1,\ldots,tn)}\ \text{si } f \in S_f \text{ et } f \text{ est d'arité } n$$

On note $T(S_f,V)$ l'ensemble des mots construits par ces règles.

Par exemple, voici un terme de l'arithmétique:

$$+(*(s(0),x), s(s(0)))$$

---

10. c'est la représentation de Peano des entiers naturels.

qui n'est autre que 1*x+2 en notation mathématique usuelle...

Si on fait de l'analyse, on ajoute à l'alphabet la fonction sinus, la racine carrée, la puissance, la division, la constante pi, etc...:

+(sin(/(π,3)),cos(/(π,3)))

qui représente sin π/3 + cos π/3

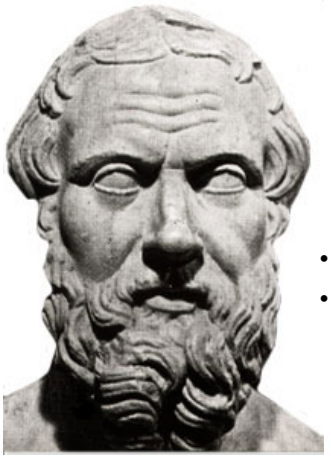
Exercice 1: donner le terme de l'analyse qui représente l'expression $\frac{1-a}{1-a^n}$

Les termes peuvent être vus aussi comme des arbres (informatiques): f(t1,...,tn) est l'arbre dont la racine est f et dont les sous-arbres sont t1...tn. C'est d'ailleurs comme cela qu'ils sont traités dans Maple, par exemple.

1.3. Formules atomiques.

Les termes jouent le rôle des objets sur lesquels vont porter les propriétés logiques. Ces propriétés logiques sont représentées par les symboles de prédicat, et formées par la règle suivante:

$$\mathrm{Rp}\ \frac{t1\ \ldots\ tn}{p(t1,\ldots,tn)}\quad \text{si } t1\ldots tn \text{ sont des termes, } p \in Sp \text{ et } p \text{ est d'arité } n$$



On les appelle "formules atomiques", non parce qu'elles sont explosives, mais parce qu'elles sont logiquement insécables, comme les atomes de Démocrite.

Exemples:

- =(+(s(0),s(0)),s(s(0))) ... qui représente 1+1=2
- homme(Socrate)

Démocrite

1.4. Formules du calcul des prédicats du premier ordre.

Elles sont formées par la règle Rp précédente, les règles suivantes:

$$\mathrm{Rq}\ \frac{f}{(\forall x, f)}\quad x \in V \qquad\qquad \mathrm{Re}\ \frac{f}{(\exists x, f)}\quad x \in V$$

et les règles du calcul des propositions:

```
       f            f    g          f    g          f    g
R1: -------    R2: --------   R3: --------   R4: --------
    (non f)        (f -> g)       (f /\ g)       (f \/ g)
```

Le syllogisme de Socrate peut maintenant être écrit comme une formule dans ce langage:

(∀x, homme(x) -> mortel(x)) /\ homme(Socrate) -> mortel(Socrate)

Autre exemple, voici une formule du calcul des prédicats pour l'arithmétique:

(∀n, (∃p, =(n,*(s(s(0)),p))) \/ (∃p, =(n,+(*(s(s(0)),p),s(0)))))

Exercice 2: la réécrire avec la syntaxe usuelle. Que signifie-t-elle?

On dit qu'on est au premier ordre, car les quantificateurs ne portent que sur les objets, et pas sur les propriétés: dans ce cas on serait au second ordre, comme par exemple avec le principe de récurrence sur les entiers naturels:

(∀P, P(0) /\ (∀n,P(n) -> P(s(n))) -> (∀n, P(n)))

qui n'est donc pas une formule du calcul des prédicats du premier ordre, mais du second ordre, car la première quantification porte sur P, qui est un prédicat.

On sent bien que toutes les mathématiques usuelles sont à notre portée avec ce langage du calcul des prédicats... C'est vrai, mais uniquement en partie, comme on va le voir plus tard.

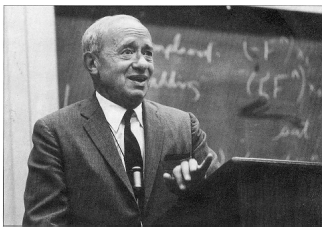
Exercice 3: écrire les axiomes de groupe en calcul des prédicats:
- $\forall\ x,y,z,\ x+(y+z)=(x+y)+z$
- $\forall\ x,\ x+0=0+x=x$
- $\forall\ x,\ x+(-x)=(-x)+x=0$

1.5. Variables libres, liées.

On dit qu'une variable est liée dans une formule, si elle apparaît après un quantificateur. Sinon, elle est libre. Plus formellement, on définit l'ensemble des variables libres d'une formule ainsi:
- Libres(f) = ∅ si f est atomique
- Libres(∀x, f) = Libres(f) - {x}
- Libres(∃x, f) = Libres(f) - {x}
- Libres(f -> g) = Libres(f /\ g) = Libres(f \/ g) = Libres(f) ∪ Libres(g)
- Libres(non f) = Libres(f)

Par exemple, Libres{=(x,x) /\ ∀x, =(+(y,x),y)} = {x,y}. En effet, les deux premiers x ne sont pas quantifiés, bien que le troisième le soit.
Une variable liée (comme le troisième x dans la formule précédente) peut être remplacée par une autre partout où elle est quantifiée: c'est ce qu'on appelle une variable muette en mathématiques. Une bonne discipline, pour éviter de confondre des variables muettes avec des variables libres, est de leur donner des noms différents. On écrira donc plutôt la formule précédente sous la forme =(x,x) /\ ∀z, =(+(y,z),y), dont on verra au paragraphe suivant qu'elle a le même sens.

2. Sémantique.



Alfred Tarski

La sémantique du calcul des prédicats n'est pas très surprenante, c'est juste une extension de celle du calcul des propositions avec les quantificateurs et la théorie des ensembles. Il suffit de dire comment on donne une valeur de vérité aux formules atomiques p(t1,...,tn):

Définition: *un modèle est un ensemble E non vide dans lequel on interprète les fonctions et les prédicats: à tout symbole f de S_f d'arité n, on associe une fonction f_E de E^n dans E, et à chaque prédicat P de S_p d'arité n, on associe une fonction P_E de E^n dans $\mathbb{Z}_2$.*
On définit l'interprétation $[f]_E$ d'une formule quelconque f sans variable libre dans le modèle E de la manière suivante:
- si f = p(t1,...,tn), $[f]_E = p_E([t1]_E,...,[tn]_E)$, sachant que l'interprétation d'un terme est définie par $[f(t1,...tn)]_E = f_E([t1]_E,...,[tn]_E)$.
- si f = ∀x, g, [f] = 1 ssi pour tout a dans E, $[g(x<-a)]_E = 1$, où g(x<-a) est la formule g dans laquelle on a remplacé x par a. Cette opération introduit des éléments de E dans les termes: on convient qu'ils s'interprètent en eux mêmes. Notons qu'on a pas besoin d'interpréter les variables, puisque la formule est supposée sans variable libre.
- si f = ∃x, g, [f] = 1 ssi il existe un a dans E, $[g(x<-a)]_E = 1$.
- si f = g -> h, $[f]_E = [g]_E([h]_E + 1) + 1$
- si f = g /\ h, $[f]_E = [g]_E[h]_E$
- si f = g \/ h, $[f]_E = [g]_E[h]_E + [g]_E + [h]_E$
- si f = non g, $[f]_E = [g]_E + 1$

Si f a des variables libres x1,...xn, alors on convient que $[f]_E = [\forall x1,...,\forall xn, f]_E$.

Si $[f]_E = 1$, on dit que *f est vraie dans le modèle E*. On le note aussi E |= f.

Par exemple, prenons comme modèle de l'arithmétique l'ensemble $\mathbb{Z}$, avec:
$+_{\mathbb{Z}}(x,y) = x+y$
$-_{\mathbb{Z}}(x) = -x$
$s_{\mathbb{Z}}(x)= x+1$
$0_{\mathbb{Z}} = 0$
$=_{\mathbb{Z}}(x,y) = 1 <=> x = y$
etc.

Dans ce modèle, la formule atomique =(+(s(0),s(0)),s(s(0))) est vraie. En effet, on a

$[=(+(s(0),s(0)),s(s(0)))]_{\mathbb{Z}}$
$= =_{\mathbb{Z}}([+(s(0),s(0))]_{\mathbb{Z}},[s(s(0))]_{\mathbb{Z}})$
$= =_{\mathbb{Z}}(+_{\mathbb{Z}}([s(0)]_{\mathbb{Z}},[s(0)]_{\mathbb{Z}}),s_{\mathbb{Z}}([s(0)]_{\mathbb{Z}}))$
$= =_{\mathbb{Z}}(+_{\mathbb{Z}}(1,1),s_{\mathbb{Z}}(1]_{\mathbb{Z}}))$ car $[s(0)]_{\mathbb{Z}} = s_{\mathbb{Z}}([0]_{\mathbb{Z}}) = s_{\mathbb{Z}}(0) = 1$
$= =_{\mathbb{Z}}(2,2)$
$= 1$

Par contre, choisissons le modèle singleton M = {a}, avec $=_M(x,y) = 0$ pour tout x et y.
Dans ce cas, on voit bien que =(+(s(0),s(0)),s(s(0))) est fausse.

Par exemple, dans le modèle $\mathbb{Z}$, la formule

$$\forall x, \text{ non } (x = x + 1)$$

est vraie.
Par contre, dans le modèle singleton M' = {a} avec $=_{M'}(x,y) = 1$ ssi x = y, elle est fausse, puisque x et x+1 sont forcément interprétés en a.

Exercice 4: trouver un modèle où la formule $\forall x, \exists y, x + y = y$ est vraie et un modèle où elle est fausse.

Définition: *on appelle tautologie toute formule vraie dans tous les modèles.*

Par exemple, $\forall x, x = x \rightarrow$ non (non (x = x)) est une tautologie[11]. En effet, remplaçons la formule atomique x = x par une variable propositionnelle A: on obtient A -> A, qui est une tautologie du calcul des propositions.

Autre exemple: la formule $(\exists y, \forall x, r(y,x)) \rightarrow (\forall x, \exists y, r(y,x))$ est une tautologie.
Ici, c'est plus subtil, car on ne peut se réduire au calcul des propositions. Pour le montrer, il suffit d'appliquer la définition, en procédant par l'absurde, ce qui est pratique quand on veut montrer une implication. Supposons qu'il y ait un modèle E tel que la formule soit fausse. C'est donc que $[\exists y, \forall x, r(y,x))]_E = 1$ et $[\forall x, \exists y, r(y,x)]_E = 0$.
Ainsi il existe un a dans E tel que $[\forall x, r(a,x)]_E = 1$. Et donc, pour tout b dans E, on a $[r(a,b)]_E = 1$. Mais alors, il existe un y dans E tel que $[r(y,b)]_E = 1$, c'est y = a. Donc $[\exists y, r(y,b)]_E = 1$. C'est vrai pour tout b dans E. Et donc $[\forall x, \exists y, r(y,x)]_E = 1$. Contradiction.

Exercice 5: montrer que la formule $(\forall x, p(x) \rightarrow q(x)) \wedge p(a) \rightarrow q(a)$ est une tautologie. En déduire que Socrate est mortel...

Définition: on dit que deux formules f et g sont équivalentes, noté f |=| g, lorsqu'elles sont vraies dans les mêmes modèles.

Exercice 6: montrer les équivalences suivantes:

---

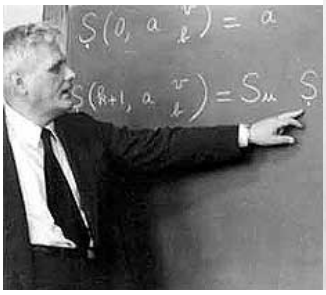
11. pourtant, $\forall x, x = x$ n'en est pas une (penser au modèle singleton M précédent).

non (∀x, f) |=| ∃x, non f
non (∃x, f) |=| ∀x, non f
∀x, ∀y, f |=| ∀y, ∀x, f
∃x, ∃y, f |=| ∃y, ∃x, f
∀x, f /\ g |=| (∀x, f) /\ (∀x, g)
∃x, f \/ g, |=| (∃x, f) \/ (∃x, g)

Contrairement à la sémantique au calcul des propositions, où l'on peut tester tous les cas pour savoir si une formule est toujours vraie, ici on ne peut tester tous les modèles: il y en a une infinité! Revenir à la définition, comme on l'a fait dans le dernier exemple, ne marche bien que pour les formules très simples... Y-a-t-il une méthode plus algorithmique?

NON!

Théorème (Church, Turing): *le calcul des prédicats du premier ordre est indécidable, ce qui signifie qu'il n'existe pas d'algorithme disant si une formule est une tautologie ou non.*



Alonzo Church *Alan Turing (mdr du bon coup qu'il a fait avec son théorème)*

MAIS...

Le bon côté des choses, c'est que les mathématiciens auront toujours du travail, puisque les machines ne peuvent résoudre toutes les mathématiques, les sottes...

Les mathématiciens ont assez rapidement mis au point des méthodes semi-décidables, c'est-à-dire qui, lorsqu'une formule est une tautologie, garantissent qu'elles vont en trouver une preuve (mais en général ces méthodes bouclent à l'infini si la formule n'est pas une tautologie). C'est pourquoi on va maintenant s'intéresser aux preuves dans le calcul des prédicats.

3. Preuves.

Comme pour la sémantique, on étend les règles de déductions du calcul des propositions pour traiter les quantificateurs.

3.1. Déduction naturelle.

Il suffit d'ajouter 2 règles:

```
          H |- f
qs_intro ------------ si x n'est pas libre dans H
          H |- ∀x, f

          H |- ∀x, f
qs_elim -------------- où f[x<-t] est f dans laquelle on a
         H |- f[x<-t]    substitué à x le terme t
```

et de traduire partout ∃x, f par non (∀x, non f).

Capture de variables.
La deuxième règle a une condition un peu technique qu'on a pas écrite, et qui est que le terme t n'introduit pas de *capture de variable* dans f: une variable de t ne doit pas devenir quantifiée une fois la substitution faite. Par exemple, si on substitue a+b à x dans ∀a, x+a = 0, on obtient ∀a, (a+b)+a=0, ce qui est idiot: le a de ∀a, x+c=0 est une variable muette qui n'a aucune raison d'être égale à celui de a+b; dans ce cas on renomme la variable muette: ∀a1, x+a1 = 0, et la substitution donne ∀a1, (a+b)+a1=0, ce qui est correct.

Nous pouvons (enfin) démontrer le syllogisme de Socrate, avec ces règles de la déduction naturelle. Pour abréger, notons homme par h, mortel par m et Socrate par S:

```
                                                    Ax ----------------------------------------------------------------------
                                                (∀x,h(x)->m(x))/\h(S) |-  (∀x,h(x)->m(x))/\h(S)
Ax ------------------------------------------------------  et_elim1 ------------------------------------------------------------
--------
(∀x,h(x)->m(x))/\h(S) |- (∀x,h(x)->m(x))/\h(S)                    (∀x,h(x)->m(x))/\h(S) |- (∀x,h(x)->m(x))
et_elim2 ------------------------------------------------------  qs_elim ------------------------------------------------------------
--------
(∀x,h(x)->m(x))/\h(S) |- h(S)                                     (∀x,h(x)->m(x))/\h(S) |- h(S)->m(S)
imp_elim ------------------------------------------------------------------------------------------------------------------------
----------
                                    (∀x, h(x) -> m(x)) /\ h(S) |- m(S)
                        imp_intro ----------------------------------------------
```

|- (∀x, h(x) -> m(x)) /\ h(S) -> m(S)

Cette méthode est satisfaisante, car elle est consistante, i.e. on ne démontre avec que des tautologies (c'est facile à montrer), et complète: toute tautologie est démontrable (difficile à démontrer).

Le premier à avoir démontré cela été Kurt Gödel, pour un système de déduction comparable (celui de Russell et Whitehead, le premier de tous):

Théorème (Gödel): *la déduction dans le calcul des prédicats est complète: toute tautologie est démontrable.*

Néanmoins, bien qu'on sache qu'une tautologie a toujours une démonstration, on sait qu'il n'existe pas d'algorithme qui nous dise si une formule est effectivement une tautologie... (voir plus haut).

Kurt Gödel

Exercice 1: trouver une preuve de la formule (∃y, ∀x, r(y,x)) -> (∀x, ∃y, r(y,x)).

Toutefois, comme avec le calcul des propositions, la règle du modus ponens rend les preuves difficiles à trouver, bien qu'elles soient assez naturelles à lire. Le calcul des séquents est bien plus adapté à la construction de preuves:

3.3 Calcul des séquents.

Reprenons les règles du calcul des séquents propositionnel:

```
AxS -----------------
     f, H |- f, G

 F_gauche -----------
           F, H |- G

            H |- f, G   g, H |- G                    f, H |- g, G
imp_gauche ----------------------      imp_droite ----------------
                f -> g, H |- G                      H |- f -> g, G

           f, g, H |- G                        H |- f, G   H |- g, G
et_gauche ----------------          et_droite -----------------------
           f /\ g , H |- G                      H |- f /\ g, G
```

```
          f, H |- G   g, H |- G                H |- f, g, G
ou_gauche ---------------------      ou_droite ----------------
             f \/ g, H |- G                    H |- f \/ g, G

             H |- f, G                           f, H |- G
non_gauche --------------            non_droite ---------------
            non f, H |- G                        H |- non f, G
```

et ajoutons la règle de coupure, et les règles qs_intro et qs_elim adaptées:

```
         f, H |- G    H |- f, G
Coupure ------------------------
                H |- G

           f[x<-t], H |- f, G
qs_gauche --------------------  t est un terme dont les variables libres ne
            (∀x,f) , H |- G     sont pas liées dans f

              H |- f, G
qs_droite -----------------  x est une variable non libre dans H
           H |- (∀x,f), G
```

De plus, on s'autorise à renommer les variables liées dans les formules.

Avec ce système, on démontre les mêmes théorèmes qu'avec la déduction naturelle, mais plus facilement. Par exemple, démontrons le syllogisme de Socrate:

```
       AxS -------------------    AxS -------------------

         h(S) |- h(S), m(S)       m(S), h(S) |- m(S)
imp_gauche ---------------------------------------------

               h(S) -> m(S), h(S) |- m(S)
     qs_gauche ------------------------------------ avec t = S
              (∀x,h(x)->m(x)), h(S) |- m(S)

et_gauche    ----------------------------------
             (∀x, h(x) -> m(x)) /\ h(S) |- m(S)
imp_droite ----------------------------------------
            |- (∀x, h(x) -> m(x)) /\ h(S) -> m(S)
```

La règle à appliquer est déterminée par la formule qu'on veut décomposer à droite ou à gauche du séquent à

prouver: c'est automatique. Sauf pour la règle qs_gauche, où on doit donner le bon terme t à substituer à x: et c'est cela précisément qui rend indécidable la déduction dans le calcul des prédicats!

Exercice 2: prouver que dans un bar, il existe toujours une personne qui, si elle boit, alors tout le monde boit. Autrement dit:

$$\exists x,\ b(x) \rightarrow (\forall x,\ b(x))$$

(rappel: traduire partout $\exists x$, f par non ($\forall x$, non f))
Indication: distinguer le cas où tout le monde boit du cas où il existe quelqu'un qui ne boit pas.

IV. Logique intuitionniste.

Reprenons l'exemple de Jack Palmer:

A = "Padmé a rencontré Dark Vador l'autre nuit"
B = "Dark Vador est le meurtrier"
C = "Padmé est une menteuse"
D = "le crime a eu lieu après minuit"

$$(\text{non } A \rightarrow B \vee C) \wedge (\text{non } B \rightarrow \text{non } A \wedge D) \wedge (D \rightarrow B \vee \text{non } C) \rightarrow B$$

On a démontré que cette formule est une tautologie. Jack Palmer sait donc que Dark Vador est le meurtrier, mais il ne sais pas quand a eu lieu le crime, ni comment... En fait, Jack Palmer n'a pas vraiment de preuve matérielle, il ne peut pas reconstituer le crime. Tout ça n'est pas très constructif, et l'erreur judiciaire n'est pas loin (qui sait, c'est peut-être Palpatine qui a fait le coup, finalement, au moment du changement d'heure...)!

C'est ce genre de doute qui pousse Luitzen Brouwer à développer l'intuitionnisme. Il se méfie du tiers-exclus:

$$A \vee \text{non } A$$

et des démonstrations par l'absurde:

$$(\text{non } A \rightarrow F) \rightarrow A$$

Ainsi que les preuves de "il existe un x tel que P(x)" qui ne construisent pas explicitement x. Et de l'axiome du choix (tout produit d'ensembles non vides est non vide).

Exemple.

Montrons qu'il existe des nombres a et b irrationnels[12] tels que a^b est rationnel. Considérons le nombre $\sqrt{2}^{\sqrt{2}}$:
- s'il est rationnel, alors a = $\sqrt{2}$ et b = $\sqrt{2}$ conviennent.
- s'il est irrationnel, alors a = $\sqrt{2}^{\sqrt{2}}$ et b = $\sqrt{2}$ conviennent. En effet $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\sqrt{2})} = \sqrt{2}^2 = 2$ qui est rationnel.
Cette démonstration est un peu frustrante... on ne sait pas quels a et b conviennent![13]

La logique propositionnelle intuitionniste est en gros la logique classique (celle qu'on vient de voir) moins le tiers-exclus et l'axiome du choix. On va s'intéresser dans ce cours seulement au calcul propositionnel intuitionniste (pas aux prédicats).
La sémantique de la logique intuitionniste est un peu compliquée. Il ne suffit plus de donner des valeurs de vérité, même en grand nombre (Gödel l'a démontré - aussi!-). Par contre, les preuves sont plus simples.

Il existe des liens assez simples entre les tautologies intuitionnistes et les tautologies classiques (celles du calcul des propositions):

Propriété: *toute tautologie intuitionniste est une tautologie classique.*

Il y a donc moins de choses vraies en logique intuitionniste qu'en logique classique. En pratique, faire des maths avec la logique intuitionniste, c'est un peu comme si on faisait un 100m pieds nus... Ça va moins vite, ça peut faire mal, mais on va aussi loin, et on ne dépend pas de Décathlon...[14]

Propriété: *f est une tautologie classique ssi non non f est une tautologie intuitionniste.*

Quelques exemples de tautologies classiques qui ne sont pas des tautologies intuitionnistes:

A \/ non A
non non A -> A
((A -> B) -> A) -> A (formule de Pierce)
non (A -> B) -> (B -> A)
Jack_Palmer
(((A -> B) -> C) -> (((B -> A) -> C) -> C))

1. Sémantiques.

1.1. Sémantique de Heyting.

[Wikipedia][ref]

On présente ici une version réduite (mais néanmoins aussi puissante) de la sémantique de Heyting.

---

12. un nombre est irrationnel quand il ne peut s'écrire comme une fraction de nombres entiers.
13. mais on peut montrer par ailleurs que $\sqrt{2}^{\sqrt{2}}$ est irrationnel. Ainsi c'est le deuxième cas qui convient, et la démonstration devient constructive.
14. cette comparaison vaut ce qu'elle vaut...

Au lieu d'interpréter les variables propositionnelles dans {0,1} on les interprète dans la topologie de $\mathbb{R}$, c'est-à-dire l'ensemble des ouverts de $\mathbb{R}$. L'ensemble vide joue le rôle de faux, et $\mathbb{R}$ lui-même joue le rôle de vrai. Les règles de calcul sont les suivantes, où i(X) dénote l'intérieur de X:

Définition: *une interprétation intuitionniste est une application de CP (les formules du calcul des propositions) dans l'ensemble des ouverts de $\mathbb{R}$ qui vérifie:*
a. v(non f) = i($\mathbb{R}$ - v(f))
b. v(f /\ g) = v(f) $\cap$ v(g)
c. v(f \/ g) = v(f) $\cup$ v(g)
d. v(f -> g) = i(($\mathbb{R}$ - v(f)) $\cup$ v(g))

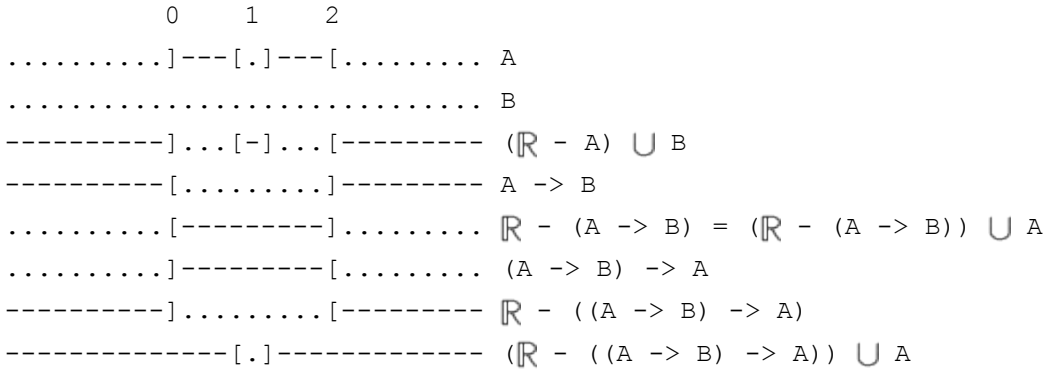
Définition: *une formule f est une tautologie intuitionniste ssi pour toute interprétation intuitionniste v, v(f) = $\mathbb{R}$.*

Exemples:

1. A \/ non A n'est pas une tautologie intuitionniste:
il suffit de trouver une interprétation qui donne autre chose que $\mathbb{R}$. Prenons v(A) =]0;+∞[.
On a v(A \/ non A) = v(A) $\cup$ v(non A)
=]0;+∞[$\cup$ i($\mathbb{R}$ - v(A))
=]0;+∞[$\cup$ i($\mathbb{R}$ -]0;+∞[)
=]0;+∞[$\cup$ i(]-∞; 0])
=]0;+∞[$\cup$]-∞; 0[
= $\mathbb{R}^*$
CQFD

2. non (A /\ non A) est une tautologie intuitionniste:
v(non (A /\ non A)) = i($\mathbb{R}$ - v(A /\ non A)) = i($\mathbb{R}$ - (v(A) $\cap$ i($\mathbb{R}$ - v(A))))
Or un ouvert et l'intérieur de son complémentaire sont toujours disjoints (puisque l'intérieur d'une partie est incluse dans cette partie). Donc v(non (A /\ non A)) = i($\mathbb{R}$) = $\mathbb{R}$
CQFD

3. la formule de Pierce ((A -> B) -> A) -> A n'est pas une tautologie intuitionniste:
v(A) =]0;1[$\cup$]1;2[
v(B) = $\emptyset$
Avec un dessin c'est plus clair:



```
                                = ((A -> B) -> A) -> A
```

4. non non A -> A n'est pas une tautologie intuitionniste: prendre le v(A) précédent.

```
          0    1    2
.........]---[.]---[......... A
----------]...[-]...[--------- (ℝ - A)
----------[.........]--------- non A
.........[---------]......... ℝ - non A
.........]---------[......... non non A
----------].........[--------- ℝ - non non A
--------------[.]------------- (ℝ - non non A) ∪ A
                                = non non A -> A
```

1.2. Sémantique de Kripke.

Cette sémantique est plus compliquée, mais elle a l'avantage de pourvoir faire intervenir des modèles finis.
Soit un ensemble W muni d'un pré-ordre (i.e. une relation $\leq$ réflexive et transitive),
et d'une application e associant à chaque variable propositionnelle de V un cône e(A) de W, i.e. une partie de W vérifiant: w $\in$ e(A) et w < w' => w' $\in$ e(A)
Pour tout w dans W, on définit l'interprétation $v_w(f)$ d'une formule de CP par:
$v_w(A) = 1$ ssi w $\in$ e(A)
$v_w(A \wedge B) = v_w(A)v_w(B)$
$v_w(A \vee B) = v_w(A)v_w(B) + v_w(A) + v_w(B)$
$v_w(\text{non } A) = 1$ ssi pour tout w' $\geq$ w, $v_{w'}(A) = 0$
$v_w(A \to B) = 1$ ssi pour tout w' $\geq$ w, si $v_{w'}(A) = 1$ alors $v_{w'}(B) = 1$

On a alors:

Théorème: *f est une tautologie intuitionniste ssi pour tout W, tout e et tout w, $v_w(f) = 1$.*

En fait on peut se réduire aux W finis avec un pré-ordre arborescent, dont les éléments sont les ensembles de parties de l'ensemble des sous-formules de f.

2. Preuves.

2.1. Interprétation de Brouwer–Heyting–Kolmogorov.

L'idée est simple: on voit une proposition logique comme **l'ensemble de ses preuves**.
Une preuve de f /\ g sera un couple (a,b), où a est une preuve de f, et b une preuve de g.
Une preuve de f \/ g est une preuve de f ou une preuve de g.
Une preuve de f -> g est une fonction p telle que pour toute preuve a de f, p(a) est une preuve de g.
F n'a pas de preuve: c'est l'ensemble vide.

La notion de fonction que l'on prend pour définir les preuves de f -> g n'est pas la notion classique d'application d'un ensemble dans un autre, mais celle de fonction **calculable** (par un algorithme, un programme, une machine de Turing, etc), ce qui restreint considérablement les preuves possibles (en particulier tout devient dénombrable).

2.2. Déduction naturelle intuitionniste.

[ref]

En logique intuitionniste, on ne peut traduire les connecteurs /\ et \/ en fonction de -> et F sans perdre leur sens. Il fat donc donner des règles pour ces connecteurs. Par contre, la négation non f reste traduisible par f -> F.

Les règles de déduction sont les suivantes:

```
                                          H |- F
Ax ----------                       F_elim --------
   f, H |- f                              H |- f

          H |- f   H |- f -> g                 f, H |- g
imp_elim ----------------------       imp_intro -------------
                H |- g                         H |- f -> g

          H |- f   H |- g            H |- f /\ g           H |- f /\ g
et_intro -----------------  et_elim1 ------------ et_elim2 ----------------
           H |- f /\ g                 H |- f                  H |- g

             H |- f                  H |- g
ou_intro1 ------------ ou_intro2 ------------
           H |- f \/ g             H |- f \/ g

           H |- f \/ g   f, H |- h   g, H |- h
ou_elim ---------------------------------------
                      H |- h
```

On peut faire le parallèle avec les tactiques de Coq:

Règle de déduction naturelle	Tactique de Coq
Ax	assumption

F_elim	assert False; contradiction
imp_elim	apply
imp_intro	intro
et_intro	split
et_elim1	decompose [and]
et_elim2	decompose [and]
ou_intro1	left
ou_intro2	right
ou_elim	decompose [or]

Théorème: *f est prouvable avec les règles de la déduction naturelle intuitionniste ssi c'est une tautologie intuitionniste.*

2.3. Calcul des séquents intuitionnistes.

La déduction naturelle intuitionniste fait encore intervenir le modus ponens, qui est compliqué car pour l'utiliser il faut inventer la formule f de ses hypothèses.
Comme dans le cas classique, on peut rendre plus algorithmiques les déductions, avec le calcul des séquents intuitionnistes:

Règles LJ de Gentzen [ref]:

```
Ax ----------                    Fg -----------
   A, H |- A                                F, H |- f

     f, g, H |- h                      H |- f   H |- g
etg ----------------             etd -----------------
     f /\ g, H |- h                       H |- f /\ g

      f, H |- h   g, H |- h              H |- f                 H |- g
oug ------------------------     oud1 -------------     oud2 -------------
         f \/ g, H |- h                  H |- f \/ g            H |- f \/ g

         H |- f                        f, H |- F
nong ---------------             nond ------------
      non f, H |- g                       H |- non f
```

```
      H |- f   g, H |- h                f, H |- g
impg --------------------         impd -------------
       f -> g, H |- h                    H |- f -> g
```

en restreignant aux connecteurs -> et F:

```
Ax ----------                     Fg -----------
   f, H |- f                          F ,H |- f

      H |- f   g, H |- h                 f, H |- g
impg ---------------------------  impd -------------
       f -> g, H |- h                     H |- f -> g
```

Théorème: f est une tautologie intuitionniste ssi elle est démontrable par le système LJ.

Théorème: *la logique intuitionniste est décidable.*

En effet, il suffit d'essayer d'appliquer les règles précédentes au séquent à prouver. Comme elle conduisent à des séquents plus petits (en nombre de symboles), ce processus termine forcément.

Exemple: montrons que non non A -> A n'est pas démontrable.
Pour montrer |- non non A -> A, une seule règle convient: impd, qui conduit à
non non A |- A
ici, une seule règle convient: nong, qui donne
|- non A
seule nond convient, et donne A |- F, qui n'est pas démontrable. CQFD

Par contre, A -> non non A est démontrable:

```
  Ax --------
      A |- A
nong -------------
      non A, A |-
nond ----------------
      A |- non non A
impd --------------------
       |- A -> non non A
```

On va maintenant étudier une formalisation de l'interprétation de Brouwer–Heyting–Kolmogorov (qui, rappelons-le, voit les propositions comme ensemble de leurs preuves): c'est le lambda-calcul. On le verra par la

suite, cette formalisation fait le lien avec la déduction naturelle et les programmes.

V. Lambda-calcul

Le lambda-calcul, dû à Alonzo Church, est fondé sur la notion de fonction mathématique:

$$x \longmapsto f(x)$$

Vous pouvez (devez...) regarder en vidéo le cours de Gérard Berry au Collège de France (il y a 3 cours sur le lambda-calcul, le premier est introductif, les autres sont plus poussés).

Les objets du lambda-calcul sont uniquement des fonctions. Ce qui est remarquable, c'est qu'avec une syntaxe minimale, on parvient à représenter tous les algorithmes, tous les programmes d'ordinateur, leurs propriétés, et plus généralement la notion de calculabilité. Il donne aussi une sémantique claire à la logique intuitionniste et, grâce à l'isomorphisme de Curry-Howard, à la notion de preuve formelle. C'est aussi en informatique ce qui a inspiré scheme, caml, et tous les langages de programmation fonctionnels.

1. Syntaxe.

Soit l'alphabet S = {λ , <virgule> , (,) , <espace> } $\cup$ V, où V = {x,y,z,..., x',x",...} est un ensemble infini de variables.
Les termes du lambda-calcul sont construits avec trois règles:

$$\text{Var: } \frac{}{x} \text{ si } x \in V \qquad \text{Abs: } \frac{x \quad t}{(\lambda x, t)} \qquad \text{App: } \frac{f \quad a}{(f\ a)}$$

Les termes construits avec la règle Abs sont appelés des "abstractions", ceux construits avec la règle App, des "applications".

Rien de très nouveau, en fait: en math classique, λx,t représente la fonction qui à x associe l'expression t: $x \longmapsto t$; et (f a) représente l'application de la fonction f à l'argument: f(a).

Dans la suite, pour éviter la lettre grecque λ, qui n'est pas sur le clavier (sauf en Grèce), on utilisera le caractère \ à la place de λ.

Exemples de \-termes:

- la fonction identité $x \longmapsto t$: c'est le terme I = (\x,x)

- la composition des fonctions: $f \longmapsto g \longmapsto (x \longmapsto f(g(x)))$:
c'est le terme o = (\f,(\f,(\x,(f (g x)))))

notez que c'est la même fonction que (f,g) ⟼ (x ⟼ f(g(x))): au lieu de prendre ses deux arguments d'un coup (avec un couple), elle les prend l'un après l'autre.

- la fonction qui applique son argument à lui-même: c'est le terme D = (\x,(x x))

Comme il y vite beaucoup de parenthèses, on adopte des conventions:
1. on enlève les parenthèses extérieures
2. on enlève même toutes les parenthèses... non, je blague! On les enlève quand:
3. elles sont à gauche des applications: (f a) b est noté f a b, mais f (a b) reste f (a b)
4. elles sont à droite du point: \f,(\x,(f x)) est noté \f,\x,f x
Ainsi:

I = \x,x
o = \f,\f,\x,f (g x)
D = \x,x x

2. Calcul: la béta-réduction.

Le \-calcul a deux côtés. Le côté lambda, on a vu. Reste le côté "calcul".
Là encore rien de nouveau: pour calculer f(a), lorsque par exemple f = x ⟼ x+1, on remplace x par a dans l'expression x+1, et on obtient a+1. C'est aussi ce qu'on fait pour évaluer une fonction dans un programme (Java par exemple): on remplace l'argument formel de la fonction par la valeur réelle dans la définition de la fonction.
Dans le \-calcul, cela revient à l'opération suivante:

(\x,t) a --> t[x<-a]

où t[x<-a] est le terme t dans lequel on a remplacé partout x par a.
On appelle ça la béta-réduction. Qu'on note avec la flêche -->

Calculons le terme (I a) avec cette définition:
I a = (\x,x) a --> x[x<-a] = a
autrement dit le résultat du calcul de(I a) est a. Normal, puisque I est l'identité!

Autre exemple:
o u v = [15] (o u) v = ((\f,\x,f (g x)) u) v --> (\f,\x,f (g x))[f<-u] v
= (\f,\x,u (g x)) v --> (\x,u (g x))[g<-v] = \x,u (v x) qui est bien la fonction u o v.

Calculons (D D):
D D = (\x,x x) D --> (x x)[x<-D] = D D = (\x,x x) D --> (x x)[x<-D] = D D = ...

---

15. quand les parenthèses sont omises dans une application ambigüe, c'est qu'elles sont à gauche.

voilà un calcul qui ne s'arrête pas, qui boucle, même!

Capture de variables.
Les \-termes contiennent des variables liées et de variables libres: comme pour les formules du calcul des prédicats: le symbole \ joue le rôle du symbole $\forall$, et on voit bien que le nom de la variable qui suit est muette, i.e. son nom n'importe pas. Par exemple on aurait pu définir l'identité par \y,y, ou \z,z, etc.
Lorsque l'on substitue une variable x par un \-terme a dans un terme t, on risque d'avoir une capture des variables libres de a par les \ de t.
Exemple: (\x,y x)[y<- x] = \x, x x = I, alors que (\z,y z)[y<- x] = \z, x z, qui est le bon résultat. Le x orange n'a pas de raison de devenir égal au x vert!

Il faut donc définir la substitution et la béta-réduction plus précisément pour éviter les captures:

Définition: *on définit la substitution t[x<-a] ainsi:*
- si t = a, t[x<-a] = a
- si t $\in$ V et t $\neq$ x, t[x<-a] = t
- si t = f b, t[x<-a] = f[x<-a] b[x<-a]
- si t = \x,b, t[x<-a] = t
- si t = \y,b et y $\neq$ x, t[x<-a] = \y,b[x<-a]

Définition: *on définit la béta-réduction par:*

(\x,t) a --> t[x<-a]

à condition que les variables libres de a ne soient pas liées dans t. Si ce n'est pas le cas, on renomme les variables liées de t pour les rendre différentes des variables libres de a.

Bien. Découvrons maintenant comment le \-calcul est un véritable langage de programmation.

3. Le \-calcul comme langage de programmation.

De quoi a-t-on besoin pour avoir un langage de programmation?
Tout d'abord, de calcul sur les entiers naturels. Ensuite de tests if...then...else. Puis de mémoire. Enfin de boucles for, while, récursion. Allons-y.

NB: les calculs de ce paragraphe, un peu fastidieux à la main, peuvent être faits en Coq: td6.v

3.1. Arithmétique.

L'idée de Chuch, pour représenter un entier naturel n est d'utiliser l'itération n fois d'une fonction quelconque : f $\mapsto f^n$

Définitions: *les entiers de Church sont*
0 = \f,\x,x

1 = *\f,\x,f x*
2 = *\f,\x,f (f x)*
...
n = *\f,\x,f (f ...(f x)...) où f est appliquée n fois.*

Addition: n+m, c'est $f \mapsto f^{n+m}$, mais $f^{n+m}(x)= f^n(f^m(x))$, d'où le \-terme suivant pour l'addition:
plus= \n,\m,\f,\x,(n f) ((m f) x) et, en enlevant les parenthèses inutiles:

plus = \n,\m,\f,\x,n f (m f x)

Exemple: calculons 2+3 (on surligne en vert les sous-termes réduits - les "rédex" -)

2+3 = plus 2 3 = (\n,\m,\f,\x,n f (m f x)) 2 3 --> (\m,\f,\x,2 f (m f x)) 3 --> \f,\x,2 f (3 f x)

= \f,\x,(\f,\y,g (g y)) f (3 f x) en renommant les variables liées de 2

--> \f,\x,(\y,f (f y)) (3 f x) --> \f,\x,f (f (3 f x))

= \f,\x,f (f ((\f,\y,g (g (g y))) f x)) --> \f,\x,f (f ((\y,f (f (f y))) x))

--> \f,\x,f (f (f (f x)))) = 5

On notera les successions de --> par -->*. Ainsi 2+3 -->* 5.

Multiplication: n * m, c'est $f \mapsto f^{nm}$, mais $f^{nm}= (f^n)^m$, d'où le \-terme suivant

mult = \n,\m,\f,m (n f)

Exemple: 2 x 3 = mult 2 3 = (\n,\m,\f,m (n f)) 2 3 --> (\m,\f,m (2 f)) 3 --> \f,3 (2 f)

= \f, 3 ((\f,\y,g (g y)) f) --> \f, 3 (\y,f (f y)) = \f,(\f,\z,g (g (g z))) (\y,f (f y))

--> \f,\z,(\y,f (f y)) ((\y,f (f y)) ((\y,f (f y)) z))

--> \f,\z,(\y,f (f y)) ((\y,f (f y)) (f (f z)))

--> \f,\z,(\y,f (f y)) (f (f (f (f z))))

--> \f,\z,f (f (f (f (f (f z))))) = 6

Puissance: n^m, c'est élever f à la puissance n m fois de suite, i.e. itérer m fois l'élévation à la puissance n: puis = \n,\m,\f,m (n f). Ou même, encore plus simplement:

puis = \n,\m,m n

Exemple: 2^3 = puis 2 3 = (\n,\m,m n) 2 3 --> (\m,m 2) 3 --> 3 2

= (\f,\x,f (f (f x))) 2 --> \x, 2 (2 (2 x)) = \x, 2 (2 ((\f,\y,f (f y)) x))

--> \x,2 (2 (\y,x (x y))) = \x,2 ((\f,\z,g (g z)) (\y,x (x y)))

--> \x,2 (\z, (\y,x (x y)) ((\y,x (x y)) z))

--> \x,2 (\z,(\y,x (x y)) (x (x z)))

--> \x, 2 (\z,x (x (x (x z))))

... --> 8 (complétez en exercice, j'en ai marre, je ne suis pas une machine...).

3.2. Tests.

Prenons les deux \-termes les plus simples, à par I, et nommons-les:

T = \x,\y,x
F = \x,\y,y

Soit maintenant **If = \c,\x,\y,c x y**

Le calcul de If T a b donne a, celui de If F a b donne b:

If T a b = (\c,\x,\y,c x y) T a b --> (\x,\y,T x y) a b --> (\y,T a y) b --> T a b --> (\x,\y,x) a b
--> (\y,a) b --> a

Exercice 1 : calculer If F a b.

Ainsi T joue le rôle de la valeur vraie et F celui de la valeur fausse!

Les connecteurs booléens sont facilement définissables, il suffit de les exprimer avec If par exemple. Leurs définitions ne sont pas uniques. En voici quelques unes:

and = \x,\y,x y F
or = \x,\y,x T y
imp = \x,\y,x y T
not x = \x,x F T

Exercice 2:

- montrez que and T F -->* F, and F T -->* F, and T T -->* T, and F F -->* F.
- trouver un \-terme pour l'équivalence logique.

3.3. Mémoire.

Soient **couple= \x,\y,\c,c x y**
gauche= \c, c T
droite= \c, c F

On vérifie facilement:

gauche(couplea b) -->* a
droite(couple a b) -->* b

Cons se comporte donc comme un constructeur d'arbre binaire, car et cdr comme les accès aux sous-arbres droit et gauche.

3.4. Récursion.

Il ne manque plus que la possibilité de définir des fonctions récursives, et on aura la puissance des ordinateurs avec le \-calcul. Prenons par exemple la fonction factorielle. On a envie d'écrire:

fact = \n,If (zero n) 1 (mult n (fact (pred n)))

en supposant qu'on a des \-termes zero et pred[16] tels que:
zero 0 -->* T et zero n -->* F si n <> 0
pred 0 -->* 0 et pred n -->* n-1 si n <> 0

mais fact apparaît des deux côtés de l'égalité. Il faudrait résoudre cette équation...
Si on considère la fonction Fact = \f,\n,If (zero n) 1 (mult n (fact (pred n)))

alors l'équation devient

Fact fact = fact

autrement dit, on cherche un point fixe de la fonction Fact.

Chance, il existe des \-termes qui donnent un point fixe à chaque fonction!

Par exemple, le point fixe de Church: **Y_C = \f,(\x,f (x x)) (\x,f (x x))**

Notons a <-->* b le fait que l'on puisse passer de a à b avec une suite de --> ou <--.

Propriété: *pour tout \-terme f, f (Y_C f) <-->* Y_C f*

---

16. voir plus loin la définition de zero et pred.

Démonstration:
Y_C f = (\f,(\x,f (x x)) (\x,f (x x))) f --> (\x,f (x x)) (\x,f (x x))
--> f ((\x,f (x x)) (\x,f (x x))) <-- f (Y_C f)
CQFD.

On peut maintenant définir la fonction fact comme point fixe de Fact:

fact = Y_C Fact = Y_C g\n,If (zero n) 1 (mult n (fact (pred n)))

Calculons fact 3:

fact 3 = Y_C Fact 3 --> Fact (Y_C Fact) 3 -->* If (zero 3) 1 (mult 3 (Y_C Fact (pred 3)))
-->* (mult 3 (Y_C Fact 2))
-->* (mult 3 (mult 2 (Y_C Fact 1)))
-->* (mult 3 (mult 2 (mult 1 (Y_C Fact 0))))
-->* (mult 3 (mult 2 (mult 1 1)))
-->* 6

Il nous reste à donner la définition de zero et pred:
- prenons **zero = \n,n (\y,F) T**: on itère n fois la fonction constante F en partant de T.
- soit **succ = \n,\f,\x,f (n f x)** la fonction qui ajoute 1 à son argument. L'idée pour définir pred n est d'itérer n fois la fonction (x,y) $\mapsto$ (y,y+1) en partant de (0,0): on tombe sur (n-1,n) et on prend la partie gauche:

pred = \n, (gauche (n (\p, couple (droite p) (succ (droite p)))
(couple 0 0)))

un peu compliqué quand même...

Bien, le lambda-calcul est un langage de programmation. Etudions maintenant ses propriétés.

4. Propriétés du lambda-calcul.

Le lambda-calcul jouit de 2 propriétés fondamentales: l'unicité des calculs et la possibilité de terminer les calculs quand c'est possible.
Précisons cela.

Théorème (confluence): *si t -->* t1 et t -->* t2 alors il existe t3 tel que t1 -->* t3 et t2 -->* t3*

Autrement dit, si on peut calculer un terme de deux façons différentes, on est assuré de pouvoir continuer les calculs pour obtenir le même résultat.
La démonstration de ce théorème est difficile.

Définition: *un \-terme est dit **irréductible** s'il ne peut se béta-réduire. Autrement dit, il ne contient pas de rédex.*

Définition: *un terme t est dit **normalisable** ssi il existe t' irréductible tel que t -->* t'*
t' est appelé "une forme normale de t".

Définition: *un terme t est dit **fortement normalisable** s'il n'existe pas de suite infinie de béta-réduction*
t --> t_1 --> t_2 --> ... ∞

Exemples:
- le terme D = \x,x est fortement normalisable, car il ne contient pas de rédex
- D D n'est pas normalisable, puisque D D ne contient qu'un seul rédex et que D D --> D D --> ... ∞
- le terme (\y,z) (D D) est normalisable, mais pas fortement normalisable.
en effet comme D D --> D D, (\y,z) (D D) --> (\y,z) (D D) --> ... ∞ et (\y,z) (D D) n'est pas fortement normalisable; comme (\y,z) (D D) --> z, (\y,z) (D D) est donc normalisable.

Corollaire (unicité des calculs): *si t est normalisable, alors sa forme normale est unique.*

Démonstration: en effet si t -->* t1 et t -->* t2 avec t1 et t2 irréductibles, alors il existe t3 tel que t1 -->* t3 et t2 -->* t3 par le théorème de confluence, et comme t1 et t2 sont irréductibles, c'est que t1 = t3 et t2 = t3, d'où t1 = t2
CQFD

Reste la question de savoir si un terme est normalisable. D'un coté, on peut montrer que cette question est indécidable: il est équivalent au problème de l'arrêt d'une machine de Turing (voir une preuve en annexe).
D'un autre côté, les termes intéressants sont souvent normalisables mais pas fortement. En fait, dès qu'ils contiennent une fonction récursive, puisqu'une fonction récursive est construite avec un opérateur de point fixe, qui n'est pas fortement normalisable.
Du moins, si un terme est normalisable, peut-on trouver sa forme normale?
La réponse est oui:

Propriété: *si t a une forme normale, alors on l'obtient en réduisant en choisissant toujours le rédex qui commence le plus à gauche.*

Appelons cette stratégie de réduction la "béta-réduction gauche".
Par exemple, pour calculer (fact 3), si on ne fait pas attention on peut réduire à l'infini, par exemple en choisissant de réduire des rédex qui se trouvent dans l'opérateur de point fixe. Par contre, en choisissant toujours le rédex le plus à gauche, on termine (voir l'exemple en 3.4.)

En fait ceci a une portée plus générale.
La béta-réduction gauche correspond à l'**appel par nom** en informatique, par opposition à l'**appel par valeur**: quand on doit calculer f(a) soit on choisit de calculer d'abord a, puis f, c'est l'appel par valeur, soit on choisit de calculer f puis a, c'est l'appel par nom, et c'est la béta-réduction gauche en \-calcul.
On peut aussi mettre ceci en parallèle avec le calcul numérique qui correspond à un appel par valeur, et le calcul formel, qui correspond à un appel par nom. L'appel par valeur a l'avantage de partager les calculs des arguments d'une fonction, donc favorise l'efficacité des calculs, alors que l'appel par nom garantit leur terminaison.

Pour montrer qu'un terme est fortement normalisable, on dispose de plusieurs techniques, on va en étudier une, qui offre de plus des avantages par elle-même hors du \-calcul: le typage.

VI. Types.

Dans ce chapitre, on va faire le lien entre logique et calcul, et donner les fondements du système Coq.

1. Types simples.

1.1. Introduction.

On a l'habitude, en mathématiques, de donner un ensemble de départ, et un ensemble d'arrivée aux fonctions:

$$f:\mathbb{R} \to \mathbb{R}$$
$$x \to \frac{1}{1+x^2}$$

L'expression $\mathbb{R} \to \mathbb{R}$ est appelé le "type" de la fonction f. Ici c'est celui des fonctions réelles d'une variable réelle.
On peut faire la même chose pour les \-termes. Par exemple l'identité aura pour type A -> A, pour tout type A.
La relation "a pour type" est notée ":"; ainsi on écrira \x,x : A -> A

En gros, un type est un ensemble, l'ensemble des objets dont c'est le type. Les lambda-termes sont essentiellement des fonctions, c'est pourquoi on va leur donner des types composés de flèches.

1.2. Encore des exemples:

T = \x,\y,x est une fonction qui prend un argument x, disons de type A, et qui rend la fonction constante \y,x. Donnons le type B à y, ainsi \y,x est de type B -> A, et T est de type A -> (B -> A).
De la même manière F = \x,\y,y sera de type A -> (B -> B). On peut choisir B = A, et dans ce cas, A->(A->A) est un type commun à T et F, et est donc candidat pour représenter le type des booléens.

Plus compliqué, essayons de donner un type à If = \b,\x,\y,b x y:
b doit être une fonction de deux arguments puisqu'il s'applique à x et y. Il a donc un type de la forme A -> B -> C. A doit être le type de x, B celui de y.
Ainsi If a le type (A -> (B -> C)) -> (A -> (B -> C)), qui est (A -> B -> C) -> A -> B -> C si on enlève les parenthèses inutiles.

Passons aux entiers de Church:
0 = \f,\x,x = F a pour type A -> B -> B
1 = \f,\x,f x: f doit être une fonction puisqu'elle s'applique à x, donc son type est de la forme A -> B, où A est le type de x. Ainsi le type de 1 est (A -> B) -> A -> B.
2 = \f,\x,f (f x): f doit être une fonction, disons de type A -> B, mais f doit s'appliquer à (f x) qui est de type B, donc A = B. Ainsi 2 a pour type (A -> A) -> (A -> A).
On vérifie que (A -> A) -> (A -> A) est un type possible pour tous les entiers plus grands que 2. Mais aussi pour 0: il suffit de remplacer A par (A -> A) et B par A dans le type de 0. De même pour 1, en prenant B = A.
Ainsi (A -> A) -> (A -> A) est un type possible pour tous les entiers: c'est un candidat pour représenter $\mathbb{N}$.

1.3. Règles de typage.

Formalisons maintenant cette notion de types.

Définition: *soit T_S l'ensemble $T(V_S,\{->\})$ des termes construits avec des variables de types V_S et le symbole ->, qui prend 2 arguments. C'est l'ensemble des* ***types simples****.*

Définition*: un environnement de typage est un ensemble fini de couples x:T, où x est une variable de \-terme et T un type.*

Définition*: un jugement de typage est un couple E |- t:T où E est un environnement de typage, t un \-terme et T un type simple.*

Règles de typage:

```
Var -----------------
      x:T, G |- x:T

    G |- f:A -> B   G |- a:A
App ------------------------
         G |- (f a): B

        x:A, G |- t:B
Abs ---------------------
     G |- (\x,t): A -> B
```

Définition: *on dit que t est de type T, noté t:T, ssi le jugement |- t:T a un arbre de preuve avec les règles Var, Abs, App.*

Par exemple, voici un arbre de preuve que \x,x est de type A -> A:

```
Var ------------
     x:A |- x:A
Abs ----------------
     |- \x,x:A -> A
```

de même pour \f,\x,f x : (A -> B) -> A -> B

```
Var --------------------------     Var ---------------------
     f:A -> B, x:A |- f:A -> B           f:A -> B, x:A |- x:A
App ------------------------------------------------------------
                  f:A -> B, x:A |- f x: A -> B
            Abs ----------------------------
                  f:A -> B |- \x,f x : A -> B
          Abs -----------------------------------
```

```
|- \f,\x,f x :  (A -> B) -> A -> B
```

Exercices: prouver que T:A->B->A, F:A->B->B, 2:(A->A)->A->A, If:(A -> B -> C) -> A -> B -> C

Tous les types qu'on a trouvé pour des \-termes ont une propriété remarquable...ce sont des tautologies!

Théorème: *si t:T, alors T est une tautologie.*

Démonstration: effaçons les \-termes des règles:

```
 -----------------
    T, G |- T

  G |- A -> B   G |- A
-------------------------
       G |- B

      A, G |- B
---------------------
    G |- A -> B
```

Que reste-t-il? Les règles Ax, imp_elim et imp_intro de la déduction naturelle!
Et on a vu en II.3.2 que les théorèmes de la déduction coincidaient avec les tautologies. CQFD.

Effaçons maintenant les types dans les règles:

```
-----------------
    x, G |- x

  G |- f   G |- a
-------------------------
       G |- (f a)

      x, G |- t
---------------------
   G |- (\x,t)
```

On retrouve les règles de formation des \-termes! (V.1.)

Ces trois règles de typage unifient le \-calcul et la déduction naturelle pour ce qui concerne le calcul des propositions restreint à l'implication. **La flèche des fonctions est identifiée à l'implication.** C'est d'ailleurs pourquoi on les note identiquement depuis le début du cours :-)
A chaque arbre de preuve de déduction naturelle correspond ainsi un \-terme et inversement: on peut donc considérer qu'un \-terme est une preuve de son type. Inversement, on peut considérer qu'une proposition est

l'ensemble de ses preuves, i.e. le type des \-termes qui l'ont pour type. Cette identification est appelée "correspondance formules/types", ou encore "isomorphisme de Curry-Howard".

C'est un exemple de cette aphorisme de Henri Poincaré:

La mathématique, c'est l'art de donner le même nom à des choses différentes

La correspondance formule/types dans 4 mondes:

Informatique	*Théorie des types*	*Logique*	*Mathématiques*
programme	lambda-terme	arbre de preuve	démonstration
spécification	type	formule	énoncé

Comme toutes les propositions ne sont pas des tautologies, **tous les types ne sont pas le type d'un \-terme** (on dit "habité"). Par exemple A -> B n'est pas un type de \-terme... du moins dans un environnement vide. Car on peut montrer le jugement b:B |- \x,b:A -> B.

Inversement, **tous les \-termes n'ont pas forcément un type.**
Prenons le terme D = \x,x x.; s'il a un type T, alors on peut montrer |- \x,x x : T.
Seule la règle Abs peut le prouver et à condition que T soit de la forme A -> B; on est alors conduit à prouver x:A |- x x: B, donc à utiliser la règle App; pour cela le x de gauche dans x x doit être d'un type de fonction qui s'applique au x de droite, qui est de type A dans l'environnement, donc de la forme A -> B, pour que l'on puisse ensuite montrer que x x est de type B. Ainsi on otient deux jugements à prouver: x:A |-x:A->B et x:A |-x:A .
On ne peut prouver x:A |- x:A -> B que si A = A -> B, avec la règle Var; mais on peut remplacer A et B par ce que l'on veut, A ne peut être égal à A -> B, il y a une flèche de plus à droite!
Le terme D n'est donc pas typable avec les types simples.

Comment savoir si un terme est typable? Comment alors trouver son type?

Voici une méthode, qui est emploie un algorithme fondamental en logique, l'unification, qu'on peut voir comme un cas particulier de l'algorithme de Gauss.

1.4. Algorithme d'unification.

Principe: pour donner un type à un terme, on construit un système d'équations sur des variables de types, et on en cherche une solution avec l'algorithme d'élimination de Gauss.

En fait il s'agit d'une méthode qui s'applique à la recherche de solution de systèmes d'équations sur des algèbres de termes du premier ordre quelconques (voir la définition en III.1.2.)

Définition: *soit T une fonction injective de l'ensemble des \-termes dans V_S, les variables de types. Notons T(t) par T_t. A chaque \-terme t non variable on associe une ou équation E(t) de la manière suivante:*

- si t = \x,u, E(t):= $T_t = T_x \rightarrow T_u$

- si t = (f a), E(t):= $T_f = T_a \rightarrow T_t$

On définit S(t) comme l'ensemble des équations des sous-termes non variables de t.

Exemple: calculons S(1)
les sous-termes non variables de \f,\x,f x sont \f,\x,f x, \x,f x, et f x
E(\f,\x,f x): $T_{\f,\x,f\ x} = T_f \rightarrow T_{\x,f\ x}$
E(\x,f x): $T_{\x,f\ x} = T_x \rightarrow T_{f\ x}$
E(f x): $T_f = T_x \rightarrow T_{f\ x}$
donc S(1) = { $T_{\f,\x,f\ x} = T_f \rightarrow T_{\x,f\ x}$
$T_{\x,f\ x} = T_x \rightarrow T_{f\ x}$
$T_f = T_x \rightarrow T_{f\ x}$ }

On résoud ensuite S(t) par la méthode de Gauss: on choisit une variable qu'on remplace partout par sa valeur et on recommence. On obtient alors un système triangulaire où les équations ont la forme X = U où X est une variable de type et U est un type sans les variables qui sont à gauche de ces équations; ou bien on échoue en trouvant une équation T = U où T est une variable et U est un type contenant la variable T et différent de T: dans ce cas l'équation n'a pas de solution. Dans le cas d'un succès, le type recherché pour t est le terme U de l'équation $T_t = U$.

Exemple: résolvons S(1)

la première équation donne la valeur de $T_{\backslash f,\backslash x,f\ x}$ en fonction des autres variables, et cette variable n'apparaît pas ailleurs, on passe à la seconde. La seconde équation donne la valeur de $T_{\backslash x,f\ x}$; on l'utilise pour remplacer partout ailleurs cette variable par sa valeur, le système devient:

$\{T_{\backslash f,\backslash x,f\ x} = T_f \to T_x \to T_{f\ x}$
$T_{\backslash x,f\ x} = T_x \to T_{f\ x}$
$T_f = T_x \to T_{f\ x}\}$

La troisième équation donne la valeur de T_f, qu'on remplace partout:

$\{T_{\backslash f,\backslash x,f\ x} = (T_x \to T_{f\ x}) \to T_x \to T_{f\ x}$
$T_{\backslash x,f\ x} = T_x \to T_{f\ x}$
$T_f = T_x \to T_{f\ x}\}$

Le système est maintenant triangulaire: les variables de gauche des équations n'apparaissent pas à droite. Le type de 1 est la valeur de T_1: $(T_x \to T_{f\ x}) \to T_x \to T_{f\ x}$, qui dépend de 2 paramètres, libres, T_x et $T_{f\ x}$. En renommant ces variables par A et B on retrouve le type qu'on avait donné à 1: (A -> B) -> A -> B.

Description formelle.

L'algorithme peut être décrit par des règles d'inférence:

Définition*: on appelle problème d'unification un couple* $\{X_1 = T_1, ...\}$ |- $\{U_1 = V_1, ...\}$
où les X_i *sont des variables de types distinctes, et* T_i, U_j, V_j *sont des types.*

Règles d'unification:

```
RU0 ----------
      E |- {}

      E |- S
RU1 -----------------
      E |- U = U, S

      E |- X = U, S
RU2 ----------------- X est une variable et U non
      E |- U = X, S

      E |- A = C, B = D, S
RU3 -------------------------
     E |- A -> B = C -> D, S

      X = U, G[X<-U] |- S[X<-U]
RU4 ---------------------------- si X est une variable
      E |- X = U, S              et n'apparaît pas dans U
```

Propriété: *un système S a une solution ssi on peut construire un arbre de preuve de {} |- S avec les 5 règles précédentes. La solution est donnée par les équations E dans la feuille de l'arbre.*

Exemple: reprenons le système S(1) = {$T_{\f,\x,f\ x} = T_f \to T_{\x,f\ x}$, $T_{\x,f\ x} = T_x \to T_{f\ x}$, $T_f = T_x \to T_{f\ x}$ }
on construit l'arbre de preuve suivant

$\{T_{\f,\x,f\ x} = (T_x \to T_{f\ x}) \to T_{\x,f\ x},\ T_{\x,f\ x} = T_x \to T_{f\ x},\ \ T_f = T_x \to T_{f\ x}\} \vdash \{\}$
RU4 ----------
$\{T_{\f,\x,f\ x} = \ T_f \to T_{\x,f\ x},\ \ T_{\x,f\ x} = T_x \to T_{f\ x}\} \vdash \{T_f = T_x \to T_{f\ x}\}$
RU4 ----------
$\{T_{\f,\x,f\ x} = T_f \to T_{\x,f\ x}\} \vdash \{T_{\x,f\ x} = T_x \to T_{f\ x},\ T_f = T_x \to T_{f\ x}\}$
RU4 ----------
$\{\} \vdash \{T_{\f,\x,f\ x} = T_f \to T_{\x,f\ x},\ T_{\x,f\ x} = T_x \to T_{f\ x},\ T_f = T_x \to T_{f\ x}\}$

Remarquons qu'on ne peut construire d'arbre de preuve infini: chaque règle fait décroître soit le nombre d'équations su système, soit la taille totale du système (nombre de symboles), soit la taille des mebres gauches des équations. Appliquer les règles tant qu'on peut donne donc un algorithme de résolution, et donc un algorithme de typage:

Propriété: *pour typer un \-terme, on tente de résoudre S(t) avec les règles d'unification. En cas de succès, le type de t est la valeur de T_t dans la feuille de l'arbre construit.*

Exercices:
- essayer de typer \x,x x
- typer couple = \x,\y,\c,c x y

1.5. Normalisation.

Théorème: *tout terme typable est fortement normalisable.*

Inversement, les termes fortement normalisables sont exactement les termes typables si on ajoute des types intersection, i.e. les règles de typage suivantes:

$$\text{et\_intro}\ \frac{\Gamma \vdash t:A \qquad \Gamma \vdash t:B}{\Gamma \vdash t:A \wedge B}$$

$$\text{et\_elimg}\ \frac{\Gamma \vdash t:A \wedge B}{\Gamma \vdash t:A}$$

$$\text{et\_elimd}\ \frac{\Gamma \vdash t:A \wedge B}{\Gamma \vdash t:B}$$

mais le typage avec ces règles devient indécidable, puisqu'il est indécidable de savoir si un terme est fortement normalisable...

On voit qu'il n'est pas facile de généraliser la correspondance formule/type à des formules qui n'utilisent pas seulement l'implication!

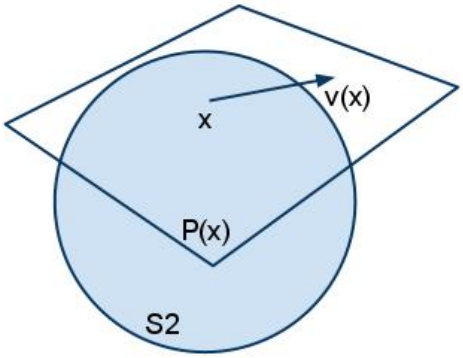
2. Types dépendants.

On va étendre la correspondance formule/type au calcul des prédicats, en étendant la notion de fonction: le type d'arrivée peut dépendre de l'argument de la fonction:

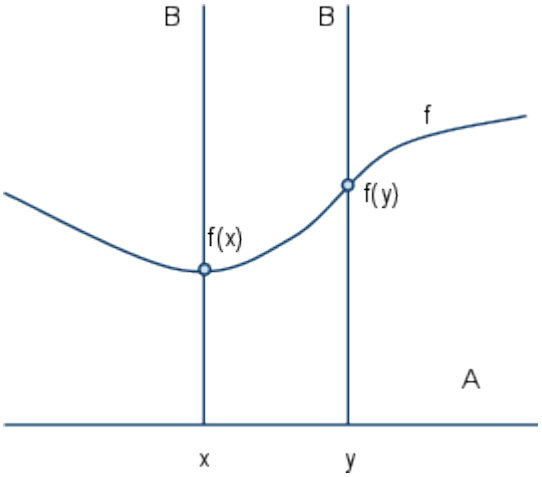
$$f{:}A \to B(x)$$
$$x \to f(x)$$

Par exemple, un champ de vecteur v sur la sphère S_2 est une fonction qui à un point x de S_2 associe un vecteur v(x) dans le plan tangent à S_2 en x:

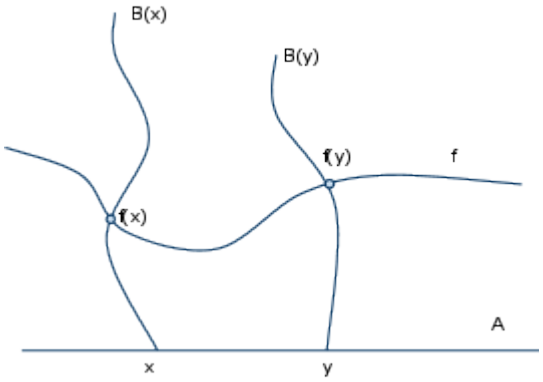
$$v{:}S_2 \to P(x)$$
$$x \to v(x)$$



Le graphe d'une fonction de A dans B est contenu dans le produit cartésien A x B:



Dans le cas de fonction de A dans B(x), il est contenu dans un espace fibré, qu'on appelle produit dépendant, ou **type dépendant**, qu'on va noter $\forall\, x{:}A,B(x)$ ou, pour employer les caractères du clavier, \/x:A,B(x). Le graphe de la fonction est alors une section de ce fibré:



C'est expliqué dans cette vidéo de V.Voevodsky (21mn-28mn précisément 00:25:40; la vidéo originale est ici, mais est lente à charger).
Lorsque B(x) ne dépend pas de x, alors on retrouve une fonction classique, de type A -> B.
Ainsi, la flèche peut être vue comme un cas particulier de la quantification:

Notation: *quand B ne dépend pas de x, \/x:A,B est noté A -> B.*

Replaçons-nous dans le contexte de la correspondance formule/type. L'air de rien, nous venons d'étendre les formules du calcul des propositions restreint à l'implication (les types simples), au calcul des prédicats (les types dépendants)!

Le **règles de typage** des \-termes deviennent alors:

```
VarD ----------------
      x:T, G |- x:T

     G |- f:\/x:A,B   G |- a:A
AppD ------------------------
          G |- (f a): B[x<-a]

        x:A, G |- t:B
AbsD ---------------------
      G |- (\x,t): \/x:A,B
```

En fait, il faut préciser les choses: les variables du \-calcul apparaissent dans les types (règle AbsD), ainsi que les \-termes eux-mêmes (substitution dans la règle AppD). Le langage des types et le \-calcul sont inclus dans un langage unique, et pour éviter des paradoxes, il faut se donner des règles plus précises. C'est le langage de Coq: le **calcul des constructions**. Mais les trois règles principales restent similaires à VarD, AppD, AbsD.

Avec ces règles, les **termes typables restent fortement normalisables**.

Traitons quelques exemples pour comprendre ce que sont les types dépendants.

Exemple 1.

Cherchons un terme de type \/A:B, A -> A, B étant un type. Cela revient à trouver un terme t tel que le jugement |- \/A:B, A -> A soit prouvable.
On peut construire un tel terme avec AbsD, ce qui conduit à trouver un terme t1 tel t = \A,t1, et prouver le jugement A:B |- t1:A ->A.
A->A est une notation pour \/y:A,A, donc on peut utiliser la règle AbsD, et chercher un terme t2 tel que t1 = \y,t2 et prouver le jugement A:B, y:A |- t2:A
Il suffit de choisir t2 = y, et la règle VarD, pour terminer la preuve.
Ainsi on a construit le terme t = \A,\y,y de type \/A:B, A->A, avec l'arbre de preuve:

```
VarD  ------------------
       A:B, y:A |- y:A
AbsD ---------------------------
       A:B |- \y,y:A -> A
AbsD ---------------------------
       |- \A,\y,A: \/A:B, A -> A
```

Tactique Coq et règles de typage.
En fait, la contruction de l'arbre de preuve est le travail fait par Coq: quand on modifie un but avec une tactique, Coq applique une règle de typage, et construit une partie d'un arbre de preuve. A chaque règle de typage correspond une tactique: Var est exact, AppD est apply, et AbsD est intro. A la fin de la preuve, Coq a construit un terme dont le type est le but de départ.

Exemple 2.
Prenons le syllogisme de Socrate: A et B sont types donnés, cherchons un terme t de type
\/h:A -> B,\/m:A -> B, \/ a:A, (\/x, h(x) -> m(x)) -> h(a) -> m(a).
La facilité consiste à utiliser la règle AbsD, qui dit qu'une abstraction a pour type un type dépendant. Mais la règle AppD, qui est une généralisation du Modus ponens, peut aussi donner pour type d'une application un type dépendant.
Cédons à la facilité: soit t = \h,t1 et cherchons t1 de type \/m:A -> B, \/ a:A, (\/x, h(x) -> m(x)) -> h(a) -> m(a).
De même, ce type est un type dépendant, cherchons donc t1 = \m, t2, avec t2 de type \/ a:A, (\/x, h(x) -> m(x)) -> h(a) -> m(a). En fait, à chaque quantification \/x:... correspond un terme \x,...
De même pour une implication A -> B, qui est une quantification non dépendante, correspond un terme \x,...
Ainsi on construit le terme t = \h,\m,\a,\H,\K, u, où h et m sont de type A->B, a de type A, H de type (\/x, h(x) -> m(x)), K de type h(a), et u un terme inconnu qui doit être de type m(a).
Comment contruire le terme u, de type m(a)? C'est un peu un jeu de Légo: on a à notre disposition les pièces suivantes:
h et m qui sont des fonctions de A dans B, a qui est un élément de A, H qui est une fonction de A dans le type dépendant h(x)->m(x), et K qui est un élément de h(a).
Assembler les pièces, c'est appliquer une fonction à un argument.
H peut s'appliquer à a, et (H a) est une fonction de h(a) dans m(a).
Pour obtenir un terme de m(a), il suffit alors de contruire ((H a) K).

On a donc trouvé t = \h,\m,\a,\H,\K,H a K ,
de type \/h:A -> B,\/m:A -> B, \/ a:A, (\/x, h(x) -> m(x)) -> h(a) -> m(a).

En Coq, on aurait écrit les tactiques: intros h m a H K. apply H. exact K.

Annexe

1. Le problème de l'arrêt.

Supposons qu'on a numéroté avec les entiers tous les programmes possibles, qu'un programme prend en entrée un entier ou deux et soit s'arrête et rend un entier, soit ne s'arrête pas. On représentera un programme par son numéro.

Théorème: (Turing) *il n'existe pas de programme qui prend un programme p, une entrée d,*
qui rend 1 si le programme p termine avec l'entrée d,
et rend 0 si le programme p ne termine pas avec l'entrée d.

Démonstration: par l'absurde. Soit t un tel programme.
Soit z le programme tel que z(0) termine et rend 1, et z(n) ne termine pas si $n \neq 0$.
Soit u le programme défini par

$$u(p) = z(t(p,p))$$

Supposons que u(u) termine:
u(u) termine => z(t(u,u)) termine => t(u,u) = 0 => u(u) ne termine pas: impossible car u(u) termine par hypothèse.

Supposons que u(u) ne termine pas:
u(u) ne termine pas => z(t(u,u)) ne termine pas => $t(u,u) \neq 0$ => t(u,u) = 1 => u(u) termine: impossible car u(u) ne termine pas, par hypothèse.

Conclusion: dans les deux cas, on a une contradiction, et donc le programme t ne peut exister. CQFD.