Autour de la logique

CSC_3INM3_TA

ENSTA Paris - TC 1ère année

François Pessaux

U2IS

2024-2025

prenom.nom@ensta-paris.fr

Préambule

Avant de commencer. . .

Moi == François Pessaux
U2IS, bureau R223

- Ne pas hésiter à poser des questions, même en cours.
- Posez votre question que vous jugez « bête » : d'autres ici ont la même.
- Ne pas hésiter à dire « *M'sieur, vous vous z'êtes pas trompé sur xyz ?* »
- Mon bureau et mon mail sont facilement accessibles.
- Ne pas hésiter à venir me voir, me parler. Si si !
- Terminez et conservez vos exercices, vos programmes.

Au boulot...

Vous en pensez quoi ?

- Si je suis à l'ENSTA alors je suis à Palaiseau.
- Je ne suis pas à l'ENSTA.
- Donc je ne suis pas à Palaiseau.

- Les cancérigènes nuisent à la santé.
- Le tabac nuit à la santé.
- Donc le tabac est un cancérigène.

- La démocratie est la voix de la majorité.
- Elle fait donc taire les minorités.
- On doit lutter contre l'écrasement des minorités.
- Donc on doit lutter contre la démocratie.
- Donc la dictature doit s'imposer.

Au boulot...

Vous en pensez quoi ?

Soient 2 entiers non nuls, x et y tels que $x = y$.

On en déduit :

$$
\begin{aligned}
x\,y &= y\,y \\
x\,y &= y^2 \\
x\,y - x^2 &= y^2 - x^2 \\
x\,(y - x) &= (y + x)\,(y - x) \\
x &= y + x \\
x &= x + x \\
x &= 2\,x \\
1 &= 2
\end{aligned}
$$

□

Au boulot...

```
let rec all_1s l =
  match l with
  | [] -> true
  | h :: q -> if h = 1 then all_1s q else false

let rec sum_list l =
  match l with
  | [] -> 0
  | h :: q -> h + sum_list q

let rec length l =
  match l with
  | [] -> 0
  | h :: q -> 1 + length q
```

Est-ce que (et pourquoi si oui) pour toute liste `l`, si `all_1s l` retourne `true` alors `sum_list l = length l` ?

Pourquoi dois-je subir des cours de logique ?

- Autodéfense intellectuelle pour les discussions de tous les jours.
  - Détection des arguments fallacieux.
  - Savoir les manipuler pour convaincre abusivement ? `:D`
- Pour faire des preuves de maths plus rigoureuses.
- Pour faire des programmes en lesquels on a (plus) confiance.
- Pour comprendre comment l'on raisonne en tant qu'être humain, voire scientifique.

Avant / Maintenant

- Vous avez déjà vu :
  - Le calcul propositionnel.
  - Peut-être un peu de calcul des prédicats.
  - La déduction naturelle comme règles de déduction.
- Buts de ce cours :
  - Réviser les notions déjà vues en prépa.
  - Enrichir petit-à-petit l'expressivité de la logique.
  - Formaliser des systèmes logiques.
  - Faire le lien entre logique et programmation.
  - Programmer (en OCaml).
  - Démontrer (à la main, puis avec des outils).
  - Automatiser des démonstrations.

Modalités d'évaluation des connaissances

- On part sur un examen de 3h max.
- A priori :
  - un peu de démonstrations sur papier,
  - un peu de Coq sur machine,
  - un peu de cours.
- En tout cas, des choses similaires à ce que l'on aura fait ensemble.

Quelques rappels - Les preuves informelles

Énoncés semi-formels

« Démontrer que si x et y sont impairs alors $x + y$ est pair. »

Supposons que x et y sont impairs.

Il existe donc des entiers m et n tels que : $x = 2\ m + 1$ et $y = 2\ n + 1$

On a donc : $x + y = 2\ m + 1 + 2\ n + 1 = 2\ m + 2\ n + 2 = 2\ (m + n + 1)$

Donc $x + y$ est pair.

- Structure des formules parfois évasive.
  - « si, alors », « et », « il existe ».
  - x et y viennent d'où ?
  - Ils sont quoi d'ailleurs ?
- Forme des démonstrations parfois approximative.
  - « il existe donc », « on a donc ».
  - Pour quelles raisons « bien connues » ?
  - Pourquoi choisir une forme de démonstration ?

Quelques schémas usuels macroscopiques de preuves

- Preuve directe : démontrer la proposition en partant des hypothèses et en arrivant à la conclusion par un enchaînement d'implications logiques.
- Preuve par contraposée : pour démontrer « si A alors B », on démontre « si non B alors non A ».
- Preuve par l'absurde : pour démontrer « A », supposer « non A », démontrer que contradiction, donc en déduire que « A ».
  - **Attention** : valable seulement en logique classique.
  - **Différent de** : pour démontrer « non A », supposer « A », démontrer que contradiction, donc en déduire que « non A ».
- Preuve par cas : pour montrer « si A ou B, alors, C », on montre que « si A alors C » et « si B alors C »
- Preuve par récurrence : pour démontrer une formule $P(n)$ pour tout n entier.
  - Montrer $P(0)$.
  - Sous l'hypothèse que $P(n)$ est « vraie », démontrer $P(n+1)$.
- ...

Formes de preuve (1/3)

Prouvons A et B
- Prouvons A
  - ...
- Prouvons B
  - ...

Prouvons A ou B
- Prouvons A
  - ...

Prouvons A ou B
- Prouvons B
  - ...

Prouvons si A alors B
- Supposons $H : A$
- Prouvons B
  - ...

Prouvons A si et seulement si B
- Prouvons si A alors B
  - ...
- Prouvons si B alors A
  - ...

Formes de preuve (2/3)

Prouvons non A
- Supposons $H : A$
- Prouvons une contradiction
  - ...

Prouvons il existe x tel que $P(x)$
- Exhibons un témoin $\overline{x}$
- Prouvons $P(\overline{x})$
  - ...

Prouvons pour tout x, $P(x)$
- Supposons H : un x quelconque
- Prouvons $P(x)$
  - ...

Prouvons A
- Par hypothèse H si $H : A$

Formes de preuve (3/3)

- Et quand le but n'est pas dirigé par un connecteur particulier ?
- Il faut jouer avec les hypothèses.
- Choix moins automatique, souvent moins évident.
- Nécessite de l'intuition.
- Justement, on formalisera tout cela plus tard.
- Par exemple :

  Prouvons A

  Prouvons A et B

  ...
- Il faut imaginer le B qui n'est pas dans le but à prouver :-/

Quelques rappels - Calcul propositionnel

Décrire les propositions : syntaxe

- Langue naturelle : trop ambiguë.
- Ensemble infini dénombrable de variables propositionnelles (atomes) : $\mathcal{P} = \{ A, B, C \dots \}$.
- Alphabet : $\mathcal{P} \cup \{ \neg, \wedge, \Rightarrow, \Leftrightarrow, (,) \}$
- Formule :
  - Toute variable de $\mathcal{P}$ est une formule.
  - Si P est une formule, alors $\neg P$ et (P) sont des formules.
  - Si P_1 et P_2 sont des formules, alors
    $$P_1 \wedge P_2 \qquad P_1 \vee P_2 \qquad P_1 \Rightarrow P_2 \qquad P_1 \Leftrightarrow P_2$$
    sont des formules.
- Priorité, associativité :
  - $\neg \; > \; \wedge \; > \; \vee \; > \; \Rightarrow \; > \; \Leftrightarrow$
  - $A \Rightarrow B \Rightarrow C \quad \equiv \quad A \Rightarrow (B \Rightarrow C)$.
  - $A \Leftrightarrow B \Leftrightarrow C \quad \equiv \quad A \Leftrightarrow (B \Leftrightarrow C)$.
  - Si vous avez des doutes : mettez des parenthèses !
  - Si ce n'est pas limpide : mettez des parenthèses !

C'est quoi être vrai ?

- Variables propositionnelles ne sont pas intrinsèquement vraies ou intrinsèquement fausses.
- Se détacher de l'intuition qui donne des valeurs de vérité à un fait.
- « *Il pleut* » n'a pas de valeur de vérité a priori :
  - peut être vrai,
  - peut être faux.
- « Prouver » (ou plutôt, démontrer) une formule :
  - argumenter qu'elle est toujours vraie,
  - quelles que soient les valeurs de vérité de ses atomes.
- Une formule démontrée s'appelle une tautologie.
- Ce que l'on fait quand « on démontre un théorème ».

Ajouter de la sémantique à la syntaxe

- Donner une signification aux connecteurs logiques.
- Deux valeurs logiques : « vrai » ($\top$), « faux » ($\bot$).
  - $\mathbb{B} = \{\bot, \top\}$
- Tables de vérité :

P	Q	$\neg_{\mathbb{B}} P$	$P \wedge_{\mathbb{B}} Q$	$P \vee_{\mathbb{B}} Q$	$P \Rightarrow_{\mathbb{B}} Q$	$P \Leftrightarrow_{\mathbb{B}} Q$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

- Assigner une valeur logique à chaque « atome ».
- Pour plus tard, $\wedge_{\mathbb{B}}$, $\vee_{\mathbb{B}}$, etc. : feront référence au comportement des tables de vérité.

Définition (Valuation, assignation)

Une valuation ν est une application de l'ensemble des variables d'une formule dans $\mathbb{B} = \{\bot, \top\}$.

$\nu : \mathcal{P} \mapsto \mathbb{B}$

Quand une formule est-elle toujours vraie ?

- « Évaluer » la formule pour toutes les valuations possibles
  - utiliser $\wedge_{\mathbb{B}}$, $\vee_{\mathbb{B}}$ etc.
- Si valeur toujours $\top$: formule toujours vraie
  « démontrée vraie ».
- Vérification : simple calcul.

- Revient à écrire un petit évaluateur de formules logiques.
- La sémantique des connecteurs est donnée par les tables de vérité.
- Environnement d'évaluation : valuation.
- Notation : $[\![P]\!]_\nu : \mathcal{F} \mapsto (\mathcal{P} \mapsto \mathbb{B}) \mapsto \mathbb{B}$

Définitions

Définition (Formule valide, tautologie $\vdash P$)

La formule P est une tautologie (ou dite valide) si et seulement si sa « valeur » est $\top$ pour toute valuation ν, i.e. si $\forall \nu, [\![P]\!]_\nu = \top$.

Définition (Modèle $\models_\nu P$)

Une valuation ν est un modèle d'une formule P si et seulement si elle lui donne la valeur $\top$, i.e. ssi $[\![P]\!]_\nu = \top$.

Au boulot...

Vérifiez si la formule $(A \Rightarrow (B \Rightarrow C)) \Rightarrow (B \Rightarrow A) \Rightarrow B \Rightarrow C$ est une tautologie.

Notez les parenthèses superflues autour de $B \Rightarrow C$. Mais c'est un peu plus lisible, quand même, non ?

Au boulot...

Récupérez l'archive `01_brute_force.zip` depuis le Moodle.

Elle contient des analyseurs lexical et syntaxique, permettant de construire une représentation de formules lues depuis un fichier ou l'entrée standard.

1. Renommez le fichier `core-eleves.ml` en `core.ml`
2. Invoquez la commande `touch .depend`
3. Invoquez la commande `make depend`
4. Invoquez la commande `make` pour finalement compiler.
5. Testez l'exécutable obtenu en lançant `./bf.x` et en saisissant des formules terminées par un `.` final.

Syntaxe :

« $\wedge$ »	« $\vee$ »	« $\neg$ »	« $\Rightarrow$ »	« $\Leftrightarrow$ »
/\	\/	~	->	<->

Au boulot...

Le type OCaml représentant les formules issues de l'analyse syntaxique est `expr_t` du fichier `ast.ml`.

```
type var_t = string
type expr_t =
  | E_And of (expr_t * expr_t)
  | E_Or of (expr_t * expr_t)
  | E_Imply of (expr_t * expr_t)
  | E_Equiv of (expr_t * expr_t)
  | E_Not of expr_t
  | E_Atom of var_t
  | E_True
  | E_False
```

Dans le fichier `core.ml`, complétez la fonction `eval` qui prend en argument une valuation `v`, une formule logique `e` et retourne la valeur booléenne de `e` dans `v`.

On représentera $\top$ et $\bot$ par les booléens d'OCaml.

Au boulot...

On souhaite écrire une fonction `enumerate` qui énumère toutes les valuations possibles pour une liste de variables prise en argument.

On représentera $\top$ et $\bot$ par les booléens d'OCaml.

Si l'on fait abstraction des variables en elles-mêmes, à quoi ressemble ce problème ?

Au boulot...

Écrivez la fonction `enumerate` en question. Pour le moment, elle pourra se contenter d'afficher chaque valuation.

Indication : Le module `Printf` d'OCaml vous met à disposition une fonction `printf` similaire à celle de C.

```
$ rlwrap ocaml
# Printf.printf "Foo: %d %f %b %s\n" (40 + 2) 3.14 true "bar" ;;
Foo: 42 3.140000 true bar
- : unit = ()
```

Au boulot...

Complétez la fonction `find_atoms` qui retourne la liste des variables propositionnelles présentes dans la formule passée en argument.

Modifiez la fonction `enumerate` que vous avez précédemment écrite pour qu'elle vérifie si la formule est une tautologie.

Complétez le point d'entrée `start` pour qu'il lance la vérification et affiche le verdict final.

Au boulot...

Testez votre prouveur avec différentes formules.

Testez avec `A1 -> A2 -> ... -> A20 -> A20.`

Testez avec `A1 -> A2 -> ... -> A25 -> A25.`

Qu'en concluez-vous ?

Vers le raisonnement

Vers une première modélisation du raisonnement

- Idée : essayer de faire le tour des déductions « élémentaires » que l'on fait naturellement.
- Doivent être des formules valides.
- On les pose comme axiomes : admises sans démonstration.
- ...
- « *Si A et B sont vraies, alors A est vraie.* »
- « *Si A et B sont vraies, alors B est vraie.* »
- « *Si A est vraie, si B est vraie, alors A et B sont vraies.* »
- « *Si A est vraie, alors A ou B sont vraies.* »
  Tiens, on parle en conclusion de B qui n'apparaît pas dans « l'hypothèse ».
- ...

Axiomes de Frege-Hilbert pour le calcul propositionnel

- Sont des arguments finaux de démonstration.
- Doivent bien entendu être valides.

$A \Rightarrow (B \Rightarrow A)$ $\quad (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$ $\quad A \wedge B \Rightarrow A$

$A \wedge B \Rightarrow B$ $\quad A \Rightarrow B \Rightarrow (A \wedge B)$ $\quad A \Rightarrow (A \vee B)$ $\quad B \Rightarrow (A \vee B)$

$(A \vee B) \Rightarrow ((A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow C))$ $\quad A \Rightarrow (\neg A \Rightarrow \bot)$

$(A \Rightarrow \bot) \Rightarrow \neg A$ $\quad \bot \Rightarrow A$ $\quad (A \Leftrightarrow B) \Rightarrow (A \Rightarrow B)$ $\quad (A \Leftrightarrow B) \Rightarrow (B \Rightarrow A)$

$(A \Rightarrow B) \Rightarrow ((B \Rightarrow A) \Rightarrow (A \Leftrightarrow B))$ $\quad (A \vee \neg A$ en logique classique$)$

Au boulot...

Ça y est, on est bons ?

Avec les axiomes de Frege-Hilbert ci-dessus pour la logique propositionnelle, démontrez la formule $P \Rightarrow P$.

Les règles de déduction

- Idée : essayer de faire le tour des déductions « élémentaires » que l'on fait naturellement.
- Doivent être des formules valides.
- On les pose comme axiomes.
- . . .

- Manque de quoi déduire de nouvelles formules à partir d'existantes.
- Rajout de règle(s) de déduction.
- Frege-Hilbert : 1 règle, modus ponens.

$$\frac{A \Rightarrow B \qquad A}{B}\,MP$$

- On verra plus tard les questions de correction.

Preuve de $P \Rightarrow P$ dans Frege-Hilbert

$$\dfrac{\dfrac{\dfrac{}{(P \Rightarrow ((P \Rightarrow P) \Rightarrow P)) \Rightarrow ((P \Rightarrow (P \Rightarrow P)) \Rightarrow (P \Rightarrow P))}\,Ax\ \substack{[A \leftarrow P;\\ B \leftarrow (P \Rightarrow P);\\ C \leftarrow P]} \qquad \dfrac{}{P \Rightarrow ((P \Rightarrow P) \Rightarrow P)}\,Ax.\ \substack{[A \leftarrow P;\\ B \leftarrow (P \Rightarrow P)]}}{(P \Rightarrow (P \Rightarrow P)) \Rightarrow P \Rightarrow P}\,MP \qquad \dfrac{}{P \Rightarrow (P \Rightarrow P)}\,Ax\ \substack{[A \leftarrow P;\\ B \leftarrow P]}}{P \Rightarrow P}\,MP$$

Démontrez $(A \Rightarrow (B \Rightarrow C)) \Rightarrow (B \Rightarrow A) \Rightarrow B \Rightarrow C$ avec les règles à la Frege-Hilbert.

Vous avez 3 heures `:-D`

Remarques sur Frege-Hilbert

- Beaucoup d'axiomes.
- Très peu de règles de déduction.
- Assez éloigné de la manière dont on démontre « dans la vie de tous les jours ».
- Preuve compliquée pour un énoncé trivial.
- Prouver $P \Rightarrow P$:
  1. Supposons P, prouvons P.
  2. Trivial car P dans mes hypothèses.
- Envie d'introduire la notion d'hypothèses au cours des déductions.
- Envie de réduire le nombre d'axiomes à connaître par cœur.

La déduction naturelle pour la logique propositionnelle

Objectifs

- Réduire le nombre d'axiomes à 1 (2 en logique classique).
- Privilégier des règles de déduction.
- Hypothèses font partie des règles de déduction.
  - Permettent d'introduire des hypothèses.
  - Permettent d'exploiter des hypothèses.
- Introduite par Gerhard Gentzen en 1934.

Définition (Séquent)

Un séquent est une paire $\Gamma \vdash P$ où Γ représente les hypothèses, i.e. une liste de propositions supposées « vraies ».

Forme des règles de déduction

Définition (Règle de déduction en déduction naturelle)

$$\frac{\Gamma_1 \vdash P_1 \qquad \dots \qquad \Gamma_n \vdash P_n}{\Gamma \vdash P}\ nom$$

- Au-dessus de la barre : prémisse(s), en dessous : conclusion.
- Lecture :
  - si sous les hypothèses Γ_1 la formule P_1 est démontrable,
  - et ...
  - et sous les hypothèses Γ_n la formule P_n est démontrable,
  - alors sous les hypothèses Γ la formule P est démontrable.
- Ou bien :
  - pour démontrer P sous les hypothèses Γ, il faut
  - démontrer P_1 sous les hypothèses Γ_1
  - et ...
  - démontrer P_n sous les hypothèses Γ_n
- Axiome : règle de déduction sans prémisse.

Règles de la déduction naturelle (logique propositionnelle)

$$\frac{\text{si } A \in \Gamma}{\Gamma \vdash A}\ \text{Axiome} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\ \text{intro} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\ \text{elim} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\ \text{elim}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\ \text{intro} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\ \text{intro} \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\ \text{elim}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\ \text{intro} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\ \text{elim} \qquad \frac{\Gamma, A \vdash B \quad \Gamma, B \vdash A}{\Gamma \vdash A \Leftrightarrow B}\ \Leftrightarrow\ \text{intro}$$

$$\frac{\Gamma \vdash A \Leftrightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \Leftrightarrow\ \text{elim} \qquad \frac{\Gamma \vdash A \Leftrightarrow B \quad \Gamma \vdash B}{\Gamma \vdash A}\ \Leftrightarrow\ \text{elim} \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg\ \text{intro}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg\ \text{elim} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot\ \text{elim} \qquad \left(\frac{}{\Gamma \vdash A \vee \neg A}\ \textit{LEM}\ \text{en logique classique} \right)$$

Remarques

- Mnémotechnique :
  - « intro » : règle introduit le connecteur dans la conclusion.
  - « élim » : règle fait disparaître le connecteur de la conclusion.
  - Pas de règle $\bot$ *intro* : c'est bien heureux, mais pourquoi ?
- Un seul connecteur par règle (sauf tiers exclu, $\neg$*intro* et $\neg$*elim*).
- Règles ont plusieurs prémisses : preuve n'est plus linéaire.
  - C'est un arbre de preuve.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, A \Rightarrow B \vdash A \Rightarrow B}\,Ax \qquad \dfrac{}{A, A \Rightarrow B \vdash A}\,Ax}{A, A \Rightarrow B \vdash B}\Rightarrow e}{A \vdash (A \Rightarrow B) \Rightarrow B}\Rightarrow i}{\vdash A \Rightarrow (A \Rightarrow B) \Rightarrow B}\Rightarrow i$$

Au boulot...

Donnez l'arbre de preuve de $(A \Rightarrow (B \Rightarrow C)) \Rightarrow (B \Rightarrow A) \Rightarrow B \Rightarrow C$.

Oui, encore la même formule ;-)

Au boulot...

Donnez l'arbre de preuve de $((P \vee Q) \Rightarrow R) \Rightarrow P \Rightarrow R$.

Au boulot...

Donnez **2** arbres de preuve de $(A \Rightarrow B) \Rightarrow A \Rightarrow B$.

Tiers exclus, logique classique, logique intuitionniste

$$\frac{}{\Gamma \vdash A \vee \neg A}\,LEM$$

- N'exige pas une preuve de l'un ou de l'autre.
- Quelle formule est effectivement démontrable ?

- Règle équivalente à $\dfrac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}$
- $\neg\neg A$: « A n'est pas faux ».
- Plus faible que : « A est vrai ».
- « Va, je ne te hais point » (Le Cid, Corneille) est-il « Va, je t'aime » ?

- Logique classique : autorise le tiers exclus.
- Logique intuitionniste : refuse le tiers exclus.

- Preuves intuitionnistes : lien très fort avec l'extraction d'algorithmes.
- On y reviendra plus tard...
- Pour ce cours : (majoritairement) systèmes intuitionnistes.

Lois de De Morgan en logiques classique et intuitionniste

- En logique classique :
  - $\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$
  - $\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$
  - $A \vee B \Leftrightarrow \neg(\neg A \wedge \neg B)$
  - $A \wedge B \Leftrightarrow \neg(\neg A \vee \neg B)$
- En logique intuitionniste :
  - $\neg(A \wedge B) \Leftarrow \neg A \vee \neg B$
  - $\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$
  - $A \vee B \Rightarrow \neg(\neg A \wedge \neg B)$
  - $A \wedge B \Rightarrow \neg(\neg A \vee \neg B)$

Une question bien légitime : quelle cohérence ?

- Qu'est-ce qui nous garantit que ce système est correct ?
- Pourquoi ces règles seraient cohérentes ?
- Axiomes doivent être des tautologies
  - si l'on n'utilise pas le tiers exclu : facile, juste règle Ax.
- Règles doivent permettre de démontrer seulement des tautologies.
- Déductions sont faites à partir des formules en hypothèses :
  - $\Rightarrow$ conditions sur le contexte d'hypothèses.

Quelques inévitables définitions préalables

Définition (Modèle d'un contexte $\models_\nu \Gamma$)

Une valuation est un modèle de l'ensemble de formules d'un contexte Γ *si elle est un modèle de toutes les formules contenues dans* Γ.

Définition (Conséquence logique $A \models B$)

A *a pour conséquence logique* B *si toute valuation rendant* A *vraie rend* B *vraie.*

Équivalent à : tout modèle de A *est un modèle de* B.

Définition (Conséquence d'un contexte $\Gamma \models P$)

Un contexte Γ *a pour conséquence* P *si toute valuation étant un modèle de* Γ *est un modèle de* P.

Correction de la déduction naturelle

Théorème (Correction de la DN pour la logique propositionnelle)

Si une formule P est démontrable dans un contexte Γ alors P est une conséquence de Γ.
Plus formellement, si $\Gamma \vdash P$ alors $\Gamma \models P$.

- Équivalent à :
  Si P est démontrable dans Γ, alors toute valuation étant un modèle de Γ est un modèle de P.
- Preuve par induction sur « *le fait d'être une preuve de P* en déduction naturelle ».
- Preuve par induction sur l'arbre de preuve de $\Gamma \vdash P$.

Preuve : règle axiome

$$\frac{\text{si } A \in \Gamma}{\Gamma \vdash A}\ \textit{axiome}$$

Hypothèse **H0** : $A \in \Gamma$

Prouvons : pour toute ν, si $\models_\nu \Gamma$ alors $\models_\nu A$

- Hypothèse **H1** : $\models_\nu \Gamma$
- Prouvons : $\models_\nu A$
  - Par H1, on déduit **H2** : $\forall P \in \Gamma, \models_\nu P$
  - Appliquons H2, il reste à prouver $A \in \Gamma$
    - CQFD par l'hypothèse H0

Preuve : $\wedge intro$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge intro$$

Par hypothèse d'induction, **H0** : $\Gamma \models A$
Par hypothèse d'induction, **H1** : $\Gamma \models B$
Prouvons : pour toute ν, si $\models_\nu \Gamma$ alors $\models_\nu A \wedge B$
  Hypothèse **H2** : $\models_\nu \Gamma$
  Prouvons : $\models_\nu A \wedge B$, càd $[\![A \wedge B]\!]_\nu = \top$
    Par H0, on déduit **H3** : $\forall \nu', \models_{\nu'} \Gamma \Rightarrow \models_{\nu'} A$
    Par H1, on déduit **H4** : $\forall \nu', \models_{\nu'} \Gamma \Rightarrow \models_{\nu'} B$
    Par H2 et H3, on a **H5** : $\models_\nu A$
    Par H2 et H4, on a **H6** : $\models_\nu B$
    Par H5 : $[\![A]\!]_\nu = \top$
    Par H6 : $[\![B]\!]_\nu = \top$
    $[\![A \wedge B]\!]_\nu = [\![A]\!]_\nu \wedge_{\mathbb{B}} [\![B]\!]_\nu$
    Par H5 et H6, $[\![A]\!]_\nu \wedge_{\mathbb{B}} [\![B]\!]_\nu = \top \wedge_{\mathbb{B}} \top = \top$

Preuve : $\wedge elim$ (gauche)

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge elim$$

Par hypothèse d'induction, **H0** : $\Gamma \models A \wedge B$

Prouvons : pour toute ν, si $\models_\nu \Gamma$ alors $\models_\nu A$

Hypothèse **H1** : $\models_\nu \Gamma$

Prouvons : $\models_\nu A$, càd $[\![A]\!]_\nu = \top$

Par H0, on déduit **H2** : $\forall \nu', \models_{\nu'} \Gamma \Rightarrow \models_{\nu'} A \wedge B$

H1 et H2 on a **H3** : $\models_\nu A \wedge B$

Par H3 on a **H4** : $[\![A \wedge B]\!]_\nu = \top$

$[\![A \wedge B]\!]_\nu = [\![A]\!]_\nu \wedge_{\mathbb{B}} [\![B]\!]_\nu = \top$

Par la table de vérité du $\wedge_{\mathbb{B}}$, on a **H5** : $[\![A]\!]_\nu = \top$ et **H6** : $[\![B]\!]_\nu = \top$

CQFD par H5

Preuve : $\vee$ *intro* (gauche)

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee \textit{intro}$$

Par hypothèse d'induction, **H0** : $\Gamma \models A$

Prouvons : pour toute ν, si $\models_\nu \Gamma$ alors $\models_\nu A \vee B$

Hypothèse **H1** : $\models_\nu \Gamma$

Prouvons : $\models_\nu A \vee B$, càd $[\![A \vee B]\!]_\nu = \top$

Par H0, on déduit **H2** : $\forall \nu', \models_{\nu'} \Gamma \Rightarrow \models_{\nu'} A$

Par H1 et H2 on a **H3** : $\models_\nu A$

Par H3 on a **H4** : $[\![A]\!]_\nu = \top$

H5 : $[\![A \vee B]\!]_\nu = [\![A]\!]_\nu \vee_{\mathbb{B}} [\![B]\!]_\nu$

Par H4 et H5, on a **H6** : $[\![A \vee B]\!]_\nu = \top \vee_{\mathbb{B}} [\![B]\!]_\nu$

Par la table de vérité du $\vee$, on a **H7** : $\top \vee_{\mathbb{B}} [\![B]\!]_\nu = \top$

CQFD par H6 et H7

Preuve : $\neg$*intro* (1/2)

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg intro$$

Par hypothèse d'induction, **H0** : $\Gamma, A \models \bot$

Prouvons : pour toute ν, si $\models_\nu \Gamma$ alors $\models_\nu \neg A$

Hypothèse **H1** : $\models_\nu \Gamma$

Prouvons : $\models_\nu \neg A$, càd $[\![\neg A]\!]_\nu = \top$

Par H0 on déduit **H2** : $\forall \nu', \models_{\nu'} \Gamma, A \Rightarrow \models_{\nu'} \bot$

Par H2 on déduit **H3** : $\forall \nu', \models_{\nu'} \Gamma, A \Rightarrow [\![\bot]\!]_{\nu'} = \top$

Par H3 on déduit **H4** : $\forall \nu', \models_{\nu'} \Gamma, A \Rightarrow \bot = \top$

Raisonnons par cas sur $[\![A]\!]_\nu$

Cas **H5** : $[\![A]\!]_\nu = \top$

Par H5 on a alors $[\![\neg A]\!]_\nu = \bot$

Il faut donc prouver $\bot = \top$

Par H1 et H5 on a **H6** : $\models_\nu \Gamma, A$

CQFD par H4 et H6

Preuve : $\neg intro$ (2/2)

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg intro$$

Par hypothèse d'induction, **H0** : $\Gamma, A \models \bot$

Prouvons : pour toute ν, si $\models_\nu \Gamma$ alors $\models_\nu \neg A$

Hypothèse **H1** : $\models_\nu \Gamma$

Prouvons : $\models_\nu \neg A$, càd $[\![\neg A]\!]_\nu = \top$

Par H0 on déduit **H2** : $\forall \nu', \models_{\nu'} \Gamma, A \Rightarrow \models_{\nu'} \bot$

Par H2 on déduit **H3** : $\forall \nu', \models_{\nu'} \Gamma, A \Rightarrow [\![\bot]\!]_{\nu'} = \top$

Par H3 on déduit **H4** : $\forall \nu', \models_{\nu'} \Gamma, A \Rightarrow \bot = \top$

Raisonnons par cas sur $[\![A]\!]_\nu$

Cas **H5** : $[\![A]\!]_\nu = \top$

Cas **H5** : $[\![A]\!]_\nu = \bot$

Par H5 on a alors $[\![\neg A]\!]_\nu = \top$

CQFD

Preuve : $\neg elim$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg elim$$

Par hypothèse d'induction, **H0** : $\Gamma \models A$
Par hypothèse d'induction, **H1** : $\Gamma \models \neg A$
Prouvons : pour toute ν, si $\models_\nu \Gamma$ alors $\models_\nu \bot$
  Hypothèse **H2** : $\models_\nu \Gamma$
  Prouvons : $\models_\nu \bot$ càd $[\![\bot]\!]_\nu = \top$
    Par H0, on déduit **H3** : $\forall \nu', \models_{\nu'} \Gamma \Rightarrow \models_{\nu'} A$
    Par H1, on déduit **H4** : $\forall \nu', \models_{\nu'} \Gamma \Rightarrow \models_{\nu'} \neg A$
    Par H2 et H3, on déduit **H5** : $\models_{\nu'} A$, càd $[\![A]\!]_\nu = \top$
    Par H2 et H4, on déduit **H6** : $\models_{\nu'} \neg A$, càd $[\![\neg A]\!]_\nu = \top$
    Par H6 et la table de vérité de $\neg_{\mathbb{B}}$, on a **H7** : $[\![\neg A]\!]_\nu = \neg [\![A]\!]_\nu = \top$
    Par H5 et H7, on a **H8** : $\neg [\![A]\!]_\nu = \neg \top = \bot$
    CQFD par contradiction entre H7 et H8.

Au boulot...

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow \textit{intro}$$

Donnez la preuve de correction pour la règle $\Rightarrow$ *intro*.

Au boulot...

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow \textit{elim}$$

Donnez la preuve de correction pour la règle $\Rightarrow$ *elim*.

Complétude de la déduction naturelle

Théorème (Complétude de la DN pour la logique propositionnelle)

Si un contexte Γ *a pour conséquence une formule* P *alors* P *est démontrable dans* Γ.
Plus formellement, si $\Gamma \models P$ *alors* $\Gamma \vdash P$.

- Preuve nettement moins facile que la correction :-(
- Correction : hypothèse $\Gamma \vdash P$: objet inductif.
- Complétude : hypothèse $\Gamma \models P$: n'est plus inductif.
- Admettons-la lâchement cette fois-ci ;-)

Outiller les preuves en déduction naturelle

- Fastidieux de construire les arbres de preuve à la main.
- Aucune vérification d'erreur dans l'enchaînement des règles.
- Pourtant, vérification très automatique : dépend
  - de la forme du but,
  - du contenu du contexte.
- Nouveau but à prouver automatique en fonction de la règle utilisée.
- Faisons notre propre outil d'aide à la preuve.

Au boulot...

Récupérez l'archive `02_dn_tool.zip` depuis le Moodle.

Elle contient des analyseurs lexical et syntaxique, permettant de construire une représentation de formules lues depuis un fichier ou l'entrée standard ainsi que la reconnaissance de commandes correspondant aux règles de la déduction naturelle.

1. Renommez le fichier `core-eleves.ml` en `core.ml`
2. Invoquez la commande `touch .depend`
3. Invoquez la commande `make depend`
4. Invoquez la commande `make` pour finalement compiler.
5. Testez l'exécutable obtenu en lançant `./dnt.x` et en saisissant une formule terminée par un `.` final.
6. L'outil passe en mode preuve et attend que vous rentriez des commandes.

Syntaxe des formules : la même que précédemment.

Syntaxe des commandes

- `Ax` $\dfrac{\text{si } A \in \Gamma}{\Gamma \vdash A}\, Ax$
- `/\i` $\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\, \wedge i$
- `/\el` (A) $\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\, \wedge e$
- `/\er` (B) $\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\, \wedge e$
- `\/il` $\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\, \vee i$
- `\/ir` $\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\, \vee i$
- `\/e` (A, B) $\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\, \vee e$
- `->i` $\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\, \Rightarrow i$
- `->e` (A) $\dfrac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\, \Rightarrow e$
- `<->i` $\dfrac{\Gamma, A \vdash B \quad \Gamma, B \vdash A}{\Gamma \vdash A \Leftrightarrow B}\, \Leftrightarrow i$
- `<->el` (A) $\dfrac{\Gamma \vdash A \Leftrightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\, \Leftrightarrow e$
- `<->er` (B) $\dfrac{\Gamma \vdash A \Leftrightarrow B \quad \Gamma \vdash B}{\Gamma \vdash A}\, \Leftrightarrow e$
- `~i` $\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\, \neg intro$
- `~e` (A) $\dfrac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\, \neg e$
- `Fe` $\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}\, \bot elim$
- `ExclMid` $\dfrac{}{\Gamma \vdash A \vee \neg A}\, LEM$

Au boulot...

Regardons ensemble le fichier `ast.ml` qui contient les types permettant de représenter les formules et les commandes.

Regardons ensemble le fichier `core.ml` qui contient le squelette de la fonction qui va gérer le déroulement de la preuve dictée par l'utilisateur.

Complétez ce squelette pour chaque cas de commandes.

Le calcul des séquents pour la logique propositionnelle

Je ne veux pas travailler

- Pour le moment, encore et toujours en logique propositionnelle.
- Déduction naturelle : formalisation du raisonnement.
- Énumération des tables de vérité d'une formule : processus fini.
- $\Rightarrow$ Prouvabilité d'une formule est décidable.
- Appliquer manuellement les règles : c'est amusant mais bon...
- Envie d'une recherche automatique d'un arbre de preuve.

Constat en déduction naturelle

- À prouver : $P, Q \vdash P \wedge Q$
  - Facile $$\frac{P, Q \vdash P \qquad P, Q \vdash Q}{P, Q \vdash P \wedge Q} \wedge i$$
  - Connecteur du but dicte la règle à utiliser.
- À prouver : $P \wedge Q \vdash Q$
  - Moins facile $$\frac{P \wedge Q \vdash P \wedge Q}{P \wedge Q \vdash Q} \wedge e$$
  - Connecteur du but ne donne pas d'indication.
  - C'est le contexte (les hypothèses) qui aide.
  - Nécessité de « deviner » l'utilisation de P qui n'est pas dans le but.
- Dissymétrie entre règles intro et elim.
- Ne va pas aider pour automatiser la recherche de preuves : (

Re-constat en déduction naturelle

- Certaines règles ne sont pas inversibles.
- Conclusion peut être vraie sans que les prémisses ne soient vraies
  - $\vee i$ (L/R), $\wedge e$ (L/R), $\Rightarrow e$ (et $\exists i$, $\forall e$ plus tard).

- Exemple : $$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge elim$$
  - Si cette règle est choisie mais que B n'est pas prouvable : impasse.
  - Pas moyen de continuer l'arbre pour une seconde chance.

- Exemple : $$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee intro$$
  - Si cette règle est choisie mais que c'était B qui était prouvable : impasse.
  - Pas moyen de continuer l'arbre pour une seconde chance.
- Oblige à revenir en arrière dans la recherche de preuve.
- Ne va pas aider pour automatiser efficacement la recherche de preuves : (

Objectifs

- Briser la dissymétrie entre les règles *elim* et *intro*.
- Rendre si possible les règles de déduction inversibles.
  - Permet des optimisations dans la recherche de preuve.
- Assurer que chaque séquent en prémisse d'une règle contient moins de connecteurs logiques que celui en conclusion.
  - Application itérative des règles termine forcément.
  - Plutôt agréable pour la recherche de preuves.
- Garder les règles intro de la déduction naturelle.
- Remplacer les règles elim par des introduction sur les hypothèses du séquent.
  - $*$-intro $\rightsquigarrow$ $*$-droite
    - introduit $*$ à droite du séquent conclusion de la règle.
  - $*$-elim $\rightsquigarrow$ $*$-gauche
    - introduit $*$ à gauche du séquent conclusion de la règle.

Forme des séquents

- Forme générale (LK) : $P_1, \ldots, P_n \vdash Q_1, \ldots, Q_m$
  - P_i : hypothèses.
  - Q_i : conclusion**s**.
  - « Si tous les P_i sont vrais, alors au moins l'un des Q_i est vrai. »
  - Hypothèses : conjonction.
  - Conclusions : disjonction.

- Ici, forme plus restreinte issue des modifications de Gentzen pour la logique intuitionniste (LJ) : une seule formule à droite (conclusion).

Règles du calcul des séquents

$$\frac{}{A \vdash A}\,Ax \qquad \frac{\Gamma \vdash A}{\Gamma, B \vdash A}\,Thin \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\,Perm \qquad \frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}\,Cut$$

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C} \Rightarrow g \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow d \qquad \frac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \Leftrightarrow B \vdash C} \Leftrightarrow g$$

$$\frac{\Gamma \vdash B \qquad \Gamma, A \vdash C}{\Gamma, A \Leftrightarrow B \vdash C} \Leftrightarrow g \qquad \frac{\Gamma, A \vdash B \qquad \Gamma, B \vdash A}{\Gamma \vdash A \Leftrightarrow B} \Leftrightarrow d \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C} \wedge g$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge d \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C} \vee g \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee d \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee d$$

$$\frac{\Gamma \vdash A}{\Gamma, \neg A \vdash B} \neg g \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg d \qquad \frac{}{\Gamma, \bot \vdash A} \bot g \qquad \left(\frac{}{A \vee \neg A}\,LEM \text{ en logique classique} \right)$$

Quelques remarques

- Gestion très explicite du contexte (axiome, affaiblissement, permutation).

$$\frac{}{A \vdash A}\,Ax \qquad \frac{\Gamma \vdash A}{\Gamma, B \vdash A}\,Thin \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\,Perm$$

- Règle axiome n'est plus « si $A \in \Gamma$ » : d'où besoin de *Thin* et *Perm*.
- Règles gauches détruisent des hypothèses. Ex :

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C} \Rightarrow g$$

$A \Rightarrow B$ disparaît des contextes prémisses.

- Sera problématique lorsque l'on aura le $\exists$ dans la logique.
- On rajoutera une règle :-)

Un petit exemple de preuve

- Comparons la preuve de $(M \wedge N) \Rightarrow N$ en DN et CS.
- Déduction naturelle.

$$\dfrac{\dfrac{M \wedge N \vdash M \wedge N}{M \wedge N \vdash N}\,\wedge e}{\vdash (M \wedge N) \Rightarrow N}\,\Rightarrow i$$

- Calcul des séquents.

$$\dfrac{\dfrac{M, N \vdash N}{M \wedge N \vdash N}\,\wedge g}{\vdash (M \wedge N) \Rightarrow N}\,\Rightarrow d$$

- Plus besoin de « deviner » une formule sortie de nulle part.

Les coupures

- Fréquent en maths : soit à prouver $\forall x,\ (2+3)\,x = 2\,x + 3\,x$.
- Choix 1 : prouver exactement cet énoncé.
- Choix 2 : utiliser des théorèmes.
  - Distributivité : $\forall x\,y\,z,\ x\,(y+z) = x\ y + x\,z$
  - Commutativité : $\forall x\ y,\ x\,y = y\,x$
- Démontre un cas général pour l'appliquer au cas particulier.
- Preuves beaucoup plus lisibles.
- Permet la réutilisabilité des arbres de preuve des théorèmes.

Définition (Coupure – en déduction naturelle)

Une coupure est une démonstration terminée par un elim d'un connecteur, dont la prémisse où ce connecteur apparaît est démontrée par un intro de ce connecteur.

$$\dfrac{\dfrac{\dfrac{\dots}{\Gamma, P \Rightarrow Q}}{\Gamma \vdash P \Rightarrow Q}\Rightarrow i \qquad \dfrac{\dots}{\Gamma \vdash P}}{\Gamma \vdash Q}\Rightarrow e$$

La règle *Cut*

$$\frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}\ \mathit{Cut}$$

- Rend explicite les coupures.
- Problème : fait apparaître des formules n'apparaissant pas dans le but.
- Problème : Prémisse gauche peut avoir plus de connecteurs que la conclusion.
- Ne va pas aider pour automatiser la recherche de preuves :(
- Bonne nouvelle sur la diapo suivante ;-)

Vers une automatisation des démonstration en calcul des séquents

Théorème (Sous-formule)

Si une preuve de $\Gamma \vdash P$ est sans coupure ni tiers exclus, alors toutes les formules apparaissant dans cette preuve sont des sous-formules de Γ et de P.

Théorème (Terminaison de la recherche de preuve)

Pour la logique propositionnelle (donc, sans quantificateurs), l'ensemble des éléments d'une preuve sans coupure du séquent $\Gamma \vdash P$ est fini.

Théorème (Élimination des coupures (G. Gentzen))

Toute preuve en calcul des séquents utilisant des coupures peut être transformée en preuve sans coupures.

Un algorithme simple de recherche de preuve

- Formule atomique : variable propositionnelle ou $\bot$.
- Démontrer $(\Gamma \vdash P)$:
  1. S'il existe une formule P' non atomique dans Γ, appliquer la règle (gauche) r correspondant au connecteur logique de P' puis récursivement Démontrer (les prémisses de r).
  2. Sinon, si P est une formule non atomique, appliquer la règle (droite) r correspondant au connecteur logique de P puis récursivement Démontrer (les prémisses de r).
  3. Sinon appliquer $\bot g$ ou Ax si c'est possible.
  4. Sinon échec.

Remarques sur l'algorithme

- Simuler *Thin* et *Perm* par une recherche dans la liste que représente le contexte.
- Étape 1 : parcourt tout le contexte.
  - Si sous-preuve échoue pour une formule choisie, tester avec les suivantes.
  - Peut faire abandonner tout un sous-arbre de preuve pour revenir en arrière : backtrack.
  - Algo pas très efficace :-/
  - Mieux : appliquer d'abord les règles inversibles et quand on ne peut plus, on passe aux non-inversibles.
  - Retarde le backtrack et permet de backtracker moins loin.
  - Ne pas se rappeler récursivement immédiatement :
    - ⋆ Appliquer les règles tant que possible (dans conclusion puis contexte).
    - ⋆ Accumuler les sous-buts à prouver.
    - ⋆ Se rappeler sur les sous-buts accumulés quand plus de règles applicables.
    - ⋆ Peut utiliser une *working-list*.

Au boulot (1/2)...

Écrivons notre prouveur automatique en calcul des séquents (version naïve).

Récupérez l'archive `03_cs_auto.zip` depuis le Moodle.

Elle contient des analyseurs lexical et syntaxique, permettant de construire une représentation de formules lues depuis un fichier ou l'entrée standard ainsi qu'un squelette du futur prouveur.

1. Renommez le fichier `core-eleves.ml` en `core.ml`
2. Invoquez la commande `touch .depend`
3. Invoquez la commande `make depend`
4. Invoquez la commande `make` pour finalement compiler.
5. Testez l'exécutable obtenu en lançant `./sca.x` et en saisissant une formule terminée par un `.` final.
6. L'outil vous affiche la formule lue... et à vous de compléter pour qu'il dise si elle est démontrable.

Syntaxe des formules : la même que précédemment.

Au boulot : quelques conseils (2/2)...

- Proposition de structuration du code :
  - Une fonction `apply_rule_left` qui en fonction de la formule qu'elle reçoit, détermine quelle règle gauche appliquer, génère les nouveaux séquents prémisses puis appelle récursivement la fonction de recherche de preuve dessus.
  - Idem avec une fonction `apply_rule_right` pour règles droites.
  - Une fonction `try_left_in_hyps` qui effectue les traitement de l'étape 1 de l'algorithme.
  - Une fonction `process_goal` qui applique les 3 étapes de l'algorithme sur le but (séquent) qu'elle reçoit et doit démontrer.
  - Une fonction `process_goals` qui itère le traitement d'un séquent sur une liste de séquents.
- En cas d'échec d'une (sous-) preuve, lever l'exception `Proof_failed` (que vous devrez parfois rattraper `;-)`).
- Le programme devra juste afficher si la formule initiale est prouvable (pas besoin de construire l'arbre de preuve).
- Pour tester : fichiers `provable.pr` et `nprovable.pr`.

Correction, complétude

Théorème (Correction du CS pour la logique propositionnelle)

Si une formule P est démontrable dans un contexte Γ alors P est une conséquence de Γ.
Plus formellement, si $\Gamma \vdash P$ alors $\Gamma \models P$.

Théorème (Complétude du CS pour la logique propositionnelle)

Si un contexte Γ a pour conséquence une formule P alors P est démontrable dans Γ.
Plus formellement, si $\Gamma \models P$ alors $\Gamma \vdash P$.

- Le calcul des séquents est équivalent à la déduction naturelle :
  - toute preuve dans l'un peut être transformée en une preuve dans l'autre.

Calcul des séquents → déduction naturelle

- Preuve par induction sur la forme d'une preuve en CS.
- Soit une preuve Π en CS de $\Gamma \vdash P$.
- Π à la forme : $\dfrac{\Gamma_1 \vdash Q_1 \quad \dots \quad \Gamma_n \vdash Q_n}{\Gamma \vdash P}\, r$
- Chaque, $\Gamma_i \vdash Q_i$ a une preuve π_i en CS.
- Par hypothèses d'induction, chaque $\Gamma_i \vdash Q_i$ a une preuve π'_i en DN.
- « Reste » à construire une preuve Π' en DN de $\Gamma \vdash P$ pour la règle r.
- Cas $\wedge g$: $\dfrac{\dfrac{\pi}{\Gamma, A, B \vdash C}}{\Gamma, A \wedge B \vdash C}\wedge g \quad \rightarrow \quad \dfrac{\overline{\pi'}}{\Gamma, A \wedge B \vdash C}$ où $\overline{\pi'} = \pi'$ dans laquelle :
  - retire des séquents toutes les hypothèses A et B et ajoute $A \wedge B$
  - remplace les axiomes $\Delta \vdash A$ (resp. B) par $\dfrac{\Delta \vdash A \wedge B}{\Delta \vdash A \ (resp.\ B)}\wedge e$

Déduction naturelle → calcul des séquents

- Preuve par induction sur la forme d'une preuve en CS.
- Même principe que dans l'autre sens.
- Par hypothèses d'induction, chaque $\Gamma_i \vdash Q_i$ a une preuve π'_i en CS.

- Cas $\Rightarrow e$:

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash A \Rightarrow B} \qquad \dfrac{\pi_2}{\Gamma \vdash A}}{\Gamma \vdash B} \Rightarrow e$$

$\longrightarrow$

$$\dfrac{\dfrac{\pi'_1}{\Gamma \vdash A \Rightarrow B} \qquad \dfrac{\dfrac{\pi'_2}{\Gamma \vdash A} \qquad \dfrac{}{\Gamma, B \vdash B}\, Ax}{\Gamma, A \Rightarrow B \vdash B}}{\Gamma \vdash B}\, Cut$$

Au boulot...

Le calcul des séquents sans la règle *Cut* est-il encore équivalent à la déduction naturelle ? Pourquoi ?

Premier contact avec un véritable outil : Coq

On a vu jusqu'à présent...

- Trois formalisations de la logique propositionnelle :
  - système à la Frege-Hilbert,
  - déduction naturelle,
  - calcul des séquents.
- Quelques programmes autour des formules propositionnelles :
  - prouveur force brute,
  - petit outil d'aide à la preuve pour la déduction naturelle,
  - prouveur automatique pour le calcul des séquents.
- Grand temps de s'aider d'un véritable outil.

Coq

- https://coq.inria.fr
- Développé à Inria depuis le début des années 80.
- Atelier d'aide à la preuve.
- Tactiques d'automatisation pour différents domaines :
  - procédures de décision ou heuristiques.
- Théorie sous-jacente : calcul des constructions + définitions inductives.
- Système de types très riche.
- Logique d'ordre supérieur (quantification sur fonctions et prédicats).
- Développé en OCaml.
- Réalisations phares (mais pas les seules d'envergure) :
  - théorème des quatre couleurs (G. Gonthier, B. Werner),
  - théorème de Feit-Thompson (G. Gonthier),
  - CompCert (X. Leroy, S. Blazy et ali.).

Lancer Coq

- D'abord, l'installer : `https://coq.inria.fr/download`.
- Depuis `Emacs` : installer `ProofGeneral`.
- Depuis `VSCode` : installer le plugin `VSCoq`.
- Utiliser l'IDE : installer `coqide`.
- Dans un terminal : lancer `coqtop`.
  - Mieux : installer `rlwrap` puis lancer `rlwrap coqtop`.
- On privilégiera rapidement un éditeur de textes :
  - vous écrivez vos programmes OCaml dans le terminal ? ;-)

Le minimum pour commencer en logique propositionnelle

- Déclarer des variables propositionnelles : **Variable** `A B C` : **Prop**.
  - `Prop` : type des formules logiques.
- Syntaxe des connecteurs : le même que la nôtre `:-)`
  - `/\` `\/` `->` `<->` `~` **False**
- Introduire un but : **Goal** `(A -> B)-> A -> B`.
- Passe en mode preuve et attend des commandes : tactiques de preuve.
- Toujours terminées par un `.` final.
- Mémorise les buts en suspens générés par les tactiques.
- Affiche le but courant.
- Vérifié l'adéquation de la commande en fonction du but courant.
- Mais c'est à vous de bosser `;-)`

Première rencontre

- Exemple en interactif.

```
$ rlwrap coqtop
Welcome to Coq 8.18.0

Coq < Variable A B : Prop.
A is declared
B is declared

Coq < Goal (A -> B) -> A -> B.
1 goal

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

Unnamed_thm <
```

- Oui, il attend des tactiques, càd que vous démontriez. . .

Commandes en fonction du but (v1)

- On augmentera la *cheat-sheet* plus tard, pour le calcul des prédicats.
- *Grosso modo* règles droites du calcul des séquents.
- Pour le moment...

Quand le but est	utiliser la tactique	Règle
`P`	`exact` H_x si existe hypothèse H_x : `P`	Ax
`P`	`assumption` si existe une hypothèse H_x : `P`	Ax
`P /\ Q`	`split`	$\wedge d$
`P \/ Q`	`left ou right`	$\vee d$
`P -> Q`	`intro`	$\Rightarrow d$
`P <-> Q`	`split`	$\Leftrightarrow d$
`~P`	`intro`	$\neg d$
On s'est trompé	`Undo`	`:-/`
On abandonne	`Abort`	`:-(`

- `intros` : plusieurs $\Rightarrow d$ en chaîne.
- `intro` *Hfoo* : un $\Rightarrow d$ et nomme l'hypothèse « Hfoo ».
- `intros` *Hfoo Hbar Hgee* : nomme plusieurs hypothèses en chaîne.

Commandes en fonction des hypothèses (v1)

- *Grosso modo* règles gauches du calcul des séquents.
- Pour le moment...

Pour utiliser l'hyp.H	utiliser la tactique	Règle
`P /\ Q`	`destruct H`	$\wedge g$
`P \/ Q`	`destruct H`	$\vee g$
`P -> Q`	`apply H`	À peu près $\Rightarrow g$
`P <-> Q`	`apply H`	À peu près $\Leftrightarrow g$
`~P`	`apply H`	$\neg g$
`False`	`contradiction`	À peu près $\bot g$

Commandes diverses

- De l'administratif. . ..

`Qed`	Clôt une preuve dont tous les buts sont prouvés
`Proof`	Débute un script de preuve – inutile mais c'est plus joli pour fermer un `Qed ;-)`
`Undo`	Revient une étape de preuve en arrière
`Abort`	Abandonne lâchement la preuve
`Check (`*e*`)`	Affiche le type de l'expression *e*

- Éventuellement pour plus tard. . .

`simpl`	Simplifie dans le but
`simpl in` *H*	Simplifie dans l'hypothèse *H*
`Compute (`*e*`)`	Évalue l'expression *e*
`assert (`*P*`)`	Coupure

Au boulot...

On teste ensemble des tactiques pour voir les transformations du but avec des règles « droites ».

`split`, `intros`, `exact`, `assumption`.

À prouver : $A \Rightarrow B \Rightarrow (A \wedge B)$.

Au boulot...

On teste ensemble les tactiques pour voir les transformations du but avec des règles « droites ».

`intro`, `apply`, `destruct`.

À prouver : $((A \Rightarrow B) \Rightarrow C) \Rightarrow (A \wedge B) \Rightarrow C$

Mémo (v1)

```
assumption      split                  left               right
H : P
===== --> .     ====== --> ===  ===     ====== --> ===     ====== --> ===
P               P /\ Q     P    Q       P \/ Q     P       P \/ Q     Q
```

```
intro                 split                          intro
           H : P                                             H : P
====== --> =====      ======= --> ======  ======     === --> =====
P -> Q     Q          P <-> Q     P -> Q  Q -> P     ~P      False
```

```
destruct H                destruct H
               H : P
H : P /\ Q     H0 : Q     H : P \/ Q     H : P  H : Q
========== --> ======     ========== --> =====  =====
R              R          R              R      R
```

```
apply H                        apply H                        apply H
H : P -> Q     H : P -> Q      H : P <-> Q     H : P <-> Q     H : P <-> Q     H : P <-> Q
========== --> ==========      =========== --> ===========     =========== --> ===========
Q              P               P               Q               Q               P
```

```
apply H                contradiction        contradiction
                                            H : P
H : ~P     H : ~P      H : False            H0 : ~P
====== --> ======      ========= --> .      ======= ---> .
False      P           P                    Q
```

On indente le code, on indente les démonstrations

- Plus joli de commencer une preuve par **Proof**.
- Très rare de travailler directement dans le terminal (avec `coqtop`).
- Utilise plutôt un éditeur de texte (avec un mode adéquat).
- Code source édité dans un fichier.
- Certaines tactiques génèrent plusieurs sous-buts.
- Exemple :

```
Goal (A \/ B) <-> (B \/ A).
Proof.
  split . intro . destruct H. right . assumption. left . assumption. intro . destruct H.
  right . assumption. left . assumption.
Qed.
```

- Bien plus lisible d'utiliser des *bullets* :

```
Goal (A \/ B) <-> (B \/ A).
Proof.
  split . (* Casse l'équivalence . *)
  - (* Cas -> *)
    intro . destruct H. (* Par cas sur le \/ hypothèse. *)
    + (* Cas H : A *) right . assumption.
    + (* Cas H : B *) left . assumption.
  - (* Cas <- *)
    intro . destruct H.
    + (* Cas H : B *) right . assumption.
    + (* Cas H : A *) left . assumption.
Qed.
```

Les *bullets* sont vos amies

- Permet d'indenter pour suivre les imbrications de sous-buts.
- Indentation automatique sous n'importe quel bon environnement.
- Permet à Coq de signaler une mauvaise imbrication, une sous-preuve non terminée :

```
Goal (A \/ B) <-> (B \/ A).
Proof.
  split . (* Casse l'équivalence. *)
  - (* Cas -> *)
    intro. destruct H. (* Par cas sur le \/ hypothèse. *)
    + right.
    +
```

▶ `[Focus] Wrong bullet +: Current bullet + is not finished.`

- Aide au moment de la rédaction pour ne pas s'y perdre.
- Avec des commentaires, aide au moment de la relecture.
- Niveaux d'imbrication : `-`, `+`, `*`, `--`, `++`, `**`, etc.

Au boulot...

À vous de démontrer les formules suivantes :

1. $(P \Rightarrow Q) \Rightarrow P \Rightarrow Q$
2. $(P \wedge Q) \Rightarrow P$
3. $(Q \wedge P) \Rightarrow (P \vee Q)$
4. $(P \wedge (Q \Rightarrow (R \vee S))) \Rightarrow (P \vee Q)$
5. $(P \wedge Q) \Leftrightarrow (Q \wedge P)$
6. $((P \Rightarrow Q) \wedge (P \Rightarrow R)) \Rightarrow (P \Rightarrow (Q \wedge R))$
7. $(P \vee (Q \wedge R)) \Rightarrow ((P \vee Q) \wedge (P \vee R))$
8. $(P \wedge (Q \vee R)) \Rightarrow ((P \wedge Q) \vee (P \wedge R))$
9. $(P \Leftrightarrow Q) \Rightarrow ((P \Rightarrow Q) \wedge (Q \Rightarrow P))$
10. $((P \Rightarrow Q) \Rightarrow P \Rightarrow Q) \Leftrightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow Q))$
11. $(\neg P \vee Q) \Rightarrow P \Rightarrow Q$
12. $B \vee \neg A \Rightarrow \neg A \vee B$
13. $(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow ((A \vee B) \Rightarrow C)$
14. $(P \vee (Q \wedge R)) \Leftrightarrow ((P \vee Q) \wedge (P \vee R))$
15. $\neg(A \vee B) \Rightarrow \neg A \wedge \neg B$
16. $((P \vee Q) \Rightarrow R) \Leftrightarrow ((P \Rightarrow R) \wedge (Q \Rightarrow R))$
17. $((P \vee Q) \wedge (R \vee P) \wedge (\neg Q \vee \neg R \vee P)) \Leftrightarrow P$

C'est bien gentil tout ça...

- On a dit que la vérification en logique propositionnelle était décidable.
- On a même programmé 2 prouveurs automatiques.
- Coq ne saurait pas faire ces preuves tout seul ?

- Si, mais il fallait bien s'entraîner un peu ;-)
- *The silver bullet* : la tactique `tauto`.

```
Coq < Variable P Q R S : Prop.

Coq < Goal (P \/ (Q /\ R)) <-> ((P \/ Q) /\ (P \/ R)).
1 goal

  ============================
  P \/ Q /\ R <-> (P \/ Q) /\ (P \/ R)

Unnamed_thm < tauto.
No more goals.
```

Coupure, raisonnement arrière et raisonnement avant

- Jusqu'à présent, preuves en remontant à partir du but.
- Raisonnement arrière (*backward*).

- Parfois envie de déduire une nouvelle hypothèse à partir d'hypothèses existantes.
  - Ce que l'on a beaucoup fait dans nos preuves « manuelles ».
- Parfois envie de démontrer une formule intermédiaire :
  - qui n'est pas le but mais...
  - dont ce dernier sera une conséquence.
- $\sim$ Équivalent à démontrer un lemme intermédiaire.
- Raisonnement avant (*forward*).

- Notion de coupure dans une preuve.

$$\frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}\,Cut$$

- En Coq : **assert** (P). $\{$ *Preuve* $\}$
- Accolades pour structurer la preuve (comme les *bullets*).

Exemple de raisonnement arrière

- Ce que l'on utilisé en Coq jusqu'à présent.

```
Coq < Goal (A /\ B -> C) -> A -> B -> C.
1 goal

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

Unnamed_thm < intros.
1 goal

  H : A /\ B -> C
  H0 : A
  H1 : B
  ======================
  C

Unnamed_thm < apply H.
1 goal

  H : A /\ B -> C
  H0 : A
  H1 : B
  ======================
  A /\ B
```

```
Unnamed_thm < split.
2 goals

  H : A /\ B -> C
  H0 : A
  H1 : B
  ======================
  A

goal 2 is:
 B

Unnamed_thm < assumption.
1 goal

  H : A /\ B -> C
  H0 : A
  H1 : B
  ======================
  B

Unnamed_thm < assumption.
No more goals.
```

Exemple de raisonnement avant

```
Coq < Goal (A /\ B -> C) -> A
      -> B -> C.
1 goal

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

Unnamed_thm < intros.
1 goal

  H : A ∧ B -> C
  H0 : A
  H1 : B
  ====================
  C

Unnamed_thm < assert(A ∧ B).
2 goals

  H : A /\ B -> C
  H0 : A
  H1 : B
  ====================
  A ∧ B

goal 2 is:
 C

Unnamed_thm < split.
3 goals

  H : A /\ B -> C
  H0 : A
  H1 : B
  ====================
  A

goal 2 is:
 B
goal 3 is:
 C

Unnamed_thm < assumption.
2 goals

  H : A /\ B -> C
  H0 : A
  H1 : B
  ====================
  B

goal 2 is:
 C

Unnamed_thm < assumption.
1 goal

  H : A ∧ B -> C
  H0 : A
  H1 : B
  H2 : A ∧ B
  ====================
  C

Unnamed_thm < apply H in H2.
1 goal

  H : A /\ B -> C
  H0 : A
  H1 : B
  H2 : C
  ====================
  C

Unnamed_thm < assumption.
No more goals.
```

- Assertion puis utilisation d'une hypothèse dans une hypothèse...
- pour en déduire une nouvelle hypothèse dont le but est conséquence.

Utiliser une hypothèse dans une hypothèse : détails

- **apply** `H` **in** `H'`.
- Hypothèse peut être de la forme $H : P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow P$.
- Effet de **apply** `H` **in** `H'` :
  - Vérifie que H' est « égale » à P_1 (on reviendra ici sur « égale »).
  - Remplace H' par P
  - Génère autant de sous-buts $P_2, \ldots, P_n$ à prouver.

```
1 goal
  H : A -> B -> C -> D
  H0 : A
  ============================
  D

Unnamed_thm < apply H in H0.
3 goals
  H : A -> B -> C -> D
  H0 : D
  ============================
  D
goal 2 is:          goal 3 is:
B                   C
```

Au boulot...

À vous de démontrer la formule $A \Rightarrow B \Rightarrow (A \wedge B) \wedge (A \wedge B)$ avec une belle coupure.

Pourquoi est-elle intéressante ici ?

Au boulot...

À vous de (re) démontrer la formule $(A \Rightarrow B) \Rightarrow A \Rightarrow B$ avec un raisonnement avant, en appliquant une hypothèse dans une autre pour en déduire le but recherché.

Ok, c'est totalement artificiel, mais c'est pour appliquer cette méthode ;-)

Re logique classique, logique intuitionniste

- Par défaut Coq : logique classique.
- Pas de tiers exclu, pas de double négation.
- La preuve par De Morgan :

```
Coq < Variable A B : Prop.

Coq < Goal ~ (A /\ B) -> (~ A \/ ~ B).
1 goal

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

Unnamed_thm < tauto.
Toplevel input, characters 0-5:
> tauto.
> ^^^^^
Error: Tactic failure: tauto failed.
```

Logique classique en Coq

- **Require Import** `Logic.Classical_Prop`.
- Importe l'axiome d'élimination de la double-négation
  `NNPP` : $\forall P : Prop, \neg\neg P \Rightarrow P$.
- Formule qui quantifie sur les prédicats.

```
Coq < Require Import Logic.Classical_Prop.
Coq < Variable A B : Prop.

Coq < Goal ~ (A /\ B) -> (~ A \/ ~ B).
1 goal

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

Unnamed_thm < tauto.
No more goals.
```

Au boulot...

Démontrez le principe de raisonnement par contraposée, càd $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$.

Indication : il y a besoin de l'axiome de double-négation quelque part (raisonnement par l'absurde).

Indication : pour invoquer l'axiome d'élimination de la double-négation :

- **apply** `NNPP`.

(on verra plus tard ce qu'est ce **apply**, mais en fait on l'a déjà vu).

Au boulot...

Oublions l'élimination de la double négation.

On a vu qu'elle était équivalente au tiers exclus.

Définissons le tiers exclu en axiome :

Axiom `LEM` : **forall** `P` : **Prop**, `P \/ ~P`.

Donc, plus besoin d'importer `Logic.Classical_Prop` ici.

Re-démontrez le principe de raisonnement par contraposée, càd $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$.

Indication : pour invoquer l'axiome du tiers exclu : **apply** `LEM`.
(on verra plus tard ce qu'est ce **apply**, mais en fait on l'a déjà vu).

~~Ne~~ faites ~~surtout pas~~ ça chez vous pour vous amuser

- Démontrez qu'avec le tiers exclu en axiome,
  $(A \vee \neg A) \Rightarrow (\neg\neg A \Rightarrow A)$

```
Variable A B : Prop.
Axiom LEM: forall P : Prop, P \/ ~P.
Goal (A \/ ~A) -> (~~A -> A).
Proof.
  (* Au boulot. *)
```

- Démontrez qu'avec l'élimination de la double négation en axiome,
  $(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$.

```
Variable A B : Prop.
Axiom NNPP: forall P : Prop, ~~P -> P.
Goal (~~A -> A) -> (A \/ ~A).
Proof.
  (* Au boulot. *)
```

Le calcul des prédicats

Jusqu'à présent...

- Formules logiques avec des connecteurs...
- entre des variables propositionnelles.
- « Atomes » représentent des faits figés
  - « Je suis à l'ENSTA. », A, B, P, Q.
- Pourtant, énoncé notre preuve de correction de la DN :

  « *Si une formule P est démontrable dans un contexte Γ alors...* »
- « Une formule » $\equiv$ « n'importe laquelle ».
- Comment écrire que :
  - « Quel que soit un entier n, n est pair ou n est impair. »
  - « Il y a un entier n tel que quel que soit un autre entier m, $n \times m = 0$. »
- Il manque des lieurs pour introduire des variables :
  - « quel que soit », « il y a un ».
- Il manque des « atomes paramétrés » :
  - « *est pair*(n) », « *est impair*(n) ».

Des formules plus riches

- Connecteur permettant d'introduire une variable : quantificateur.
- « Pour tout » : quantificateur universel $\forall x, \mathit{formule}$
- « Il existe » : quantificateur existentiel $\exists x, \mathit{formule}$
- Formules doivent pouvoir mentionner les variables dans
  - des fonctions : $n \times m$, $x + 1$
  - des prédicats : $\mathit{Pair}(x)$, $x < y$
- Prédicat : relation.
- Prédicat : « fonction » à valeur dans $\mathbb{B} = \{ \top, \bot \}$.
- Logique du premier ordre : pas de quantification sur fonctions ou prédicats.

Signature

- Ensemble infini dénombrable de variables : $\mathcal{V} = \{ x, y, z \dots \}$

Définition (Signature)

La signature Σ (d'un langage du premier ordre) est la donnée de deux ensembles finis de symboles. Chaque symbole est muni d'un nombre positif ou nul nommé arité.

- *$\mathcal{F}$: ensemble de symboles de fonctions.*
- *$\mathcal{R}$: ensemble de symboles de prédicats.*

- Hypothèse : $\mathcal{V}, \mathcal{F}$ et $\mathcal{R}$ sont 2 à 2 disjoints.
- Exemple : $\Sigma = \{ (\texttt{zero}, 0), (\texttt{succ}, 1), (+, 2) \}, \{ (<, 2), (\texttt{Pair}, 1) \}$
- Intuition arité : « nombre d'arguments ».

Termes

- But : représenter des « objets », des « individus ».
- N'ont pas de « valeur de vérité ».

Définition (Terme)

Soit un ensemble de variables $\mathcal{V}$, une signature $\Sigma = (\mathcal{F}, \mathcal{R})$. L'ensemble $\mathcal{T}_{\Sigma \cup \mathcal{V}}$ des termes avec variables sur $\Sigma \cup \mathcal{V}$ est défini inductivement par :

- *Toute variable de $\mathcal{V}$ est un terme.*
- *Si $t_1, \ldots, t_n$ sont des termes, si f est un symbole de $\mathcal{F}$ d'arité n, alors $f(t_1, \ldots, t_n)$ est un terme.*

- $+(\texttt{zero}, \texttt{succ}(x))$: terme.
- $\texttt{zero}(\texttt{zero})$: pas un terme (mauvaise arité).
- $<(\texttt{zero}, y)$: pas un terme ($< \notin \mathcal{F}$).

Définition (Terme clos)

Un terme clos est un terme sans variables.

Formules

Définition (Formule atomique)

Soit une signature $\Sigma = (\mathcal{F}, \mathcal{R})$. Une formule atomique sur Σ est de la forme $R(t_1, \ldots, t_n)$ où $R \in \mathcal{R}$ est d'arité n et $t_1, \ldots, t_n$ sont des termes.

Définition (Formule)

Soit une signature $\Sigma = (\mathcal{F}, \mathcal{R})$ et un ensemble $\mathcal{V}$ de variables. L'ensemble des formules sur $\Sigma \cup \mathcal{V}$ est défini inductivement par :

- *Une formule atomique est une formule.*
- *Si F est une formule, alors $\neg F$ et (F) sont des formules.*
- *Si F_1 et F_2 sont des formules, alors*
  $$F_1 \wedge F_2 \qquad F_1 \vee F_2 \qquad F_1 \Rightarrow F_2 \qquad F_1 \Leftrightarrow F_2$$
  sont des formules.
- *Si F est une formule, si $x \in \mathcal{V}$, alors $\forall x\ F$ est une formule.*
- *Si F est une formule, si $x \in \mathcal{V}$, alors $\exists x\ F$ est une formule.*

Au boulot...

L'axiome de l'élimination de la double-négation vu précédemment :

$$(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A))$$

est-il une formule du premier ordre ?

La *propriété de substitution* qui exprime que :

$$\forall f\ \forall x\ \forall y\ x = y \Rightarrow f(x) = f(y)$$

est-elle une formule du premier ordre ?

Au boulot...

Modélisez les phrases suivantes en logique du premier ordre. Vous pourrez introduire les prédicats que vous jugez nécessaires.

1. Un éléphant ça trompe énormément. (ref.)
2. Il faut qu'une porte soit ouverte ou fermée. (ref.)
3. Les élèves disent les profs sont pénibles.
4. Les élèves, disent les profs, sont pénibles.
5. Le monde se divise en 2 catégories, ceux qui ont un pistolet chargé et ceux qui creusent. (ref.)
6. Quand on est enfant, on a toujours une bêtise à faire.
7. Tout le monde a un cerveau.

Sémantique des formules

- En logique propositionnelle :
  - Valuation pour donner une valeur aux variables propositionnelles.
  - Tables de vérité pour la sémantique des connecteurs.
- En calcul des prédicats :
  - Sémantique des variables ?
  - Sémantique des symboles de fonction ?
  - Sémantique des prédicats ?
  - Besoin de « prédéfinir » le sens des prédicats et fonctions.
  - **Rappel** : pas de quantification sur eux au premier ordre.

Structure (d'interprétation) de signature

Définition

Soit une signature $\Sigma = (\mathcal{F}, \mathcal{R})$. Une structure d'interprétation M de signature Σ est la donnée de :

- *un ensemble D non vide appelé domaine,*
- *une application $I(f) : D^n \to D$ pour chaque symbole de fonction $f \in \mathcal{F}$ d'arité n.*
- *une application $I(R) : D^n \to \mathbb{B}$ pour chaque symbole de prédicat $R \in \mathcal{R}$ d'arité n.*

- Donne la sémantique des termes et des prédicats.

Remarque sur les fonctions et prédicats 0-aires

- Constantes et variables propositionnelles rentrent dans le cas général.
- Constantes : $D^0 \to D$
  - D^0 contient exactement un élément : le n-uplet $()$
    - pas de valeur à l'intérieur.
  - $D^0 \to D$: donner une valeur de la fonction pour $()$
    - donc une constante.
- Variables propositionnelles : $D^0 \to \mathbb{B}$
  - Relation sur D^0 : un sous-ensemble d'un ensemble à un élément $()$
    - Relation vide : interprétation comme $\bot$
    - Relation avec $\{()\}$: interprétation comme $\top$

Exemples de structures de signature

- $\Sigma = (\{ (\mathit{zero}, 0), (\mathit{one}, 0), (\mathtt{plus}, 2), (\mathtt{mult}, 2) \}, \{ (\mathtt{lt}, 2) \})$
- Structure 1 :
  - $D = \mathbb{N} \cup (\mathbb{N} \to \mathbb{N})$
  - $I(\mathtt{zero}) = 0$, $I(\mathtt{one}) = 1$.
  - $I(\mathit{plus}) = +_{\mathbb{N}}$, $I(\mathtt{mult}) = \times_{\mathbb{N}}$.
  - $I(\mathit{lt}) = <_{\mathbb{N}}$.
- Structure 2 :
  - $D = \mathbb{B} \cup (\mathbb{B} \to \mathbb{B})$
  - $I(\mathtt{zero}) = \bot$, $I(\mathtt{one}) = \top$.
  - $I(\mathit{plus}) = \vee_{\mathbb{B}}$, $I(\mathtt{mult}) = \wedge_{\mathbb{B}}$.
  - $I(\mathit{lt}) =$
    $\{ (\mathtt{zero}, \mathtt{zero}, \bot), (\mathtt{zero}, \mathtt{one}, \top), (\mathtt{one}, \mathtt{zero}, \bot), (\mathtt{one}, \mathtt{one}, \bot) \}$.

Valuation

- Donner des valeurs aux variables.

Définition (Valuation)

Soit une structure M. Une valuation est une application de l'ensemble des variables (de terme) dans D le domaine de M.
$\nu : \mathcal{V} \mapsto D$

- Notation : $\nu[x \leftarrow d](y) = \begin{cases} d & \text{si } y = x \\ \nu(y) & \text{sinon} \end{cases}$
- Valuation ν dans laquelle on rajoute / remplace la valeur « liée » à x.

Interprétation des termes

- Donner des valeurs aux termes.

Définition (Interprétation des termes)

Soit une structure M de signature $\Sigma = (\mathcal{F}, \mathcal{R})$. L'interprétation $[\![t]\!]_\nu^M$ du terme t est définie inductivement par :

- *Si t est une variable x, alors $[\![t]\!]_\nu^M = \nu(x)$.*
- *Si t est de la forme $f(t_1, \ldots, t_n)$, alors $[\![t]\!]_\nu^M = I(f)([\![t_1]\!]_\nu^M, \ldots, [\![t_n]\!]_\nu^M)$.*

Interprétation des formules atomiques

- Donner des valeurs de vérité aux formules atomiques.

Définition (Interprétation des formules atomiques)

Soit une structure M de signature $\Sigma = (\mathcal{F}, \mathcal{R})$. L'interprétation $[\![r]\!]_\nu^M$ du prédicat r est définie par :

- *Forcément r est de la forme $R(t_1, \ldots, t_n)$,*

$$[\![r]\!]_\nu^M = \begin{cases} \top & si\ ([\![t_1]\!]_\nu^M, \ldots, [\![t_n]\!]_\nu^M) \in I(R) \\ \bot & sinon \end{cases}$$

Au boulot...

Soit la signature :
$\Sigma = (\{ (\texttt{Rdc}, 0), (\texttt{Et1}, 0), (\texttt{Et2}, 0), (\texttt{up}, 1), (\texttt{down}, 1) \}, \{ (\texttt{next}, 2) \})$
Soit la structure M telle que :

- $D = \mathbb{N}$
- $I(\texttt{Rdc}) = 0$, $I(\texttt{Et1}) = 1$, $I(\texttt{Et2}) = 2$
- $I(\texttt{up})(x) = \begin{cases} 1 & \text{si } x = 0 \\ 2 & \text{si } x = 1 \\ 2 & \text{si } x = 2 \end{cases}$ $I(\texttt{down})(x) = \begin{cases} 0 & \text{si } x = 0 \\ 0 & \text{si } x = 1 \\ 1 & \text{si } x = 2 \end{cases}$
- $I(\texttt{next}) = \{ (0,0), (0,1), (1,2), (2,2), (2,1), (1,0) \}$

Donnez l'interprétation des formules atomiques suivantes :

- `next (up (Rdc), down (Et2))`
- `next (down (Rdc), down (Et1))`
- `next (down (Et2), next (down (Rdc), up (Et1)))`
- `next (down (down (Et2)), up (down (Rdc)))`

Interprétation des formules (générales)

Définition (Interprétation des formules)

Soit une structure M de signature $\Sigma = (\mathcal{F}, \mathcal{R})$. L'interprétation $[\![\mathcal{F}]\!]^M_\nu$ de la formule F est définie inductivement par :

- *Si F est une formule atomique r, alors $[\![F]\!]^M_\nu = [\![r]\!]^M_\nu$.*
- *Si F est de la forme $F_1 \wedge F_2$, alors $[\![F]\!]^M_\nu = [\![F_1]\!]^M_\nu \wedge_{\mathbb{B}} [\![F_2]\!]^M_\nu$.*
- *Si F est de la forme $F_1 \Rightarrow F_2$, alors $[\![F]\!]^M_\nu = [\![F_1]\!]^M_\nu \Rightarrow_{\mathbb{B}} [\![F_2]\!]^M_\nu$.*
- *Si F est de la forme $F_1 \Leftrightarrow F_2$, alors $[\![F]\!]^M_\nu = [\![F_1]\!]^M_\nu \Leftrightarrow_{\mathbb{B}} [\![F_2]\!]^M_\nu$.*
- *Si F est de la forme $\neg F_1$, alors $[\![F]\!]^M_\nu = \neg_{\mathbb{B}} [\![F_1]\!]^M_\nu$.*
- *Si F est de la forme $\forall x, F_1$, alors*

$$[\![F]\!]^M_\nu = \begin{cases} \top & \text{si pour toute valeur } d \text{ de } D, [\![F]\!]^M_{\nu[x \leftarrow d]} = \top \\ \bot & \text{sinon} \end{cases}$$

- *Si F est de la forme $\exists x, F_1$, alors*

$$[\![F]\!]^M_\nu = \begin{cases} \top & \text{s'il existe une valeur } d \text{ de } D, \text{t.q. } [\![F]\!]^M_{\nu[x \leftarrow d]} = \top \\ \bot & \text{sinon} \end{cases}$$

Au boulot...

Soit la signature $\Sigma = (\{(\texttt{Vert}, 0), (\texttt{Orange}, 0), (\texttt{Rouge}, 0)\}, \{(\texttt{next}, 1)\}, \{(\texttt{run}, 1), (\texttt{stop}, 1)\})$

Soit la structure M telle que :

- $D = \{V, O, R\}$
- $I(\texttt{Vert}) = V$, $I(\texttt{Orange}) = O$, $I(\texttt{Rouge}) = R$
- $I(\texttt{next})(x) = \begin{cases} O & \text{si } x = V \\ R & \text{si } x = O \\ V & \text{si } x = R \end{cases}$
- $I(\texttt{run}) = \{V, O\}$
- $I(\texttt{stop}) = \{R\}$

Donnez l'interprétation des formules suivantes :

- $\exists f\ \texttt{run}(\texttt{next}(f))$
- $\exists f\ \texttt{stop}(f) \Rightarrow \texttt{stop}(\texttt{next}(f))$
- $\forall f\ \texttt{run}(\texttt{next}(f)) \Rightarrow \texttt{stop}(f)$
- $\forall f\ \texttt{run}(f) \vee \texttt{run}(\texttt{next}(f))$

Variables d'un terme, d'une formule atomique

Définition

Soit une signature $\Sigma = (\mathcal{F}, \mathcal{R})$. *L'ensemble des variables* $\texttt{Vars}(t)$ *d'un terme* t *sur* Σ *est défini inductivement par :*

- *Si* t *est de la forme* x, *alors* $\texttt{Vars}(t) = \{ x \}$
- *Si* t *est de la forme* $f(t_1, \ldots, t_n)$, *alors* $\texttt{Vars}(t) = \texttt{Vars}(t_1) \cup \ldots \cup \texttt{Vars}(t_n)$

Définition

Soit une signature $\Sigma = (\mathcal{F}, \mathcal{R})$. *L'ensemble des variables* $\texttt{Vars}(r)$ *d'une formule atomique* r *sur* Σ *est défini par :*

- *Forcément* r *est de la forme* $R(t_1, \ldots, t_n)$, $\texttt{Vars}(r) = \texttt{Vars}(t_1) \cup \ldots \cup \texttt{Vars}(t_n)$

- Simplement l'ensemble des variables présentes dans le terme ou la formule atomique, quoi :-)

Variables libres, liées

Définition

Soit une formule F. L'ensemble des variables libres $\mathtt{FV}(F)$ *(*free vars*) et l'ensemble des variables liées* $\mathtt{BV}(F)$ *(*bound vars*) de F sont définis inductivement par :*

- *Si F est une formule atomique, alors*
  - $\mathtt{FV}(F) = \mathtt{Vars(F)}$
  - $\mathtt{BV}(F) = \varnothing$
- *Si F est de la forme $\neg F_1$ ou (F), alors*
  - $\mathtt{FV}(F) = \mathtt{FV}(F_1)$
  - $\mathtt{BV}(F) = \mathtt{BV}(F_1)$
- *Si F est de la forme $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \Rightarrow F_2$, $F_1 \Leftrightarrow F_2$, alors*
  - $\mathtt{FV}(F) = \mathtt{FV}(F_1) \cup \mathtt{FV}(F_2)$
  - $\mathtt{BV}(F) = \mathtt{BV}(F_1) \cup \mathtt{BV}(F_2)$
- *Si F est de la forme $\forall x\ F_1$ ou $\exists x\ F_1$, alors*
  - $\mathtt{FV}(F) = \mathtt{FV}(F_1) \setminus \{x\}$
  - $\mathtt{BV}(F) = \mathtt{BV}(F_1) \cup \{x\}$

Au boulot (1/2)...

Récupérez l'archive `05_free.zip` depuis le Moodle.

Elle contient des analyseurs lexical et syntaxique, permettant de construire une représentation de formules lues depuis un fichier ou l'entrée standard et d'interroger si une variable est libre.

1. Renommez le fichier `free-eleves.ml` en `free.ml`
2. Invoquez la commande `touch .depend`
3. Invoquez la commande `make depend`
4. Invoquez la commande `make` pour finalement compiler.
5. Testez l'exécutable obtenu en lançant `./free.x` et en saisissant une formule, `?`, une variable puis un `.` final.

Syntaxe des formules : la même que précédemment avec en plus

- `P (x)` `Q (f (x, g (y)), z)`
- `all x, P (x, y, z)` `ex y, Q /\ R (y)`

Interrogation variable est libre ou pas : *formule* `?` *variable*.

Au boulot (2/2)...

Regardons ensemble le fichier `ast.ml` qui contient les types permettant de représenter les formules.

Dans `free.ml`, complétez les fonctions :

- `is_free ~var_x ~in_p =`
- `let is_free_gamma ~var_x ~in_g =`

`is_free_gamma` : est-ce qu'une variable est libre dans une des formules du contexte Γ.

Remarque : notons l'utilisation d'arguments nommés d'OCaml pour une meilleure lisibilité.

Tests : fichier `tests.pr`.

Attention : ça nous resservira plus tard :-)

Formule close

Définition (Formule close)

Une formule F est dite close si $\mathtt{FV}(F) = \varnothing$.

Théorème

Soit une structure M, une formule F et deux valuations ν_1, ν_2. Si pour toute variable $x \in \mathtt{FV}(F)$ on a $\nu_1(x) = \nu_2(x)$ alors $[\![F]\!]^M_{\nu_1} = [\![F]\!]^M_{\nu_2}$.

- Démonstration par induction sur la forme de F.

Théorème

Soit une structure M, une formule close F, alors pour toutes valuations ν_1, ν_2, on a $[\![F]\!]^M_{\nu_1} = [\![F]\!]^M_{\nu_2}$.

- Démonstration triviale puisque $\mathtt{FV}(F)$ est $\varnothing$.
- En clair : l'interprétation d'une formule close ne dépend pas de la valuation.

Modèle d'une formule (en calcul des prédicats)

Définition (Clôture universelle)

Soit une formule F. La clôture universelle $\overrightarrow{\forall} F$ de F est la formule obtenue en quantifiant toutes les variables libres de F.

- $\overrightarrow{\forall}(\exists x, P(x) \wedge Q(y) \vee R(z)) =$
  - $\forall y, \forall z, \exists x, P(x) \wedge Q(y) \vee R(z)$
  - $\forall z, \forall y, \exists x, P(x) \wedge Q(y) \vee R(z)$

Définition (Modèle d'une formule $M \models \mathcal{F}$)

Une structure M est un modèle d'une formule F si et seulement si $[\![\overrightarrow{\forall} F]\!]_\nu^M = \top$ pour une valuation ν.

- Puisque $\overrightarrow{\forall} F$ est close, ν n'a aucune importance.
- Semblable à un modèle en logique propositionnelle :
  - structure et non juste valuation.

Formules satisfiables, valides

Définition (Formule satisfiable)

Soit une signature $\Sigma = (\mathcal{F}, \mathcal{R})$. Une formule F est satisfiable si et seulement si elle admet un modèle.

Définition (Formule valide, tautologie)

Soit une signature $\Sigma = (\mathcal{F}, \mathcal{R})$. Une formule F est valide si et seulement si toute structure d'interprétation sur Σ est un modèle de F.

- Semblable à une tautologie en logique propositionnelle.

Au boulot...

En vous inspirant de l'algorithme vu en tout début de cours pour vérifier automatiquement si une formule de logique propositionnelle est une tautologie, proposez-en un pour le calcul des prédicats.

Au boulot...

Soit la signature $\Sigma = (\{\}, \{(\mathtt{R}, 2)\})$.

La formule $\forall x\ \mathtt{R}(x, x)$ est-elle satisfiable, est-elle valide ?

Au boulot...

Soit la signature $\Sigma = (\{\}, \{(\mathtt{R}, 2)\})$.

La formule $(\exists x, \forall y, \mathtt{R}(x, y)) \Rightarrow (\forall y, \exists x, \mathtt{R}(x, y))$ est-elle satisfiable, est-elle valide ?

La déduction naturelle pour le calcul des prédicats

Substitution dans une formule

Définition (Substitution d'une variable par un terme $F[x \leftarrow t]$)

La substitution $F[x \leftarrow t]$ de la variable x par le terme t dans la formule F est définie inductivement par :

- $x[x \leftarrow t] = t$
- $y[x \leftarrow t] = y$ *si* $y \neq x$
- $f(t_1, \ldots, t_n)[x \leftarrow t] = f(t_1[x \leftarrow t], \ldots, t_n[x \leftarrow t])$
- $R(t_1, \ldots, t_n)[x \leftarrow t] = R(t_1[x \leftarrow t], \ldots, t_n[x \leftarrow t])$
- $\bot[x \leftarrow t] = \bot$
- $(F_1 \wedge F_2)[x \leftarrow t] = (F_1[x \leftarrow t]) \wedge (F_2[x \leftarrow t])$
- $(F_1 \vee F_2)[x \leftarrow t] = (F_1[x \leftarrow t]) \vee (F_2[x \leftarrow t])$
- $(F_1 \Rightarrow F_2)[x \leftarrow t] = (F_1[x \leftarrow t]) \Rightarrow (F_2[x \leftarrow t])$
- $(F_1 \Leftrightarrow F_2)[x \leftarrow t] = (F_1[x \leftarrow t)] \Leftrightarrow (F_2[x \leftarrow t)]$
- $(\forall x\ F)[x \leftarrow t] = \forall x\ F$
- $(\exists x\ F)[x \leftarrow t] = \exists x\ F$
- *Si* $y \neq x$, *alors* $(\forall y\ F)[x \leftarrow t] = \forall z\ F[y \leftarrow z][x \leftarrow t]$, *avec* $z \notin \mathtt{FV}(t) \cup \mathtt{FV}(F)$
- *Si* $y \neq x$, *alors* $(\exists y\ F)[x \leftarrow t] = \exists z\ F[y \leftarrow z][x \leftarrow t]$, *avec* $z \notin \mathtt{FV}(t) \cup \mathtt{FV}(F)$

Substitution et capture

Si $y \neq x$, alors $(\exists y\ F)[x \leftarrow t] = \exists z\ F[y \leftarrow z][x \leftarrow t]$, avec $z \notin \mathtt{FV}(t) \cup \mathtt{FV}(F)$

- Renommage de y en z.
- Variable z fraîche (i.e. n'apparaît nulle part).
- Remarque : pas utile si $y \notin \mathtt{FV}(t)$.
- Évite de lier des variables abusivement :
  - $\forall x\ P(x, y)[y \leftarrow Q(x)] \quad \not\rightarrow \quad \forall x\ P(x, Q(x))$
  - $\forall x\ P(x, y)[y \leftarrow Q(x)] \quad \rightarrow \quad \forall z\ P(z, Q(x))$

Au boulot (1/2)...

Programmons les substitutions.

Récupérez l'archive `06_subst.zip` depuis le Moodle.

Elle contient des analyseurs lexical et syntaxique, permettant de construire une représentation de formules et substitutions à y appliquer.

1. Renommez le fichier `subst-eleves.ml` en `subst.ml`
2. Invoquez les commandes `touch .depend`, `make depend`, `make`.
3. Testez l'exécutable obtenu en lançant `./subst.x` et en saisissant une formule et une substitution terminées par un `.` final.

Syntaxe des formules : la même que précédemment.

Substitution : *formule*`[`*var* `<-` *terme*`]`.

Au boulot (2/2)...

Vous allez avoir besoin d'une fonction qui génère des noms de variables uniques.

Proposez une fonction `gen_new_var` qui effectue ce traitement.

Dans `subst.ml`, complétez les fonctions :

- `subst_expr ~in_t ~var_x ~by_t =`
- `subst_term ~in_p ~var_x ~by_t =`

Remarque : notons l'utilisation d'arguments nommés d'OCaml pour une meilleure lisibilité.

Tests : fichier `tests.pr`.

Attention : ça nous resservira plus tard :-)

Modifications à faire aux règles de déduction

- Deux nouveaux connecteurs.
- Deux nouvelles règles :
  - d'introduction,
  - d'élimination.

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A}\,\forall i \quad \text{si } x \text{ n'apparaît pas libre dans } \Gamma \qquad \frac{\Gamma \vdash \forall x\ A}{\Gamma \vdash A[x \leftarrow t]}\,\forall e$$

$$\frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x\ A}\,\exists i \qquad \frac{\Gamma \vdash \exists x\ A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}\,\exists e \quad \text{si } x \text{ libre ni dans } \Gamma \text{ ni dans } B$$

Règles de la déduction naturelle (calcul des prédicats)

$$\frac{\text{si } A \in \Gamma}{\Gamma \vdash A}\ \textit{axiome} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\textit{intro} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\textit{elim} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\textit{elim}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\textit{intro} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\textit{intro} \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\textit{elim}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\textit{intro} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\textit{elim} \qquad \frac{\Gamma, A \vdash B \quad \Gamma, B \vdash A}{\Gamma \vdash A \Leftrightarrow B}\ \Leftrightarrow\textit{intro}$$

$$\frac{\Gamma \vdash A \Leftrightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \Leftrightarrow\textit{elim} \qquad \frac{\Gamma \vdash A \Leftrightarrow B \quad \Gamma \vdash B}{\Gamma \vdash A}\ \Leftrightarrow\textit{elim} \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg\textit{intro}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg\textit{elim} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot\textit{elim} \qquad \left(\frac{}{\Gamma \vdash A \vee \neg A}\ \textit{LEM}\ \text{en logique classique}\right)$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A}\ \forall i \quad \text{si } x \text{ n'apparaît pas dans } \Gamma \qquad \frac{\Gamma \vdash \forall x\ A}{\Gamma \vdash A[x \leftarrow t]}\ \forall e \qquad \frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x\ A}\ \exists i$$

$$\frac{\Gamma \vdash \exists x\ A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}\ \exists e \quad \text{si } x \text{ n'apparaît ni dans } \Gamma \text{ ni dans } B$$

Au boulot...

Commentez cette preuve :

$$\dfrac{\dfrac{\dfrac{}{\forall x\ \exists y\ R(x,y) \vdash \forall x\ \exists y\ R(x,y)}\,Ax}{\forall x\ \exists y\ R(x,y) \vdash \exists y\ R(y,y)}\,\forall e \quad \text{avec } [x \leftarrow y]}{\vdash (\forall x\ \exists y\ R(x,y)) \Rightarrow (\exists y\ R(y,y))}\,\Rightarrow i$$

Encore des ho/e-rreurs

$$\dfrac{\dfrac{P(x) \vdash P(x)}{P(x) \vdash \forall x\ P(x)}\forall i \quad \text{Erreur : } \in \mathtt{FV}(\Gamma)!}{\vdash P(x) \Rightarrow \forall x\ P(x)} \Rightarrow i$$

$$\dfrac{\dfrac{\dfrac{}{\exists x\ P(x) \vdash \exists x\ P(x)}Ax \qquad \dfrac{}{\exists x\ P(x), P(x) \vdash P(x)}Ax}{\exists x\ P(x) \vdash P(x)}\exists e \quad \text{Erreur : } x \in \mathtt{FV}(P(x)!)}{\vdash (\exists x\ P(x)) \Rightarrow P(x)}$$

La même formule qu'il y a quelques diapos

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \exists x'\ \forall y\ R(x',y)}\,Ax \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, \forall y\ R(x',y) \vdash \forall y\ R(x',y)}\,Ax}{\Gamma, \forall y\ R(x',y) \vdash R(x',y')}\,\forall e}{\Gamma, \forall y\ R(x',y) \vdash \exists x\ R(x,y')}\,\exists i \text{ avec } [x \leftarrow x']}{\Gamma \vdash \exists x\ R(x,y')}\,\exists e}{\Gamma = \exists x\ \forall y\ R(x,y) \vdash \forall y\ \exists x\ R(x,y)}\,\forall i \text{ avec } [y \leftarrow y']}{\vdash (\exists x\ \forall y\ R(x,y)) \Rightarrow (\forall y\ \exists x\ R(x,y))}\,\Rightarrow i$$

Complétude de la déduction naturelle

Théorème (Correction de la DN pour le calcul des prédicats)

Si une formule P est démontrable dans un contexte Γ alors P est une conséquence de Γ.
Plus formellement, si $\Gamma \vdash P$ alors $\Gamma \models P$.

Théorème (Complétude de la DN pour le calcul des prédicats)

Si un contexte Γ a pour conséquence une formule P alors P est démontrable dans Γ.
Plus formellement, si $\Gamma \models P$ alors $\Gamma \vdash P$.

- Rien de nouveau sous le soleil.
- Comme pour la DN en logique propositionnelle :-)

Le calcul des séquents pour le calcul des prédicats

Modifications à faire aux règles

- Même genre de choses que pour la déduction naturelle...
- Deux nouveaux connecteurs.
- Deux nouvelles règles :
  - gauche,
  - droites.

$$\frac{\Gamma, A[x \leftarrow t] \vdash B}{\Gamma, \forall x\ A \vdash B}\,\forall g \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A}\,\forall d \quad \text{si } x \text{ n'apparaît pas libre dans } \Gamma$$

$$\frac{\Gamma, A \vdash B}{\Gamma, \exists x\ A \vdash B}\,\exists g \quad \text{si } x \text{ libre ni dans } \Gamma \text{ ni dans } B \qquad \frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x\ A}\,\exists d$$

- Un petit plus dupliquer les hypothèses : contraction.
- Nécessaire pour $\exists g$:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A, \vdash B}\,Contr$$

Règles du calcul des séquents (calcul des prédicats)

$$\frac{}{A \vdash A}\,Ax \qquad \frac{\Gamma \vdash A}{\Gamma, B \vdash A}\,Thin \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\,Contr \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\,Perm$$

$$\frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}\,Cut \qquad \frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C} \Rightarrow g \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow d$$

$$\frac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \Leftrightarrow B \vdash C} \Leftrightarrow g \qquad \frac{\Gamma \vdash B \qquad \Gamma, A \vdash C}{\Gamma, A \Leftrightarrow B \vdash C} \Leftrightarrow g \qquad \frac{\Gamma, A \vdash B \qquad \Gamma, B \vdash A}{\Gamma \vdash A \Leftrightarrow B} \Leftrightarrow d$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C} \wedge g \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge d \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C} \vee g \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee d$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee d \qquad \frac{\Gamma \vdash A}{\Gamma, \neg A \vdash B} \neg g \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg d \qquad \frac{}{\Gamma, \bot \vdash A} \bot g$$

$$\left(\frac{}{A \vee \neg A}\,LEM \text{ en logique classique} \right) \qquad \frac{\Gamma, A[x \leftarrow t] \vdash B}{\Gamma, \forall x\ A \vdash B} \forall g$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A} \forall d \quad \text{si } x \text{ n'apparaît pas libre dans } \Gamma \qquad \frac{\Gamma, A \vdash B}{\Gamma, \exists x\ A \vdash B} \exists g \quad \text{si } x \text{ libre ni dans } \Gamma \text{ ni dans } B \qquad \frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x\ A} \exists d$$

Au boulot...

Démontrez en calcul des séquents la formule :

$$\forall x\ \forall y\ (P(x) \wedge P(y)) \Rightarrow (P(x) \vee P(y))$$

Au boulot...

Démontrez en calcul des séquents la formule :

$$(\forall x\ (P(x)) \Rightarrow (\exists y\ P(y))$$

Au boulot...

Déjà vue en déduction naturelle, démontrez en calcul des séquents la formule :

$$(\exists x\ \forall y\ R(x,y)) \Rightarrow (\forall y\ \exists x\ R(x,y))$$

Complétude du calcul des séquents

Théorème (Correction du CS pour le calcul des prédicats)

Si une formule P est démontrable dans un contexte Γ alors P est une conséquence de Γ.
Plus formellement, si $\Gamma \vdash P$ alors $\Gamma \models P$.

Théorème (Complétude du CS pour le calcul des prédicats)

Si un contexte Γ a pour conséquence une formule P alors P est démontrable dans Γ.
Plus formellement, si $\Gamma \models P$ alors $\Gamma \vdash P$.

- Rien de nouveau sous le soleil.
- Comme pour la DN en calcul des prédicats :-)

Au boulot...

Récupérez l'archive `07_cp_tool.zip` depuis le Moodle.

Elle contient des analyseurs lexical et syntaxique, permettant de construire une représentation de formules lues depuis un fichier ou l'entrée standard ainsi que la reconnaissance de commandes correspondant aux règles du calcul des séquents pour le calcul des prédicats.

1. Renommez le fichier `core-eleves.ml` en `core.ml`
2. Invoquez la commande `touch .depend`
3. Invoquez la commande `make depend`
4. Invoquez la commande `make` pour finalement compiler.
5. Testez l'exécutable obtenu en lançant `./cpt.x` et en saisissant une formule terminée par un `.` final.
6. L'outil passe en mode preuve et attend que vous rentriez des commandes.

Syntaxe des formules : la même que précédemment.

Syntaxe des commandes

- `->l` (A, B) $\dfrac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C} \Rightarrow g$
- `->r` $\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow d$
- `<->ll` (A, B) $\dfrac{\Gamma \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \Leftrightarrow B \vdash C} \Leftrightarrow g$
- `<->lr` (A, B) $\dfrac{\Gamma \vdash B \qquad \Gamma, A \vdash C}{\Gamma, A \Leftrightarrow B \vdash C} \Leftrightarrow g$
- `<->r` $\dfrac{\Gamma, A \vdash B \qquad \Gamma, B \vdash A}{\Gamma \vdash A \Leftrightarrow B} \Leftrightarrow d$
- `/\l` (A, B) $\dfrac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C} \wedge g$
- `/\r` $\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge d$
- `\/l` (A, B) $\dfrac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C} \vee g$
- `\/rl` $\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee d$
- `\/rr` $\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee d$
- `~l` (A) $\dfrac{\Gamma \vdash A}{\Gamma, \neg A \vdash B} \neg g$
- `~r` $\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg d$
- `Fl` $\dfrac{}{\Gamma, \bot \vdash A} \bot g$
- `ExclMid` $\dfrac{}{A \vee \neg A} LEM$
- `Ax` $\dfrac{}{A \vdash A} Ax$
- `alll` (x, A, t) $\dfrac{\Gamma, A[x \leftarrow t] \vdash B}{\Gamma, \forall x\ A \vdash B} \forall g$
- `allr` $\dfrac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A} \forall d$ si x n'apparaît pas libre dans Γ
- `exl` (x, A) $\dfrac{\Gamma, A \vdash B}{\Gamma, \exists x\ A \vdash B} \exists g$ si x libre ni dans Γni dans B
- `exr` (t) $\dfrac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x\ A} \exists d$

Au boulot...

Regardons ensemble :

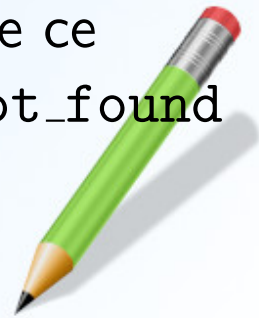
- `ast.ml` : types (des formules déjà vues) et des commandes.
- `core.ml` : squelette de la fonction qui va gérer le déroulement de la preuve dictée par l'utilisateur.
  - Règles droites déjà écrites (mêmes que règles *intro* de la DN).
  - Moteur de traitement des buts : même que pour notre outil en DN.

Attention : on a déjà écrit le test de variable libre et la substitution `:-)`

Simplification : on ignore les règles structurelles (*Perm*, *Contr*, *Thin*) et on remplace *Ax* par un test d'appartenance au contexte Γ.

Simplification : on est fainéants, pas de coupure `:-)`

Conseil : écrire une fonction `filter_gamma p gamma` qui recherche la première occurrence de la formule p dans le contexte Γ et retourne ce contexte privé de cette occurrence de formule. Lève l'exception `Not_found` si $p \notin \Gamma$.



Complétez ce squelette pour chaque cas de commandes.

Retour à Coq pour le calcul des prédicats

Quantificateurs du calcul des prédicats en Coq

- Quantification universelle : **forall** x : *type* , *P*.
- Quantification existentielle : **exists** x : *type* , *P*.
- Coq est typé : spécifier le type de la variable quantifiée.
- Parfois optionnel car Coq sait faire de l'inférence de types.

- Précédemment, **Variable** `A B` : **Prop**
- En fait, quantification universelle cachée sur des prédicats.
  - **Goal forall** `A B` : **Prop**, `(A /\ B) -> (B /\ A)`.

- Types prédéfinis : `Prop`, `nat`, `bool`...
- Possibilité de définir des types (par induction, plus tard).
- Autres types plus techniques (hors sujet) : `Set`, `Type`.
- « Prédicats prédéfinis » (bibliothèque standard) : =, <>, < (sur `nat`)...

Exemples

Goal forall `P : Prop, P -> P.`

Goal forall `P : nat -> Prop, forall x : nat, P x -> P x.`

Goal forall `LT : nat -> nat -> Prop, forall x y : nat,`
`LT x y -> ~ (LT y x).`

Goal forall `P : nat -> Prop, forall f : nat -> nat, forall x y : nat,`
`x = y -> f x = f y.`

- Possibilité de lier plusieurs variables de même type d'un coup.
- « Prédicat » : « fonction » dans **Prop**.
- « Fonction » : dans autre chose que **Prop**.
- Plusieurs arguments : comme en OCaml (curryfication).
- Logique d'ordre supérieure quantification sur fonctions (et prédicats).

Tactique quand le but est $\forall$

- **intro**.
- À peu près règle $\forall d$.

```
Goal forall P : nat -> Prop, forall x y : nat, x = y -> P x = P y.
1 goal
  ============================
   forall (P : nat -> Prop) (x y : nat), x = y -> P x = P y
Unnamed_thm < intro.
1 goal
  P : nat -> Prop
  ============================
   forall x y : nat, x = y -> P x = P y
Unnamed_thm < intros.
1 goal
  P : nat -> Prop
  x, y : nat
  H : x = y
  ============================
  P x = P y
```

- Introduit la variable quantifiée en hypothèse.
- Similaire à **intro** pour ->

Tactique pour utiliser une hypothèse $H : \forall$

- **apply** *H*.

```
Coq < Goal forall P : nat -> Prop, ((forall x : nat, P x) -> P 42).
1 goal
  ============================
   forall P : nat -> Prop, (forall x : nat, P x) -> P 42
Unnamed_thm < intros.
1 goal
  P : nat -> Prop
  H : forall x : nat, P x
  ============================
  P 42
Unnamed_thm < apply H.
No more goals.
```

- Vérifie que le but est bien une substitution du corps de l'hypothèse $\forall$
  - $\texttt{P 42} = (\texttt{P x})[\texttt{x} \leftarrow 42]$.
- Vue volontairement simpliste : on y reviendra un peu plus tard.

Tactique quand le but est $\exists$

- Objectif : exhiber un témoin.
- **exists** t.
- Règle $\exists d$.

```
Goal forall P : nat -> Prop, P 42 -> exists x : nat, P x.
1 goal
  ============================
   forall P : nat -> Prop, P 42 -> exists x : nat, P x
Unnamed_thm < intros.
1 goal
  P : nat -> Prop
  H : P 42
  ============================
   exists x : nat, P x
Unnamed_thm < exists 42.
1 goal
  P : nat -> Prop
  H : P 42
  ============================
  P 42
```

Tactique pour utiliser une hypothèse $H : \exists$

- **destruct** *H*.

```
Goal forall P : bool -> Prop, (exists x, P x) -> (P true \/ P false).
 goal
  ============================
   forall P : bool -> Prop, (exists x : bool, P x) -> P true \/ P false

Unnamed_thm < intros.
1 goal
  P : bool -> Prop
  H : exists x : bool, P x
  ============================
  P true \/ P false
Unnamed_thm < destruct H.
1 goal
  P : bool -> Prop
  x : bool
  H : P x
  ============================
  P true \/ P false
```

- (Suite de la preuve par cas sur `x : bool` par **destruct** `x`.)

Mémo (v2)

```
assumption      split                  left              right             intro
H : P                                                                          H : P
===== --> .     ====== --> ===  ===     ====== --> ===    ====== --> ===     ====== --> =====
P               P /\ Q     P    Q       P \/ Q     P      P \/ Q     Q      P -> Q     Q
```

```
split                          intro            intro                        exists v
                                       H : P                      x : t
======= --> ======  ======     === --> =====    ============== --> =====     ============== --> =========
P <-> Q     P -> Q  Q -> P     ~P      False    forall x : t, P     P         exists x : t, P     P[x <- v]
```

```
destruct H              destruct H                  apply H
            H : P
H : P /\ Q  H0 : Q      H : P \/ Q     H : P  H : Q  H : P -> Q     H : P -> Q
========== --> ======   ========== --> =====  =====  ========== --> ==========
R           R           R              R      R      Q              P
```

```
apply H                        apply H
H : P <-> Q     H : P <-> Q    H : P <-> Q     H : P <-> Q
=========== --> ===========    =========== --> ===========
P               Q              Q               P
```

```
apply H                                        destruct H
                                                                 x : t
H : forall x, Q -> P x     H : forall x, Q -> P x     H : exists x : t, P x     H : P x
====================== --> ======================    ===================== --> ========
P v                        Q                          Q                         Q
```

```
apply H              contradiction      contradiction
                                        H : P
H : ~P     H : ~P    H : False          H0 : ~P
====== --> ======    ========= --> .    ======= --> .
False      P         P                  Q
```

Au boulot...

À vous de démontrer les formules suivantes, pour P et Q opérant sur des `nat`. On vous laisse les introduire dans les formules.

1. $(\forall x\ P(x)) \Rightarrow (\exists x\ P(x))$
2. $\neg(\exists x\ Px) \Rightarrow \neg P(42)$
3. $(\forall x\ \forall y\ P(x,y)) \Rightarrow (\forall x\ P(x,x))$
4. $(\exists x\ \forall y\ P(x,y)) \Rightarrow (\forall y\ \exists x\ P(x,y))$
5. $(\exists x\ (P(x) \wedge Q(x))) \Rightarrow (\exists x\ (P(x) \vee Q(x)))$
6. $(\exists x\ P(x) \wedge Q(x)) \Rightarrow ((\exists x\ P(x)) \wedge (\exists x\ Q(x)))$
7. $(\forall x\ P(x)) \Rightarrow \neg(\exists x\ \neg P(x))$
8. $(\exists x\ \neg P(x)) \Rightarrow \neg(\forall x\ P(x))$

Rappel : Coq sait faire de l'inférence types `:-)` Pas obligé d'annoter les types partout.

Utiliser une hypothèse $H : \forall$: la vérité

- Exemples simples jusqu'à présent.

```
  P : nat −> Prop
  H :  forall  x : nat, P x
  ============================
  P 42
Unnamed_thm < apply H.
No more goals.
```

- But courant résolu par le **apply**.

- Hypothèse peut être de la forme $H : \forall x\ P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow P$.
- Effet de **apply** :
  - Vérifie que le but est obtenu par une substitution $[x \leftarrow t]$ sur P.
  - Génère autant de sous-buts $P_1[x \leftarrow t], \ldots, P_n[x \leftarrow t]$ à prouver.
- Hypothèse H peut être vue comme un « théorème ».
- $P_1, \ldots, P_n$: « hypothèses » nécessaires à l'application du théorème.

Au boulot...

Démontrez les formules suivantes et analysez les effets de **apply**.

Note : `S` d'arité 1 existe déjà en Coq.
`S n` (terme) est le successeur du `nat` n.
Donc `S (S (S 0))` c'est... 3.

1. $(\forall x\ P(x) \Rightarrow Q(x)) \Rightarrow P(0) \Rightarrow Q(0)$
2. $P(0) \Rightarrow (\forall x\ P(x) \Rightarrow P(S(x))) \Rightarrow P(S(S(S(0))))$
3. $(\forall x\ P(x) \vee Q(x) \Rightarrow R(x)) \Rightarrow (\exists y\ P(y) \vee Q(y)) \Rightarrow (\exists z\ R(z))$

Utiliser une hypothèse $H : \forall$ dans une hypothèse

- Déjà vu à propos du raisonnement « avant » : **apply** `H` **in** `H'`.
- Hypothèse peut être de la forme $H : \forall x\ P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow P$.
- Effet de **apply** `H` **in** `H'` :
  - Vérifie que H' est obtenue par une substitution $[x \leftarrow t]$ sur P_1.
  - Remplace H' par $P[x \leftarrow t]$
  - Génère autant de sous-buts $P_2[x \leftarrow t], \ldots, P_n[x \leftarrow t]$ à prouver.

```
1 goal
  P, Q, R, S : nat -> Prop
  H : forall x : nat, P x -> Q x -> R x -> S x
  H0 : P 42
  ============================
  S 42
Unnamed_thm < apply H in H0.
3 goals
  P, Q, R, S : nat -> Prop
  H : forall x : nat, P x -> Q x -> R x -> S x
  H0 : S 42
  ============================
  S 42
goal 2 is :                    goal 3 is :
 Q 42                           R 42
```

Au boulot...

Redémontrez les formules suivantes en mode « avant » et analysez les effets de **apply in**.

1. $(\forall x\ P(x) \Rightarrow Q(x)) \Rightarrow P(0) \Rightarrow Q(0)$
2. $P(0) \Rightarrow (\forall x\ P(x) \Rightarrow P(S(x))) \Rightarrow P(S(S(S(0))))$

Programmer ou démontrer, il faut choiser ?

Type vs formule

- On considère la logique minimale : uniquement la $\Rightarrow$.
- Soit la formule $(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow A \Rightarrow C$
- Soit le type OCaml $(\tau_1 \to \tau_2) \to (\tau_2 \to \tau_3) \to \tau_1 \to \tau_3$
- Se ressemblent quand même pas mal...

Au boulot...

Donnez l'arbre de preuve (sans coupure) de la formule (précédente) en déduction naturelle :

$$(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow A \Rightarrow C$$

Au boulot...

Trouvez une expression OCaml qui a le type (précédent) :

$$(\tau_1 \to \tau_2) \to (\tau_2 \to \tau_3) \to \tau_1 \to \tau_3$$

Règles de typage pour OCaml atrophié

- Ensemble infini dénombrable de variables de types : $\mathcal{V} = \{ \alpha_1, \alpha_2, \ldots \}$
- Algèbre de types : $\tau ::= \alpha \mid \tau \to \tau$
- Besoin de connaître le type des variables.
  - Environnement de typage $\Gamma : \mathcal{V} \to \tau$

$$\frac{x \in \mathit{Dom}(\Gamma) \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \mathit{Var} \qquad \frac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash (\texttt{fun x -> } e) : \tau_1 \to \tau_2}\ \mathit{Fun}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ e_2) : \tau}\ \mathit{App}$$

Au boulot...

Donnez l'arbre de typage de cette expression OCaml.

Comparaison (1)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \vdash A \Rightarrow B} \quad \dfrac{}{\ldots \vdash A}}{\ldots \vdash B} \quad \dfrac{}{\ldots \vdash B \Rightarrow C}}{A \Rightarrow B, B \Rightarrow C, A \vdash C}}{A \Rightarrow B, B \Rightarrow C \vdash A \Rightarrow C}}{A \Rightarrow B \vdash (B \Rightarrow C) \Rightarrow A \Rightarrow C}\Rightarrow i}{(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow A \Rightarrow C}\Rightarrow i$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots \vdash f : \tau_1 \to \tau_2}\,Var \quad \dfrac{}{\ldots \vdash x : \tau_1}\,Var}{\ldots \vdash f\ x : \tau_2}\,App \quad \dfrac{}{\ldots \vdash g : \tau_2 \to \tau_3}\,Var}{(f, \tau_1 \to \tau_2), (g, \tau_2 \to \tau_3), (x, \tau_1) \vdash g\ (f\ x) : \tau_3}\,App}{(f, \tau_1 \to \tau_2), (g, \tau_2 \to \tau_3) \vdash \mathtt{fun}\ x \to g\ (f\ x) : \tau_1 \to \tau_3}\,Fun}{(f, \tau_1 \to \tau_2) \vdash \mathtt{fun}\ g \to \mathtt{fun}\ x \to g\ (f\ x) : (\tau_2 \to \tau_3) \to \tau_1 \to \tau_3}\,Fun}{\vdash \mathtt{fun}\ f \to \mathtt{fun}\ g \to \mathtt{fun}\ x \to g\ (f\ x) : (\tau_1 \to \tau_2) \to (\tau_2 \to \tau_3) \to \tau_1 \to \tau_3}\,Fun$$

Comparaison (2)

$$\frac{\text{si } A \in \Gamma}{\Gamma \vdash A}\,Ax \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\Rightarrow i$$

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\Rightarrow e$$

$$\frac{x \in Dom(\Gamma) \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau}\,Var \qquad \frac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash (\texttt{fun x -> } e) : \tau_1 \to \tau_2}\,Fun$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ e_2) : \tau}\,App$$

Intermède : λ-calcul

- OCaml avec juste fonction et application : $\equiv$ λ-calcul (simplement typé).
- λ-termes définis inductivement par :

$$\begin{array}{rcll} t & ::= & x & \text{variable} \\ & | & t\ t & \text{application} \\ & | & \lambda x.t & \text{abstraction (i.e. fonction)} \end{array}$$

- λ-calcul : modèle de calcul.
- Règle de calcul, β-réduction : $(\lambda x.t)\ t' \to t[x \leftarrow t']$
- Alonzo Church, 193x.
- Puissance d'expression des fonctions calculables.
- λ-calcul $\approx$ machines de Turing.
- Beaucoup de choses à dire dessus : on ne rentre pas dans les détails.

Correspondance de Curry-Howard

- Concerne la logique intuitionniste en déduction naturelle.
- Met en bijection formules et types.
- Met en bijection preuves et λ-termes.

- Différentes logiques (pouvoir d'expression).
- Différentes « variantes » de λ-calcul.
- Formules plus compliquées $\Rightarrow$ λ-calcul sous-jacent plus compliqué.

- Conséquences :
  - Si on arrive à trouver un terme d'un certain type, alors la formule correspondante est démontrable.
  - Typer un terme c'est trouver de quelle formule le programme est la preuve.
  - On peut reconstruire une preuve en inférant le type du terme.
  - Le terme est une syntaxe pour la preuve.
  - Programmer c'est exhiber un témoin qu'une formule est démontrable.

La minute anecdote historique

- 1957 - Haskell Curry
  - Cadre de la logique combinatoire.
  - Formules $\leftrightarrow$ types.
  - Démontrable $\leftrightarrow$ type est habité.
- 1960 - William Alvin Howard
  - Formules $\leftrightarrow$ types.
  - Preuves $\leftrightarrow$ termes.
  - Notes manuscrites qui ont fait le tour du monde.
  - Publication en 1980 !

Au boulot...

Trouvez une expression OCaml ayant le type suivant :

$$(\tau_1 \to \tau_2) \to \tau_1 \to (\tau_2 \to \tau_3) \to \tau_4$$

Qu'en concluez-vous sur la formule correspondante en logique propositionnelle ?

Interprétation BHK pour la logique minimale

- Luitzen Egbertus Jan Brouwer, Arend Heyting, Andreï Kolmogorov.
- Interprète une formule par le type de ses preuves (pas par un booléen).
- Type d'une formule $[\![F]\!]_T$ défini inductivement par :
  - Variables propositionnelles (distinctes) $\mapsto$ variables de types (distinctes)
    - $[\![A]\!]_T = \tau_1, [\![B]\!]_T = \tau_2, \ldots$
  - Implication $\mapsto$ fonction
    - $[\![F_1 \Rightarrow F_2]\!]_T = [\![F_1]\!]_T \to [\![F_2]\!]_T$

Théorème (Correspondance de Curry-Howard)

1. *Si une formule F est démontrable à partir des hypothèses $C_1, \ldots, C_n$ alors il existe un terme t de type $[\![F]\!]_T$ dans l'environnement de typage $(x_1 : [\![C_1]\!]_T), \ldots, (x_n : [\![C_n]\!]_T)$.*
2. *Si le terme t a pour type τ dans l'environnement de typage $(x_1 : [\![C_1]\!]_T), \ldots, (x_n : [\![C_n]\!]_T)$, alors il existe une formule F telle que $[\![F]\!]_T = \tau$, F est démontrable dans le contexte $C_1, \ldots, C_n$ où $[\![C_1]\!]_T = \tau_1, \ldots, [\![C_n]\!]_T = \tau_n$.*

Démonstration (1), règle *Ax*

- Cas preuve de F obtenue par règle Ax.

Hypothèse **H0** : $\dfrac{\text{si F} \in C_1, \ldots, C_n}{C_1, \ldots, C_n \vdash F}\, Ax$

Prouvons $\exists t : [\![F]\!]_T$ dans l'env^t $\Gamma = (x_1 : [\![C_1]\!]_T), \ldots, (x_n : [\![C_n]\!]_T)$

Par H0, F est C_i l'une des formules du contexte

Prouvons $\exists t : [\![C_i]\!]_T$ dans l'env^t $\Gamma = (x_1 : [\![C_1]\!]_T), \ldots, (x_n : [\![C_n]\!]_T)$

Hypothèse **H1** : dans l'env^t Γ, on a $x_i : [\![C_i]\!]_T$

Par la règle *Var* qui dit $\dfrac{x \in Dom(\Gamma) \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau}\, Var$

on déduit que le terme t cherché est x_i.

Démonstration (2), règle $\Rightarrow i$

- Cas preuve de F obtenue par règle $\Rightarrow i$.

Hypothèse **H0** : $\dfrac{C_1, \ldots, C_n, F_1 \vdash F_2}{C_1, \ldots, C_n \vdash F_1 \Rightarrow F_2} \Rightarrow i$

Prouvons $\exists t : [\![F_1 \Rightarrow F_2]\!]_T$ dans l'env$^{\mathsf{t}}$ $\Gamma = (x_1 : [\![C_1]\!]_T), \ldots, (x_n : [\![C_n]\!]_T)$

Hypothèse d'induction **H1** : $\exists t_2 : [\![F_2]\!]_T$ dans l'env$^{\mathsf{t}}$ $\Gamma, (x : [\![F_1]\!]_T)$.

Par H1 et la règle *Fun* qui dit $\dfrac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash (\texttt{fun x -> } e) : \tau_1 \to \tau_2} Fun$

on déduit **H2** : $\texttt{fun x -> } t_2 : [\![F_1]\!]_T \to [\![F_2]\!]_T$

CQFD car $[\![F_1]\!]_T \to [\![F_2]\!]_T = [\![F_1 \Rightarrow F_2]\!]_T$

Au boulot...

- Cas preuve de F obtenue par règle $\Rightarrow e$.

Hypothèse **H0** : $\dfrac{C_1, \ldots, C_n \vdash F_0 \Rightarrow F \quad \Gamma \vdash F_0}{C_1, \ldots, C_n \vdash F} \Rightarrow elim$

Prouvons ...

Démonstration (4, règle *Var*)

- Cas t typé avec la règle *Var*.

Hypothèse **H0** : $\dfrac{x_i \in Dom(\Gamma) \qquad \Gamma(x_i) = \tau}{\Gamma = (x_1 : [\![C_1]\!]_T), \ldots, (x_n : [\![C_n]\!]_T) \vdash x_i : \tau}\ Var$

Prouvons $\exists F$ t.q. $[\![F]\!]_T = \tau$ et $C_1, \ldots C_n \vdash F$.

Par H0, on a **H1** : $x_i : [\![C_i]\!]_T$.

Donc le F cherché est C_i.

Montrons que $C_1, \ldots C_n \vdash C_i$.

CQFD par la règle Ax qui dit $\dfrac{\text{si } A \in \Gamma}{\Gamma \vdash A}\ Ax$

Démonstration (5, règle *Fun*)

- Cas t typé avec la règle *Fun*.

Notation : $\Gamma = (x_1 : [\![C_1]\!]_T), \ldots, (x_n : [\![C_n]\!]_T)$

Hypothèse **H0** : $$\frac{\Gamma, (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash (\texttt{fun x -> } e) : \tau_1 \to \tau_2}\ Fun$$

Prouvons $\exists F$ t.q. $[\![F]\!]_T = \tau_1 \to \tau_2$ et $C_1, \ldots C_n \vdash F$.

Hypothèse d'induction **H1** : $\exists F_2$ t.q. $[\![F_2]\!]_T = \tau_2$

et **H2** : $C_1, \ldots, C_n, C_x \vdash F_2$ avec $[\![C_x]\!]_T = \tau_1$

Par H2 et la règle $\Rightarrow i$ qui dit que $$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow i$$

on déduit **H3** : $C_1, \ldots, C_n \vdash C_x \Rightarrow F_2$

Donc le F recherché est $C_x \Rightarrow F_2$

Montrons que $[\![C_x \Rightarrow F_2]\!]_T = \tau_1 \to \tau_2$

$[\![C_x \Rightarrow F_2]\!]_T = [\![C_x]\!]_T \to [\![F_2]\!]_T$

Par H3, on déduit **H4** : $[\![C_x \Rightarrow F_2]\!]_T = \tau_1 \to [\![F_2]\!]_T$

CQFD par H3, H4 $[\![C_x \Rightarrow F_2]\!]_T = \tau_1 \to \tau_2$

Démonstration (6, règle *App*)

- Cas t typé avec la règle *App*.

Notation : $\Gamma = (x_1 : [\![C_1]\!]_T), \ldots, (x_n : [\![C_n]\!]_T)$

Hypothèse **H0** : $$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ e_2) : \tau}\ App$$

Prouvons $\exists F$ t.q. $[\![F]\!]_T = \tau$ et $C_1, \ldots C_n \vdash F$.

Hypothèse d'induction **H1** : $\exists F_1$ t.q. $[\![F_1]\!]_T = \tau_2 \to \tau$
et **H2** : $C_1, \ldots C_n \vdash F_1$
et **H3** : $\exists F_2$ t.q. $[\![F_2]\!]_T = \tau_2$
et **H4** : $C_1, \ldots C_n \vdash F_2$
Par H1, hypothèse **H5** : $F_1 = F_1' \Rightarrow F_1''$ avec $[\![F_1']\!]_T = \tau_2$ et $[\![F_1'']\!]_T = \tau$
Par H3, H5, hypothèse **H6** : $F_1' = F_2$
Par H5, H6, hypothèse **H7** : $F_1 = F_2 \Rightarrow F_1''$

Par H2, H4, H7 et la règle $\Rightarrow e$ $$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow e$$

on déduit **H8** $C_1, \ldots C_n \vdash F_1''$
Donc le F recherché est F_1''
Montrons que $[\![F_1'']\!]_T = \tau$
CQFD par H5.

Extension aux formules avec $\wedge$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge i$$

- Preuve de $A \wedge B$: couple d'une preuve de A et d'une preuve de B.
- Rajouter des couples dans notre λ-calcul.
- Pouvoir typer les couples.

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge e \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge e$$

- Élimination du $\wedge$: besoin d'extraire A ou B d'un couple.
- Rajouter les projections droite et gauche dans notre λ-calcul :
  - `fst`
  - `snd`
- Pouvoir typer les projections.

Ajout des couples dans le λ-calcul

- Algèbre de types : $\tau ::= \alpha \mid \tau \to \tau \mid \tau \times \tau$
- Termes : $t ::= x \mid t\ t \mid \lambda x.t \mid (t, t) \mid \texttt{fst t} \mid \texttt{snd t}$
- Nouvelles règles de typage :

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2}\,\mathit{Pair} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ t : \tau_1}\,\mathit{Fst} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\ t : \tau_2}\,\mathit{Snd}$$

Ajout des couples dans la correspondance

- Conjonction $\mapsto$ produit.
- $[\![A \wedge B]\!]_T = [\![A]\!]_T \times [\![B]\!]_T$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge i \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,\wedge e \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,\wedge e$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2}\,Pair \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ t : \tau_1}\,Fst \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}\ t : \tau_2}\,Snd$$

- La correspondance tient toujours.
- Même style de preuve.

Autres extensions pour la logique propositionnelle

- Reste de la logique propositionnelle intuitionniste.
  - Le $\vee$:
    - ⋆ Types sommes.
    - ⋆ Injections dans un type somme.
    - ⋆ Filtrage (`match with` d'OCaml).
  - $\bot$ et $\top$
    - ⋆ Type vide.
    - ⋆ Type « unit ».
  - Le $\neg$: $\neg A$ c'est juste $A \Rightarrow \bot$
- Logique propositionnelle classique.
  - Besoin de continuations.
    - ▶ $\approx$ Fonction qui représente un calcul en suspens.
  - Besoin d'une construction spéciale (`callcc`).
  - $\lambda\mu$ calcul de Parigot.
  - Hors du sujet de ce cours.

Extension au premier ordre : $\forall$

- Idée : preuve de $\forall x\ F$ est une fonction $t \longrightarrow F[x \leftarrow t]$.
- Quel type pour $[\![\forall x\ F]\!]_T$?
- Besoin de paramétrer les types par des termes.
- Types dépendants : produit dépendant.

- $\lambda\Pi$-calcul.
- Exemple : f fonction qui crée une « liste de booléens » de longueur n.
  - $f\ 0 = [] : (\texttt{blist}\ 0)$.
  - $f\ 3 = [\mathit{false}; \mathit{false}; \mathit{false}] : (\texttt{blist}\ 3)$.
- Besoin de lier des termes dans les types.
- $\Pi_{x:\tau_1}\ \tau_2(x)$
- f a pour type $\Pi_{n:\texttt{int}}\ (\texttt{blist}\ n)$.
- Remarque : $\texttt{int} \longrightarrow \texttt{bool} \equiv \Pi_{n:\texttt{int}}\ \texttt{bool}$ (n non utilisé)

Extension au premier ordre : $\exists$

- Idée : preuve de $\exists x\ F$ est un couple $(t, F[x \leftarrow t])$.
- Quel type pour $[\![\exists x\ F]\!]_T$?
- Autre forme de dépendance : somme dépendante.

- Étendre le $\lambda\Pi$-calcul en $\lambda 1$-calcul.
- Rajouter types produit, union, vide.
- $\Sigma_{x:\tau_1}\ \tau_2(x)$.
- Remarque : $\texttt{int} \times \texttt{bool} \equiv \Sigma_{x:\texttt{int}}\ \texttt{bool}$. ($x$ non utilisé)

Programmer et démontrer

Un λ-calcul (beaucoup) plus riche

- Extensions de Curry-Howard $\Rightarrow$ λ-calculs avec plus de constructions.
- On s'est rapprochés de « vrais » langages de programmation.
- Coq : logique très expressive $\Rightarrow$ constructions très expressives.

- But : programmer puis démontrer des propriétés sur ces programmes.
- Pas une étude de toutes les possibilités de Coq.
- Ce qu'il faut pour commencer à s'amuser (et suer des neurones).

- Pour aller plus loin :
  `https://softwarefoundations.cis.upenn.edu`
- Volume 1 — les autres, pour aller **beaucoup** plus loin.

Traits de programmation

- Jusqu'à présent : uniquement des propriétés et des démonstrations.
- Pour programmer : définir des types de données.
- Pour programmer : définir des fonctions.
- Programmation exclusivement fonctionnelle.
- Rappelle OCaml... en parfois plus contraignant.
- Exemples :
  - fonctions récursives doivent terminer,
  - pas d'exceptions,
  - inférence de types pas toujours possible,
  - polymorphisme explicite.
- Contrepartie : on peut démontrer formellement des choses.

Définir des types somme : les inductifs

- Non polymorphes pour commencer.
- **Inductive** `couleur` := `Rouge` | `Vert` | `Bleu`.
- **Check** `couleur`. ⟶ `Set`.
- `Set` : type des « ensembles » (`nat`, `bool`...)
- **Inductive** `int` := `Zero` | `Succ (n : int)`.
- Type inductif récursif.
- Remarque : `nat` défini inductivement (« 2 » : notation pour `S (S O)`).
- **Inductive** `ilist` := `Nil` | `Cons (n : nat) (l : ilist)`.
- Constructeurs de valeurs ≡ fonctions.
- `Succ (Succ Zero)` `Cons 3 (Cons 4 Nil)`

Au boulot...

Donnez en Coq la définition du type `expr_t` représentant les formules de la logique propositionnelle. On vous propose de nommer les constructeurs : `ETrue`, `EFalse`, `EPred`, `ENot`, `EAnd`, `EOr`, `EImpl` et `EEquiv`.

Pour accéder au type `string` et ses notations, débutez votre session par :

From `Coq` **Require Import** `Strings.String`.
Open Scope `string_scope`.

Construisez ensuite le terme représentant la formule $A \wedge B \Rightarrow B$ et appelez la commande **Check** en lui passant ce terme en argument pour vérifier son type.

Les expressions

- Valeurs de types sommes.
  - Définis par nous ou prédéfinis dans la bibliothèque standard.
  - Parfois besoin d'un **Require Import** `xyz`. qui va bien.
  - `true`, `42`, `S (S (S O))`.
- Fonctions :
  - `fun x => x + 1`
  - `fun x : bool => x`
  - De nombreuses prédéfinies dans la bibliothèque standard (e.g. +).
  - Parfois besoin d'un **Require Import** `xyz`. qui va bien.
- Filtrage :
  - `fun x =>` **match** `x` **with true** $\Rightarrow 0 \mid$ **false** $\Rightarrow 1$ **end**
  - **end** obligatoire.
  - Doit être exhaustif.
  - Cas « tous les autres » : `_`
- Conditionnelle :
  - `fun x =>` **if** `x =?` 0 **then** 1 **else** 0
  - Attention : opère sur des `bool` (on y reviendra).

Les définitions

- Besoin de nommer des expressions.
- **Definition** *nom* := *expr* .

```
Definition two := S (S O).
Definition is_null := fun x => match x with O => true | _ => false
    end.
```

- Syntaxe abrégée pour les fonctions :

```
Definition next n := S n.
Definition is_null x := match x with O => true | _ => false end.
```

- Inférence de type souvent, mais parfois besoin d'expliciter les types.

```
Definition id x := x.
> Definition id x := x.
> ^^^^^^^^^^^^^^^^^^^^
Error:
The following term contains unresolved implicit arguments:
  (fun x : ?T => x)
Definition id (x : nat) := x.
Definition id' (x : nat) : nat := x.
```

La conditionnelle, l'égalité, `bool`, **Prop**

- Source de confusion majeure chez les débutants.
- **Prop** : type des propositions (formules).
  - **Prop** = { **False**, **True** }.
  - Avec des majuscules.
  - Pas de filtrage dessus.
  - Opérateurs relationnels logiques : = / <>, ~, <, <=, >, >= : dans **Prop**.
  - Comme \/, /\, =>, <=>.
- `bool` : type booléen calculatoire.
  - `bool` = { `false`, `true` }.
  - Avec des minuscules.
  - Filtrage, conditionnelle dessus.
  - Opérateurs relationnels calculatoires : =?,<>?, `negb`, <?, <=?, >?, >=?, `orb`, `andb` : dans `bool`.
- **Prop** : pour des formules (et des preuves).
- `bool` : pour des programmes.
- On peut (bien sûr) faire le lien entre les deux
  - mais on n'étudiera pas ce point.

Au boulot...

Vous vous souvenez des tables de vérité qui donnaient la sémantique des connecteurs ?

Définissez un type `bool_t`, de constructeurs `T` et `F`, isomorphe aux booléens.

Écrivez les définitions correspondant à la sémantique « booléenne » des connecteurs de la logique propositionnelle.

Vous les nommerez `sem_true`, `sem_false`, `sem_and`, `sem_or`, `sem_impl` et `sem_equiv`.

Calculer en Coq

- **Check** : affiche le type.
- **Compute** : calcule et affiche la valeur.

```
Coq < Definition falso_quodlibet := sem_impl F T.
falso_quodlibet is defined
Coq < Check (falso_quodlibet).
falso_quodlibet
     : bool_t
Coq < Compute (falso_quodlibet).
     = T
     : bool_t
```

Au boulot...

Écrivez la fonction `is_binary` qui retourne **True** si une formule est constituée d'un constructeur de tête d'arité 2, **False** sinon.

Remarque : nous choisissons de la définir dans **Prop** et non dans `bool` car se sera plus simple pour faire des preuves.

Énoncez la propriété (formule) Coq qui dit que si une formule est `is_binary`, alors soit c'est un « et », un « ou », un « implique » ou bien un « équivalent ».

Quel est l'idée (l'argumentaire) de démonstration de cette propriété ?

Preuve par cas sur une hypothèse inductive

- **destruct** *H* **eqn**: *nom*.
- Génère autant de sous-buts que de cas possibles pour la forme de *H*.
- **eqn**: *nom* : permet de conserver en hypothèse le lien avec *H*.
- **eqn**: pas toujours nécessaire (mais ne coûte rien).

```
Coq < Goal forall b : bool, b = true \/ b = false.
1 goal
  ============================
   forall b : bool, b = true \/ b = false
Unnamed_thm < intros.
1 goal
  b : bool
  ============================
  b = true \/ b = false
Unnamed_thm < destruct b eqn: H_what_is_b.
2 goals
  b : bool
  H_what_is_b : b = true
  ============================
  true = true \/ true = false
goal 2 is:
 false = true \/ false = false
```

Preuve par « définition » d'une expression (1/2)

- « Dépliage » d'une définition : **unfold** *nom*.
  - Remplace les occurrences de *nom* par le corps de la définition de *nom*.
  - **unfold** *nom* **in** *H* : idem dans une hypothèse *H*.

```
Definition mkpair (x y : nat) := (x, y).
Coq < Goal forall x y : nat, x = y -> mkpair x x = mkpair x y.
1 goal
  ============================
   forall x y : nat, x = y -> mkpair x x = mkpair x y
Unnamed_thm < intros.
1 goal
  x, y : nat
  H : x = y
  ============================
  mkpair x x = mkpair x y
Unnamed_thm < unfold mkpair.
1 goal
  x, y : nat
  H : x = y
  ============================
  (x, x) = (x, y)
```

Preuve par « définition » d'une expression (2/2)

- « simplification » : **simpl**.
  - Effectue des réécritures heuristiques (et des dépliages) dans le but.
  - **simpl in** H : idem dans une hypothèse H.
  - Ne « simplifie » pas toujours :
    - ★ peut ne rien changer,
    - ★ peut donner un terme nettement plus « gros ».
- Stratégie : on tente **simpl** et sinon (**Undo**) des **unfold** ciblés.

```
Unnamed_thm < intros.
1 goal
  x, y : nat
  H : x = y
  ============================
  mkpair x x = mkpair x y
Unnamed_thm < simpl.
1 goal
  x, y : nat
  H : x = y
  ============================
  mkpair x x = mkpair x y  (* Ben... raté :/ *)
```

Au boulot...

Reprenez la propriété de tout à l'heure, à propos de `is_binary` et faites-en la démonstration.

Plutôt que d'utiliser **Goal** pour l'introduire, donnez lui un nom avec **Lemma** ou **Theorem**.

Lemma `is_binary_spec` : **forall** ...

Une fois prouvée, elle deviendra un théorème que nous pourrons utiliser plus tard :-)

Indication : Quand le but est une égalité triviale $t = t$, on utilise la tactique **reflexivity** . On reviendra sur l'égalité plus tard.

Les fonctions récursives

- Sujet pouvant être très compliqué en Coq.
- Doivent toujours terminer.
- Terminaison à démontrer dans le cas général.
- Terminaison structurelle : Coq sait vérifier.
- Décroissance structurelle :
  - Filtrage des types inductifs.
  - Il existe des cas de base (constructeurs 0-aires).
  - Chaque appel récursif est fait sur des arguments strictement plus petits.
  - Filtrage exhaustif.
  - Taille : nombre de constructeurs dans le terme.
- `S (S n)` $<_{struct}$ `S (S (S (n)))`.
- `(n - 1)` $\not<_{struct}$ `n`.

Les fonctions récursives structurelles

- **Fixpoint** *nom* $(arg_1 : ty_1)$ ... $(arg_n : ty_n)$ **:=** *expr*.
- Pour spécifier explicitement l'argument décroissant : { **struct** *arg* }.

Fixpoint `idnat (n : nat)` := **match** `n` **with** `O => O | S m => idnat m` **end**.

Fixpoint `idnat_stupid (n : nat) (dummy : nat) { struct n}` :=
  match `n` **with** `O => O | S m => idnat_stupid m dummy` **end**.

- Définition de type inductif : définit automatiquement un (des) principe d'induction.
- Permet la preuve par cas.
- Permet la preuve par induction (« généralisation » de la récurrence).

Au boulot...

Écrivez la fonction `size` qui retourne la taille d'une formule, càd son nombre de constructeurs.

Testez-là sur quelques formules.

Et si j'ai pas envie que ça termine ?

```
Axiom magic_order : forall A : Set, A -> A -> Prop. (* Matériel de TRICHE... *)
Axiom magic_proof : forall P : Prop, P.
Definition f_wforder (quote_a : Set) (x y : bool) : Prop := magic_order _ x y.

Section f.
  Variable quote_a : Set.
  Function f (x: bool) { wf (f_wforder quote_a) x} : quote_a := f (negb x).
  apply magic_proof. apply magic_proof. (* TRICHONS... *)
  Qed.
End f.
```

- En OCaml, **let rec** `f x` $= f$ (**not** `x`) : $\forall \alpha$, `bool` $\rightarrow \alpha$
- En Coq, $\forall \alpha$: Set, `bool` $\rightarrow \alpha$
  - On reviendra plus tard sur le polymorphisme en Coq.
  - Instiancions α par False >:-]

```
Coq < Check ((f False true)).
f False true
     : False

Goal 1 = 2.
  assert False. apply (f False true). contradiction.
Qed.
```

Lien démonstration / programmation : un autre

- En programmation :
  « *Est-ce qu'il n'existerait pas une fonction qui fait le travail ?* »
- *En démonstration :*
  « Est-ce qu'il n'existerait pas un théorème qui fait le travail ? »
- Commande **Search** (*motif*).
- Motif (de filtrage) : expression avec des _

```
Coq < Search (S _ = _ -> _ = _).
eq_add_S: forall n m : nat, S n = S m -> n = m
Coq < Search (0 = _ * 0).
mult_n_O: forall n : nat, 0 = n * 0
```

- Possibilité de motifs dépendants : ?a, ?b, ...

```
Coq < Search (?a + ?b = ?b + ?a).
Nat.add_comm: forall n m : nat, n + m = m + n
```

Utiliser des théorèmes

- On a déjà démontré une propriété : elle est devenue un théorème.
- On a trouvé, avec **Search**, un théorème.

- Un théorème est exactement comme une hypothèse.
- Peut l'utiliser avec **apply** :
  - pour faire avancer le but courant,
  - dans une hypothèse.
  - C.f. diapos ici, là, par ici, et par là.

Theorem `zero_neutral` : **forall** `n : nat`, $0 * n = 0$.
Proof. ... **Qed**.

```
1 goal
  x, y, z : nat
  ============================
  0 * (x + y * z) = 0
Unnamed_thm < apply zero_neutral.
No more goals.
```

Au boulot...

Énoncez et démontrez la propriété `size_never_null` disant que la taille d'une formule est toujours strictement supérieure à 0.

Souvenez-vous des tactiques que nous avons vues dernièrement.

Un peu d'automatisation

- Précédente preuve : pénible de répéter la même chose pour tous les buts.
- `T ; T'` : applique la tactique `T` sur le but courant puis `T'` sur tous les sous-buts générés.
- `[ T; [T1 | T2 | ... | Tn] ]` : applique la tactique `T` sur le but courant puis T_1 sur 1^{er} sous-but, T_2 sur 2^{nd} sous-but, etc.
- Tactique **auto** : résout les buts avec une combinaison de **intros** et **apply** d'hypothèses présentes dans le contexte.
- Ça ne coûte rien d'essayer, des fois que `:-)`

Au boulot...

Redémontrez la propriété `size_never_null` à coup d'automatisation.

Tactiques pour l'égalité

- Si le but est $t = t$: **reflexivity** .
- Utiliser une hypothèse $t_1 = t_2$ pour transformer le but : **rewrite**.
  - But : transformer toutes les occurrences de t_1 en t_2 ou l'inverse.
  - **rewrite** −> H : gauche ↦ droite. (par défaut si juste **rewrite** H).
  - **rewrite** <− H : droite ↦ gauche.

```
  n, m : nat
  H : n = m
  ============================
  n * m = n * n
Unnamed_thm < rewrite -> H.
1 goal
  n, m : nat
  H : n = m
  ============================
  m * m = m * m
```

```
  n, m : nat
  H : n = m
  ============================
  n * m = n * n
Unnamed_thm < rewrite <- H.
1 goal
  n, m : nat
  H : n = m
  ============================
  n * n = n * n
```

- Marche aussi avec un théorème statuant une égalité.

Quand **rewrite** est trop zélé (1/2)

- Parfois **rewrite** « *rewrite* » trop car partout.

```
1 goal
n1, n2 : nat
  H : forall n m p : nat, n <= m -> p + n <= p + m
  H0 : forall n : nat, n + 0 = n
  ============================
  n1 <= n1 + n2 (* Envie de réécrire le but en: n1 + 0 <= n1 + n2 *)
Unnamed_thm < rewrite <- (H0 n1). (* Spécifie par quoi remplacer n. *)
1 goal
  n1, n2 : nat
  H : forall n m p : nat, n <= m -> p + n <= p + m
  H0 : forall n : nat, n + 0 = n
  ============================
  n1 + 0 <= n1 + 0 + n2 (* Arg, on ne voulait pas réécrire le n1 de droite. *)
```

- Remarque : utilise une hypothèse en explicitant le ou les arguments.
  - Remplacer `n` de `H0` par le `n1` du but.
  - Autre choix possible (par défaut ici) : par le `n2` du but.

Quand **rewrite** est trop zélé (2/2)

- **replace** t_1 **with** t_2.
  - Remplace dans le but t_1 par t_2.
  - Génère le but en suspens $t_2 = t_1$.

```
1 goal
n1, n2 : nat
  H : forall n m p : nat, n <= m -> p + n <= p + m
  H0 : forall n : nat, n + 0 = n
  ============================
  n1 <= n1 + n2 (* Envie de réécrire le but en: n1 + 0 <= n1 + n2 *)

2 goals
Unnamed_thm < replace (n1 <= n1 + n2) with (n1 + 0 <= n1 + n2).
  n1, n2 : nat
  H : forall n m p : nat, n <= m -> p + n <= p + m
  H0 : forall n : nat, n + 0 = n
  ============================
  n1 + 0 <= n1 + n2
goal 2 is:
 (n1 + 0 <= n1 + n2) = (n1 <= n1 + n2)
```

Au boulot...

Pour pratiquer les tactiques présentées (`rewrite`, `replace`, `Search` et les plus anciennes), démontrez la formule suivante :

Goal forall `e e1 e2 : expr_t, e = EAnd e1 e2 -> size e1 < size e.`

Remarque : il sera peut-être utile de se souvenir que l'on a déjà démontré des théorèmes précédemment...

Preuves par induction structurelle

- Fonctions définies par récurrence $\Rightarrow$ souvent preuves par récurrence.
- Fonctions définies par induction $\Rightarrow$ souvent preuves par induction.

- Principe de récurrence :
  $\forall P,\ (P(0) \wedge \forall n \in \mathbb{N},\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n \in \mathbb{N},\ P(n)$
- Prouver $\forall n \in \mathbb{N},\ P(n)$ est vraie :
  - prouver que $P(0)$ est vraie
  - prouver que, si l'on suppose $P(n)$ vraie pour un n quelconque, alors on peut prouver que $P(n+1)$ est vraie.

- Principe d'induction : même chose pour
  - chaque cas de base ($\equiv P(0)$),
  - chaque cas d'induction ($P(n)$).

Principe d'induction : exemple généralisable

- Généré automatiquement par Coq pour chaque **Inductive** t.
- Nommé `t_ind`.

```
Coq < Inductive t := A | B | C (n : nat) | D (x : t) | E (x y : t).
...
t_ind is defined
...
Coq < Check t_ind.
t_ind : forall P : t -> Prop,
  P A ->
  P B ->
  ( forall n : nat, P (C n)) ->
  ( forall x : t, P x -> P (D x)) ->
  ( forall x : t, P x -> forall y : t, P y -> P (E x y)) ->
   forall t : t, P t
```

- Constructeurs paramétrés : autant d'hypothèses que de paramètres.

Tactique pour l'induction

- Tactique **induction** *H*.
- Peut être appliquée direct^{t} sur une variable :
  - fait automatiq^{t} un **intro** de la variable.
- S'applique à toute hypothèse inductive.
  - Y compris à des prédicats inductfs (c.f. plus tard).
- Coq propose automatiq^{t} les différents buts à prouver.

Preuve par induction : exemple

```
Inductive t := AE_True | AE_False | AE_And (e1 e2 : t).

Fixpoint is_true (e : t) :=
  match e with
  | AE_True => True | AE_False => False
  | AE_And e1 e2 => is_true e1 /\ is_true e2
  end.

Fixpoint has_false (e : t) :=
  match e with
  | AE_True => False | AE_False => True
  | AE_And e1 e2 => has_false e1 \/ has_false e2
  end.

Lemma true_if_no_AE_False : forall e : t, (is_true e) -> ~ (has_false e).
Proof.
  induction e. (* On laisse la tactique faire le intro toute seule. *)
  - (* Cas e = AE_True. But: is_true AE_True -> ~ has_false AE_True *)
    intros. simpl. intro. assumption.
  - intros. simpl in H. contradiction.
  - (* Cas e = AE_And. But: is_true (AE_And e1 e2) -> ~ has_false (AE_And e1 e2) *)
    intro H_true_and.
    simpl in H_true_and. (* Faire apparaitre le /\ *)
    destruct H_true_and. (* Couper le /\ en 2 en hypotheses. *)
```

Argument du cas inductif : prouver que ~ has_false (AE_And e1 e2)

```
IHe1 : is_true e1 -> ~ has_false e1
IHe2 : is_true e2 -> ~ has_false e2
H : is_true e1
H0 : is_true e2
```

On a donc ~ has_false e1 et ~ has_false e2

Comme le AE_And c'est un « et », CQFD par déf. de has_false.

```
    simpl.      (* Simplifier par la definition de has_false. *)
    intro.      (* On veut prouver des contradictions, i.e. False à partir de ~ (has_false e1 \/ has_false e2). *)
    destruct H1. (* Par cas sur has_false e1 \/ has_false e2 *)
    + (* Cas has_false e1. *) apply IHe1 in H. contradiction.
    + (* Cas has_false e2. *) apply IHe2 in H0. contradiction.
Qed.
```

Au boulot...

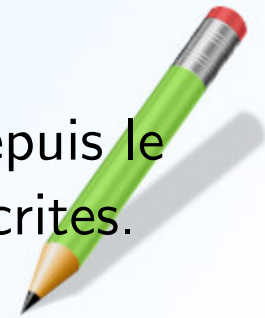
On se donne la fonction récursive `mirror_and_or` suivante qui permute récursivement les opérandes des « et » et des « ou » :

```
Fixpoint mirror_and_or (e : expr_t) :=
 match e with
 | ETrue | EFalse | EPred _ => e
 | ENot e' => ENot (mirror_and_or e')
 | EAnd e1 e2 => EAnd (mirror_and_or e2) (mirror_and_or e1)
 | EOr e1 e2 => EOr (mirror_and_or e2) (mirror_and_or e1)
 | EImpl e1 e2 => EImpl (mirror_and_or e1) (mirror_and_or e2)
 | EEquiv e1 e2 => EEquiv (mirror_and_or e1) (mirror_and_or e2)
end.
```

Démontrez les trois propriétés suivantes `double_mirror_is_id`, `mirror_keeps_size`, `double_mirror_keeps_size` :

1. **forall** `e : expr_t, mirror_and_or (mirror_and_or e)= e`
2. **forall** `e : expr_t, (size e) = size (mirror_and_or e)`
3. **forall** `e : expr_t, (size e) = size (mirror_and_or (mirror_and_or e))`

Attention : vous pouvez récupérer l'archive `09_coq_prog.zip` depuis le Moodle qui contient un squelette avec toutes les fonctions déjà écrites. Renommez le fichier `induction-eleves.v` en `induction.v`.



Lien démonstration / programmation : encore un autre

- En programmation :
  « *Ouch, c'est trop compliqué / long à faire dans la même fonction.* »
- ⇒ On écrit des fonctions intermédiaires.
- En démonstration :
  « *Ouch, c'est trop compliqué / long à faire dans la même preuve.* »
- ⇒ On écrit des lemmes intermédiaires.
- En programmation :
  « *Zut, je suis en train de réécrire le même code en changeant juste quelques constantes.* »
- ⇒ On écrit des fonctions intermédiaires.
- En démonstration :
  « *Zut, je suis en train de réécrire la même preuve en changeant juste quelques constantes.* »
- ⇒ On écrit des lemmes intermédiaires.

Le polymorphisme

- En OCaml : **type** 'a list = [ ] | :: **of** ('a * 'a list)
- En Coq, 'a est un paramètre explicite de type **Type**.

```
Inductive list_t (X : Type) : Type :=
  Nil : list_t X | Cons : X -> list_t X -> list_t X.
```

- list_t est une fonction : Type → Type.
- Besoin de spécifier le paramètre de type :

```
Coq < Check (Cons 1 (Cons 2 Nil)).
Error: The term "1" has type "nat" while it is expected to have
     type "Type".

Coq < Check (Cons nat 1 (Cons nat 2 (Nil nat))).
Cons nat 1 (Cons nat 2 (Nil nat))
     : list_t nat
```

Définitions polymorphes

- Le paramètre de type est comme tous les autres paramètres.
- Le spécifier partout, sauf dans les motifs où il faut des « underscore ».

```
Fixpoint map (X : Type) (f : X -> X) (l : list_t X) :=
  match l with
  | Nil _ => Nil X
  | Cons _ h q => Cons X (f h) (map X f q)
  end.
```

- Heureusement Coq fait de l'inférence de types !
- Il sait souvent deviner le paramètre.
- Mettre des _ à la place.

```
Coq < Check (Cons _ 1 (Cons _ 2 (Nil _))).
Cons nat 1 (Cons nat 2 (Nil nat))
     : list_t nat

Fixpoint map (X : Type) (f : X -> X) l :=
  match l with
  | Nil _ => Nil X
  | Cons _ h q => Cons _ (f h) (map _ f q)
  end.
```

Arguments implicites : s'alléger l'écriture

- Lourd de devoir mettre les paramètres de type partout :-(
- Lourd même si ce sont des « underscore ».

- Déclarer certains paramètres comme implicites.
- Plus besoin de les spécifier (et interdit même de les spécifier).

```
Arguments Nil {X}.
Arguments Cons {X} _ _.

Fixpoint map (X : Type) (f : X -> X) l :=
  match l with
  | Nil => Nil
  | Cons h q => Cons (f h) (map _ f q)    /* Arg. X de map pas encore implicite
      dans map. */
  end.

Arguments map {X} _ _.
Compute (map (fun x => x) (Cons 1 (Cons 2 Nil))).   /* Arg. X de map maintenant
    implicite. */
```

Arguments implicites directement dans les définitions

- Précédente définition de `map` : appel récursif avec argument de type `_`.
- Rendre directement implicite : déclaration entre { } à la place de ( ).

```
Inductive list_t {X : Type} : Type :=
  Nil : list_t | Cons : X -> list_t -> list_t.

Fixpoint map {X : Type} (f : X -> X) l :=
  match l with
  | Nil => Nil
  | Cons h q => Cons (f h) (map f q)
  end.

Compute (map (fun x => x) (Cons 1 (Cons 2 Nil))).
```

- Enfin une syntaxe vivable :-)
- Remarque : ∃ une possibilité pour les re-spécifier explicitement :
  - préfixer le nom du type par `@`.

Au boulot...

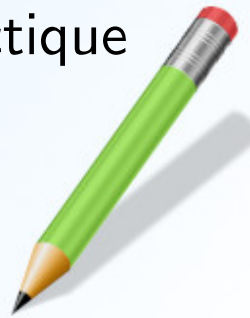
Récupérez l'archive `09_coq_prog.zip` sur le Moodle et renommez le fichier `mylists-eleves.v` en `mylists.v`.

Vous sont données différentes fonctions « banales » sur les listes, `length`, `map`, `mem`, `all1s`, `sum_list`.

Démontrez les deux lemmes suivants, sur « des programmes banals » :

1. **forall** `l : list_t, all1s l -> l = Nil \/ mem 1 l`
2. **forall** `(l : list_t), all1s l -> (sum_list l)= (length l)`
3. **forall** `(X : Type), forall (l : list_t), forall x : X,`
   `mem x l -> mem x (map (fun y => y) l)`

Indication : quand un but a la forme $f\ t_1 = f\ t_2$, on utilise la tactique `f_equal` pour le transformer en $t_1 = t_2$.



Au boulot...

Pour vous divertir chez vous, démontrez les propriétés suivantes qui utilisent `map` :

1. **forall** (`X Y` : **Type**), **forall** (`l` : `list_t`), **forall** (`f` : `X -> Y`),
   `length l = length (map f l)`
2. **forall** (`l` : `list_t`), `all1s l -> ~ mem 0 l`
3. **forall** (`l` : `list_t`), `all1s l -> l = Nil \/` **exists** `x` : `nat`, `x = 1`
4. **forall** (`l l'` : `list_t`),
   `all1s l -> l' = map (fun x => x + 1) l ->`
   `(l' = Nil \/ ~ (all1s l'))`

Remarque : plus assez de batterie pour rédiger la solution, à vous de chercher et de trouver ;-)

Retour dans le monde des programmes

- Correspondance de Curry-Howard : ça vous rappelle quelque chose ?
- Possible d'extraire du code OCaml à partir du code Coq.
- Tactique `Extraction` *ident*.

```
Coq < Require Extraction.
Coq < Extraction list_t.
type 'x list_t =
| Nil
| Cons of 'x * 'x list_t

Coq < Extraction sum_list.
let rec sum_list = function
| Nil -> O
| Cons (h, q) -> add h (sum_list q)
```

- Code OCaml garanti correct vis-à-vis du code Coq.
- Propriétés démontrées sur le code Coq tiennent sur le code OCaml.

Listes en Coq

- Listes polymorphes existent déjà en Coq :-)
- Importer par `Require Import List. Import ListNotations.`
- Constructeurs `nil` et `cons`.
- Syntaxe `[]` et `::` infixe.

```
Coq < Require Import List. Import ListNotations.
Coq < Check [true; false ; true ].
[true; false ; true]
     : list bool
Coq < Check List.map.
map
     : forall A B : Type, (A -> B) -> list A -> list B
Coq < Fixpoint stupid_id {X : Type} l : (list X) :=
        match l with [] => [] | h :: q => h :: (stupid_id q) end.
stupid_id is defined
Coq < Definition is_null {X : Type} (l : list X) :=
    <   match l with [] => True | _ => False end.
is_null is defined
Coq < Compute (is_null [1;2;3]).
     = False
     : Prop
```

Démontrer pour s'amuser

Au boulot...

Le fils de cet homme est le père de mon fils. Sachant que je ne suis pas une femme, quel est le lien de parenté entre cet homme et moi ?

Résolvez cette énigme.

Qu'avez-vous eu besoin de poser ?

Axiomatiser

- Envie de prouver des propriétés de sur des concepts de haut niveau.
- Pas toujours envie (réaliste) de tout formaliser depuis des types inductifs.
- Axiomatiser : décider d'axiomes a priori.
- Puis démontrer des propriétés sur la base de ces axiomes.
- Rappelez-vous : système à la Frege-Hilbert ;-)

- Difficile de poser un ensemble d'axiomes suffisants.
- Et surtout cohérents !
  - On ne veut surtout pas pouvoir démontrer $P \wedge \neg P$.

- Ça à l'air simple, mais pas du tout !

Au boulot...

Le fils de cet homme est le père de mon fils. Sachant que je ne suis pas une femme, quel est le lien de parenté entre cet homme et moi ?

On se donne les axiomes et les relations suivante :

```
Variable Homme : Set -> Prop.
Variable Femme : Set -> Prop.
Variable EstFils : Set -> Set -> Prop.
Variable EstPere : Set -> Set -> Prop.

(* Une homme n'est pas une femme et inversement. *)
Axiom HommeEqNotFemme : forall p : Set, Homme p <-> ~ Femme p.
(* Père est un homme. *)
Axiom PereHomme : forall p e : Set, EstPere p e -> Homme p.
(* Réciprocité des relations père/fils. *)
Axiom FilsPere :
  forall p e : Set, (EstFils e p /\ Homme p) <-> EstPere p e.
(* Père unique. *)
Axiom PereUnique :
  forall p1 p2 e : Set, EstPere p1 e /\ EstPere p2 e -> p1 = p2.
```

Démontrez que la solution que vous avez trouvée est la bonne.

Au boulot...

Pour vous amuser plus tard :-)

Plus y'a de gruyère plus y'a de trous. Plus y'a de trous moins y'a de gruyère.

Montrez que plus y'a de gruyère moins y'a de gruyère. Déduisez-en une contradiction.

Posez les **Variable** et **Axiom** nécessaires.

Au boulot...

Pour vous amuser plus tard :-)

1. *Alexandre est plus jeune que Catherine, mais plus vieux que Benjamin qui est plus vieux qu'Adrien.*
2. *Catherine est plus jeune que Thérèse, mais plus âgée qu'Adrien.*
3. *Adrien est plus jeune que François.*
4. *Thérèse est plus vieille que Catherine, mais plus jeune que François.*
5. *François est plus vieux que Alexandre.*

Laquelle de ces personnes est la plus âgée ?

Trouvez la solution de cette énigme et démontrez sa véracité en Coq.

Indication : Coq dispose d'une procédure de décision pour l'arithmétique entière linéaire nommée `lia`.
Elle nécessite simplement un

Require Import `Arith Lia`.

préalable.
Ça pourrait vous simplifier grandement la vie ;-)

Au boulot...

Plus dur, pour vous amuser plus tard :-)

1. *À gauche d'une reine, il y a le valet.*
2. *À la gauche d'un pique, il y a un carreau.*
3. *À la droite d'un cœur, il y a un roi.*
4. *À la droite d'un roi, il y a un pique.*

Reliez les symboles aux couleurs.

Appréciez le polymorphisme. On propose l'axiomatisation suivante :

```
Inductive position_t := Zero | Un | Deux.
Inductive carte_t := Valet | Reine | Roi.
Inductive dessin_t := Carreau | Coeur | Pique.
Variable Place : forall X : Type, X -> position_t.
Axiom MemePlaceEgal : forall X : Type, forall e1 e2 : X, Place X e1 = Place X e2 <-> e1 = e2.
Variable EstADroite : forall X Y : Type, X -> Y -> Prop.
Axiom EstADroiteSpec : forall X Y : Type, forall e1 : X, forall e2 : Y,
    EstADroite X Y e1 e2 <->
      ((Place X e1 = Deux /\ Place Y e2 = Un) \/ (Place X e1 = Deux /\ Place Y e2 = Zero) \/
       (Place X e1 = Un /\ Place Y e2 = Zero)).
Variable EstAGauche : forall X Y : Type, X -> Y -> Prop.
Axiom EstAGaucheSpec : forall X Y : Type, forall e1 : X, forall e2 : Y,
    EstAGauche X Y e1 e2 <->
      ((Place X e1 = Zero /\ Place Y e2 = Un) \/ (Place X e1 = Zero /\ Place Y e2 = Deux) \/
       (Place X e1 = Un /\ Place Y e2 = Deux)).
```

Démontrez que la place de chaque symbole est égale à la place de la couleur correspondante.

Un peu de formalisation de logique propositionnelle

Dans des exercices précédents...

- Définition du type `bool_t` (avec `F` et `T`).
- Définition du type des formules, `expr_t`.
- Définition de la sémantique des connecteurs
  - `sem_true`, `sem_not`, `sem_and`, etc.
- Formalisons quelques définitions et propriétés déjà démontrées « à la main »
- ... maintenant en Coq.

Au boulot...

Récupérez l'archive `11_coq_log.zip` depuis le Moodle et renommez le fichier `lprop-eleves.v` en `lprop.v`.

On se donne une définition (alias) représentant les valuations.

Definition `valuation_t := string -> bool_t.`

Écrivez la fonction `eval_expr v e` qui évalue une formule e dans une valuation v.

Remarque : oui, ça ressemble très fort à ce que nous avons écrit en OCaml en début de cours ;-)

Au boulot...

Définissez `is_a_model v p` qui exprime que la valuation v est un modèle de la formule p.

Définissez `is_a_tautology p` qui exprime que la formule p est une tautologie.

Au boulot...

Énoncez et démontrez le théorème statuant que $P \Rightarrow P$ est une tautologie quelle que soit la formule P.

Quel est le grand principe de cette preuve ?

Indication : ne vous privez pas d'utiliser `auto`, il y a suffisamment à faire pour le reste `;-)`.

Petit retour dans le passé

- Diapo « *Quelques schémas usuels macroscopiques de preuves* ».
- Preuve par l'absurde : pour démontrer « A », supposer « non A », démontrer que contradiction, donc en déduire que « A ».
  - **Attention** : valable seulement en logique classique.
  - **Différent de** : pour démontrer « non A », supposer « A », démontrer que contradiction, donc en déduire que « non A ».
- Revient à dire que $P_1 \wedge \neg P_2 \models \bot \Rightarrow P_1 \models P_2$
- Rappel : $\models$ est « *est une conséquence* ».
- Rappel : conséquence si $\forall \nu \ (\llbracket P_1 \rrbracket_\nu = \top) \Rightarrow (\llbracket P_2 \rrbracket_\nu = \top)$

Au boulot...

Définissez `has_consequence p1 p2` qui exprime que p_1 a pour conséquence p_2.

Énoncez et démontrez le théorème justifiant le principe de preuve par l'absurde présenté dans la diapo précédente.

Prédicats inductifs

Des Inductive dans Prop

- Vous vous souvenez des listes avec uniquement des 1 ?

```
Fixpoint all1s l :=
  match l with
  | [] => True
  | h :: q => h = 1 /\ all1s q
  end.
```

- Fonction (terme calculatoire) vérifiant si une liste contient uniquement des 1.
- Pourquoi pas définir le prédicat (terme logique) disant ce qu'est une liste contenant uniquement des 1 ?

```
Inductive plist1s : list nat -> Prop :=
| nill_1s : plist1s []
| cons_1s : forall x, forall l, x = 1 -> plist1s l -> plist1s (x :: l).
```

- `plist1s` : prédicat « est une liste de 1s »
  - La liste vide « est une liste de 1s »
  - Si x est un 1, si l « est une liste de 1s », alors $x :: l$ « est une liste de 1s ».

Utiliser les constructeurs comme « lemmes »

```
Inductive plist1s : list nat -> Prop :=
| nill_1s : plist1s []
| cons_1s : forall x, forall l, x = 1 -> plist1s l -> plist1s (x :: l).
```

- Chaque constructeur peut être vu comme un lemme.
- Si le but a la forme d'une « conclusion » de constructeur :
  - **apply** nom du constructeur,
  - nouveaux buts : « hypothèses » du constructeur.

```
Coq < Goal plist1s [1].
1 goal
  ============================
  plist1s [1]

Unnamed_thm < apply cons_1s.
2 goals
  ============================
  1 = 1
goal 2 is:
 plist1s []
```

```
Unnamed_thm < reflexivity .
1 goal
  ============================
  plist1s []

Unnamed_thm < apply nill_1s.
No more goals.
```

Utiliser des hypothèses « prédicat inductif » : par cas

- Pour preuve par cas : **destruct** *H*.
- Similaire aux autres définitions inductives vues précédemment.

```
forall l,
  plist1s l ->
    l = [] \/ (exists x', exists l', x' = 1 /\ l = x' :: l' /\ plist1s l').
Proof.
  intros l Hi.
  destruct Hi. (* Par cas sur la forme de l'hypothèse "être une liste de 1s" *)
  – (* Cas [] *) .....
  – (* Cas x : l avec x = 1 et plist1s l. *) ....
```

- Lemme dit d'inversion.
- Dit que si on « est une liste de 1s », alors
  - soit on est `[]`,
  - soit on est `x :: l` avec `x = 1` et `l` « est une liste de 1s ».
- À partir d'une hypothèse « est une liste de 1s » on peut déduire que l'on est uniquement dans l'un ou l'autre des cas.
- Heureusement, existe une tactique qui fait ça tout seul :-)
- Rendez-vous dans quelques diapos. . .

Au boulot...

Grand temps de commencer à prouver que la fonction récursive et le prédicat inductif représentent la même « chose ».

D'abord dans un sens...

Démontrez le lemme suivant :

Lemma `fun_is_pred` : **forall** `l`, `all1s l` −> `plist1s l`

Dit : « *si la fonction* `all1s` *dit que* `l` *ne contient que des 1s* », *alors* `l` « *est une liste de 1s* ».

Utiliser des hypothèses « prédicat inductif » : par induction

- Pour une preuve par induction : **induction** H.
- Similaire aux autres définitions inductives vues précédemment.

Utiliser des hypothèses « prédicat inductif » : par inversion

- Inversion d'hypothèse : **inversion** *H*.
- Revient à appliquer le lemme d'inversion automatiquement généré par Coq.

```
H : plist1s (x :: l)
======================
blabla
Unnamed_thm < inversion H.

H : plist1s (x :: l)
H2 : x = 1
H3 : plist1s l
======================
blabla
```

- À partir de *H*, déduit tout ce qui est forcément avéré.
- Nouvelles hypothèses :
  - forcément x est 1,
  - et forcément l « est une liste de 1s ».

Au boulot...

Récupérez l'archive `12_coq_dn.zip` sur le Moodle et renommez le fichier `listsagain-eleves.v` en `listsagain.v`.

Grand temps de **finir** de prouver que la fonction récursive et le prédicat inductif représentent les mêmes « choses ».

Maintenant dans l'autre sens...

Lemma `pred_is_fun` : **forall** `l`, `plist1s l -> all1s l`

Représenter les règles de la déduction naturelle

- Seule façon de construire un arbre de preuve : imbriquer les règles existantes.
- Logique propositionnelle intuitionniste : 15 règles possibles.
- Chaque noeud d'arbre : une de ces règles.
- Ressemble bien à une définition par cas : un **Inductive**.

- Idée : définir inductivement le prédicat :
  - « être une preuve en déduction naturelle ».
- Un constructeur par règle.
- Prémisses de la règle : « hypothèses du constructeur ».
- Conclusion de la règle : « conclusion du constructeur ».

Règles et prédicat

- `pr_in_nat_ded` : définit que dans un Γ, une formule p est la conclusion d'une règle de déduction naturelle.

```
Inductive pr_in_nat_ded : gamma_t -> expr_t -> Prop :=
```

$$\frac{\text{si } A \in \Gamma}{\Gamma \vdash A}\ Ax$$

```
| axiom : forall g a, In p g -> pr_in_nat_ded g a
```

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow intro$$

```
| imp_intro : forall g a b, pr_in_nat_ded (a :: g) b -> pr_in_nat_ded g (EImpl a b)
...
```

- On utilise les listes pour représenter Γ.
- Prédicat `In` : « appartient » des listes

```
Fixpoint In (a:A) (l:list A) : Prop :=
  match l with
    | [] => False
    | b :: m => b = a \/ In a m
  end.
```

- Constructeurs paramétrés par des propositions.
- `Inductive` dont le type « résultat » est proposition.

Au boulot...

Récupérez l'archive `12_coq_dn.zip` sur le Moodle et renommez le fichier `dn-eleves.v` en `dn.v`.

Ce squelette contient toutes les définitions relatives à la déduction naturelle des exercices en Coq précédents.

Il contient en plus le début de la définition du type `pr_in_nat_ded` (c.f. diapo précédente).

Complétez la définition de ce type pour toutes les règles de la déduction naturelle.

Remarque : le nom des constructeurs correspondant aux règles vous sont déjà donnés (ainsi, on aura tous les mêmes).

Au boulot...

Et on revient éternellement à la même formule que l'on traîne depuis le début ;-)

Démontrez une énième fois $(A \Rightarrow (B \Rightarrow C)) \Rightarrow (B \Rightarrow A) \Rightarrow B \Rightarrow C$ mais en utilisant la formalisation que nous avons faite en Coq.

Dans le squelette présent dans l'archive, la formule vous est déjà donnée pour vous épargner de construire l'arbre de syntaxe, fort fastidieux, à la main.

Indication : suivez l'arbre de preuve de la solution de l'exercice que nous avions abordé ici.

Indication : soyez fainéants et copiez-collez Γ depuis le but que Coq va vous indiquer :-)

Souvenez-vous de la preuve de correction

- On avait dit ici :
  - « Si une formule P est démontrable dans un contexte Γ alors P est une conséquence de Γ »
  - « Preuve par induction sur *le fait d'être une preuve de* P ».
- Vous avez défini ici « être un modèle » pour une valuation.
- Vous avez défini ici « être une conséquence » pour une formule.
- Il nous manque juste :
  - qu'une valuation est un modèle de Γ,

```
Definition is_a_model_of_gamma (v : valuation_t) (g : gamma_t) :=
  forall (p : expr_t), In p g -> is_a_model v p.
```

  - qu'un Γ a pour conséquence une formule.

```
Definition gamma_has_consequence (g : gamma_t) (p : expr_t) :=
  forall (v : valuation_t), is_a_model_of_gamma v g -> is_a_model v p.
```

- Au boulot...

```
Lemma nat_dec_correct : forall g : gamma_t, forall p : expr_t,
  pr_in_nat_ded g p -> gamma_has_consequence g p.
Proof.
  intros. induction H.
```

Au boulot (ensemble) : règle axiome

$$\frac{\text{si } A \in \Gamma}{\Gamma \vdash A}\ \textit{axiome}$$

Hypothèse **H0** : $A \in \Gamma$
Prouvons : pour toute ν, si $\models_\nu \Gamma$ alors $\models_\nu A$
  Hypothèse **H1** : $\models_\nu \Gamma$
  Prouvons : $\models_\nu A$
    Par H1, on déduit **H2** : $\forall P \in \Gamma, \models_\nu P$
    Appliquons H2, il reste à prouver $A \in \Gamma$
      CQFD par l'hypothèse H0

```
— unfold gamma_has_consequence.
  intros .
  (* H : In a g
     H0 : is_a_model_of_gamma v g
     ============================
     is_a_model v a *)
  unfold is_a_model_of_gamma in H0.
  (* H : In a g
     H0 : forall p : expr_t, In p g ->
          is_a_model v p
     ============================ *)
    apply H0.
    (* ============================
       In a g *)
    assumption.
```

- Pour la règle `top`, rien de méchant, que du trivial :-)
- Vous la ferez dans la foulée.

Au boulot (ensemble) : règle $\wedge$intro

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge intro$$

Par hypothèse d'induction, **H0** : $\Gamma \models A$
Par hypothèse d'induction, **H1** : $\Gamma \models B$
Prouvons : pour toute ν, si $\models_\nu \Gamma$ alors $\models_\nu A \wedge B$
  Hypothèse **H2** : $\models_\nu \Gamma$
  Prouvons : $\models_\nu A \wedge B$, càd $[\![A \wedge B]\!]_\nu = \top$
    Par H0, on déduit **H3** : $\forall \nu', \models_{\nu'} \Gamma \Rightarrow \models_{\nu'} A$
    Par H1, on déduit **H4** : $\forall \nu', \models_{\nu'} \Gamma \Rightarrow \models_{\nu'} B$
    Par H2 et H3, on a **H5** : $\models_\nu A$
    Par H2 et H4, on a **H6** : $\models_\nu B$
    Par H5 : $[\![A]\!]_\nu = \top$
    Par H6 : $[\![B]\!]_\nu = \top$
    $[\![A \wedge B]\!]_\nu = [\![A]\!]_\nu \wedge [\![B]\!]_\nu$
    Par H5 et H6, $[\![A]\!]_\nu \wedge [\![B]\!]_\nu = \top \wedge \top = \top$

```
— unfold gamma_has_consequence.
  intro v. intro.
 (* IHpr_in_nat_ded1 : g ||— a
    IHpr_in_nat_ded2 : g ||— b
    v : valuation_t
    H1 : ||— v g
    ======================
    is_a_model v (EAnd a b) *)
  assert (is_a_model v a). { (* Notre H5. *)
    unfold gamma_has_consequence in
           IHpr_in_nat_ded1.
    apply IHpr_in_nat_ded1. assumption. }
  assert (is_a_model v b). { (* Notre H6. *)
    unfold gamma_has_consequence in
           IHpr_in_nat_ded2.
    apply IHpr_in_nat_ded2. assumption. }
  unfold is_a_model. unfold is_a_model in H2, H3.
 (* H1 : is_a_model_of_gamma v g
    H2 : eval_expr v a = T
    H3 : eval_expr v b = T
    ======================
    eval_expr v (EAnd a b) = T *)
  (* Par cas sur les valeurs de a et de b. *)
  destruct (eval_expr v a) eqn: H_ev_v_a.
  + destruct (eval_expr v b) eqn: H_ev_v_b.
    * simpl. rewrite H_ev_v_a. rewrite H_ev_v_b. simpl
           . reflexivity .
    * simpl. rewrite H_ev_v_a. rewrite H_ev_v_b. simpl
           . assumption.
  + destruct (eval_expr v b) eqn: H_ev_v_b.
    * ... (* Le reste est en gros la même chose. *)
```

Au boulot...

À vous de jouer pour démontrer d'autres cas de règles.

Pas obligé de toutes les faire en séance.

Suggestion : gardez le $\Leftrightarrow$intro pour la fin ;-)

Suggestion : les $\bot$elim, $\neg$intro, $\wedge$elim, $\vee$elim sont plus simples :-)

Remarque : dans le squelette de preuve fourni, les directives **admit** permettent d'admettre temporairement un but pour passer au suivant. Ne permet pas de conclure définitivement une preuve : les buts suspendus empêcheront le **Qed** final.

Conclusion

Chemin suivi

- Fondements de la logique propositionnelle :
  - syntaxe et sémantique des formules.
- Introduction du raisonnement pour éviter le calcul :
  - déduction naturelle.
- Système de règles plus intuitif et plus automatisable :
  - calcul des séquents.
- Passage au calcul des prédicats : $\forall$, $\exists$.
- Découverte d'un outil et d'une logique de formalisation : Coq.
- Lien entre programmes et démonstrations.
- Démontration de propriétés sur des programmes.
- Démontration des propriétés sur « n'importe quoi », (énigmes).
- Formalisation de systèmes logiques avec des systèmes logiques.

- Logique comme sujet d'étude, logique comme outil.
- Démonstration, programmation.

Qu'est-ce que l'on n'a pas vu

- Des tonnes de trucs !
- Des procédures de démonstration automatique (partielles) :
  - résolution, SAT (DPLL, CDCL), tableaux...
- L'ajout de théories équationnelles : SAT modulo théories
- Des logiques différentes :
  Hoare (pour l'impératif), logiques temporelles (LTL, CTL, ...),
  logique linéaire...
- Le λ-calcul (de base) et des λ-calculs plus riches
- Les systèmes de type associés à ces λ-calculs.
- Les démonstrations de diverses propriétés sur les différents formalismes logiques et λ-calculs.
- L'induction mutuelle, non structurelle, les types coinductifs en Coq ...
- D'autres assistants de preuve et prouveurs automatiques.
- Et, et, et, et, et...

Je tiens à remercier (par ordre alphabétique) Catherine Dubois, Gilles Dowek, Thérèse Hardin et Mathieu Jaume : ils ont été de belles sources d'inspiration, que ce soit en tant que profs ou en tant que collègues. . .parfois les deux.

Fin

Index