Introduction à la théorie de la démonstration

Béranger Seguin

Cours aux Ernests
16 octobre 2020

1 De quoi parle-t-on ?

1.1 Introduction

On a l'habitude, en mathématiques, de manipuler des *objets* tels que des nombres, des figures géométriques, des ensembles, etc. Avec ces objets, on peut faire des *opérations* : on peut additionner les nombres, faire tourner les figures, prendre l'intersection des ensembles. De plus, on distingue dans chaque cas des *propriétés remarquables* que peuvent satisfaire ou non les objets : les nombres peuvent être entiers, les figures peuvent être des rectangles, les ensembles peuvent être finis, etc.

La *théorie de la démonstration* est une branche des mathématiques (plus spécifiquement de la logique) dont le point de départ consiste à considérer les *preuves mathématiques* (autrement dit les *démonstrations*) comme des objets mathématiques à part entière. Comme dans les exemples ci-dessus, on décrira alors des opérations qu'on peut effectuer sur les preuves, et des propriétés qu'elles peuvent ou non vérifier. On verra par exemple qu'on peut *normaliser* les démonstrations, ou encore que les démonstrations *intuitionnistes* sont une classe particulière importante parmi les démonstrations.

1.2 Les démonstrations « ordinaires »

Une démonstration a pour but d'établir la vérité d'une proposition mathématique. Pour cela, on se donne dès le départ certains énoncés qu'on suppose vrais « par définition » : ce sont les *axiomes*. Les axiomes permettent d'avoir un point de départ, sans lequel on ne peut à peu près rien montrer. À partir de ces axiomes, on se donne le droit d'appliquer quelques étapes de raisonnements spécifiques, qu'on appelle *règles de déduction*.

Voici un exemple de démonstration, au sens où la plupart des mathématicien·ne·s l'entendent :

Montrons qu'il y a une infinité de nombres premiers. Pour cela, on raisonne par l'absurde en supposant qu'il y a un nombre fini de nombres premiers qu'on note $p_1, p_2, \dots, p_r$*. On considère alors l'entier* $P := p_1 \times p_2 \times \dots p_r + 1$*, qui est supérieur à* 2 *puisque* 2 *est premier, et qui est donc divisible par un nombre premier, disons* p_i*. Mais alors* p_i *divise* $P - p_1 \times p_2 \times \dots p_r = 1$*, ce qui est absurde. Cela montre que notre hypothèse de départ est fausse.*

S'il s'agit d'une démonstration valide, et formalis**able** en théorie, les preuves sous cette forme ne sont pas très adaptées au traitement mathématique. Nous allons ébaucher une définition des démonstrations, qui permettra un traitement mathématique que ne permettent pas les preuves en langage naturel. Cette précision a un prix en complexité : on s'intéressera donc à des preuves beaucoup plus simples, mais il est important de garder en tête que les preuves au sens « habituel » peuvent, en principe, être formalisées, et que ce qu'on va affirmer s'appliquera tout à fait aux démonstrations telles qu'on les rédige habituellement.

1.3 Point sur le vocabulaire

Dans la suite, on définira la notion d'*énoncé* (au sens logique), qu'on considérera comme un synonyme de *proposition*. De même, les mots *preuve* et *démonstration* seront utilisés de manière interchangeable, sans volonté de marquer une nuance.

Il faudra faire attention à l'utilisation qu'on fera des mots *Vrai* et *Faux* : l'expression « A est faux » ne sera en principe qu'un raccourci pour dire « Non(A) est vrai », ou encore « supposer A mène à une

contradiction » (nous reverrons cela plus tard). Autant l'impossibilité pour un énoncé d'être à la fois vrai et faux, que l'impossibilité pour un énoncé de n'être ni vrai ni faux, sont des choses qu'on ne prend pas pour acquises mais dont on discutera le moment venu.

2 Les preuves comme objets mathématiques

Pour ne pas encombrer l'explication avec une formalisation lourde dont l'intérêt philosophique est maigre, on évitera d'introduire les multiples notations nécessaires pour une définition complète. Le principal objectif de ce cours est de donner une idée du sens de la notion de démonstration.

2.1 Énoncés

Avant de prouver quoi que ce soit, il faut être capable d'écrire des énoncés. Le but de cette sous-section est de définir la notion d'énoncé. Dans chaque partie de cette sous-section, on proposera d'abord une version courte (« vulgarisée ») de son contenu, ce qui devrait permettre de suivre la suite sans avoir d'une définition mathématique rigoureuse des objets en question.

2.1.1 Le langage

Version courte. Le *langage* est l'ensemble des symboles qu'on utilisera pour écrire des énoncés logiques. Ce sera donc notre « alphabet ». Par exemple, pour manipuler des entiers, on utilise les symboles « 0 », « + », « × », etc.

Version détaillée. Un langage est un ensemble de symboles répartis en trois types différents :

- Les symboles de *constantes* sont voués à désigner des « objets » spécifiques. Par exemple, dans le contexte de l'arithmétique, on pourrait avoir une constante « 0 » ; pour les ensembles, on pourrait avoir une constante « ∅ » (ensemble vide), etc.
- De façon analogue, les symboles de *fonctions* sont voués à désigner des fonctions (ou des applications). Par exemple, pour l'arithmétique, on aurait la fonction « Successeur » (d'arité 1 : elle prend un entier et en renvoie un autre) et les fonctions « + » et « × » (d'arité 2 : elles prennent deux entiers en entrée et en renvoient un).
- Les symboles de *relations* sont voués à désigner des propriétés précises, pouvant être vraies ou fausses. Par exemple, on prendra toujours le symbole égal = comme symbole de relation (d'arité 2).[1] Comme attendu, on note $x = y$ pour $=(x, y)$. Voici d'autres exemples : lorsqu'on parle de nombres, on pourrait définir les relations « être pair » (d'arité 1), « être plus petit que... » (d'arité 2), « être premiers entre eux » (d'arité 2), etc.

Par exemple, le langage de l'arithmétique est donné par la constante 0 et par les symboles de fonctions Successeur, + et ×.

D'autres symboles ont un rôle à part : les variables. On utilisera systématiquement $x_1, x_2, x_3, \ldots$ comme symboles pour les variables, ou x et y lorsqu'il y en a moins de deux.

Dans tous ces cas, il faut garder en tête que les symboles ne sont rien d'autre que des symboles ! Ce sera ensuite le rôle des axiomes de faire en sorte que ces symboles se comportent effectivement comme les éléments (constantes/fonctions/relations) qu'on cherche à leur faire représenter : en l'absence d'axiomes pour leur faire dépasser leur statut de symboles, leur comportement est indépendant du rôle et du sens qu'on leur prête par ailleurs. Rien n'assure a priori que ces symboles se comportent comme on le souhaite (ou tels que leur nom semble suggérer) tant qu'on ne les y a pas contraints par des axiomes.

Les termes. Le langage étant défini, on définit les *termes* via une définition inductive (la circularité de la définition n'est pas un problème : on peut définir des termes de « niveau » de plus en plus élevé) :

- Les symboles de constante et les variables sont des termes ;
- Si f est un symbole de fonction d'arité n, et que $T_1, T_2, \ldots, T_n$ sont des termes, alors $f(T_1, T_2, \ldots, T_n)$ est un terme.

1. Contrairement à tous les autres symboles, on forcera dans la suite systématiquement l'interprétation du symbole égal à coïncider avec l'égalité des objets correspondants.

2.1.2 Les énoncés logiques

Version courte. Les *énoncés* sont les « phrases » de notre langage. Ce sont eux qui ont la possibilité d'être vrais ou faux, et qu'on va donc essayer de démontrer ou de réfuter. Un énoncé est obtenu à partir des symboles du langage en les combinant de façon « raisonnable » : par exemple, quand on utilise le symbole « + », il faut qu'il y ait quelque chose à sa droite et à sa gauche. De plus, on peut combiner les énoncés à l'aide des connecteurs « et », « ou », « $\Rightarrow$ » (l'énoncé « $A \Rightarrow B$ » peut se lire « Si A alors B »), ainsi que des connecteurs plus compliqués tels que « Il existe une valeur de x satisfaisant $P(x)$ » et « Toute valeur de x satisfait $P(x)$ ».

Version détaillée. Les énoncés sont définis par étapes (c'est à nouveau une définition inductive) :

— **Les énoncés atomiques.** D'abord, on définit les *énoncés atomiques* : un énoncé atomique est de la forme « $P(y_1, y_2, \ldots, y_n)$ » où P est un symbole de relation d'arité n (cela peut être l'égalité) et où $y_1, \ldots, y_n$ sont des termes. Si une variable se trouve dans un énoncé atomique (à un niveau quelconque), on dit que cette variable est *libre* dans l'énoncé. Donnons des exemples d'énoncés atomiques, dans le langage de l'arithmétique :
  — Les énoncés suivants sont valides et ne possèdent pas de variables libres : « Successeur(0) + Successeur(0) = Successeur(0)×Successeur(0) », 0×0 = 0+0, (Successeur(0)+Successeur(0))× (Successeur(0) + 0) = 0.
    On notera que leur statut d'énoncé ne dit rien sur leur véracité.
  — Les « énoncés » suivants n'en sont en fait pas : « +Successeur(0) = Successeur(0) », 0 = ×, 0 + 0.
  — Les énoncés suivants sont valides et ont x comme variable libre : « x+0 = Successeur(0) × x », « $x = 0$ », « x = Successeur(x) ».

  Le faux. On introduit aussi un énoncé atomique particulier, $\perp$ (lu « Faux »). Cet énoncé correspond à quelque chose qu'on *sait* être faux, en d'autres mots une *contradiction*.

— **Conjonctions, disjonctions, implications.** Si A et B sont des énoncés [2], on définit les énoncés « A et B », « A ou B » et « $A \Rightarrow B$ » (qu'on notera également « Si A alors B » lorsque ce ne sera pas trop désagréable graphiquement).

— **Quantificateurs.** Si x est une variable libre d'un énoncé A, on définit les énoncés « Il existe une valeur de x satisfaisant A » et « Toute valeur de x satisfait A ». La variable x est alors *liée* dans l'énoncé obtenu, ce qui signifie qu'on ne peut plus l'utiliser comme variable par la suite (cela afin d'éviter les énoncés insensés tels que « Il existe un x tel que tout x vérifie $P(x)$. »). Par exemple, dans l'énoncé suivant, x est une variable libre, et y est une variable liée : « Il existe un entier y tel que x = Successeur(y) ».

En fait, les énoncés pour lesquels cela a du sens de se poser la question de leur vérité sont les énoncés *sans variable libre*, c'est-à-dire ceux dont toutes les variables sont quantifiées.

On utilisera aussi « Non(A) » et « A est faux » comme abréviations de « $A \Rightarrow \perp$ ». Cette définition, qui peut surprendre, a pour principal intérêt de limiter le nombre d'énoncés à traiter : les règles que nous donnerons pour l'implication entraîneront automatiquement des règles analogues sur la négation. Si on y réfléchit un peu, on constate que « $A \Rightarrow \perp$ » peut se lire « A ne peut pas être vrai sans qu'il y ait une contradiction. », ce qui correspond bien au sens de l'expression « A est faux. ».

2.2 Démonstrations

Dans cette section, on introduit un certain formalisme pour les démonstrations, qui s'appelle la *déduction naturelle*. D'autres choix sont possibles, mais celui-ci est assez simple à exposer.

Arbres de preuve. Dans ce formalisme, on note par une écriture de ce type :

$$\frac{A \qquad B}{C}$$

2. Il faut en fait que toute variable apparaissant à la fois dans A et dans B soit ou liée ou libre dans les deux énoncés à la fois.

le raisonnement qui permet à partir de deux énoncés A et B (dont on sait qu'ils sont vrais), de déduire que l'énoncé C est vrai. Plus généralement, un *arbre de preuve* pourra avoir une forme plus complexe. Par exemple, l'arbre

$$\dfrac{\dfrac{A \qquad B}{C} \qquad \dfrac{\dfrac{E}{F} \qquad G}{H}}{I} \qquad D$$

signifie : « De A et de B, je déduis C. De E, je déduis F ; puis de F et de G, je déduis H. À présent, de C, D et H, je déduis I. ».

2.2.1 Règles de déduction

Pour qu'une preuve soit *valide*, il faut que chaque étape de la déduction, dans l'arbre de preuve, soit *valide* ; c'est-à-dire, par définition, qu'elle corresponde à une des différentes règles de déduction qu'on s'apprête à lister.

Modus ponens. Voici une première règle de déduction, le *modus ponens*, noté (MP) :

$$\dfrac{A \qquad \text{Si } A \text{ alors } B}{B}\ (\text{MP})$$

Cette règle signifie que, chaque fois qu'on a sur la même ligne d'un arbre un énoncé de la forme « Si A alors B » (ou « $A \Rightarrow B$ ») en même temps que l'énoncé A correspondant, on peut en déduire B sur la ligne suivante. Par exemple, si on a montré « $(n \geq 3) \Rightarrow P(n)$ », et si on sait que $n \geq 3$, alors la preuve suivante est valide, en utilisant le modus ponens :

$$\dfrac{n \geq 3 \qquad (n \geq 3) \Rightarrow P(n)}{P(n)}\ (\text{MP})$$

Conjonctions. Voici une deuxième règle de déduction, la *règle d'introduction du « et »*, notée (Iet) :

$$\dfrac{A \qquad B}{A \text{ et } B}\ (\text{Iet})$$

Cette règle permet de faire apparaître le connecteur « et ». Symétriquement, deux règles permettent de le faire disparaître : ce sont les *règles d'élimination du « et »*, notées (Eet) :

$$\dfrac{A \text{ et } B}{A}\ (\text{Eet}) \qquad (\text{à gauche}) \qquad\qquad \dfrac{A \text{ et } B}{B}\ (\text{Eet}) \qquad (\text{à droite})$$

Disjonctions. De même, le connecteur « ou » dispose de deux règles d'introduction, notées (Iou) :

$$\dfrac{A}{A \text{ ou } B}\ (\text{Iou}) \qquad (\text{à gauche}) \qquad\qquad \dfrac{B}{A \text{ ou } B}\ (\text{Iou}) \qquad (\text{à droite})$$

La règle d'élimination du « ou » est un peu particulière : il s'agit de la *disjonction de cas*, notée (DC) :

$$\dfrac{A \text{ ou } B \qquad \text{Si } A \text{ alors } C \qquad \text{Si } B \text{ alors } C}{C}\ (\text{DC})$$

(Exercice : montrer que (MP) peut en fait se déduire de (DC) et de (Iou))

Implications. La règle d'introduction du connecteur « Si … alors … » est déroutante au premier abord. Montrer

$$\frac{\mathcal{A}}{\text{Si } B \text{ alors } C}$$

revient en fait à construire une preuve :

$$\frac{\mathcal{A} \qquad B}{\begin{array}{c}\vdots\\ C\end{array}}$$

En général, pour indiquer que B n'est qu'une hypothèse temporaire servant à démontrer « Si B alors C », on la note entre crochets. Ainsi, on écrirait plutôt :

$$\frac{\mathcal{A} \qquad [B]}{\begin{array}{c}\vdots\\ C\end{array}}$$

et on déduirait de cet arbre l'inférence suivante :

$$\frac{\mathcal{A}}{\text{Si } B \text{ alors } C}$$

Ex falso quodlibet. Voyons encore une règle, qui conditionne le fonctionnement de $\bot$. Il s'agit du *ex falso quodlibet*, noté (EFQ) : pour n'importe quel énoncé A, on a la règle :

$$\frac{\bot}{A}\ (\text{EFQ})$$

Autrement dit, de $\bot$, on peut déduire n'importe quel énoncé. Si $\bot$ est vrai, on parle d'*incohérence* : tout énoncé et son contraire sont simultanément démontrables !

Remarque (pour les curieuxses). Les règles de déduction peuvent être vues comme des sortes d'opérations sur les preuves, qui combinent les démonstrations de différents énoncés en une démonstration plus complexe. Si on note $\text{Dem}(A)$ l'ensemble des démonstrations d'un énoncé A, par exemple, on peut voir la règle d'introduction du « et » comme une fonction

$$\text{Dem}(A) \times \text{Dem}(B) \to \text{Dem}(A \text{ et } B)$$

De même, la règle de disjonction de cas peut être vue comme une fonction

$$\text{Dem}(A \text{ ou } B) \times \text{Dem}(A \Rightarrow C) \times \text{Dem}(B \Rightarrow C) \to \text{Dem}(C),$$

et ainsi de suite. Ces remarques sont, dans une certaine mesure, liées à la correspondance de Curry–Howard.

2.2.2 Exemples de démonstrations simples

Afin d'illustrer le fonctionnement des arbres de preuve, on démontre quelques propriétés.

Le principe de non-contradiction. Montrons que, pour tout énoncé A, l'énoncé suivant est vrai :

$$\text{Non}(A \text{ et } \text{Non}(A))$$

Autrement dit, un énoncé et son contraire ne peuvent pas être simultanément vrais. D'après ce qu'on a dit précédemment, « $\text{Non}(A \text{ et } \text{Non}(A))$ » est en fait un raccourci pour « $(A \text{ et } (A \Rightarrow \bot)) \Rightarrow \bot$ ». Pour démontrer un énoncé de la forme « Si … alors … il faut utiliser la règle d'introduction correspondante. Autrement dit, on doit montrer :

$$\dfrac{\dfrac{[A \text{ et } (A \Rightarrow \bot)]}{\vdots}}{\bot}$$

Une preuve est donnée par l'arbre suivant :

$$\dfrac{\dfrac{[A \text{ et } (A \Rightarrow \bot)]}{A}\,(\text{Eet}) \qquad \dfrac{[A \text{ et } (A \Rightarrow \bot)]}{A \Rightarrow \bot}\,(\text{Eet})}{\bot}\,(\text{MP})$$

Le « non-non-tiers exclu ». Démontrons, pour tout énoncé A, l'énoncé suivant :

$$\text{Non}(\text{Non}(A \text{ ou } \text{Non}(A)))$$

Là encore, il s'agit en fait d'un raccourci pour

$$((A \text{ ou } (A \Rightarrow \bot)) \Rightarrow \bot) \Rightarrow \bot$$

Pour montrer ce résultat, on doit construire un arbre valide de la forme

$$\dfrac{\dfrac{[(A \text{ ou } \text{Non}(A)) \Rightarrow \bot]}{\vdots}}{\bot}$$

(Remarquons, pour clarifier les choses, le fait général suivant, qu'on voit pour la deuxième fois : pour montrer un résultat de la forme $\text{Non}(A)$, on suppose A et on arrive à une contradiction. Ce n'est *pas* un raisonnement par l'absurde. Un coup d'œil sur les règles de déduction vues jusqu'ici permet de constater que c'est à ce stade l'unique moyen d'arriver à un tel énoncé.)

Montrons d'abord que $((A \text{ ou } \text{Non}(A)) \Rightarrow \bot) \Rightarrow \text{Non}(A)$. Pour cela, on considère l'arbre :

$$\dfrac{(A \text{ ou } \text{Non}(A)) \Rightarrow \bot \qquad \dfrac{[A]}{A \text{ ou } \text{Non}(A)}\,(\text{Iou})}{\bot}\,(\text{MP})$$

qui permet effectivement de déduire

$$\dfrac{(A \text{ ou } \text{Non}(A)) \Rightarrow \bot}{A \Rightarrow \bot}$$

ou, dit autrement :

$$\dfrac{\text{Non}(A \text{ ou } \text{Non}(A))}{\text{Non}(A)}$$

On peut alors construire l'arbre final de notre preuve :

$$\dfrac{[\text{Non}(A \text{ ou } \text{Non}(A))] \qquad \dfrac{\dfrac{[\text{Non}(A \text{ ou } \text{Non}(A))]}{\text{Non}(A)}\,(\text{vu à l'instant})}{A \text{ ou } \text{Non}(A)}\,(\text{Iou})}{\bot}\,(\text{MP})$$

On obtient donc bien finalement :

$$\dfrac{}{\text{Non}(\text{Non}(A \text{ ou } \text{Non}(A)))}$$

Remarque (très facultative). Dans cette preuve, à un moment, nous avons « dédoublé » certaines des hypothèses, par exemple $\text{Non}(A \text{ ou } \text{Non}(A))$. La possibilité d'effectuer ce dédoublement est un principe que nous n'avons pas explicitement formulé, même si elle est sous-entendue par la notation. Cependant, certains formalismes alternatifs (par exemple le calcul des séquents) demandent une règle additionnelle pour pouvoir dédoubler une hypothèse (de A on déduit A, A) ou même pour pouvoir éliminer une hypothèse inutile (de A, Γ on déduit Γ). Il est possible de se priver de la possibilité de dédoubler/éliminer des hypothèses ; cela conduit à des logiques très particulières telles que la « logique linéaire », où on ne peut pas gratuitement utiliser plusieurs fois (ou ne pas utiliser) les hypothèses.

L'introduction de la double négation. Montrons l'énoncé $A \Rightarrow \text{Non}(\text{Non}(A))$ ou, de manière équivalente, donnons une preuve de

$$\dfrac{\begin{array}{c}A\\ \vdots\end{array}}{\text{Non}(\text{Non}(A))}$$

Cela revient à montrer

$$\dfrac{A \qquad \text{Non}(A)}{\begin{array}{c}\vdots\\ \bot\end{array}}$$

Cela découle du modus ponens, comme dans l'exemple du principe de non-contradiction vu plus haut :

$$\dfrac{A \qquad A \Rightarrow \bot}{\bot}\ (\text{MP})$$

Dans ces trois exemples de démonstrations, on constate l'intérêt de considérer $\text{Non}(A)$ comme une simple abréviation de $A \Rightarrow \bot$: cela permet d'utiliser les règles d'introduction de l'implication, la disjonction de cas et le modus ponens sans avoir à introduire des règles spécifiques à la négation.

2.3 Le cas des quantificateurs

Pour simplifier l'exposition, on omet le cas des quantificateurs « Il existe » et « Pour tout ». Disons simplement que « Pour tout » se comporte comme un « et » généralisé, et que « Il existe » se comporte comme « ou », avec des règles d'introduction et d'élimination analogues. Par exemple, si on est capable de démontrer un énoncé P avec une varible libre x (donc, indépendemment de x), alors on en déduit une preuve de « Pour tout x, il est vrai que $P(x)$ » : cette règle est analogue à (Iet). Deviner les autres règles est un exercice intéressant.

2.4 Le tiers exclu

Dans les démonstrations qu'on a faites jusque ici, on a fait de la logique *intuitionniste*. Pour retrouver la logique habituelle, il faut rajouter une dernière règle de déduction :

$$\dfrac{}{A \text{ ou } \text{Non}(A)}\ (\text{LEM})$$

Cette règle permet, sans aucune hypothèse, d'affirmer que tout énoncé est soit vrai soit faux. En quelque sorte, on « force » la logique à faire un choix, à se prononcer sur la véracité de tout énoncé.

La différence principale entre la logique classique et la logique intuitionniste est la suivante : en logique intuitionniste, une preuve de « A ou B » peut toujours être transformée en une preuve de A ou bien une preuve de B (par un processus, parfois coûteux, qui s'appelle normalisation, qu'on verra plus tard). Cela est lié au fait que la seule manière d'introduire un « ou » est la règle d'introduction du « ou ». On parle parfois de mathématiques *constructives*, car les objets dont on montre l'existence sont toujours, au fond, construits explicitement (même s'il peut être compliqué en pratique de récupérer le témoin explicite à partir des preuves d'existence). Avec le tiers exclu, tout change : d'une preuve de « A ou B », on ne peut plus forcément extraire une preuve de A ou une preuve de B. Cela se voit directement en prenant $B = \text{Non}(A)$ pour un énoncé A quelconque, puisque (LEM) permet de démontrer « A ou $\text{Non}(A)$ » sans rien savoir de A, qui peut très bien n'être ni démontrable ni réfutable ! Quand on se permet d'utiliser cette règle, on dit qu'on fait de la logique *classique*.

Théorème. *Le tiers exclu (LEM) est équivalent à la règle suivante, dite* règle d'élimination de la double négation *:*

$$\dfrac{\text{Non}(\text{Non}(A))}{A}\ \textit{(EDN)}$$

(EDN) dit que « ce qui n'est pas faux est vrai ».

Démonstration. Montrons les deux sens indépendamment :

— Supposons d'abord qu'on s'autorise le tiers exclu. Soit A un énoncé quelconque. On cherche une preuve de la forme :

$$\dfrac{\dfrac{[\mathrm{Non}(\mathrm{Non}(A))]}{\vdots}}{A}$$

On va mettre à profit la règle de disjonction de cas. Montrons que, sous l'hypothèse $\mathrm{Non}(\mathrm{Non}(A))$, c'est-à-dire $\mathrm{Non}(A) \Rightarrow \bot$, on a $\mathrm{Non}(A) \Rightarrow A$. En effet :

$$\dfrac{\dfrac{\mathrm{Non}(A) \Rightarrow \bot \qquad \mathrm{Non}(A)}{\bot}\,(\mathrm{MP})}{A}\,(\mathrm{EFQ})$$

De plus, on a bien $A \Rightarrow A$. Ainsi :

$$\dfrac{\dfrac{}{A \text{ ou } \mathrm{Non}(A)}\,(\mathrm{LEM}) \quad \dfrac{}{A \Rightarrow A} \quad \dfrac{[\mathrm{Non}(\mathrm{Non}(A))]}{\mathrm{Non}(A) \Rightarrow A}}{A}\,(\mathrm{DC})$$

qui est bien une preuve de la forme souhaitée.

— Cette fois, on s'autorise la règle d'élimination de la double négation. Soit un énoncé A quelconque. On cherche à montrer « A ou $\mathrm{Non}(A)$ » à partir de rien. Dans un paragraphe précédent, on a montré (sans utiliser le tiers exclu), le résultat suivant :

$$\dfrac{\dfrac{\begin{array}{c}\text{—}\\ \vdots\end{array}}{\mathrm{Non}(\mathrm{Non}(A \text{ ou } \mathrm{Non}(A)))}}{}$$

Ne reste qu'à compléter...

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\text{—}\\ \vdots\end{array}}{\mathrm{Non}(\mathrm{Non}(A \text{ ou } \mathrm{Non}(A)))}}{A \text{ ou } \mathrm{Non}(A)}\,(\mathrm{EDN})}{}$$

□

La logique classique permet le raisonnement suivant : pour prouver un énoncé A, on suppose $\mathrm{Non}(A)$ et on arrive à une contradiction. A priori, on a alors montré $\mathrm{Non}(\mathrm{Non}(A))$. Cependant, puisqu'on dispose en plus de la règle d'élimination de la double négation, on en déduit une preuve (classique) de A. C'est ce qu'on appelle le *raisonnement par l'absurde*.

Remarque. Comme on l'a montré dans le paragraphe 2.2.2, on a toujours $\mathrm{Non}(\mathrm{Non}(A \text{ ou } \mathrm{Non}(A)))$, c'est-à-dire que le tiers exclu n'est pas faux. Pour cette raison, si un énoncé P admet une preuve classique, alors $\mathrm{Non}(\mathrm{Non}(P))$ admet une preuve intuitionniste.

3 Un point sur les liens syntaxe/sémantique

3.1 Syntaxe et sémantique

Dans ce qui précède, on a parlé des preuves comme objets purement syntaxiques (il s'agit de propositions logiques qu'on manipule en tant que telles — pour le dire simplement, c'est « juste du texte ») ayant pour but d'établir la *vérité* des énoncés correspondants. La notion de vérité possède de nombreux sens en mathématiques, mais on va évoquer l'un d'entre eux et voir à comment cette notion *sémantique* de vérité interagit avec la notion syntaxique de preuve.

Précisons rapidement ce qu'on entend par la dichotomie syntaxique/sémantique. Là où une preuve d'un énoncé à partir d'axiomes ne fait que manipuler des énoncés logiques (donc de la syntaxe) à l'aide de règles de déduction, on a le sentiment qu'il existe « en dehors des preuves » des vérités mathématiques concernant les objets qu'on étudie : qu'on puisse construire un arbre de preuve menant à l'énoncé « $2+2 = 4$ » est intéressant, mais il est encore plus intéressant que le résultat de l'addition de deux et de deux soit

effectivement quatre. C'est ce qu'on entend quand on parle de *sémantique*. Si la nuance peut sembler trop vague pour être étudiée mathématiquement, on peut en fait démontrer un certain nombre de résultats dont on peut considérer qu'ils nous apprennent quelque chose sur les liens entre syntaxe et sémantique.

3.2 Résultats en théorie des modèles

Imaginez qu'on s'intéresse à un énoncé P, par exemple en essayant de le démontrer à partir d'un ensemble d'axiomes $\mathcal{A}$. Une branche de la logique particulière, la *théorie des modèles*, permet de considérer l'intégralité des « mondes mathématiques possibles » M dans lesquels les axiomes de $\mathcal{A}$ sont vérifiés [3] : on dit que M est un *modèle* de $\mathcal{A}$. On dira alors que l'énoncé P est *vrai* (modulo $\mathcal{A}$) lorsque P est vrai dans *chacun* des modèles de $\mathcal{A}$. Il est important de noter que « P est fausse » signifie que P est fausse dans *tous* les modèles de $\mathcal{A}$, ce qui est beaucoup plus fort que de dire que P n'est pas vraie. Ainsi, de nombreuses propositions ne sont si vraies ni fausses : on les appelle alors *indécidables*.

Un premier résultat rassurant est le suivant :

Théorème (théorème de correction)**.** *S'il existe une preuve de P à partir des axiomes de $\mathcal{A}$, alors P est vraie modulo $\mathcal{A}$.*

Démonstration. L'idée est assez simple : soit un modèle M de $\mathcal{A}$. Puisqu'on dispose d'une preuve (syntaxique) de P à partir des axiomes de $\mathcal{A}$, et que M vérifie ces axiomes, il « suffit » de lire la preuve étape par étape et de recopier le même raisonnement en l'appliquant directement dans M. □

Le résultat précédent signifie que tout ce que l'on est capable de démontrer est vrai. Un résultat autrement plus fort est le suivant :

Théorème (théorème de complétude en logique du premier ordre)**.** *Si P est vraie modulo $\mathcal{A}$, alors il existe une preuve (classique) de P à partir des axiomes de $\mathcal{A}$.*

Cela signifie que si un énoncé est une conséquence inévitable des axiomes, alors il est possible de le démontrer « syntaxiquement » en partant de ces même axiomes, ce qui est remarquable ! Par exemple, les énoncés indécidables sont exactement ceux qu'on ne peut pas prouver, mais dont on ne peut pas non plus prouver la négation. Ce théorème de complétude est puissant puisqu'il établit la correspondance (dans le formalisme particulier qu'on nomme « logique classique du premier ordre ») entre sémantique et syntaxe. Il a des conséquences simples et tout aussi remarquables :

Corollaire. *Si un ensemble d'axiomes $\mathcal{A}$ est cohérent, alors il admet un modèle.*

Démonstration. Si $\mathcal{A}$ n'admettait pas de modèle, alors $\bot$ serait vrai modulo $\mathcal{A}$ (de manière vide), donc on aurait une preuve de $\bot$ à partir de $\mathcal{A}$ d'après le théorème de complétude, ce qui contredit la cohérence de $\mathcal{A}$. La réciproque est également vraie (en utilisant le théorème de correction). □

Corollaire (théorème de compacité)**.** *Soit un ensemble infini d'axiomes $\mathcal{A}$. Pour montrer qu'il existe un modèle de $\mathcal{A}$, il suffit de montrer que tout choix d'un nombre fini d'axiomes de $\mathcal{A}$ admet un modèle.*

Démonstration. L'ensemble d'axiomes $\mathcal{A}$ n'admet pas de modèle si et seulement s'il existe une preuve de $\bot$ à partir des axiomes de $\mathcal{A}$. Une telle preuve étant nécessairement de taille finie, elle ne peut utiliser qu'un nombre fini d'axiomes de $\mathcal{A}$, et on obtiendrait alors une partie finie incohérente de $\mathcal{A}$. □

4 Liens avec l'informatique : premier volet

(**Point technique :** dans la suite, on se place dans la situation où le langage et les axiomes sont *récursivement énumérables*, c'est-à-dire qu'un ordinateur peut en faire la liste en un temps potentiellement infini. Cette restriction étant relativement technique, nous ne l'expliquerons pas.)

3. On omet les détails, mais pour que cette phrase ait un sens, il faut bien sûr qu'on ait associé aux éléments du langage des « incarnations » dans M.

4.1 Machines de Turing et décidabilité

Durant la première moitié du vingtième siècle, le mathématicien Alan Turing a inventé un modèle de calcul reposant sur une machine hypothétique, la machine de Turing. Nous nous passerons de la définition : il suffit de savoir que les programmes informatiques au sens où nous l'entendons habituellement sont capables d'effectuer exactement les mêmes calculs que la machine de Turing. [4]

Une première remarque intéressante est la suivante : il est possible d'écrire un programme qui explore l'ensemble des preuves. Par exemple, s'il y a un nombre fini d'axiomes et un langage fini, le programme pourrait d'abord lister tous les axiomes, puis toutes les preuves qu'on peut faire en une étape à partir des axiomes, puis toutes les preuves qu'on peut faire en deux étapes à partir des axiomes, et ainsi de suite. Il se trouve qu'il est possible de concevoir un programme analogue dans le cas plus général où nous sommes (cela est lié à l'hypothèse faite dans le point technique ci-dessus).

En particulier, si on donne un énoncé P à ce programme, trois situations sont possibles :

— À un certain moment, le programme trouve une preuve de P. On lui demande alors de s'arrêter.

— À un certain moment, le programme trouve une preuve de $\mathrm{Non}(P)$. On lui demande alors de s'arrêter.

— Le programme ne trouve jamais de preuve ni de P ni de $\mathrm{Non}(P)$.

Dans le troisième cas, on ne peut jamais décider, à un instant donné, si P est vrai (mais sa preuve se trouve plus loin dans la liste des preuves...), si P est faux (mais la preuve de $\mathrm{Non}(P)$ se trouve plus loin...), ou bien si P est en fait indécidable, auquel cas le programme tournera éternellement, sans s'arrêter.

Cela est lié à un problème historique de l'informatique : le problème de l'arrêt. La question est de savoir s'il existe un programme informatique qui, quand on lui fournit en entrée un programme informatique, est capable de dire si ce programme tourne indéfiniment ou bien s'il finit par s'arrêter.

Supposons à présent que tout énoncé est décidable (autrement dit, il existe toujours une preuve soit de P, soit de $\mathrm{Non}(P)$). En formalisant la notion de programme, on constate qu'un programme informatique est un objet mathématique comme un autre (techniquement, on le représente en fait comme un entier) et on montre qu'il est possible d'écrire mathématiquement la proposition « Le programme s'arrête ». Par hypothèse, cet énoncé est décidable : il existe toujours soit une preuve de l'arrêt d'un programme donné, ou une preuve du fait qu'il ne s'arrête jamais, et en énumérant les preuves comme dans le programme décrit ci-dessus (qui dans ce cas termine toujours), on obtient un algorithme qui répond au problème de l'arrêt. Cela mène à la conclusion suivante :

Théorème. *Si tout énoncé est décidable, alors il existe une solution au problème de l'arrêt.*

(**Point technique 2 :** En fait, pour que cela fonctionne, il a fallu que notre langage et nos axiomes soient assez puissants pour pouvoir manipuler des entiers en les additionnant et en les multipliant, de manière à pouvoir raisonner sur les programmes. Par la suite, on se place dans cette situation.)

4.2 Le premier théorème d'incomplétude de Gödel

En 1931, le mathématicien Kurt Gödel a montré le théorème suivant :

Théorème (premier théorème d'incomplétude de Gödel)**.** *Si le langage et les axiomes sont récursivement énumérables et suffisamment puissants pour pouvoir manipuler des entiers, alors il existe des énoncés indécidables (on dit que la théorie est incomplète). En particulier, les axiomes de Peano ou la théorie des ensembles*[5] *sont des théories incomplètes.*

À son époque, ce résultat a coupé court aux ambitions de celles et ceux qui cultivaient l'espoir de résoudre toutes les questions mathématiques. Gödel a montré que, aussi forts que soient les axiomes que l'on choisirait, il resterait toujours des énoncés hors de portée de la démonstration. Vu ce qu'on a dit précédemment, il suffit pour démontrer le théorème de Gödel de démontrer le fait suivant :

Théorème. *Il n'existe pas de solution au problème de l'arrêt.*

4. Précisons que le mot « capable » utilisé ici ne fait absolument pas référence à de la *puissance de calcul* au sens de la performance : ici, la possibilité de calculer quelque chose est envisagée sous un angle parfaitement théorique, indépendant des contraintes matérielles et technologiques (notamment, sur l'utilisation de mémoire), et en tolérant des temps de calculs arbitrairement longs (mais finis !).

5. Théories déjà rencontrées dans des cours d'introduction précédents des *Cours aux Ernests*.

Démonstration. Tout au long de cette preuve, appelons *métaprogrammes* les programmes qui prennent en entrée le code source d'un programme.

Raisonnons par l'absurde : supposons qu'on ait un métaprogramme `HALT` qui prenne en entrée le code d'un métaprogramme quelconque, et qui renvoie `Vrai` si le programme termine lorsqu'on lui fournit son propre code en entrée, et `Faux` dans le cas contraire. Si le problème de l'arrêt a une solution, alors un tel métaprogramme `HALT` existe.

Définissons alors le métaprogramme `DIAG`, qui prend en entrée le code source d'un métaprogramme P, qui utilise `HALT` pour déterminer si P termine lorsqu'on lui fournit son propre code en entrée (c'est à dire que $P(P)$ termine), puis qui, une fois la réponse obtenue, adopte le comportement suivant :

— Si P termine lorsqu'on lui fournit son propre code en entrée, `DIAG` entre dans une boucle infinie.

— Sinon, `DIAG` s'arrête.

On essaie alors de calculer `HALT(DIAG)`. Par définition de `HALT`, le résultat de `HALT(DIAG)` est `Vrai` si et seulement `DIAG(DIAG)` termine, c'est-à-dire (par définition de `DIAG`) si et seulement si le résultat de `HALT(DIAG)` est `Faux`. Ainsi, `HALT(DIAG)` ne peut valoir ni `Vrai` ni `Faux`, ce qui est impossible. Le programme `HALT` ne peut donc pas exister. □

Plus tard, Gödel a montré un résultat encore plus surprenant, qui donnait non seulement un exemple concret de résultat indécidable, mais enlevait de plus aux mathématiciennes et mathématiciens l'espoir de montrer que la théorie des ensembles est cohérente :

Théorème (second théorème d'incomplétude de Gödel)**.** *Si le langage et les axiomes sont récursivement énumérables et suffisamment puissants pour pouvoir manipuler des entiers, et si les axiomes sont cohérents, alors les axiomes ne peuvent pas démontrer leur propre cohérence (l'énoncé correspondant à la cohérence est alors indécidable).*

En particulier, il est impossible de montrer dans ZFC (le système d'axiomes de la théorie des ensembles généralement utilisé en mathématiques) que ZFC est cohérente. Nous ne serons donc jamais à l'abri d'un jour découvrir une contradiction dans la théorie des ensembles !

La preuve de ce théorème est plus compliquée, et la formalisation nécessaire pour exprimer « la théorie est cohérente » sous la forme d'un énoncé arithmétique est assez subtile.

5 Lambda-calcul et correspondance de Curry-Howard

Au cours du vingtième siècle, les mathématiciens et mathématiciennes ont pris conscience du fait qu'il existait une correspondance entre les programmes informatiques et les démonstrations mathématiques. Pour l'illustrer, nous allons rapidement décrire un langage de programmation différent du formalisme des machines de Turing (mais, ultimement, équivalent) : le *lambda-calcul typé*. Pour simplifier l'exposition, nous ne nous préoccuperons pas des énoncés contenant des quantificateurs « Pour tout » et « Il existe ».

5.1 Le lambda-calcul pur

Le *lambda-calcul* est une sorte de langage de programmation très simplifié, ce qui le rend adapté au traitement mathématique. Ce langage de programmation est exactement aussi puissant (en termes de calculabilité) que les machines de Turing, ce qui signifie qu'on peut légitimement considérer les programmes dans ce langage comme des « programmes informatiques ».

Le lambda-calcul est un langage d'un type très particulier : c'est un langage *fonctionnel*, ce qui signifie qu'en lambda-calcul, tous les objets sont des sortes de fonctions, nommées *termes*. Ainsi, si x et y sont des termes, on peut définir le terme xy (« x évalué en y »). De plus, si $A(x)$ est un terme possédant une variable libre x (par exemple, xx), on peut définir le terme suivant :

$$\lambda x.A(x)$$

qu'on interprète essentiellement comme signifiant « la fonction qui à x associe $A(x)$ ». On a alors la règle de calcul suivante, nommée « réduction » :

$$\big(\lambda x.A(x)\big)\, y \Rightarrow A(y)$$

Cette règle contient l'essentiel de la puissance calculatoire du lambda-calcul : elle correspond à une étape de l'exécution d'un programme informatique.

5.2 Le lambda calcul simplement typé

On introduit désormais une variante du lambda-calcul qui possède un système de *types*. Pour cela, on demande à tous les termes d'avoir un *type*. On indiquera que le terme x est du type α en écrivant « $x\colon \alpha$ ». Les types contraignent ce qu'on a le droit d'écrire : en plus des règles de calcul, on a des règles de grammaire et de *typage*, qui disent quels sont les termes « licites », et qui permettent de déterminer leur type.

Par exemple, un terme f du type $\alpha \to \beta$ ne peut-être évalué qu'en un terme x de type α, c'est-à-dire qu'on n'a le droit d'écrire fx qu'à condition que $x\colon \alpha$, et alors on a $fx\colon \beta$. Autrement dit, moralement, $\alpha \to \beta$ est le type des « fonctions » qui prennent en entrée un terme de type α et renvoient un terme de type β. Lorsque $A(x)\colon \beta$ est une expression dépendant d'un terme $x\colon \alpha$, on donne alors au terme $\lambda(x\colon \alpha)\,.\,A(x)$ le type $\alpha \to \beta$. En particulier, pour tout type α, il existe un terme spécial de type $\alpha \to \alpha$:

$$\texttt{id}_\alpha = \lambda(x\colon \alpha)\,.\,(x\colon \alpha).$$

Types produits. Étant donné deux types α et β, il existe un type $\alpha \times \beta$, dont les membres correspondent aux couples (x, y) ·où $x\colon \alpha$ et $y\colon \beta$. Plus formellement, on a des règles de réécriture (et de typage) qui conditionnent le fonctionnement de ce type (les symboles `fst`, `snd`, `tupl` sont ajoutés au langage du lambda-calcul) :

$$\begin{aligned}
\texttt{tupl}\ (x\colon \alpha)\,(y\colon \beta) &\Rightarrow (\texttt{tupl}\ x\,y\colon \alpha \times \beta) && \text{correspond au couple } (x, y)\\
\texttt{fst}\ (\texttt{tupl}\ x\,y\colon \alpha \times \beta) &\Rightarrow (x\colon \alpha) && \text{récupère la première coordonnée}\\
\texttt{snd}\ (\texttt{tupl}\ x\,y\colon \alpha \times \beta) &\Rightarrow (y\colon \beta) && \text{récupère la deuxième coordonnée}
\end{aligned}$$

Concrètement, cela signifie que chaque fois qu'on voit écrit $\texttt{fst}\,(\texttt{tupl}\,x\,y)$, par exemple, on peut le remplacer (en une étape de calcul) par x.

Types sommes. De même, pour tous types α et β, il y a un type $\alpha + \beta$ dont les membres sont soit de type α, soit de type β. Ces types sont implémentés formellement par l'ajout dans le langage de symboles `inl`, `inr` (injections à gauche et à droite, respectivement) et `case`, régis par les règles de réécriture et de typage suivantes :

$$\begin{aligned}
\texttt{inl}\,(x\colon \alpha) &\Rightarrow (\texttt{inl}\ x\colon \alpha + \beta)\\
\texttt{inr}\,(y\colon \beta) &\Rightarrow (\texttt{inr}\ y\colon \alpha + \beta)
\end{aligned}$$

ainsi qu'à deux autres règles de calcul, qui implémentent la disjonction de cas. Le terme $\texttt{case}\ f\,g\,x$, où $x : \alpha + \beta$, $f : \alpha \to \gamma$ et $g : \beta \to \gamma$, signifie essentiellement « si x est de type α, renvoie fx, si x est de type β, renvoie gx ». Voici les règles de réécriture/typage correspondantes :

$$\begin{aligned}
\texttt{case}\,(f\colon \alpha \to \gamma)\,(g\colon \beta \to \gamma)\,(\texttt{inl}\ x\colon \alpha + \beta) &\Rightarrow (fx\colon \gamma)\\
\texttt{case}\,(f\colon \alpha \to \gamma)\,(g\colon \beta \to \gamma)\,(\texttt{inr}\ y\colon \alpha + \beta) &\Rightarrow (gy\colon \gamma)
\end{aligned}$$

Comparaison avec le lambda-calcul pur. En réalité, l'ajout du système de types (et des contraintes associées) diminue la puissance calculatoire du lambda-calcul : le lambda-calcul simplement typé est strictement moins puissant que le lambda-calcul pur. Cependant, les termes typés peuvent toujours être vus (en effaçant les types) comme des termes sans types et donc comme des programmes informatiques. Par conséquent, les termes du lambda-calcul typé sont *des* programmes, mais pas tous *les* programmes.

Remarque technique (pour les plus pointilleuxses et courageuxses). L'opération d'effacement des types, qui permet d'obtenir des termes du lambda-calcul pur à partir des termes typés, mérite quelques explications. Une possibilité est de garder les symboles `tupl`, `fst`, `snd`, `inl`, `inr`, `case` et les règles de réécritures correspondantes en oubliant simplement les règles de typage, mais cela ne donne pas tout à fait du lambda-calcul pur. Si on tient à s'en débarrasser, on peut faire les substitutions suivantes :

$$\begin{aligned}
\texttt{tupl} &\Rightarrow (\lambda abf.fab)\\
\texttt{fst} &\Rightarrow (\lambda p.p(\lambda ab.a))\\
\texttt{snd} &\Rightarrow (\lambda p.p(\lambda ab.b))
\end{aligned}$$

On peut vérifier (en exercice...) qu'on a bien $\texttt{fst}\ (\texttt{tupl}\ x\, y) = x$ et $\texttt{snd}\,(\texttt{tupl}\ x\, y) = y$. De même, on peut effectuer les substitutions :

$$\begin{aligned} \texttt{inl} &\Rightarrow (\lambda pab.ap) \\ \texttt{inr} &\Rightarrow (\lambda pab.bp) \\ \texttt{case} &\Rightarrow (\lambda abc.cab) \end{aligned}$$

et on a bien $\texttt{case}\, f\, g\, (\texttt{inl}\, x) = fx$ et $\texttt{case}\, f\, g\, (\texttt{inr}\, x) = gx$.

5.3 Le lambda-calcul et la logique : Les propositions comme types

Afin d'établir une correspondance entre les preuves et les termes du lambda-calcul typé, on voit les énoncés logiques comme les types de notre lambda-calcul. Voici comment sont construits les types :

- $\texttt{Faux}$ est un type $\bot$ fixé avec, pour tout type α, un terme $\texttt{efq}_\alpha : \bot \to \alpha$;
- Les énoncés atomiques du langage (ceux qui n'utilisent ni conjonctions, ni disjonctions, ni implications) sont ajoutés en tant que types de base. Par exemple, pour les entiers, il peut s'agir d'énoncés comme « $2 + 2 = 5$ ».
- À partir de ces types de base, on peut construire des types plus complexes en utilisant $\times$ pour la conjonction, $+$ pour la disjonction, et $\to$ pour l'implication. Par exemple, l'énoncé

  $$\text{« Si } 2 + 2 = 5 \text{ ou si } 5 + 6 = 11, \text{ alors } 7 \times 9 < 4. \text{ »}$$

  correspondra au type
  $$(2 + 2 = 5) + (5 + 6 = 11) \to (7 \times 9 < 4).$$
- La négation de α correspond au type $\alpha \to \bot$.

On donne par ailleurs pour tout axiome α un terme du type correspondant, qu'on note $\texttt{ax}_\alpha : \alpha$.

5.4 Le lambda-calcul et la logique : La correspondance de Curry-Howard

On constate que chacune des règles d'inférence de la logique intuitionniste correspond à une des règles de calcul du lambda-calcul simplement typé :

- Le modus ponens
  $$\frac{A \qquad A \to B}{B}$$
  correspond à l'évaluation d'un terme $f : A \to B$ en un terme $x : A$, qui est bien un terme fx de type B.
- L'introduction et l'élimination du « et » :
  $$\frac{A \qquad B}{A \text{ et } B}$$
  $$\frac{A \text{ et } B}{A}$$
  $$\frac{A \text{ et } B}{B}$$
  correspondent respectivement à $\texttt{tupl}$, $\texttt{fst}$ et $\texttt{snd}$.
- L'introduction du « ou » :
  $$\frac{A}{A \text{ ou } B}$$
  $$\frac{B}{A \text{ ou } B}$$
  correspond à $\texttt{inl}$ et $\texttt{inr}$, et la disjonction de cas :
  $$\frac{A \Rightarrow C \qquad B \Rightarrow C \qquad A \text{ ou } B}{\text{C}}$$
  correspond à $\texttt{case}$.

La correspondance est parfaite : une preuve intuitionniste de l'énoncé A correspond exactement à un terme (du lambda-calcul typé) de type A. Ainsi, si on arrive à construire un élément d'un type donné, cela signifie qu'on a réussi à démontrer l'énoncé logique associé, et vice-versa.

On remarque que ce sont bien les preuves *intuitionnistes* (et non pas les preuves classiques) qui correspondent à des programmes informatiques, puisque le tiers exclu ne correspond à aucune règle de calcul. [6]

En fait, quand on montre l'existence d'un objet (vérifiant certaines propriétés) de manière intuitionniste, on peut toujours « extraire » de la preuve un témoin de cette existence, c'est-à-dire un tel objet. L'opération algorithmique qui permet de récupérer ce témoin est simplement l'exécution du programme correspondant à la preuve d'existence !

5.5 Le processus de normalisation

On a vu qu'une preuve pouvait être transformée en un programme informatique. Cela dit, il semble y avoir une différence importante : alors qu'un programme informatique peut être exécuté, une telle opération n'a pas d'équivalent évident en termes de preuves.

En fait, l'opération sur les preuves intuitionnistes correspondant (via la correspondance de Curry–Howard) à l'exécution des programmes (c'est-à-dire à la réécriture des termes du lambda-calcul simplement typé) s'appelle la *normalisation*. C'est cette opération qui permet d'extraire les témoins de preuves d'existence à partir des preuves intuitionnistes, leur conférant leur nom de preuves constructives.

Par un théorème de Gentzen, les programmes associés aux termes du lambda-calcul simplement typé terminent tous : cela entraîne que toute preuve intuitionniste peut être normalisée jusqu'à atteindre une preuve qui ne peut plus être normalisée. On parle aussi de preuve *sans coupure*, et la normalisation s'appelle alors l'*élimination des coupures*. Il faut savoir que le coût en calcul de cette opération de normalisation peut être énorme, et les preuves qui en résultent peuvent facilement atteindre des tailles complètement déraisonnables (en gros, on n'a plus le droit d'utiliser de lemmes : tout ce qui sert plusieurs fois doit être démontré à chaque fois qu'il sert) : pour cette raison, il est très utile d'autoriser les coupures dans les preuves, même s'il est théoriquement possible de les éliminer.

Pour conclure ce cours, on va lister les règles correspondant à la normalisation, lesquelles peuvent être directement déduites des règles de calcul du lambda-calcul simplement typé en les traduisant via la correspondance de Curry–Howard :

Règle de normalisation	**Règle de calcul correspondante**
$\dfrac{\dfrac{\vdots}{B} \quad \dfrac{\mathcal{A}}{B \Rightarrow C}}{C}$ avec $\dfrac{\dfrac{\mathcal{A} \quad [B]}{\vdots}}{C} \Rightarrow \dfrac{\mathcal{A} \quad \dfrac{\vdots}{B}}{\dfrac{\vdots}{C}}$	$(\lambda x.t)y \Rightarrow t[x/y]$ où $x, y\colon B$ et $t\colon C$
$\dfrac{\dfrac{\dfrac{\vdots}{A} \quad \dfrac{\vdots}{B}}{A \text{ et } B}}{A} \Rightarrow \dfrac{\vdots}{A}$ et de même pour B	$\texttt{fst}\,(\texttt{tupl}\,x\ y) \Rightarrow x$ et $\texttt{snd}\,(\texttt{tupl}\,x\ y) \Rightarrow y$
$\dfrac{\dfrac{\vdots}{A}}{\dfrac{A \text{ ou } B \quad \dfrac{\vdots}{A \Rightarrow C} \quad \dfrac{\vdots}{B \Rightarrow C}}{C}}$ et de même pour B $\Rightarrow \dfrac{\dfrac{\vdots}{A} \quad \dfrac{\vdots}{A \Rightarrow C}}{C}$	$\texttt{case}\,f\ g\ (\texttt{inl}\,x) \Rightarrow fx$ et $\texttt{case}\,f\ g\ (\texttt{inr}\,y) \Rightarrow gy$

6. Pour la culture, certains langages de programmation possèdent en fait une opération de type $((A \to \bot) \to \bot) \to A$ qui s'appelle `call-cc`.