Partie I: Calculabilité

INF110 (Logique et Fondements de l'Informatique)

David A. Madore
Télécom Paris
david.madore@enst.fr

2023–2025

http://perso.enst.fr/madore/inf110/transp-inf110.pdf

Git: df53831 Fri Apr 25 14:43:57 2025 +0200

Plan

Qu'est-ce que la calculabilité ?

- À l'interface entre **logique mathématique** et **informatique théorique**
  - née de préoccupations venues de la logique (Hilbert, Gödel),
  - à l'origine des 1^{ers} concepts informatiques (λ-calcul, machine de Turing).

- But : étudier les limites de ce que **peut ou ne peut pas faire un algorithme**
  - sans limite de ressources (temps, mémoire juste « finis »),
  - sans préoccupation d'efficacité ($\neq$ complexité, algorithmique),
  - y compris résultats négatifs (« *aucun* algorithme ne peut... »),
  - voire relatifs (calculabilité relative),
  - admettant diverses généralisations (calculabilité supérieure).

Quelques noms

- Muḥammad ibn Mūsá al-H̲wārizmī (v.780–v.850) : $\rightsquigarrow$« algorithme »
- Blaise Pascal (1623–1662) : machine à calculer $\rightsquigarrow$automates
- Charles Babbage (1791–1871) : *Analytical Engine* (Turing-complète !)
- Ada (née Byron) Countess of Lovelace (1815–1852) : programmation
- Richard Dedekind (1831–1916) : définitions primitives récursives
- David Hilbert (1862–1943) : *Entscheidungsproblem* (décider la vérité)
- Jacques Herbrand (1908–1931) : fonctions générales récursives
- Kurt Gödel (1906–1978) : incomplétude en logique
- Haskell Curry (1900–1982) : logique combinatoire, lien preuves-typage
- Alonzo Church (1903–1995) : λ-calcul
- Alan M. Turing (1912–1954) : machine de Turing, problème de l'arrêt
- Emil Post (1897–1954) : ensembles calculablement énumérables
- Stephen C. Kleene (1909–1994) : μ-récursion, th. de récursion, forme normale

Fonction calculable

« Définition » : une fonction f est **calculable** quand il existe un algorithme qui

- prenant en entrée un x du domaine de définition de f,
- **termine en temps fini**,
- et renvoie la valeur $f(x)$.

Difficultés :

- Comment définir ce qu'est un algorithme ?
- Quel type de valeurs (acceptées et renvoyées) ?
- Et si l'algorithme ne termine pas ?
- Distinction entre intention (l'algorithme) et extension (la fonction).

Sans préoccupation d'efficacité

- La calculabilité ne s'intéresse pas à l'efficacité des algorithmes qu'elle étudie, uniquement leur **terminaison en temps fini**.

P.ex. : pour savoir si n est premier, on peut tester si $i \times j = n$ pour tout i et j allant de 2 à $n-1$. (Hyper inefficace ? On s'en fiche.)

- La calculabilité n'a pas peur des grands entiers.

P.ex. : **fonction d'Ackermann** définie par :

$$A(m, n, 0) = m + n$$
$$A(m, 1, k+1) = m$$
$$A(m, n+1, k+1) = A(m,\ A(m, n, k+1),\ k)$$

définition algorithmique (par appels récursifs qui terminent), donc calculable.

Mais $A(2, 6, 3) = 2^{2^{2^{2^2}}} = 2^{2^{65\,536}}$ et $A(2, 4, 4) = A(2, 65\,536, 3)$ est inimaginablement grand (et que dire de $A(100, 100, 100)$?).
$\Rightarrow$ Ingérable sur un vrai ordinateur.

Approches de la calculabilité

- Approche informelle : **algorithme = calcul finitiste** mené par un humain ou une machine, selon des instructions précises, en temps fini, sur des données finies
- Approche pragmatique : tout ce qui peut être fait sur un langage de programmation « Turing-complet » (Python, Java, C, Caml...) idéalisé
  - sans limites d'implémentation (p.ex., entiers arbitraires !),
  - sans source de hasard ou de non-déterminisme.
- Approches formelles, p.ex. :
  - fonctions générales récursives (Herbrand-Gödel-Kleene),
  - λ-calcul (Church) ($\leftrightarrow$ langages fonctionnels),
  - machine de Turing (Turing),
  - machines à registres (Post...).
- **« Thèse » de Church-Turing** : tout ceci donne la même chose.

Thèse de Church-Turing

- **Théorème** (Post, Turing) : les fonctions (disons $\mathbb{N} \dashrightarrow \mathbb{N}$) **(1)** générales récursives, **(2)** représentables en λ-calcul, et **(3)** calculables par machine de Turing, coïncident toutes.

$\Rightarrow$ On parle de calculabilité au sens de Church-Turing.

- **Observation** : tous les langages de programmation informatiques généraux usuels, idéalisés, calculent aussi exactement ces fonctions ($\rightarrow$ « Turing-complets »).

- **Thèse philosophique** : la calculabilité de C-T définit précisément la notion d'algorithme finitiste.

- **Conjecture physique** : la calculabilité de C-T correspond aux calculs réalisables mécaniquement dans l'Univers (en temps/énergie finis mais illimités).

$\uparrow$ (même avec un ordinateur quantique)

Pour toutes ces raisons, le sujet mérite d'être étudié !

Trois grandes approches

On va décrire trois approches des (mêmes !) fonctions calculables au sens de Church-Turing, et esquisser leur équivalence :

- Les **fonctions générales récursives** sont mathématiq$^{\text{t}}$ plus commodes :
  - « tout est un entier » (fonctions $\mathbb{N}^k \dashrightarrow \mathbb{N}$),
  - définition inductive, numérotation associée.
- Les **machines de Turing** représentent des ordinateurs très simples :
  - travaillent sur une « bande » illimitée a priori (mémoire),
  - aspect algorithmique évident, plus proche d'un « vrai » ordinateur,
  - approche la plus commode pour la complexité (pas considérée ici).
- Le λ-**calcul** pur non typé est un système symbolique :
  - proche des langages de program$^{\text{tion}}$ fonctionnels (Lisp, Haskell, OCaml...),
  - plus facile à « programmer » réellement, mais nombreuses subtilités.

Données finies

Un algorithme travaille sur des **données finies**.

Qu'est-ce qu'une « donnée finie » ? Tout objet représentable informatiquement : booléen, entier, chaîne de caractères, structure, liste/tableau de ces choses, ou même plus complexe (p.ex., graphe).

$\rightarrow$ Comment y voir plus clair ?

Deux approches opposées :

- **typage** : distinguer toutes ces sortes de données,
- **codage de Gödel** : tout représenter comme des entiers !

Le typage est plus élégant, plus satisfaisant, plus proche de l'informatique réelle, on en reparlera.

Le codage de Gödel simplifie l'approche/définition de la calculabilité (on étudie juste des fonctions $\mathbb{N} \dashrightarrow \mathbb{N}$).

Codage de Gödel (« tout est un entier »)

- Représenter **n'importe quelle donnée finie par un entier**.
- Codage des couples : par exemple,

$$\langle m, n\rangle := m + \frac{1}{2}(m+n)(m+n+1)$$

définit une bijection calculable $\mathbb{N}^2 \to \mathbb{N}$ (calculable dans les deux sens).

- Codage des listes finies : par exemple,

$$\langle\!\langle a_0, \ldots, a_{k-1}\rangle\!\rangle := \langle a_0, \langle a_1, \langle \cdots, \langle a_{k-1}, 0\rangle + 1 \cdots\rangle + 1\rangle + 1$$

définit une bijection calculable $\{\text{suites finies dans } \mathbb{N}\} \to \mathbb{N}$ (avec $\langle\!\langle\rangle\!\rangle := 0$).

- Il sera aussi utile de représenter même les programmes par des entiers.
- Les détails précis du codage sont **sans importance**.
- Ne pas utiliser dans la vraie vie (hors calculabilité) !

Fonctions partielles

- Même si on s'intéresse à des algorithmes qui **terminent**, la définition de la calculabilité doit forcément passer aussi par ceux qui ne terminent pas.
(Aucun langage Turing-complet ne peut exprimer uniquement des algorithmes qui terminent toujours, à cause de l'indécidabilité du problème de l'arrêt.)

- Lorsque l'algorithme censé calculer $f(n)$ ne termine pas, on dira que f n'est pas définie en n, et on notera $f(n)\uparrow$. Au contraire, s'il termine, on note $f(n)\downarrow$.

- Notation : $f\colon \mathbb{N} \dashrightarrow \mathbb{N}$: une fonction $D \to \mathbb{N}$ définie sur une partie $D \subseteq \mathbb{N}$.
- Notation : $f(n)\downarrow$ signifie « $n \in D$ », et $f(n)\uparrow$ signifie « $n \notin D$ ».
- Notation : $f(n)\downarrow = g(m)$ signifie « $f(n)\downarrow$ et $g(m)\downarrow$ et $f(n) = g(m)$ ».
- Convention : $f(n) = g(m)$ signifie « $f(n)\downarrow$ ssi $g(m)\downarrow$, et $f(n) = g(m)$ si $f(n)\downarrow$ ». (Certains préfèrent écrire $f(n) \simeq g(m)$ pour ça.)

- Convention : si $g_i(\underline{x})\uparrow$ pour un i, on convient que $h(g_1(\underline{x}), \ldots, g_k(\underline{x}))\uparrow$.

- Terminologie : une fonction $f\colon \mathbb{N} \to \mathbb{N}$ définie sur $\mathbb{N}$ est dite **totale**.
Une fonction totale est un cas particulier de fonction partielle !

Terminologie à venir (avant-goût)

- Une fonction partielle $f : \mathbb{N} \dashrightarrow \mathbb{N}$ est dite **calculable** (partielle) lorsqu'il existe un algorithme qui prend n en entrée et :
  - termine (en temps fini) et renvoie $f(n)$ lorsque $f(n) \downarrow$,
  - ne termine pas lorsque $f(n) \uparrow$.

- Une partie $A \subseteq \mathbb{N}$ est dite **décidable** lorsque sa fonction indicatrice $\mathbb{N} \to \mathbb{N}$

$$\mathbf{1}_A : n \mapsto \begin{cases} 1 & \text{si } n \in A \\ 0 & \text{si } n \notin A \end{cases}$$

est calculable (répondre « oui » ou « non » selon que $n \in A$ ou $n \notin A$).

- Une partie $A \subseteq \mathbb{N}$ est dite **semi-décidable** lorsque sa fonction partielle « semi-indicatrice » $\mathbb{N} \dashrightarrow \mathbb{N}$ (d'ensemble de définition A)

$$n \mapsto \begin{cases} 1 & \text{si } n \in A \\ \uparrow & \text{si } n \notin A \end{cases}$$

est calculable (répondre « oui » ou « ... » selon que $n \in A$ ou $n \notin A$).

Point terminologique : « récursif »

Le mot « récursif » et ses cognats (« récursion », « récursivité ») a plusieurs sens apparentés mais non identiques :

- « récursif » = « défini par récurrence » (Dedekind 1888) → fonctions primitives récursives, générales récursives (cf. après) ;
- « récursif » = « calculable » (par glissement à cause de la définition de la calculabilité par les fonctions générales récursives) ;
- « récursif » = « faisant appel à lui-même dans sa définition » (appels récursifs, récursivité en informatique).

On va définir les fonctions « **primitives récursives** » (1^{er} sens) et « **(générales) récursives** » (1^{er} et aussi 2^{e} sens) ci-après.

Pour le 3^{e} sens, on dira « appels récursifs ».

Fonctions primitives récursives : aperçu

- Avant de définir les fonctions générales récursives ($\cong$ calculables), on va commencer par les **primitives récursives**, plus restreintes.
« primitivement récursives » ?

- Historiquement antérieures à la calculabilité de Church-Turing.

- Pédagogiquement utile comme « échauffement ».

- À cheval entre calculabilité (**PR** est une petite classe de calculabilité) et complexité (c'est une grosse classe de complexité).

- Correspond à des programmes à **boucles bornées a priori**.

- Énormément d'algorithmes usuels sont p.r.

- Mais pas tous : p.ex. la fonction d'Ackermann n'est pas p.r.

Fonctions primitives récursives : définition

- **PR** est la plus petite classe de fonctions $\mathbb{N}^k \dashrightarrow \mathbb{N}$ (en fait $\mathbb{N}^k \to \mathbb{N}$), pour k variable qui :
  - contient les projections $\underline{x} := (x_1, \ldots, x_k) \mapsto x_i$;
  - contient les constantes $\underline{x} \mapsto c$;
  - contient la fonction successeur $x \mapsto x + 1$;
  - est stable par composition : si $g_1, \ldots, g_\ell \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ et $h \colon \mathbb{N}^\ell \dashrightarrow \mathbb{N}$ sont p.r. alors $\underline{x} \mapsto h(g_1(\underline{x}), \ldots, g_\ell(\underline{x}))$ est p.r. ;
  - est stable par récursion primitive : si $g \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ et $h \colon \mathbb{N}^{k+2} \dashrightarrow \mathbb{N}$ sont p.r., alors $f \colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est p.r., où :

$$f(\underline{x}, 0) = g(\underline{x})$$
$$f(\underline{x}, z+1) = h(\underline{x}, f(\underline{x}, z), z)$$

Les fonctions p.r. sont automatiq$^{\text{t}}$ totales, mais il est commode de garder la définition avec $\dashrightarrow$.

Fonctions primitives récursives : exemples

- $f\colon (x,z) \mapsto x+z$ est p.r. :

$$f(x,0) = x$$
$$f(x,z+1) = f(x,z)+1$$

où $x \mapsto x$ et $(x,y,z) \mapsto y+1$ sont p.r.

- $f\colon (x,z) \mapsto x \cdot z$ est p.r. :

$$f(x,0) = 0$$
$$f(x,z+1) = f(x,z)+x$$

- $f\colon (x,z) \mapsto x^z$ est p.r.

- $f\colon (x,y,0) \mapsto x,\ (x,y,z) \mapsto y$ si $z \geq 1$ est p.r. :

$$f(x,y,0) = x$$
$$f(x,y,z+1) = y$$

- $u \mapsto \max(u-1,0)$ est p.r. (exercice !), comme $(u,v) \mapsto \max(u-v,0)$ ou $(u,v) \mapsto u\%v$ ou $(u,v) \mapsto \lfloor u/v \rfloor$.

Fonctions primitives récursives : programmation

Les fonctions p.r. sont celles définies par un **langage de programmation à boucles bornées**, c'est-à-dire que :

- les variables sont des entiers naturels (illimités !),
- les manipulations de base sont permises (constantes, affectations, test d'égalité, conditionnelles),
- les opérations arithmétiques basiques sont disponibles,
- on peut faire des appels de fonctions sans appels récursifs,
- on ne peut faire que des boucles de nombre borné *a priori* d'itérations.

Les programmes dans un tel langage **terminent forcément par construction**.

N.B. $(m, n) \mapsto \langle m, n\rangle := m + \frac{1}{2}(m+n)(m+n+1)$ et $\langle m, n\rangle \mapsto m$ et $\langle m, n\rangle \mapsto n$ sont p.r.

Fonctions primitives récursives : lien avec la complexité

En anticipant sur la notion de machine de Turing :

- La fonction $(M, C) \mapsto C'$ qui à une machine de Turing M et une configuration ($=$ ruban+état) C de M associe la configuration suivante **est p.r.**
- Conséquence : la fonction $(n, M, C) \mapsto C^{(n)}$ qui à $n \in \mathbb{N}$ et une machine de Turing M et une configuration C de M associe la configuration atteinte après n étapes d'exécution, **est p.r.**

(Par récursion primitive sur le point précédent.)

- Conséquence : une fonction calculable en complexité p.r. par une machine de Turing est elle-même p.r.

(Calculer une borne p.r. sur le nombre d'étapes, puis appliquer le point précédent.)

- Réciproquement : une p.r. est calculable en complexité p.r.
- Moralité : p.r. $\Leftrightarrow$ de complexité p.r.

Notamment **EXPTIME** $\subseteq$ **PR**.

Fonctions primitives récursives : limitations

La classe **PR** est « à cheval » entre la calculabilité et la complexité.

Rappel : la **fonction d'Ackermann** (pour $m = 2$) définie par :

$$A(2, n, 0) = 2 + n$$
$$A(2, 1, k + 1) = 2$$
$$A(2, n + 1, k + 1) = A(2,\ A(2, n, k + 1),\ k)$$

devrait être calculable. Mais cette définition n'est pas une récursion primitive (pourquoi ?).

- On peut montrer que : si $f : \mathbb{N}^k \to \mathbb{N}$ est p.r., il existe r $(= r(f))$ tel que
$$f(x_1, \ldots, x_k) \leq A(2,\ (x_1 + \cdots + x_k + 3),\ r)$$
- Notamment, $r \mapsto A(2, r, r)$ **n'est pas p.r.**

Pourtant, elle est bien définie par un algorithme clair (et terminant clairement).

Fonctions primitives récursives : numérotation (idée)

- On veut coder les fonctions p.r. (et plus tard : gén$^{\text{ales}}$ récursives) par des entiers.

- Pour (certains) entiers $e \in \mathbb{N}$, on va définir $\psi_e^{(k)} : \mathbb{N}^k \to \mathbb{N}$ primitive récursive, la fonction p.r. codée par e ou ayant e comme **code** (source) / « programme ».

- Toute fonction p.r. $f : \mathbb{N}^k \to \mathbb{N}$ sera un $\psi_e^{(k)}$ pour un certain e.
- Ce e décrit la manière dont f est construite selon la définition de $\mathbf{PR}$ (cf. transp. 16).
- Il faut l'imaginer comme le code source de f (au sens informatique).
- Il n'est pas du tout unique : $f = \psi_{e_1}^{(k)} = \psi_{e_2}^{(k)} = \cdots$
(e = « intention » / f = « extension »)

- On va ensuite se demander si $(e, \underline{x}) \mapsto \psi_e^{(k)}(\underline{x})$ est elle-même p.r. (divulgâchis : non).

Fonctions primitives récursives : numérotation (définition)

On définit $\psi_e^{(k)} : \mathbb{N}^k \dashrightarrow \mathbb{N}$ par induction suivant la défⁿ de $\mathbf{PR}$ (cf. transp. 16) :

- si $e = \langle\langle 0, k, i\rangle\rangle$ alors $\psi_e^{(k)}(x_1 \ldots, x_k) = x_i$ (projections) ;
- si $e = \langle\langle 1, k, c\rangle\rangle$ alors $\psi_e^{(k)}(x_1 \ldots, x_k) = c$ (constantes) ;
- si $e = \langle\langle 2\rangle\rangle$ alors $\psi_e^{(1)}(x) = x + 1$ (successeur) ;
- si $e = \langle\langle 3, k, d, c_1, \ldots, c_\ell\rangle\rangle$ et $g_i := \psi_{c_i}^{(k)}$ et $h := \psi_d^{(\ell)}$, alors $\psi_e^{(k)} : \underline{x} \mapsto h(g_1(\underline{x}), \ldots, g_\ell(\underline{x}))$ (composition) ;
- si $e = \langle\langle 4, k, d, c\rangle\rangle$ et $g := \psi_c^{(k)}$ et $h := \psi_d^{(k+2)}$, alors (récursion primitive)

$$\psi_e^{(k+1)}(\underline{x}, 0) = g(\underline{x})$$
$$\psi_e^{(k+1)}(\underline{x}, z+1) = h(\underline{x}, \psi_e^{(k+1)}(\underline{x}, z), z)$$

(Autres cas non définis, i.e., donnent $\uparrow$.)

- Alors $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ est p.r. ssi $\exists e \in \mathbb{N}.\, (f = \psi_e^{(k)})$ par définition.

P.ex., $e = \langle\langle 4, 1, \langle\langle 3, 3, \langle\langle 2\rangle\rangle, \langle\langle 0, 3, 2\rangle\rangle\rangle, \langle\langle 0, 1, 1\rangle\rangle\rangle = 1\,459\,411\,784\,487 \ldots 780\,615\,609\,825 \approx 1.459 \times 10^{357}$ définit $\psi_e^{(2)}(x, z) = x + z$ avec les conventions de codage du transp. 11.

Manipulation de programmes (version p.r.)

- Penser à e dans $\psi_e^{(k)}$ comme un programme écrit en « langage p.r. ».
- La fonction $\psi_e^{(k)} : \mathbb{N}^k \dashrightarrow \mathbb{N}$ « interprète » le programme e.
- Une fonction p.r. donnée a beaucoup d'indices : $\psi_{e_1}^{(k)} = \psi_{e_2}^{(k)} = \cdots$ (programmes équivalents).

*

La numérotation (transp. précédent) rend p.r. beaucoup de manipulations usuelles de programmes (composition, récursion, etc.). Notamment :

- **Théorème s-m-n** (Kleene) : il existe $s_{m,n} : \mathbb{N}^{m+1} \to \mathbb{N}$ p.r. telle que

$$(\forall e, \underline{x}, \underline{y}) \quad \psi^{(n)}_{s_{m,n}(e,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \psi_e^{(m+n)}(x_1, \ldots, x_m,\ y_1, \ldots, y_n)$$

Preuve : $s_{m,n}(e, \underline{x}) = \langle\langle 3, n, e, \langle\langle 1, n, x_1\rangle\rangle, \ldots, \langle\langle 1, n, x_m\rangle\rangle,\ \langle\langle 0, n, 1\rangle\rangle, \ldots, \langle\langle 0, n, n\rangle\rangle\rangle$ avec nos conventions (composition de fonctions constantes et de projections). □

En clair : $s_{m,n}$ prend un programme e qui prend $m + n$ arguments en entrée et « fixe » la valeur des m premiers arguments à $x_1, \ldots, x_m$, les n arguments suivants $(y_1, \ldots, y_n)$ étant gardés variables.

Digression : l'astuce de Quine (intuition)

Le nom de Willard Van Orman Quine (1908–2000) a été associé à cette astuce par Douglas Hofstadter. En fait, l'astuce est plutôt due à Cantor, Gödel, Turing ou Kleene.

Les mots suivants suivis des mêmes mots entre guillemets forment une phrase intéressante : « les mots suivants suivis des mêmes mots entre guillemets forment une phrase intéressante ».

Pseudocode :

```
str="somefunc(code) { /*...*/ }\nsomefunc(\"str=\"+quote(str)+str);\n";
somefunc(code) { /*...*/ }
somefunc("str="+quote(str)+str);
```

$\Rightarrow$ La fonction `somefunc` (arbitraire) est appelée avec le code source du programme tout entier.

Exercice : utiliser cette astuce pour écrire un programme écrivant son propre code source.

Moralité : on peut toujours donner aux programmes accès à leur code source, même si ce n'est pas prévu par le langage.

Le théorème de récursion de Kleene (version p.r.)

Version formelle de l'astuce de Quine

- **Théorème** (Kleene) : si $h\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est p.r., il existe e tel que

$$(\forall \underline{x}) \quad \psi_e^{(k)}(\underline{x}) = h(e, \underline{x})$$

Plus précisément, il existe $b\colon \mathbb{N}^2 \to \mathbb{N}$ p.r. telle que $e := b(k, d)$ vérifie

$$(\forall d)\,(\forall \underline{x}) \quad \psi_e^{(k)}(\underline{x}) = \psi_d^{(k+1)}(e, \underline{x})$$

Preuve : soit $s := s_{1,k}$ donné par le théorème s-m-n. La fonction $(t, \underline{x}) \mapsto h(s(t,t), \underline{x})$ est p.r., disons $= \psi_c^{(k+1)}(t, \underline{x})$. Alors

$$\psi_{s(c,c)}^{(k)}(\underline{x}) = \psi_c^{(k+1)}(c, \underline{x}) = h(s(c,c), \underline{x})$$

donc $e := s(c,c)$ convient. Les fonctions $d \mapsto c$ et $c \mapsto e$ sont p.r. □

Moralité : on peut donner aux programmes accès à leur propre numéro (= « code source », ici e), cela ne change rien.

Fonctions primitives récursives : absence d'universalité

- **Théorème :** il n'existe pas de fonction p.r. $u\colon \mathbb{N}^2 \to \mathbb{N}$ telle que $u(e,x) = \psi_e^{(1)}(x)$ si $\psi_e^{(1)}(x)\downarrow$.

Preuve : par l'absurde : si un tel u existe, alors $(e,x) \mapsto u(e,x)+1$ est p.r. Par le théorème de récursion de Kleene, il existe e tel que $\psi_e^{(1)}(x) = u(e,x)+1$, ce qui contredit $u(e,x) = \psi_e^{(1)}(x)$. □

*

Moralité : un interpréteur du langage p.r. ne peut pas être p.r. (preuve : on peut interpréter l'interpréteur s'interprétant lui-même, en ajoutant 1 au résultat ceci donne un paradoxe ; c'est un argument diagonal de Cantor).

- Cet argument dépend du théorème s-m-n et du fait que les fonctions p.r. sont totales. Pour définir une théorie satisfaisante de la calculabilité, on va sacrifier la totalité pour sauver le théorème s-m-n.

Cette même preuve deviendra alors la preuve de l'indécidabilité du problème de l'arrêt.

Fonctions primitives récursives : absence d'universalité (variante)

Rappel : la **fonction d'Ackermann** est définie par :

$$A(m,n,0) = m+n$$
$$A(m,1,k+1) = m$$
$$A(m,n+1,k+1) = A(m,\, A(m,n,k+1),\, k)$$

- Pour un k fixé, la fonction $(m,n) \mapsto A(m,n,k)$ est p.r. (par récurrence sur k, récursion primitive sur $A(m,n,k-1)$).

- Il existe même $k \mapsto a(k)$ p.r. telle que $\psi^{(2)}_{a(k)}(m,n) = A(m,n,k)$.

I.e., on peut calculer de façon p.r. en k le code d'un programme p.r. qui calcule $(m,n) \mapsto A(m,n,k)$.

- Si (une extension de) $(e,n) \mapsto \psi^{(1)}_e(n)$ était p.r., on pourrait calculer $(n,k) \mapsto \psi^{(1)}_{s_{1,1}(a(k),2)}(n) = \psi^{(2)}_{a(k)}(2,n) = A(2,n,k)$, or elle n'est pas p.r.

Fonctions générales récursives : aperçu

- On a vu que les fonctions p.r. sont limitées et ne couvrent pas la notion générale d'algorithme :
  - les algorithmes p.r. terminent toujours car
  - le langage ne permet pas de boucles non bornées ;
  - concrètement, il n'implémente pas la fonction d'Ackermann ;
  - il ne peut pas s'interpréter lui-même.

- On veut modifier la définition des fonctions p.r. pour lever ces limitations. On va autoriser les boucles infinies.
$\rightarrow$ Fonctions **générales récursives** ou simplement « **récursives** ».
Ce seront aussi nos fonctions **calculables** !

- En ce faisant, on obtient forcément des cas de non-terminaisons, donc on doit passer par des fonctions partielles.

N.B. Terminologie fluctuante : fonctions « générales récursives » ? juste « récursives » ? « récursives partielles » ? « calculables » ? « calculables partielles » ?

L'opérateur μ de Kleene

Définition : si $g\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ et $\underline{x} \in \mathbb{N}^k$, alors $\mu g(\underline{x})$ est le plus petit z tel que $g(z, \underline{x}) = 0$ et $g(i, \underline{x}) \downarrow$ pour $0 \leq i < z$, s'il existe.

Autrement dit, $\mu g(\underline{x}) = z$ signifie

- $g(z, \underline{x}) = 0$,
- $g(i, \underline{x}) > 0$ (sous-entendant $g(i, \underline{x}) \downarrow$) pour tout $0 \leq i < z$.

Concrètement, penser à μg comme la fonction

```
i=0; while (true) { if (g(i,x)==0) { return i; } i++; }
```

- Ceci permet toute sorte de recherche non bornée.

- L'opérateur μ associe à une fonction $g\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ une fonction $\mu g\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$.

Fonctions générales récursives : définition

- **R** est la plus petite classe de fonctions $\mathbb{N}^k \dashrightarrow \mathbb{N}$, pour k variable qui :
  - contient les projections $\underline{x} := (x_1, \ldots, x_k) \mapsto x_i$;
  - contient les constantes $\underline{x} \mapsto c$;
  - contient la fonction successeur $x \mapsto x + 1$;
  - est stable par composition : si $g_1, \ldots, g_\ell \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ et $h \colon \mathbb{N}^\ell \dashrightarrow \mathbb{N}$ sont récursives alors $\underline{x} \mapsto h(g_1(\underline{x}), \ldots, g_\ell(\underline{x}))$ est récursive ;
  - est stable par récursion primitive : si $g \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ et $h \colon \mathbb{N}^{k+2} \dashrightarrow \mathbb{N}$ sont récursives, alors $f \colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est récursive, où :
  $$f(\underline{x}, 0) = g(\underline{x})$$
  $$f(\underline{x}, z+1) = h(\underline{x}, f(\underline{x}, z), z)$$
  - est stable par l'opérateur μ : si $g \colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est récursive, alors $\mu g \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ est récursive (← nouveau !).

Cette fois le langage permet les boucles non bornées !

Fonctions générales récursives : numérotation (idée)

- On veut coder les fonctions générales récursives par des entiers.

Exactement comme on l'a fait pour les fonctions p.r., on change juste la notation de ψ en φ.

- Pour (certains) entiers $e \in \mathbb{N}$, on va définir $\varphi_e^{(k)} : \mathbb{N}^k \dashrightarrow \mathbb{N}$ générale récursive, celle codée par e ou ayant e comme **code** (source) / « programme ».

- Toute fonction récursive (partielle !) $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ sera un $\varphi_e^{(k)}$ pour un certain e.
- Ce e décrit la manière dont f est construite selon la définition de $\mathbf{R}$ (cf. transp. 30).
- Il faut l'imaginer comme le code source de f (au sens informatique).
- Il n'est pas du tout unique : $f = \varphi_{e_1}^{(k)} = \varphi_{e_2}^{(k)} = \cdots$

($e =$ « intention » / $f =$ « extension »)

- On va ensuite se demander si $(e, \underline{x}) \mapsto \varphi_e^{(k)}(\underline{x})$ est elle-même récursive (divulgâchis : oui, contrairement au cas p.r.).

Fonctions générales récursives : numérotation (définition)

On définit $\varphi_e^{(k)} : \mathbb{N}^k \dashrightarrow \mathbb{N}$ par induction suivant la déf$^{\text{n}}$ de $\mathbf{R}$ (cf. transp. 30) :

- si $e = \langle\langle 0, k, i \rangle\rangle$ alors $\varphi_e^{(k)}(x_1 \ldots, x_k) = x_i$ (projections) ;
- si $e = \langle\langle 1, k, c \rangle\rangle$ alors $\varphi_e^{(k)}(x_1 \ldots, x_k) = c$ (constantes) ;
- si $e = \langle\langle 2 \rangle\rangle$ alors $\varphi_e^{(1)}(x) = x + 1$ (successeur) ;
- si $e = \langle\langle 3, k, d, c_1, \ldots, c_\ell \rangle\rangle$ et $g_i := \varphi_{c_i}^{(k)}$ et $h := \varphi_d^{(\ell)}$, alors $\varphi_e^{(k)} : \underline{x} \mapsto h(g_1(\underline{x}), \ldots, g_\ell(\underline{x}))$ (composition) ;
- si $e = \langle\langle 4, k, d, c \rangle\rangle$ et $g := \varphi_c^{(k)}$ et $h := \varphi_d^{(k+2)}$, alors (récursion primitive)

$$\varphi_e^{(k+1)}(\underline{x}, 0) = g(\underline{x})$$
$$\varphi_e^{(k+1)}(\underline{x}, z+1) = h(\underline{x}, f(\underline{x}, z), z)$$

- si $e = \langle\langle 5, k, c \rangle\rangle$ et $g := \varphi_c^{(k+1)}$, alors $\varphi_e^{(k)} = \mu g$.

(Autres cas non définis, i.e., donnent $\uparrow$.)

- Alors $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ est récursive ssi $\exists e \in \mathbb{N}.\, (f = \varphi_e^{(k)})$ par définition.

Fonctions générales récursives : universalité

Cette fois, il existe bien une fonction récursive universelle, c'est-à-dire :

- $(e, \underline{x}) \mapsto \varphi_e^{(k)}(\underline{x})$ est récursive : $\exists u_k \in \mathbb{N}. (\varphi_{u_k}^{(k+1)}(e, \underline{x}) = \varphi_e^{(k)}(\underline{x}))$.

Variante : $\exists u. (\varphi_u^{(1)}(\langle\langle e, \underline{x} \rangle\rangle) = \varphi_e^{(k)}(\underline{x}))$ en codant programme e et arguments $\underline{x}$ en un entier.

Qu'est-ce que ça nous dit ?

- Mathématiquement, que $\varphi^{(k)}$ est elle-même récursive en tous ses arguments, y compris l'indice de numérotation.
- Informatiquement, qu'il existe un interpréteur u du langage général récursif dans le langage général récursif.
- Philosophiquement, que la notion d'« exécuter un algorithme » est elle-même algorithmique.

Mais comment le prouver ? On va esquisser une méthode par **arbres de calcul**.

Arbres de calcul pour les fonctions récursives

Un **arbre de calcul** de $\varphi_e^{(k)}(\underline{x})$ de résultat y est un arbre (fini, enraciné, ordonné, étiqueté par des entiers naturels) qui « atteste » que $\varphi_e^{(k)}(\underline{x}) = y$ en détaillant les étapes du calcul.

- La racine est étiquetée « $\varphi_e^{(k)}(\underline{x}) = y$ » codé disons « $\langle\langle e, \langle\langle \underline{x} \rangle\rangle, y \rangle\rangle$ ».
- Le sous-arbre porté par chaque fils de la racine est lui-même un arbre de calcul pour une sous-expression utilisée dans le calcul de $f(\underline{x}) = y$.
- Pour les cas projection, constante, successeur, il n'y a pas de fils.
- Pour la composition $h(g_1(\underline{x}), \ldots, g_\ell(\underline{x}))$, les fils portent des arbres de calcul attestant $g_1(\underline{x}) = v_1, \ldots, g_\ell(\underline{x}) = v_\ell$ et $h(\underline{v}) = y$ (donc $\ell + 1$ fils).
- Pour la récursion primitive $f(\underline{x}, z)$, on a soit un seul fils attestant $g(\underline{x}) = y$ lorsque $z = 0$ soit deux fils attestant $f(\underline{x}, z') = v$ et $h(\underline{x}, v, z') = y$ lorsque $z = z' + 1$ (donc 1 ou 2 fils).
- Pour $f = \mu g$, on a des fils attestant $g(0, \underline{x}) = v_0, \ldots, g(y, \underline{x}) = v_y$ où $v_i > 0$ si $0 \leq i < y$ et $v_y = 0$ (donc $y + 1$ fils).

On a $\varphi_e^{(k)}(\underline{x}) = y$ ssi il en existe un arbre de calcul qui l'atteste.

Arbres de calcul : formalisation

Formellement :

- la racine est étiquetée $\langle\!\langle e, \langle\!\langle \underline{x} \rangle\!\rangle, y \rangle\!\rangle$,
- soit e est $\langle\!\langle 0, k, i \rangle\!\rangle$ ou $\langle\!\langle 1, k, c \rangle\!\rangle$ ou $\langle\!\langle 2 \rangle\!\rangle$, elle n'a pas de fils, et y vaut resp$^{\text{t}}$ x_i, c ou $x+1$,
- soit $e = \langle\!\langle 3, k, d, c_1, \ldots, c_\ell \rangle\!\rangle$ et les $\ell+1$ fils portent des arbres de calculs attestant $\varphi_{c_1}^{(k)}(\underline{x}) = v_1$, ..., $\varphi_{c_\ell}^{(k)}(\underline{x}) = v_\ell$ et $\varphi_d^{(\ell)}(\underline{v}) = y$.
- soit $e = \langle\!\langle 4, k', d, c \rangle\!\rangle$ où $k' = k-1$ et
  - soit $x_k = 0$ et l'unique fils porte arbre de calcul attestant $\varphi_c^{(k')}(x_1, \ldots, x_{k'}) = y$,
  - soit $x_k = z+1$ et les deux fils portent arbres de calcul attestant $\varphi_e^{(k'+1)}(x_1, \ldots, x_{k'}, z) = v$ et $\varphi_d^{(k'+2)}(x_1, \ldots, x_{k'}, v, z) = y$.
- soit $e = \langle\!\langle 5, k, c \rangle\!\rangle$ et les $y+1$ fils portent des arbres de calcul attestant $\varphi_c^{(k+1)}(0, x_1, \ldots, x_k) = v_0$, ..., $\varphi_c^{(k+1)}(y, x_1, \ldots, x_k) = v_y$ où $v_y = 0$ et $v_0, \ldots, v_{y-1} > 0$.

On encode l'arbre $\mathscr{T}$ par l'entier $\mathrm{code}(\mathscr{T}) := \langle\!\langle n, \mathrm{code}(\mathscr{T}_1), \ldots, \mathrm{code}(\mathscr{T}_s) \rangle\!\rangle$ où n est l'étiquette de la racine et $\mathscr{T}_1, \ldots, \mathscr{T}_s$ les codes des sous-arbres portés par les s fils de celle-ci.

Arbres de calcul $\Rightarrow$ universalité

Les points-clés :

- On a $\varphi_e^{(k)}(\underline{x}) = y$ ssi il existe un arbre de calcul $\mathscr{T}$ l'attestant.
- Vérifier si $\mathscr{T}$ est un arbre de calcul valable est primitif récursif en $\mathrm{code}(\mathscr{T})$. (On peut vérifier les règles à chaque nœud avec des boucles bornées.)
- De même, extraire $e, \underline{x}, y$ de $\mathscr{T}$ est primitif récursif.

D'où l'algorithme « universel » pour calculer $\varphi_e^{(k)}(\underline{x})$ en fonction de $e, \underline{x}$:

- parcourir $n = 0, 1, 2, 3, 4, \ldots$ (boucle non bornée),
- pour chacun, tester s'il code un arbre de calcul valable de $\varphi_e^{(k)}(\underline{x})$,
- si oui, terminer et renvoyer le y contenu.

La boucle non-bornée est précisément ce que permet μ. Tout le reste est p.r.
$\Rightarrow$ Ceci montre l'existence de u (code de l'algorithme décrit ci-dessus).

Ne pas coder un interpréteur comme ça dans la vraie vie !

Théorème de la forme normale

On a montré un peu plus que l'universalité : on peut exécuter n'importe quel algorithme avec une unique boucle non bornée. Plus exactement :

- **Théorème de la forme normale** (Kleene) : il existe un prédicat p.r. T sur $\mathbb{N}^3$ et une fonction p.r. $U : \mathbb{N} \to \mathbb{N}$ tels que :

$$\varphi_e^{(k)}(x_1, \ldots, x_k) = U(\mu T(e, \langle\langle x_1, \ldots, x_k \rangle\rangle))$$

Précisément, $T(n, e, \langle\langle x_1, \ldots, x_k \rangle\rangle)$ teste si n est le code d'un arbre de calcul valable de $\varphi_e^{(k)}(\underline{x})$, et U extrait le résultat de cet arbre.

*

Exemple d'application : **lancement en parallèle** :

$$U(\mu(T(e_1, \langle\langle \underline{x} \rangle\rangle) \text{ ou } T(e_2, \langle\langle \underline{x} \rangle\rangle)))$$

définit (de façon p.r. en e_1, e_2) un e tel que

$$\varphi_e(\underline{x})\downarrow \iff \varphi_{e_1}(\underline{x})\downarrow \text{ ou } \varphi_{e_2}(\underline{x})\downarrow$$

Théorème s-m-n (version générale récursive)

Exactement comme la version p.r. :

- **Théorème s-m-n** (Kleene) : il existe $s_{m,n} : \mathbb{N}^{m+1} \to \mathbb{N}$ p.r. telle que

$$(\forall e, \underline{x}, \underline{y}) \quad \varphi^{(n)}_{s_{m,n}(e,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \varphi^{(m+n)}_e(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

Noter que $s_{m,n}$ est primitive récursive même si on s'intéresse ici aux fonctions générales récursives.

Les manipulations de programmes sont typiquement p.r. (même si les programmes manipulés sont des fonctions générales récursives).

Arité et encodage des tuples

Remarque qui aurait dû être faite avant ?

Pour tout $k \geq q$, les fonctions

$$\begin{cases} \mathbb{N}^k \to \mathbb{N} \\ (x_1, \ldots, x_k) \mapsto \langle\langle x_1, \ldots, x_k \rangle\rangle \end{cases} \quad \text{et} \quad \begin{cases} \mathbb{N} \to \mathbb{N} \\ \langle\langle x_1, \ldots, x_k \rangle\rangle \mapsto x_i \end{cases}$$

sont p.r. Par conséquent,

$$f : \mathbb{N}^k \dashrightarrow \mathbb{N} \text{ récursive} \iff \begin{cases} \mathring{f} : \mathbb{N} \dashrightarrow \mathbb{N} \\ \quad \langle\langle x_1, \ldots, x_k \rangle\rangle \mapsto f(x_1, \ldots, x_k) \end{cases} \text{ récursive}$$

et de plus, un numéro e de f (i.e., $f = \varphi_e^{(k)}$) se calcule de façon p.r. à partir d'un numéro e' de $\mathring{f}$ (i.e., $\mathring{f} = \varphi_{e'}^{(1)}$) et vice versa.

Ceci justifie d'omettre parfois abusivement l'arité
(par défaut, « φ_e » désigne « $\varphi_e^{(1)}$ »).

Même chose, *mutatis mutandis* (avec ψ) pour les fonctions p.r. elles-mêmes.

Le théorème de récursion de Kleene (version générale récursive)

Exactement comme la version p.r. :

- **Théorème** (Kleene) : si $h\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est récursive, il existe e tel que

$$(\forall \underline{x}) \quad \varphi_e^{(k)}(\underline{x}) = h(e, \underline{x})$$

Plus précisément, il existe $b\colon \mathbb{N}^2 \to \mathbb{N}$ p.r. telle que $e := b(k, d)$ vérifie

$$(\forall d)\,(\forall \underline{x}) \quad \varphi_e^{(k)}(\underline{x}) = \varphi_d^{(k+1)}(e, \underline{x})$$

Même preuve : soit $s := s_{1,k}$ donné par le théorème s-m-n. La fonction $(t, \underline{x}) \mapsto h(s(t,t), \underline{x})$ est générale récursive, disons $= \varphi_c^{(k+1)}(t, \underline{x})$. Alors

$$\varphi_{s(c,c)}^{(k)}(\underline{x}) = \varphi_c^{(k+1)}(c, \underline{x}) = h(s(c,c), \underline{x})$$

donc $e := s(c,c)$ convient. Les fonctions $d \mapsto c$ et $c \mapsto e$ sont p.r. □

Moralité : on peut donner aux programmes accès à leur propre numéro (= « code source »), cela ne change rien.

Le théorème du point fixe de Kleene-Rogers

Reformulation du théorème de récursion utilisant l'universalité :

- **Théorème** (Kleene-Rogers) : si $F\colon \mathbb{N} \dashrightarrow \mathbb{N}$ est récursive et $k \in \mathbb{N}$, il existe e tel que

$$\varphi_e^{(k)} = \varphi_{F(e)}^{(k)}$$

Preuve : $h\colon (e, \underline{x}) \mapsto \varphi_{F(e)}^{(k)}(\underline{x})$ est récursive car $e \mapsto F(e)$ l'est et que $(e', \underline{x}) \mapsto \varphi_{e'}^{(k)}(\underline{x})$ l'est (universalité). Par le théorème de récursion, il existe e tel que $\varphi_e^{(k)}(\underline{x}) = h(e, \underline{x}) = \varphi_{F(e)}^{(k)}(\underline{x})$. □

Moralité : quelle que soit la transformation F calculable effectuée sur le source d'un programme, il y a un programme e qui fait la même chose que son transformé $F(e)$.

Récursion !

Le langage des fonctions générales récursives, malgré le nom ne permet pas les définitions par appels récursifs.

Uniquement des opérations élémentaires, appels de fonctions précédemment définies, boucles.

Comment permettre quand même les appels récursifs ?

Par le théorème de récursion de Kleene ! (ou théorème du point fixe) :

- je veux définir (comme fonction générale récursive) une fonction f dont la définition fait appel à f elle-même :
- par le théorème de récursion de Kleene (« astuce de Quine »), je peux supposer que f a accès à son propre numéro (« code source »),
- je convertis chaque appel à f depuis f en un appel à la fonction universelle (interpréteur) sur le numéro de f.

Ne pas implémenter la récursion comme ça dans la vraie vie !

Kids, don't try this at home !

Pseudocode :

```
fibonacci(n) {
str = "self = \"fibonacci(n) {\\nstr = \" + quote(str) + str;\n\
if (n==0 || n==1) return n;\n\
return interpret(self, n-1) + interpret(self, n-2);\n\
}";
self = "fibonacci(n) {\nstr = " + quote(str) + str;
if (n==0 || n==1) return n;
return interpret(self, n-1) + interpret(self, n-2);
}
```

*

Défi : trouver explicitement un e tel que $\varphi_e^{(3)}$ soit la fonction d'Ackermann.

(La fonction d'Ackermann a été définie par des appels récursifs donc elle est bien censée être calculable.)

Le problème de l'arrêt

Le terme « problème de l'arrêt » prendra plus de sens pour les machines de Turing.

- **Problème :** donné un programme e (mettons d'arité $k = 1$) et une entrée x à ce programme, comment savoir si l'algorithme e termine (c'est-à-dire $\varphi_e^{(1)}(x) \downarrow$) ou non ($\varphi_e^{(1)}(x) \uparrow$) sur cette entrée ?

Cette question est-elle algorithmique ?

Réponse de Turing : non.

- Intuition de la preuve : supposons que j'aie un moyen algorithmique pour savoir si un algorithme termine ou pas, je peux lui demander ce que « je » vais faire (astuce de Quine !), et faire le contraire, ce qui conduit à un paradoxe.

L'indécidabilité du problème de l'arrêt

- **Théorème** (Turing) : il n'existe pas de fonction récursive $h\colon \mathbb{N}^2 \to \mathbb{N}$ telle que $h(e,x) = 1$ si $\varphi_e^{(1)}(x) \downarrow$ et $h(e,x) = 0$ si $\varphi_e^{(1)}(x) \uparrow$.

Preuve : par l'absurde : si un tel h existe, alors la fonction

$$v\colon (e,x) \mapsto \begin{cases} 42 & \text{si } h(e,x) = 0 \\ \uparrow & \text{si } h(e,x) = 1 \end{cases}$$

est générale récursive (tester is $h(e,x) = 0$, si oui renvoyer 42, sinon faire une boucle infinie, p.ex. $\mu(x \mapsto 1)$).

Par le théorème de récursion de Kleene, il existe e tel que $\varphi_e^{(1)}(x) = v(e,x)$.

Si $\varphi_e^{(1)}(x) \downarrow$ alors $h(e,x) = 1$ donc $v(e,x) \uparrow$ donc $\varphi_e^{(1)}(x) \uparrow$, contradiction.
Si $\varphi_e^{(1)}(x) \uparrow$ alors $h(e,x) = 0$ donc $v(e,x) \downarrow$ donc $\varphi_e^{(1)}(x) \downarrow$, contradiction. □

L'indécidabilité du problème de l'arrêt : redite

Notons φ pour $\varphi^{(1)}$.

- **Théorème** (Turing) : il n'existe pas de fonction récursive $h\colon \mathbb{N}^2 \to \mathbb{N}$ telle que $h(e,x) = 1$ si $\varphi_e(x) \downarrow$ et $h(e,x) = 0$ si $\varphi_e(x) \uparrow$.

Preuve (incluant celle du théorème de récursion) : considérons la fonction $v\colon \mathbb{N} \dashrightarrow \mathbb{N}$ qui à e associe 42 si $h(e,e) = 0$ et $\uparrow$ (non définie) si $h(e,e) = 1$.
Supposons par l'absurde h est calculable : alors cette fonction (partielle) v est calculable, disons $v = \varphi_c$.
Si $\varphi_c(c) \downarrow$ alors $h(c,c) = 1$ donc $v(c) \uparrow$, c'est-à-dire $\varphi_c(c) \uparrow$, contradiction.
Si $\varphi_c(c) \uparrow$ alors $h(c,c) = 0$ donc $v(c) \downarrow$, c'est-à-dire $\varphi_c(c) \downarrow$, contradiction. □

C'est un argument diagonal : on utilise h pour construire une fonction qui diffère en tout point de la diagonale $c \mapsto \varphi_c(c)$, donc elle ne peut pas être une φ_c.

Pour les fonctions p.r. (qui terminent toujours !), le même argument diagonal donnait l'inexistence d'un programme universel (transp. 26).

Comparaison fonctions primitives récursives et générales récursives

Récapitulation :

- Les fonctions p.r. sont totales ; les générales récursives sont possiblement partielles.
- Les fonctions p.r. sont un langage limité (pas de boucle non bornées a priori) ; les générales récursives coïncideront avec les fonctions « calculables » (équivalence avec machines de Turing et λ-calcul à voir).
- Les fonctions p.r. ne permettent pas d'interpréter les fonctions p.r. ; les générales récursives peuvent s'interpréter elles-mêmes (universalité) et donc réaliser n'importe quelle sorte d'appels récursifs.
- Le problème de l'arrêt pour les fonctions p.r. est trivial (elles sont totales !) ; pour les fonctions générales récursives, il est indécidable (= pas calculable par une fonction générale récursive).

Machines de Turing : explication informelle

La **machine de Turing** est une modélisation d'un ordinateur extrêmement simple, réalisant des calculs indiscutablement finitistes.

C'est une sorte d'automate doté d'un **état** interne pouvant prendre un nombre fini de valeurs, et d'une mémoire illimitée sous forme de **bande** linéaire divisée en cellules (indéfiniment réécrivibles), chaque cellule pouvant contenir un **symbole**.

La machine peut observer, outre son état interne, une unique case de la bande, là où se trouve sa **tête de lecture/écriture**.

Le **programme** de la machine indique, pour chaque combinaison de l'état interne et du symbole lu par la tête :

- dans quel état passer,
- quel symbole écrire à la place de la tête,
- la direction dans laquelle déplacer la tête (gauche ou droite).

La machine suit son programme jusqu'à tomber dans un état spécial 0 (« arrêt »).

Machines de Turing : définition

Une **machine de Turing** (déterministe) à (1 bande, 2 symboles et) $m \geq 2$ états est la donnée de :

- un ensemble fini Q de cardinal m d'**états**, qu'on identifiera à $\{0, \dots, m-1\}$,
- un ensemble Σ de (ici) 2 **symboles de bande** qu'on identifiera à $\{0, 1\}$,
- une fonction
$$\delta \colon (Q \setminus \{0\}) \times \Sigma \to Q \times \Sigma \times \{\mathtt{L}, \mathtt{R}\}$$
appelé **programme** de la machine.

(Il y a donc $(4m)^{2(m-1)}$ machines à m états.)

Une **configuration** d'une telle machine est la donnée de :

- un élément $q \in Q$ appelé l'**état courant**,
- une fonction $\beta \colon \mathbb{Z} \to \Sigma$ ne prenant qu'un nombre fini de valeurs $\neq 0$, appelée la **bande**,
- un entier $i \in \mathbb{Z}$ appelé la **position de la tête** sur la bande.

Machines de Turing : exécution d'une étape

Si (q, β, i) est une configuration de la machine de Turing où $q \neq 0$, et δ le programme, la **configuration suivante** est (q', β', i') où :

- $(q', y, d) = \delta(q, \beta(i))$ est l'**instruction exécutée**,
- q' est le **nouvel état**,
- $i' = i - 1$ si $d = \mathtt{L}$ et $i' = i + 1$ si $d = \mathtt{R}$,
- $\beta'(j) = \beta(j)$ pour $j \neq i$ tandis que $\beta'(i) = y$.

En clair : le programme indique, pour chaque configuration d'un état $\neq 0$ et d'un symbole $x = \beta(i)$ lu sur la bande :

- le nouvel état q' dans lequel passer,
- le symbole y à écrire à la place de x à l'emplacement i de la bande,
- la direction dans laquelle déplacer la tête (gauche ou droite).

Machines de Turing : exécution complète

- Si $C = (q, \beta, i)$ est une configuration d'une machine de Turing, la **trace d'exécution** à partir de C est la suite finie ou infinie $C^{(0)}, C^{(1)}, C^{(2)}, \ldots$, où
  - $C^{(0)} = C$ est la configuration donnée (configuration initiale),
  - si $C^{(n)} = (q^{(n)}, \beta^{(n)}, i^{(n)})$ avec $q^{(n)} = 0$ alors la suite s'arrête ici, on dit que **la machine s'arrête**, que $C^{(n)}$ est la **configuration finale**, et que l'exécution a duré n **étapes**,
  - sinon, $C^{(n+1)}$ est la configuration suivante (définie transp. précédent).

En clair : la machine continue à exécuter des instructions tant qu'elle n'est pas tombée dans l'état 0. Elle s'arrête quand elle tombe dans l'état 0.

Simulation des machines de Turing par les fonctions récursives

- On peut coder un programme et/ou une configuration sous forme d'entiers naturels.
  Le ruban a un nombre fini de symboles $\neq 0$, donc on peut le coder par la liste de leurs positions comptées, disons, à partir du symbole $\neq 0$ le plus à gauche.

- La fonction $(M, C) \mapsto C'$ qui à une machine de Turing M et une configuration C de M associe la configuration suivante **est p.r.**

- Conséquence : la fonction $(n, M, C) \mapsto C^{(n)}$ qui à $n \in \mathbb{N}$ et une machine de Turing M et une configuration C de M associe la configuration atteinte après n étapes d'exécution, **est p.r.**

- La fonction qui à (M, C) associe la configuration finale (et/ou le nombre d'étapes d'exécution) si la machine s'arrête, et $\uparrow$ (non définie) si elle ne s'arrête pas, est **générale récursive**.

Moralité : les fonctions récursives peuvent simuler les machines de Turing.

Calculs sur machines de Turing : une convention

On dira qu'une fonction $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ est **calculable par machine de Turing** lorsqu'il existe une machine de Turing qui, pour tous $x_1, \ldots, x_k$:

- part de la configuration initiale suivante : l'état est 1, les symboles $\beta(j)$ du ruban pour $j < 0$ sont arbitraires (tous 0 sauf un nombre fini), la tête est à l'emplacement 0,
- les symboles $\beta(j)$ pour $j \geq 0$ du ruban initial forment le mot suivant :
$$01^{x_1}01^{x_2}0\cdots01^{x_k}0$$
(suivi d'une infinité de 0), c'est-à-dire $\beta(0) = 0$, $\beta(j) = 1$ si $1 \leq j \leq x_1$, $\beta(1 + x_1) = 0$, $\beta(j) = 1$ si $2 + x_1 \leq j \leq 1 + x_1 + x_2$, etc.,
- si $f(x_1, \ldots, x_k)\uparrow$, la machine ne s'arrête pas,
- si $f(x_1, \ldots, x_k)\downarrow = y$, la machine s'arrête avec la tête à l'emplacement 0 (le même qu'au départ), le ruban $\beta(j)$ non modifié pour $j < 0$, et
- les symboles $\beta(j)$ pour $j \geq 0$ du ruban final forment le mot 01^y0 (suivi d'une infinité de 0) (« codage unaire » de y).

Calculs par les machines de Turing des fonctions récursives

- On peut montrer par induction suivant la déf^{n} de $\mathbf{R}$ que toute fonction générale récursive est calculable par machine de Turing avec les conventions du transp. précédent.

- La démonstration est fastidieuse mais pas difficile : il s'agit essentiellement de programmer en machine de Turing chacune des formes de construction des fonctions générales récursives (projections, constantes, successeur, composition, récursion primitive, μ-récursion).

- Les conventions faites, notamment le fait d'ignorer et de ne pas modifier $\beta(j)$ pour $j < 0$, permettent à l'induction dans la preuve de fonctionner.

Par exemple, pour la composition, on va utiliser cette propriété pour « sauvegarder » les $x_1, \ldots, x_k$ initiaux, ainsi que les valeurs de $g_j(\underline{x})$ calculées, lorsqu'on appelle chacune des fonctions $g_1, \ldots, g_\ell$ (à chaque fois, on les recopie $x_1, \ldots, x_k$ à droite des valeurs à ne pas toucher, et on appelle la machine calculant g_j sur ces valeurs recopiées).

Équivalence entre machines de Turing et fonctions récursives

- Toute fonction générale récursive $\mathbb{N}^k \dashrightarrow \mathbb{N}$ est calculable par machine de Turing (sous les conventions données) : transp. précédent.

- Réciproquement, toute fonction $\mathbb{N}^k \dashrightarrow \mathbb{N}$ calculable par machine de Turing sous ces conventions est générale récursive, car les fonctions récursives peuvent simuler les machines de Turing, calculer une configuration initiale convenable, et décoder la configuration finale (cf. transp. 52).

- Bref, $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ est calculable par machine de Turing ssi elle est générale récursive.

- De plus, cette équivalence est constructive : il existe des fonctions p.r. :
  - l'une prend en entrée le numéro e d'une fonction générale récursive (et l'arité k) et renvoie le code d'une machine de Turing qui calcule cette $\varphi_e^{(k)}$,
  - l'autre prend en entrée le code d'une machine de Turing qui calcule une fonction f, et son arité k, et renvoie un numéro e de f dans les fonctions générales récursives $f = \varphi_e^{(k)}$.

Machines de Turing : variations

On a choisi ici une notion de machine de Turing assez restreinte (1 bande, 2 symboles de bande). Il existe toutes sortes de variations :

- machines à plusieurs bandes (mais en nombre fini ; le programme choisit en fonction du symbole lu sur chaque bande, et écrit et déplace chaque tête indépendamment), voire à plusieurs têtes par bande, parfois avec des bandes en lecture seule (pour les entrées), ou en écriture seule (pour les sorties),
- autres symboles que 0 et 1 (mais en nombre fini),
- machine non-déterministe (plusieurs instructions possibles dans une configuration donnée ; la machine termine si au moins l'un des chemins d'exécution termine).

Du point de vue calculabilité, ces modifications ne rendent pas la machine plus puissante, et, sauf, cas dégénérés (p.ex., un seul symbole sur le ruban !) elles ne la rendent pas moins puissante non plus. Ceci confirme la robustesse du modèle de Church-Turing.

Pour la complexité, en revanche, c'est une autre affaire.

Machines de Turing : reprise de résultats déjà vus

- **Universalité :** pour un codage raisonnable, il existe une machine de Turing « universelle » qui prend en entrée sur sa bande le programme d'une autre machine de Turing M, et une configuration initiale C pour celle-ci, et qui simule l'exécution de M sur C (notamment, elle s'arrête ssi M s'arrête).

- **Forme normale :** la fonction $(n, M, C) \mapsto C^{(n)}$ qui à $n \in \mathbb{N}$ et une machine de Turing M et une configuration C de M associe la configuration après n étapes d'exécution, est p.r., et en particulier, calculable par une machine de Turing.

  $\Rightarrow$ En particulier, on peut tester algorithmiquement si une machine de Turing donnée, depuis une configuration initiale donnée, s'arrête *en moins de n étapes*.

- **Indécidabilité du problème de l'arrêt :** la fonction qui à (M, C) associe 1 si la machine de Turing s'arrête en partant de la configuration initiale C, et 0 sinon, n'est pas calculable.

  $\Rightarrow$ On ne peut pas tester algorithmiquement si une machine de Turing donnée, depuis une configuration initiale donnée, s'arrête « un jour ».

Indécidabilité du problème de l'arrêt (départ bande vierge)

On ne peut même pas tester algorithmiquement si une machine de Turing s'arrête à partir d'une bande vierge :

- **Indécidabilité du problème de l'arrêt :** la fonction qui à M associe 1 si la machine de Turing s'arrête en partant de la configuration vierge C_0 (c'est-à-dire celle où $\beta = 0$, état initial 1), et 0 sinon, n'est pas calculable.

Preuve : Supposons par l'absurde qu'on puisse tester algorithmiquement si une machine de Turing s'arrête à partir d'une configuration vierge. On va montrer qu'on peut tester si une machine de Turing M s'arrête à partir de C quelconque.

Données M et C, on peut algorithmiquement calculer une machine N qui « prépare » C à partir de la configuration vierge C_0, donc une machine M^* qui exécute successivement N puis M ($\leftarrow$ ceci est un théorème s-m-n).

Ainsi M^* (calculé algorithmiquement) termine sur C_0 ssi M termine sur C.

Donc tester la terminaison de M^* permettrait de tester celle de M sur C, ce qui n'est pas possible ($\leftarrow$ ceci est une preuve « par réduction »). □

Le castor affairé

- La fonction **castor affairé** associe à m le nombre maximal $B(m)$ d'étapes d'exécution d'une machine de Turing à $\leq m$ états qui termine (à partir de la configuration vierge C_0).
- La fonction B croît trop vite pour être calculable :

$$\forall f : \mathbb{N} \to \mathbb{N} \text{ calculable.} \quad \exists m \in \mathbb{N}. \quad (B(m) > f(m))$$

Preuve : supposons au contraire $\forall m \in \mathbb{N}.\ (B(m) \leq f(m))$ avec $f : \mathbb{N} \to \mathbb{N}$ calculable. Donnée une machine de Turing M, on peut alors algorithmiquement décider si M s'arrête à partir de C_0 :

- calculer $f(m)$ où m est le nombre d'états de M,
- exécuter M à partir de C_0 pendant $f(m)$ étapes (ce nombre est $\geq B(m)$ par hypothèse),
- si elle a terminé en temps imparti, M termine sur C_0, et on renvoie « oui » ; sinon, elle ne termine jamais par définition de $B(m)$, on renvoie « non ».

Ceci est impossible donc f n'est pas calculable. □

Le castor affairé (amélioration)

$B(m) =$ nombre maximal d'étapes d'exécution d'une machine de Turing à $\leq m$ états qui termine à partir d'une bande vierge.

- On peut faire mieux : B domine toute fonction calculable :

$$\forall f\colon \mathbb{N} \to \mathbb{N} \text{ calculable.} \quad \exists m_0 \in \mathbb{N}. \quad \forall m \geq m_0. \quad (B(m) > f(m))$$

Preuve : soit $f\colon \mathbb{N} \to \mathbb{N}$ calculable. Soit $\gamma(r) = A(2, r, r)$ (en fait, 2^r doit suffire ; noter γ croissante). Pour $r \in \mathbb{N}$, on considère la machine de Turing M_r qui

- prépare r, calcule $\gamma(r+1)$ puis $f(0) + f(1) + \cdots + f(\gamma(r+1)) + 1$,
- attend ce nombre-là d'étapes, et termine.

Le nombre d'états de M_r est une fonction p.r. $b(r)$ de r (même $b(r) = r + \text{const}$ convient). Pour $r \geq r_0$ on a $b(r) \leq \gamma(r)$. Soit $m_0 = \gamma(r_0)$. Si $m \geq m_0$, soit $r \geq r_0$ tel que $\gamma(r) \leq m \leq \gamma(r+1)$. Alors M_r calcule $\cdots + f(m) + \cdots + 1$, donc attend $> f(m)$ étapes. Donc $B(b(r)) > f(m)$. Mais $b(r) \leq \gamma(r) \leq m$ donc $B(m) > f(m)$. □

*

- Variations du castor affairé : nombre de symboles écrits sur la bande, $n \mapsto \max\{\varphi_e(e) : 0 \leq e \leq n \text{ et } \varphi_e(e)\downarrow\}$ (mêmes propriétés).

Terminologie calculable/décidable

- Une fonction partielle $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ est dite **calculable** (partielle) lorsqu'elle est (c'est équivalent) :
  - générale récursive,
  - calculable par machine de Turing,
  - à voir $\rightarrow$ représentable dans le λ-calcul.

- Une partie $A \subseteq \mathbb{N}^k$ est dite **décidable** lorsque sa fonction indicatrice $\mathbb{N}^k \to \mathbb{N}$

$$\mathbf{1}_A : n \mapsto \begin{cases} 1 & \text{si } n \in A \\ 0 & \text{si } n \notin A \end{cases}$$

est calculable (répondre « oui » ou « non » selon que $n \in A$ ou $n \notin A$).

- Une partie $A \subseteq \mathbb{N}^k$ est dite **semi-décidable** lorsque sa fonction partielle « semi-indicatrice » $\mathbb{N} \dashrightarrow \mathbb{N}$ (d'ensemble de définition A)

$$n \mapsto \begin{cases} 1 & \text{si } n \in A \\ \uparrow & \text{si } n \notin A \end{cases}$$

est calculable (répondre « oui » ou « ... » selon que $n \in A$ ou $n \notin A$).

Fluctuations terminologiques

- Synonymes de **calculable** pour une fonction partielle $\mathbb{N}^k \dashrightarrow \mathbb{N}$:
  - « semi-calculable » (réservant « calculable » pour les fonctions *totales*),
  - « (générale) récursive ».
- Synonymes de **décidable** pour une partie $\subseteq \mathbb{N}^k$:
  - « calculable »,
  - « récursive ».
- Synonymes de **semi-décidable** pour une partie $\subseteq \mathbb{N}^k$:
  - « semi-calculable »,
  - « calculablement énumérable »,
  - « récursivement énumérable ».

(La raison du mot « énumérable » sera expliquée après.)

Décidable = semi-décidable de complémentaire semi-décidable

- Si $A \subseteq \mathbb{N}^k$ est décidable, alors son complémentaire $\complement A := \mathbb{N}^k \setminus A$ l'est aussi.

Preuve : échanger 0 et 1 dans la réponse. □

- Si A est décidable, alors A est semi-décidable.

Preuve : si réponse 0, faire une boucle infinie. □

- Donc : si A est décidable, alors A et $\complement A$ sont semi-décidables.

- La réciproque est également valable : si A et $\complement A$ sont semi-décidables alors A est décidable.

Idée : lancer « en parallèle » un algorithme qui semi-décide A et un qui semi-décide $\complement A$; l'un des deux finira par donner la réponse voulue.

Mais que signifie « lancer en parallèle » ici ?

Lancement en parallèle

On suppose que :

- $\varphi_{e_1}(\underline{x}) \downarrow$ ssi $\underline{x} \in A$
- $\varphi_{e_2}(\underline{x}) \downarrow$ ssi $\underline{x} \notin A$

Comment décider si $\underline{x} \in A$ en terminant à coup sûr ?

Grâce au th. de la forme normale (transp. 37) : il y a un prédicat T p.r. tel que

- $\varphi_{e_1}(\underline{x}) \downarrow$ ssi $\exists n \in \mathbb{N}.\, T(n, e_1, \langle\langle \underline{x} \rangle\rangle)$
- $\varphi_{e_2}(\underline{x}) \downarrow$ ssi $\exists n \in \mathbb{N}.\, T(n, e_2, \langle\langle \underline{x} \rangle\rangle)$

On a alors $\exists n \in \mathbb{N}.\, (T(n, e_1, \langle\langle \underline{x} \rangle\rangle)$ ou $T(n, e_2, \langle\langle \underline{x} \rangle\rangle))$ à coup sûr.

Algorithme (terminant à coup sûr) :

- parcourir $n = 0, 1, 2, 3, 4, \ldots$ (boucle non bornée),
- pour chacun, tester si $T(n, e_1, \langle\langle \underline{x} \rangle\rangle)$ et si $T(n, e_2, \langle\langle \underline{x} \rangle\rangle)$,
- si le premier vaut, renvoyer « oui, $\underline{x} \in A$ », si le second vaut, renvoyer « non, $\underline{x} \notin A$ » (sinon, continuer la boucle).

Lancement en parallèle (variante machines de Turing)

On suppose que :

- la machine M_1 s'arrête sur $\underline{x}$ ssi $\underline{x} \in A$
- la machine M_2 s'arrête sur $\underline{x}$ ssi $\underline{x} \notin A$

Comment décider si $\underline{x} \in A$ en s'arrêtant à coup sûr ?

On va simuler M_1 et M_2 pour n étapes jusqu'à ce que l'une d'elles s'arrête.

Algorithme (terminant à coup sûr) :

- parcourir $n = 0, 1, 2, 3, 4, \ldots$ (boucle non bornée),
- pour chacun, tester si l'exécution de M_1 s'arrête sur $\underline{x}$ en $\leq n$ étapes et si l'exécution de M_2 s'arrête sur $\underline{x}$ en $\leq n$ étapes,
- si le premier vaut, renvoyer « oui, $\underline{x} \in A$ », si le second vaut, renvoyer « non, $\underline{x} \notin A$ » (sinon, continuer la boucle).

C'est exactement la même chose que dans le transp. précédent, avec un nombre d'étapes d'exécution n au lieu d'un arbre de calcul (détail sans importance).

Problème de l'arrêt

Le **problème de l'arrêt** est :

$$\mathscr{H} := \{(e,x) \in \mathbb{N}^2 : \varphi_e(x) \downarrow\}$$

- Il n'est pas décidable (transp. 45).
- Il est semi-décidable (par universalité : donné (e,x), on peut exécuter $\varphi_e(x)$, et, s'il termine, renvoyer « oui »).
- Donc $\complement\mathscr{H}$ n'est pas semi-décidable.

- Toutes sortes de variantes possibles, p.ex. :
  - $\{e \in \mathbb{N} : \varphi_e(e) \downarrow\}$ n'est pas décidable (preuve dans transp. 46)
  - $\{e \in \mathbb{N} : \varphi_e(0) \downarrow\}$ n'est pas décidable (théorème s-m-n : $\varphi_e(x) = \varphi_{s(e,x)}(0)$ avec s p.r. ; cf. transp. 58)

Image d'un ensemble décidable

- Si $A \subseteq \mathbb{N}$ est décidable et $f : \mathbb{N} \to \mathbb{N}$ (totale) calculable, alors l'image
$$f(A) := \{f(i) : i \in A\}$$
est semi-décidable.

Preuve : donné $m \in \mathbb{N}$, pour semi-décider si $m \in f(A)$, parcourir $i = 0, 1, 2, 3 \ldots$, et pour chacun, décider si $i \in A$ et, si oui, calculer $f(i)$ et comparer à m. Si $i \in A$ et $f(i) = m$, renvoyer « oui » ; sinon, continuer la boucle. □

- Réciproquement, si $B \subseteq \mathbb{N}$ est semi-décidable, il existe $A \subseteq \mathbb{N}$ décidable et $f : \mathbb{N} \to \mathbb{N}$ (totale) calculable tels que $B = f(A)$.

Preuve : soit e tel que $B = \{m : \varphi_e(m) \downarrow\}$; soit A l'ensemble des $\langle n, m \rangle$ tels que $T(n, e, \langle\langle m \rangle\rangle)$: alors A est décidable et son image par $\langle n, m \rangle \mapsto m$ est B. □

Redite : soit M une machine de Turing qui s'arrête sur m ssi $m \in B$; soit A l'ensemble des $\langle n, m \rangle$ tels que M s'arrête sur m en $\leq n$ étapes : alors A est décidable et son image par $\langle n, m \rangle \mapsto m$ est B. □

- Variante : $B \subseteq \mathbb{N}$ *non vide* est semi-décidable ssi il existe $f : \mathbb{N} \to \mathbb{N}$ totale calculable telle que $f(\mathbb{N}) = B$. D'où le terme « calculablement énumérable ».

Stabilités par opérations booléennes

Les ensembles **décidables** sont stables par :

- réunions finies,
- intersections finies,
- complémentaire,
- mais pas par projection $\mathbb{N}^k \to \mathbb{N}^{k'}$ (où $k' \leq k$).
  (Le problème de l'arrêt est une projection d'un ensemble décidable, cf. transp. précédent.)

Les ensembles **semi-décidables** sont stables par :

- réunions finies (par lancement en parallèle !),
- intersections finies,
- projection $\mathbb{N}^k \to \mathbb{N}^{k'}$ (où $k' \leq k$),
  (Les ensembles semi-décidables sont projections d'ensembles décidables donc sont eux-mêmes stables par projections, cf. transp. précédent et idées proches.)
- mais pas par complémentaire.

Le théorème de Rice : énoncé

Soit $\mathbf{R}^{(1)}$ l'ensemble des fonctions partielles $\mathbb{N} \dashrightarrow \mathbb{N}$ calculables (= générales récursives), et $\Phi : e \mapsto \varphi_e^{(1)}$ qui définit une surjection $\mathbb{N} \to \mathbf{R}^{(1)}$.

Si e est l'« intention » (l'algorithme, le programme), alors $\Phi(e)$ est l'« extension » (la fonction, i.e., son graphe) définie par e.

- **Théorème** (Rice) : si $F \subseteq \mathbf{R}^{(1)}$ est un ensemble de fonctions partielles tel que $\Phi^{-1}(F) := \{e \in \mathbb{N} : \varphi_e^{(1)} \in F\}$ est *décidable*, alors $F = \varnothing$ ou $F = \mathbf{R}^{(1)}$.

Moralité : aucune propriété non-triviale de la fonction $\varphi_e^{(1)}$ calculée par un programme n'est décidable en regardant le programme e.

Exemples :

- $\{e \in \mathbb{N} : \varphi_e^{(1)}(0)\downarrow\}$ n'est pas décidable ($\Rightarrow$ Rice généralise l'indécidabilité du pb. de l'arrêt).
- $\{e \in \mathbb{N} : \varphi_e^{(1)}$ totale$\}$ n'est pas décidable.
- $\{e \in \mathbb{N} : \forall n.\, (\varphi_e^{(1)}(n)\downarrow \Rightarrow \varphi_e^{(1)}(n) = 0)\}$ n'est pas décidable.

Le théorème de Rice : preuve par théorème de récursion

$\mathbf{R}^{(1)} = \{f : \mathbb{N} \dashrightarrow \mathbb{N} : f \text{ calculable}\}$

- **Théorème** (Rice) : si $F \subseteq \mathbf{R}^{(1)}$ est un ensemble de fonctions partielles tel que $\Phi^{-1}(F) := \{e \in \mathbb{N} : \varphi_e^{(1)} \in F\}$ est *décidable*, alors $F = \varnothing$ ou $F = \mathbf{R}^{(1)}$.

La preuve est très analogue à celle de l'indécidabilité du problème de l'arrêt.

Preuve : Supposons par l'absurde $\Phi^{-1}(F)$ décidable avec $F \neq \varnothing$ et $F \neq \mathbf{R}^{(1)}$. Soient $f \in F$ et $g \notin F$. Soit

$$h(e,x) := \begin{cases} f(x) & \text{si } e \notin \Phi^{-1}(F) \\ g(x) & \text{si } e \in \Phi^{-1}(F) \end{cases}$$

Alors $h : \mathbb{N}^2 \dashrightarrow \mathbb{N}$ est calculable par hypothèse (on peut décider si $e \in \Phi^{-1}(F)$). Par le théorème de récursion de Kleene (transp. 40), il existe e tel que

$$\varphi_e^{(1)}(x) = h(e,x)$$

Si $e \in \Phi^{-1}(F)$ alors $h(e,x) = g(x)$ pour tout x, donc $\Phi(e) = g$ donc $e \notin \Phi^{-1}(F)$, une contradiction. Si $e \notin \Phi^{-1}(F)$ alors $h(e,x) = f(x)$ pour tout x, donc $\Phi(e) = f$ donc $e \in \Phi^{-1}(F)$, une contradiction. □

Réductions : introduction

- Situation typique : on veut montrer qu'une question D (« problème de décision », souvent déjà semi-décidable) est indécidable. Ceci se fait typiquement en réduisant le problème de l'arrêt à D, c'est-à-dire :

« Supposons par l'absurde que D soit décidable, c'est-à-dire que j'ai un algorithme qui répond à la question D (comprendre : "$n \in D$?").
Je montre qu'en utilisant cet algorithme je peux résoudre le problème de l'arrêt.
Ceci est une contradiction (car le problème de l'arrêt est indécidable), donc D est indécidable. »

- Les notions de réduction formalisent cet argument : intuitivement,

« A se réduit à B »
signifie
« si B est décidable alors A est décidable »
(mais constructivement)

Le théorème de Rice : preuve par réduction (1/2)

$\mathbf{R}^{(1)} = \{f : \mathbb{N} \dashrightarrow \mathbb{N} : f \text{ calculable}\}$

- **Théorème** (Rice) : si $F \subseteq \mathbf{R}^{(1)}$ est tel que $F \neq \varnothing$ et $F \neq \mathbf{R}^{(1)}$, alors $\Phi^{-1}(F) := \{e \in \mathbb{N} : \varphi_e^{(1)} \in F\}$ *n'est pas décidable*.

Preuve : Soit $F \subseteq \mathbf{R}^{(1)}$ avec $F \neq \varnothing$ et $F \neq \mathbf{R}^{(1)}$. Quitte à remplacer F par $\complement F$, o.p.s. $\uparrow \notin F$ où $\uparrow$ est la fonction nulle part définie. Soit $f \in F$ où $f = \varphi_a^{(1)}$.
Pour $(e, x) \in \mathbb{N}^2$, considérons l'algorithme suivant, prenant en entrée $m \in \mathbb{N}$:

- simuler $\varphi_e^{(1)}(x)$ avec la machine universelle, puis, si l'exécution termine,
- calculer $f(m) = \varphi_a^{(1)}(m)$ et (si l'exécution termine) renvoyer sa valeur.

Soit $b(e, x)$ le code de l'algorithme qu'on vient de décrire :

$$\varphi_{b(e,x)}^{(1)} = f \text{ si } \varphi_e^{(1)}(x) \downarrow \quad \text{et} \quad \varphi_{b(e,x)}^{(1)} = \uparrow \text{ si } \varphi_e^{(1)}(x) \uparrow$$

notamment $\varphi_{b(e,x)}^{(1)} \in F$ ssi $\varphi_e^{(1)}(x) \downarrow$ ($\leftarrow$ c'est là la réduction). .../...

Le théorème de Rice : preuve par réduction (2/2)

$\mathbf{R}^{(1)} = \{f : \mathbb{N} \dashrightarrow \mathbb{N} : f \text{ calculable}\}$; on a supposé $F \subseteq \mathbf{R}^{(1)}$ avec $\uparrow \notin F$ et $f \in F$

On a construit (transp. précédent) un $b(e,x)$, avec $b : \mathbb{N}^2 \to \mathbb{N}$ calculable (même p.r.) tel que $\varphi^{(1)}_{b(e,x)} \in F$ ssi $\varphi^{(1)}_e(x)\downarrow$, c'est-à-dire

$$b(e,x) \in \Phi^{-1}(F) \iff (e,x) \in \mathscr{H}$$

où $\mathscr{H} := \{(e,x) \in \mathbb{N}^2 : \varphi_e(x)\downarrow\}$ est le problème de l'arrêt.

Si $\Phi^{-1}(F)$ était décidable, alors $\mathscr{H}$ le serait aussi, par l'algorithme :

- données e, x, calculer $b(e,x)$, décider si $b(e,x) \in \Phi^{-1}(F)$,
- si oui, répondre « oui », sinon répondre « non ».

Or $\mathscr{H}$ n'est pas décidable, donc $\Phi^{-1}(F)$ non plus. □

On dit qu'on a réduit le problème de l'arrêt à $\Phi^{-1}(F)$ (via la fonction b).

Réduction « many-to-one »

Définition : Si $A, B \subseteq \mathbb{N}$, on note $A \leq_{\mathrm{m}} B$ lorsqu'il existe $\rho\colon \mathbb{N} \to \mathbb{N}$ *calculable totale* telle que

$$\rho(m) \in B \iff m \in A$$

(c'est-à-dire $A = \rho^{-1}(B)$).

Intuitivement : si j'ai un gadget qui répond à la question "$n \in B$?", je peux répondre à la question "$m \in A$?" en transformant m en $\rho(m) =: n$ et en utilisant le gadget (une seule fois, à la fin).

Clairement, si $A \leq_{\mathrm{m}} B$ avec B décidable (resp. semi-décidable), alors A est décidable (resp. semi-décidable).

Notamment, si $\mathscr{H} \leq_{\mathrm{m}} D$ alors D *n'est pas* décidable.

La relation $\leq_{\mathrm{m}}$ est réflexive et transitive (c'est un « préordre ») ; la relation $\equiv_{\mathrm{m}}$ définie par $A \equiv_{\mathrm{m}} B$ ssi $A \leq_{\mathrm{m}} B$ et $B \leq_{\mathrm{m}} A$ est une relation d'équivalence, les classes pour laquelle s'appellent « degrés many-to-one » et sont partiellement ordonnés par $\leq_{\mathrm{m}}$.

Réduction de Turing : présentation informelle

Informellement : Si $A, B \subseteq \mathbb{N}$, on note $A \leq_{\mathrm{T}} B$ s'il existe un algorithme qui

- prend en entrée $m \in \mathbb{N}$,
- peut à tout moment demander à savoir si $n \in B$ (« interroger l'oracle »),
- termine en temps fini,
- et renvoie « oui » si $m \in A$, et « non » si $m \notin A$.

Intuitivement : à la différence de la réduction many-to-one où on ne peut poser la question "$n \in B$?" que sur une seule valeur $\rho(n)$ à la fin du calcul, ici on peut interroger l'oracle de façon libre et illimitée (mais finie !) au cours de l'algorithme.

La relation $A \leq_{\mathrm{T}} B$ est beaucoup plus faible que $A \leq_{\mathrm{m}} B$.

Par exemple, $(\complement B) \leq_{\mathrm{T}} B$ pour tout $B \subseteq \mathbb{N}$ (savoir décider "$n \in B$?" permet évidemment de décider "$n \notin B$?"), alors que $(\complement \mathscr{H}) \not\leq_{\mathrm{m}} \mathscr{H}$ car $\complement \mathscr{H}$ n'est pas semi-décidable.

Mais comment formaliser cette « interrogation » ?

Réduction de Turing : formalisation(s) possible(s)

Comment définir $A \leq_{\mathrm{T}} B$ pour $A, B \subseteq \mathbb{N}$? (I.e., « A est calculable en utilisant B ».)

Formalisation 1 : la fonction indicatrice $\mathbf{1}_A$ de A appartient à la plus petite classe de fonctions qui contient les projections, les constantes, la fonction successeur et la fonction indicatrice $\mathbf{1}_B$ de B et stable par composition, récursion primitive et opérateur μ.

Formalisation 2 : il existe une fonction calculable qui prend en entrée $m \in \mathbb{N}$ et une liste $\langle\langle\langle n_0, \mathbf{1}_B(n_0)\rangle, \ldots, \langle n_k, \mathbf{1}_B(n_k)\rangle\rangle\rangle$ de réponses à des questions "$n \in B$?", et qui (si ces réponses cont correctes !) termine et renvoie

- soit une réponse finale à la question "$m \in A$?" (disons encodée comme $\langle 0, \mathbf{1}_A(m)\rangle$),
- soit une nouvelle interrogation "$n \in B$?" (disons encodée comme $\langle 1, n\rangle$),

de sorte que si on commence par $k = 0$ et qu'on ajoute à chaque fois la réponse correcte $\langle n_{k+1}, \mathbf{1}_B(n_{k+1})\rangle$ à l'interrogation $\langle 1, n_{k+1}\rangle$ posée, alors la fonction finit par produite la réponse finale correcte ($\langle 0, \mathbf{1}_A(m)\rangle$).

Réduction de Turing : quelques propriétés

Clairement, si $A \leq_{\mathrm{T}} B$ avec B décidable, alors A est décidable.
(Ceci ne vaut pas pour « semi-décidable ».)

Notamment, si $\mathscr{H} \leq_{\mathrm{T}} D$ alors D *n'est pas* décidable.

La relation $\leq_{\mathrm{T}}$ est réflexive et transitive (c'est un « préordre ») ; la relation $\equiv_{\mathrm{T}}$ définie par $A \equiv_{\mathrm{T}} B$ ssi $A \leq_{\mathrm{T}} B$ et $B \leq_{\mathrm{T}} A$ est une relation d'équivalence, les classes pour laquelle s'appellent « degrés de Turing » et sont partiellement ordonnés par $\leq_{\mathrm{T}}$.

Comme $A \leq_{\mathrm{m}} B$ implique $A \leq_{\mathrm{T}} B$, chaque degré de Turing est une réunion de degrés many-to-one (la relation d'équivalence $\equiv_{\mathrm{T}}$ est plus grossière que $\equiv_{\mathrm{m}}$).

Les parties décidables de $\mathbb{N}$ forment le plus petit degré de Turing, souvent noté $\mathbf{0}$. Le degré de Turing de $\mathscr{H}$ est noté $\mathbf{0}'$. (Il existe des ensembles de degré strictement compris entre $\mathbf{0}$ et $\mathbf{0}'$, même des ensembles semi-décidables, mais il semble qu'aucun n'apparaît naturellement.)

Le λ-calcul : aperçu

Le λ-**calcul non typé** manipule des expressions du type

$$\lambda x.\lambda y.\lambda z.((xz)(yz))$$
$$\lambda f.\lambda x.f(f(f(f(f(fx)))))$$
$$(\lambda x.(xx))(\lambda x.(xx))$$

Ces expressions s'appelleront des **termes** du λ-calcul.

Il faut comprendre intuitivement qu'un terme représente une sorte de fonction qui prend une autre telle fonction en entrée et renvoie une autre telle fonction.

Deux constructions fondamentales :

- **application** : (PQ) : appliquer la fonction P à la fonction Q ;
- **abstraction** : $\lambda v.E$: créer la fonction qui prend un argument et le remplace pour v dans l'expression E (en gros $v \mapsto E$).

Le λ-calcul : termes

- Un **terme** du λ-calcul est (inductivement) :
  - une **variable** (a, b, c... en nombre illimité),
  - une **application** (PQ) où P et Q sont deux termes,
  - une **abstraction** $\lambda v.E$ où v est une variable et E un terme ; on dira que la variable v est **liée** dans E par ce λ.
- Conventions d'écriture :
  - l'application n'est pas associative : on parenthèse implicitement vers la gauche : « xyz » signifie « $((xy)z)$ » ;
  - abréviation de plusieurs λ : on note « $\lambda uv.E$ » pour « $\lambda u.\lambda v.E$ » ;
  - l'abstraction est moins prioritaire que l'application : « $\lambda x.xy$ » signifie $\lambda x.(xy)$ pas $(\lambda x.x)y$.
- Une variable non liée est dite **libre** : $(\lambda x.x)x$ (le dernier x est libre).
- Un terme sans variable libre est dit **clos**.
- Les variables liées sont muettes : $\lambda x.x \equiv \lambda y.y$, comprendre $\lambda\bullet.\bullet$.

Le λ-calcul : variables liées

On appelle α-**conversion** le renommage des variables liées : ces termes sont considérés comme équivalents.

- $\lambda x.x \equiv \lambda y.y$ et $\lambda xyz.((xz)(yz)) \equiv \lambda uvw.((uw)(vw))$
- Attention à ne pas capturer de variable libre : $\lambda y.xy \not\equiv \lambda x.xx$.
- En cas de synonymie, la variable est liée par le λ le plus intérieur pour ce nom ($\cong$ portée lexicale) : $\lambda x.\lambda x.x \equiv \lambda x.\lambda v.v \not\equiv \lambda u.\lambda x.u$.
- Mieux vaut ne pas penser aux termes typographiquement, mais à chaque variable liée comme un *pointeur vers la λ-abstraction qui la lie* :

$$\lambda x.(\lambda y.y(\lambda z.z))(\lambda z.xz) \equiv \lambda\bullet.(\lambda\bullet.\bullet(\lambda\bullet.\bullet))(\lambda\bullet.\bullet\bullet)$$

- Autre convention possible : **indices de De Bruijn** : remplacer les variables liées par le numéro du λ qui la lie, en comptant du plus intérieur (1) vers le plus extérieur :

$$\lambda x.(\lambda y.y(\lambda z.z))(\lambda z.xz) \equiv \lambda.(\lambda.1(\lambda.1))(\lambda.21)$$

deux termes sont α-équivalents ssi leur écriture avec indice de De Bruijn est identique.

Le λ-calcul : β-réduction

On travaille désormais sur des termes à α-équivalence près.

- Un **redex** (« reducible expression ») est un terme de la forme $(\lambda v.E)T$.
Son **réduit** est le terme $E[v \backslash T]$ obtenu par remplacement de T pour v dans E, en évitant tout conflit de variables.

Exemples :

- $(\lambda x.xx)y \to yy$
- $(\lambda x.xx)(\lambda x.xx) \to (\lambda x.xx)(\lambda x.xx)$ (est son propre réduit)
- $(\lambda xy.x)z \to \lambda y.z$ (car $\lambda xy.x$ abrège $\lambda x.\lambda y.x$)
- $(\lambda xy.x)y \to \lambda y_1.y$ (attention au conflit de variable !)
- $(\lambda x.\lambda x.x)y \to \lambda x.x$ (car $\lambda x.\lambda x.x \equiv \lambda z.\lambda x.x$: le λ extérieur ne lie rien)

- Un terme n'ayant pas de redex en sous-expression est dit en **forme (β-)normale**. Ex. : $\lambda xyz.((xz)(yz))$.
- On appelle **β-réduction** le remplacement en sous-expression d'un redex par son réduit. Ex. : $\lambda x.(\lambda y.y(\lambda z.z))(\lambda z.xz) \to \lambda x.(\lambda z.xz)(\lambda z.z)$.

Le λ-calcul : normalisation par β-réductions

On note :

- $T \to T'$ (ou $T \to_\beta T'$) si T' s'obtient par β-réduction d'un redex de T.
- $T \twoheadrightarrow T'$ (ou $T \twoheadrightarrow_\beta T'$) si T' s'obtient par une suite finie de β-réductions ($T = T_0 \to \cdots \to T_n = T'$, y compris $n = 0$ soit $T' = T$).
- T est **faiblement normalisable** lorsque $T \twoheadrightarrow T'$ avec T' en forme normale (une certaine suite de β-réductions termine).
- T est **fortement normalisable** lorsque toute suite de β-réductions termine (sur un terme en forme normale).

Exemples :

- $(\lambda x.xx)(\lambda x.xx)$ n'est pas faiblement normalisable (la β-réduction boucle).
- $(\lambda uz.z)((\lambda x.xx)(\lambda x.xx))$ n'est pas fortement normalisable mais il est faiblement normalisable $\to \lambda z.z$.
- $(\lambda uz.u)((\lambda t.t)(\lambda x.xx))$ est fortement normalisable $\twoheadrightarrow \lambda zx.xx$.

Le λ-calcul : confluence et choix d'un redex

- **Théorème** (Church-Rosser) : si $T \twoheadrightarrow T'_1$ et $T \twoheadrightarrow T'_2$ alors il existe T'' tel que $T'_1 \twoheadrightarrow T''$ et $T'_2 \twoheadrightarrow T''$.

En particulier, si T'_1, T'_2 sont en forme normale, alors $T'_1 \equiv T'_2$ (unicité de la normalisation).

Pour éviter ce théorème, on va faire un choix simple de redex à réduire :

- On appelle **redex extérieur gauche** d'un λ-terme le redex dont le λ est le plus à gauche. Exemples : $\lambda x.x((\lambda y.y)x)$; $\lambda x.(\lambda y.(\lambda z.z)y)x$.

- On écrira $T \to_{\text{lft}} T'$ lorsque T' s'obtient par β-réduction du redex extérieur gauche, et $T \twoheadrightarrow_{\text{lft}} T'$ pour une suite de telles réductions.

On peut montrer (mais on évitera d'utiliser) :

- **Théorème** (Curry &al) : si $T \twoheadrightarrow T'$ avec T' en forme normale, alors $T \twoheadrightarrow_{\text{lft}} T'$ (i.e., la réduction ext. gauche normalise les termes faiblement normalisables).

Réduction extérieure gauche : exemples

Divers noms utilisés : « réduction en ordre normal », « réduction gauche », etc.

On a noté $T \twoheadrightarrow_{\text{lft}} T'$ lorsque T' s'obtient par une succession de β-réductions à chaque fois du redex dont le λ est le plus à gauche.

Exemples :

- $(\lambda x.xx)(\lambda x.xx) \to_{\text{lft}} (\lambda x.xx)(\lambda x.xx) \to_{\text{lft}} \cdots$ (boucle)
- $(\lambda uz.z)((\lambda x.xx)(\lambda x.xx)) = (\lambda u.\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_{\text{lft}} \lambda z.z$
- $(\lambda uz.u)((\lambda t.t)(\lambda x.xx)) = (\lambda u.\lambda z.u)((\lambda t.t)(\lambda x.xx)) \to_{\text{lft}}$ $\lambda z.((\lambda t.t)(\lambda x.xx)) \to_{\text{lft}} \lambda z.\lambda x.xx = \lambda zx.xx$

Intérêt :

- cette stratégie de réduction est déterministe,
- (Curry &al :) si (« terme faiblement normalisant ») une réduction quelconque termine sur une forme normale, alors $\twoheadrightarrow_{\text{lft}}$ le fait.

Simulation du λ-calcul par les fonctions récursives

- On peut coder un terme du λ-calcul sous forme d'entiers naturels.
- La fonction $T \mapsto 1$ qui à un terme T associe 0 si T est en forme normale et 1 si non, **est p.r.**
- La fonction $T \mapsto T'$ qui à un terme T associe sa réduction extérieure gauche **est p.r.**
- Conséquence : la fonction $(n, T) \mapsto T^{(n)}$ qui à $n \in \mathbb{N}$ et un terme T associe le terme obtenu après n réductions extérieures gauches **est p.r.**
- La fonction qui à T associe la forme normale (et/ou le nombre d'étapes d'exécution) si la réduction extérieure gauche termine, et $\uparrow$ (non définie) si elle ne termine pas, est **générale récursive**.

Moralité : les fonctions récursives peuvent simuler la réduction extérieure gauche du λ-calcul (ou n'importe quelle autre réduction, mais on se concentre sur celle-ci).

Entiers de Church

On définit les termes en forme normale $\overline{n} := \lambda f x.f^{\circ n}(x)$ pour $n \in \mathbb{N}$, c-à-d :

- $\overline{0} := \lambda f x.x$
- $\overline{1} := \lambda f x.fx$
- $\overline{2} := \lambda f x.f(fx)$
- $\overline{3} := \lambda f x.f(f(fx))$, etc.

Intuitivement, $\overline{n}$ prend une fonction f et renvoie sa n-ième itérée.

- Posons $A := \lambda m f x.f(mfx) = \lambda m.\lambda f.\lambda x.f(mfx)$

Alors

$$\begin{aligned} A\overline{n} &= (\lambda m.\lambda f.\lambda x.f(mfx))(\lambda g.\lambda y.g^{\circ n}(y)) \\ &\to_{\text{lft}} \lambda f.\lambda x.f(((\lambda g.\lambda y.g^{\circ n}(y))fx) \\ &\to_{\text{lft}}\to_{\text{lft}} \lambda f.\lambda x.f(f^{\circ n}(x)) = \lambda f.\lambda x.f^{\circ(n+1)}(x) = \overline{n+1} \end{aligned}$$

Calculs dans le λ-calcul : une convention

On dira qu'une fonction $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ est **représentable par un λ-terme** lorsqu'il existe un terme clos t tel que, pour tous $x_1, \ldots, x_k \in \mathbb{N}$:

- si $f(x_1, \ldots, x_k)\downarrow = y$ alors $t\overline{x_1} \cdots \overline{x_k} \twoheadrightarrow_{\mathsf{lft}} \overline{y}$,
- si $f(x_1, \ldots, x_k)\uparrow$ alors $t\overline{x_1} \cdots \overline{x_k} \rightarrow_{\mathsf{lft}} \cdots$ ne termine pas,

où $\overline{z}$ désigne l'entier de Church associé à $z \in \mathbb{N}$.

Exemples :

- $\lambda mfx.f(mfx)$ représente $m \mapsto m + 1$ (transp. précédent),
- $\lambda mnfx.nf(mfx)$ représente $(m, n) \mapsto m + n$,
- $\lambda mnf.n(mf)$ représente $(m, n) \mapsto mn$ (itérer n fois l'itérée m-ième),
- $\lambda mn.nm$ représente $(m, n) \mapsto m^n$ (itérer n fois l'itération m-ième).
- $\lambda mnp.p(\lambda y.n)m$ représente $(m, n, p) \mapsto \begin{cases} m & \text{si } p = 0 \\ n & \text{si } p \geq 1 \end{cases}$

  (itérer p fois « remplacer par n »).

Représentation des fonctions p.r. : cas faciles

(Cf. transp. 16.)

Fonction p.r. facilement représentables par un λ-terme :

- $\lambda x_1 \cdots x_k.x_i$ représente $(x_1, \ldots, x_k) \mapsto x_i$;
- $\lambda x_1 \cdots x_k.\overline{c}$ représente $(x_1, \ldots, x_k) \mapsto c$;
- $A := \lambda mfx.f(mfx)$ représente $x \mapsto x + 1$;
- si $v_1, \ldots, v_\ell$ représentent $g_1, \ldots, g_\ell$ et w représente h, alors $\lambda x_1 \cdots x_k.w(v_1 x_1 \cdots x_k) \cdots (v_\ell x_1 \cdots x_k)$ représente $(x_1, \ldots, x_k) \mapsto h(g_1(x_1, \ldots, x_k), \ldots, g_\ell(x_1, \ldots, x_k))$;
- si v représente g et w représente h, alors

$$\lambda x_1 \cdots x_k z.z(wx_1 \cdots x_k)(vx_1 \cdots x_k)$$

représente f définie par la récursion primitive

$$f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k)$$
$$f(x_1, \ldots, x_k, z+1) = h(x_1, \ldots, x_k, f(x_1, \ldots, x_k, z))$$

mais on veut $f(x_1, \ldots, x_k, z+1) = h(x_1, \ldots, x_k, f(x_1, \ldots, x_k, z), z)$...?

Représentation des couples d'entiers

(Oublions $x_1, \ldots, x_k$ pour ne pas alourdir les notations.)

Comment passer de

$$\begin{cases} f(0) = g \\ f(z+1) = h(f(z)) \end{cases} \quad \text{à} \quad \begin{cases} f(0) = g \\ f(z+1) = h(f(z), z) \end{cases} \quad ?$$

On voudrait définir

$$\tilde{f}(z) = (f(z), z) \quad \text{soit} \quad \begin{cases} \tilde{f}(0) = (g, 0) \\ \tilde{f}(z+1) = \tilde{h}(\tilde{f}(z)) \end{cases} \quad \text{où} \quad \tilde{h}(y, z) = (h(y, z), z+1)$$

On va définir (temporairement ?)

$$\begin{aligned} \overline{m,n} &:= \lambda f g x.f^{\circ m}(g^{\circ n}(x)) \quad \text{si } m, n \in \mathbb{N} \\ \Pi &:= \lambda m n f g x.(mf)(ngx) \quad \text{donc } \Pi \overline{m}\,\overline{n} \twoheadrightarrow_{\mathsf{lft}} \overline{m,n} \\ \pi_1 &:= \lambda p f x.pf(\lambda z.z)x \quad \text{donc } \pi_1 \overline{m,n} \twoheadrightarrow_{\mathsf{lft}} \overline{m} \\ \pi_2 &:= \lambda p g x.p(\lambda z.z)gx \quad \text{donc } \pi_2 \overline{m,n} \twoheadrightarrow_{\mathsf{lft}} \overline{n} \end{aligned}$$

Représentation de la récursion primitive générale

Maintenant qu'on a une représentation des couples d'entiers naturels dans le λ-calcul donnée par Π (formation de paires) et π_1, π_2 (projections).

- Si v représente $g\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ et w représente $h\colon \mathbb{N}^{k+2} \dashrightarrow \mathbb{N}$, alors $f\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est représentée par

$$\lambda x_1 \cdots x_k z.\pi_1(z(\lambda p.\Pi(wx_1 \cdots x_k(\pi_1 p)(\pi_2 p))A(\pi_2 p))(\Pi(vx_1 \cdots x_k)\overline{0}))$$

où

$$f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k)$$
$$f(x_1, \ldots, x_k, z+1) = h(x_1, \ldots, x_k, f(x_1, \ldots, x_k, z), z)$$

(toujours avec $A := \lambda mfx.f(mfx)$).

D'autres encodages des paires sont possibles et possiblement plus simples, p.ex., $\Pi := \lambda rsa.ars$ et $\pi_1 := \lambda p.p(\lambda rs.r)$ et $\pi_2 := \lambda p.p(\lambda rs.s)$ (fonctionnent sur plus que les entiers de Church).

Bref, (au moins) les fonctions p.r. sont représentables par λ-termes.

Le combinateur Y de Curry

- Pour représenter toutes les fonctions récursives, on va implémenter les appels récursifs dans le λ-calcul.

- Pour ça, on va utiliser la même idée que le théorème de récursion de Kleene (transp. 25).

Posons

$$\mathsf{Y} := \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$$

Idée :

$$\begin{aligned}\mathsf{Y} &:= \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))\\ &\to \lambda f.f((\lambda x.f(xx))(\lambda x.f(xx)))\\ &\to \lambda f.f(f((\lambda x.f(xx))(\lambda x.f(xx)))) \to \cdots\end{aligned}$$

- Le terme (non normalisable !) Y "**recherche**" un point fixe de son argument.

- Permet d'implémenter les appels récursifs (transp. suivant).

Récursion avec le combinateur Y

- On veut implémenter une définition par appels récursifs dans le λ-calcul, `let rec` $h = (\ldots h \ldots)$, disons `let rec` $h = E$ où $E = (\ldots h \ldots)$ est un terme faisant intervenir h.

L'idée est comme dans le transp. 42.

- On considère le terme :

$$\begin{aligned} \mathsf{Y}(\lambda h.E) &= (\lambda f.((\lambda x.f(xx))(\lambda x.f(xx))))(\lambda h.E) \\ &\to (\lambda x.(\lambda h.E)(xx))(\lambda x.(\lambda h.E)(xx)) =: h_{\text{fix}} \\ &\to (\lambda h.E)((\lambda x.(\lambda h.E)(xx))(\lambda x.(\lambda h.E)(xx))) = (\lambda h.E)(h_{\text{fix}}) \\ &\to E[h \backslash h_{\text{fix}}] \text{ (substitution de } h_{\text{fix}} \text{ pour } h \text{ dans } E) \end{aligned}$$

Donc h_{fix} (et donc $\mathsf{Y}(\lambda h.E)$) se comporte, à des β-réductions près, comme la fonction récursive recherchée.

- Si l'évaluation (i.e., la β-réduction) de E termine et ne fait pas intervenir h, alors h_{fix} donne juste E, sinon elle itère avec h_{fix} pour h jusqu'à ce que ce soit le cas : c'est bien ce que fait un appel récursif.

Digression / variante : le combinateur Z

- Le bon fonctionnement du combinateur Y dépend du fait que la stratégie de β-réduction utilisée est extérieure gauche. Sinon le redex $(\lambda x.f(xx))(\lambda x.f(xx))$ peut causer un cycle de β-réductions.

- La variante suivante évite ce problème pour définir une fonction par appels récursifs dans un langage qui n'évalue pas « dans les λ » :

$$\mathsf{Z} := \lambda f.((\lambda x.f(\lambda v.xxv))(\lambda x.f(\lambda v.xxv)))$$

Cette fois :

$$\begin{aligned}\mathsf{Z}(\lambda h.E) &= (\lambda f.((\lambda x.f(\lambda v.xxv))(\lambda x.f(\lambda v.xxv))))(\lambda h.E)\\ &\to (\lambda x.(\lambda h.E)(\lambda v.xxv))(\lambda x.(\lambda h.E)(\lambda v.xxv)) =: h_{\mathrm{fix}}\\ &\to (\lambda h.E)(\lambda v.(\lambda x.(\lambda h.E)(xx))(\lambda x.(\lambda h.E)(xx))v) = (\lambda h.E)(\lambda v.h_{\mathrm{fix}}v)\\ &\to E[h\backslash\lambda v.h_{\mathrm{fix}}v] \text{ (substitution de } \lambda v.h_{\mathrm{fix}}v \text{ pour } h \text{ dans } E)\end{aligned}$$

La forme $\lambda v.h_{\mathrm{fix}}v$ maintient h_{fix} non-évalué (exemple transp. suivant).

Digression (suite) : exemple en Scheme

On prend ici l'exemple de Scheme pour avoir affaire à un langage fonctionnel non typé (le typage empêche l'implémentation directe du combinateur Y ou Z en OCaml ou Haskell).

- Récursion sans combinateurs :

```
(define proto-fibonacci
(lambda (self) ; Pass me as argument!
(lambda (n)
(if (<= n 1) n
(+ ((self self) (- n 1)) ((self self) (- n 2)))))))
(define fibonacci (proto-fibonacci proto-fibonacci))
(map fibonacci '(0 1 2 3 4 5 6))
→ (0 1 1 2 3 5 8)
```

- L'idée est ici exactement celle de l'astuce de Quine : pour m'appeler « moi-même », je m'attends à me recevoir moi-même en argument, et je reproduis ceci lors de l'appel.
- Le combinateur Y (ou Z) automatise cette construction.

Digression (suite) : exemple en Scheme

```
;; (define y-combinator
;;   (lambda (f)
;;     ((lambda (x) (f (x x))) (lambda (x) (f (x x))))))
(define z-combinator
(lambda (f)
((lambda (x) (f (lambda (v) ((x x) v))))
(lambda (x) (f (lambda (v) ((x x) v)))))))
(define pre-fibonacci
(lambda (fib)
(lambda (n)
(if (<= n 1) n
(+ (fib (- n 1)) (fib (- n 2)))))))
(define fibonacci (z-combinator pre-fibonacci))
```

Représentation de l'opérateur μ de Kleene

Rappel : $\mu g(x_1, \ldots, x_k)$ est le plus petit z tel que $g(z, x_1, \ldots, x_k) = 0$ et $g(i, x_1, \ldots, x_k) \downarrow$ pour $0 \leq i < z$, s'il existe.

- On peut faire l'algorithme « rechercher à partir de z » par appels récursifs :

$$h(z, x_1, \ldots, x_k) = \begin{cases} z & \text{si } g(z, x_1, \ldots, x_k) = 0 \\ h(z+1, x_1, \ldots, x_k) & \text{(récursivement) sinon} \end{cases}$$

Alors $\mu g(x_1, \ldots, x_k) = h(0, x_1, \ldots, x_k)$.

- $T := \lambda pmn.p(\lambda y.n)m$ représente $(p, m, n) \mapsto \begin{cases} m & \text{si } p = 0 \\ n & \text{si } p \geq 1 \end{cases}$
- $A := \lambda mfx.f(mfx)$ représente $z \mapsto z + 1$

- La récursion est implémentée avec le combinateur Y :

$$\mathsf{Y}(\lambda hzx_1 \cdots x_k.T(vzx_1 \cdots x_k)z(h(Az)x_1 \cdots x_k))$$
$$\to \mathsf{Y}(\lambda hzx_1 \cdots x_k.(vzx_1 \cdots x_k)(\lambda y.h(Az)x_1 \cdots x_k)z)$$

Équivalence entre λ-calcul et fonctions récursives

- Toute fonction générale récursive (i.e., calculable !) $\mathbb{N}^k \dashrightarrow \mathbb{N}$ est représentée par un terme du λ-calcul (sous les conventions données : application aux entiers de Church, réduction extérieure gauche).

- Réciproquement, toute fonction $\mathbb{N}^k \dashrightarrow \mathbb{N}$ représentable par un terme du λ-calcul est calculable, car on peut implémenter la réduction extérieure gauche.

- Bref, $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ est représentable par un terme du λ-calcul ssi elle est générale récursive.

- De plus, cette équivalence est constructive : il existe des fonctions p.r. :
  - l'une prend en entrée le numéro e d'une fonction générale récursive (et l'arité k) et renvoie le code d'un terme du λ-calcul qui représente cette $\varphi_e^{(k)}$,
  - l'autre prend en entrée le code d'un terme du λ-calcul qui représente une fonction f, et son arité k, et renvoie un numéro e de f dans les fonctions générales récursives $f = \varphi_e^{(k)}$.

Récapitulation

- **Théorème** : les fonctions $\mathbb{N}^k \dashrightarrow \mathbb{N}$ **(1)** générales récursives, **(2)** représentables en λ-calcul, et **(3)** calculables par machine de Turing, coïncident toutes.

On les appelle les **fonctions calculables**.

- De plus, ces équivalences sont constructives : on peut passer algorithmiquement (= calculablement !) d'une représentation à l'autre.
Ce sont des formes de compilation d'un langage en un autre.

- Les questions suivantes ne sont pas décidables algorithmiquement :
  - savoir si une machine de Turing donnée s'arrête,
  - savoir si un terme du λ-calcul est normalisable.

Turing-complétude

Un langage de programmation est dit **Turing-complet** lorsque (convenablement idéalisé !) il permet d'implémenter précisément les fonctions calculables au sens de Church-Turing.

Un ordinateur réel ne peut jamais faire plus qu'une machine de Turing (sauf p.-ê. : faire du hasard vrai). La question est de savoir si le langage permet autant.

Tous les langages de programmation généralistes sont Turing-complets : Python, Java, JavaScript, C, C++, OCaml, Haskell, Lisp, Perl, Ruby, Smalltalk, Prolog...

Certains le sont même plus ou moins « par accident » : CSS, TeX, XSLT, m4... (parfois sous conditions, ou sous réserve d'interprétation).

Pas toujours clair : assembleurs (pas évident d'idéaliser les entiers).

Conséquence du problème de l'arrêt : on ne peut pas algorithmiquement décider si un programme donné (en Python, etc.) termine ou non.

Turing tarpit

« Fosse à bitume de Turing » ?

- Tous les langages usuels se valent du point de vue de la calculabilité. Ce n'est pas pour autant qu'ils se valent en pratique ! (En commodité et/ou efficacité.)

- Ça ne signifie pas qu'un langage Turing-complet peut forcément « tout » faire. Par exemple, un langage qui ne permet comme entrée/sortie que d'afficher des entiers peut être Turing-complet et ne permet pas d'écrire « bonjour ».

- Si en principe on peut convertir toute fonction calculable dans tout langage Turing-complet, la conversion peut devenir extrêmement inefficace, malcommode ou illisible.

λ-calcul non typé et type récursif

Remarque faite pour plus tard.

Le fait d'avoir un type t tel que $t \cong (t \to t)$ permet d'implémenter dans ce type le λ-calcul « non typé » (donc tue l'espoir de décider la terminaison).

Exemple en OCaml (ici, `loop` produit une boucle infinie) :

```
type t = T of (t -> t)
let apply : t -> t -> t = fun (T rator) -> fun rand -> rator rand
let id : t = T (fun x -> x)                                    (* λx.x *)
let ch0 : t = T (fun f -> T (fun x -> x))                      (* λfx.x *)
let ch1 : t = T (fun f -> T (fun x -> apply f x))              (* λfx.fx *)
let ch2 : t = T (fun f -> T (fun x -> apply f (apply f x)))    (* λfx.f(fx) *)
let om : t = T (fun x -> apply x x)                            (* λx.xx *)
let loop : t = apply om om                                     (* (λx.xx)(λx.xx) *)
(* let loop = (fun (T h) -> h (T h)) (T (fun (T h) -> h (T h))) *)
```

Remarquer qu'ici on arrive à provoquer une boucle infinie sans aucun `let rec` (et malgré le typage).

Une méditation googologique

Ceci est une sorte de digression, pour inviter à la réflexion.

« googologie » = étude des grands nombres ; de « googol », nom fantaisiste de 10^{100}

On cherche à minorer calculabl[t] la fonction « castor affairé », c-à-d :

- concevoir un programme dans un langage de programmation idéalisé (machine de Turing, λ-calcul, Python, OCaml...),
- de taille « humainement raisonnable » (peu importent les détails),
- qui termine en temps fini (théoriquement !),
- mais calcule un nombre aussi grand que possible (variante : attend un temps aussi long que possible).

Exemple : implémenter $A_\Delta : n \mapsto A(n,n,n)$ (fonction d'Ackermann diagonale) et calculer $A_\Delta(A_\Delta(\cdots(100))) = A_\Delta^{\circ 100}(100)$ ou qqch du genre.
...On peut faire beaucoup plus grand !

Partie II: Typage simple et calcul propositionnel

INF110 (Logique et Fondements de l'Informatique)

David A. Madore
Télécom Paris
david.madore@enst.fr

2023–2025

http://perso.enst.fr/madore/inf110/transp-inf110.pdf

Git: df53831 Fri Apr 25 14:43:57 2025 +0200

Plan

Qu'est-ce que le typage ?

Philosophie opposée du « codage de Gödel » en calculabilité, lequel représente toute donnée finie par un entier.

- Informellement, un **système de typage** est une façon d'affecter à toute expression et/ou valeur manipulée par un langage informatique un **type** qui contraint son usage (p.ex., « entier », « fonction », « chaîne de caractères »).

Buts :

- attraper plus tôt des **erreurs de programmation** (« ajouter un entier et une chaîne de caractères » est probablement une erreur, ou demande une conversion explicite),
- éviter des **plantages ou problèmes de sécurité** (exécution d'un entier),
- garantir certaines **propriétés du comportement** des programmes ($\rightarrow$ analyse statique), forcément de façon limitée (cf. théorème de Rice), p.ex., la terminaison,
- aider à l'**optimisation** (fonction pure : sans effet de bord).

Cf. systèmes d'unités (homogénéité) en physique.

Variétés de typage

Il y a autant de systèmes de typage que de langages de programmation !

- Typage **statique** (à compilation) vs **dynamique** (à exécution) ou mixte.

Mixte = « graduel ». On préfère : erreur de compilation > erreur à l'exécution > plantage.

- Typage **inféré** par le langage ou **annoté** par le programmeur.

Type = promesse donnée par le langage au programmeur ou par le programmeur au langage ?

- Typage « sûr » ou partiel/contournable (*cast* de la valeur, manipulation de la représentation mémoire, suppression ou report à l'exécution de vérification).
- Typage superficiel (« ceci est une liste ») ou complexe (« ceci est une liste d'entiers »).
- Diverses annotations possibles (« cette fonction est pure », « soulève une exception »).
- Liens avec les mécanismes de sécurité (qui peut faire quoi ?), de gestion de la mémoire (système de typage linéaire), l'évaluation (exceptions), la mutabilité.
- Les types sont-ils citoyens de première classe (:= manipulables dans le langage) ?

Juvénal : « Quis custodiet ipsos custodes? » Quel est le type des types eux-mêmes ?

Opérations de base sur les types

En plus de types de base (p.ex. `Nat` = entiers, `Bool` = booléens), les opérations suivantes sur les types sont *souvent* proposées par les systèmes de typage :

- Types **produits** (= couples, triplets, k-uplets). P.ex. $\texttt{Nat} \times \texttt{Bool}$ = type des paires formées d'un entier et d'un booléen.

  Composantes éventuellement nommées → structures (= enregistrements).

  Produit vide = type trivial, `Unit` (une seule valeur).

- Types **sommes** (= unions). P.ex. $\texttt{Nat} + \texttt{Bool}$ = type pouvant contenir un entier *ou* un booléen, avec un sélecteur de cas.

  Cas particulier : $\texttt{Unit} + \cdots + \texttt{Unit}$ = type « énumération » (pur sélecteur).

  Somme vide = type inhabité (impossible : aucune valeur).

- Types **fonctions** (= exponentielles). P.ex. $\texttt{Nat} \to \texttt{Bool}$ ($\texttt{f}^{\texttt{n}}$ de `Nat` vers `Bool`).

- Types **listes**. P.ex. `List Nat` = type des listes d'entiers.

Quelques fonctionnalités fréquentes

- **Sous-typage** = les valeurs d'un type sont automatiquement des valeurs possibles d'un autre.

- **Polymorphisme** = utilisation de plusieurs types possibles, voire de n'importe quel type (cf. transp. suivant). P.ex. la fonction « identité » $(\forall \mathtt{t})\ \mathtt{t} \to \mathtt{t}$.

- **Familles de types** = fabriquent un type à partir d'un (ou plusieurs) autres. P.ex. `List` (fabrique le type « liste de `t` » à partir de `t`).

- **Types récursifs** = construits par les opérateurs (produits, sommes, fonctions, familles...) à partir des types définis eux-mêmes. P.ex. `Tree` = `List Tree`.

- **Types dépendants** = un type à partir d'une valeur. P.ex. $k \mapsto \mathtt{Nat}^k$.

- **Types opaques** (abstraits, privés...) = types dont les valeurs sont cachées, l'usage est limité à une interface publique.

Polymorphisme

On distingue deux (trois ?) sortes de polymorphismes :

- Polymorphisme **paramétrique** (ou « génériques ») : la même fonction s'applique à l'identique à une donnée de n'importe quel type.

Exemples :

- `head` : $(\forall \mathtt{t})\ \mathtt{List\ t} \to \mathtt{t}$ (renvoie le premier élément d'une liste)
- $\lambda xy.\langle x, y\rangle$: $(\forall \mathtt{u}, \mathtt{v})\ \mathtt{u} \to \mathtt{v} \to \mathtt{u} \times \mathtt{v}$ (fabrique un couple)
- $\lambda xyz.xz(yz)$: $(\forall \mathtt{u}, \mathtt{v}, \mathtt{w})\ (\mathtt{u} \to \mathtt{v} \to \mathtt{w}) \to (\mathtt{u} \to \mathtt{v}) \to \mathtt{u} \to \mathtt{w}$

Pas seulement pour les fonctions ! `[]` : $(\forall \mathtt{t})\ \mathtt{List\ t}$ (liste vide)

Et même : `while true do pass done` : $(\forall \mathtt{t})\ \mathtt{t}$ (boule infinie)

- Polymorphisme **ad hoc** (ou « surcharge » / « *overloading* ») : la fonction agit différemment en fonction du type de son argument (connu à la compilation !).

Le sous-typage est parfois considéré comme une forme de polymorphisme, voire la coercion (*cast*). Les limites de ces notions sont floues.

Tâches d'un système de typage

- **Vérification** de type : vérifier qu'une expression *annotée* a bien le type prétendu par les annotations.

- **Inférence** (« reconstruction » / « assignation ») de type : calculer le type de l'expression en l'absence d'annotations (ou avec annotations partielles).

Algorithme important : **Hindley-Milner** (inférence de type dans les langages fonctionnels à polymorphisme paramétrique). Utilisé dans OCaml, Haskell, etc.

N.B. Dans un système de typage trop complexe, l'inférence (voire la vérification !) peut devenir indécidable (notamment si types de première classe / dépendant de valeurs arbitraires à l'exécution).

Rarement utile en informatique mais essentiel en logique ($\cong$ recherche de preuves) :

- **Habitation** de type : trouver un terme (=expression) ayant un type donné.

P.ex. : y a-t-il un terme de type $(\forall p, q)\ ((p \to q) \to p) \to p$?

N.B. Dans les langages usuels, tous les types sont habités par une boucle infinie (« `while true do pass done` » ou « `let rec f () = f () in f ()` »), même le type vide.

Utilisations du typage au-delà des valeurs stockées

- Annotation des **exceptions soulevables** (fréquent, p.ex. Java).

- Annotation de la **mutabilité** par le type. P.ex. `Nat` = type d'un entier (immuable) mais `Ref Nat` = type d'une référence vers un entier (mutable).

- Annotation des **effets de bord** par le type. P.ex., en Haskell, `Char` = caractère = fonction de zéro argument renvoyant un caractère (fonction pure : toujours le même retour), mais `IO Char` = action avec effets de bord renvoyant un caractère (`IO` est une famille de type appelé « monade » I/O).

- Typage **linéaire** (forme de typage « sous-structurel ») ou types à unicité : assure qu'une valeur est utilisée une et une seule fois dans un calcul (ni duplication ni perte sauf manœuvre spéciale).

Permet d'optimiser la gestion de la mémoire (Rust) et/ou d'annoter les effets de bord (Clean).

Quelques exemples (1)

Les langages impératifs *tendent* à avoir des systèmes de typage moins complexes que les langages fonctionnels.

Éviter les termes de typage « faible » et « fort », qui veulent tout (ou rien) dire.

- Assembleur (langage machine) : aucun typage (tout est donnée binaire).
Idem : machine de Turing, fonctions générales récursives (tout est entier), λ-calcul non typé (tout est fonction).

- C : annoté, vérifié à la compilation (*aucune* vérification à l'exécution), moyennement complexe et contournable (pointeurs génériques `void*`).

- Python, JavaScript, Perl, Scheme, etc. : typage vérifié à l'exécution, superficiel. Suffisant pour éviter les comportements indéfinis.

- Java : annoté, mixte compilation/exécution (double vérification), initialement superficiel (listes non typées), puis introduction de « génériques » ($\rightarrow$ polymorphisme paramétrique) avec Java 5, puis diverses sortes d'inférence.

- Rust : interaction avec la gestion de la mémoire ($\approx$ typage linéaire/affine).

Quelques exemples (2)

Qqs exemples de langages, généralement fonctionnels, ayant un système de typage (très) complexe, mélangeant plusieurs fonctionnalités évoquées (interactions parfois délicates !) :

- OCaml : inférence de type à la H-M, types récursifs, polymor^{sme} paramétrique.

Système de « modules » (« signatures » $\approx$ interfaces abstraites, « foncteurs » entre signatures...), comparable aux « classes » des langages orientés objet.

- Haskell : beaucoup de similarité avec OCaml :

+ polymorphisme ad hoc : « classes de type », comparable aux « modules » de OCaml, aux « classes » des langages orientés objet.

+ purement fonctionnel (toutes les fonctions sont « pures ») : les effets de bord sont enrobés dans des « monades ».

- Mercury (langage de type fonctionnel+logique, inspiré de Haskell+Prolog) : typage comparable à Haskell ; + sous-typage, linéarité.

- Idris : langage fonctionnel + assistant de preuve.

Quelques curiosités

- En F#, les unités de mesure sont intégrées au système de typage, qui peut donc vérifier l'homogénéité physique.

- En Rust, le système de typage évite non seulement les fuites de mémoire mais aussi certains problèmes de concurrence (absence de *race condition* sur les données).

- Certains langages spécialisés assurent, éventuellement en lien avec leur système de typage, des propriétés diverses de leurs programmes : terminaison garantie en temps polynomial (Bellantoni & Cook 1992), voire constant (langage Usuba), validation XML (langage CDuce), absence de fuite d'information (langage Flow Caml), etc.

- En Idris, le système de typage est aussi un système de preuve (cf. Curry-Howard après) et permet de certifier des invariants quelconques d'un programme (p.ex., correction d'un algorithme de tri).

Typage et terminaison

Un système de typage peut garantir que **tout programme bien typé termine**.

Ce n'est pas le cas pour les langages de programmation usuels (en OCaml, Haskell, etc., un programme bien typé peut faire une boucle infinie).

Si on veut que le système de typage soit décidable, ceci met forcément des limites sur l'expressivité du langage ($\leftarrow$ indécidabilité du problème de l'arrêt).

Notamment, le langage ne permettra pas d'écrire son propre interpréteur (même argument « diagonal » que pour les fonctions p.r. par thm. de réc$^{\text{sion}}$ de Kleene).

Notamment aussi : pas de boucle illimitée, pas d'appels récursifs *non contraints*, pas de type tel que $\mathtt{t} \cong (\mathtt{t} \to \mathtt{t})$. Aucun terme de type vide !

Néanmoins, le langage peut être beaucoup plus puissant que les fonctions p.r.

Exemple de tel langage : Coq.

La correspondance de Curry-Howard (avant-goût)

Idée générale : établir une analogie, voire une correspondance précise entre

- **types** et **termes** (=programmes) de ces types dans un système de typage,
- **propositions** et **preuves** de ces propositions dans un système logique.

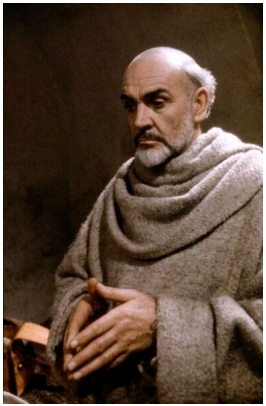
P.ex., la preuve évidente de l'affirmation $(A \wedge B) \Rightarrow A$ (« si A et B sont vraies alors A est vraie ») correspond à la « première projection » de type $a \times b \to a$.

Ceci « explique » que le type d'un terme tel que $\lambda xy.x$, soit $u \to v \to u$, soit une vérité logique, en l'occurrence $U \Rightarrow V \Rightarrow U$ (« si U est vrai alors, si V est vrai alors U est vrai »).

Beaucoup de variations sur cette correspondance, mais il y a des restrictions :

- le langage informatique doit garantir la terminaison (pas d'appels récursifs !), sinon cela reviendrait à permettre les preuves circulaires,
- la logique concernée est plutôt « intuitionniste » (sans tiers exclu),
- diverses subtilités notamment dans la correspondance entre types sommes paramétriques (types Σ) et $\exists$ côté logique.

La correspondance de Curry-Howard (illustration graphique)



Preuves mathématiques
(contemplatives ?)

Programmes informatiques
(dynamiques ?)

On a du mal à le croire, mais c'est la même chose !
(N.B. : ne pas prendre ce résumé simpliste trop au sérieux.)

La correspondance de Curry-Howard (premier aperçu)

Divulgâchis pour la suite !

La correspondance de Curry-Howard fait notamment correspondre :

- type **fonction** $A \to B$ avec **implication** logique $A \Rightarrow B$,
- **application** d'une fonction ($A \to B$ à un argument de type A) avec ***modus ponens*** (si $A \Rightarrow B$ et A, alors B),
- **abstraction** (= création d'une fonction) avec ouverture d'une **hypothèse** (« supposons A : alors (...) donc B ; ceci prouve $A \Rightarrow B$ »),
- type **produit** (= couples) $A \times B$ avec **conjonction** (« et ») logique $A \wedge B$,
- type **somme** $A + B$ avec **disjonction** (« ou ») logique $A \vee B$,
- types **trivial** (`Unit`) et **vide** avec **vrai** $\top$ et **faux** $\bot$ logiques,
- mais pour la **négation**... c'est plus délicat.

Les détails demandent un système de typage et un système logique précisément définis.

Paradoxe de Curry (interlude distrayant)

Variante plus sophistiquée de « cette phrase est fausse ». C'est un exemple de preuve circulaire ($\leftrightarrow$ combinateur Y de Curry par la correspondance de C-H), invalide en logique.

Je tiens l'affirmation : « si j'ai raison, alors je suis un grand génie ».

- clairement, si j'ai raison, je suis un grand génie ;
- mais c'était justement mon affirmation : donc j'ai raison ;
- donc je suis un grand génie. $\square$

Remplacer « j'ai raison » par « cette phrase est vraie » (voire utiliser l'astuce de Quine pour éviter « cette phrase ») et « je suis un grand génie » par absolument n'importe quoi.

- Ce qui est correct : si on peut construire un énoncé A tel que $A \Leftrightarrow (A \Rightarrow B)$ (ici A est l'affirmation tenue et B est la conclusion voulue) alors B vaut :

$$(A \Leftrightarrow (A \Rightarrow B)) \Rightarrow B$$

(Preuve correcte : supposons A : par hypothèse, ceci signifie $A \Rightarrow B$; on a donc B ; tout ceci prouve $A \Rightarrow B$. Bref on a $A \Rightarrow B$. Mais par hypothèse c'est A. Donc A vaut. Donc B aussi.)

- Ce qui est fallacieux : la supposition tacite qu'on peut construire un tel A.

L'astuce de Quine permet de faire une auto-référence sur la syntaxe, pas sur la vérité.

Théories des types pour la logique (très bref aperçu)

- Langages spécialisés pour servir à la fois à l'écriture de programmes informatiques et de preuves mathématiques ; ils peuvent être soit sous forme de systèmes abstraits, soit sous forme d'implémentation informatique (« assistants de preuve » : Coq, Agda, Lean...).

- Dans tous les cas il s'agit de langages assurant la terminaison des programmes (cf. transp. 13), ou, puisqu'il s'agit de variantes du λ-calcul, la *normalisation forte*. Le type vide doit être inhabité !

Quelques grandes familles (chacune avec énormément de variantes) :

- théories des types à la Martin-Löf (« MLTT ») : suit jusqu'au bout la correspondance de Curry-Howard (*aucune* distinction entre types et propositions ; pas de $\exists$ mais plutôt un Σ), $\rightarrow$ Agda ;
- variantes du « calcul des constructions » (« CoC »), $\rightarrow$ Coq ;
- théorie homotopique des types (« HoTT ») : « égalité = isomorphisme ».

Le λ-cube de Barendregt (très bref aperçu)

Il s'agit d'un ensemble de $8 = 2^3$ théories des types pour la logique : la plus faible est le « λ-calcul simplement typé » ($\lambda_{\to}$, décrit plus loin), la plus forte le « calcul des constructions » (« CoC »).

Chaque théorie du cube est caractérisée par l'absence ou la présence de chacune des 3 fonctionnalités suivantes ($\lambda_{\to}$ n'a aucune des trois, CoC a les trois) :

- termes pouvant dépendre de types, ou polymorphisme paramétrique « explicitement quantifié » (p.ex. $\prod t.\ (t \to t)$) ;
- types pouvant dépendre de types, ou familles de types ;
- types pouvant dépendre de termes, ou « types dépendants », correspondant côté logique à la quantification sur les individus.

On peut ensuite encore ajouter des fonctionnalités : par exemple, Coq est basé sur le « calcul des constructions inductives » qui ajoute au calcul des constructions des mécanismes systématiques pour former des types (positivement !) récursifs.

Le λ-calcul simplement typé : description sommaire

- Le λ-calcul simplement typé (=: λCST ou $\lambda_{\to}$) est une variante typée du λ-calcul, assurant la propriété de terminaison (normalisation forte).
- Il a une seule opération sur les types, le type **fonction** : donnés deux types σ, τ, on a un type $\sigma \to \tau$ pour les fonctions de l'un vers l'autre.
- L'application et l'abstraction doivent respecter le typage :
  - si P a pour type $\sigma \to \tau$ et Q a type σ alors PQ a pour type τ,
  - si E a pour type τ en faisant intervenir une variable v libre de type σ alors $\lambda(v : \sigma).E$ (« fonction prenant v de type σ et renvoyant E ») a pour type $\sigma \to \tau$.
- Typage **annoté** (=« à la Church ») : on écrit $\lambda(v : \sigma).E$ pas juste $\lambda v.E$.
- On notera « $x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash E : \tau$ » pour dire « E est bien typé avec pour type τ lorsque $x_1, \ldots, x_k$ sont des variables libres de types $\sigma_1, \ldots, \sigma_k$:
  - $x_1 : \sigma_1, \ldots, x_k : \sigma_k$ s'appelle un **contexte** de typage (souvent Γ),
  - $\Gamma \vdash E : \tau$ s'appelle un **jugement** de typage.

Exemples de termes et de typages

On donnera les règles précises plus tard, commençons par quelques exemples.

- $f : \alpha \to \beta,\ x : \alpha \ \vdash\ fx : \beta$
  Lire : « dans le contexte où f est une variable de type $\alpha \to \beta$ et x une variable de type α, alors fx est de type β ».
- $f : \beta \to \alpha \to \gamma,\ x : \alpha,\ y : \beta \ \vdash\ fyx : \gamma$
  (Parenthéser $\beta \to \alpha \to \gamma$ comme $\beta \to (\alpha \to \gamma)$ et fyx comme $(fy)x$.)
- $f : \beta \to \alpha \to \gamma,\ x : \alpha \ \vdash\ \lambda(y : \beta).fyx \ :\ \beta \to \gamma$
  Comprendre $\lambda(y : \beta).fyx$ comme « fonction prenant y de type β et renvoyant fyx ».
- $x : \alpha \ \vdash\ \lambda(f : \alpha \to \beta).fx \ :\ (\alpha \to \beta) \to \beta$
  « Si x est de type α alors le terme $\lambda(f : \alpha \to \beta).fx$ est de type $(\alpha \to \beta) \to \beta$. »
- $\vdash\ \lambda(x : \alpha).\lambda(f : \alpha \to \beta).fx \ :\ \alpha \to (\alpha \to \beta) \to \beta$
  « Le terme $\lambda(x : \alpha).\lambda(f : \alpha \to \beta).fx$ a type $\alpha \to (\alpha \to \beta) \to \beta$ dans le contexte vide. »
  (Parenthéser $\alpha \to (\alpha \to \beta) \to \beta$ comme $\alpha \to ((\alpha \to \beta) \to \beta)$.)

Types et prétermes

- Un **type** du λCST est (inductivement) :
  - une **variable de type** (α, β, γ... en nombre illimité),
  - un **type fonction** $(\sigma \to \tau)$ où σ et τ sont deux types.

- Un **préterme** du λCST est (inductivement) :
  - une **variable de terme** (a, b, c... en nombre illimité),
  - une **application** (PQ) où P et Q sont deux termes,
  - une **abstraction** $\lambda(v : \sigma).E$ avec v variable, σ type et E préterme.

- Conventions d'écriture :
  - « $\rho \to \sigma \to \tau$ » signifie « $(\rho \to (\sigma \to \tau))$ » ; « xyz » signifie « $((xy)z)$ » ;
  - on note « $\lambda(x : \sigma, t : \tau).E$ » ou « $\lambda(x : \sigma)(t : \tau).E$ » pour « $\lambda(x : \sigma).\lambda(t : \tau).E$ » ;
  - l'abstraction est moins prioritaire que l'application ;
  - on considère les termes à renommage près des variables liées (α-conversion).

Règles de typage

- Un **typage** est la donnée d'un préterme M et d'un type σ. On note « $M : \sigma$ ».
- Un **contexte** est un ensemble fini Γ de typages $x_i : \sigma_i$ où $x_1, \ldots, x_k$ sont des variables de terme distinctes. On le note « $x_1 : \sigma_1, \ldots, x_k : \sigma_k$ ».
- Un **jugement** de typage est la donnée d'un contexte Γ et d'un typage $E : \tau$, sujet aux règles ci-dessous. On note $\Gamma \vdash E : \tau$ (ou juste $\vdash E : \tau$ si $\Gamma = \varnothing$), et on dit que E est un « **terme** (= bien typé) de type τ dans le contexte Γ ».

Règles de typage du λCST : (quels que soient $\Gamma, x, \sigma, \ldots$,)

- (« variable ») si $(x : \sigma) \in \Gamma$ alors $\Gamma \vdash x : \sigma$;
- (« application ») si $\Gamma \vdash P : \sigma \to \tau$ et $\Gamma \vdash Q : \sigma$ alors $\Gamma \vdash PQ : \tau$;
- (« abstraction ») si $\Gamma, v : \sigma \vdash E : \tau$ alors $\Gamma \vdash \lambda(v : \sigma).E : \sigma \to \tau$.

(Comprendre : l'ensemble des jugements de typage est l'ensemble engendré par les règles ci-dessus, i.e., le plus petit ensemble qui les respecte.)

- Une **dérivation** d'un jugement est un arbre (d'instances de) règles qui aboutit au jugement considéré.

Représentation des règles de typage

Les trois règles de typage du λCST :

$$\textsc{Var}\ \frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\textsc{App}\ \frac{\Gamma \vdash P : \sigma \to \tau \qquad \Gamma \vdash Q : \sigma}{\Gamma \vdash PQ : \tau}$$

$$\textsc{Abs}\ \frac{\Gamma, v : \sigma \vdash E : \tau}{\Gamma \vdash \lambda(v : \sigma).E : \sigma \to \tau}$$

- Chaque « fraction » indique que le jugement écrit en-dessous découle par la règle inscrite à côté à partir des hypothèses portées au-dessus.

Exemple de dérivation

Montrons le jugement selon lequel $\lambda(f : \beta \to \alpha \to \gamma).\lambda(x : \alpha).\lambda(y : \beta).fyx$ est un terme de type $(\beta \to \alpha \to \gamma) \to (\alpha \to \beta \to \gamma)$ dans le contexte vide :

$$\textsc{Abs}\ \dfrac{\textsc{Abs}\ \dfrac{\textsc{Abs}\ \dfrac{\textsc{App}\ \dfrac{\textsc{App}\ \dfrac{\textsc{Var}\ \dfrac{}{f:\beta\to\alpha\to\gamma,\ x:\alpha,y:\beta \vdash f : \beta \to \alpha \to \gamma} \qquad \textsc{Var}\ \dfrac{}{f:\beta\to\alpha\to\gamma,\ x:\alpha,y:\beta \vdash y : \beta}}{f:\beta\to\alpha\to\gamma,\ x:\alpha,y:\beta \vdash fy : \alpha \to \gamma} \qquad \textsc{Var}\ \dfrac{}{f:\beta\to\alpha\to\gamma,\ x:\alpha,y:\beta \vdash x : \alpha}}{f : \beta \to \alpha \to \gamma, x : \alpha, y : \beta \vdash fyx : \gamma}}{f : \beta \to \alpha \to \gamma, x : \alpha \vdash \lambda(y : \beta).fyx : \beta \to \gamma}}{f : \beta \to \alpha \to \gamma \vdash \lambda(x : \alpha).\lambda(y : \beta).fyx : \alpha \to \beta \to \gamma}}{\vdash \lambda(f : \beta \to \alpha \to \gamma).\lambda(x : \alpha).\lambda(y : \beta).fyx : (\beta \to \alpha \to \gamma) \to (\alpha \to \beta \to \gamma)}$$

Chaque barre horizontale justifie le jugement écrit en-dessous par la règle inscrite à côté à partir des hypothèses portées au-dessus.

Ceci est typographiquement abominable et hautement redondant, ce qui explique qu'on écrive rarement de tels arbres complètement.

Propriétés du typage

Les propriétés suivantes sont faciles mais utiles :

- **Affaiblissement :** si $\Gamma \subseteq \Gamma'$ sont deux contextes et $\Gamma \vdash M : \sigma$ alors $\Gamma' \vdash M : \sigma$ aussi.

On pouvait présenter les règles en limitant la règle « variable » à « $x : \sigma \vdash x : \sigma$ » et en prenant l'affaiblissement comme règle. C'est peut-être préférable.

- **Duplication :** si $\Gamma, x : \rho, x' : \rho \vdash M : \sigma$ alors $\Gamma, x : \rho \vdash M[x'\backslash x] : \sigma$ aussi (où $M[x'\backslash x]$ désigne la substitution correcte de x' par x).

Remarque : Le typage linéaire supprime notamment l'affaiblissement et la duplication.

- **Variables libres :** si $x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash E : \tau$ alors toute variable libre de E est une des $x_1, \ldots, x_k$.

- **Variables inutiles :** si $\Gamma \vdash E : \tau$ alors $\Gamma|_{\mathrm{free}(E)} \vdash E : \tau$ où $\Gamma|_{\mathrm{free}(E)}$ est la partie de Γ concernant les variables libres de E.

- **Renommage :** si $\Gamma \vdash E : \tau$ alors ceci vaut encore après tout renommage des variables libres.

Construction de la dérivation

La dérivation du jugement de typage $\Gamma \vdash M : \sigma$ d'un (pré)terme M du λCST se construit systématiquement et évidemment à partir de M (aucun choix à faire !) :

- si $M = x$ est une variable, alors un $x : \sigma$ doit être dans le contexte Γ (sinon : échouer), et la dérivation consiste en la règle « variable » ;
- si $M = PQ$ est une application, construire des dérivations de jugements $\Gamma \vdash P : \rho$ et $\Gamma \vdash Q : \sigma$, on doit avoir $\rho = (\sigma \to \tau)$ (sinon : échouer), et la dérivation finit par la règle « application » et donne $\Gamma \vdash M : \tau$;
- si $M = \lambda(v : \sigma).E$ est une abstraction, construire une dérivation de jugement $\Gamma' \vdash E : \tau$ dans $\Gamma' := \Gamma \cup \{v : \sigma\}$ (quitte à renommer v), et la dérivation finit par la règle « abstraction » et donne $\Gamma \vdash M : \sigma \to \tau$.

- **Donc :** aussi bien la vérification de type (vérifier $\Gamma \vdash M : \sigma$) que l'assignation de type (trouver σ à partir de Γ, M) sont (très facilement) décidables dans le λCST. De plus, σ est unique (donnés Γ et M).

Problèmes faciles et moins faciles

- Facile (transp. précédent) : donné un (pré)terme M, trouver son (seul possible) type σ est facile (notamment : vérifier que M est un terme $=$ bien typé, ou vérifier un jugement de type).

  Le terme donne directement l'arbre de dérivation.

- Facile aussi : réciproquement, donné un arbre de dérivation où on a effacé les termes pour ne garder que les types, retrouver un terme (transp. suivant).

  L'arbre de dérivation redonne directement le terme.

- Moins facile : donné un terme « désannoté », i.e., un terme du λ-calcul non typé, p.ex. $\lambda xyz.xz(yz)$, savoir s'il correspond à un terme (typable) du λCST.

  $\rightarrow$ Algorithme de Hindley-Milner (*inférence* de type).

- Moins facile : donné un type, savoir s'il existe un terme ayant ce type.

  $\rightarrow$ Décidabilité du calcul propositionnel intuitionniste.

Reconstruction du terme typé

Peut-on retrouver les termes manquants dans un arbre de dérivation dont on n'a donné que les types et les règles appliquées ?

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\overset{?:\beta\to\alpha\to\gamma,}{?:\alpha,?:\beta} \vdash ? : \beta\to\alpha\to\gamma}\,\textsc{Var} \qquad \dfrac{}{\overset{?:\beta\to\alpha\to\gamma,}{?:\alpha,?:\beta} \vdash ? : \beta}\,\textsc{Var}}{\overset{?:\beta\to\alpha\to\gamma,}{?:\alpha,?:\beta} \vdash ? : \alpha\to\gamma}\,\textsc{App} \qquad \dfrac{}{\overset{?:\beta\to\alpha\to\gamma,}{?:\alpha,?:\beta} \vdash ? : \alpha}\,\textsc{Var}}{? : \beta\to\alpha\to\gamma, ? : \alpha, ? : \beta \vdash ? : \gamma}\,\textsc{App}}{? : \beta\to\alpha\to\gamma, ? : \alpha \vdash ? : \beta\to\gamma}\,\textsc{Abs}}{? : \beta\to\alpha\to\gamma \vdash ? : \alpha\to\beta\to\gamma}\,\textsc{Abs}}{\vdash ? : (\beta\to\alpha\to\gamma)\to(\alpha\to\beta\to\gamma)}\,\textsc{Abs}$$

Oui (aux noms des variables près) à condition, en cas de types identiques dans le contexte, d'identifier l'élément utilisé pour chaque règle « variable ».

- Le terme typé représente exactement l'arbre de dérivation du jugement le concernant.

Lemme de substitution

- **Lemme :** Si $\Gamma, v : \sigma \vdash E : \tau$ et $\Gamma \vdash T : \sigma$ alors $\Gamma \vdash E[v \backslash T] : \tau$, où $E[v \backslash T]$ désigne le terme obtenu par substitution correcte de v par T dans E.

Esquisse de preuve : reprendre l'arbre de dérivation de $\Gamma, v : \sigma \vdash E : \tau$ en supprimant $v : \sigma$ du contexte et en substituant v par T partout à droite du '$\vdash$'. Les règles s'appliquent à l'identique, sauf la règle « variable » introduisant $v : \sigma$, qui est remplacée par l'arbre de dérivation de $\Gamma \vdash T : \sigma$. □

*

Remarquer la similarité entre

$$\textsc{App}\ \frac{\textsc{Abs}\ \dfrac{\Gamma, v : \sigma \vdash E : \tau}{\Gamma \vdash \lambda(v : \sigma).E : \sigma \to \tau} \qquad \Gamma \vdash T : \sigma}{\Gamma \vdash (\lambda(v : \sigma).E)T : \tau}$$

et

$$\textsc{Subs}\ \frac{\Gamma, v : \sigma \vdash E : \tau \qquad \Gamma \vdash T : \sigma}{\Gamma \vdash E[v \backslash T] : \tau}$$

Typage et β-réduction

On rappelle que la « β-réduction » désigne le remplacement en sous-expression d'un « redex » $(\lambda(v:\sigma).E)T$ par son « réduit » $E[v\backslash T]$.

On note $X \to_\beta X'$ pour une β-réduction, et $X \twoheadrightarrow_\beta X'$ pour une suite de β-réductions.

- D'après le lemme de substitution, si $\Gamma \vdash (\lambda(v:\sigma).E)T : \tau$ alors on a aussi $\Gamma \vdash E[v\backslash T] : \tau$.
- Toujours d'après ce lemme, si $X \twoheadrightarrow_\beta X'$ et $\Gamma \vdash X : \tau$ alors $\Gamma \vdash X' : \tau$.

Moralité : le typage est compatible avec l'évaluation du λ-calcul (un terme bien typé reste bien typé quand on effectue la β-réduction et le type ne change pas).

Normalisation forte

- **Théorème** (Turing, Tait, Girard) : le λ-calcul simplement typé est **fortement normalisant**, c'est-à-dire que toute suite de β-réductions sur un terme (bien typé !) termine.

Idée cruciale de la preuve : une induction directe échoue (P, Q fortement normalisables $\not\Rightarrow PQ$ fortement normalisable). À la place, on introduit une notion *plus forte* permettant l'induction :

- **Définition** technique : un terme P de type ρ est dit « fortement calculable » lorsque :
  - si $\rho = \alpha$ est une variable de type (type « atomique ») : P est fortement normalisable,
  - si $\rho = (\sigma \to \tau)$ est un type fonction : pour tout Q de type σ fortement calculable, PQ est fortement calculable.

On prouve alors successivement :

- tout terme fortement calculable est fortement normalisable (induction facile sur le type),
- si $x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash E : \tau$ et si $P_1, \ldots, P_k$ sont de types $\sigma_1, \ldots, \sigma_k$, fortement calculables et n'impliquant pas $x_1, \ldots, x_k$, alors la substitution de P_i pour x_i dans E est fortement calculable (preuve par induction sur la dérivation du jugement, i.e., induction sur E : le cas délicat est l'abstraction).

Le calcul propositionnel : présentation sommaire

- Le **calcul propositionnel** est une forme de logique qui ne parle que de « propositions » (énoncés logiques) : pas de variables d'individus (p.ex. entiers naturels, ensembles...), pas de quantification (pas de $\forall, \exists$).

- Les **connecteurs logiques** sont : $\Rightarrow$ (implication), $\wedge$ (conjonction logique : « et »), $\vee$ (disjonction : « ou »), $\top$ (« vrai ») et $\bot$ (« faux » ou « absurde »). Tous sont binaires sauf $\top$, $\bot$ (nullaires).

On peut y ajouter $\neg$ (négation logique) unaire : $\neg P$ est une abréviation de $P \Rightarrow \bot$ (soit : « P est absurde »).

On distingue (les règles précises seront décrites plus loin) :

- la logique **classique** ou **booléenne** qui admet la règle du tiers exclu (« tertium non datur ») sous une forme ou une autre : logique usuelle des maths ; on peut la modéliser par 2 valeurs de vérité (« vrai » et « faux ») ;
- la logique **intuitionniste**, plus *faible*, qui n'admet pas cette règle : il n'y a pas vraiment de « valeur de vérité ».

Le calcul propositionnel : syntaxe

- Une **formule** du calcul propositionnel est (inductivement) :
  - une **variable propositionnelle** (A, B, C...),
  - l'application d'un connecteur : $(P \Rightarrow Q)$, $(P \wedge Q)$, $(P \vee Q)$ où P, Q sont deux formules, ou encore $\top$, $\bot$.
- Conventions d'écriture :
  - $\neg P$ abrège « $(P \Rightarrow \bot)$ » ;
  - on omet certaines parenthèses : $\neg$ est prioritaire sur $\wedge$ qui est prioritaire sur $\vee$ qui est prioritaire sur $\Rightarrow$ (ne pas abuser de l'omission des parenthèses, merci) ;
  - $\wedge$ et $\vee$ associent à gauche disons (ça n'aura pas d'importance) ; mais $\Rightarrow$ associe à droite : $P \Rightarrow Q \Rightarrow R$ signifie $(P \Rightarrow (Q \Rightarrow R))$;
  - parfois : $P \Leftrightarrow Q$ pour $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$ (priorité encore plus basse ?).
- Si $P_1, \ldots, P_r, Q$ sont des formules, $P_1, \ldots, P_r \vdash Q$ s'appelle un **séquent**, à comprendre comme « sous les hypothèses $P_1, \ldots, P_r$, on a Q ».
- Notation historique : « $\supset$ » pour notre « $\Rightarrow$ », et « $\Rightarrow$ » pour notre « $\vdash$ ».

Le calcul propositionnel intuitionniste : connecteurs

- Chaque connecteur logique a des règles d'**introduction** permettant de démontrer ce connecteur, et des règles d'**élimination** permettant de l'utiliser.

En français, et un peu informellement :

- pour introduire $\Rightarrow$: on démontre Q sous l'hypothèse P, ce qui donne $P \Rightarrow Q$ (sans hypothèse : « hypothèse déchargée ») ;
- pour éliminer $\Rightarrow$: si on a $P \Rightarrow Q$ et P, on a Q (*modus ponens*) ;
- pour introduire $\wedge$: on démontre Q_1 et Q_2, ce qui donne $Q_1 \wedge Q_2$; pour l'éliminer : si on a $Q_1 \wedge Q_2$ on en tire Q_1 resp. Q_2 ;
- pour introduire $\vee$: on démontre Q_1 et Q_2, ce qui donne $Q_1 \vee Q_2$;
- pour éliminer $\vee$: si on a $P_1 \vee P_2$, on démontre Q successivement sous les hypothèses P_1 et P_2, ce qui donne Q (hypothèses déchargées) ;
- pour introduire $\top$, c'est trivial, et on ne peut pas l'éliminer ;
- pour éliminer $\bot$, on tire la conclusion qu'on veut (*ex falso quodlibet*), et on ne peut pas l'introduire.

Le calcul propositionnel intuitionniste : règles

Axiomes : si $Q \in \{P_1, \ldots, P_r\}$ alors $P_1, \ldots, P_r \vdash Q$.

Introduction et élimination des connecteurs en style « déduction naturelle » :

	Intro	Élim
$\Rightarrow$	$\dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$	$\dfrac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}$
$\wedge$	$\dfrac{\Gamma \vdash Q_1 \qquad \Gamma \vdash Q_2}{\Gamma \vdash Q_1 \wedge Q_2}$	$\dfrac{\Gamma \vdash Q_1 \wedge Q_2}{\Gamma \vdash Q_1} \qquad \dfrac{\Gamma \vdash Q_1 \wedge Q_2}{\Gamma \vdash Q_2}$
$\vee$	$\dfrac{\Gamma \vdash Q_1}{\Gamma \vdash Q_1 \vee Q_2} \qquad \dfrac{\Gamma \vdash Q_2}{\Gamma \vdash Q_1 \vee Q_2}$	$\dfrac{\Gamma \vdash P_1 \vee P_2 \qquad \Gamma, P_1 \vdash Q \qquad \Gamma, P_2 \vdash Q}{\Gamma \vdash Q}$
$\top$	$\dfrac{}{\Gamma \vdash \top}$	(néant)
$\bot$	(néant)	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash Q}$

Dans tout ça, Γ désigne un jeu (non ordonné, mais avec multiplicités) d'hypothèses.
Les hypothèses en vert sont « déchargées », c'est-à-dire qu'elles disparaissent.

Exemples de démonstrations

Indiquant à côté de chaque barre une abréviation de la règle utilisée (« $\Rightarrow$Int » pour introduction de $\Rightarrow$, « $\wedge$Élim$_2$ » pour la 2$^{\text{e}}$ règle d'élimination de $\wedge$, etc.) :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\,\textsc{Ax}}{A \wedge B \vdash B}\,\wedge\textsc{Élim}_2 \qquad \dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\,\textsc{Ax}}{A \wedge B \vdash A}\,\wedge\textsc{Élim}_1}{A \wedge B \vdash B \wedge A}\,\wedge\textsc{Int}}{\vdash A \wedge B \Rightarrow B \wedge A}\,\Rightarrow\textsc{Int}$$

$$\dfrac{\dfrac{\dfrac{}{A \vee B \vdash A \vee B}\,\textsc{Ax} \qquad \dfrac{\dfrac{}{A \vee B, A \vdash A}\,\textsc{Ax}}{A \vee B, A \vdash B \vee A}\,\vee\textsc{Int}_2 \qquad \dfrac{\dfrac{}{A \vee B, B \vdash B}\,\textsc{Ax}}{A \vee B, B \vdash B \vee A}\,\vee\textsc{Int}_1}{A \vee B \vdash B \vee A}\,\vee\textsc{Élim}}{\vdash A \vee B \Rightarrow B \vee A}\,\Rightarrow\textsc{Int}$$

Présentation différente

On peut aussi n'écrire que les conclusions (partie droite du signe « $\vdash$ »), à condition d'indiquer pour chaque hypothèse déchargée à quel endroit elle l'a été (ceci sacrifie de la lisibilité pour un gain de place) :

$$\dfrac{\dfrac{\dfrac{}{A \wedge B}\,u}{B}\,\wedge\text{É}\text{LIM}_2 \qquad \dfrac{\dfrac{}{A \wedge B}\,u}{A}\,\wedge\text{É}\text{LIM}_1}{B \wedge A}\,\wedge\text{INT}}{A \wedge B \Rightarrow B \wedge A}\,\Rightarrow\text{INT}(u)$$

$$\dfrac{\dfrac{}{A \vee B}\,u \qquad \dfrac{}{A}\,v}{B \vee A}\,\vee\text{INT}_2 \qquad \dfrac{}{B}\,v'}{B \vee A}\,\vee\text{INT}_1}{A \vee B \Rightarrow B \vee A}\,\vee\text{É}\text{LIM}(v,v')\ \Rightarrow\text{INT}(u)$$

N.B. : une même hypothèse *peut* être déchargée sur plusieurs endroits (u dans le 1er exemple).

Encore une présentation différente

Présentation « drapeau », plus proche de l'écriture naturelle, commode à vérifier :

(1)	$\boxed{A \wedge B}$	
(2)	$\quad B$	$\wedge$Élim$_2$ sur (1)
(3)	$\quad A$	$\wedge$Élim$_1$ sur (1)
(4)	$\quad B \wedge A$	$\wedge$Int sur (2), (3)
(5)	$A \wedge B \Rightarrow B \wedge A$	$\Rightarrow$Int de (1) dans (4)

(1)	$\boxed{A \vee B}$	
(2)	$\quad \boxed{A}$	
(3)	$\quad\quad B \vee A$	$\vee$Int$_2$ sur (2)
(4)	$\quad \boxed{B}$	
(5)	$\quad\quad B \vee A$	$\vee$Int$_1$ sur (4)
(6)	$\quad B \vee A$	$\vee$Élim sur (1) de (2) dans (3) et de (4) dans (5)
(7)	$A \vee B \Rightarrow B \vee A$	$\Rightarrow$Int de (1) dans (6)

Quelques tautologies du calcul propositionnel intuitionniste

- Lorsque $\vdash P$ (sans hypothèse), on dit que P est **valable** dans le calcul propositionnel intuitionniste, ou est une **tautologie** de celui-ci.

Quelques tautologies importantes (« $P \Leftrightarrow Q$ » abrège « $P \Rightarrow Q$ et $Q \Rightarrow P$ ») :

$A \Rightarrow A \qquad A \Rightarrow B \Rightarrow A$	$(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$
$(A \Rightarrow B \Rightarrow C) \Leftrightarrow (A \wedge B \Rightarrow C)$	$A \wedge A \Leftrightarrow A \qquad A \vee A \Leftrightarrow A$
$A \wedge B \Rightarrow A \qquad A \wedge B \Rightarrow B$	$A \Rightarrow A \vee B \qquad B \Rightarrow A \vee B$
$A \Rightarrow B \Rightarrow A \wedge B$	$(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow A \vee B \Rightarrow C$
$A \wedge B \Leftrightarrow B \wedge A$	$A \vee B \Leftrightarrow B \vee A$
$(A \wedge B) \wedge C \Leftrightarrow A \wedge (B \wedge C)$	$(A \vee B) \vee C \Leftrightarrow A \vee (B \vee C)$
$(A \Rightarrow B) \wedge (A \Rightarrow C) \Leftrightarrow (A \Rightarrow B \wedge C)$	$(A \Rightarrow C) \wedge (B \Rightarrow C) \Leftrightarrow (A \vee B \Rightarrow C)$
$(A \Rightarrow B) \vee (A \Rightarrow C) \Rightarrow (A \Rightarrow B \vee C)$	$(A \Rightarrow C) \vee (B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)$
$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$	$A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$
$\top$	$\bot \Rightarrow C$
$\top \wedge A \Leftrightarrow A \qquad \bot \wedge A \Leftrightarrow \bot$	$\top \vee A \Leftrightarrow \top \qquad \bot \vee A \Leftrightarrow A$
$\neg A \wedge \neg B \Leftrightarrow \neg(A \vee B)$	$\neg A \vee \neg B \Rightarrow \neg(A \wedge B)$
$A \Rightarrow \neg\neg A$	$(\neg\neg A \Rightarrow \neg\neg B) \Leftrightarrow \neg\neg(A \Rightarrow B)$
$(\neg\neg A \wedge \neg\neg B) \Leftrightarrow \neg\neg(A \wedge B)$	$(\neg\neg A \vee \neg\neg B) \Rightarrow \neg\neg(A \vee B)$
$(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$	$\neg A \Leftrightarrow \neg\neg\neg A$
$\neg(A \wedge \neg A)$	$\neg\neg(A \vee \neg A)$

Quelques non-tautologies

Rappelons qu'on ne permet pas de raisonnement par l'absurde dans notre logique.

Les énoncés suivants ne sont pas des tautologies du calcul propositionnel intuitionniste (même s'ils *sont* valables en calcul propositionnel *classique*) :

- $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ (« loi de Peirce »)
- $A \vee \neg A$ (« tiers exclu »)
- $\neg\neg A \Rightarrow A$ (« élimination de la double négation »)
- $\neg(A \wedge B) \Rightarrow \neg A \vee \neg B$ (une des « lois de De Morgan »)
- $(A \Rightarrow B) \Rightarrow \neg A \vee B$ (la réciproque est bien valable intuitionnistement)
- $(\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$ (même remarque)
- $(A \Rightarrow B) \vee (B \Rightarrow A)$ (« loi de Dummett »)
- $\neg A \vee \neg\neg A$ (« tiers exclu faible »)
- $(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$ (même si $(\neg\neg A \Rightarrow A)$ pour *tout* A donne $A \vee \neg A$ pour tout A)

Mais comment sait-on que quelque chose *n'est pas* une tautologie, au juste ?

Preuve de la négation vs raisonnement par l'absurde

- Rappel : la négation $\neg A$ abrège $A \Rightarrow \bot$ (« si A est vrai alors ABSURDITÉ »).

Bien distinguer :

- d'une part la **preuve de la négation** de A :

« Supposons A [vrai]. Alors (...), ce qui est absurde. Donc A est faux [i.e., $\neg A$]. »

qui est valable intuitionnistement : c'est prouver $A \Rightarrow \bot$ par la règle $\Rightarrow$Intro,

- et la **preuve par l'absurde** de A d'autre part :

« Supposons A faux [i.e., $\neg A$]. Alors (...), ce qui est absurde. Donc A est vrai. »

qui n'est pas valable intuitionnistement : tout ce qu'on peut en tirer est $\neg\neg A$.

- On peut lire $\neg A$ comme « A est faux » et $\neg\neg A$ comme « A n'est pas faux ».
- Noter que $\neg\neg\neg A$ équivaut à $\neg A$ (donc $\neg\neg\neg\neg A$ à $\neg\neg A$, etc.).

Logique classique

La logique **classique** ou **booléenne** modifie la règle

$$\bot\text{Élim}\ \frac{\Gamma \vdash \bot}{\Gamma \vdash Q} \qquad \text{en} \qquad \text{Absurde}\ \frac{\Gamma, \neg Q \vdash \bot}{\Gamma \vdash Q}$$

Elle est donc plus forte (= a plus de tautologies) que la logique intuitionniste.

- **Théorème** (« complétude de la sémantique booléenne du calcul propositionnel classique ») : P est une tautologie de la logique classique ssi pour toute substitution de $\bot$ ou $\top$ à chacune des variables propositionnelles de P a la valeur de vérité $\top$. (Voir transp. 97 plus loin pour précisions.)

Tableaux de vérité :

$\wedge$	$\bot$	$\top$
$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$

$\vee$	$\bot$	$\top$
$\bot$	$\bot$	$\top$
$\top$	$\top$	$\top$

$A \Rightarrow B$	$B = \bot$	$B = \top$
$A = \bot$	$\top$	$\top$
$A = \top$	$\bot$	$\top$

Un peu d'histoire des maths : Brouwer et l'intuitionnisme

- L'**intuitionnisme** est à l'origine une philosophie des mathématiques développée (à partir de 1912) par L. E. J. Brouwer (1881–1966) en réaction/opposition au **formalisme** promu par D. Hilbert (1862–1943).

Quelques principes de l'intuitionnisme *à l'origine* :

- rejet de la loi du tiers exclu (« principe d'omniscience ») pour demander des preuves constructives (pour Brouwer, montrer que x ne peut pas ne pas exister n'est pas la même chose que montrer que x existe),
- refus de la logique formelle (pour Brouwer, les maths sont « créatives »),
- principes de continuité (notamment : toute fonction $\mathbb{R} \to \mathbb{R}$ est continue).

- Idées rejetées par Hilbert (et la plupart des mathématiciens).

$\to$ « controverse Hilbert-Brouwer » (rapidement devenue querelle personnelle)

Hilbert : « Retirer au mathématicien le principe du tiers exclu serait comme empêcher à un boxeur d'utiliser ses poings. »

Les maths constructives après Brouwer

- L'intuitionnisme a été ensuite reformulé et modifié par les élèves de Brouwer et leurs propres élèves (notamment A. Heyting, A. Troelstra) et d'autres écoles (analyse constructive d'E. Bishop, constructivisme russe de A. Markov fils) :
  - dans le cadre de la logique formelle (ou compatible avec elle),
  - sans impliquer de principe *contredisant* les maths classiques,
  - mais toujours sans la loi du tiers exclu.

- Les termes de « **mathématiques constructives** » et « intuitionnisme » sont devenus plus ou moins interchangeables pour « maths sans le tiers exclu ».

- Dans *certains* tels systèmes, prouver $P \vee Q$ resp. $\exists x.P(x)$ passe forcément par la preuve de P ou de Q, resp. par la construction d'un x vérifiant $P(x)$.

- Les maths « normales » se font en logique classique, mais certains bouts (explicit[t] signalés comme tels) de la littérature sont en logique intuitionniste. On peut y faire de l'algèbre, de l'analyse, etc., constructives.

Pourquoi vouloir « boxer sans ses poings » ?

Évidence : l'écrasante majorité des maths se fait en logique classique (loi du tiers exclu admise) !

Une preuve « constructive » (sans tiers exclu) est plus restrictive qu'une preuve classique, donc apporte plus :

- principe de parcimonie ;
- curiosité purement théorique ;
- intérêt philosophique (pas de « principe d'omniscience ») ;
- validité dans un cadre mathématique plus large ($\rightarrow$ « topos ») ;
- lien avec les systèmes de typage (via Curry-Howard) ;
- extraction d'algorithme (p.ex., d'une preuve de $\forall m.\exists n.P(m,n)$ dans certains systèmes on peut extraire un algo qui calcule n à partir de m) ;
- compatibilité avec des axiomes qui contredisent les maths classiques (p.ex. « toute fonction $\mathbb{R} \rightarrow \mathbb{R}$ est continue », « toute fonction $\mathbb{N} \rightarrow \mathbb{N}$ est calculable » [$\leftarrow$ « calculabilité synthétique »]) qu'on peut vouloir étudier.

Preuves classiques pas toujours satisfaisantes

Que penser de la preuve suivante ?

Affirmation : Il existe un algo qui donné $k \in \mathbb{N}$ termine en temps fini et renvoie

- soit le rang de la première occurrence de k chiffres 7 dans l'écriture décimale de π,
- soit « ∞ » si une telle occurrence n'existe pas.

Preuve (classique !) : soit $f : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ la fonction qui à k associe le rang recherché.
De deux choses l'une :

- soit il existe k_0 tel que $f(k) = \infty$ pour $k \geq k_0$: alors f est calculable car f est déterminée par k_0 et un nombre *fini* de valeurs $(f(0), \ldots, f(k_0 - 1))$;
- soit $f(k)$ est fini pour tout $k \in \mathbb{N}$: alors f est calculable par l'algorithme qui, donné k, calcule indéfiniment des décimales de π jusqu'à en trouver k consécutives valant 7, et renvoie le rang (cet algorithme termine toujours par hypothèse).

Dans tous les cas f est calculable. □

Objection : cette preuve (correcte en maths classiques) ne nous donne pas du tout d'algorithme ! Elle montre juste qu'il « existe » classiquement.

L'interprétation de Brouwer-Heyting-Kolmogorov

Interprétation informelle/intuitive des connecteurs de la logique intuitionniste, due à A. Kolmogorov, A. Heyting, G. Kreisel, A. Troelstra et d'autres :

- un témoignage de $P \wedge Q$, est un témoignage de P et un de Q,
- un témoignage de $P \vee Q$, est un témoignage de P ou un de Q, et la donnée duquel des deux on a choisi,
- un témoignage de $P \Rightarrow Q$ est un moyen de transformer un témoignage de P en un témoignage de Q,
- un témoignage de $\top$ est trivial,
- un témoignage de $\bot$ n'existe pas,
- un témoignage de $\forall x.P(x)$ est un moyen de transformer un x quelconque en un témoignage de $P(x)$,
- un témoignage de $\exists x.P(x)$ est la donnée d'un certain x_0 et d'un témoignage de $P(x_0)$.

J'écris « témoignage », mais Kolmogorov parlait de « solution » d'un problème, Heyting de « preuve », etc.

Correspondance de Curry-Howard : implication seule

Typage du λ-calcul simplement typé	Calcul propositionnel intuitionniste
$\textsc{Var}\ \dfrac{}{\Gamma, x : \sigma \vdash x : \sigma}$	$\textsc{Ax}\ \dfrac{}{\Gamma, P \vdash P}$
$\textsc{App}\ \dfrac{\Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash fx : \tau}$	$\Rightarrow\textsc{Élim}\ \dfrac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}$
$\textsc{Abs}\ \dfrac{\Gamma, v : \sigma \vdash t : \tau}{\Gamma \vdash \lambda(v : \sigma).t : \sigma \to \tau}$	$\Rightarrow\textsc{Int}\ \dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$

- Ce sont exactement les mêmes règles, aux notations/nommage près...
- ...sauf que la colonne de droite n'a pas le terme typé, mais on sait qu'on peut le reconstruire (transp. 29), à partir de l'arbre de dérivation.
- On peut donc identifier termes du λCST et arbres de preuve en déduction naturelle du calcul propositionnel intuitionniste restreint au seul connecteur $\Rightarrow$.

Correspondance de Curry-Howard : exemple avec implication

- Transformons en démonstration le terme

$$\lambda(f : \beta \to \alpha \to \gamma).\lambda(x : \alpha).\lambda(y : \beta).fyx$$

qu'on a typé (transp. 25) comme $(\beta \to \alpha \to \gamma) \to (\alpha \to \beta \to \gamma)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B \Rightarrow A \Rightarrow C, A, B \vdash B \Rightarrow A \Rightarrow C}\,\text{Ax} \quad \dfrac{}{B \Rightarrow A \Rightarrow C, A, B \vdash B}\,\text{Ax}}{B \Rightarrow A \Rightarrow C, A, B \vdash A \Rightarrow C}\,\Rightarrow\text{Élim} \quad \dfrac{}{B \Rightarrow A \Rightarrow C, A, B \vdash A}\,\text{Ax}}{B \Rightarrow A \Rightarrow C, A, B \vdash C}\,\Rightarrow\text{Élim}}{B \Rightarrow A \Rightarrow C, A \vdash B \Rightarrow C}\,\Rightarrow\text{Int}}{B \Rightarrow A \Rightarrow C \vdash A \Rightarrow B \Rightarrow C}\,\Rightarrow\text{Int}}{\vdash (B \Rightarrow A \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)}\,\Rightarrow\text{Int}$$

Correspondance de Curry-Howard : conjonction

On veut étendre le λCST avec un **type produit** pour refléter les règles de la conjonction logique :

Typage du λ-calcul	Calcul propositionnel intuitionniste
$\textsc{Proj}_1 \dfrac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1 t : \tau_1}$	$\wedge\textsc{Élim}_1 \dfrac{\Gamma \vdash Q_1 \wedge Q_2}{\Gamma \vdash Q_1}$
$\textsc{Proj}_2 \dfrac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2 t : \tau_2}$	$\wedge\textsc{Élim}_2 \dfrac{\Gamma \vdash Q_1 \wedge Q_2}{\Gamma \vdash Q_2}$
$\textsc{Pair} \dfrac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \langle t_1, t_2 \rangle : \tau_1 \times \tau_2}$	$\wedge\textsc{Int} \dfrac{\Gamma \vdash Q_1 \qquad \Gamma \vdash Q_2}{\Gamma \vdash Q_1 \wedge Q_2}$

- Ici, $\langle —, — \rangle$ sert à construire des couples, et π_1, π_2 à les déconstruire.
- Nouvelle règle de réduction : $\pi_i \langle t_1, t_2 \rangle$ se réduit en t_i (pour $i \in \{1, 2\}$).

Correspondance de Curry-Howard : exemple avec conjonction

- Transformons en programme la démonstration qu'on a donnée de $A \wedge B \Rightarrow B \wedge A$ (transp. 37 et suivants) :

$$\dfrac{\dfrac{\dfrac{}{u : \alpha \times \beta \vdash u : \alpha \times \beta}\,\textsc{Var}}{u : \alpha \times \beta \vdash \pi_2 u : \beta}\,\textsc{Proj}_2 \qquad \dfrac{\dfrac{}{u : \alpha \times \beta \vdash u : \alpha \times \beta}\,\textsc{Var}}{u : \alpha \times \beta \vdash \pi_1 u : \alpha}\,\textsc{Proj}_1}{u : \alpha \times \beta \vdash \langle \pi_2 u, \pi_1 u \rangle : \beta \times \alpha}\,\textsc{Pair}$$
$$\dfrac{}{\vdash \lambda(u : \alpha \times \beta).\, \langle \pi_2 u, \pi_1 u \rangle : \alpha \times \beta \to \beta \times \alpha}\,\textsc{Abs}$$

- Il s'agit de la fonction $\lambda(u : \alpha \times \beta).\, \langle \pi_2 u, \pi_1 u \rangle$ (polymorphe de type $\alpha \times \beta \to \beta \times \alpha$) qui échange les deux termes d'un couple.

Correspondance de Curry-Howard : disjonction

On veut étendre le λCST avec un **type somme** pour refléter les règles de la disjonction logique :

Typage du λ-calcul	Calcul propositionnel intuitionniste
$\textsc{Inj}_1\ \dfrac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \iota_1^{(\tau_1,\tau_2)} t : \tau_1 + \tau_2}$	$\vee\textsc{Int}_1\ \dfrac{\Gamma \vdash Q_1}{\Gamma \vdash Q_1 \vee Q_2}$
$\textsc{Inj}_2\ \dfrac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \iota_2^{(\tau_1,\tau_2)} t : \tau_1 + \tau_2}$	$\vee\textsc{Int}_2\ \dfrac{\Gamma \vdash Q_2}{\Gamma \vdash Q_1 \vee Q_2}$
ci-dessous ↓	$\vee\textsc{Élim}\ \dfrac{\Gamma \vdash P_1 \vee P_2 \qquad \Gamma, P_1 \vdash Q \qquad \Gamma, P_2 \vdash Q}{\Gamma \vdash Q}$

$$\textsc{Match}\ \frac{\Gamma \vdash r : \sigma_1 + \sigma_2 \qquad \Gamma, v_1 : \sigma_1 \vdash t_1 : \tau \qquad \Gamma, v_2 : \sigma_2 \vdash t_2 : \tau}{\Gamma \vdash (\texttt{match}\ r\ \texttt{with}\ \iota_1 v_1 \mapsto t_1,\ \iota_2 v_2 \mapsto t_2) : \tau}$$

N.B. : v_1, v_2 sont des variables qui sont liées par le `match` ; et r, t_1, t_2 sont des termes.

Remarques sur types produits et sommes

- ι_1, ι_2 sont des *constructeurs* dans la terminologie OCaml.
- À côté de la β-réduction usuelle du λ-calcul, $(\lambda(v : \sigma).e)t \rightsquigarrow e[v\backslash t]$, on introduit des nouvelles règles de réduction pour la conjonction et la disjonction :
  - $\pi_i\langle t_1, t_2\rangle \rightsquigarrow t_i$ (pour $i \in \{1, 2\}$)
  - $(\texttt{match } \iota_i^{(\tau_1,\tau_2)} s \texttt{ with } \iota_1 v_1 \mapsto t_1,\ \iota_2 v_2 \mapsto t_2) \rightsquigarrow t_i[v_i\backslash s]$ (pour $i \in \{1, 2\}$)
- Côté démonstrations : cette réduction court-circuite une règle INTRO immédiatement suivie par sa règle ÉLIM (ce qu'on appelle un « détour »).
- Le λ-calcul simplement typé reste fortement normalisant avec ces extensions par types produits et sommes et les réductions ci-dessus (et types 1 et 0 après).
- Les injections $\iota_i : \tau_i \to \tau_1 + \tau_2$ portent l'exposant (τ_1, τ_2) pour que les annotations de type soient complètes (mais il est inutile dans le matching).
- Aucune des notations π_i, $\langle, \rangle$, ι_i, `match...with` n'est standardisée (bcp de variations existent), mais il n'y a aucun doute sur la correspondance elle-même.

Correspondance de Curry-Howard : exemple avec disjonction

- Transformons en programme la démonstration qu'on a donnée de $A \vee B \Rightarrow B \vee A$ (transp. 37 et suivants) :

$$\dfrac{\dfrac{\dfrac{}{u : \alpha + \beta \vdash u : \alpha + \beta}\ \textsc{Var}}{\dots \vdash \iota_2^{(\beta,\alpha)} v : \beta + \alpha}\ \textsc{Inj}_2 \quad \dfrac{\dfrac{}{\dots, v' : \beta \vdash v' : \beta}\ \textsc{Var}}{\dots \vdash \iota_1^{(\beta,\alpha)} v' : \beta + \alpha}\ \textsc{Inj}_1}{u : \alpha + \beta \vdash (\texttt{match}\ u\ \texttt{with}\ \iota_1 v \mapsto \iota_2^{(\beta,\alpha)} v,\ \iota_2 v' \mapsto \iota_1^{(\beta,\alpha)} v') : \beta + \alpha}\ \textsc{Match}$$
$$\vdash \lambda(u : \alpha + \beta).(\texttt{match}\ u\ \texttt{with}\ \iota_1 v \mapsto \iota_2^{(\beta,\alpha)} v,\ \iota_2 v' \mapsto \iota_1^{(\beta,\alpha)} v') : \alpha + \beta \to \beta + \alpha\ \textsc{Abs}$$

- Il s'agit de la fonction

$$\lambda(u : \alpha + \beta).(\texttt{match}\ u\ \texttt{with}\ \iota_1 v \mapsto \iota_2^{(\beta,\alpha)} v,\ \iota_2 v' \mapsto \iota_1^{(\beta,\alpha)} v')$$

(polymorphe de type $\alpha + \beta \to \beta + \alpha$) qui échange les deux cas d'une somme.

Correspondance de Curry-Howard : vrai et faux

On veut étendre le λCST avec un **type unité** et un **type vide** pour refléter les règles du vrai et du faux :

Typage du λ-calcul	Calcul propositionnel intuitionniste
$\textsc{Unit}\ \dfrac{}{\Gamma \vdash \bullet : 1}$	$\top\textsc{Int}\ \dfrac{}{\Gamma \vdash \top}$
$\textsc{Void}\ \dfrac{\Gamma \vdash r : 0}{\Gamma \vdash \texttt{exfalso}^{(\tau)} r : \tau}$	$\bot\textsc{Élim}\ \dfrac{\Gamma \vdash \bot}{\Gamma \vdash Q}$

- Ici, $\bullet$ désigne la valeur triviale de type unité (`()` en OCaml), et `exfalso` est un matching vide. (Notations pas standardisées du tout.)
- Pas de nouvelle règle de réduction à ajouter.

En OCaml :

```
type void = | ;; let exfalso = fun (r:void) -> match r with _ -> . ;;
```

Correspondance de Curry-Howard : exemples divers

- L'implication $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$ est démontrée par le terme (« combinateur S ») :

$$\lambda(x : \alpha \to \beta \to \gamma).\, \lambda(y : \alpha \to \beta).\, \lambda(z : \alpha).\, xz(yz)$$

- L'équivalence $(A \wedge B \Rightarrow C) \Leftrightarrow (A \Rightarrow B \Rightarrow C)$ est démontrée par les fonctions de « curryfication »

$$\lambda(f : \alpha \times \beta \to \gamma).\, \lambda(x : \alpha).\, \lambda(y : \beta).\, f\langle x, y\rangle$$

et « décurryfication »

$$\lambda(f : \alpha \to \beta \to \gamma).\, \lambda(z : \alpha \times \beta).\, f(\pi_1 z)(\pi_2 z)$$

- L'équivalence $A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$ est démontrée par les termes

$$\lambda(u : \alpha + (\beta \times \gamma)).\, (\texttt{match}\ u\ \texttt{with}\ \iota_1 v \mapsto \langle \iota_1^{(\alpha,\beta)} v, \iota_1^{(\alpha,\gamma)} v\rangle,\ \iota_2 w \mapsto \langle \iota_2^{(\alpha,\beta)}(\pi_1 w), \iota_2^{(\alpha,\gamma)}(\pi_2 w)\rangle)$$

et

$$\lambda(u : (\alpha + \beta) \times (\alpha + \gamma)).\, (\texttt{match}\ \pi_1 u\ \texttt{with}\ \iota_1 v \mapsto \iota_1^{(\alpha,\beta\times\gamma)} v,\ \iota_2 v' \mapsto (\texttt{match}\ \pi_2 u\ \texttt{with}\ \iota_1 w \mapsto \iota_1^{(\alpha,\beta\times\gamma)} w,\ \iota_2 w' \mapsto \iota_2^{(\alpha,\beta\times\gamma)}\langle v', w'\rangle))$$

Correspondance de Curry-Howard : exemple en OCaml

Reprise de l'équivalence $A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$ typée par OCaml :

```
let pi1 = fun (x,y) -> x ;; (* π1 *)
val pi1 : 'a * 'b -> 'a = <fun>
let pi2 = fun (x,y) -> y ;; (* π2 *)
val pi2 : 'a * 'b -> 'b = <fun>
type ('a, 'b) sum = Inj1 of 'a | Inj2 of 'b ;; (* α+β *)
type ('a, 'b) sum = Inj1 of 'a | Inj2 of 'b
fun u -> match u with Inj1 v -> (Inj1 v, Inj1 v)
| Inj2 w -> (Inj2 (pi1 w), Inj2 (pi2 w)) ;;
- : ('a, 'b * 'c) sum -> ('a, 'b) sum * ('a, 'c) sum = <fun>
fun u -> (match (pi1 u) with Inj1 v -> Inj1 v
| Inj2 v_ -> (match (pi2 u) with Inj1 w -> Inj1 w
| Inj2 w_ -> Inj2 (v_,w_))) ;;
- : ('a, 'b) sum * ('a, 'c) sum -> ('a, 'b * 'c) sum = <fun>
```

Les preuves ne sont pas uniques

La question de l'égalité est compliquée, on ne l'abordera guère.

- Une même proposition peut avoir des preuves (fonctionnellement) différentes.

Par exemple, les entiers de Church typés :

- $\overline{0}_\alpha := \lambda(f : \alpha \to \alpha).\lambda(x : \alpha).x$
- $\overline{1}_\alpha := \lambda(f : \alpha \to \alpha).\lambda(x : \alpha).fx$
- $\overline{2}_\alpha := \lambda(f : \alpha \to \alpha).\lambda(x : \alpha).f(fx)$ etc.

tous de type $(\alpha \to \alpha) \to (\alpha \to \alpha)$ prouvent tous $(A \Rightarrow A) \Rightarrow (A \Rightarrow A)$.

($\overline{0}$) : « [**(1)** Supposons $A \Rightarrow A$. [**(2)** Supposons A.] **(3)** On a $A \Rightarrow A$ par $\Rightarrow$Int de (2) dans (2).] **(4)** On a $(A \Rightarrow A) \Rightarrow (A \Rightarrow A)$ par $\Rightarrow$Int de (1) dans (3). »

($\overline{1}$) : « [**(1)** Supposons $A \Rightarrow A$. [**(2)** Supposons A. **(3)** On a A par $\Rightarrow$Élim sur (1) et (2).] **(4)** On a $A \Rightarrow A$ par $\Rightarrow$Int de (2) dans (3).] **(5)** On a $(A \Rightarrow A) \Rightarrow (A \Rightarrow A)$ par $\Rightarrow$Int de (1) dans (4). »

($\overline{2}$) : « [**(1)** Supposons $A \Rightarrow A$. [**(2)** Supposons A. **(3)** On a A par $\Rightarrow$Élim sur (1) et (2). **(4)** On a A par $\Rightarrow$Élim sur (1) et (3).] **(5)** On a $A \Rightarrow A$ par $\Rightarrow$Int de (2) dans (4).] **(6)** On a $(A \Rightarrow A) \Rightarrow (A \Rightarrow A)$ par $\Rightarrow$Int de (1) dans (5). »

Curry-Howard : récapitulation

- La correspondance de Curry-Howard permet d'identifier
  - types du λ-calcul simplement typé, éventuellement enrichi de constructions de types produit ($\times$), somme ($+$), trivial (1) et vide (0), et
  - propositions (=formules logiques) du calcul propositionnel intuitionniste avec pour connecteurs l'implication ($\Rightarrow$) et éventuellement la conjonction ($\wedge$), disjonction ($\vee$), vrai ($\top$) et faux ($\bot$) respectivement,

en identifant aussi

- termes ayant les types en question,
- preuves des propositions en question, dans le style « déduction naturelle », en calcul propositionnel intutionniste.

- Noter aussi : abstraction$\leftrightarrow$ouverture d'hypothèse ; application$\leftrightarrow$*modus ponens* ; variables$\leftrightarrow$hypothèses ; variables liées$\leftrightarrow$hypothèses déchargées.

- On se permettra maintenant parfois des abus de notation justifiés par Curry-Howard, p.ex., traiter $\rightarrow$ et $\Rightarrow$ comme interchangeables.

La négation et la double négation

Qu'est-ce qui correspond à la proposition $\neg P$ par Curry-Howard ?

C'est le type $\sigma \to 0$ des fonctions prenant un argument de type σ et renvoyant une valeur impossible, i.e., ne peuvent jamais renvoyer.

Mais on parle d'un langage (λCST enrichi) où tous les programmes terminent (« normalisation forte ») ! Donc une telle fonction prouve que σ est lui-même vide ; et la fonction est triviale : c'est une « pure preuve » de vacuité de σ :

- le type $\sigma \to 0$ (ou « $\neg\sigma$ ») est le type des « témoignages de vacuité » de σ (si on me fournit un truc de type σ, je soulève une exception parce que c'est impossible),
- le type $(\sigma \to 0) \to 0$ (ou « $\neg\neg\sigma$ ») est une sorte de type des témoignages de non-vacuité de σ,
- mais ce dernier ne permet pas « magiquement » d'en tirer une valeur :
- pas de terme de type $((\alpha \to 0) \to 0) \to \alpha$ ou bien $\alpha + (\alpha \to 0)$ dans le λCST : on est bien en logique intuitionniste.

Un embryon de polymorphisme

- Les types du λCST sont écrits avec des variables de types $\alpha, \beta, \gamma...$ En principe ce sont des types opaques. En pratique, une fonction comme $\lambda(x : \alpha).\lambda(y : \beta).x$ de type $\alpha \to \beta \to \alpha$ se comporte comme polymorphe : on peut imaginer un $\forall \alpha, \beta$ (implicite) devant :

- En effet, substituer n'importe quel type σ à une variable de type α dans un terme du λCST (enrichi) donne encore un terme correct (la dérivation de typage est la même, après substitution).

- Conséquence côté logique : substituer des propositions quelconques aux variables propositionnelles d'une tautologie donne encore une tautologie.

P.ex. : $(A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)$ est une tautologie, donc

$$((D \Rightarrow E) \wedge (E \Rightarrow D) \Rightarrow D) \Rightarrow ((D \Rightarrow E) \Rightarrow (E \Rightarrow D) \Rightarrow D)$$

en est (par subst$^{\text{ion}}$ de $D \Rightarrow E$ pour A, de $E \Rightarrow D$ pour B, et de D pour C).

- Attention, ce polymorphisme s'applique aux conclusions, pas aux hypothèses (l'hypothèse A ne permet pas de tout déduire !).

« Fonctorialité » des connecteurs logiques

- Donnés $t_1 : \tau_1 \to \tau_1'$ et $t_2 : \tau_2 \to \tau_2'$, le terme $\lambda(u : \tau_1 \times \tau_2).\langle t_1(\pi_1 u), t_2(\pi_2 u)\rangle$ est de type $\tau_1 \times \tau_2 \to \tau_1' \times \tau_2'$.
- Donnés $t_1 : \tau_1 \to \tau_1'$ et $t_2 : \tau_2 \to \tau_2'$, le terme $\lambda(u : \tau_1 + \tau_2).(\texttt{match}\ u\ \texttt{with}\ \iota_1 v_1 \mapsto \iota_1^{(\tau_1',\tau_2')}(t_1 v_1),\ \iota_2 v_2 \mapsto \iota_2^{(\tau_1',\tau_2')}(t_2 v_2))$ est de type $\tau_1 + \tau_2 \to \tau_1' + \tau_2'$.
- Donnés $s : \sigma' \to \sigma$ (attention au sens !) et $t : \tau \to \tau'$, le terme $\lambda(u : \sigma \to \tau).\lambda(x : \sigma').t(u(sx))$ est de type $(\sigma \to \tau) \to (\sigma' \to \tau')$.

Donc, par Curry-Howard :

- Si $Q_1 \Rightarrow Q_1'$ et $Q_2 \Rightarrow Q_2'$ alors $(Q_1 \wedge Q_2) \Rightarrow (Q_1' \wedge Q_2')$.
- Si $Q_1 \Rightarrow Q_1'$ et $Q_2 \Rightarrow Q_2'$ alors $(Q_1 \vee Q_2) \Rightarrow (Q_1' \vee Q_2')$.
- Si $P' \Rightarrow P$ (attention au sens !) et $Q \Rightarrow Q'$ alors $(P \Rightarrow Q) \Rightarrow (P' \Rightarrow Q')$.

- On dit que $\wedge$ et $\vee$ sont **croissants** (ou covariants) en leurs deux arguments, et que $\Rightarrow$ l'est dans son argument de droite, mais qu'il est **décroissant** (ou contravariant) dans son argument de gauche.

Sous-formules positives et négatives

Dans une formule propositionnelle, on définit les sous-formules **positives** (voire **strictement positives**) et **négatives** par induction :

- les sous-formules positives de $Q_1 \wedge Q_2$ et $Q_1 \vee Q_2$ sont la formule tout entière, et les sous-formules positives de Q_1 et celles de Q_2,
- les sous-formules négatives de $Q_1 \wedge Q_2$ et $Q_1 \vee Q_2$ sont les sous-formules négatives de Q_1 et celles de Q_2,
- les sous-formules positives de $P \Rightarrow Q$ sont la formule tout entière, les sous-formules positives de Q et les *négatives* de P,
- les sous-formules négatives de $P \Rightarrow Q$ sont les sous-formules négatives de Q et les *positives* de P.

▶ Les sous-formules strict[t] positives de $Q_1 \wedge Q_2$ et $Q_1 \vee Q_2$, resp. $P \Rightarrow Q$ sont la formule tout entière et les sous-formules strict[t] positives de Q_1 et de Q_2 (resp. Q).

P.ex. dans $$(A \Rightarrow (B \Rightarrow C)) \wedge ((D \Rightarrow E) \Rightarrow F)$$

les occurrences C, D, F sont positives (C et F strictement), tandis que A, B, E sont négatives.

« Fonctorialité » des formules

Conséquence de la fonctorialité des connecteurs logiques :

- Si S est une formule propositionnelle et S' obtenue en remplaçant une sous-formule positive Q par Q' telle que $Q \Rightarrow Q'$, alors $S \Rightarrow S'$.
- Si S est une formule propositionnelle et S' obtenue en remplaçant une sous-formule négative P par P' telle que $P' \Rightarrow P$, alors $S \Rightarrow S'$.

Comme toute sous-formule est soit positive soit négative, on en déduit :

- Corollaire : Si S est une formule propositionnelle et S' obtenue en remplaçant une sous-formule quelcone R par R' telle que $R \Leftrightarrow R'$, alors $S \Leftrightarrow S'$.

P.ex., on peut remplacer $Q_1 \wedge Q_2$ par $Q_2 \wedge Q_1$ ou $(Q_1 \wedge Q_2) \wedge Q_3$ par $Q_1 \wedge (Q_2 \wedge Q_3)$ n'importe où dans une formule et on obtient ainsi une formule équivalente.

Calcul des séquents

- Le **calcul des séquents** est une autre présentation de la (même) logique propositionnelle intuitionniste (mêmes tautologies, mêmes séquents).
- Cette présentation est peut-être moins « naturelle », mais elle est plus symétrique, et a divers intérêts théoriques.
- La différence principale porte sur les règles d'élimination (transp. suivant) : au lieu d'une règle d'introduction et d'une règle d'élimination, on a une **règle droite** (= introduction) et une **règle gauche** ($\leftrightarrow$ élimination), qui introduisent le connecteur à droite ou à gauche du symbole « $\vdash$ ».
- On garde la règle $\textsc{Ax}\ \dfrac{}{\Gamma, Q \vdash Q}$; on va reparler des règles de « contraction » et « coupure » :

$$\textsc{Contr}\ \frac{\Gamma, P, P \vdash Q}{\Gamma, P \vdash Q} \qquad \textsc{Cut}\ \frac{\Gamma \vdash P \qquad \Gamma, P \vdash Q}{\Gamma \vdash Q}$$

Calcul des séquents : règles (intuitionnistes) des connecteurs

La colonne de droite est la même que la colonne de gauche (intro) du transp. 36. C'est la colonne de gauche qui est « nouvelle » :

	Gauche	Droite
$\Rightarrow$	$\dfrac{\Gamma \vdash M \qquad \Gamma, P \vdash R}{\Gamma, M \Rightarrow P \vdash R}$	$\dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$
$\wedge$	$\dfrac{\Gamma, P_1 \vdash R}{\Gamma, P_1 \wedge P_2 \vdash R} \qquad \dfrac{\Gamma, P_2 \vdash R}{\Gamma, P_1 \wedge P_2 \vdash R}$	$\dfrac{\Gamma \vdash Q_1 \qquad \Gamma \vdash Q_2}{\Gamma \vdash Q_1 \wedge Q_2}$
$\vee$	$\dfrac{\Gamma, P_1 \vdash R \qquad \Gamma, P_2 \vdash R}{\Gamma, P_1 \vee P_2 \vdash R}$	$\dfrac{\Gamma \vdash Q_1}{\Gamma \vdash Q_1 \vee Q_2} \qquad \dfrac{\Gamma \vdash Q_2}{\Gamma \vdash Q_1 \vee Q_2}$
$\top$	(néant)	$\dfrac{}{\Gamma \vdash \top}$
$\bot$	$\dfrac{}{\Gamma, \bot \vdash R}$	(néant)

Exemples de démonstrations en calcul des séquents

On reprend les mêmes exemples que dans le transp. 37 :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}\,\mathrm{Ax}}{A \wedge B \vdash B}\,\wedge\mathrm{L}_2 \qquad \dfrac{\dfrac{}{A \vdash A}\,\mathrm{Ax}}{A \wedge B \vdash A}\,\wedge\mathrm{L}_1}{A \wedge B \vdash B \wedge A}\,\wedge\mathrm{R}}{\vdash A \wedge B \Rightarrow B \wedge A}\,\Rightarrow\mathrm{R} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\mathrm{Ax}}{A \vdash B \vee A}\,\vee\mathrm{R}_2 \qquad \dfrac{\dfrac{}{B \vdash B}\,\mathrm{Ax}}{B \vdash B \vee A}\,\vee\mathrm{R}_1}{A \vee B \vdash B \vee A}\,\vee\mathrm{L}}{\vdash A \vee B \Rightarrow B \vee A}\,\Rightarrow\mathrm{R}$$

Et que dans le transp. 50 :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \vdash B}\,\mathrm{Ax}}{B \Rightarrow A \Rightarrow C, A, B \vdash C}\,\Rightarrow\mathrm{L} \qquad \dfrac{\dfrac{}{C, A, B \vdash C}\,\mathrm{Ax}}{A \Rightarrow C, A, B \vdash C}\,\Rightarrow\mathrm{L}}{B \Rightarrow A \Rightarrow C, A, B \vdash C}\,\Rightarrow\mathrm{R}}{B \Rightarrow A \Rightarrow C, A \vdash B \Rightarrow C}\,\Rightarrow\mathrm{R}}{B \Rightarrow A \Rightarrow C \vdash A \Rightarrow B \Rightarrow C}\,\Rightarrow\mathrm{R}}{\vdash (B \Rightarrow A \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)}\,\Rightarrow\mathrm{R}$$

Calcul des séquents : règles « structurales »

- On peut choisir de séparer la règle $\textsc{Ax}\ \dfrac{}{\Gamma, Q \vdash Q}$ en deux, une règle d'axiome strict et une règle d'« affaiblissement » :

$$\textsc{Ax}\ \frac{}{Q \vdash Q} \quad \Big| \quad \textsc{Weak}\ \frac{\Gamma \vdash Q}{\Gamma, P \vdash Q}$$

- Comme en déduction naturelle, les hypothèses n'ont pas d'ordre.
- Si elles ont des multiplicités, la règle de contraction est nécessaire (sans elle, on ne peut pas prouver $(A \Rightarrow B) \wedge A \vdash B$) :

$$\textsc{Contr}\ \frac{\Gamma, P, P \vdash Q}{\Gamma, P \vdash Q}$$

On peut s'en dispenser en décidant que les hypothèses n'ont pas de multiplicité (forment un ensemble, pas un multi-ensemble), ou en modifiant les règles $\Rightarrow$L et $\wedge$L pour ne pas perdre d'hypothèse en remontant.

Calcul des séquents : équivalence avec la déduction naturelle

L'équivalence générale entre déduction naturelle et calcul des séquents n'est pas difficile si on utilise librement la règle de coupure.

Montrons l'exemple de la conversion entre ⇒Élim et ⇒Left :

- Dans le sens déduction naturelle → calcul des séquents :

Déduction naturelle	Calcul des séquents
$\Rightarrow\text{É}\textsc{lim}\ \dfrac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}$	$\textsc{Cut}\ \dfrac{\Gamma \vdash P \Rightarrow Q \qquad \Rightarrow\text{L}\ \dfrac{\Gamma \vdash P \qquad \textsc{Ax}\ \overline{\Gamma, Q \vdash Q}}{\Gamma, P \Rightarrow Q \vdash Q}}{\Gamma \vdash Q}$

- Dans le sens calcul des séquents → déduction naturelle :

Calcul des séquents	Déduction naturelle
$\Rightarrow\text{L}\ \dfrac{\Gamma \vdash M \qquad \Gamma, P \vdash R}{\Gamma, M \Rightarrow P \vdash R}$	$\Rightarrow\textsc{É}\textsc{lim}\ \dfrac{\Rightarrow\textsc{Intro}\ \dfrac{\Gamma, P \vdash R}{\Gamma \vdash P \Rightarrow R} \qquad \Rightarrow\textsc{É}\textsc{lim}\ \dfrac{\textsc{Ax}\ \overline{\Gamma, M \Rightarrow P \vdash M \Rightarrow P} \qquad \Gamma \vdash M}{\Gamma, M \Rightarrow P \vdash P}}{\Gamma, M \Rightarrow P \vdash R}$

Calcul des séquents : élimination des coupures

La règle de coupure exprime une forme de transitivité des démonstrations :

$$\textsc{Cut}\ \frac{\Gamma \vdash P \qquad \Gamma, P \vdash Q}{\Gamma \vdash Q}$$

(On dira que P est la « formule coupée ».)

- **Théorème** (Gentzen) : la règle de « coupure » n'est pas nécessaire en calcul des séquents : tout séquent prouvable avec elle est encore prouvable sans elle.

La preuve est même constructive et se fait par des transformations « locales ».

- En déduction naturelle, l'élimination des coupures est facile : pour éliminer une coupure il suffit de reprendre la démonstration de $\Gamma, P \vdash Q$ avec l'hypothèse P en moins, et d'utiliser celle de $\Gamma \vdash P$ partout où P est invoquée (cf. transp. 30). (La difficulté est plutôt l'élimination de « détours » Intro+Élim $\approx$ normal$^{\text{ion}}$.)

- En calcul des séquents, la difficulté supplémentaire est que P peut faire intervenir un connecteur qui est introduit à droite dans $\Gamma \vdash P$ et à gauche dans $\Gamma, P \vdash Q$ (cf. transp. 73).

Exemples d'étape d'élimination des coupures (1)

- Cas « déplaçants » : la dernière règle d'une des branches de la coupure n'opère pas sur la formule coupée : on fait remonter la coupure sur cette branche (quitte à choisir).

$$\textsc{Cut}\ \dfrac{\wedge\mathrm{R}\ \dfrac{\dfrac{\vdots}{\Gamma \vdash P_1} \quad \dfrac{\vdots}{\Gamma \vdash P_2}}{\Gamma \vdash P_1 \wedge P_2} \qquad \dfrac{\vdots}{\Gamma', P_1 \wedge P_2 \vdash Q'}}{\Gamma \vdash Q}\ ? \quad \text{devient} \quad \dfrac{\wedge\mathrm{R}\ \dfrac{\dfrac{\vdots}{\Gamma \vdash P_1} \quad \dfrac{\vdots}{\Gamma \vdash P_2}}{\Gamma \vdash P_1 \wedge P_2} \quad \textsc{Cut}\ \dfrac{\vdots}{\Gamma', P_1 \wedge P_2 \vdash Q'}}{\Gamma, \Gamma' \vdash Q}}{\Gamma \vdash Q}\ ?$$

(On n'a pas montré ici les règles structurales (contraction+affaiblissement) permettant d'avoir Γ et/ou Γ' comme hypothèses.)

- Cas final : une des branches de la coupure est la règle « axiome » sur la formule coupée : la coupure disparaît :

$$\textsc{Cut}\ \dfrac{\textsc{Ax}\ \dfrac{}{\Gamma \vdash P} \qquad \dfrac{\vdots}{\Gamma, P \vdash Q}}{\Gamma \vdash Q} \quad \text{devient} \quad \dfrac{\vdots}{\Gamma \vdash Q}$$

Exemples d'étape d'élimination des coupures (2)

- Cas « principaux » : la dernière règle de chaque branche de la coupure concerne le connecteur logique de la formule coupée : la coupure ne remonte pas et peut même se multiplier, mais le « degré » de complexité de la formule coupée diminue :

$$\textsc{Cut}\ \dfrac{\wedge\mathrm{R}\ \dfrac{\vdots\ \Gamma \vdash P_1 \qquad \vdots\ \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2} \qquad \dfrac{\vdots\ \Gamma, P_1 \vdash Q}{\Gamma, P_1 \wedge P_2 \vdash Q}\ \wedge\mathrm{L}_1}{\Gamma \vdash Q} \quad \text{devient} \quad \textsc{Cut}\ \dfrac{\vdots\ \Gamma \vdash P_1 \qquad \vdots\ \Gamma, P_1 \vdash Q}{\Gamma \vdash Q}$$

$$\textsc{Cut}\ \dfrac{\Rightarrow\mathrm{R}\ \dfrac{\vdots\ \Gamma, M \vdash P}{\Gamma \vdash M \Rightarrow P} \qquad \dfrac{\vdots\ \Gamma \vdash M \qquad \vdots\ \Gamma, P \vdash Q}{\Gamma, M \Rightarrow P \vdash Q}\ \Rightarrow\mathrm{L}}{\Gamma \vdash Q} \quad \text{devient} \quad \textsc{Cut}\ \dfrac{\vdots\ \Gamma \vdash M \qquad \dfrac{\vdots\ \Gamma, M \vdash P \qquad \vdots\ \Gamma, P \vdash Q}{\Gamma, M \vdash Q}\ \textsc{Cut}}{\Gamma \vdash Q}$$

Élimination des coupures : récurrence

- Le **degré** $\deg P$ d'une formule propositionnelle est : 0 pour une variable propositionnelle ou $\top$, $\bot$, et $\max(\deg P_1, \deg P_2) + 1$ si $P = P_1 \circ P_2$ pour un connecteur $\circ \in \{\Rightarrow, \wedge, \vee\}$. C'est donc la profondeur de l'arbre de cette formule. P.ex., $\deg(((A \Rightarrow B) \Rightarrow A) \Rightarrow A) = 3$.

- Le degré d'une coupure est le degré de la formule coupée. Le **degré de coupure** d'une démonstration en calcul des séquents est le plus grand degré d'une coupure (ou -1 s'il n'y en a pas). Une **coupure critique** est une coupure de degré maximal.

- La preuve de l'élimination des coupures se fait par une double récurrence :
  - récurrence principale sur le degré de coupure de la démonstration,
  - récurrence secondaire, à degré donné, sur la somme des hauteurs des deux branches de la coupure à éliminer.

- La démonstration est constructive (algorithmique) ; l'algorithme esquissé est p.r. (même « élémentaire »). Il n'est pas déterministe (ni même confluent).

Élimination des coupures : conséquences

Quel intérêt de pouvoir éliminer les coupures ? Les preuves sans coupure ne peuvent pas faire apparaître de formule inattendue.

- **Propriété de la disjonction :** si $\vdash Q_1 \vee Q_2$ (sans hypothèse !) alors $\vdash Q_1$ ou bien $\vdash Q_2$. (Preuve : une démonstration sans coupure de $\vdash Q_1 \vee Q_2$ doit finir par la règle $\vee R_1$ ou $\vee R_2$. □)
- **Corollaire :** $\vdash A \vee \neg A$ n'est pas prouvable en logique intuitionniste.
- **Propriété de la sous-formule :** si $P_1, \ldots, P_r \vdash Q$ alors une preuve sans coupure ne fait intervenir que des formules qui sont des sous-formules de $P_1, \ldots, P_r$ ou Q. (Preuve : c'est clair sur chacune des règles. □)
- **Corollaire :** on peut décider algorithmiquement si $P_1, \ldots, P_r \vdash Q$.

Algorithme : construire l'ensemble Φ de toutes les sous-formules d'une de $P_1, \ldots, P_r$ ou Q, et l'ensemble de tous les séquents possibles $\Gamma \vdash R$ avec $\Gamma \subseteq \Phi$ et $R \in \Phi$. Marquer comme « valables » ceux qui découlent de séquents déjà marqués comme valables par application d'une des règles du calcul des séquents (autre que la coupure), et répéter jusqu'à ce que le séquent recherché ait été marqué ou que plus aucun séquent ne soit marqué. □

Calcul des séquents et Curry-Howard

- On a vu que les preuves en déduction naturelle correspondent aux termes du λ-calcul simplement typé (+extensions).

On peut trouver un correspondant aux preuves en calcul des séquents (avec de petites modifications) : le $\overline{\lambda}$-calcul de Herbelin.

Le $\overline{\lambda}$-calcul a deux sortes d'objets : les *termes* t et les *listes d'arguments* (notées $[t_1; \ldots; t_r]$).

- En travaillant un peu, on peut déduire l'élimination des coupures de la conversion en $\overline{\lambda}$-calcul de termes normaux du λ-calcul (donc de sa normal[ion]).

$$\dfrac{\dfrac{}{y : C \vdash y : C} \quad \dfrac{}{x_1 : A_1 \vdash x_1 : A_1}}{\dfrac{y : C, x_1 : A_1 \vdash yx_1 : A_2 \to B}{y : C, x_1 : A_1, x_2 : A_2 \vdash yx_1x_2 : B}} \quad \dfrac{}{x_2 : A_2 \vdash x_2 : A_2} \quad \rightsquigarrow \quad \dfrac{\dfrac{}{x_1 : A_1 \vdash x_1 : A_1} \quad \dfrac{\dfrac{}{x_2 : A_2 \vdash x_2 : A_2} \quad \dfrac{}{\_ : B \vdash \_[] : B}}{x_2 : A_2; \_ : A_2 \to B \vdash \_[x_2] : B}}{\dfrac{x_1 : A_1, x_2 : A_2; \_ : C \vdash \_[x_1; x_2] : B}{y : C, x_1 : A_1, x_2 : A_2 \vdash y[x_1; x_2] : B}}$$

(où $C := A_1 \to A_2 \to B$)

(à gauche le λ-calcul, à droite le $\overline{\lambda}$-calcul ; remarquez l'inversion des arguments)

(y est une *variable* ici, pas une application ni une abstraction)

Combinateurs S, K, I

- **Axiomes de Hilbert :** le calcul prop$^{\text{nel}}$ intuitionniste peut être défini par la seule règle du modus ponens (« si $P \Rightarrow Q$ et P alors Q ») à partir des schémas d'axiomes suivants, où $A, B, C, \ldots$ sont remplacés par des formules qcqes :
  - (I) : $A \Rightarrow A$
  - (K) : $A \Rightarrow B \Rightarrow A$
  - (S) : $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$
  - $A \wedge B \Rightarrow A$
  - $A \wedge B \Rightarrow B$
  - $A \Rightarrow B \Rightarrow A \wedge B$
  - $\top$
  - $\bot \Rightarrow C$
  - $A \Rightarrow A \vee B$
  - $B \Rightarrow A \vee B$
  - $(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow (A \vee B \Rightarrow C)$

- Via Curry-Howard, ceci correspond au fait qu'on peut définir un λ-terme quelconque (à β-réduction près) par application des trois combinateurs $\mathsf{I} := \lambda x.x$, $\mathsf{K} := \lambda x.\lambda y.x$ et $\mathsf{S} := \lambda x.\lambda y.\lambda z.xz(yz)$. Ceci vaut pour le λ-calcul non typé comme simplement typé. (On n'a même pas besoin de I qui peut s'écrire SKK.)

- Idée de la conversion : remplacer $\lambda x.x$ par I, $\lambda x.y$ par Ky (si x n'apparaît pas dans y) et $\lambda x.uv$ par $\mathsf{S}(\lambda x.u)(\lambda x.v)$. (On peut aussi « η-convertir » $\lambda x.fx$ en f.) P.ex. : $\lambda f.\lambda x.f(fx)$ se réécrit $\lambda f.\mathsf{S}(\mathsf{K}f)f$ donc S(S(KS)K)I.

- On peut donc en théorie imaginer un langage de programmation fonctionnel Turing-complet sans variables, basé sur les seules fonctions S, K, I. (Mais personne ne serait assez fou pour faire ça.)

Qu'est-ce qu'une continuation ?

- Dans un langage de programmation, la **continuation** de l'appel d'une fonction f est « l'état de la machine qui attend que f renvoie une valeur ».

On peut y penser comme (une copie de) la pile d'appels jusqu'à l'appel de f.

- Certains langages donnent la citoyenneté de première classe aux continuations, i.e., permettent de les passer, stocker et invoquer explicitement :
  - capturer la contin$^{\text{ion}}$ de f revient à garder une copie de la pile d'appels de f,
  - invoquer la contin$^{\text{ion}}$ avec une valeur v a pour effet de faire retourner à f la valeur v (même si elle avait déjà terminé),
  - c'est une sorte de `goto` jusqu'au point à f termine, avec restauration de la pile (en C : `getcontext` pour capturer, `setcontext` pour restaurer).

- Variante : capturer la continuation revient à créer un fil d'exécution (*thread*) mis en attente au moment du retour de f, l'invoquer revient à terminer le fil actuel et réactiver le fil en attente, avec en lui transmettant v : il va de nouveau créer un fil en attente, puis considérer v comme valeur de retour de f.

- Fondement théorique : le $\lambda\mu$-calcul de Parigot (exten$^{\text{ion}}$ du λ-calcul avec continuations)

Applications des continuations

À quoi sert d'avoir des continuations réifiées (= de première classe) ?

- elles peuvent prendre la place d'un mécanisme d'exceptions ou de sortie de boucles (au lieu de soulever une exception, on invoque la continuation « exceptionnelle » pour abandonner le calcul normal),
- mais les continuations, contrairement aux exceptions, peuvent reprendre un calcul interrompu (si sa continuation a été capturée),
- elles permettent donc d'implémenter notamment : des générateurs ou du multitâche coopératif.

Le coût d'implémentation réside essentiellement dans la gestion de la pile :

- soit on recopie toute la pile à chaque capture/invocation de continuation,
- soit le séquencement d'appels cesse d'être une pile et doit être géré par le *garbage-collector* (= ramasse-miettes), c'est ce qui se passera avec CPS (cf. plus loin).

Informellement, les *continuations* sont aux appels de fonctions ce que les *clôtures* sont aux données locales.

La fonction call/cc

- « call/cc » = « call-with-current-continuation »
- La fonction call/cc existe dans plusieurs langages de programmation (notam^{t} : Scheme, SML/NJ, Ruby) : elle prend en argument une fonction g et
  - capture sa propre continuation (= celle du retour du call/cc),
  - passe celle-ci en argument de la fonction g, et renvoie soit la valeur de retour de g soit celle passée à la continuation.
- En Scheme, la continuation se présente comme une fonction, qu'on peut appeler avec un argument v :
  - la continuation elle-même ne termine jamais (puisque c'est, justement, une continuation : elle remplace la pile d'appels par celle qui a été capturée),
  - elle a pour effet de faire retourner v au call/cc qui l'a créée.
- En SML/NJ, la fonction s'appelle `SMLofNJ.Cont.callcc` et les continuations doivent être invoquées avec la fonction `throw` ; mais on peut facilement créer une fonction analogue à celle du Scheme :

```
val callcc = fn g => SMLofNJ.Cont.callcc (fn k => g (fn v =>
SMLofNJ.Cont.throw k v))
```

Exemples en Scheme

```
(define call/cc call-with-current-continuation)     ;; Shorthand
(call/cc (lambda (k) 42))                           ;; Return normally
→ 42
(call/cc (lambda (k) (k 42)))  ;; Return by invoking continuation
→ 42
(call/cc (lambda (k) (+ (k 42) 1)))      ;; (+ _ 1) never reached
→ 42
(* (call/cc (lambda (k) (k 42))) 2)         ;; Nothing weird here
→ 84
((call/cc (lambda (k) k))
(call/cc (lambda (k) k)))                   ;; Endless loop: why?
```

Très difficile à comprendre :

```
((lambda (yin)
((lambda (yang) (yin yang))
((lambda (kk) (display #\*) kk) (call/cc (lambda (k) k)))))
((lambda (kk) (newline) kk) (call/cc (lambda (k) k))))
```

Quel est le type de call/cc ?

- Une continuation ne retourne jamais. On peut donc la typer comme $\alpha \to \bot$ ou bien $\alpha \to \beta$ avec β un type arbitraire.
- La fonction g passée au call/cc doit renvoyer le même type α qu'accepte la continuation qu'on lui a passée : donc $(\alpha \to \beta) \to \alpha$.
- La fonction call/cc elle-même renvoie ce même type α : donc elle a pour type $((\alpha \to \beta) \to \alpha) \to \alpha$.
- C'est le type correspondant à la **loi de Peirce** $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$, une des formulations de la logique classique.

Moralité : la présence du call/cc transforme la logique du typage de logique intuitionniste en logique classique.

Ceci est dit de façon informelle, mais on peut introduire une variante du λ-calcul, le $\lambda\mu$-calcul, qui rend précise cette idée. Le $\lambda\mu$-calcul simplement typé garantit encore la terminaison des programmes : on ne peut pas faire de boucles avec le seul call/cc *bien typé*.

Oui mais il y a de la triche !

Le tiers exclu $A \vee \neg A$ dit moralement « soit je te donne une valeur de type α soit je te donne une promesse qu'il n'en existe pas (type $\alpha \to 0$) ». En λCST on ne peut pas faire ça !

Regardons comment le call/cc permet d'implémenter le tiers exclu $A \vee \neg A$:

$$\texttt{callcc}\left(\lambda(k : (\alpha + (\alpha \to 0)) \to 0).\ \iota_2^{(\alpha,\alpha\to 0)}\Big(\lambda(v : \alpha).\ k(\iota_1^{(\alpha,\alpha\to 0)} v)\Big)\right)$$

$$\text{a pour type :} \quad \alpha + (\alpha \to 0)$$

La fonction en argument du call/cc est a pour type $((\alpha + (\alpha \to 0)) \to 0) \to (\alpha + (\alpha \to 0))$, c'est-à-dire qu'elle correspond à une preuve de $(\neg(A \vee \neg A)) \Rightarrow (A \vee \neg A)$ (intuitionist$^{\text{t}}$ valable).

Que fait ce code ?

- il renvoie provisoirement $\iota_2^{(\alpha,\alpha\to 0)}(\cdots)$, où $(\cdots)$ définit une promesse qu'il n'y a pas de valeur de type α,
- si on invoque cette promesse (avec une valeur v de type α, donc !), il utilise la continuation k pour « revenir dans le temps » et changer d'avis et renvoie finalement $\iota_1^{(\alpha,\alpha\to 0)}(v)$.

Oui mais il y a de la triche : quelle est la morale ?

En SML/NJ :

```
val callcc = fn g => SMLofNJ.Cont.callcc (fn k => g (fn v =>
SMLofNJ.Cont.throw k v))
datatype ('a,'b) sum = Inj1 of 'a | Inj2 of 'b
val exclmiddle = callcc (fn k => Inj2 (fn v => k (Inj1 v))) ;;
```

*

Moralité :

- Il est facile de tenir ses promesses quand on peut voyager dans le temps avec l'aide de call/cc.
- Le call/cc change la logique du typage en logique classique, mais l'intérêt des types somme est grandement diminué : les valeurs ne sont pas stables.
- La logique classique n'a pas la propriété de la disjonction : on a $\vdash A \vee \neg A$ en logique classique, mais ni $\vdash A$ ni $\vdash \neg A$ (pour A opaque). Elle n'est pas constructive. On peut lui appliquer Curry-Howard, mais c'est moins intéressant.

Continuation Passing Style

Et si le langage n'a pas de call/cc ?

- Tous les langages n'ont pas de continuations de première classe (« réifiées »), mais dans un langage fonctionnel, on peut réécrire le code en « Continuation Passing Style » :
  - au lieu d'écrire des fonctions qui renvoient une valeur v, on les fait prendre en argument une autre fonction k, qui est une continuation-de-fait, et appellent cette fonction sur la valeur v ;
  - donc aucune fonction ne renvoie jamais rien : elle termine en invoquant son argument continuation-de-fait ou, plus souvent, en invoquant une autre fonction en lui passant une continuation-de-fait construite exprès pour continuer le calcul (ceci rend le style excessivement lourd et pénible) ;
  - ceci ne fonctionne bien que dans un langage supportant la récursion terminale propre (car tous les appels deviennent terminaux).
- C'est plus ou moins le concept des « promesses » de JavaScript, par exemple.

Continuation Passing Style : exemple en OCaml

```
let sum_cps = fun x -> fun y -> fun k -> k(x+y) ;;
val sum_cps : int -> int -> (int -> 'a) -> 'a = <fun>
let minus_cps = fun x -> fun y -> fun k -> k(x-y) ;;
val minus_cps : int -> int -> (int -> 'a) -> 'a = <fun>
let rec fibonacci_cps = fun n -> fun k -> if n <= 1 then k n
else minus_cps n 1 (fun n1 -> minus_cps n 2 (fun n2 ->
fibonacci_cps n1 (fun v1 -> fibonacci_cps n2 (fun v2 ->
sum_cps v1 v2 k)))) ;;
val fibonacci_cps : int -> (int -> 'a) -> 'a = <fun>
fibonacci_cps 8 (fun x -> x) ;;
- : int = 21
let callcc_cps = fun g -> fun k -> g (fun v -> fun k0 -> k v) k ;;
(* translation of: callcc (fun kf -> ((kf 42) + 1)) *)
callcc_cps (fun kf -> fun k -> kf 42 (fun v -> sum_cps v 1 k)) (fun x -> x) ;;
- : int = 42
```

Continuation Passing Style : systématisation

Définissons la transformation CPS de façon systématique.

Prenons ici les notations logiques pour les types : notamment, $\Rightarrow$ désigne le type fonction.

- Fixons un type Z de « retour ultime ». On pose $\sim P := (P \Rightarrow Z)$ pour le type d'« une continuation qui attend une valeur de type P » et $\sim\sim P$ pour « une valeur P passée par continuation ».
- Noter que $x : P \vdash \lambda(k : \sim P).kx : \sim\sim P$ (transformation d'une valeur « directe » en valeur passée par continuation).

On si on préfère voir ça comme une fonction : $\lambda(x : P).\,\lambda(k : \sim P).\,kx$ a pour type $P \Rightarrow \sim\sim P$.

- On va passer toutes les valeurs par continuation : tous les types transformés prendront la forme $\sim\sim T$.

Mais en plus de ça, les fonctions renvoient leur valeur par continuation : une fonction de type $P \Rightarrow Q$ va devenir $P \Rightarrow \sim Q \Rightarrow Z$, c'est-à-dire $P \Rightarrow \sim\sim Q$, et sera elle-même passée par continuation, donc $\sim\sim(P \Rightarrow \sim\sim Q)$ (sans compter que P et Q peuvent eux-mêmes changer).

Continuation Passing Style : l'essence de la transformation

- Définissons la transformation CPS d'abord dans le λ-calcul non typé : par induction sur la complexité du terme :
  - $v^{\mathrm{CPS}} = \lambda k.kv$ si v est une variable (c'est la transformation de P en $\sim\sim P$ définie ci-dessus).
  - $(\lambda v.t)^{\mathrm{CPS}} = \lambda k.\, k(\lambda v.\, t^{\mathrm{CPS}})$ (idem pour une fonction, dont le corps est CPS-ifié).
  - pour l'application :
    $$(fx)^{\mathrm{CPS}} = \lambda k.\, f^{\mathrm{CPS}}(\lambda f_0.\, x^{\mathrm{CPS}}(\lambda x_0.\, f_0 x_0 k))$$
    ce code se comprend ainsi : on invoque f^{CPS} pour recevoir sa valeur « directe » f_0 (qui est quand même une fonction dans le style CPS), puis x^{CPS} pour recevoir sa valeur « directe » x_0, puis on appelle la fonction f_0 avec la valeur x_0 et la continuation k de l'ensemble de l'expression,
  - $\texttt{callcc}^{\mathrm{CPS}} = \lambda \ell.\, \ell(\lambda g.\, \lambda k.\, g(\lambda v.\, \lambda k_0.\, kv)\, k)$

- Ce code porte les graines d'un interpréteur (eval/apply) du λ-calcul dans le λ-calcul : il impose d'ailleurs l'évaluation en appel-par-valeurs.

Continuation Passing Style : transformation des types

Il reste à typer tout ça !

- Rappelons qu'on a fixé Z et posé $\sim P := (P \Rightarrow Z)$. On définit une transformation $P \mapsto P^\diamond$ des types (=propositions) par (inductivement) :
  - $A^\diamond = A$ si A est une variable de type,
  - $(P \Rightarrow Q)^\diamond = (P^\diamond \Rightarrow \sim\sim Q^\diamond)$,
  - $(P \wedge Q)^\diamond = P^\diamond \wedge Q^\diamond$,
  - $(P \vee Q)^\diamond = P^\diamond \vee Q^\diamond$,
  - $\top^\diamond = \top$,
  - $\bot^\diamond = \bot$.
- On pose enfin $P^{\mathrm{CPS}} := \sim\sim P^\diamond$ (comprendre : $\sim\sim(P^\diamond)$).

En gros :

- $P^\diamond$ est le type P dans lequel toutes les fonctions ont été réécrites en style CPS (= renvoient leurs valeurs par continuation),
- $P^{\mathrm{CPS}} := \sim\sim P^\diamond$ est le type en question lui-même passé par continuation.

- Noter : $(P \Rightarrow Q)^{\mathrm{CPS}} = \sim\sim(P^\diamond \Rightarrow \sim\sim Q^\diamond) = \sim\sim(P^\diamond \Rightarrow Q^{\mathrm{CPS}})$.

Continuation Passing Style : transformation des termes

- On définit maintenant la transformation $t \mapsto t^{\mathrm{CPS}}$ sur les termes du λCST de manière à ce que si $\vdash t : P$ alors $\vdash t^{\mathrm{CPS}} : P^{\mathrm{CPS}}$ (rappel : $\sim P := (P \Rightarrow Z)$ et $P^{\mathrm{CPS}} := \sim\sim P^{\diamond}$) :
  - $v^{\mathrm{CPS}} = \lambda(k : \sim P^{\diamond}).\, kv$ lorsque v est une variable de type P,
  - $(fx)^{\mathrm{CPS}} = \lambda(k : \sim Q^{\diamond}).\, f^{\mathrm{CPS}}(\lambda(f_0 : P^{\diamond} \Rightarrow Q^{\mathrm{CPS}}).\, x^{\mathrm{CPS}}(\lambda(x_0 : P^{\diamond}).\, f_0 x_0 k))$ lorsque $f : P \Rightarrow Q$ et $x : Q$ (rappel : $(P \Rightarrow Q)^{\mathrm{CPS}} = \sim\sim(P^{\diamond} \Rightarrow Q^{\mathrm{CPS}})$),
  - $(\lambda(v : P).t)^{\mathrm{CPS}} = \lambda k.\, k(\lambda(v : P^{\diamond}).\, t^{\mathrm{CPS}})$ lorsque $\Gamma, v : P \vdash t : Q$,
  - $\langle x, y\rangle^{\mathrm{CPS}} = \lambda(k : \sim(Q_1^{\diamond} \wedge Q_2^{\diamond})).\, x^{\mathrm{CPS}}(\lambda(x_0 : Q_1^{\diamond}).\, y^{\mathrm{CPS}}(\lambda(y_0 : Q_2^{\diamond}).\, k\langle x_0, y_0\rangle))$,
  - $(\pi_i z)^{\mathrm{CPS}} = \lambda(k : \sim Q_i^{\diamond}).\, z^{\mathrm{CPS}}(\lambda(z_0 : Q_1^{\diamond} \wedge Q_2^{\diamond}).\, k(\pi_i z_0))$ (pour $i \in \{1, 2\}$),
  - $(\iota_i^{(Q_1,Q_2)} z)^{\mathrm{CPS}} = \lambda(k : \sim(Q_1^{\diamond} \vee Q_2^{\diamond})).\, z^{\mathrm{CPS}}(\lambda(z_0 : Q_i^{\diamond}).\, k(\iota_i^{(Q_1^{\diamond},Q_2^{\diamond})} z_0))$ (pour $i \in \{1, 2\}$),
  - $(\texttt{match}\ r\ \texttt{with}\ \iota_1 v_1 \mapsto t_1,\ \iota_2 v_2 \mapsto t_2)^{\mathrm{CPS}} = \lambda(k : \sim Q^{\diamond}).\, r^{\mathrm{CPS}}(\lambda(r_0 : P_1^{\diamond} \vee P_2^{\diamond}).\, (\texttt{match}\ r_0\ \texttt{with}\ \iota_1 v_1 \mapsto t_1^{\mathrm{CPS}} k,\ \iota_2 v_2 \mapsto t_2^{\mathrm{CPS}} k))$
  - $\bullet^{\mathrm{CPS}} = \lambda(k : \sim\top).k\bullet$
  - $(\texttt{exfalso}^{(Q)} r)^{\mathrm{CPS}} = \lambda(k : \sim Q^{\diamond}).\, r^{\mathrm{CPS}}(\lambda(r_0 : \bot).\, (\texttt{exfalso}^{(Q^{\diamond})} r_0))$
- $\texttt{callcc}^{\mathrm{CPS}} = \lambda(\ell : \cdots).\, \ell(\lambda(g : (P^{\diamond} \Rightarrow Q^{\mathrm{CPS}}) \Rightarrow P^{\mathrm{CPS}}).\, \lambda(k : \sim P^{\diamond}).\, g(\lambda(v : P^{\diamond}).\, \lambda(k_0 : \sim Q^{\diamond}).\, kv)\, k)$, de type $(((P \Rightarrow Q) \Rightarrow P) \Rightarrow P)^{\mathrm{CPS}}$.

Continuation Passing Style : remarques informatiques

- Programmer en CPS fait disparaître l'utilisation de la pile : tous les appels de fonctions deviennent terminaux, donc seront remplacés par des sauts dans un langage supportant la récursion terminale propre.
- La conversion vers CPS fixe l'ordre d'évaluation des valeurs. Par exemple, on a défini $\langle x, y\rangle^{\text{CPS}} = \lambda k.\, x^{\text{CPS}}(\lambda x_0.\, y^{\text{CPS}}(\lambda y_0.\, k\langle x_0, y_0\rangle))$ qui évalue x avant y, mais on pouvait aussi $\lambda k.\, y^{\text{CPS}}(\lambda y_0.\, x^{\text{CPS}}(\lambda x_0.\, k\langle x_0, y_0\rangle))$ pour inverser l'ordre.
- La conversion vers CPS fixe même une stratégie d'évaluation : on a choisi « call-by-value » ici, mais on aurait pu prendre « call-by-name » :

« call-by-value »	« call-by-name »
$v^{\text{CPS}} = \lambda k.kv$	$v^{\text{CPS}} = \lambda k.vk$ (ou juste v)
	mais pour une *valeur* : $x^{\text{CPS}} = \lambda k.kx$
$(\lambda v.t)^{\text{CPS}} = \lambda k.\, k(\lambda v.\, t^{\text{CPS}})$	$(\lambda v.t)^{\text{CPS}} = \lambda k.\, k(\lambda v.\, t^{\text{CPS}})$
$(fx)^{\text{CPS}} = \lambda k.\, f^{\text{CPS}}(\lambda f_0.\, x^{\text{CPS}}(\lambda x_0.\, f_0 x_0 k))$	$(fx)^{\text{CPS}} = \lambda k.\, f^{\text{CPS}}(\lambda f_0.\, f_0 x^{\text{CPS}} k)$
$(P \Rightarrow Q)^\diamond = (P^\diamond \Rightarrow \sim\sim Q^\diamond)$	$(P \Rightarrow Q)^\diamond = (\sim\sim P^\diamond \Rightarrow \sim\sim Q^\diamond)$

Continuation Passing Style : remarques informatiques (suite)

- Le CPS donne le call/cc « pour rien ». En fait, en style CPS, chaque fonction a contrôle complet sur l'exécution de tout le programme (il n'y a plus de pile !).
- Le CPS peut servir de point de départ pour la compilation (Appel, *Compiling with Continuations*, 1992).
- On peut concevoir un langage où chaque fonction a p.ex. deux continuations : une « continuation de succès » et une « continuation d'échec », qu'on chaîne avec des opérateurs « et » et « ou » : c'est essentiellement le Prolog (où ces opérateurs sont notés ‘,’ et ‘;’).
- Le CPS est utile pour séquencer explicit^{t} les évaluations. Voir notamment les promesses de JavaScript, et la monade `Control.Monad.Cont` de Haskell.

Continuation Passing Style : remarques logiques

- Par la transformation $t \mapsto t^{\mathrm{CPS}}$ et le fait d'avoir trouvé un terme de type $(((P \Rightarrow Q) \Rightarrow P) \Rightarrow P)^{\mathrm{CPS}}$, on a montré que :

$$\text{si CPC} \vdash P \text{ alors IPC} \vdash P^{\mathrm{CPS}}$$

où « IPC $\vdash P$ » signifie « P est prouvable en logique intuitionniste » et « CPC $\vdash P$ » signifie « P est prouvable en logique classique ».

- Si on prend $Z := \bot$, la proposition P^{CPS} ajoute simplement des $\neg\neg$ un peu partout, ce qui est équivalent en logique classique, donc on a évidemment

$$\text{CPC} \vdash P \text{ ssi CPC} \vdash P^{\mathrm{CPS}}$$

Comme par ailleurs CPC $\vdash Q$ implique IPC $\vdash Q$, on déduit des deux affirmations ci-dessus que (pour $Z = \bot$) :

$$\text{CPC} \vdash P \text{ ssi IPC} \vdash P^{\mathrm{CPS}}$$

On dit qu'on a « interprété » la logique (propositionnelle) classique en logique (propositionnelle) intuitionniste par la traduction « double négation ».

- En fait, on peut se contenter de mettre un $\neg\neg$ devant les disjonctions et formules atomiques (traduction de Gödel-Gentzen), ou bien (en calcul propositionnel !) d'en mettre un seul devant toute la formule (traduction de Glivenko).

Qu'est-ce qu'une « sémantique » en logique ?

De façon très (trop ?) vague :

- Une **logique** L est définie par un certain nombre de règles (axiomes, modes d'inférence) syntactique opérant sur des « formules » et définit une notion de **théorème** comme les formules ayant une **preuve** selon ces règles. On note (qqch comme) $\mathsf{L} \vdash \varphi$ pour « φ est un théorème [= est démontrable] selon L ».

- Un **modèle** pour L est une structure mathématique générale qui donne un « sens » au symboles utilisés dans L et qui notamment déclare certaines formules comme **vraies** (noté $\mathcal{M} \vDash \varphi$). Une **sémantique** est un ensemble de modèles (formés sur le même schéma). On dit que la sémantique S **valide** une formule φ lorsque φ est vraie dans chacun de ses modèles (on note $\mathsf{S} \vDash \varphi$).

On dit que

- S est **correcte** pour L lorsque $\mathsf{L} \vdash \varphi$ implique $\mathsf{S} \vDash \varphi$ (« tout théorème est vrai dans tout modèle »). On veut toujours ça !
- S est **complète** pour L lorsque $\mathsf{S} \vDash \varphi$ implique $\mathsf{L} \vdash \varphi$ (« toute formule vraie dans tout modèle est un théorème »).

Exemples de sémantiques

- La sémantique des **tableaux de vérité booléens** est correcte et complète pour le calcul propositionnel classique : φ est démontrable ssi toute affectation de « vrai » ou « faux » à chacune de ses variables donne « vrai ».

  Pour le calcul propositionnel intuitionniste, elle n'est que correcte.

- Le plan réel $\mathbb{R}^2$ est un modèle des axiomes de la géométrie euclidienne. Ce modèle est une sémantique correcte et complète à lui seul : un énoncé géométrique est démontrable à partir des axiomes ssi il est vrai dans $\mathbb{R}^2$.

- $\mathbb{N}$ est un modèle de ce qu'on appellera l'« **arithmétique de Peano du premier ordre** » PA (c'est le « modèle souhaité » de PA). Ce modèle pris tout seul est une sémantique correcte mais non complète (théorème d'incomplétude de Gödel : il y a des énoncés vrais dans $\mathbb{N}$ mais non démontrables dans PA).

- En revanche, si on élargit les modèles aux « modèles du premier ordre » (« modèles au sens de Tarski »), la sémantique devient complète (théorème de complétude de Gödel pour la logique du premier ordre : tout énoncé vrai dans tout modèle est démontrable).

Sémantiques du calcul propositionnel intuitionniste ?

- On a vu les règles syntaxiques du calcul propositionnel intuitionniste (déduction naturelle, ou calcul des séquents). Mais quel est le **sens** des connecteurs ?
  - En logique classique, c'est facile : la sémantique des tableaux de vérité est correcte et complète, donc on peut considérer qu'elle définit le sens.
  - En logique intuitionniste, on n'a donné pour l'instant qu'une interprétation intuitive (Brouwer-Heyting-Kolmogorov) des connecteurs.
- Avoir une sémantique (correcte) complète, ou « moins incomplète » que celle des tableaux de vérité permet de prouver qu'une formule logique n'est pas démontrable (si elle n'est pas validée par cette sémantique).
- On peut considérer Curry-Howard comme une sémantique : le « sens » de $\Rightarrow, \wedge, \vee, \top, \bot$ est donné par les opérations sur les types d'un langage de programmation, et les énoncés validés par le modèle sont ceux dont le type est habité. Elle est complète si le langage est le λCST. Mais elle est peu maniable et/ou un peu triviale !

Sémantique 0 : tableaux de vérité

(Reprise de ce qui a déjà été dit.)

- On définit $\mathbb{B} := \{0, 1\}$, $\dot{\top} = 1$, $\dot{\bot} = 0$ et $\dot{\wedge}, \dot{\vee}, \dot{\Rightarrow} : \mathbb{B}^2 \to \mathbb{B}$ par les tableaux de vérité usuels :

$\dot{\wedge}$	0	1
0	0	0
1	0	1

$\dot{\vee}$	0	1
0	0	1
1	1	1

$A \dot{\Rightarrow} B$	$B = 0$	$B = 1$
$A = 0$	1	1
$A = 1$	0	1

- Si φ est une formule propositionnelle en r variables, ceci définit une fonction $\dot{\varphi} : \mathbb{B}^r \to \mathbb{B}$ par composition.
- Un **modèle booléen** du calcul propositionnel est une affectation $\{\text{variables}\} \to \mathbb{B}$: chaque formule φ a donc une valeur de vérité (0 ou 1) dans le modèle, donnée par $\dot{\varphi}$. On dit que $\mathcal{M} \vDash \varphi$ lorsque c'est 1.
- On dit que la sémantique booléenne valide φ lorsque $\mathcal{M} \vDash \varphi$ pour tout modèle (i.e., $\dot{\varphi}$ vaut constamment 1).

Correction et complétude classique des tableaux de vérité

Théorème : la sémantique booléenne est correcte et complète pour le calcul propositionnel classique.

Esquisse de preuve :

- Correction : on montre par induction sur la preuve que si $\eta_1, \ldots, \eta_r \vdash \varphi$ alors $\dot{\varphi} = 1$ dans tout modèle où $\dot{\eta}_1 = \cdots = \dot{\eta}_r = 1$: la vérification est très facile sur chaque règle du calcul propositionnel.

- Complétude : on démontre classiquement $\vdash A_i \vee \neg A_i$ pour chaque variable A_i, puis on utilise l'élimination du $\vee$ pour se placer dans chacun des 2^r cas possibles (i.e., avec A_i ou $\neg A_i$ dans les hypothèses). Dans chaque cas on a $A_i \Leftrightarrow \top$ ou $A_i \Leftrightarrow \bot$ pour chaque variable, donc en suivant les tableaux de vérité on a φ (vraie). L'élimination du $\vee$ montre alors φ vraie. □

Intuitionnistement, la correction vaut toujours, mais plus la complétude ($A \vee \neg A$ n'est pas démontrable).

Sémantique 1 : cadres de Kripke

Définitions :

- un **cadre de Kripke** est (dans ce contexte) un ensemble partiellement ordonné $(W, \leq)$; les éléments de W s'appellent les **mondes** ; si $w \leq w'$, on dit que w' est **accessible** depuis w ;
- dans un cadre de Kripke, une **affectation de vérité** est une fonction $p \colon W \to \mathbb{B}$ (où $\mathbb{B} = \{0, 1\}$) telle que si $p(w) = 1$ et $w \leq w'$ alors $p(w') = 1$ (i.e., p est croissante, on dit aussi « permanente ») ;
- on définit des opérations $\dot{\wedge}, \dot{\vee}, \dot{\Rightarrow}$ sur les affectations de vérité :
  - $(q_1 \mathbin{\dot{\wedge}} q_2)(w) = 1$ ssi $q_1(w) = 1$ et $q_2(w) = 1$;
  - $(q_1 \mathbin{\dot{\vee}} q_2)(w) = 1$ ssi $q_1(w) = 1$ ou $q_2(w) = 1$;
  - $(q_1 \mathbin{\dot{\Rightarrow}} q_2)(w) = 1$ ssi pour tout $w' \geq w$ t.q. $q_1(w') = 1$, on a $q_2(w') = 1$;
  - $\dot{\top}(w) = 1$ partout ; ▶ $\dot{\bot}(w) = 0$ partout.
- un **modèle de Kripke** est (dans ce contexte) un cadre de Kripke $(W, \leq)$ muni d'une affectation de vérité pour chaque variable propositionnelle ;
- φ est **validée** par le modèle de Kripke lorsque $\dot{\varphi}$ vaut 1 dans tout monde w.

Cadres de Kripke (suite)

Recopie : ▶ $(q_1 \mathbin{\dot\wedge} q_2)(w) = 1$ ssi $q_1(w) = 1$ et $q_2(w) = 1$; ▶ $(q_1 \mathbin{\dot\vee} q_2)(w) = 1$ ssi $q_1(w) = 1$ ou $q_2(w) = 1$; ▶ $(q_1 \mathbin{\dot\Rightarrow} q_2)(w) = 1$ ssi pour tout $w' \geq w$ t.q. $q_1(w') = 1$, on a $q_2(w') = 1$.

Par exemple :

- $\dot\neg p$ (c'est-à-dire $p \mathbin{\dot\Rightarrow} \dot\bot$) vaut 1 dans le monde w lorsque p vaut 0 dans tout monde accessible depuis w ;
- $p \mathbin{\dot\vee} \dot\neg p$ vaut 1 dans le monde w lorsque p vaut 0 dans tout monde accessible depuis w ou bien 1 dans w (donc dans tout monde accessible depuis w, par permanence) : p est « décidé » à vrai ou à faux ;
- $\dot\neg\dot\neg p$ vaut 1 dans le monde w lorsque $\forall w' \geq w.\, \exists w'' \geq w'.\, p(w'') = 1$ (« dans tout futur possible, p finit par devenir vrai »).

▶ La sémantique de Kripke modélise plus ou moins l'intuition selon laquelle « $A \Rightarrow B$ » ne signifie pas juste « soit A est vrai soit B est faux » (sens en logique classique) mais bien « dans tout monde possible où A est vrai, B l'est ».

Sémantique des cadres de Kripke

Définitions (suite) :

- (déjà dit :) φ est **validée** par le modèle de Kripke (= valide dedans, = valable) lorsque $\dot{\varphi}$ vaut 1 dans tout monde w ;
- φ est **validée** par le cadre de Kripke lorsqu'elle est validée par toute affectation de vérité pour chacune de ses variables (= tout modèle sur ce cadre) ;
- φ est **validée** par la sémantique de Kripke lorsque φ est validée par tout cadre (i.e., tout modèle de Kripke).

Théorème (Kripke) : la sémantique de Kripke est correcte et complète pour le calcul propositionnel intuitionniste :

- correction : tout théorème propositionnel intuitionniste est valide dans tout modèle de Kripke,
- complétude : toute formule propositionnelle valide dans tout modèle de Kripke est démontrable en calcul propositionnel intuitionniste.

Cadres de Kripke : remarques

La sémantique de Kripke n'est complète que quand on considère tous les cadres. Si on se limite aux modèles sur un cadre donné, ils peuvent valider des formules non démontrables.

- Le cadre réduit à un singleton valide la logique classique : c'est exactement la sémantique booléenne ;
- Si $(W, \leq)$ est totalement ordonné, il valide la formule $(A \Rightarrow B) \vee (B \Rightarrow A)$ (preuve : sinon il y a un monde w dans lequel $p_A(w) = 1$ mais $p_B(w) = 0$ et un w' dans lequel $p_A(w') = 0$ mais $p_B(w') = 1$; mais si $w \leq w'$ ceci contredit la permanence de p_A et si $w \geq w'$ ceci contredit la permanence de p_B $\square$) qui n'est pas prouvable en logique propositionnelle intuitionniste.
- Le cadre le plus simple après un singleton est $\{u, v\}$ avec $u \leq v$, qui correspond à une logique à 3 valeurs de vérité, aux tableaux suivants.

p	u	v
0	0	0
q	0	1
1	1	1

$\dot\wedge$	0	q	1
0	0	0	0
q	0	q	q
1	0	q	1

$\dot\vee$	0	q	1
0	0	q	1
q	q	q	1
1	1	1	1

$A \dot\Rightarrow B$	$B = 0$	$B = \mathsf{q}$	$B = 1$
$A = 0$	1	1	1
$A = \mathsf{q}$	0	1	1
$A = 1$	0	q	1

Sémantique 2 : les ouverts en topologie

Fixons X un espace topologique : si on ne sait pas ce que c'est, penser à un espace métrique ou simplement $\mathbb{R}^m$ avec sa topologie ordinaire. On note $\mathcal{O}(X)$ la topologie de X, c'est-à-dire l'ensemble des ouverts de X.

On définit les opérations suivantes sur $\mathcal{O}(X)$:

- $U \dot\wedge V := U \cap V$ ▶ $U \dot\vee V := U \cup V$ ▶ $\dot\top := X$ ▶ $\dot\bot := \varnothing$
- $(U \dot\Rightarrow V) := \mathrm{int}((X \setminus U) \cup V)$ (où int désigne l'intérieur) est l'ensemble des points $x \in X$ tels qu'il y ait un ouvert $W \ni x$ t.q. $(U \cap W) \subseteq (V \cap W)$ (= les points « au voisinage desquels » $U \subseteq V$), ou, si on préfère, le plus grand ouvert W tel que $(U \cap W) \subseteq (V \cap W)$;
- notamment, $\dot\neg U := \mathrm{int}(X \setminus U)$, plus grand ouvert disjoint de U ;
- notamment, $\dot\neg\dot\neg U := \mathrm{int}(\mathrm{adh}(U))$ (intérieur de l'adhérence de U, ou « régularisé » de U).

- Si $\varphi(A_1, \ldots, A_r)$ est une formule propositionnelle et $U_1, \ldots, U_r \in \mathcal{O}(X)$, on définit $\dot\varphi(U_1, \ldots, U_r)$ de façon évidente (par induction).

Sémantique des ouverts : correction et complétude

De façon surprenante, les opérations qu'on a définies sur les ouverts de X fournissent un modèle du calcul propositionnel intuitionniste :

Théorème (Tarski) : la sémantique des ouverts est correcte et complète pour le calcul propositionnel intuitionniste :

- correction : si $\varphi(A_1, \ldots, A_r)$ est un théorème du calcul propositionnel intuitionniste, alors quel que soit X et $U_1, \ldots, U_r$ ouverts de X, on a $\dot{\varphi}(U_1, \ldots, U_r) = \dot{\top}$ (l'espace tout entier) ;
- complétude : réciproquement, si $\dot{\varphi}(U_1, \ldots, U_r) = \dot{\top}$ pour tous ouverts $U_1, \ldots, U_r$ de tout espace topologique X, ou même simplement de $\mathbb{R}^n$ pour un $n \geq 1$ donné, alors φ est démontrable en calcul propositionnel intuitionniste.

- La sémantique des ouverts modélise plus ou moins l'intuition que la vérité est une « notion locale » (si quelque chose est vrai en un point, c'est vrai autour de ce point).

Sémantique des ouverts : exemples

- Soit $U =]0,1[$ dans $X = \mathbb{R}$. Alors
  - $\dot{\neg} U =]-\infty, 0[\cup]1, +\infty[$
  - $\dot{\neg}\dot{\neg} U =]0,1[= U$
  - $(\dot{\neg}\dot{\neg} U \dot{\Rightarrow} U) = \mathbb{R}$
  - $U \dot{\vee} \dot{\neg} U = \mathbb{R} \setminus \{0,1\}$
  - $((\dot{\neg}\dot{\neg} U \dot{\Rightarrow} U) \dot{\Rightarrow} (U \dot{\vee} \dot{\neg} U)) = \mathbb{R} \setminus \{0,1\}$

Par correction, ceci montre que $((\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A))$ n'est pas prouvable en logique intuitionniste.

- Dans $X = \mathbb{R}^2$, soit $U = \{(x_1, x_2) : x_1 < 0 \text{ et } x_2 < 0\}$ et $V_1 = \{x_1 > 0\}$ et $V_2 = \{x_2 > 0\}$.
  - $\dot{\neg} U = \{(x_1, x_2) : x_1 > 0 \text{ ou } x_2 > 0\} = V_1 \dot{\vee} V_2$
  - Donc $(\dot{\neg} U \dot{\Rightarrow} (V_1 \dot{\vee} V_2)) = \mathbb{R}^2$
  - $(\dot{\neg} U \dot{\Rightarrow} V_1) = \{(x_1, x_2) : x_1 < 0 \text{ ou } x_2 > 0\}$
  - $(\dot{\neg} U \dot{\Rightarrow} V_2) = \{(x_1, x_2) : x_1 > 0 \text{ ou } x_2 < 0\}$
  - $(\dot{\neg} U \dot{\Rightarrow} V_1) \dot{\vee} (\dot{\neg} U \dot{\Rightarrow} V_2) = \mathbb{R}^2 \setminus \{(0,0)\}$

Ceci montre que $(\neg A \Rightarrow (B_1 \vee B_2)) \Rightarrow (\neg A \Rightarrow B_1) \vee (\neg A \Rightarrow B_2)$ (« axiome de Kreisel-Putnam ») n'est pas prouvable en logique intuitionniste (et *a fortiori* $(C \Rightarrow B_1) \vee (C \Rightarrow B_2) \Rightarrow (C \Rightarrow (B_1 \vee B_2))$ ne l'est pas).

Sémantique 3 : la réalisabilité propositionnelle

On reprend les notations de la calculabilité, mais on notera $\Phi_e : \mathbb{N} \dashrightarrow \mathbb{N}$ pour la e-ième fonction générale récursive (pour éviter un conflit de notation).

On définit les opérations suivantes sur l'ensemble $\mathcal{P}(\mathbb{N})$ des parties de $\mathbb{N}$:

- $P \mathbin{\dot\wedge} Q = \{\langle m, n\rangle : m \in P, n \in Q\}$ (codage de Gödel du produit $P \times Q$)
- $P \mathbin{\dot\vee} Q = \{\langle 0, m\rangle : m \in P\} \cup \{\langle 1, n\rangle : n \in Q\}$ (sorte de réunion disjointe)
- $(P \mathbin{\dot\Rightarrow} Q) = \{e \in \mathbb{N} : \Phi_e(P)\!\downarrow \subseteq Q\}$, ce qui signifie $\forall m \in P.\, \Phi_e(m)\!\downarrow \in Q$ (programmes définis sur tout P et l'envoyant dans Q)
- $\dot\top = \mathbb{N}$ ▶ $\dot\bot = \varnothing$
- notamment, $\dot\neg P$ vaut $\mathbb{N}$ si $P = \varnothing$ et $\varnothing$ sinon (« promesses » que $P = \varnothing$) ;
- notamment, $\dot\neg\dot\neg P$ vaut $\varnothing$ si $P = \varnothing$ et $\mathbb{N}$ sinon (« promesses » que $P \neq \varnothing$).

- Si $\varphi(A_1, \ldots, A_r)$ est une formule propositionnelle et $P_1, \ldots, P_r \subseteq \mathbb{N}$, on définit $\dot\varphi(P_1, \ldots, P_r)$ de façon évidente (par induction).
- Si $n \in \dot\varphi(P_1, \ldots, P_r)$, on dit aussi que n **réalise** $\varphi(P_1, \ldots, P_r)$.

La réalisabilité propositionnelle

- Les définitions de $\dot{\Rightarrow}, \dot{\wedge}, \dot{\vee}$ suivent précisément l'idée informelle de l'interprétation de Brouwer-Heyting-Kolmogorov (transp. 48) en les rendant précises avec des parties de $\mathbb{N}$ et des fonctions générales récursives.

- On dit qu'une formule propositionnelle $\varphi(A_1, \ldots, A_r)$ est **réalisable** (plus précisément, « uniformément réalisable ») lorsqu'il existe un même entier n qui réalise $\varphi(P_1, \ldots, P_r)$ quels que soient $P_1, \ldots, P_r \subseteq \mathbb{N}$:

$$n \in \bigcap_{P_1,\ldots,P_r} \dot{\varphi}(P_1, \ldots, P_r)$$

- **Théorème** (Dragalin, Troelstra, Nelson) : la sémantique de la réalisabilité est correcte pour le calcul propositionnel intuitionniste : si $\varphi(A_1, \ldots, A_r)$ est un théorème du calcul propositionnel intuitionniste, alors il existe n qui réalise $\varphi(P_1, \ldots, P_r)$ quels que soient $P_1, \ldots, P_r \subseteq \mathbb{N}$.

Mieux : ce n se construit à partir de la preuve de φ : c'est une forme d'extraction d'algorithme.

En fait on peut voir ça comme une conséquence de Curry-Howard.

La réalisabilité propositionnelle : exemple

- Essayons de réaliser $A \wedge B \Rightarrow B \wedge A$: on cherche donc un même entier dans $P \dot{\wedge} Q \dot{\Rightarrow} Q \dot{\wedge} P$ pour *tous* $P, Q \subseteq \mathbb{N}$.

- Autrement dit, on veut trouver un programme e tel que Φ_e soit défini sur $P \dot{\wedge} Q = \{\langle m, n\rangle : m \in P, n \in Q\}$ et l'envoie dans $Q \dot{\wedge} P = \{\langle n, m\rangle : n \in Q, m \in P\}$ (sans connaître P, Q).

Il suffit de permuter les coordonnées du couple !

Mais c'est ce que faisait le programme $\lambda z.\langle \pi_2 z, \pi_1 z\rangle$ associé à la preuve via Curry-Howard (cf. transp. 52) !

- Ceci marche en général : pour réaliser φ où φ est prouvable, on prend le λ-terme de preuve, on oublie les types (tout est entier !), on interprète l'application d'une fonction f sur n comme $\Phi_f(n)$, les couples et les sommes comme dans $\dot{\wedge}$ et $\dot{\vee}$, et le programme en question réalise φ quels que soient $P_1, \ldots, P_r$ puisqu'ils correspondent à des types dans le terme de preuve.

La réalisabilité propositionnelle : contre-exemples

- La formule $A \vee \neg A$ n'est pas réalisable : il s'agirait de trouver un même entier qui quel que soit P soit de la forme $\langle 0, n\rangle$ avec $n \in P$ ou bien de la forme $\langle 1, n\rangle$ si $P = \varnothing$. Visiblement c'est impossible (sans information sur P) !

Remarque : en fait, si $\varphi \vee \psi$ est réalisable, l'une de φ ou ψ l'est (selon que l'entier réalisant $\varphi \vee \psi$ est de la forme $\langle 0, m\rangle$ ou $\langle 1, m\rangle$).

- La formule $\neg\neg A \Rightarrow A$ n'est pas réalisable : il s'agirait de trouver un même programme qui, si P est non-vide (si bien que $\dot{\neg}\dot{\neg} P = \mathbb{N}$), termine sur n'importe quel entier et renvoie un élément de P. Visiblement c'est impossible (sans information sur P) !

- Plus subtilement, $(\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)$ n'est pas réalisable : il s'agirait de trouver un programme qui transforme une solution du 2e problème en une solution du 1er.

De nouveau, ceci montre que ces formules ne sont pas prouvables.

En revanche, $(A \vee \neg A) \Rightarrow (\neg\neg A \Rightarrow A)$ est réalisable car démontrable : le λ-terme $\lambda(z : A \vee \neg A).\lambda(u : \neg\neg A).(\texttt{match}\ z\ \texttt{with}\ \iota_1 v_1 \mapsto v_1,\ \iota_2 v_2 \mapsto \texttt{exfalso}^{(A)}(uv_2))$ le prouve et explique comment le réaliser.

La réalisabilité propositionnelle : incomplétude

La réalisabilité suit tellement près les idées de Curry-Howard qu'on pourrait naturellement croire que la réciproque est vraie, que toute proposition réalisable donnera un λ-terme de preuve, i.e., que la sémantique est complète.

Surprise : non !

On connaît des formules propositionnelles, p.ex. (transp. suivant) :

$$\Big(\neg(A_1 \wedge A_2) \wedge (\neg A_1 \Rightarrow (B_1 \vee B_2)) \wedge (\neg A_2 \Rightarrow (B_1 \vee B_2))\Big)$$
$$\Rightarrow \Big((\neg A_1 \Rightarrow B_1) \vee (\neg A_2 \Rightarrow B_1) \vee (\neg A_1 \Rightarrow B_2) \vee (\neg A_2 \Rightarrow B_2)\Big)$$

qui sont réalisables mais non prouvables en logique intuitionniste.

C'est-à-dire que bien qu'on ait un algorithme (transp. suivant) qui réalise cette formule pour toutes parties A_1, A_2, B_1, B_2 de $\mathbb{N}$ (et moralement : toutes données), on ne peut pas typer un tel algorithme, lequel dépend de la possibilité de faire temporairement des « fausses promesses » pour donner finalement un résultat juste.

Problèmes ouverts : l'ensemble des formules réalisables est-il décidable ? Semi-décidable ?

Réalisabilité de la formule de Tseitin

Ceci est une digression mais je pense que c'est très instructif pour la calculabilité de comprendre la différence entre typage et réalisabilité.

La formule suivant, bien que non démontrable, est réalisable :

$$(\neg(A_1 \wedge A_2) \wedge (\neg A_1 \Rightarrow (B_1 \vee B_2)) \wedge (\neg A_2 \Rightarrow (B_1 \vee B_2)))$$
$$\Rightarrow ((\neg A_1 \Rightarrow B_1) \vee (\neg A_2 \Rightarrow B_1) \vee (\neg A_1 \Rightarrow B_2) \vee (\neg A_2 \Rightarrow B_2))$$

Algorithme (appelons A_1, A_2, B_1, B_2 les parties concernées) :

- On reçoit en entrée une promesse que l'un de A_1 et A_2 est vide, ainsi que deux algorithmes, l'un $e_1 \in (\dot{\neg} A_1 \dot{\Rightarrow} (B_1 \dot{\vee} B_2))$ prenant en entrée une promesse que $A_1 = \varnothing$ et renvoyant un élément de B_1 ou B_2, et l'autre e_2 prennant promesse $A_2 = \varnothing$ et renvoyant un él$^{\text{t}}$ de B_1 ou B_2.
- On lance en parallèle e_1 resp. e_2 sur un entier qcque promettant (peut-être faussement !) $A_1 = \varnothing$ resp. $A_2 = \varnothing$. Au moins l'une de ces promesses est vraie (par la promesse en entrée) donc l'un de e_1 ou e_2 va terminer, et renvoyer soit un élément annoncé de B_1 soit de B_2.

Disons sans perte de généralité que e_1 renvoie un élément n annoncé de B_1.

- On renvoie alors comme élément de $\dot{\neg} A_1 \dot{\Rightarrow} B_1$ un programme renvoyant constamment n. Il est bien dans $\dot{\neg} A_1 \dot{\Rightarrow} B_1$ car s'il reçoit une promesse que $A_1 = \varnothing$, à l'étape précédente le programme a tourné sur une vraie promesse et donc a vraiment renvoyé un élément de B_1.

Sémantique 4 : les problèmes finis

- Un **problème fini** est un couple (X, S) où X est un ensemble fini non vide appelé les **candidats** du problème et $S \subseteq X$ est un sous-ensemble appelé les **solutions** du problème.

On définit les opérations suivantes sur les problèmes finis :

- $(X, S) \dot{\wedge} (Y, T) = (X \times Y, S \times T)$ (produit cartésien)
- $(X, S) \dot{\vee} (Y, T) = (X \uplus Y, S \uplus T)$ (réunion disjointe)
- $(X, S) \dot{\Rightarrow} (Y, T) = (Y^X, U)$ où $U := \{f : X \to Y : f(S) \subseteq T\}$ (fonctions envoyant une solution de (X, S) en une solution de (Y, T))
- $\dot{\top} = (\{\bullet\}, \{\bullet\})$
- $\dot{\bot} = (\{\bullet\}, \varnothing)$

- Si $\varphi(A_1, \ldots, A_r)$ est une formule propositionnelle et $(X_1, S_1), \ldots, (X_r, S_r)$ des problèmes finis, on définit le problème $\dot{\varphi}((X_1, S_1), \ldots, (X_r, S_r))$ de façon évidente (par induction).

Sémantique des problèmes finis

- Si $\varphi(A_1, \ldots, A_r)$ est une formule propositionnelle et $(X_1, S_1), \ldots, (X_r, S_r)$ des problèmes finis, noter que l'ensemble Z de candidats de du problème $\dot{\varphi}((X_1, S_1), \ldots, (X_r, S_r))$ ne dépend que des ensembles $X_1, \ldots, X_r$ de candidats donnés.

- On dit que φ est Medvedev-valide si pour tous $X_1, \ldots, X_r$ il existe un même $z \in Z$ tel que $z \in \dot{\varphi}((X_1, S_1), \ldots, (X_r, S_r))$ quels que soient $S_1, \ldots, S_r$ (sous-ensembles respectifs de $X_1, \ldots, X_r$).

- Il s'agit d'une autre façon de donner un sens précis à l'interprétation de Brouwer-Heyting-Kolmogorov, cette fois sur des ensembles finis (donc pas de question de calculabilité).

- **Théorème** (Medvedev) : cette sémantique est correcte pour le calcul propositionnel intuitionniste.

Elle n'est pas complète : p.ex., $(\neg A \Rightarrow (B_1 \vee B_2)) \Rightarrow (\neg A \Rightarrow B_1) \vee (\neg A \Rightarrow B_2)$ (Kreisel-Putnam, déjà évoqué) ou bien $((\neg\neg A \Rightarrow A) \Rightarrow (A \vee \neg A)) \Rightarrow (\neg A \vee \neg\neg A)$ (Scott) sont Medvedev-valides mais non démontrables.

L'algorithme de Hindley-Milner : description sommaire

Rappel : le λ-calcul simplement typé (λCST), pour nous, porte *toutes* les annotations de type.

- Donné un type du λCST, on appelle **désannotation** de celui-ci le type du λ-calcul non typé obtenu en retirant toutes les annotations de type : p.ex.

$$\lambda(x:\alpha).\,\lambda(f:((\alpha\to\beta)\to\beta)\to\gamma).\,f(\lambda(h:\alpha\to\beta).hx)$$

de type $\alpha\to((\alpha\to\beta)\to\beta)\to\gamma\to\gamma$ se désannote en $\lambda xf.\,f(\lambda h.hx)$.

- L'**inférence de type** pour le λCST est le problème, donné un terme du λ-calcul non typé, de :
  - décider s'il est typable, i.e., est la désannotation d'un terme du λCST,
  - si oui, retrouver son type et toutes les annotations, et même
  - trouver la solution la « plus générale » possible (p.ex., $\lambda x.x$ doit retrouver $\lambda(x:\alpha).x$ et pas $\lambda(x:\alpha\to\beta).x$).

- L'**algorithme de Hindley-Milner** (ou Damas-Milner, car Damas a prouvé sa correction) ou **algorithme W** (ou **J**, c'est quasi pareil) accomplit ces buts.

- On fait ainsi des « types opaques » du λCST un vrai polymorphisme.

L'algorithme de Hindley-Milner : esquisse

L'algorithme de Hindley-Milner procède en deux phases :

- **Première phase : collection des contraintes** : donnée un terme à typer, on alloue une variable de type fraîches à chaque nouvelle variable introduite ou application et on enregistre une contrainte (équation) à chaque application. Cette phase n'échoue pas.

  P.ex.: sur $\lambda x.\lambda y.xy$, on va enregistrer successivement les types $x : \eta_1$, $y : \eta_2$, $xy : \eta_3$ et la contrainte $\eta_1 = (\eta_2 \to \eta_3)$, et renvoyer $\eta_1 \to \eta_2 \to \eta_3$.

- **Deuxième phase : résolution des contraintes par unification** : on cherche une « substitution » des variables de type qui résout toutes les contraintes.

  Pour ça, on prend de façon répétée une équation $\zeta_1 = \zeta_2$ du jeu de contraintes et on cherche à la satisfaire : si ζ_1 ou ζ_2 est une variable, c'est facile, sinon, on décompose chacune ($\zeta_i = (\xi_i \to \chi_i)$) en produisant de nouvelles contraintes ($\xi_1 = \xi_2$ et $\chi_1 = \chi_2$). Cette phase peut échouer.

- L'algorithme termine, donne une solution s'il en existe une, et même la « **solution principale** » dont toutes les autres se dérivent par substitution.

Première phase : collection des contraintes

La première phase de l'algorithme de H-M fonctionne comme la vérification de type du λCST (transp. 27) mais en introduisant des variables de type « fraîches » au vol et en collectant des contraintes au fur et à mesure.

Soit t le terme à typer : l'algo est écrit ici en style impératif et opère sur un contexte Γ et un jeu de contraintes $\mathscr{C}$, il renvoie un type et modifie $\Gamma, \mathscr{C}$:

- si $t = v$ est une variable, elle doit être dans le contexte : renvoyer son type ;
- si $t = \lambda v.e$ est une abstraction, allouer une nouvelle variable de type η, ajouter $v : \eta$ au contexte, appliquer l'algorithme récursivement à e, qui donne ζ, et renvoyer $\eta \to \zeta$;
- si $t = fx$ est une application, appliquer l'algorithme récursivement à f et x, qui donne ζ et ξ, allouer une nouvelle variable de type η, ajouter l'équation $\zeta = (\xi \to \eta)$ aux contraintes et renvoyer η.

- Ceci s'adapte sans difficulté au cas où on dispose d'annotations de type partielles.

- Le typage final s'obtiendra en appliquant la substitution trouvée dans la deuxième phase.

Collection des contraintes : exemples

- Soit le terme à typer $\lambda xyz.xz(yz)$. On collecte les types et contraintes suivants : $x : \eta_1$, $y : \eta_2$, $z : \eta_3$, $xz : \eta_4$ avec $\eta_1 = (\eta_3 \to \eta_4)$, $yz : \eta_5$ avec $\eta_2 = (\eta_3 \to \eta_5)$, $xz(yz) : \eta_6$ avec $\eta_4 = (\eta_5 \to \eta_6)$, et on renvoie finalement le type $\eta_1 \to \eta_2 \to \eta_3 \to \eta_6$.
- Soit le terme à typer $\lambda fx.f(\lambda g.gx)$. On collecte les types et contraintes suivants : $f : \eta_1$, $x : \eta_2$, $g : \eta_3$, $gx : \eta_4$ avec $\eta_3 = (\eta_2 \to \eta_4)$, $f(\lambda g.gx) : \eta_5$ avec $\eta_1 = ((\eta_3 \to \eta_4) \to \eta_5)$, et on renvoie finalement le type $\eta_1 \to \eta_2 \to \eta_5$.
- Soit le terme à typer $\lambda x.xx$. On collecte les types et contraintes suivants : $x : \eta_1$, $xx : \eta_2$ avec $\eta_1 = (\eta_1 \to \eta_2)$, et on renvoie finalement le type $\eta_1 \to \eta_2$.
- Soit le terme à typer $t := (\lambda xy.x)(\lambda x'y'.x')$. On collecte les types et contraintes suivants : $x : \eta_1$, $y : \eta_2$, $x' : \eta_3$, $y' : \eta_4$ enfin $t : \eta_5$ avec $(\eta_1 \to \eta_2 \to \eta_1) = ((\eta_3 \to \eta_4 \to \eta_3) \to \eta_5)$.

Attention : ce terme aurait pu être écrit $(\lambda xy.x)(\lambda xy.x)$: les deux x ne sont pas le même !

Seconde phase : résolution des contraintes

L'algorithme d'**unification** prend en entrée un ensemble $\mathscr{C}$ de contraintes (équations $\zeta_1 = \zeta_2$ avec ζ_1, ζ_2 deux types) et renvoie une **substitution** des variables de type ($\{\eta \mapsto \tau\}$ avec η des variables de type et τ des types ne faisant pas intervenir les variables substituées) qui vérifie les contraintes, ou « échec » :

- Si $\mathscr{C}$ est vide, renvoyer la substitution vide.
- Sinon, soit $\zeta_1 = \zeta_2$ une contrainte, et $\mathscr{C}' := \mathscr{C} \setminus \{(\zeta_1 = \zeta_2)\}$ le reste des contraintes :
  - si $\zeta_1 = \zeta_2$ déjà, unifier $\mathscr{C}'$,
  - si ζ_1 est une variable de type η : si ζ_2 ne fait pas intervenir η, ajouter (et appliquer) $\eta \mapsto \zeta_2$ à la substitution, et unifier $\mathscr{C}'$ où η est remplacé par ζ_2 ; sinon, échouer (« type récursif ») ;
  - si ζ_2 est une variable de type : symétriquement ;
  - sinon, $\zeta_1 = (\xi_1 \to \chi_1)$ et $\zeta_2 = (\xi_2 \to \chi_2)$: unifier $\mathscr{C}'' := \mathscr{C}' \cup \{(\xi_1 = \xi_2), (\chi_1 = \chi_2)\}$.

Résolution des contraintes : exemple

Reprenons l'exemple $t := (\lambda xy.x)(\lambda x'y'.x')$. La première phase a donné : $x : \eta_1$, $y : \eta_2$, $x' : \eta_3$, $y' : \eta_4$ et le type final η_5 avec la constrainte $(\eta_1 \to \eta_2 \to \eta_1) = ((\eta_3 \to \eta_4 \to \eta_3) \to \eta_5)$.

- On examine la seule contrainte $(\eta_1 \to \eta_2 \to \eta_1) = ((\eta_3 \to \eta_4 \to \eta_3) \to \eta_5)$: ceci retire cette contrainte et on rajoute $\eta_1 = (\eta_3 \to \eta_4 \to \eta_3)$ et $(\eta_2 \to \eta_1) = \eta_5$.
- On examine $\eta_1 = (\eta_3 \to \eta_4 \to \eta_3)$: comme η_1 est une variable, et n'apparaît pas à droite, on enregistre la substitution $\eta_1 \mapsto (\eta_3 \to \eta_4 \to \eta_3)$ et on l'applique à la contrainte restante qui devient $(\eta_2 \to \eta_3 \to \eta_4 \to \eta_3) = \eta_5$.
- Comme η_5 est une variable et n'apparaît pas à gauche, on enregistre la substitution $\eta_5 \mapsto (\eta_2 \to \eta_3 \to \eta_4 \to \eta_3)$ et il ne reste plus de contrainte.

Finalement, on a typé :
$\vdash (\lambda(x : \eta_3 \to \eta_4 \to \eta_3).\, \lambda(y : \eta_2).\, x)\ (\lambda(x' : \eta_3).\, \lambda(y' : \eta_4).\, x')\ :$
$\eta_2 \to \eta_3 \to \eta_4 \to \eta_3$.

Propriétés de l'algorithme de Hindley-Milner

La difficulté concerne essentiellement la seconde phase (résolution des contraintes par unification). On peut prouver :

- **L'algorithme termine** (il est même p.r. et mieux).

Idée de la preuve : double récurrence, sur le nombre de variables de type (récurrence principale) et sur le degré total des formules dans les contraintes.

- **L'algorithme est correct** : s'il retourne une substitution, celle-ci résout les contraintes. (C'est à peu près évident.)

- (Damas) **L'algorithme est « complet » :** toute autre solution des contraintes $\mathscr{C}$ s'obtient en effectuant une substitution sur celle retournée par l'algorithme : on dit qu'il trouve la **solution principale** (notamment, celle-ci existe !).

Idée de la preuve : récurrence sur le nombre d'appels récursifs à la fonction d'unification.

*

- L'algorithme se généralise sans problème à l'ajout de types produits, sommes, 1 et 0 ($\rightarrow$ nvx cas d'échecs d'unification, p.ex., si $\zeta_1 = (? \rightarrow ?)$ et $\zeta_2 = (? \times ?)$).

Types récursifs ?

- On a imposé lors de la résolution de la contrainte $\zeta_1 = \zeta_2$, lorsque ζ_1 est une variable η, que ζ_2 ne fasse pas intervenir η.
- Ceci sert à empêcher d'inférer un type p.ex. à $\lambda x.xx$ (il fallait résoudre $\eta_1 = (\eta_1 \to \eta_2)$). De fait, ce terme n'est pas typable dans le λCST, sinon $(\lambda x.xx)(\lambda x.xx)$ le serait, contredisant la normalisation forte du λCST.
- On peut néanmoins étendre l'algorithme de H-M à de tels types récursifs sans constructeur, c'est ce que fait `ocaml -rectypes` :

```
$ ocaml -rectypes
fun x -> x x ;;
- : ('a -> 'b as 'a) -> 'b = <fun>
```

(Ceci doit faire de OCaml un interpréteur du λ-calcul non typé, au moins pour une certaine notion d'évaluation.)

Mais c'est une mauvaise idée de s'en servir : mieux vaut définir un type récursif explicite.

Le problème du polymorphisme du « let »

- Dans les langages fonctionnels, « `let` v=x `in` t » peut être vu comme un sucre syntaxique pour « `(fun` $v \mapsto t)x$ » (i.e., $(\lambda v.t)x$, cf. transp. 31).

- Ceci pose un problème au typage à la H-M : mettons qu'on veuille utiliser l'entier de Church $\overline{2} := \lambda f x.f(fx)$, typé par H-M comme $(\alpha \to \alpha) \to (\alpha \to \alpha)$, avec deux types α différents, par exemple en OCaml :

```
let twice = fun f -> fun x -> f(f x) in (twice ((+)1) 42, twice not true) ;;
- : int * bool = (44, true)
(fun twice -> (twice ((+)1) 42, twice not true))(fun f -> fun x -> f(f x)) ;;
Error: This expression has type bool -> bool
but an expression was expected of type int -> int
Type bool is not compatible with type int
```

Dans l'expression `fun twice -> ...`, le type de `twice` est fixé et ne peut pas être polymorphe.

- Pour réparer ce problème, très grossièrement : (a) on type d'abord x, puis (b) on « généralise » chaque variable de type (non présente dans le contexte d'ensemble) pour rendre x polymorphe (cf. 62), c'est-à-dire que (c) chaque apparition de v dans t reçoit des variables de type fraîches à ces places.

La « restriction de valeur »

- La solution trouvée pour rendre `let` polymorphe (transp. précédent) apporte ses propres difficultés quand le langage permet les effets de bord (variables mutables) : considérons l'expression OCaml :

```
let r = ref (fun x -> x) in r := (fun x -> x+1) ; (!r) true ;;
```

Ici, on ne veut pas généraliser `r` à pouvoir servir à la fois comme `(int -> int) ref` et comme `(bool -> bool) ref` car le code précédent causerait l'appel d'une fonction `int -> int` sur une valeur de type `bool` (crash potentiel !).

- La solution dans un langage comme OCaml est la « restriction de valeur » : le v dans « `let` v=x `in` t » n'est rendu polymorphe (i.e., n'est « généralisé ») que lorsque x est une « valeur statique » déterminable à la compilation.

- Dans un langage comme Haskell, le problème ne se pose pas, car toutes les fonctions et valeurs sont pures.

Partie III: Introduction aux quantificateurs

INF110 (Logique et Fondements de l'Informatique)

David A. Madore
Télécom Paris
david.madore@enst.fr

2023–2025

http://perso.enst.fr/madore/inf110/transp-inf110.pdf

Git: df53831 Fri Apr 25 14:43:57 2025 +0200

Plan

Limitations du calcul propositionnel

- On a parlé pour l'instant de **calcul propositionnel**, qui ne connaît que les affirmations logiques et les connecteurs propositionnels $\Rightarrow, \wedge, \vee, \top, \bot$.
- Mais il y a deux notations logiques essentielles en mathématiques au-delà de ces connecteurs : les **quantificateurs** $\forall, \exists$, qui :
  - prennent une formule $P(v)$ dépendant d'une variable v libre (de type I),
  - lient cette variable, pour former une nouvelle formule : $\forall(v : I).P(v)$ ou $\exists(v : I).P(v)$ (parfois juste $\forall v.P(v)$ et $\exists v.P(v)$).
- Intuitivement, il faut penser à $\forall$ et $\exists$ comme des « $\wedge$ et $\vee$ en famille », c'est-à-dire que :
  - $\forall v.P(v)$, parfois noté $\bigwedge_v P(v)$ est à $P \wedge Q$ ce que $\prod_i p_i$ est à $p \times q$,
  - $\exists v.P(v)$, parfois noté $\bigvee_v P(v)$ est à $P \vee Q$ ce que $\sum_i p_i$ est à $p + q$.
- Il existe de nombreux systèmes logiques différant notamment en ce qu'on a le droit de quantifier (qui sont les v ici ? quel est leur domaine I ?).

Commençons par une discussion informelle de $\forall$ et $\exists$.

L'interprétation BHK des quantificateurs

On a déjà vu l'interprétation informelle des connecteurs, on introduit maintenant les quantificateurs :

- un témoignage de $P \wedge Q$, est un témoignage de P et un de Q,
- un témoignage de $P \vee Q$, est un témoignage de P ou un de Q, et la donnée duquel des deux on a choisi,
- un témoignage de $P \Rightarrow Q$ est un moyen de transformer un témoignage de P en un témoignage de Q,
- un témoignage de $\top$ est trivial,
- un témoignage de $\bot$ n'existe pas,
- un témoignage de $\forall v.P(v)$ est un moyen de transformer un x quelconque en un témoignage de $P(x)$,
- un témoignage de $\exists v.P(v)$ est la donnée d'un certain t_0 et d'un témoignage de $P(t_0)$.

Curry-Howard pour le $\forall$

- On a vu que Curry-Howard fait correspondre conjonction logique $P \wedge Q$ (« un témoignage de P et un de Q ») avec type produit $\sigma \times \tau$ (« une valeur de σ et une de τ »).
- De façon analogue, la quantification universelle $\forall v.P(v)$ (« une façon de transformer v en un témoignage de $P(v)$ »), qui est une sorte de *conjonction en famille* $\bigwedge_v P(v)$, correspondra au type produit en famille $\prod_v \sigma(v)$ (« fonction renvoyant pour chaque v une valeur de $\sigma(v)$ »).
- Ceci présuppose l'existence de familles de types $v \mapsto \sigma(v)$ (= types dépendant de quelque chose) dont on puisse prendre le produit.
- Une preuve de $\forall(v : I).P(v)$ correspondra à un terme de forme $\lambda(v : I).(\cdots)$, où le type de $(\cdots)$ correspond à $P(v)$.
- Remarquer que $\forall(v : I).P$, si P ne dépend pas de v, « ressemble » à $I \Rightarrow P$ de la même manière que $\prod_{i \in I} S = S^I$ (ensemblistement ou numériquement).

(Les détails dépendent de la nature de la quantification.)

Curry-Howard pour le $\exists$

- On a vu que Curry-Howard fait correspondre disjonction logique $P \vee Q$ (« un témoignage de P ou un de Q, avec la donnée duquel on a choisi ») avec type somme $\sigma + \tau$ (« une valeur de σ ou une de τ, avec un sélecteur entre les deux »).

- De façon analogue, la quantification existentielle $\exists v.P(v)$ (« la donnée d'un t_0 et d'un témoignage de $P(t_0)$ »), qui est une sorte de *disjonction en famille* $\bigvee_v P(v)$, correspondra au type somme en famille $\sum_v \sigma(v)$ (« donnée d'un t_0 et d'une valeur de type $\sigma(t_0)$ »).

- Une preuve de $\exists(v : I).P(v)$ correspondra à un terme de forme $\langle t_0, \cdots \rangle$, où le type de $(\cdots)$ correspond à $P(t_0)$. (De nouveau, il faut des « familles de types ».)

- Remarquer que $\exists(v : I).P$, si P ne dépend pas de v, « ressemble » à $I \times P$ de la même manière que $\sum_{i \in I} S = I \times S$. (Les détails dépendent de la nature de la quantification.)

- Mais Curry-Howard atteint ses limites : il n'est pas dit que d'une preuve de $\exists v.P(v)$ on puisse extraire le t_0 correspondant dans autre chose qu'une preuve. (Les détails dépendent du système logique précis considéré et si Martin-Löf est dans la salle.)

Règles d'introduction et d'élimination de $\forall$

Les règles ci-dessous (et transp. suivants) sont incomplètes : il manque des explications sur le type I sur lequel on quantifie et comment on peut en former des « termes d'individus ».

- Introduction du $\forall$: pour montrer $\forall(v : I).\, Q$, on s'arrange (quitte à renommer la variable liée) pour que $v : I$ soit « frais », c'est-à-dire qu'il n'apparaisse (libre) dans aucune hypothèse en cours : si on montre Q sur ce v « arbitraire », on peut conclure $\forall(v : I).\, Q$.
(Rédaction : « soit v arbitraire (...) on a $Q(v)$; donc $\forall(v : I).\, Q(v)$ ».)

Ceci donnera un λ-terme noté $\lambda(v : I).(\cdots)$ comme l'ouverture d'une hypothèse.

- Élimination du $\forall$: pour utiliser $\forall(v : I).Q$, on peut l'appliquer à un t quelconque (un terme de type I).
(Rédaction : « on a $\forall(v : I).Q(v)$ et t de type I ; en particulier, on a $Q(t)$ ».)

Ceci donnera un λ-terme noté ft comme l'application d'une implication.

Règles d'introduction et d'élimination de $\exists$

- Introduction du $\exists$: pour montrer $\exists(v : I).Q$, on peut le montrer sur un terme t quelconque de type I.

(Rédaction : « pour ce t de type I on a $Q(t)$; en particulier, on a $\exists(v : I).Q(v)$ ».)

Ceci donnera un λ-terme noté $\langle t, \cdots \rangle$ comme pour une conjonction.

- Élimination du $\exists$: pour utiliser $\exists(v : I).P(v)$ pour montrer une conclusion Q, on s'arrange (quitte à renommer la variable liée) pour que v soit « frais », c'est-à-dire qu'il n'apparaisse (libre) dans aucune hypothèse en cours ni dans la conclusion Q : si on montre Q à partir de P sur ce v « arbitraire », on peut conclure Q à partir de $\exists v.P$.

(Rédaction : « on a $\exists(v : I).P(v)$: soit v arbitraire tel que $P(v)$: (...) on a Q ; donc Q ».)

Ceci donnera un λ-terme noté $(\texttt{match}\ \cdots\ \texttt{with}\ \langle v, h\rangle \mapsto \cdots)$.

Aperçu d'ensemble des règles de la déduction naturelle

	Intro	Élim
$\Rightarrow$	$\dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$	$\dfrac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}$
$\wedge$	$\dfrac{\Gamma \vdash Q_1 \qquad \Gamma \vdash Q_2}{\Gamma \vdash Q_1 \wedge Q_2}$	$\dfrac{\Gamma \vdash Q_1 \wedge Q_2}{\Gamma \vdash Q_1}$ $\dfrac{\Gamma \vdash Q_1 \wedge Q_2}{\Gamma \vdash Q_2}$
$\vee$	$\dfrac{\Gamma \vdash Q_1}{\Gamma \vdash Q_1 \vee Q_2}$ $\dfrac{\Gamma \vdash Q_2}{\Gamma \vdash Q_1 \vee Q_2}$	$\dfrac{\Gamma \vdash P_1 \vee P_2 \qquad \Gamma, P_1 \vdash Q \qquad \Gamma, P_2 \vdash Q}{\Gamma \vdash Q}$
$\top$	$\dfrac{}{\Gamma \vdash \top}$	(néant)
$\bot$	(néant)	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash Q}$ (ou pour la logique classique : $\dfrac{\Gamma, \neg Q \vdash \bot}{\Gamma \vdash Q}$)
$\forall$	$\dfrac{\Gamma, v : I \vdash Q}{\Gamma \vdash \forall(v : I).\, Q}$ (v frais)	$\dfrac{\Gamma \vdash \forall(v : I).\, Q \qquad \Gamma \vdash t : I}{\Gamma \vdash Q[v \backslash t]}$
$\exists$	$\dfrac{\Gamma \vdash t : I \qquad \Gamma \vdash Q[v \backslash t]}{\Gamma \vdash \exists(v : I).\, Q}$	$\dfrac{\Gamma \vdash \exists(v : I).\, P \qquad \Gamma, v : I, P \vdash Q}{\Gamma \vdash Q}$ (v frais)

- « v frais » = « v n'apparaît nulle part ailleurs » (cf. transp. précédents).
- Le contexte Γ peut contenir des formules (hypothèses) et des variables d'« individus » (avec leur type, p.ex. $v : I$).

Notations des λ-termes pour $\forall$

- On a vu en calcul propositionnel qu'on peut noter les démonstrations par des « λ-termes » qui peuvent ensuite être réinterprétés comme des programmes (c'est Curry-Howard). Complétons ces notations pour $\forall, \exists$:

- Introduction du $\forall$: si s désigne une preuve de Q faisant intervenir v variable libre de type I, on notera $\lambda(v : I).\, s$ la preuve de $\forall(v : I).\, Q$ obtenue par introduction du $\forall$. (**N.B.** v peut apparaître dans Q mais pas dans Γ.)

$$\frac{\Gamma, v : I \vdash s : Q}{\Gamma \vdash \lambda(v : I).\, s \ : \ \forall(v : I).\, Q}$$

- Élimination du $\forall$: si f désigne une preuve de $\forall(v : I).\, Q$ et t un terme de type I, on notera ft la preuve de $Q[v\backslash t]$ (c'est-à-dire Q avec v remplacé par t) obtenue par élimination du $\forall$ sur ce terme.

(**N.B.** on n'explique pas comment le « terme d'individu » t est formé.)

$$\frac{\Gamma \vdash f \ : \ \forall(v : I).\, Q \qquad \Gamma \vdash t : I}{\Gamma \vdash ft \ : \ Q[v\backslash t]}$$

- Ceci est conforme à l'idée de BHK : une preuve de $\forall(v : I).\, Q(v)$ prend un x de type I et renvoie une preuve de $Q(x)$.

Notations des λ-termes pour $\exists$

- Introduction du $\exists$: si t désigne un terme de type type I et z une preuve de $Q[v\backslash t]$ (pour ce t-là, donc), on notera $\langle t, z\rangle$ la preuve de $\exists(v : I).\, Q$ obtenue par introduction du $\exists$.

$$\frac{\Gamma \vdash t : I \qquad \Gamma \vdash z \ : \ Q[v\backslash t]}{\Gamma \vdash \langle t, z\rangle \ : \ \exists(v : I).\, Q}$$

- Élimination du $\exists$: si z désigne une preuve de $\exists(v : I).\, P$ et s une preuve, faisant intervenir v variable libre de type I, de Q qui ne fait pas intervenir v, et h hypothèse supposant P (pour ce v-là, donc), on notera $(\texttt{match}\ z\ \texttt{with}\ \langle v, h\rangle \mapsto s)$ la preuve de Q obtenue par élimination du $\exists$.
(**N.B.** v peut apparaître dans s mais pas dans Γ ni Q.)

$$\frac{\Gamma \vdash z \ : \ \exists(v : I).\, P \qquad \Gamma,\, v : I,\, h : P \vdash s \ : \ Q}{\Gamma \vdash (\texttt{match}\ z\ \texttt{with}\ \langle v, h\rangle \mapsto s) \ : \ Q}$$

- Ceci est conforme à l'idée de BHK : une preuve de $\exists(v : I).\, P(v)$ est la donnée d'un t de type I d'une preuve de $Q(t)$.

Récapitulatif des notations

(À comparer au transp. 9.)

	Intro	Élim
$\Rightarrow$	$\dfrac{\Gamma, v : P \vdash s : Q}{\Gamma \vdash \lambda(v : P).\, s : P \Rightarrow Q}$	$\dfrac{\Gamma \vdash f : P \Rightarrow Q \qquad \Gamma \vdash z : P}{\Gamma \vdash fz : Q}$
$\wedge$	$\dfrac{\Gamma \vdash z_1 : Q_1 \qquad \Gamma \vdash z_2 : Q_2}{\Gamma \vdash \langle z_1, z_2 \rangle : Q_1 \wedge Q_2}$	$\dfrac{\Gamma \vdash z : Q_1 \wedge Q_2}{\Gamma \vdash \pi_1 z : Q_1} \qquad \dfrac{\Gamma \vdash z : Q_1 \wedge Q_2}{\Gamma \vdash \pi_2 z : Q_2}$
$\vee$	$\dfrac{\Gamma \vdash z : Q_i}{\Gamma \vdash \iota_i^{(Q_1,Q_2)} z : Q_1 \vee Q_2}\ (i \in \{1,2\})$	$\dfrac{\Gamma \vdash r : P_1 \vee P_2 \qquad \Gamma, h_1 : P_1 \vdash s_1 : Q \qquad \Gamma, h_2 : P_2 \vdash s_2 : Q}{\Gamma \vdash (\texttt{match}\ r\ \texttt{with}\ \iota_1 h_1 \mapsto s_1,\ \iota_2 h_2 \mapsto s_2) : Q}$
$\top$	$\dfrac{}{\Gamma \vdash \bullet : \top}$	(néant)
$\bot$	(néant)	$\dfrac{\Gamma \vdash r : \bot}{\Gamma \vdash \texttt{exfalso}^{(Q)} r : Q}$
$\forall$	$\dfrac{\Gamma, v : I \vdash s : Q}{\Gamma \vdash \lambda(v : I).\, s \ :\ \forall(v : I).\, Q}$	$\dfrac{\Gamma \vdash f \ :\ \forall(v : I).\, Q \qquad \Gamma \vdash t : I}{\Gamma \vdash ft \ :\ Q[v \backslash t]}$
$\exists$	$\dfrac{\Gamma \vdash t : I \qquad \Gamma \vdash z \ :\ Q[v \backslash t]}{\Gamma \vdash \langle t, z \rangle \ :\ \exists(v : I).\, Q}$	$\dfrac{\Gamma \vdash z \ :\ \exists(v : I).\, P \qquad \Gamma, v : I, h : P \vdash s \ :\ Q}{\Gamma \vdash (\texttt{match}\ z\ \texttt{with}\ \langle v, h \rangle \mapsto s) \ :\ Q}$

- Les séquents en gris sont des formations de termes d'individus (cf. transp. suivant).

Monde des termes et monde logique

- Les règles pour les démonstrations écrites notamment transp. 9 à 12 ne sont pas complètes, il manque les explications sur les séquents en gris (formation des termes).
- Selon le système logique, on peut distinguer deux « mondes » plus ou moins séparés ou confondus :
  - le monde des termes (individus) et types (d'individus),
  - le monde logique, avec preuves et propositions.
- Les règles données aux transp. précédents sont les règles de construction des démonstrations ($\rightarrow$ monde logique), où se placent tous les séquents sauf ceux marqués en gris.
- Les règles du monde des termes peuvent être calquées sur le monde des démonstrations, plus simples, ou différentes.
- En Coq, les deux mondes sont séparés mais parallèles : $Prop$ pour le type des propositions et $Type$ pour le type des types d'individus.

Que peut-on quantifier au juste ?

- Outre la question des termes d'individus et leur séparation du monde logique, il manque les explications sur ce qu'on a le droit de quantifier et d'abstraire ; notamment :
  - peut-on former des propositions comme $\forall(A : *).\,(A \Rightarrow A)$ (où « $*$ » est le type des propositions) avec preuve $\lambda(A : *).\,\lambda(h : A).h$, i.e., quantifier sur les propositions (et abstraire dessus dans les termes) ?
  - peut-on former des objets comme $\lambda(A : *).\,A$ de type $* \to *$, i.e., abstraire sur les propositions ?
  - peut-on former des propositions comme $\forall(x : I).\,A(x)$ où A a pour type $I \to *$, i.e., quantifier et abstraire sur des individus ?
- Différents systèmes logiques diffèrent dans la réponse à ces questions, notamment, les 8 systèmes du « λ-cube » de Barendregt (jusqu'à mélanger complètement preuves et individus).

Le problème du $\exists$ et des types sommes

Doit-on croire à ceci (pour U et V deux types) ?

$$(\forall(x : U).\, \exists(y : V).\, P(x, y)) \Rightarrow (\exists(f : U \Rightarrow V).\, \forall(x : U).\, P(x, f(x)))$$

« preuve(?) » : $\lambda(h : \cdots).\, \langle \lambda(x : U).\, (\texttt{match}\ hx\ \texttt{with}\ \langle v, z\rangle \mapsto v),$
$\lambda(x : U).\, (\texttt{match}\ hx\ \texttt{with}\ \langle v, z\rangle \mapsto z)\rangle$

(Cet énoncé porte le nom d'axiome du choix : c'est un analogue pour la théorie des types de l'axiome du choix (de Zermelo) en théorie des ensembles.)

- Si on voit $\forall$ et $\exists$ comme des types produit et somme en famille respectivement, oui : $\forall(x : U).\, \exists(y : V).\, P(x, y)$ représente une fonction qui prend un x de type U et renvoie un y de type V ainsi qu'un $P(x, y)$ correspondant : on peut collecter tous ces y en une fonction $f : U \Rightarrow V$.
- Si on voit $\exists$ comme un quantificateur logique, alors non : le y renvoyé par $\exists$ ne peut servir qu'à l'intérieur d'une preuve, pas être collecté en une fonction.
- C'est ici la différence principale entre des systèmes comme Coq (où l'énoncé ci-dessus ne sera pas prouvable pour $P : U \times V \to Prop$) et les systèmes à la Martin-Löf comme Agda (où Curry-Howard est suivi « jusqu'au bout » : il n'y a pas de $\exists$ uniquement logique, et cet énoncé est prouvable comme indiqué).

Imprédicativité

- On appelle **imprédicativité** la possibilité de définir une proposition ou un type en quantifiant sur toutes les propositions ou types y compris celui qu'on définit : c'est une forme de circularité.

- P.ex., $\forall(Z : *).\,(Z \Rightarrow A)$ représente le type des fonctions capables de renvoyer un type A à partir d'un type Z quelconque, y compris celui qu'on définit.

Cette imprédicativité est utile pour définir des constructions sur les types.

Exemples (informellement, et en notant « $*$ » le « type des types » imprédicatif) :

- $A \cong \forall(Z : *).\,(Z \Rightarrow A)$: donné une valeur x de type A on peut en fabriquer une de type $Z \Rightarrow A$ comme $\lambda(z : Z).\,x$ pour tout type Z, mais réciproquement, donné une valeur de type $\forall(Z : *).\,(Z \Rightarrow A)$ on peut l'appliquer à $Z = \top$ pour obtenir une valeur de type A.
- $A \cong \forall(Z : *).\,((A \Rightarrow Z) \Rightarrow Z)$: dans un sens on fabrique $\lambda(k : A \Rightarrow Z).\,kx$ comme pour le CPS, dans l'autre sens, appliquer à $Z = A$ et l'identité.
- $\bot \cong \forall(Z : *).\,Z$
- $A \wedge B \cong \forall(Z : *).\,((A \Rightarrow B \Rightarrow Z) \Rightarrow Z)$
- $A \vee B \cong \forall(Z : *).\,((A \Rightarrow Z) \Rightarrow (B \Rightarrow Z) \Rightarrow Z)$

- Cela peut donner des incohérences logiques (paradoxe de Girard).

Logique du premier ordre : principe

- La **logique du premier ordre** ou **calcul des prédicats** (du 1er ordre) est la plus simple qui ajoute les quantificateurs. Les « choses » sur lesquelles on a le droit de quantifier s'appellent des **individus**.

- Côté typage, elle n'est pas très heureuse : les « individus » apparaissent comme un type I unique, *ad hoc*, qu'on ne peut presque pas manipuler (la logique ne permet pas de faire des couples, fonctions, etc., des individus).

- Néanmoins, elle a une grande importance mathématique car le dogme « orthodoxe » est que :

> Les mathématiques se font dans la « théorie des ensembles de Zermelo-Fraenkel en logique du premier ordre » (ZFC).

Le manque d'expressivité de la logique (pas de couples, fonctions, etc.) est compensé par la théorie elle-même (constructions ensemblistes des couples, fonctions, etc.).

- La sémantique (Tarskienne) de la logique du premier ordre a aussi des propriétés agréables (théorème de complétude de Gödel).

Logique du premier ordre : sortes de variables et syntaxe

- En (pure) logique du premier ordre, on a diverses sortes de variables :
  - les **variables d'individus** (x, y, z...) en nombre illimité,
  - les **variables de prédicats** n-aires, ou de **relations** n-aires [entre individus] ($A^{(n)}$, $B^{(n)}$, $C^{(n)}$...), pour chaque entier naturel n.
- L'indication d'arité des variables de prédicats est généralement omise (elle peut se lire sur la formule).
- Une **formule** (logique) est (inductivement) :
  - l'application $A^{(n)}(x_1, \ldots, x_n)$ d'une variable propositionnelle à n variables d'individus,
  - l'application d'un connecteur : $(P \Rightarrow Q)$, $(P \wedge Q)$, $(P \vee Q)$ où P, Q sont deux formules, ou encore $\top$, $\bot$,
  - une quantification : $\forall x.P$ ou $\exists x.P$ (pour $\forall(x : I).P$ ou $\exists(x : I).P$), qui lie la variable d'individu x dans P.
- On ne peut quantifier que sur les individus (« premier ordre »).

Exemples de formules du premier ordre

- Les **formules propositionnelles** sont encore des formules du premier ordre, en interprétant chaque variable propositionnelle comme une variable de prédicat 0-aire (« nullaire ») : $A \wedge B \Rightarrow B \wedge A$ par exemple.

Autres exemples (qui seront par ailleurs tous démontrables) :

- $(\forall x.A(x)) \wedge (\exists x.\top) \Rightarrow (\exists x.A(x))$ (ici, A est un prédicat unaire)
- $(\forall x.\neg A(x)) \Leftrightarrow (\neg\exists x.A(x))$ (idem)
- $(\exists x.\neg A(x)) \Rightarrow (\neg\forall x.A(x))$ (idem)
- $(\exists x.A) \Leftrightarrow (\exists x.\top) \wedge A$ (ici, A est un prédicat nullaire)
- $(\forall x.A) \Leftrightarrow ((\exists x.\top) \Rightarrow A)$ (idem)
- $(\exists x.\forall y.B(x,y)) \Rightarrow (\forall y.\exists x.B(x,y))$ (ici, B est un prédicat binaire)

N.B. On a suivi la convention que $\forall, \exists$ ont une priorité plus faible que les connecteurs $\Rightarrow, \vee, \wedge, \neg$. Tout le monde n'est pas d'accord avec cette convention !

N.B.2 : Il serait peut-être préférable de noter Bxy que $B(x,y)$.

Remarques sur la logique du premier ordre

- Le type I (non écrit) des « individus », le seul sur lequel on peut quantifier, est complètement spécial en logique du premier ordre : on ne l'écrit même pas, on ne peut pas former $I \times I$ ni $I \to I$ ni rien d'autre.
- Ce type I ne se « mélange » pas aux relations $A, B, C, \ldots$: les individus vivent dans un monde hermétiquement séparé des preuves.
- Dans la variante la plus simple, les seuls termes d'individus sont les variables d'individus (i.e., la seule façon d'obtenir $t : I$ est d'avoir $t : I$ dans le contexte !).
- On ne suppose pas I habité, i.e. $\exists x.\top$ n'est pas démontrable (pas plus que $\forall x.A(x) \Rightarrow \exists x.A(x)$). Cf. transp. 26.
- Pour avoir le droit d'écrire $A(x)$, la variable x doit avoir été introduite : la règle d'axiome devrait s'écrire correctement :

$$\frac{}{\Gamma,\ x : I \vdash x : I} \quad \text{et} \quad \frac{\Gamma \vdash x_1 : I \qquad \cdots \qquad \Gamma \vdash x_n : I}{\Gamma,\ A(x_1, \ldots, x_n) \vdash A(x_1, \ldots, x_n)}$$

Logique du premier ordre : reprise des règles logiques

	Intro	Élim
$\Rightarrow$	$\dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$	$\dfrac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}$
$\wedge$	$\dfrac{\Gamma \vdash Q_1 \qquad \Gamma \vdash Q_2}{\Gamma \vdash Q_1 \wedge Q_2}$	$\dfrac{\Gamma \vdash Q_1 \wedge Q_2}{\Gamma \vdash Q_1} \qquad \dfrac{\Gamma \vdash Q_1 \wedge Q_2}{\Gamma \vdash Q_2}$
$\vee$	$\dfrac{\Gamma \vdash Q_1}{\Gamma \vdash Q_1 \vee Q_2} \qquad \dfrac{\Gamma \vdash Q_2}{\Gamma \vdash Q_1 \vee Q_2}$	$\dfrac{\Gamma \vdash P_1 \vee P_2 \qquad \Gamma, P_1 \vdash Q \qquad \Gamma, P_2 \vdash Q}{\Gamma \vdash Q}$
$\top$	$\dfrac{}{\Gamma \vdash \top}$	(néant)
$\bot$	(néant)	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash Q}$ (ou pour la logique classique : $\dfrac{\Gamma, \neg Q \vdash \bot}{\Gamma \vdash Q}$)
$\forall$	$\dfrac{\Gamma, x : I \vdash Q}{\Gamma \vdash \forall x. Q}$ (x frais)	$\dfrac{\Gamma \vdash \forall x. Q \qquad \Gamma \vdash t : I}{\Gamma \vdash Q[x \backslash t]}$
$\exists$	$\dfrac{\Gamma \vdash t : I \qquad \Gamma \vdash Q[x \backslash t]}{\Gamma \vdash \exists x. Q}$	$\dfrac{\Gamma \vdash \exists x. P \qquad \Gamma, x : I, P \vdash Q}{\Gamma \vdash Q}$ (x frais)

- Les hypothèses en gris $\Gamma \vdash t : I$ (formation des termes) ne sont parfois pas écrites dans les arbres de démonstration, mais sont essentielles (cf. transp. 26).

Exemple de preuve en logique du premier ordre

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x.\forall y.B(x,y) \vdash \exists x.\forall y.B(x,y)}\,\text{Ax}}{\exists x.\forall y.B(x,y),\ y' : I \vdash \exists x'.B(x',y')}\,\exists\text{É}\text{LIM} \qquad \dfrac{\dfrac{\dfrac{}{x : I,\ \forall y.B(x,y),\ y' : I \vdash \forall y.B(x,y)}\,\text{Ax}}{x : I,\ \forall y.B(x,y),\ y' : I \vdash B(x,y')}\,\forall\text{É}\text{LIM}}{x : I,\ \forall y.B(x,y),\ y' : I \vdash \exists x'.B(x',y')}\,\exists\text{INT}}{\exists x.\forall y.B(x,y),\ y' : I \vdash \exists x'.B(x',y')}\,\forall\text{INT}}{\exists x.\forall y.B(x,y) \vdash \forall y'.\exists x'.B(x',y')}\,\Rightarrow\text{INT}}{\vdash (\exists x.\forall y.B(x,y)) \Rightarrow (\forall y'.\exists x'.B(x',y'))}$$

Présentation avec les seules conclusions :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x.\forall y.B(x,y)}\,u \qquad \dfrac{\dfrac{\dfrac{}{\forall y.B(x,y)}\,v}{B(x,y')}\,\forall\text{É}\text{LIM}}{\exists x'.B(x',y')}\,\exists\text{INT}}{\exists x'.B(x',y')}\,\exists\text{É}\text{LIM}(x,v)}{\forall y'.\exists x'.B(x',y')}\,\forall\text{INT}(y')}{(\exists x.\forall y.B(x,y)) \Rightarrow (\forall y'.\exists x'.B(x',y'))}\,\Rightarrow\text{INT}(u)$$

Exemple de preuve : présentation drapeau

(1)	$\boxed{\exists x.\forall y.B(x,y)}$	
(2)	$\quad\boxed{y'}$	
(3)	$\quad\quad\boxed{x,\ \forall y.B(x,y)}$	
(4)	$\quad\quad B(x,y')$	$\forall$Élim sur (3) et y'
(5)	$\quad\quad \exists x'.B(x',y')$	$\exists$Int sur x et (4)
(6)	$\quad \exists x'.B(x',y')$	$\exists$Elim sur (1) de (3) dans (5)
(7)	$\forall y'.\exists x'.B(x',y')$	$\forall$Int de (2) dans (6)
(8)	$(\exists x.\forall y.B(x,y)) \Rightarrow (\forall y'.\exists x'.B(x',y'))$	$\Rightarrow$Int de (1) dans (7)

« Supposons $\exists x.\forall y.B(x,y)$. Considérons un y' arbitraire. Considérons un x tel que $\forall y.B(x,y)$. En particulier, on a $B(x,y')$. En particulier, $\exists x'.B(x',y')$. Or on pouvait trouver un tel x car $\exists x.\forall y.B(x,y)$, donc on a bien la conclusion $\exists x'.B(x',y')$. Le choix de y' étant arbitraire, $\forall y'.\exists x'.B(x',y')$. Finalement, on a prouvé $(\exists x.\forall y.B(x,y)) \Rightarrow (\forall y'.\exists x'.B(x',y'))$. »

Notation des preuves par des λ-termes

On reprend la preuve donnée aux transp. précédents :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x.\forall y.B(x,y) \vdash \exists x.\forall y.B(x,y)}\,\text{Ax}}{\exists x.\forall y.B(x,y) \vdash \exists x'.\forall y'.B(x',y')}\,\exists\text{Élim}}{\exists x.\forall y.B(x,y),\ y' : I \vdash \exists x'.B(x',y')}\,\forall\text{Int}}{\exists x.\forall y.B(x,y) \vdash \forall y'.\exists x'.B(x',y')}\,\Rightarrow\text{Int}}{\vdash (\exists x.\forall y.B(x,y)) \Rightarrow (\forall y'.\exists x'.B(x',y'))}$$

$$\dfrac{\dfrac{\dfrac{}{x : I,\ \forall y.B(x,y),\ y' : I \vdash \forall y.B(x,y)}\,\text{Ax}}{x : I,\ \forall y.B(x,y),\ y' : I \vdash B(x,y')}\,\forall\text{Élim}}{x : I,\ \forall y.B(x,y),\ y' : I \vdash \exists x'.B(x',y')}\,\exists\text{Int}$$

Avec les notations du transp. 12 elle se note :

$$\lambda(u : \exists x.\forall y.B(x,y)).\ \lambda(y' : I).\ (\texttt{match}\ u\ \texttt{with}\ \langle x, v\rangle \mapsto \langle x, vy'\rangle)$$

Exemples de preuves écrites comme λ-termes

- $(\forall x.(A(x) \Rightarrow C)) \Rightarrow ((\exists x.A(x)) \Rightarrow C)$

Preuve : $\lambda(f : \forall x.(A(x) \Rightarrow C)).\, \lambda(p : \exists x.A(x)).\, (\texttt{match}\ p\ \texttt{with}\ \langle x, w\rangle \mapsto f x w)$

- $((\exists x.A(x)) \Rightarrow C) \Rightarrow (\forall x.(A(x) \Rightarrow C))$

Preuve : $\lambda(g : (\exists x.A(x)) \Rightarrow C).\, \lambda(x : I).\, \lambda(u : A(x)).\, g\langle x, u\rangle$

- Notamment pour C valant $\bot$ on a prouvé $(\forall x.\neg A(x)) \Leftrightarrow \neg(\exists x.A(x))$ ci-dessus.

- $(\exists x.(A(x) \Rightarrow C)) \Rightarrow ((\forall x.A(x)) \Rightarrow C)$

Preuve : $\lambda(p : \exists x.(A(x) \Rightarrow C)).\, \lambda(f : \forall x.A(x)).\, (\texttt{match}\ p\ \texttt{with}\ \langle x, w\rangle \mapsto w(f x))$

- Notamment pour C valant $\bot$ on a prouvé $(\exists x.\neg A(x)) \Rightarrow \neg(\forall x.A(x))$ ci-dessus.

- $(\forall x.A(x)) \Rightarrow (\exists x.\top) \Rightarrow (\exists x.A(x))$

Preuve : $\lambda(f : \forall x.A(x)).\, \lambda(p : \exists x.\top).\, (\texttt{match}\ p\ \texttt{with}\ \langle x, u\rangle \mapsto \langle x, f x\rangle)$

- $\forall z.\exists x.\top$ (cf. transp. suivant)

Preuve : $\lambda(z : I).\, \langle z, \bullet\rangle$

Pourquoi des variables d'individus avec les hypothèses ?

- L'introduction d'une variable d'individu libre porte en elle l'hypothèse que l'univers des individus est habité ($\exists x.\top$). Ce fait n'est pas prouvable sans cette hypothèse. On a $z : I \vdash \exists x.\top$ (ici z variable qcque) mais on n'a pas $\vdash \exists x.\top$.

- Exiger que de pouvoir former $t : I$ à partir de Γ permet d'écarter la démonstration incorrecte suivante :

$$\exists\textsc{Int}\ \dfrac{\top\textsc{Int}\ \dfrac{}{\vdash \top}}{\vdash \exists x.\top}$$

- En revanche, celle-ci est correcte (en utilisant le terme z pour t dans $\exists$Int) :

$$\forall\textsc{Int}\ \dfrac{\exists\textsc{Int}\ \dfrac{\textsc{Ax}\ \dfrac{}{z : I \vdash z : I} \qquad \top\textsc{Int}\ \dfrac{}{z : I \vdash \top}}{z : I \vdash \exists x.\top}}{\vdash \forall z.\exists x.\top}$$

N.B. Ces problèmes n'ont rien à voir avec la logique intuitionniste, ils sont identiques en logique classique. On n'a pas non plus $\forall x.A(x) \Rightarrow \exists x.A(x)$.

Monde des individus et monde logique

- En logique du premier ordre, on a deux « mondes » complètement séparés :
  - le monde des individus, avec un seul type (I) et des variables sur lesquelles on peut quantifier,
  - le monde logique, avec propositions et preuves.
- Les propositions ont « moralement » un type (qu'on pourrait appeler « $*$ » ou « $Prop$ »), mais on ne l'écrit pas. Les relations n-aires ont « moralement » le type $I^n \to Prop$, pas non plus écrit.
- Dans le transparent 21, tous les séquents concernent le monde logique (= construction des démonstrations), sauf ceux marqués en gris.
- Dans la version la plus simple de la logique du premier ordre (pas de *fonctions* d'individus, seulement des *relations*), la seule règle du monde des individus est :

$$\frac{}{\Gamma,\ x : I \vdash x : I}$$

(on ne peut former un terme d'individu qu'en invoquant une variable du contexte).

Curry-Howard pour la logique du premier ordre

- Il faut penser Curry-Howard dans le sens preuve $\mapsto$ programme.

(Faute de description précise de règles de typage on ne peut pas espérer mieux ici.)

- Curry-Howard va mélanger le monde logique avec le monde des individus.
- On convertit les propositions en types :
  - $\Rightarrow, \wedge, \vee, \top, \bot$ deviennent $\rightarrow, \times, +, 1, 0$ comme en calcul propositionnel,
  - $\forall, \exists$ deviennent produits et sommes $\prod, \sum$ paramétrés par $v : I$.
- On convertit preuves en programmes selon l'interprétation fonctionnelle des notations données au transp. 12.
- P.ex., la preuve de $(\exists x.\forall y.B(x,y)) \Rightarrow (\forall y'.\exists x'.B(x',y'))$ donnée transp. 24 :

$$\lambda(u : \exists x.\forall y.B(x,y)).\ \lambda(y' : I).\ (\texttt{match}\ u\ \texttt{with}\ \langle x, v\rangle \mapsto \langle x, vy'\rangle)$$

devient un programme de type

$$(\sum_{x:I}\prod_{y:I} B(x,y)) \rightarrow (\prod_{y':I}\sum_{x':I} B(x',y'))$$

L'égalité au premier ordre

- En général on veut travailler en logique du premier ordre avec égalité. C'est-à-dire qu'on introduit une relation binaire « $=$ » (notée de façon infixe) sujette aux axiomes suivants :
  - réflexivité : refl : $\forall x.(x = x)$
  - substitution : $\text{subst}^{(\lambda s.P(s))} : \forall x.\forall y.((x = y) \Rightarrow P(x) \Rightarrow P(y))$ pour toute formule $P(s)$ ayant une variable d'individu libre s.

- La logique du premier ordre montre ici ses limites : on n'a pas le droit de quantifier sur $P(s)$ ni même d'introduire $\lambda(s : I).P(s)$. Il faut donc comprendre qu'on a un « schéma d'axiomes » de substitution, dont chaque $\text{subst}^{(\lambda s.P(s))}$ est une instance (et $\lambda s.P(s)$ une notation *ad hoc*).

- Exemple de preuve : la symétrie $\forall x.\forall y.((x = y) \Rightarrow (y = x))$ est prouvée en appliquant la substitution à $P(s)$ valant « $s = x$ », donc par le λ-terme

$$\lambda(x : I).\,\lambda(y : I).\,\lambda(u : (x = y)).\,\text{subst}^{(\lambda s.s=x)}\,x\,y\,u\,(\text{refl}\,x)$$

- Autre exemple : la transitivité $\forall x.\forall y.\forall z.((x = y) \Rightarrow (y = z) \Rightarrow (x = z))$ par

$$\lambda(x : I).\,\lambda(y : I).\,\lambda(z : I).\,\lambda(u : (x = y)).\,\lambda(v : (y = z)).\,\text{subst}^{(\lambda s.x=s)}\,y\,z\,v\,u$$

L'arithmétique de Heyting et de Peano

- L'**arithmétique du premier ordre** est une (tentative d')axiomatisation des entiers naturels en logique du premier ordre. Elle est basée sur les **axiomes de Peano** (transp. suivant).
- On parle d'**arithmétique de Heyting** (HA) en logique intuitionniste, et **de Peano** (PA) en logique classique (mêmes axiomes, seule la logique change).
- Le cadre de base est la logique du premier ordre avec égalité (cf. transp. 29) et avec des opérations de formation de termes d'individus 0 (nullaire), S (unaire) et $+$, $\times$, $\triangle$ (binaires) :
  - 0 est un terme,
  - si m est un terme, (Sm) en est un,
  - si m, n sont deux termes, $(m + n)$, $(m \times n)$, $(m \triangle n)$ en sont.

Ils sont censés représenter le successeur (Sn désigne $n + 1$), la somme, le produit et l'exponentiation. On omet les parenthèses comme d'habitude. On peut abréger $1 = S0$ et $2 = S(S0)$, etc.

Les axiomes de Peano

On garde les axiomes de l'égalité :

- $\forall n.(n = n)$
- $\forall m.\forall n.((m = n) \Rightarrow P(m) \Rightarrow P(n))$ (schéma de substitution)

Les **axiomes de Peano** du premier ordre s'y ajoutent :

- $\forall n.\neg(Sn = 0)$
- $\forall m.\forall n.((Sm = Sn) \Rightarrow (m = n))$
- $P(0) \Rightarrow (\forall n.(P(n) \Rightarrow P(Sn))) \Rightarrow (\forall n.P(n))$ (schéma de récurrence)
- $\forall m.(m + 0 = m)$
- $\forall m.\forall n.(m + (Sn) = S(m + n))$
- $\forall m.(m \times 0 = 0)$
- $\forall m.\forall n.(m \times (Sn) = m \times n + m)$
- $\forall m.(m \triangle 0 = S0)$
- $\forall m.\forall n.(m \triangle (Sn) = m \triangle n \times m)$

Ci-dessus, $P(s)$ désigne une formule ayant une variable d'individu libre s : la substitution de l'égalité et la récurrence sont des schémas d'axiomes (un axiome pour chaque P possible) car on ne peut pas quantifier sur P au premier ordre.

Exemple de preuve en arithmétique de Heyting

Montrons que $\forall n.(n = 0 \vee \neg n = 0)$ (classiquement c'est une évidence logique, mais c'est aussi démontrable intuitionistement) :

- On procède par récurrence. Notons par $P(k)$ la formule $k = 0 \vee \neg k = 0$:
  - $P(0)$ vaut car $0 = 0$ vaut (réflexivité de l'égalité).
  - $P(Sn)$ vaut car $\neg(Sn = 0)$ (premier axiome de Peano). En particulier, $P(n) \Rightarrow P(Sn)$.
  - Donc, par récurrence, $\forall n.P(n)$, ce qu'on voulait prouver.

- Le λ-terme de cette preuve ressemble à quelque chose comme ceci :

$\mathsf{recurr}^{(\lambda k.(k=0 \vee \neg k=0))}\ (\iota_1^{(0=0,\neg 0=0)}(\mathsf{refl}\, 0))\ (\lambda(n : \mathsf{nat}).\, \lambda(h : (n = 0) \vee \neg(n = 0)).\ \iota_2^{(Sn=0,\neg Sn=0)}(\mathsf{succnotz}\, n))$.

Ici, « nat » a été mis pour le type des individus (entiers naturels), « succnotz » pour l'axiome de Peano qui affirme $\forall n.\neg(Sn = 0)$, et « $\mathsf{recurr}^{(\lambda k.P(k))}$ » pour celui qui affirme $P(0) \Rightarrow (\forall n.(P(n) \Rightarrow P(Sn))) \Rightarrow (\forall n.P(n))$.

Quelques théorèmes de l'arithmétique du premier ordre

On peut prouver en arithmétique de Heyting (et notamment, de Peano) que :

- l'addition est commutative, associative, a 0 pour élément neutre...,
- la multiplication est commutative, associative, a $1 = S0$ pour élément neutre...,
- les identités habituelles sur l'addition, la multiplication, l'exponentiation,
- les propriétés basiques de $m \leq n$ défini par $\exists k.(n = m + k)$,
- les propriétés basiques du codage de Gödel $\langle m, n\rangle$ défini par $m + \frac{1}{2}(m + n)(m + n + 1)$,
- les propriétés basiques des suites finies codées par $\langle\!\langle a_0, \ldots, a_{k-1}\rangle\!\rangle := \langle a_0, \langle a_1, \langle \cdots, \langle a_{k-1}, 0\rangle + 1 \cdots\rangle + 1\rangle + 1$,
- l'existence et l'unicité de la division euclidienne,
- des propriétés arithmétique de base : existence d'une infinité de nombres premiers, existence et unicité de la DFP, irrationalité de $\sqrt{2}$ $(\forall p.\forall q.((p \times p = 2 \times q \times q) \Rightarrow q = 0))$, etc.

Curry-Howard pour l'arithmétique de Heyting

- La formule $m = n$ est une relation binaire sur les entiers naturels : elle doit devenir un type (paramétré par m, n, et habité seulement lorsqu'ils sont égaux) sous l'effet de Curry-Howard.
- Il faut y penser comme le type des **témoignages d'égalité** de m et n. En pratique, ce sera un type ayant seul habitant $\{m\}$ lorsque $m = n$ et aucun sinon.
- Chaque axiome de Peano doit devenir un programme (à penser comme l'API d'une bibliothèque « entiers naturels »). Le seul non trivial est le schéma de récurrence $P(0) \Rightarrow (\forall n.(P(n) \Rightarrow P(Sn))) \Rightarrow (\forall n.P(n))$: il faut y penser comme la primitive récursion d'une fonction, qui à $c \in A_0$ et $f(n, —) \colon A_n \to A_{n+1}$ associe la suite $u_n \in \prod_{n\in\mathbb{N}} A_n$ définie par $u_0 = c$ et $u_{n+1} = f(n, u_n)$, ou en OCaml

```
let recurr = fun c -> fun f -> let rec u = fun n -> if n==0 then c else f
(n-1) (u (n-1)) in u ;;
val recurr : 'a -> (int -> 'a -> 'a) -> int -> 'a = <fun>
```

...mais avec un type qui permet à chaque u_n d'être dans un A_n différent : $A_0 \to (\prod_n (A_n \to A_{n+1})) \to (\prod_n A_n)$.

Curry-Howard pour l'arithmétique de Heyting (2)

À quoi ressemble le programme associé à une preuve dans l'arithmétique de Heyting ?

On peut souvent s'en faire une idée d'après son type, p.ex. :

- La commutativité de la multiplication $\forall m.\forall n.(m \times n = n \times m)$ prend m et n et un renvoie un témoignage d'égalité de $m \times n$ et $n \times m$ (c'est-à-dire en fait $m \times n$ calculé de deux manières différentes).
- La preuve de $\forall n.(n = 0 \vee \neg n = 0)$ donnée transp. 32 prend en entrée n et renvoie un type somme avec soit un témoignage d'égalité de n à 0 soit un programme qui donné un tel témoignage renvoie qqch d'impossible. Donc en pratique, ce programme prend n et teste si $n = 0$.
- Une preuve de $\forall m.\exists n.Q(m, n)$ va correspondre à un programme qui prend m et renvoie n ainsi qu'un témoignage de $Q(m, n)$.

Notamment, si $\forall m.\exists n.Q(m, n)$ est prouvable dans l'arithmétique de Heyting, alors on peut en déduire φ_e générale récursive totale telle que $\forall m.Q(m, \varphi_e(m))$ (**extraction** de programme à partir de la preuve).

Propriétés métamathématiques de l'arithmétique de Heyting

- L'arithmétique de Heyting a la **propriété de la disjonction** : si elle prouve $Q_1 \vee Q_2$, alors elle prouve Q_1 ou Q_2.
- Et la **propriété de l'existence** : si elle prouve $\exists n.Q(n)$, alors elle prouve $Q(n)$ pour un n explicite.

Petits caractères : ces faits, comme l'extraction de programme, dépendent d'un résultat de normalisation sur l'arithmétique de Heyting (donc de Consis(HA)).

*

- Soit P^{CPS} la formule obtenue en ajoutant « $\neg\neg$ » devant la formule tout entière, devant la conclusion de chaque $\Rightarrow$, et après chaque $\forall k$. Alors Heyting prouve P^{CPS} ssi Peano prouve P :

$$\mathrm{PA} \vdash P \text{ ssi } \mathrm{HA} \vdash P^{\mathrm{CPS}}$$

Une différence entre Heyting et Peano

- On peut formaliser les fonctions générales récursives (ou machines de Turing) en arithmétique de Heyting. Par exemple, $\varphi_e(i)\downarrow$ signifie $\exists n.T(n,e,i)$ où T est le prédicat (p.r.) de la forme normale de Kleene, « n code un arbre de calcul valable de φ_e sur l'entrée i ».
- Si PA prouve $\forall m.\varphi_e(m)\downarrow$, alors HA le prouve (la réciproque est évidente).
- Si HA prouve $\forall m.\exists n.Q(m,n)$, alors il existe e telle que HA prouve $\forall m.\varphi_e(m)\downarrow$ et $\forall m.Q(m,\varphi_e(m))$.
- La formule

$$\forall e.\forall i.(\varphi_e(i)\downarrow \vee \neg\varphi_e(i)\downarrow)$$

(c'est-à-dire $\forall e.\forall i.((\exists n.T(n,e,i)) \vee \neg(\exists n.T(n,e,i))))$ est évidemment démontrable dans l'arithmétique de Peano (= en logique classique). Elle n'est pas démontrable en arithmétique de Heyting (= en logique intuitioniste), car par Curry-Howard on pourrait extraire de la preuve un algorithme résolvant le problème de l'arrêt.

Petits caractères : ces faits, comme l'extraction de programme, dépendent d'un résultat de normalisation sur l'arithmétique de Heyting (donc de Consis(HA)).

Formalisation de la démontrabilité

Idée-clé : tester si une preuve est valable est algorithmiquement décidable (même primitif récursif).

En revanche, tester si un énoncé est un théorème est seulement semi-décidable (en parcourant toutes les preuves possibles).

Plus précisément :

- On peut construire un codage de Gödel pour les formules arithmétiques et les preuves dans l'arithmétique de Heyting (ou de Peano), et notamment écrire un prédicat primitif récursif

$$\mathsf{Pf}_{\mathsf{HA}}(n,k) \quad \text{resp.} \quad \mathsf{Pf}_{\mathsf{PA}}(n,k)$$

qui signifie « n est le code de Gödel d'une preuve dans l'arithmétique de Heyting (resp. Peano) de la formule codée par k ».

- Notamment, $\exists n.\, \mathsf{Pf}(n,k)$ peut se lire comme « k code un théorème », et l'ensemble de ces k est (au moins) semi-décidable.

Démontrabilité de l'arrêt

Idée-clé : si une machine de Turing s'arrête, on peut démontrer (dans l'arithmétique de Heyting) qu'elle s'arrête en donnant une trace d'exécution pas à pas.

Plus précisément :

- Si $T(n, e, i)$, i.e., si n est un arbre de calcul de φ_e sur l'entrée i, alors on peut de façon algorithmique (même p.r.) tirer de n une preuve de $T(n, e, i)$ dans l'arithmétique de Heyting (i.e., un n' tel que $\mathsf{Pf}_{\mathsf{HA}}(n', k)$ où k est le code de Gödel de l'énoncé $\varphi_e(i)\!\downarrow$, i.e. $\exists n.T(n, e, i)$).

Si une machine de Turing s'arrête, alors le fait qu'elle s'arrête est prouvable (dans l'arithmétique de Heyting, *a fortiori* de Peano).

- On notera aussi que si programme fait une boucle infinie *explicite évidente*, alors le fait qu'il ne termine pas est également prouvable.

Gödel, Rosser et Turing jouent ensemble

(Preuve du théorème de Gödel revue et corrigée par Turing et par Rosser.)

Soit g le programme suivant :

- g cherche en parallèle une preuve (dans l'arithmétique de Peano, disons) de l'énoncé « le programme g termine » (i.e. $\varphi_g(0)\downarrow$) et de l'énoncé « le programme g ne termine pas »,
- c'est-à-dire qu'il énumère les entiers et, pour chacun, teste s'il est le code de Gödel d'une preuve de $\varphi_g(0)\downarrow$ ou de $\neg\varphi_g(0)\downarrow$,
- s'il trouve (en premier) une preuve que g termine, alors il fait une boucle infinie explicite,
- s'il trouve (en premier) une preuve que g ne termine pas, alors il termine immédiatement.

Ce programme g a bien un sens, comme d'habitude, par l'« astuce de Quine » (théorème de récursion de Kleene) + le fait que la vérification des preuves est algorithmique (transp. 38).

Le théorème de Gödel

Admettons provisoirement ce qu'on notera « Consis(PA) » :

l'arithmétique de Peano ne prouve pas $\bot$

- Si le programme g trouve une preuve qu'il ne termine pas, alors il termine. Mais ce point donne une preuve qu'il termine (transp. 39). Donc on a une preuve de $\bot$ dans l'arithmétique de Peano, contredisant le point ci-dessus.
- Si g trouve une preuve qu'il termine, il fait une boucle infinie. Mais ce point donne une preuve que g ne termine pas (boucle infinie explicite). Donc on a une preuve de $\bot$ dans l'arithmétique de Peano, contredisant le point ci-dessus.

Conclusion : g ne trouve ni de preuve qu'il termine ni de preuve qu'il ne termine pas. Donc :

- g ne termine pas,
- ce fait-là n'est pas prouvable dans l'arithmétique de Peano,
- mais on l'a prouvé à l'aide de Consis(PA), donc Consis(PA) lui-même n'est pas prouvable dans Peano (toujours en supposant Consis(PA)).

Cohérence de Peano

- L'énoncé Consis(PA) peut se lire ainsi :

« Le programme g' qui parcourt les entiers et, pour chacun, teste s'il est le code de Gödel d'une preuve de $\bot$ dans l'arithmétique de Peano et dans ce cas termine, ne termine pas. »

Cet énoncé a un sens dans l'arithmétique du premier ordre, mais (on vient de le voir) n'est pas démontrable s'il est vrai.

- L'énoncé Consis(HA) analogue pour l'arithmétique de Heyting est équivalent à Consis(PA) par la traduction CPS (et cette équivalence est prouvable dans HA).

Notamment, Peano ne prouve pas non plus Consis(HA). Par propriété de la disjonction, HA ne prouve même pas Consis(HA) $\vee \neg$Consis(HA).

- En revanche, ZFC (le cadre usuel pour faire des mathématiques) démontre Consis(PA) : « Les axiomes de Peano sont vrais dans $\mathbb{N}$ donc leurs conséquences le sont aussi, et notamment $\bot$ ne peut pas en faire partie. (Et au passage, si PA démontre $\varphi_e(i)\downarrow$, alors $\varphi_e(i)\downarrow$ est vrai.) »

Donc ZFC est strictement plus fort que PA (même pour l'arithmétique).

Le théorème de Gödel généralisé

Pour n'importe quelle sorte de « théorie logique » (classique ou intuitioniste, pas limitée au premier ordre) T telle que :

- les énoncés et démonstrations sont codables par des entiers naturels,
- on peut algorithmiquement tester si un entier naturel code une démonstration valable dans T, et quelle est sa conclusion,
- T permet de formaliser « $\varphi_e(i)\downarrow$ » (calculablement en e et i),
- si $\varphi_e(i)\downarrow$ alors on peut tirer d'une trace d'exécution une preuve de ce fait dans T, et idem pour une boule infinie explicite,

on peut construire le programme g_T qui cherche en parallèle dans T une preuve de que g_T termine ou ne termine pas, et fait une boule infinie explicite dans le premier cas, termine immédiatement dans le second.

- Si T ne prouve pas $\bot$ (hypothèse notée « Consis(T) »), alors g_T ne termine pas, mais T ne peut pas le prouver. Notamment, T ne prouve pas Consis(T) (toujours si Consis(T)). Ceci s'applique notamment à Coq, à ZFC, etc.

L'ensemble des théorèmes est semi-décidable non décidable

Avec T comme au transparent précédent (et sous l'hypothèse Consis(T)), par exemple PA supposons par l'absurde qu'on puisse décider algorithmiquement si une formule P est un théorème de T.

Soit g'' le programme qui :

- teste si $\varphi_{g''}(0)\downarrow = 0$ est un théorème de T (grâce à l'hypothèse effectuée),
- si oui, termine et renvoie 1,
- si non, termine et renvoie 0.

- Par construction, g'' termine forcément et renvoie soit 0 soit 1. Si g'' termine et renvoie 0, alors T le prouve, donc g'' termine et renvoie 1, contradiction ; si g'' termine et renvoie 1, alors T le prouve, donc (par Consis(T)) il ne proue pas que g'' termine et renvoie 0, donc g'' termine et renvoie 0, contradiction.