Typage simple et calcul propositionnel

INF110 (Logique et Fondements de l'Informatique)

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

2023–2024

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

Git: a90f2fd Mon Jan 22 13:08:29 2024 +0100

Plan

Plan [transp. 2]

Table des matières

1 Généralités sur le typage

Qu'est-ce que le typage ? [transp. 3]

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 (→ 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 [transp. 4]

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 [transp. 5]

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 [transp. 6]

▶ **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 [transp. 7]

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` : $(\forall \mathtt{t})$ `List t` $\to$ `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 ! `[]` : $(\forall \mathtt{t})$ `List t` (liste vide)

Et même : `while true do pass done` : $(\forall \mathtt{t})$ `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 [transp. 8]

▶ **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 (≅ recherche de preuves) :

▶ **Habitation** de type : trouver un *terme* (=expression) ayant un type donné.
P.ex. : y a-t-il un terme de type $(\forall \mathtt{p}, \mathtt{q})\ ((\mathtt{p} \to \mathtt{q}) \to \mathtt{p}) \to \mathtt{p}$?
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 [transp. 9]

▶ 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) [transp. 10]

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 » (→ polymorphisme paramétrique) avec Java 5, puis diverses sortes d'inférence.

▶ Rust : interaction avec la gestion de la mémoire (≈ typage linéaire/affine).

Quelques exemples (2) [transp. 11]

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$^{\text{sme}}$ paramétrique.

Système de « modules » (« signatures » ≈ 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 [transp. 12]

▶ 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 [transp. 13]

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 (← 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) [transp. 14]

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

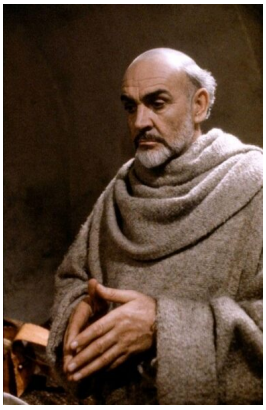
- **types** et **termes** (=programmes) de ces types dans un système de typage,
- **propositions** et **preuves** de ces propositions dans un système logique.

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) [transp. 15]



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) [transp. 16]

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) [transp. 17]

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. □

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) [transp. 18]

▶ 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), → Agda ;
- variantes du « calcul des constructions » (« CoC »), → Coq ;
- théorie homotopique des types (« HoTT ») : « égalité = isomorphisme ».

Le λ-cube de Barendregt (très bref aperçu) [transp. 19]

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.

2 Le λ-calcul simplement typé

Le λ-calcul simplement typé : description sommaire [transp. 20]

▶ 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 [transp. 21]

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 [**transp. 22**]

- 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 [**transp. 23**]

- 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 [transp. 24]

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 [transp. 25]

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 :

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

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 [transp. 26]

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 $\Gamma|_{\text{free}(E)} \vdash E : \tau$ où $\Gamma|_{\text{free}(E)}$ 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 [transp. 27]

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 [transp. 28]

▶ 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.

→ Algorithme de Hindley-Milner (*inférence* de type).

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

→ Décidabilité du calcul propositionnel intuitionniste.

Reconstruction du terme typé [transp. 29]

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

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

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 [transp. 30]

▶ **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 [transp. 31]

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 [transp. 32]

▶ **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).

3 Le calcul propositionnel intuitionniste

Le calcul propositionnel : présentation sommaire [transp. 33]

▶ 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 [transp. 34]

▶ 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 [transp. 35]

▶ 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 [transp. 36]

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

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 [transp. 37]

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.) :

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

$$\dfrac{\dfrac{\dfrac{}{A \vee B \vdash A \vee B}\,\text{Ax}}{A \vee B \vdash B \vee A}\,\vee\text{Élim}}{\vdash A \vee B \Rightarrow B \vee A}\,\Rightarrow\text{Int}$$

Présentation différente [transp. 38]

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) :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B}\,u}{B}\,\wedge\text{Élim}_2 \qquad \dfrac{\dfrac{}{A \wedge B}\,u}{A}\,\wedge\text{Élim}_1}{B \wedge A}\,\wedge\text{Int}}{A \wedge B \Rightarrow B \wedge A}\,\Rightarrow\text{Int}(u)$$

$$\dfrac{\dfrac{\dfrac{}{A \vee B}\,u \qquad \dfrac{\dfrac{}{A}\,v}{B \vee A}\,\vee\text{Int}_2 \qquad \dfrac{\dfrac{}{B}\,v'}{B \vee A}\,\vee\text{Int}_1}{B \vee A}\,\vee\text{Élim}(v,v')}{A \vee B \Rightarrow B \vee A}\,\Rightarrow\text{Int}(u)$$

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

Encore une présentation différente [transp. 39]

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 [transp. 40]

▶ 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 $Q \Rightarrow P$ ») :

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

Quelques non-tautologies [transp. 41]

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 [transp. 42]

- 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 [transp. 43]

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 [transp. 44]

▶ 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).
→ « 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 [transp. 45]

▶ 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$^{\text{t}}$ 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 » ? [transp. 46]

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 $\mathbb{R} \to \mathbb{R}$ est continue », « toute fonction $\mathbb{N} \to \mathbb{N}$ est calculable » [$\leftarrow$ « calculabilité synthétique »]) qu'on peut vouloir étudier.

Preuves classiques pas toujours satisfaisantes [transp. 47]

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 $f\colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ 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 [transp. 48]

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 [transp. 49]

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 [transp. 50]

▶ 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 $(\beta \to \alpha \to \gamma) \to (\alpha \to \beta \to \gamma)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\textsc{Ax}\ \dfrac{}{\overset{B \Rightarrow A \Rightarrow C,}{A, B} \vdash B \Rightarrow A \Rightarrow C} \qquad \dfrac{}{\overset{B \Rightarrow A \Rightarrow C,}{A, B} \vdash B}\ \textsc{Ax}}{\overset{B \Rightarrow A \Rightarrow C,}{A, B} \vdash A \Rightarrow C}\ \Rightarrow\textsc{Élim} \qquad \dfrac{}{\overset{B \Rightarrow A \Rightarrow C,}{A, B} \vdash A}\ \textsc{Ax}}{B \Rightarrow A \Rightarrow C, A, B \vdash C}\ \Rightarrow\textsc{Élim}}{B \Rightarrow A \Rightarrow C, A \vdash B \Rightarrow C}\ \Rightarrow\textsc{Int}}{B \Rightarrow A \Rightarrow C \vdash A \Rightarrow B \Rightarrow C}\ \Rightarrow\textsc{Int}}{\vdash (B \Rightarrow A \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)}\ \Rightarrow\textsc{Int}$$

Correspondance de Curry-Howard : conjonction [transp. 51]

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 [transp. 52]

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

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

▶ 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 [transp. 53]

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 ↓	$\vee\textsc{Élim}\ \dfrac{\Gamma \vdash P_1 \vee P_2 \qquad \Gamma, P_1 \vdash Q \qquad \Gamma, P_2 \vdash Q}{\Gamma \vdash Q}$

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

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 [transp. 54]

▶ 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 [transp. 55]

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

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

▶ 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 [transp. 56]

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 [transp. 57]

▶ 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 [transp. 58]

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 [transp. 59]

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 [transp. 60]

- 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 $\to$ et $\Rightarrow$ comme interchangeables.

La négation et la double négation [transp. 61]

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 [transp. 62]

- 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 [transp. 63]

- ▶ 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 [transp. 64]

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$^{\text{t}}$ 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$^{\text{t}}$ 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 [transp. 65]

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 [transp. 66]

▶ 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 [transp. 67]

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

Exemples de démonstrations en calcul des séquents [transp. 68]

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

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

Et que dans le transp. 50 :

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

Calcul des séquents : règles « structurales » [transp. 69]

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

$$\dfrac{}{Q \vdash Q}\,\mathrm{Ax} \qquad \left|\qquad \dfrac{\Gamma \vdash Q}{\Gamma, P \vdash Q}\,\textsc{Weak}\right.$$

► 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 $(A \Rightarrow B) \wedge A \vdash B$) :

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

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 [transp. 70]

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 $\Rightarrow$Élim et $\Rightarrow$Left :

► Dans le sens déduction naturelle $\rightarrow$ calcul des séquents :

Déduction naturelle	Calcul des séquents
$\dfrac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}\,\Rightarrow\textsc{Élim}$	$\dfrac{\Gamma \vdash P \Rightarrow Q \qquad \dfrac{\Gamma \vdash P \qquad \dfrac{}{\Gamma, Q \vdash Q}\,\mathrm{Ax}}{\Gamma, P \Rightarrow Q \vdash Q}\,\Rightarrow\mathrm{L}}{\Gamma \vdash Q}\,\textsc{Cut}$

► Dans le sens calcul des séquents $\rightarrow$ déduction naturelle :

Calcul des séquents	Déduction naturelle
$\dfrac{\Gamma \vdash M \qquad \Gamma, P \vdash R}{\Gamma, M \Rightarrow P \vdash R}\,\Rightarrow\mathrm{L}$	$\dfrac{\dfrac{\Gamma, P \vdash R}{\Gamma \vdash P \Rightarrow R}\,\Rightarrow\textsc{Intro} \qquad \dfrac{\dfrac{}{\Gamma, M \Rightarrow P \vdash M \Rightarrow P}\,\mathrm{Ax} \qquad \Gamma \vdash M}{\Gamma, M \Rightarrow P \vdash P}\,\Rightarrow\textsc{Élim}}{\Gamma, M \Rightarrow P \vdash R}\,\Rightarrow\textsc{Élim}$

Calcul des séquents : élimination des coupures [transp. 71]

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) [transp. 72]

▶ 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).

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

(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) [transp. 73]

▶ 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 :

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

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

Élimination des coupures : récurrence [transp. 74]

▶ 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 [transp. 75]

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 $\vee\mathrm{R}_1$ ou $\vee\mathrm{R}_2$. □)

▶ **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 [transp. 76]

▶ 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$^{\text{ion}}$).

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

(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 $\mathsf{S}, \mathsf{K}, \mathsf{I}$ [transp. 77]

▶ **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* :

- ▶ $(\mathsf{I}) : A \Rightarrow A$
- ▶ $(\mathsf{K}) : A \Rightarrow B \Rightarrow A$
- ▶ $(\mathsf{S}) : (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$
- ▶ $A \wedge B \Rightarrow A$
- ▶ $A \wedge B \Rightarrow B$
- ▶ $A \Rightarrow B \Rightarrow A \wedge B$
- ▶ $\top$
- ▶ $\bot \Rightarrow C$
- ▶ $A \Rightarrow A \vee B$
- ▶ $B \Rightarrow A \vee B$
- ▶ $(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow (A \vee B \Rightarrow C)$

▶ 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 $\mathsf{K}y$ (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 $\mathsf{S}(\mathsf{S}(\mathsf{KS})\mathsf{K})\mathsf{I}$.

▶ 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.)

4 Le call/cc et la logique classique

Qu'est-ce qu'une continuation ? [transp. 78]

▶ 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 [transp. 79]

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 [transp. 80]

- « 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 [transp. 81]

```
(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 ? [transp. 82]

▶ *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 ! [transp. 83]

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 $A \vee \neg A$:

$$\texttt{callcc}\left(\lambda(k : (\alpha + (\alpha \to 0)) \to 0).\ \iota_2^{(\alpha,\alpha\to 0)}\left(\lambda(v : \alpha).\ k(\iota_1^{(\alpha,\alpha\to 0)} v)\right)\right)$$

$$\text{a pour type :} \quad \alpha + (\alpha \to 0)$$

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 ? [transp. 84]

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 [transp. 85]

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 [transp. 86]

```
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 [transp. 87]

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 [transp. 88]

▶ 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 [transp. 89]

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 [transp. 90]

▶ 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 [transp. 91]

▶ 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 $\langle x, y\rangle^{\mathrm{CPS}} = \lambda k.\, x^{\mathrm{CPS}}(\lambda x_0.\, y^{\mathrm{CPS}}(\lambda y_0.\, k\langle x_0, y_0\rangle))$ qui évalue x avant y, mais on pouvait aussi $\lambda k.\, y^{\mathrm{CPS}}(\lambda y_0.\, x^{\mathrm{CPS}}(\lambda x_0.\, k\langle x_0, y_0\rangle))$ 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 »
$v^{\mathrm{CPS}} = \lambda k.kv$	$v^{\mathrm{CPS}} = \lambda k.vk$ (ou juste v) mais pour une *valeur* : $x^{\mathrm{CPS}} = \lambda k.kx$
$(\lambda v.t)^{\mathrm{CPS}} = \lambda k.\, k(\lambda v.\, t^{\mathrm{CPS}})$	$(\lambda v.t)^{\mathrm{CPS}} = \lambda k.\, k(\lambda v.\, t^{\mathrm{CPS}})$
$(fx)^{\mathrm{CPS}} = \lambda k.\, f^{\mathrm{CPS}}(\lambda f_0.\, x^{\mathrm{CPS}}(\lambda x_0.\, f_0 x_0 k))$	$(fx)^{\mathrm{CPS}} = \lambda k.\, f^{\mathrm{CPS}}(\lambda f_0.\, f_0 x^{\mathrm{CPS}} k)$
$(P \Rightarrow Q)^\diamond = (P^\diamond \Rightarrow \sim\sim Q^\diamond)$	$(P \Rightarrow Q)^\diamond = (\sim\sim P^\diamond \Rightarrow \sim\sim Q^\diamond)$

Continuation Passing Style : remarques informatiques (suite) [transp. 92]

▶ 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$^{\mathrm{t}}$ les évaluations. Voir notamment les promesses de JavaScript, et la monade `Control.Monad.Cont` de Haskell.

Continuation Passing Style : remarques logiques [transp. 93]

▶ 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 :

$$\text{si } \mathsf{CPC} \vdash P \text{ alors } \mathsf{IPC} \vdash P^{\mathrm{CPS}}$$

où « $\mathsf{IPC} \vdash P$ » signifie « P est prouvable en logique intuitionniste » et « $\mathsf{CPC} \vdash P$ » 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

$$\mathsf{CPC} \vdash P \text{ ssi } \mathsf{CPC} \vdash P^{\mathrm{CPS}}$$

Comme par ailleurs $\mathsf{CPC} \vdash Q$ implique $\mathsf{IPC} \vdash Q$, on déduit des deux affirmations ci-dessus que (pour $Z = \bot$) :

$$\mathsf{CPC} \vdash P \textit{ ssi } \mathsf{IPC} \vdash P^{\mathrm{CPS}}$$

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).

5 Sémantique du calcul propositionnel intuitionniste

Qu'est-ce qu'une « sémantique » en logique ? [transp. 94]

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 [transp. 95]

▶ 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 ? [transp. 96]

▶ 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 [transp. 97]

(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 {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 [transp. 98]

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 $\eta_1, \dots, \eta_r \vdash \varphi$ 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 [transp. 99]

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) [transp. 100]

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 [transp. 101]

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 [transp. 102]

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 [transp. 103]

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 $\mathcal{O}(X)$:

- ▶ $U \mathbin{\dot\wedge} V := U \cap V$ ▶ $U \mathbin{\dot\vee} V := U \cup V$ ▶ $\dot\top := X$ ▶ $\dot\bot := \varnothing$
- ▶ $(U \mathbin{\dot\Rightarrow} V) := \mathrm{int}((X \setminus U) \cup V)$ (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 [transp. 104]

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 [transp. 105]

▶ Soit $U =]0,1[$ dans $X = \mathbb{R}$. Alors

- ▶ $\dot\neg U =]-\infty, 0[\cup]1, +\infty[$ ▶ $\dot\neg\dot\neg U =]0,1[= U$ ▶ $(\dot\neg\dot\neg U \mathbin{\dot\Rightarrow} U) = \mathbb{R}$
- ▶ $U \mathbin{\dot\vee} \dot\neg U = \mathbb{R} \setminus \{0,1\}$ ▶ $((\dot\neg\dot\neg U \mathbin{\dot\Rightarrow} U) \mathbin{\dot\Rightarrow} (U \mathbin{\dot\vee} \dot\neg U)) = \mathbb{R} \setminus \{0,1\}$

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 $V_2 = \{x_2 > 0\}$.

- ▶ $\dot\neg U = \{(x_1, x_2) : x_1 > 0 \text{ ou } x_2 > 0\} = V_1 \mathbin{\dot\vee} V_2$ ▶ Donc $(\dot\neg U \mathbin{\dot\Rightarrow} (V_1 \mathbin{\dot\vee} V_2)) = \mathbb{R}^2$
- ▶ $(\dot\neg U \mathbin{\dot\Rightarrow} V_1) = \{(x_1, x_2) : x_1 < 0 \text{ ou } x_2 > 0\}$
- ▶ $(\dot\neg U \mathbin{\dot\Rightarrow} V_2) = \{(x_1, x_2) : x_1 > 0 \text{ ou } x_2 < 0\}$

▶ $(\dot\neg U \mathrel{\dot\Rightarrow} V_1) \mathrel{\dot\vee} (\dot\neg U \mathrel{\dot\Rightarrow} V_2) = \mathbb{R}^2 \setminus \{(0,0)\}$

Ceci montre que $(\neg A \Rightarrow B_1) \vee (\neg A \Rightarrow B_2) \Rightarrow (\neg A \Rightarrow (B_1 \vee B_2))$ (« 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 [transp. 106]

On reprend les notations de la calculabilité, mais on notera $\Phi_e \colon \mathbb{N} \dashrightarrow \mathbb{N}$ 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 $\mathbb{N}$:

- $P \mathrel{\dot\wedge} Q = \{\langle m, n\rangle : m \in P, n \in Q\}$ (codage de Gödel du produit $P \times Q$)
- $P \mathrel{\dot\vee} Q = \{\langle 0, m\rangle : m \in P\} \cup \{\langle 1, n\rangle : n \in Q\}$ (sorte de réunion disjointe)
- $(P \mathrel{\dot\Rightarrow} Q) = \{e \in \mathbb{N} : \Phi_e(P)\!\downarrow \subseteq Q\}$, 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 $\varphi(A_1, \dots, A_r)$ est une formule propositionnelle et $P_1, \dots, P_r \in \mathbb{N}$, on définit $\dot\varphi(P_1, \dots, P_r)$ de façon évidente (par induction).

▶ Si $n \in \dot\varphi(P_1, \dots, P_r)$, on dit aussi que n **réalise** $\varphi(P_1, \dots, P_r)$.

La réalisabilité propositionnelle [transp. 107]

▶ 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 $\varphi(A_1, \dots, A_r)$ est **réalisable** (plus précisément, « uniformément réalisable ») lorsqu'il existe un *même entier* n qui réalise $\varphi(P_1, \dots, P_r)$ quels que soient $P_1, \dots, P_r \subseteq \mathbb{N}$:

$$n \in \bigcap_{P_1,\dots,P_r} \dot\varphi(P_1, \dots, P_r)$$

▶ **Théorème** (Dragalin, Troelstra, Nelson) : la sémantique de la réalisabilité est correcte pour le calcul propositionnel intuitionniste : si $\varphi(A_1, \dots, A_r)$ est un théorème du calcul propositionnel intuitionniste, alors il existe n qui réalise $\varphi(P_1, \dots, P_r)$ quels que soient $P_1, \dots, P_r \subseteq \mathbb{N}$.

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 [transp. 108]

▶ Essayons de réaliser $A \wedge B \Rightarrow B \wedge A$: on cherche donc un même entier dans $P \mathrel{\dot\wedge} Q \mathrel{\dot\Rightarrow} Q \mathrel{\dot\wedge} P$ pour *tous* $P, Q \subseteq \mathbb{N}$.

▶ Autrement dit, on veut trouver un programme e tel que Φ_e soit défini sur $P \mathrel{\dot\wedge} Q = \{\langle m, n\rangle : m \in P, n \in Q\}$ et l'envoie dans $Q \mathrel{\dot\wedge} P = \{\langle n, m\rangle : n \in Q, m \in P\}$ (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 [transp. 109]

▶ 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 2$^{\text{e}}$ problème en une solution du 1$^{\text{er}}$.

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 [transp. 110]

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 [transp. 111]

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 [transp. 112]

▶ 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 :

- ▶ $(X, S) \mathbin{\dot\wedge} (Y, T) = (X \times Y, S \times T)$ (produit cartésien)
- ▶ $(X, S) \mathbin{\dot\vee} (Y, T) = (X \uplus Y, S \uplus T)$ (réunion disjointe)
- ▶ $(X, S) \mathbin{\dot\Rightarrow} (Y, T) = (Y^X, U)$ où $U := \{f \colon X \to Y : f(S) \subseteq T\}$ (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 [transp. 113]

▶ 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., $(\neg A \Rightarrow B_1) \vee (\neg A \Rightarrow B_2) \Rightarrow (\neg A \Rightarrow (B_1 \vee B_2))$ (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.

6 Inférence de type à la Hindley-Milner

L'algorithme de Hindley-Milner : description sommaire [transp. 114]

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 $\lambda x f.\, f(\lambda h.hx)$.

▶ 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 [transp. 115]

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 [transp. 116]

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 [transp. 117]

▶ 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 [transp. 118]

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 [transp. 119]

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 [transp. 120]

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 ($\to$ nvx cas d'échecs d'unification, p.ex., si $\zeta_1 = (? \to ?)$ et $\zeta_2 = (? \times ?)$).

Types récursifs ? [transp. 121]

▶ 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 » [transp. 122]

▶ 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 » [transp. 123]

▶ 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.