Correspondance preuve/programme entre la logique propositionnelle et le $\lambda \to +\times$, et au delà (la meilleure façon de faire de la déduction naturelle)

Pierre Karpman

Lycée Champollion TD XENS

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

De quoi parlons nous (dans le contexte de la logique prop.) ?

Correspondance théorème/type (ou formule/type)

- Les théorèmes (formules vraies) correspondent à des types de programmes

Correspondance preuve/programme

- Les preuves (des formules) correspondent aux programmes (dont les types correspondent aux formules)

Correspondance/isomorphisme de Curry-Howard

- si l'on tient à nommer ça d'après des personnes, de façon à ne plus rien comprendre

Concrètement ? Un premier exemple : $\vdash A \to (A \to B) \to B$

1. Par introduction de l'implication, on se ramène à prouver $A, A \to B \vdash B$
2. Ce qui est vrai par élimination de l'implication…

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B, A \vdash A \to B}\,ax \qquad \dfrac{}{A \to B, A \vdash A}\,ax}{A \to B, A \vdash B}\,\to_e}{A \vdash (A \to B) \to B}\,\to_i}{\vdash A \to ((A \to B) \to B)}\,\to_i$$

(Pour écrire des formules correctes en LaTeX sans se fatiguer : https://github.com/MarcdeFalco/dedunat/)

Observation

L'élimination de l'implication, qui de A et $A \to B$ déduit B ressemble furieusement à une application de fonction de type $\alpha \to \beta$ à un argument de type α pour obtenir une valeur de type β !

Un premier exemple, version type : $\alpha \to (\alpha \to \beta) \to \beta$

On a :

```
let f1rst : 'a -> ('a -> 'b) -> 'b = fun x f -> f x
```

Correspondance

- Le type de `f1rst` correspond littéralement à la proposition $A \to (A \to B) \to B$
- Le programme donné prouve que ce type est « réalisable »

Comme cette preuve est compacte !

Pourquoi c'est bien une preuve ?

- Les règles appliquées pour typer `fun x f -> f x` sont *les mêmes* que l'intro/élim de l'implication

Intro

- Si une expression `e1` a un type β en utilisant une variable libre `x` de type α (prouve $A \vdash B$) on peut écrire une fonction `f1` de type $\alpha \to \beta$ qui prend un argument `y` de type α et le substitue à `x` dans `e1` (prouve $\vdash A \to B$) (et vice-versa)

seul le *type* compte ici ; le fait que `f1` et `e1` ne s'évaluent pas forcément en les mêmes valeurs n'a aucune importance

Élim

- Si l'on dispose (comme hypothèse, par exemple *via* intro de l'implication) d'une fonction `f` de type $\alpha \to \beta$ et d'une valeur `x` de type α (on a $A, A \to B \vdash$) alors l'application `f x` est possible et a type β (prouve $A, A \to B \vdash B$)

Un second exemple : *combinateur S* ou « théorème de Frege » (log. prop.)

Preuve en style *déduction naturelle*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to (B \to C) \vdash A \to (B \to C)}\,ax \quad \dfrac{}{A \vdash A}\,ax}{A, A \to (B \to C) \vdash B \to C}\,\to_e \quad \dfrac{\dfrac{}{A \to B \vdash A \to B}\,ax \quad \dfrac{}{A \vdash A}\,ax}{A, A \to B \vdash B}\,\to_e}{A, A \to B, A \to (B \to C) \vdash C}\,\to_e}{A \to B, A \to (B \to C) \vdash A \to C}\,\to_i}{A \to (B \to C) \vdash (A \to B) \to (A \to C)}\,\to_i}{\vdash (A \to (B \to C)) \to ((A \to B) \to (A \to C))}\,\to_i$$

Preuve par programme

Beaucoup plus compact (et encore, tout est annoté et en style verbeux !), et on voit mieux ce qui se passe (qui correspond à une opération importante dans un modèle de calcul fonctionnel minimaliste)

```
let s : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c =
  fun (f : 'a -> 'b -> 'c) ->
  fun (g : 'a -> 'b) ->
  fun (x : 'a) -> f x (g x)
```

Un programme prouve son type… vraiment ?

$\vdash A \to B$??

On a les deux fonctions OCaml :

- `let w1 : 'a -> 'b = fun x -> failwith "hewwo"`
- `let rec w2 : 'a -> 'b = fun x -> w2 x`

de type $\alpha \to \beta$. Mais $\vdash A \to B$ n'est pas un théorème (de la logique classique) !

Que s'est-il passé ?

- On a levé une exception (`w1`)
  - échappe au typage
- On a utilisé un `rec` (`w2`)
  - qui fait intervenir une règle de typage supplémentaire avec point fixe, qui permet de ne plus terminer

(Deux exemples d'*effets*)

- Dans les deux cas l'*évaluation* de la fonction ne termine pas ; le type n'est plus obtenu en appliquant des règles qui correspondent à celles du raisonnement logique : la correspondance théorème/type (et donc preuve/programme) n'existe plus !

Quels sont les programmes pour lesquels la correspondance tient ?

λ-calcul simplement typé ou λst ou $\lambda \to$

$\lambda \to$

- Abstraction « λ-abstraction » : $\lambda(x : \alpha).E$ introduit x variable libre de type α dans une expression E (de λ-calcul, définie inductivement), en respectant les types !
- (Renommage « α-conversion »)
- Application (substitution) « β-réduction » : soit F de la forme $\lambda(x : \alpha).E$ et X une expression quelconque, l'application $F\,X$ peut être réécrite en E où X est substitué aux occurrences de x (si les types sont respectés !) (en renommant ce qu'il faut si nécessaire)

Exemple :

$$\lambda(x : \alpha).\lambda(f : \alpha \to \beta).f\,x$$

$\rightsquigarrow$ λ-expression (ou λ-terme) valide de type $\alpha \to (\alpha \to \beta) \to \beta$ (admis) !

Normalisation forte

Le $\lambda \to$ est *fortement normalisant* : toute suite de réduction termine !
(Faux en l'absence de type ; pour d'autres systèmes de type : le λ-calcul non typé a la même puissance de calcul que les machines de Turing)

$\lambda \rightarrow$ en OCaml

`fun fun fun`

Le $\lambda \rightarrow$ correspond au fragment d'OCaml constitué des seules fonctions anonymes `fun` ($\equiv \lambda$) (et éventuellement du seul type natif `unit`)

- Correspondance immédiate :

$$\lambda(x : \alpha).E \quad \leftrightarrow \quad \texttt{fun (x:'a) -> E}$$
$$F\,X \quad \leftrightarrow \quad \texttt{f x}$$

- Mêmes règles de calcul & de typage
- Qui cette fois respectent la correspondance voulue (admis) !
  - Idée de preuve : on a enlevé tout effet gênant ; il suffit d'expliciter la corr. entre les règles de typage restantes & celles de déduction

Et donc seulement $\rightarrow$?

On peut *enrichir* le λ-calcul de types *algébriques* traduisant $\vee$ et $\wedge$

- Type *somme* $x : \alpha + \beta$ est de type α ou (exclusif) β
  - `type ('a, 'b) sum = L of 'a | R of 'b`
- Type *produit* $x : \alpha \times \beta$ a deux composantes, l'une de type α et l'autre de type β
  - `'a * 'b`

Correspondance $+ \leftrightarrow \vee$; $\times \leftrightarrow \wedge$

$+ \leftrightarrow \vee$

- Intro : si l'on veut introduire (créer) une expression valant $A \vee B$ (de type `('a, 'b) sum`), il suffit d'en disposer d'une valant A (de type `'a`) ou B (`'b`)
- Élim : pour utiliser $A \vee B$ (`('a, 'b) sum`), il faut traiter les *deux* cas A et B (`'a` et `'b`)

$\times \leftrightarrow \wedge$

- Intro : si l'on veut introduire (créer) une expression valant $A \wedge B$ (de type `'a * 'b`), il faut disposer des deux expressions valant A (de type `'a`) et B (`'b`)
- Élim : pour utiliser $A \wedge B$ (`'a * 'b`), on peut choisir de garder seulement A ou B (`'a` ou `'b`)

En OCaml

Avec du pattern-matching *exhaustif* (comme d'habitude ; et `fst`, `snd` si l'on veut) !

Exemple : un *fold* : $(\alpha \to \gamma) \to (\beta \to \gamma) \to (\alpha + \beta) \to \gamma$

Preuve en style *déduction naturelle*

$$\dfrac{\dfrac{\dfrac{}{A \vee B \vdash A \vee B}\,ax \qquad \dfrac{\dfrac{}{A \to C \vdash A \to C}\,ax \qquad \dfrac{}{A \vdash A}\,ax}{A, A \to C \vdash C}\,\to_e \qquad \dfrac{\dfrac{}{B \to C \vdash B \to C}\,ax \qquad \dfrac{}{B \vdash B}\,ax}{B, B \to C \vdash C}\,\to_e}{A \vee B, B \to C, A \to C \vdash C}\,\vee_e}{B \to C, A \to C \vdash (A \vee B) \to C}\,\to_i}{A \to C \vdash (B \to C) \to ((A \vee B) \to C)}\,\to_i}{\vdash (A \to C) \to ((B \to C) \to ((A \vee B) \to C))}\,\to_i$$

Preuve par programme

```
let fold : ('a -> 'c) -> ('b -> 'c) -> ('a, 'b) sum -> 'c =
    fun fl fr -> function L x -> fl x | R x -> fr x
```

Plus compact, et la formule prouvée correspond à une fonction de base sur le type !

Exemple « *split* » : $(\alpha + \beta \to \gamma) \to (\alpha \to \gamma) \wedge (\beta \to \gamma)$

Preuve en style *déduction naturelle*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \vee B) \to C \vdash (A \vee B) \to C}\,ax \quad \dfrac{\dfrac{}{A \vdash A}\,ax}{A \vdash A \vee B}\,\vee_{ig}}{(A \vee B) \to C \vdash A \to C}\,\to_e}{A, (A \vee B) \to C \vdash C}\,\to_i}{(A \vee B) \to C \vdash (A \to C) \wedge (B \to C)}\,\wedge_i}{\vdash ((A \vee B) \to C) \to ((A \to C) \wedge (B \to C))}\,\to_i$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \vee B) \to C \vdash (A \vee B) \to C}\,ax \quad \dfrac{\dfrac{}{B \vdash B}\,ax}{B \vdash A \vee B}\,\vee_{id}}{(A \vee B) \to C \vdash B \to C}\,\to_e}{B, (A \vee B) \to C \vdash C}\,\to_i}{(A \vee B) \to C \vdash (A \to C) \wedge (B \to C)}\,\wedge_i}{\vdash ((A \vee B) \to C) \to ((A \to C) \wedge (B \to C))}\,\to_i$$

Preuve par programme

```
let split : (('a, 'b) sum -> 'c) -> (('a -> 'c) * ('b -> 'c)) =
  fun f -> (fun x -> f (L x)), (fun x -> f (R x))
```

La formule traduit à nouveau un concept de programmation utile !

$\top \bot \neg$

Valeurs spéciales : $\top$ et $\bot$

La logique propositionnelle contient également les propositions atomiques :

- $\top$: « top » (vrai), avec la règle $\vdash \top$
- $\bot$: « bottom » (faux), permet d'écrire la négation

Négation $\neg$

Le plus simple : $\neg A := A \to \bot$

Version programme

- $\top$: n'importe quel type *habité* par un littéral (dont on connaît une valeur) : le plus simple : `type top = unit`
  - Sera notamment utile pour exprimer des formules non constructives (version OCaml) comme implication `top -> ...` (*cf. infra*)
- $\bot$: le type *vide* (pour lequel il n'existe aucune valeur) : `type bot = |`
- $\neg$: `type 'a not = 'a -> bot`

Exemple : $(\alpha \to \neg\alpha) \to \neg\alpha$

(Trivial à prouver en utilisant le *tiers-exclu*, mais aussi vrai ici en *logique minimale* !)

Réécriture, inférence

- Première réécriture (équivalente) : $(\alpha \to \alpha \to \bot) \to (\alpha \to \bot)$
  - Prouvé par `fun (f : 'a -> 'a -> bot) -> fun (x : 'a) -> f x x`
- Mais si l'on retire les annotations, l'inférence donne le type plus général `('a -> 'a -> 'b) -> ('a -> 'b)` pour cette expression !
- L'inférence de type en $\lambda \to$ est calculable et peut trouver le type le plus général (par ex. avec les algorithmes W/J) !
  - On peut se contenter de garder les programmes, sans annotations (comme d'habitude au final) : les théorèmes *se déduisent mécaniquement des preuves* !
  - On peut sur-contraindre pour « spécialiser » une formule, si l'on veut

Coupures

- On aurait aussi pu faire `fun f -> s f (fun x -> x)` !
  - `s : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c` un combinateur S
- On réutilise un théorème déjà montré (*coupure*), simplement en appelant le programme correspondant !

Explosion

$\lambda \to +\times$: logique *minimale*

- Une logique constructive ! Une preuve d'existence d'un type donne un programme (qui termine !) qui *construit* ce type

$\lambda \to +\times$ + explosion : logique *intuitionniste*

- Toujours constructive
- Ajoute un axiome d'*explosion* : $\vdash \bot \to A$: de faux on déduit ce qu'on veut !
- Version programme : `let explo : bot -> 'a = function _ -> .`

Logique classique : le retour éternel des continuations !

`call/cc`

- On peut définir une fonction `call-with-current-continuation` « `call/cc` » t.q. `call/cc f` appelle `f` avec comme argument la continuation du contexte d'appel et renvoie ce que renvoie `f`, ou l'argument passé à la continuation
  - la continuation est « fournie par le langage » (n'existe pas en OCaml, mais par ex. en Scheme)
- Contraintes de types :
  - continuation : argument de type α (contexte local), ne termine jamais $\rightsquigarrow \alpha \to \beta$
  - `f` doit renvoyer le même type que l'argument de la continuation : « le type qui a du sens à ce point du contexte », donc de type $(\alpha \to \beta) \to \alpha$
  - `call/cc` renvoie le résultat de `f` ou l'argument de la continuation (forcément du même type α) : type $((\alpha \to \beta) \to \alpha) \to \alpha$

Terminaison

- On peut montrer que $\lambda \to +\times$ + explosion + `call/cc` (ainsi typé) est toujours fortement normalisant
- On a rajouté des effets mais de façon « cadrée » : on ne peut pas faire n'importe quoi, les types ont toujours un peu de sens : pas de $\alpha \to \beta$ par non-terminaison !

`call/cc` ↔ « Loi de Peirce »

Version logique

- Le type de `call/cc` correspond à la formule $(A \to B) \to A) \to A$: « loi de Peirce »
- Non prouvable en logique intuitionniste, mais un axiome possible pour la logique « classique »
  - Par ex. équivalent au « tiers-exclu » ou à l'élimination de la double négation
- Fait perdre le côté constructif (les termes construits dépendent d'un contexte opaque (qui n'est pas stable tout au long de l'exécution) : plus très utile)

Comparaison « *consequentia mirabilis* »

- *consequentia mirabilis* : $((\alpha \to \bot) \to \alpha) \to \alpha$
  - « si supposer qu'α est faux permet de déduire α, alors α est vrai [car il ne peut pas être à la fois faux et vrai : pur raisonnement classique en action] »
- `call/cc` : $(\alpha \to \beta) \to \alpha) \to \alpha$
  - Pareil en plus général : $\alpha \to \beta$ revient à avoir supposé qu'α est faux puis utilisé l'explosion

« Implémentation » OCaml

Fonction avec le type de `call/cc`

- On peut tricher en définissant :

```
let rec ccc : (('a -> 'b) -> 'a) -> 'a = fun x -> ccc x
```

  - contrairement au vrai `call/cc`, ne terminera jamais !
  - mais peut tout de même être utilisé dans des preuves, qui donneront des types valides en logique classique
  - pas grave si l'évaluation d'une expression ne termine pas : le typage reste décidable !
  - en cas de coupure, on peut utiliser `top` pour retarder (indéfiniment) l'évaluation

Tout un monde de logiques intermédiaires

Par la correspondance preuve/programme

- $\lambda \to +\times$: logique minimale
- $-$ + explosion : logique intuitionniste
- $-$ + `call/cc` (typé) : logique classique

Autre approche « programme » : la réalisabilité propositionnelle

- Les programmes ne doivent plus nécessairement terminer ; ce ne sont plus leurs types qui donnent les théorèmes
- Donne une logique intermédiaire entre intuitionniste et classique
- Exemple, *via* une énigme pas facile du tout (avec un dragon)

Au final, à quoi cela sert-il ?

Programmes : preuves faciles à manipuler (mécaniquement)

- Exemple : l'inférence de type (vérification + éventuelle généralisation)
- Coupures faciles à utiliser
  - Et normalisation forte $\rightsquigarrow$ gratuitement un théorème d'élimination des coupures
- Approche utilisée dans les assistants de preuves comme Coq, Agda, Lean (avec un modèle de calcul plus puissant ! Permet d'aller au delà de la logique propositionnelle)

Extraction de programmes

- D'une preuve, on peut extraire un programme qui la réalise
- Application en vérification logicielle : on peut prouver puis obtenir le programme voulu (plutôt que l'inverse)

Ressources

- Cours de Xavier Leroy au Collège de France 2018–2019
- Notes de cours de David Madore à Télécom'

Un dernier exemple : tiers-exclu doublement nié !

Presque trop simple !

```
let nnem : ('a, 'a not) sum not not =
    fun f -> f (R (fun x -> f (L x)))
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a, \neg(a \vee \neg a) \vdash a}\, ax}{a, \neg(a \vee \neg a) \vdash a \vee \neg a}\, \vee_{ig} \qquad \dfrac{}{a, \neg(a \vee \neg a) \vdash \neg(a \vee \neg a)}\, ax}{a, \neg(a \vee \neg a) \vdash \bot}\, \neg_e}{\neg(a \vee \neg a) \vdash \neg a}\, \neg_i}{\neg(a \vee \neg a) \vdash a \vee \neg a}\, \vee_{id} \qquad \dfrac{}{\neg(a \vee \neg a) \vdash \neg(a \vee \neg a)}\, ax$$

$$\dfrac{\neg(a \vee \neg a) \vdash \bot}{\vdash \neg\neg(a \vee \neg a)}\, \neg_i \qquad \neg_e$$