Programmer = démontrer?

La correspondance de Curry-Howard aujourd'hui

Deuxième cours

Polymorphisme à tous les étages!
Du système F au calcul des constructions

Xavier Leroy

Collège de France

2018-11-21

COLLÈGE
DE FRANCE
1530

Curry-Howard en 1970

Un isomorphisme entre λ-calcul simplement typé et logique intuitionniste qui met en correspondance

- types et propositions ;
- termes et démonstrations ;
- réduction et élimination des coupures.

Dans ce deuxième cours, nous allons voir comment :

- Cette correspondance s'étend à des systèmes de types plus expressifs et à des logiques plus puissantes.
- Cette correspondance a inspiré des systèmes formels qui sont à la fois logiques et langages de programmation (théorie des types de Martin-Löf, calcul des constructions, *Pure Type Systems*).

I

Polymorphisme et second ordre

Typage statique et généricité

Le typage statique avec des types simples (comme dans le lambda-calcul simplement typé, mais aussi comme dans Algol, Pascal, etc) force parfois à dupliquer du code.

Exemple

Un algorithme de tri s'applique à toute liste $\texttt{list}(t)$ d'éléments de type t, pourvu qu'on lui passe aussi la fonction $t \to t \to \texttt{bool}$ qui compare deux éléments de type t.

Avec des types simples, pour trier des listes d'entiers et des listes de chaînes, il faut deux fonctions avec des types différents, même si elles implémentent le même algorithme :

$$\texttt{sort\_list\_int} : (\texttt{int} \to \texttt{int} \to \texttt{bool}) \to \texttt{list}(\texttt{int}) \to \texttt{list}(\texttt{int})$$

$$\texttt{sort\_list\_string} : (\texttt{string} \to \texttt{string} \to \texttt{bool}) \to \texttt{list}(\texttt{string}) \to \texttt{list}(\texttt{string})$$

Typage statique et généricité

Tension forte entre typage statique et implémentations réutilisables d'algorithmes génériques.

Certains langages choisissent d'affaiblir le typage statique, p.ex. en introduisant un type universel ≪any≫ ou ≪ ?≫ avec vérifications dynamiques de types, ou même en supprimant tout typage :

```
void qsort(void * base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Au contraire, le typage polymorphe enrichit l'algèbre de types et assouplit les règles de typage jusqu'à pouvoir donner un type précis à la fonction générique.

Le lambda-calcul polymorphe

(John C. Reynolds, *Towards a theory of type structure*, 1974)

> *We suggest that a solution to [the polymorphic sort function] problem is to permit types themselves to be passed as a special kind of parameter, whose usage is restricted in a way which permits the syntactic checking of type correctness.*

Ajouter au lambda-calcul simplement typé la possibilité d'abstraire sur une variable de type et d'appliquer une telle abstraction à un type :

Termes : $M, N ::= x \mid \lambda x{:}t.\, M \mid M\ N$

$\mid \Lambda X.\, M$ abstraction sur le type X

$\mid M[t]$ instantiation au type t

Types : $t ::= X \mid t_1 \to t_2 \mid \forall X.\, t$

Le lambda-calcul polymorphe

Reprenant l'exemple du tri de listes, la fonction générique de tri peut recevoir le type

$$\texttt{sort\_list} : \forall X.\ (X \to X \to \texttt{bool}) \to \texttt{list}(X) \to \texttt{list}(X)$$

Son implémentation est de la forme

$$\texttt{sort\_list} = \Lambda X.\ \lambda cmp : X \to X \to \texttt{bool}.\ \lambda l : \texttt{list}(X).\ M$$

Elle peut être utilisée pour des listes d'entiers et pour des listes de chaînes par simple instantiation :

$$\texttt{sort\_list}[\texttt{int}] : (\texttt{int} \to \texttt{int} \to \texttt{bool}) \to \texttt{list}(\texttt{int}) \to \texttt{list}(\texttt{int})$$

$$\texttt{sort\_list}[\texttt{string}] : (\texttt{string} \to \texttt{string} \to \texttt{bool}) \to \texttt{list}(\texttt{string}) \to \texttt{list}(\texttt{string})$$

Règles de typage et de réduction

Les règles du lambda-calcul simplement typé :

$$\Gamma_1, x : t, \Gamma_2 \vdash x : t \qquad \frac{\Gamma, x : t \vdash M : t'}{\Gamma \vdash \lambda x{:}t.\, M : t \to t'} \qquad \frac{\Gamma \vdash M : t \to t' \qquad \Gamma \vdash N : t}{\Gamma \vdash M\ N : t'}$$

Plus des règles d'introduction et d'élimination du polymorphisme :

$$\frac{\Gamma \vdash M : t \qquad X \text{ non libre dans } \Gamma}{\Gamma \vdash \Lambda X.\, M : \forall X.\, t} \qquad \frac{\Gamma \vdash M : \forall X.\, t}{\Gamma \vdash M[t'] : t\{X \leftarrow t'\}}$$

Une nouvelle forme de β-réduction :

$$(\Lambda X.\, M)[t] \quad \to_\beta \quad M\{X \leftarrow t\}$$

Types abstraits

Le problème dual de la réutilisation de fonctions génériques est celui de l'indépendance vis-à-vis des représentations d'un type abstrait.

Exemple (Le type abstrait des ensembles d'entiers)

C'est la donnée d'un type `IS` (*integer set*) et des constantes et opérations suivantes :

```
{ empty: IS;  add: int -> IS -> IS;  member: int -> IS -> bool }
```

Plusieurs implémentations pour `IS` et ses opérations sont possibles, p.ex `IS` peut être un vecteur de bits, ou un arbre binaire de recherche.

Pour laisser toute liberté à changer l'implémentation, on voudrait que les codes utilisant ce type `IS` ne dépendent pas de son implémentation (vecteur de bits ou arbre binaire ?), et n'utilisent que les constantes et opérations `empty`, `add`, `member`.

Abstraction de types

(John C. Reynolds, *Towards a theory of type structure*, 1974)

Reynolds observe qu'un moyen de cacher la représentation d'un type abstrait est de rendre ses utilisateurs polymorphes vis-à-vis de son type de représentation.

Exemple (Utiliser un ensemble d'entiers)

```
let use_intset : ∀IS. {...} -> bool =
  ΛIS. λops: { empty: IS; add: int -> IS -> IS;
               member: int -> IS -> bool }.
  ops.member 1 (ops.add 2 ops.empty)
```

Le type `IS` étant une variable de type, le seul moyen de construire et d'utiliser des valeurs de type `IS` est via les opérations `ops`.

Le lambda-calcul polymorphe en pratique

Polymorphisme de deuxième classe

($\approx$ définitions polymorphes, mais valeurs monomorphes) :

- Les génériques en Ada, Java, C#.
- Le typage de Hindley-Milner dans les langages de la famille ML (SML, OCaml, Haskell, etc), avec inférence des types, des Λ et des instantiations :

Schémas de types : $\sigma ::= \forall \alpha_1, \ldots \alpha_n . \tau$ pour les var. liées par `let`

Types simples : $\tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \ldots$ pour tout le reste

Polymorphisme de première classe

($\approx$ les paramètres de fonctions peuvent avoir des types $\forall$) :

- Extensions récentes d'OCaml et de Haskell, p.ex en OCaml

```
type poly_id = { id : 'a. 'a -> 'a }
```

Système F

Quelques années avant Reynolds, vers 1970–1971, le logicien Jean-Yves Girard avait inventé le même lambda-calcul typé polymorphe sous le nom de ≪système F≫.

Les motivations de Girard n'étaient pas la programmation générique, mais l'étude de l'arithmétique du second ordre via une interprétation fonctionnelle (à la BHK).

Girard connaissait le manuscrit de Howard et reconnaît son influence sur ses travaux :

> *Le système F, contrairement au lambda-calcul simplement typé, est construit autour de Curry-Howard. C'est l'image isomorphe du calcul propositionnel intuitionniste du second ordre.*
>
> *(Jean-Yves Girard, Le point aveugle, t.1 ch.6)*

Arithmétique du premier et du second ordre

Arithmétique du premier ordre :

- entiers de Peano (zéro et successeur)
- calcul des prédicats
- quantification $\forall$, $\exists$ sur des entiers uniquement.

Arithmétique du second ordre : la même chose plus

- quantification sur des ensembles d'entiers,
  i.e. sur des prédicats $\mathbb{N} \to \texttt{Prop}$,
  i.e. sur des réels.

L'arithmétique du second ordre suffit à exprimer une grande partie de l'analyse.

Interprétations fonctionnelles de l'arithmétique

(Jean-Yves Girard, *Une extension de l'interprétation fonctionnelle de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types*, 1971)

(Jean-Yves Girard, *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, thèse d'Etat, U. Paris 7, 1972)

Gödel (1958) avait introduit le ≪système T≫ (= lambda-calcul simplement typé + récursion primitive) pour développer une interprétation fonctionnelle (à la BHK) pour l'arithmétique du premier ordre.

Comme les titres de ses publications de 1971 et 1972 le suggèrent, Girard suit une approche similaire pour l'arithmétique du second ordre.

En montrant la normalisation de son système F, il va aussi démontrer une conjecture de Takeuti (1953) concernant l'élimination des coupures pour l'arithmétique du second ordre

Expressivité du système F

Dans système T, Gödel avait dû ajouter les entiers et la récursion comme primitives au lambda-calcul simplement typé, car ce dernier ne peut pas les encoder.

Dans système F, ces notions et d'autres peuvent être exprimées par des codages similaires à ceux du lambda-calcul pur, et que système F est capable de typer de manière suffisamment polymorphe :

- Entiers naturels : $\texttt{nat} \equiv \forall X.\, X \to (X \to X) \to X$
- Booléens : $\texttt{bool} \equiv \forall X.\, X \to X \to X$
- Connecteurs logiques :
  $\bot \equiv \forall X.\, X$
  $A \wedge B \equiv \forall X.\, (A \to B \to X) \to X$
  $A \vee B \equiv \forall X.\, (A \to X) \to (B \to X) \to X$
- Quantificateurs : $\exists X.\, A \equiv \forall Y.\, (\forall X.\, A \to Y) \to Y$

Normalisation du système F

Théorème (Normalisation forte de F)

Si $\Gamma \vdash M : t$, alors toute séquence de β-réductions issue de M est finie.

La démonstration est infiniment plus difficile que celle pour le lambda-calcul simplement typé :

- Le système F est imprédicatif : dans $\forall X.t$, la variable X peut être instanciée par n'importe quel type, y compris $\forall X.t$.
- Cela casse la récurrence sur les types utilisée dans la démonstration classique de normalisation pour les types simples (Tait, 1967).

(Nous reviendrons sur ce point dans le cours sur les relations logiques.)

Un premier succès pour Curry-Howard

Les travaux sur le système F montrent la correspondance de Curry-Howard à l'œuvre, de deux manières différentes :

- A priori : Girard imagine le système F en cherchant le langage typé qui correspond à une logique préexistante. En montrant la normalisation forte du langage, il démontre une conjecture difficile de logique.
- A posteriori : Girard et Reynolds arrivent au même système de types, l'un en cherchant à résoudre des problèmes de logique, l'autre en cherchant à résoudre des problèmes de programmation.

II

Autres dimensions du polymorphisme : ordre supérieur, types dépendants

Constructeurs de types

Dans les exemples nous utilisons souvent des expressions de types $\texttt{list}(t)$, où `list` est un constructeur de type à un paramètre, c.à.d. une fonction des types dans les types : $t \mapsto \texttt{list}(t)$.

Caml, Haskell, et tous les langages fonctionnels typés fournissent des mécanismes pour définir des constructeurs de types, avec zéro, un, ou plusieurs paramètres :

```
type nat = O | S of nat
type 'a list = Nil | Cons of 'a * 'a list
type ('a, 'b) alist = ('a * 'b) list
```

Abstraire sur un constructeur de types

Il est naturel de vouloir abstraire (par un Λ) sur un constructeur de types et non juste sur un type.

Exemple (Type abstrait des piles)

Les piles (*stacks*) contenant des éléments de type *t* se représentent par un constructeur de types `STK` à un paramètre et par les opérations

```
empty: ∀t. STK(t)
push:  ∀t. t -> STK(t) -> STK(t)
pop:   ∀t. STK(t) -> option (t * STK(t))
```

Pour représenter ≪à la Reynolds≫ les clients de ce type abstrait, il faut abstraire sur le constructeur de types `STK` :

```
let use_stack =
  ΛSTK. λops: { empty: ∀t. STK(t); ... }.
```

Le système F_ω

F_ω (Girard, 1972) est une extension du système F où les expressions de types incluent des fonctions des types dans les types (i.e. des constructeurs de types), et où les termes peuvent abstraire (par Λ) sur toutes les sortes de types.

Termes :	$M, N ::= x \mid \lambda x{:}t.\, M \mid M\ N$		
	$\mid \Lambda X :: k.\, M$		abstraction sur le type X de sorte k
	$\mid M[t]$		instantiation au type t
Types :	$t ::= X \mid t_1 \to t_2$		
	$\mid \forall X :: k.\, t$		
	$\mid \lambda X :: k.\, t$		constructeur de type de paramètre X
	$\mid t_1\ t_2$		application d'un constructeur
Sortes :	$k ::= *$		type ordinaire
	$\mid k_1 \Rightarrow k_2$		constructeur de type

(P.ex. le constructeur de types `list` a pour sorte $* \Rightarrow *$)

Sortes et types

Les sortes (*kinds*) classifient les types et empêchent les types mal formés de la même manière que les types classifient les termes.

Jugement $\Gamma \vdash t :: k$ (le type t est bien formé et est de la sorte k) :

$$\Gamma_1, X :: k, \Gamma_2 \vdash X :: k$$

$$\frac{\Gamma, X :: k \vdash t :: k'}{\Gamma \vdash \lambda X :: k.\, t :: k \Rightarrow k'} \qquad \frac{\Gamma \vdash t_1 :: k \Rightarrow k' \qquad \Gamma \vdash t_2 :: k}{\Gamma \vdash t_1\ t_2 :: k'}$$

$$\frac{\Gamma \vdash t_1 :: * \qquad \Gamma \vdash t_2 :: *}{\Gamma \vdash t_1 \to t_2 :: *} \qquad \frac{\Gamma, X :: k \vdash t :: *}{\Gamma \vdash \forall X :: k.t :: *}$$

On retrouve le lambda-calcul simplement typé, juste ≪décalé vers le haut≫ et appliqué au niveau des types et des sortes.

La correspondance de Curry-Howard augmentée

Système F_ω	logique intuitionniste
sorte	type
type	proposition (évt. paramétrée)
terme	démonstration
réduction sur les types	règle de déduction (conversion)
réduction sur les termes	élimination des coupures

On retrouve le lambda-calcul de Church (1932, 1933) au niveau des types, qui sont des lambda-termes représentant des propositions. La β-réduction au niveau des types est la règle de déduction β de Church.

Au niveau ≪du dessous≫, on a des termes qui représentent les démonstrations de ces propositions.

Termes et types

Mêmes règles que pour système F plus des vérifications de sortes et une règle de conversion du type :

$$\Gamma_1, x : t, \Gamma_2 \vdash x : t$$

$$\frac{\Gamma \vdash t :: * \qquad \Gamma, x : t \vdash M : t'}{\Gamma \vdash \lambda x{:}t.\, M : t \to t'}$$

$$\frac{\Gamma \vdash M : t \to t' \qquad \Gamma \vdash N : t}{\Gamma \vdash M\ N : t'}$$

$$\frac{\Gamma, X :: k \vdash M : t}{\Gamma \vdash \Lambda X :: k.\, M : \forall X :: k.\, t}$$

$$\frac{\Gamma \vdash M : \forall X :: k.\, t \qquad \Gamma \vdash t' :: k}{\Gamma \vdash M[t'] : t\{X \leftarrow t'\}}$$

$$\frac{\Gamma \vdash M : t \qquad t =_\beta t'}{\Gamma \vdash M : t'}\ \text{(conv)}$$

Les formes de paramétrisation

Jusqu'ici nous avons vu trois mécanismes de paramétrisation :

- terme paramétré par un terme : $\lambda x.\, M$
  (= fonction des termes dans les termes)
- terme paramétré par un type : $\Lambda X.\, M$
  (= polymorphisme, système F)
- type paramétré par un type : $\lambda X.\, t$
  (= constructeur de types, système F_ω)

La quatrième forme est très intéressante aussi :

- type paramétré par un terme
  (= types dépendants)

Les types dépendants en logique

Sous l'angle ≪proposition = type≫, les types dépendants sont l'analogue des prédicats.

Par exemple, les prédicats `pair` et `impair` sur les entiers naturels n

$$\texttt{pair}(n) \stackrel{def}{=} n \bmod 2 = 0 \qquad \texttt{impair}(n) \stackrel{def}{=} n \bmod 2 = 1$$

sont des constructeurs de types qui prennent en paramètre un entier n et non pas un autre type (comme en F_ω). Le théorème

> *Si n est pair, alors $n+1$ est impair*

correspond au type

$$\forall n : \mathbb{N}.\ \texttt{pair}(n) \to \texttt{impair}(n+1)$$

Les types dépendants en programmation

En Fortran, C ou C++, le type d'un tableau $t[N]$ contient

- un type t : le type des éléments du tableau
- un ≪terme≫ (expression constante) N : la dimension du tableau.

En d'autres termes, le constructeur de types `array` prend deux paramètres, un type t et un terme N, pour former le type $\texttt{array}(t, N)$.

Si on lève la restriction ≪N est une expression constante≫ et qu'on permet de quantifier sur ces N, on peut donner des types dépendants très précis aux opérations sur les tableaux :

$$\texttt{concat} : \forall t.\forall N_1.\forall N_2.\ \texttt{array}(t, N_1) \to \texttt{array}(t, N_2) \to \texttt{array}(t, N_1 + N_2)$$

($\to$ séminaire de P. E. Dagand en semaine 2.)

Les types dépendants en programmation

Quiz : les types de tableaux ci-dessous sont-ils compatibles ?

$$\begin{aligned}
\texttt{array}(t, 6) \quad &\text{et} \quad \texttt{array}(t, 5+1) \\
\texttt{array}(t, N+M) \quad &\text{et} \quad \texttt{array}(t, M+N) \\
\texttt{array}(t, \texttt{fact}(N)/\texttt{fact}(N-1)) \quad &\text{et} \quad \texttt{array}(t, N) \\
\texttt{array}(\texttt{random}(10)) \quad &\text{et} \quad \texttt{array}(6)
\end{aligned}$$

On voudrait dire que les types $\texttt{array}(t, N)$ et $\texttt{array}(t, N')$ sont convertibles dès lors que les expressions N et N' dénotent le même entier. Mais cela est

- indécidable si le langage des expressions est suffisamment riche ;
- mal défini si le langage des expressions contient des effets.

Types dépendants et conversion

$$\frac{\Gamma \vdash M : t \qquad t \approx t'}{\Gamma \vdash M : t'} \text{ (conv)}$$

La notion de convertibilité $\approx$ n'est plus seulement la β-conversion (comme dans F_ω) mais inclut d'autres équivalences pour traiter les termes qui peuvent apparaître dans t et t'.

F* utilise un démonstrateur automatique pour prouver que t et t' sont égaux modulo des théories (arithmétique, vecteurs de bits, etc).

Coq et Agda utilisent uniquement des règles de calcul, ce qui fait que $\mathtt{array}(t, 5+1) \approx \mathtt{array}(t, 1+5)$ mais $\mathtt{array}(t, N+1) \not\approx \mathtt{array}(t, 1+N)$.

Pour aller plus loin il faut construire et utiliser des fonctions de retypage comme $f : \forall t.\, \forall N.\, \mathtt{array}(t, N+1) \to \mathtt{array}(t, 1+N)$

Automath et les types dépendants

Automath (N. de Bruijn, 1968) : le premier assistant à la démonstration ; naissance de l'idée de vérifier par ordinateur des démonstrations écrites à la main.

On a ≪démonstration = lambda-terme≫ mais pas ≪proposition = type≫ :

Types : $t ::= \texttt{bool}$ type des propositions
$\mid T(b)$ type (dépendant) des démonstrations de $b : \texttt{bool}$
$\mid t_1 \to t_2$

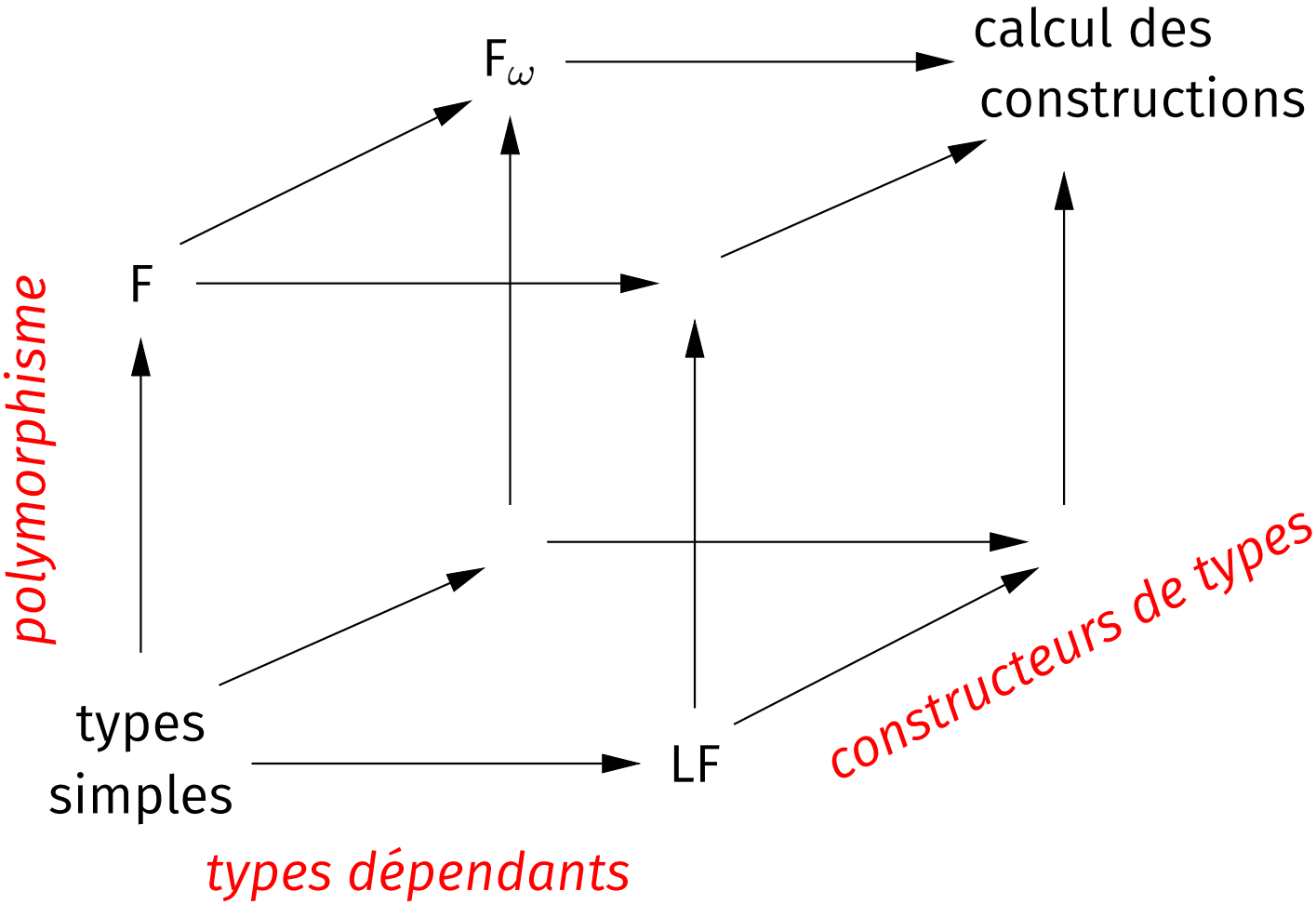
Les connecteurs logiques vivent au niveau des termes et n'ont pas besoin d'être reflétés au niveau des types

$$\wedge : \texttt{bool} \to \texttt{bool} \to \texttt{bool} \qquad \forall : (\texttt{bool} \to \texttt{bool}) \to \texttt{bool}$$

Cette approche est utilisée aussi dans le Edinburgh Logical Framework (LF), un formalisme ≪méta≫ pour définir des logiques.

Le lambda-cube de Barendregt

Trois directions pour étendre le lambda-calcul simplement typé :



III

Les PTS (*Pure Type Systems*)

Vers l'unification

Entre 1971 et 1985 apparaissent des systèmes formels

- directement inspirés par les notes de Howard ;
- utilisables comme logiques intuitionnistes aussi bien que comme lambda-calcul typés ;
- unifiant polymorphisme, constructeurs de types, et types dépendants ;
- plus expressifs que F, F_ω et LF.

Historique

La théorie des types intuitionnistes de Martin-Löf (MLTT) :

- Une logique constructive à base de types dépendants.
- Un petit nombre de constructeurs de types, dont Π (fonctions dépendantes) et Σ (produits dépendants).
- 1971–1979 : plusieurs versions successives.

Le calcul des constructions (CC) :

- Coquand & Huet, 1986–1988 : F_ω + types dépendants.
- Nombreuses extensions ultérieures au cours du développement de Coq.

Les *Pure Type Systems* (PTS) :

- Berardi, 1988 ; Terlouw, 1989 ; Barendregt, 1992 : reformulation unifiée.

Un lambda, un Pi, des termes

Dans F_ω :

- Trois catégories syntaxiques : termes, types, sortes.
- Trois lambdas : $\lambda x : t.\, M$ / $\Lambda X :: k.\, M$ / $\lambda x :: k.\, t$
- Trois types de fonctions : $t_1 \to t_2$ / $\forall X :: k.\, t$ / $k_1 \Rightarrow k_2$.

Dans les PTS :

- Même syntaxe pour les termes et les types.
- Un seul lambda $\lambda x : A.\, B$ pour toutes les formes d'abstraction.
- Un seul type $\Pi x : A.\, B$ (type dépendant de fonction) pour tous les λ.
- Des univers pour distinguer termes, types, sortes, etc.

Pure Type Systems : syntaxe

Univers :		U	$\in \mathcal{U}$	
Termes, types :	A, B, C	$::=$	x	variables
			$\lambda x : A.\, B$	abstractions
		$\mid$	$A\ B$	applications
		$\mid$	$\Pi x : A.\, B$	type de fonction
		$\mid$	U	nom d'univers

Notation : $A \to B \stackrel{def}{=} \Pi x : A.\, B$ si x non libre dans B.

Exemple : traduction de F_ω

Deux univers : $*$ pour les types, $\Box$ pour les sortes.

$$
\begin{array}{lrcll}
\text{Termes :} & \lambda x : t.\, M & \equiv & \lambda x : t.\, M & \text{avec } t : * \\
 & \Lambda X :: k.\, M & \equiv & \lambda X : k.\, M & \text{avec } k : \Box \\
\text{Types :} & t_1 \to t_2 & \equiv & t_1 \to t_2 & \text{avec } t_1, t_2 : * \\
 & \forall X :: k.\, t & \equiv & \Pi X : k.\, t & \text{avec } k : \Box, t : * \\
\text{Sortes :} & * & \equiv & * & \text{avec } * : \Box \\
 & k_1 \Rightarrow k_2 & \equiv & k_1 \to k_2 & \text{avec } k_1, k_2 : \Box
\end{array}
$$

Pure Type Systems : typage

$$\frac{(U, U') \in \mathcal{A}}{\emptyset \vdash U : U'}\ (\text{ax}) \qquad \frac{\Gamma \vdash A : U}{\Gamma, x : A \vdash x : A}\ (\text{var}) \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : U}{\Gamma, x : C \vdash A : B}\ (\text{wk})$$

$$\frac{\Gamma \vdash A : U_1 \quad \Gamma, x : A \vdash B : U_2 \quad (U_1, U_2, U_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A.B : U_3}\ (\text{pi})$$

$$\frac{\Gamma, x : A \vdash B : C \quad \Gamma \vdash \Pi x : A.C : U}{\Gamma \vdash \lambda x.A.\, B : \Pi x.A.\, C}\ (\text{abstr})$$

$$\frac{\Gamma \vdash f : \Pi x : A.\, B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\{x \leftarrow a\}}\ (\text{app})$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : U \quad B =_\beta B'}{\Gamma \vdash A : B'}\ (\text{conv})$$

Contrôler l'expressivité par les univers

Un PTS particulier s'obtient en fixant

- un ensemble $\mathcal{U}$ d'univers ;
- une relation $\mathcal{A} \subseteq \mathcal{U} \times \mathcal{U}$ qui dit quel univers appartient à quel univers ; est de quelle sorte ;
- une relation $\mathcal{R} \subseteq \mathcal{U} \times \mathcal{U} \times \mathcal{U}$ qui dit ce qui peut être paramétré par quoi.

$$\frac{(U, U') \in \mathcal{A}}{\emptyset \vdash U : U'}\ (\text{ax}) \qquad \frac{\Gamma \vdash A : U_1 \quad \Gamma, x : A \vdash B : U_2 \quad (U_1, U_2, U_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A.B : U_3}\ (\text{pi})$$

Le cube de Barendregt

Les systèmes du cube de Barendregt ont $\mathcal{U} = \{*, \Box\}$, avec

- $*$ l'univers des types
- $\Box$ l'univers des sortes (les ≪types de types≫)
- $\mathcal{A} = \{ (*, \Box) \}$ c'est à dire $* : \Box$

Les 3 dimensions du cube sont contrôlées par la relation $\mathcal{R}$ de formation des Π-types :

Trait	*élt. de* $\mathcal{R}$	*fonctions autorisées*
types simples	$(*, *, *)$	des termes dans les termes
polymorphisme	$(\Box, *, *)$	des types dans les termes
constructeurs de types	$(\Box, \Box, \Box)$	des types dans les types
types dépendants	$(*, \Box, \Box)$	des termes dans les types

Jouer sur les univers

Davantage d'univers :

P.ex. une version de CC coupe $*$ en deux univers, `Prop` et `Set`.
`Prop` : l'univers des propositions logiques
`Set` : l'univers des types de données et de fonctions qui calculent.

Une infinité d'univers : $*_0 : *_1 : \cdots : *_n : *_{n+1} : \cdots$

Stratification à la manière de Russell.

Un seul univers : $* : *$ (prononcer ≪Type deux-points Type≫)

Danger ! Paradoxe logique !!

Le paradoxe de Burali-Forti (1897)

Supposons que l'ensemble O de tous les ordinaux existe.

L'ordre $<_{ord}$ entre les ordinaux est un bon ordre sur O.

Donc O ordonné par $<_{ord}$ est un ordinal.

Par ailleurs, O est strictement plus grand que tout ordinal.

Donc $O <_{ord} O$, et contradiction.

Burali-Forti en Coq avec Type : Type

Soit U un univers tel qu'on puisse définir le type suivant (le type des types munis d'un ordre bien fondé) :

```
Record ord : U := mkord {
  carrier: U;
  rel: carrier -> carrier -> Prop;
  rel_wf: well_founded rel
}.
```

L'ordre strict sur ce type des ordinaux :

```
Record emb (A B: ord) : Prop := {
  f: carrier A -> carrier B;
  f_inj: forall x y, rel A x y -> rel B (f x) (f y);
  sup: B;
  f_sup: forall x, rel B (f x) sup
}.
```

Burali-Forti en Coq avec Type : Type

Cet ordre est bien fondé :

```
Lemma wf_emb: well_founded emb.
```

On peut donc former l'ordinal de tous les ordinaux :

```
Definition Omega: ord := mkord ord emb wf_emb.
```

Si cette définition est acceptée par Coq, on a notre paradoxe :

```
Lemma emb_Omega_Omega: emb Omega Omega.

Lemma paradox: False.   (* emb_Omega_Omega contredit wf_emb *)
```

Paradoxe évité (ou presque)

Univers `U`	options de Coq	comportement
$\texttt{Type}_i$	par défaut	`ord` : $\texttt{Type}_{i+1}$ et non `ord` : $\texttt{Type}_i$ `Omega` non définissable
`Type`	`-type-in-type`	on démontre `False`
`Prop`	par défaut	projection `carrier: ord -> Prop` non définissable
`Set`	`-impredicative-set`	projection `carrier: ord -> Set` non définissable

Le paradoxe de Girard (1972)

Système U : une extension de F_ω avec du polymorphisme sur les sortes.

- Trois univers $* : \Box : \triangle$
- Règles de formation : celles de F_ω plus $(\triangle, *, *)$ et $(\triangle, \Box, \Box)$.

Permet d'écrire p.ex.

$$\lambda k : \Box.\ \lambda \alpha : k \to k.\ \lambda \beta : k.\ \alpha\ (\alpha\ \beta) \quad : \quad \Pi k : \Box.\ (k \to k) \to (k \to k)$$

Le paradoxe de Girard : un codage du paradoxe de Burali-Forti dans le système U, plus subtil que le codage précédent avec `Type : Type`.

En corollaire, montre que MLTT 1971 (avec `Type : Type`) est incohérente. Les versions ultérieures de MLTT sont stratifiées avec des univers.

Voir aussi `https://github.com/coq-contribs/paradoxes`

Prédicativité et imprédicativité

Le ≪principe du cercle vicieux≫ de Russell :

> *Whatever involves all of a collection must not be one of the collection.*

Prédicativité : $\Pi X : U.\, A$ est dans un univers ≪au dessus≫ de U.

Autrement dit : pas de ≪cercle vicieux≫ au sens de Russell.

Exemple : $(\Pi X : \mathtt{Type}_n.\, A) : \mathtt{Type}_{n+1}$ en MLTT, Coq, Agda.

Imprédicativité : $\Pi X : U.\, A$ est dans l'univers U.

Autrement dit : on peut quantifier ≪sur soi-même≫.

Exemple : $(\forall X{:}*\,.\,t) : *$ dans les systèmes F et F_ω

Exemple : `(forall (P: Prop), Q) : Prop` en Coq.

Cumulativité et polymorphisme d'univers

Comment rendre une définition réutilisable dans plusieurs univers ?

Cumulativité : une forme de sous-typage entre univers.

$$\frac{\Gamma \vdash A : \mathtt{Type}_i}{\Gamma \vdash A : \mathtt{Type}_{i+1}}$$

Polymorphisme : ajouter des variables d'univers i et une forme de quantification universelle dessus. P.ex. en Coq récent :

```
Polymorphic Definition idtype@{i} : Type@{i+1} :=
    forall A: Type@{i}, A -> A.
Polymorphic Definition identity@{i} : idtype@{i} :=
    fun (A: Type@{i}) (x: A) => x.
```

Expressions symboliques d'univers : $u ::= U \mid i \mid u + 1 \mid \max(u_1, u_2)$

Application : Coq et Agda

	Coq	Agda
Univers	$\texttt{Prop}$ $\texttt{Set} = \texttt{Type}_0$ $\texttt{Type}_1, \ldots, \texttt{Type}_n, \ldots$	$\texttt{Set} = \texttt{Set}_0$ $\texttt{Set}_1, \ldots, \texttt{Set}_n, \ldots$
Imprédicatif ?	$\texttt{Prop}$ (et, autrefois, $\texttt{Set}$)	non
$\mathcal{R}$	$(\texttt{Type}_i, \texttt{Type}_j, \texttt{Type}_{\max(i,j)})$ $(U, \texttt{Prop}, \texttt{Prop})$	$(\texttt{Set}_i, \texttt{Set}_j, \texttt{Set}_{\max(i,j)})$
Cumulativité ?	oui	non
Polymorphisme ?	oui (récemment)	oui (récemment)

IV

Point d’étape

Curry-Howard, la suite

Vu aujourd'hui :

- Exploration de nombreuses extensions :
  polymorphisme, constructeurs de types, types dépendants,
  `Type : Type`, univers, …
- Convergence vers les *Pure Type Systems*, une famille de lambda-calculs typés utilisables aussi bien comme langages de programmation fonctionnelle que comme logiques intuitionnistes
- …mais centrés sur les fonctions et leurs Π-types, c.à.d. les connecteurs logiques $\Rightarrow$ et $\forall$.

La semaine prochaine :

- Autres types de données ? Autres connecteurs logiques ?
- Mécanismes généraux pour définir les types de données et les connecteurs logiques : les types et prédicats inductifs.

V

Bibliographie

Bibliographie

- Jean-Yves Girard. *Le point aveugle. Cours de logique. Tome 1 : vers la perfection*. Hermann, 2006. Chapitre 6.
- Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002. Chapitres 22, 23, 24, 29, 30.
- Morten Heine Sørensen, Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. 1998. Chapitres 11 à 14.

  `https://disi.unitn.it/~bernardi/RSISE11/Papers/curry-howard.pdf`.