TIPE ENS :
Vérifier et formaliser les mathématiques avec la théorie des types

STERBAC Raphaël

12 Juin 2024

Introduction

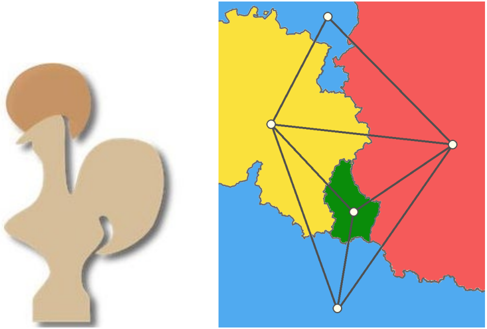


Figure: Coq, Théorème des quatres couleurs

Définitions

Ensemble des expressions du calcul des constructions :

$$E = V|\Box| * |(EE)|(\lambda V : EE)|(\Pi V : EE)|C(\bar{E})$$

Définitions :

- Descriptive : $\bar{x} : \bar{A} \triangleright a(\bar{x}) := M : N$
- Primitive (axiome) : $\bar{x} : \bar{A} \triangleright a(\bar{x}) :=\perp : N$

Contexte : couple $(\Delta; \Gamma)$ où Δ est un ensemble de définitions et Γ est un ensemble de déclarations de types/variables.

β-réduction et α-équivalence

α-équivalence :

$$\lambda x.M =_\alpha \lambda y.M^{x \to y}$$

$\to$ Équivalence des termes par renomage des variables liées. $\to$ On raisonnera sur les expressions à α-équivalence près.

β-réduction :

$$(\lambda x.M)N \to_\beta M[x := N]$$

$\to$ Substitution de la variable liée lors d'une application.

δ-réduction

Si $\bar{x} : \bar{A} \rhd a(\bar{x}) := M : N$ est un élément de l'environnement Δ,

$$a(\bar{U}) \rightarrow^{\Delta} M[\bar{x} := \bar{U}]$$

$\beta\delta$-équivalence : notée $=^{\Delta}_{\beta}$, définie comme la clôture réflexive, symétrique, transitive de $(\rightarrow_{\beta}) \cup (\rightarrow_{\delta})$

$\rightarrow$ Substitution des applications et des constantes.

Règles de dérivation

$\rightarrow$ Déterminer les jugements dérivables (expressions bien typées)

$$\text{(abst)}\ \frac{\Delta;\Gamma, x : A \vdash M : B \qquad \Delta;\Gamma \vdash \Pi x : A.B : s}{\Delta\Gamma \vdash \lambda(x : A).M : \Pi x : A.B}$$

β-Normalisation

Forme β-normale : écriture sans expressions réductibles (redex)

La β-réduction est fortement normalisante (FN) : il n'existe pas de chemin infini de β-réductions depuis une expression donnée.

$\rightarrow$ Existence de la forme normale, terminaison des calculs.

Le théorème de Church-Rosser

Soit $M \in E$ une expression telle que $M \rightarrow_{\beta}^{*} N_1$ et $M \rightarrow_{\beta}^{*} N_2$. Alors il existe $N_3 \in E$ telle que $N_1 \rightarrow_{\beta}^{*} N_3$ et $N_2 \rightarrow_{\beta}^{*} N_3$.

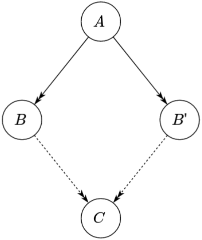


Figure: Church-Rosser (CR)

$\rightarrow$ Unicité de la forme normale, permutation de l'ordre des calculs

Une simplification

Soit $M \in E$ une expression telle que $M \to_\beta N_1$ et $M \to_\beta N_2$. Alors il existe $N_3 \in E$ telle que $N_1 \to_\beta^* N_3$ et $N_2 \to_\beta^* N_3$.

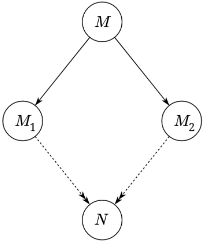


Figure: Church-Rosser faible (WCR)

$\mathrm{WCR} \wedge \mathrm{FN} \implies \mathrm{CR}$

Preuve : on montre l'unicité de la forme β-normale par l'absurde.

Soit $M \in E$ avec M_1, M_2 deux formes normales de M, on construit alors une suite infinie de réductions en utilisant WCR :

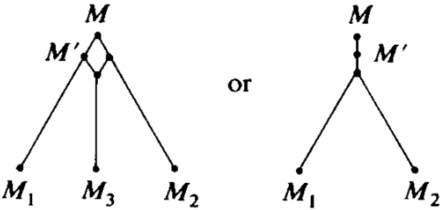


Figure: Construction d'une suite infinie de réductions

Généralisation au calcul des constructions

On se place désormais dans un contexte $(\Delta;\Gamma)$ légal, c'est à dire qu'il existe M, N telles que $\Delta;\Gamma \vdash M : N$

Dans un tel contexte, toute expression dérivable possède une unique forme $\beta\delta$-normale, obtenue par réductions successives.

Intérêt du système formel

$\rightarrow$ Formalisation constructive des mathématiques.

$\rightarrow$ Correspondance avec la logique propositionnelle intuitionniste.

Paradigme Proof as Types

On associe à tout jugement dérivable $M : N$ une proposition, et une preuve de cette proposition est alors l'instanciation d'un élément $x : M$.

On peut alors coder l'implication par $A \longrightarrow B := \Pi x : A.B$

Isomorphisme de Curry-Howard

Logique	**Théorie des types**
$A \implies B$	$\Pi x : A.B$
$\perp A$	$\Pi\alpha : *.\alpha$
$\neg A$	$A \longrightarrow \perp$
$A \wedge B$	$\Pi C : *.(A \to B \to C) \to C$
$A \vee B$	$\Pi C : *.(A \to C) \to (B \to C) \to C$
$\forall_{x \in S} P(x)$	$\Pi x : S.Px$
$\exists_{x \in S} P(x)$	$\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha)$

Table: Correspondance de Curry-Howard

Problème

Notre système ne permet pas de dériver le tiers exclu (ET) et la double négation (DN).

$\rightarrow$ On pose ET comme axiome, et on montre que l'on peut dériver DN.
En effet, si on admet qu'il existe $i_{ET} : \Pi\alpha : *.\alpha \wedge \neg\alpha$, on a :

$$(\Pi\beta : *.(\lambda x.i_{ET}\beta\beta(\lambda y : \beta.y)(\lambda z : \neg\beta.xz\beta) : \beta)) : \Pi\beta : *.\neg\neg\beta \rightarrow \beta$$

Conséquences

$\rightarrow$ On obtient alors les règles de dérivations du calcul des prédicats.

$\rightarrow$ Prouver une proposition revient à trouver un terme d'un type donné.

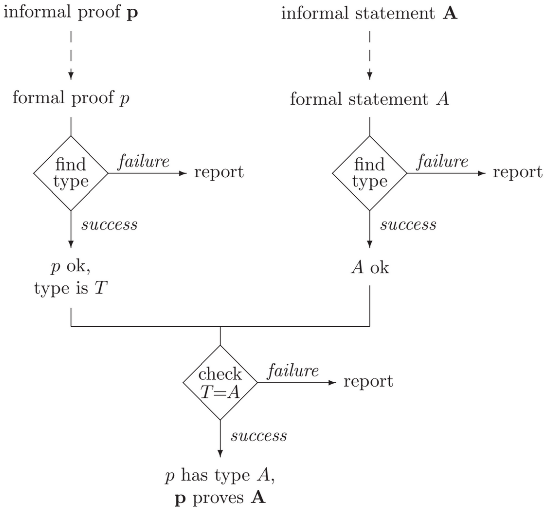
$\rightarrow$ Le calcul des constructions est cohérent (au sens logique) : on ne peut pas tout y démontrer.

Vérification d'une preuve

$\rightarrow$ On se donne un terme p dans un contexte donné $(\Gamma; \Delta)$, correspondant à une preuve d'une certaine proposition, représentée par un type A.

$\rightarrow$ Vérifier la preuve revient à vérifier que $p : A$, c'est donc la résolution du problème de Type-Assignment qui nous intéresse.

Algorithme



informal proof **p**
informal statement **A**
formal proof p
formal statement A
find type
failure
report
find type
failure
report
success
success
p ok, type is T
A ok
check $T=A$
failure
report
success
p has type A, **p** proves **A**

Indices de De Bruijn

On représente informatiquement les variables et les abstractions avec des indices de de Brujin.

$\rightarrow$ Ainsi, on ne nomme pas les variables liées.
$\rightarrow$ La α-équivalence est alors ramenée à un test syntaxique.

Exemple :

$$\lambda x.\lambda y.x \equiv \lambda\lambda 2$$

Fonction `type`

- Variable : `type` $(x) \to A$ avec $(x : A) \in \Gamma$
- Sorte : `type` $(*) \to \square$
- Application : `type` $(AB) \to N[x := B]$
  avec `type` $(A) = \Pi x : M.N$
- Abstraction : `type` $(\lambda x : A.B) \to \Pi x : A.($`type B`$)$
- Type produit : `type` $(\Pi x : A.B) \to$ `type`
- Constante : `type` $(a(\bar{U})) \to N$
  avec $a(\bar{x}) := M : N$ ou $a(\bar{x}) :=\perp : N \in \Delta$
- Sinon $\to$ Le terme n'est pas typable.

Comparaison

On montre ensuite la $\beta\delta$-équivalence des deux types.

$\rightarrow$ On reduit les deux termes successivements jusqu'à trouver leur forme normale, que l'on compare.

Implémentation : définitions

```
type sorte = Type | Kind

type terme =
    Var of int (* Indices de De Brujin *)
  | Sorte of sorte
  | Application of (terme * terme)
  | Abstraction of (terme * terme)
  | Produit of (terme * terme)
  | Constant of int

type definition = {
  definiens : terme option; (* None si la definition est un axiome *)
  ty : terme;
}

type contexte = {
  defs : definition array; (* Tableau des définitions *)
  vars : terme list (* Liste des types des variables libres *)
}
```

Implémentation : substitution

```
let rec subst_aux s = function
    | Var n -> s n
    | Application (a, b) -> Application (subst_aux s a, subst_aux s b)
    | Abstraction (a, b) -> Abstraction (subst_aux s a, subst_aux (function
      | 1 -> Var 1
      | n -> incr (s (n - 1))
      ) b)
    | Produit (a, b) -> Produit (subst_aux s a, subst_aux (function
      | 1 -> Var 1
      | n -> incr (s (n - 1))
      ) b)
    | t -> t
and incr t = subst_aux (fun k -> Var (k + 1)) t;;

(* subst n a b = a[n:=b] *)
let subst n a b = subst_aux (function
    | k when k = n -> b
    | k -> Var k) a
;;
```

Implémentation : β-réduction

```
let rec reduit = function
    | Abstraction (a, b) -> Abstraction (reduit a, reduit b)
    | Produit (a, b) -> Produit (reduit a, reduit b)
    | Application (a, b) -> ( match reduit a with
      | Abstraction (_, t) -> subst 1 t (reduit b) (* 1 est l'indice de De Bruj
      | Produit (_, t) -> subst 1 t (reduit b)
      | t -> Application (reduit t, reduit b)
    )
    | t -> t
;;
```

Implémentation : type_assignment

```
let rec type_assignment contexte t = ( match reduit t with
    | Var k ->
      let rec incr_by_n n t = match n with
      | 0 -> t
      | n -> incr_by_n (n - 1) (incr t)
      in
      let ty = List.nth contexte.vars (k - 1) in (* le type de t *)
      incr_by_n k ty (* On modifie le type pour faire correspondre au contexte *
    | Sorte Type -> Sorte Kind
    | Sorte Kind -> failwith "Le terme n'est pas typable : un Kind n'as pas de
    | Application (a, b) -> (
        match type_assignment contexte a with
      | Produit (x, y) -> Application (Produit (x, y), b)
      | t -> failwith "Le terme n'est pas typable : application non légale"
    )
    | Abstraction (a, b) -> Produit (a, type_assignment (push_var a contexte) b
    | Produit (a, b) -> type_assignment (push_var a contexte) b
    | Constant k -> contexte.defs.(k).ty
)
```