TIPE ENS :
Formaliser et vérifier les mathématiques avec la théorie des types

STERBAC Raphaël

2024

1 Introduction

Dans notre société actuelle, l'informatique est omniprésente et son bon fonctionnement est essentiel. Certains dispositifs utilisant des algorithmes nécessitent donc d'être vérifiés formellement pour garantir leur bon fonctionnement, et prévenir des catastrophes (ex : explosion d'Ariane 5). De même, les mathématiques nécessitent parfois l'écriture de preuves très longues dont la vérification et la correction sont des tâches complexes. Ainsi, cela justifie la création d'assistants de preuves, et donc le choix de mon sujet. On peut par exemple citer le théorème des quatre couleurs qui a été démontré avec l'assistant de preuve Coq, employant le calcul inductif des constructions comme système formel. Ici, je me contenterai de traiter le calcul des constructions classique, et de répondre à la problématique suivante : Comment formaliser et vérifier les mathématiques avec la théorie des types ?
Pour cela, je me fonderai sur les ouvrages [1] et [2]

2 Définitions

On se donne un ensemble V de variables, ainsi qu'un ensemble C de constantes, tels que $V \cap C = \emptyset$. On pose $s = \{*, \Box\}$ l'ensemble des sortes, supposé disjoint avec V et C.
On définit alors inductivement l'ensemble des expressions du calcul des constructions :

Définition 2.1.

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

On définit le concept de définition de la manière suivante :

Définition 2.2. Soit $a \in C$, $\bar{x} = (x_1, ..., x_n) \in V^n$, $\bar{A} = (A_1, ..., A_n) \in E^n$ et $M, N \in E^2$
Une définition descriptive est de la forme : $\bar{x} : \bar{A} \triangleright a(\bar{x}) := M : N$
Une définition primitive (ou axiome) est de la forme : $\bar{x} : \bar{A} \triangleright a(\bar{x}) :=\perp: N$

On définit alors la notion de contexte :

Définition 2.3. Un contexte est la donnée de $(\Delta; \Gamma)$ où Δ est un environnement (i.e une liste de définitions) et Γ une liste de déclarations.

On définit également l'α-équivalence et la β-réduction :

Définition 2.4 (α-équivalence)**.** Deux lambda termes sont dits α-équivalents si ils sont identiques par renommage des variables liées : $\lambda x.M =_\alpha \lambda y.M^{x \to y}$
On ne considérera désormais les lambda-termes qu'à α-équivalence près.

Définition 2.5 (β-équivalence)**.** On dit qu'une expression se β-réduit lorsque l'on applique une expression à une abstraction en substituant les occurrences de la variable liée : $(\lambda x.M)N \to_\beta M[x := N]$
On note $\to_\beta^*$ la clôture réflexive et transitive de $\to_\beta$.
Alors deux expressions M, N sont dites β-équivalentes (noté $M =_\beta N$) lorsqu'il existe un chemin du type :

$$M \equiv N_0 \leftrightarrow_\beta N_1 ... \leftrightarrow_\beta N_f \equiv N$$

Ainsi définie, $=_\beta$ est la fermeture réflexive, symétrique et transitive de $\to_\beta$.

De façon analogue, on définit la δ-réduction, qui correspond à la β-réduction pour les définitions :

Définition 2.6 (δ-équivalence). Si $\bar{x} : \bar{A} \rhd a(\bar{x}) := M : N$ est un élément de l'environnement Δ, alors :

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

Et on définit de même la δ-équivalence comme la clôture réflexive, symétrique et transitive de $\to^{\Delta}$, notée $=^{\Delta}$

Enfin, on peut définir la $\beta\delta$-équivalence.

Définition 2.7 ($\beta\delta$-équivalence). On définit la relation de $\beta\delta$-équivalence, notée $=_\beta^\Delta$ comme la clôture réflexive, symétrique, transitive de $(\to_\beta) \cup (\to_\delta)$

Informellement, cette relation dénote le fait qu'on peut passer d'une expression A à une autre expression B par substitution dans les différentes applications, et par substitution par des constantes.

Avec toutes ces définitions, nous donnons désormais les règles de dérivation de notre système formel, noté λD. Ces règles permettent de déterminer les jugements dérivables, et donc les expressions bien typées dans un contexte donné.

$$\text{(sort)} \quad \frac{}{\emptyset;\emptyset \vdash * : \Box}$$

$$\text{Si } x \notin \Delta : \text{(var)} \quad \frac{\Delta;\Gamma \vdash A : s}{\Delta;\Gamma, x : A \vdash x : A}$$

$$\text{(weak)} \quad \frac{\Delta;\Gamma \vdash A : B \qquad \Delta;\Gamma \vdash C : s}{\Delta\Gamma, x : C \vdash A : B}$$

$$\text{(form)} \quad \frac{\Delta;\Gamma \vdash A : s_1 \qquad \Delta;\Gamma, x : A \vdash B : s_2}{\Delta\Gamma \vdash \Pi x : A.B : s_2}$$

$$\text{(appl)} \quad \frac{\Delta;\Gamma \vdash M : \Pi x : A.B \qquad \Delta;\Gamma \vdash N : A}{\Delta\Gamma \vdash MN : B[x := N]}$$

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

$$\text{Si } B =_\beta^\Delta B' : \text{(conv)} \quad \frac{\Delta;\Gamma \vdash A : B \qquad \Delta;\Gamma \vdash B' : s}{\Delta;\Gamma' \vdash A : B'}$$

$$\text{Si } a \notin \Delta : \text{(def)} \quad \frac{\Delta;\Gamma \vdash K : L \qquad \Delta;\bar{x} : \bar{A} \vdash M : N}{\Delta, \bar{x} : \bar{A} \rhd a(\bar{x}) := M : N;\Gamma \vdash K : L}$$

$$\text{Si } a \notin \Delta : \text{(def-prim)} \quad \frac{\Delta;\Gamma \vdash K : L \qquad \Delta;\bar{x} : \bar{A} \vdash N : s}{\Delta, \bar{x} : \bar{A} \rhd a(\bar{x}) :=\bot: N;\Gamma \vdash K : L}$$

$$\text{Si } \bar{x} : \bar{A} \rhd a(\bar{x}) := M : N \in \Delta : \text{(inst)} \quad \frac{\Delta;\Gamma \vdash * : \Box \qquad \Delta;\Gamma \vdash \bar{U} : \overline{A[\bar{x} := \bar{u}]}}{\Delta;\Gamma \vdash a(\bar{U}) : N[\bar{x} := \bar{U}]}$$

$$\text{Si } \bar{x} : \bar{A} \rhd a(\bar{x}) :=\bot: N \in \Delta : \text{(inst-prim)} \quad \frac{\Delta;\Gamma \vdash * : \Box \qquad \Delta;\Gamma \vdash \bar{U} : \overline{A[\bar{x} := \bar{u}]}}{\Delta;\Gamma \vdash a(\bar{U}) : N[\bar{x} := \bar{U}]}$$

3 Quelques théorèmes

On se place dans un premier temps dans le système formel du lambda-calcul simplement typé, noté $\lambda \to$.
On commence par quelques définitions :

Définition 3.1 (Forme β-normale)**.** On dit que M est écrite sous forme β-normale si M ne contient pas d'expression réductible (abrégé redex).
Alors, M est dite β-normalisante si elle est β-équivalente à une expression sous forme β-normale.

3.1 Forte normalisation

Définition 3.2 (Propriété de forte normalisation)**.** On dit qu'une expression M est fortement normalisante si il n'existe pas de chemin infini de β-réduction depuis M.

Théorème 3.1.1. *La β-réduction vérifie la propriété de forte normalisation.*

Ce théorème est fondamental en ceci qu'il assure l(abst) $\dfrac{\Delta;\Gamma,x:A\vdash M:B \qquad \Delta;\Gamma\vdash \Pi x:A.B:s}{\Delta\Gamma\lambda(x:A).M:\Pi x:A.B}$

a terminaison des calculs effectués.

3.2 Church-Rosser

On peut démontrer le théorème de Church-Rosser, qui s'énonce ainsi :

Théorème 3.2.1 (CR)**.** *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 $N_2\to_\beta^* N_3$.*

Ce théorème dénote l'invariance par permutation de l'ordre des calculs de la β-réduction : on peut réduire les termes dans n'importe quel ordre. On obtient aussi l'unicité de la forme β-normale d'une expression. Ici, le système de réduction étant fortement normalisant, on peut se contenter de démontrer la version locale suivante :

Lemme 3.2.2. *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 $N_2\to_\beta^* N_3$.*
Alors si cette propriété est vérifiée, on obtient le théorème de Church Rosser.

Démonstration. La propriété locale se démontre par induction sur la structure des lambda termes.

Par forte normalisation, toute expression M possède une forme β-normale. Supposons que M se réduise en deux β-nf M_1 et M_2. Alors, en utilisant la version locale du théorème de Church-Rosser, on peut trouver M' telle que $M\to_\beta M'$ et M' possède lui-même deux β-nf distinctes. On distingue deux cas :

- Si les deux chemins se réduisent vers M' à l'étape 1, alors ce M' convient.
- Sinon, on applique le théorème local de Church-Rosser aux deux réductions distinctes de l'étape 1 notée M_1' et M_2', ce qui nous donne M_3 une autre forme β-nf de M, alors on choisit M_1' ou M_2' qui convient.

Ainsi, en itérant, on pourrait construire un chemin infini qui contredirait la propriété de forte normalisation. On obtient donc l'unicité de la forme normale, ce qui correspond au théorème de Church Rosser fort.

On illustre ces propriétés dans la figure 1.

3.3 Généralisation

On admet alors la généralisation suivante dans λD, à la $\beta\delta$-réduction du calcul des constructions :

Définition 3.3. Une expression M est dite légale si il existe un environnement Δ et un contexte Γ, ainsi qu'une expression N tel que $\Delta;\Gamma\vdash M:N$
L'association $(\Delta;\Gamma)$ est dite légale si il existe M,N tel que $\Delta;\Gamma\vdash M:N$

Théorème 3.3.1 (FN dans λD)**.** *Tout $L\in E$ légale ne possède pas de suite de $\beta\delta$-réductions infinies partant de L.*

Dans toute la suite, on se placera dans un contexte $(\Delta;\Gamma)$ légal, ce qui entraine donc que FN est vraie pour toute expression dérivable de $(\Delta;\Gamma)$.

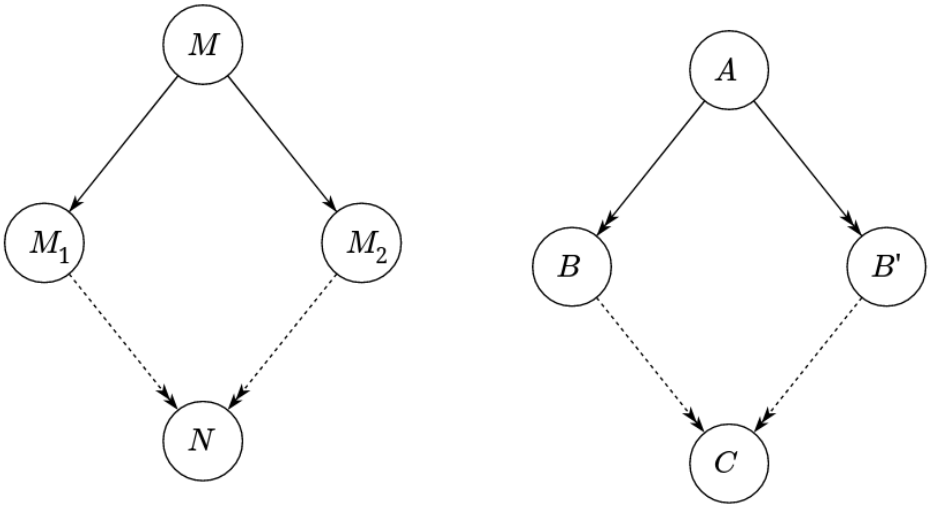

FIGURE 1 – Church-Rosser (Faible à gauche, Fort à droite)

4 Isomorphisme de Curry-Howard (PAT)

On cherche dans cette section à faire correspondre le calcul des constructions avec la logique intuitionniste. On prend ici $\Gamma = \emptyset$ et l'environnement Δ sans axiome, et on suppose que l'on dispose d'une preuve de $\Delta; \Gamma \vdash * : \Box$. Alors, $(\Delta; \Gamma)$ est légal, et FN est vérifiée.
On peut alors associer à un jugement dérivable $\Delta; \emptyset \vdash M : N$ une proposition logique, une preuve de celle-ci correspond alors à l'instantiation d'un terme $x : M$ du type M.

Définition 4.1. On notera $A \longrightarrow B := \Pi x : A.B$ l'implication logique.

Preuve. En effet, si l'on a une preuve de A notée $x : A$, on a $fx : B$ une preuve de B.

Définition 4.2. On note $\bot := \Pi\alpha : *.\alpha$ l'antilogie (absurde)

Preuve. Si $f : \bot$, alors on peut appliquer A la fonction f, pour $A : *$ une proposition, et alors $fA : A$ est une preuve de A. On peut donc en déduire toutes les propositions (antilogie)

Définition 4.3. On peut alors noter $\neg A := A \longrightarrow \bot$

Définition 4.4. $A \wedge B := \Pi C : *.(A \to B \to C) \to C$

Définition 4.5. $A \vee B := \Pi C : *.(A \to C) \to (B \to C) \to C$

Définition 4.6. $\forall_{x \in S} P(x) := \Pi x : S.Px$

Définition 4.7. $\exists_{x \in S} P(x) := \Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha)$

Alors, ces définitions permettent de travailler avec les propositions de la logique intuitionniste dans le calcul des constructions, et de les prouver en trouvant des termes associés au bon type.
Néanmoins, cette logique ne correspond pas à la logique classique, et ne permet donc pas de dériver le tiers-exclu ($A \vee \neg A$ est toujours vérifié) et la double négation ($\neg\neg A = A$). On peut ajouter un axiome à notre environnement Δ : $tiers\_exclu := \bot$: $\Pi\alpha : *.\alpha \vee \neg\alpha$. De cette manière, on peut formaliser la logique classique avec le calcul des constructions, de manière analogue à ce que l'on fait en déduction naturelle dans le cadre du programme de classe préparatoire (on dispose de règles de dérivation similaires).

"L'isomorphisme" ainsi défini est résumé dans le tableau suivant :

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

TABLE 1 – Correspondance de Curry-Howard

5 Algorithme

L'intérêt de la théorie des types, et donc en particulier du système formel que nous avons définit (le calcul des constructions) réside en son lien avec la notion d'assistant de preuve. En effet, avec l'isomorphisme précédemment établi, prouver un théorème revient à instancier un élément d'un certain type (correspondant au théorème). Ainsi, un assistant de preuve fondé sur la théorie des types vérifie pour un terme p donné dans un contexte $(\Delta; \Gamma)$ si le type de p est bien le bon selon les règles de dérivation. La vérification d'une preuve se ramène donc au problème de type-checking. On résume cela dans la figure 2, issue de [2].

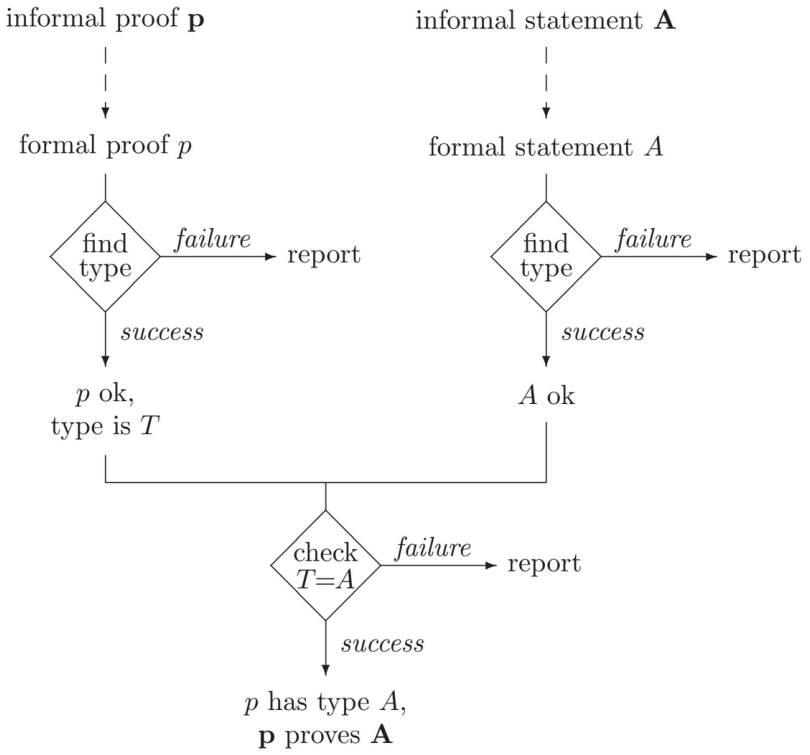

FIGURE 2 – Algorithme de vérification de correction.

Il faut donc élaborer un algorithme permettant de déterminer si p est typable, et son type si c'est bien le cas : c'est le problème de *Type Assignment*. De même, il faut établir un deuxième algorithme permettant de vérifier que deux types sont les mêmes : qu'ils sont $\beta\delta$-équivalents.

5.1 Le problème de Type Assignment

On déduit le type d'un terme dans un contexte donné de sa structure, en filtrant sur la forme des termes imbriqués et en réduisant les termes.
Pour implémenter le calcul des construction en OCaml, j'ai utilisé des indices de De Brujin afin de représenter les variables. Ce choix d'implémentation est justifié par la facilité de manipulation de ces indices : ils permettent de se passer d'α-réductions et de se ramener à des tests d'égalité syntaxique. D'autres conventions telles que celle de Barendregt auraient pu être utilisées.

Définition 5.1 (indices de De Brujin)**.** Un indice de De Brujin est un entier naturel représentant une variable liée. Pour une variable x, l'indice de x correspond aux nombre de liaisons faites dans la portée de la variable x. Ainsi, l'indice est le plus faible pour une portée minimale.

Exemple : le lambda terme $\lambda z.(\lambda y.y(\lambda x.x))(\lambda x.zx)$ s'écrit : $\lambda(\lambda 1(\lambda 1))(\lambda 21)$

Ainsi, cette convention permettra de manipuler informatiquement les lambda termes. On donne alors le filtrage par motif suivant, permettant de définir la fonction type_assignment :

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

L'implémentation OCaml est détaillée en annexe.

5.2 Vérification

Pour montrer la $\beta\delta$-équivalence des deux types, on se contente de montrer qu'ils ont la même forme normale, en appliquant des réductions successives. Cet algorithme se termine d'après le théorème 3.3.1 (forte normalisation) et est correct d'après 3.2.1 (Church-Rosser).

De même, une implémentation OCaml est proposée en annexe.

Références

[1] Barendregt. *The lambda calculus : its syntax and its semantics*.

[2] Nederpelt et Geuvers. *Type Theory and Formal Proof : An Introduction*. Cambridge Press. DOI : `https://doi.org/10.1017/CBO9781139567725`.

Annexe : Implémentation OCaml

```
type sorte = Type | Kind

type terme =
    Var of int (* Indices de De Brujin, on commence à 1 et on augmente *)
  | 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 *)
}

let push_var t contexte = {contexte with vars = t :: contexte.vars};;

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
;;

(* Beta réduit le terme en garantissant le bon typage *)
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 Brujin de la variable liante *)
      | Produit (_, t) -> subst 1 t (reduit b)
      | t -> Application (reduit t, reduit b)
    )
    | t -> t
;;

let rec show_term = function
| Var n -> string_of_int n
| Sorte Type -> "*"
```

```
| Sorte Kind -> "¤"
| Application (a, b) -> "(" ^ show_term a ^ " " ^ show_term b ^ ")"
| Abstraction (a, b) -> "( λ " ^ show_term a ^ " . " ^ show_term b ^ ")"
| Produit (a, b) -> "( Π " ^ show_term a ^ " . " ^ show_term b ^ ")"
| Constant n -> "a(" ^ string_of_int n ^ ")"

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 type"
    | 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
)


let verif_type t a =
    print_string "\n Réduction du type de p : ";
    print_string (show_term (reduit t));
    print_string "\n Réduction du type de a :";
    print_string (show_term (reduit a));
    if (reduit t) = (reduit a) then
        print_string "\n Réussite : p est bien une preuve de A !"
    else
        failwith "p n'est pas une preuve de A"
;;

let is_proof contexte p a =
    print_string " Terme en entrée  : ";
    print_string (show_term p);
    let ty = type_assignment contexte p in
    verif_type ty a
;;

(* Exemple d'utilisation : *)
is_proof {defs=[||]; vars = []}  (Abstraction(Produit(Sorte Type, Sorte Type), Abstraction(Var 1, Application

(* Sortie du programme :

 Terme en entrée  : ( λ ( Π * . * ) . ( λ 1 . (2 1)))
 Réduction du type de p : ( Π ( Π * . * ) . ( Π 1 . * ))
 Réduction du type de a :( Π ( Π * . * ) . ( Π 1 . * ))
 Réussite : p est bien une preuve de A !

*)
```