TIPE ENS

Jean

2021-2022

1 Introduction

L'outil informatique est aujourd'hui massivement déployé dans l'industrie, y compris dans des systèmes que l'on nomme « systèmes critiques ». Ce sont des systèmes dont un dysfonctionnement pourrait entraîner de graves risques pour les biens ou les personnes : un crash d'avion, un accident de train ou un dysfonctionnement dans une centrale nucléaire, par exemple. Pour remédier à cela, le code dans ces systèmes est vérifié formellement : on élabore, à l'aide d'un assistant de démonstration, une preuve du bon fonctionnement du programme, preuve que l'assistant vérifie ensuite. Dans ce cadre, on peut par exemple citer CompCert, un compilateur du langage C doté d'une preuve de fonctionnement : un exécutable compilé par CompCert fonctionne comme le code C le décrit, selon la spécification de C. De plus, les assistants de preuve sont également utiles en mathématiques : le théorème des 4 couleurs sur les graphes planaires a été démontré en partie via l'assistant Coq. Une des méthodes pour formaliser un programme consiste à le représenter dans un système formel, et à prouver des théorèmes dessus. On présentera ici le Calcul des Constructions, une théorie des types dépendants utilisées à l'origine dans Coq.

Problématique : Comment utiliser le Calcul des Constructions pour représenter des mathématiques et les vérifier ?

On présentera en 3 et en 4 comment on peut relier les mathématiques et le Calcul des Constructions, et en 5 un algorithme pour vérifier si un terme du Calcul des Constructions est bien typé.

2 Définitions

On suppose l'existence de deux ensembles $\mathcal{V}$ et $\mathcal{C}$, infinis et disjoints, dont les éléments sont respectivement appelés variables et constantes.

Les termes du calcul des constructions sont données par la syntaxe BNF suivante :

$$
\begin{array}{rll}
<term> & ::= <var> & \text{Variable} \\
& \mid * & \text{Univers des types} \\
& \mid \square & \text{Univers des sortes} \\
& \mid \lambda(<var> : <term>).<term> & \lambda\text{-abstraction} \\
& \mid \prod_{<var>:<term>} <term> & \Pi\text{-abstraction} \\
& \mid <term><term> & \text{Application} \\
& \mid [<const>] & \text{Constante}
\end{array}
$$

où $<var>$ et $<const>$ représentent respectivement des variables et des constantes. On supposera l'application associative à droite, et que Π et λ englobent tout ce qu'il y a à gauche, s'il n'y a pas de parenthèses.

On pose $\mathrm{s} = \{*, \square\}$.

Un contexte est un couple (Γ, Δ), où Γ est une liste d'éléments de la forme $(<var> : <term>)$ et Δ une liste de définitions de la forme $<const> \coloneqq <term> : <term>$ et d'axiomes de la forme $<const> \coloneqq @ : <term>$. On notera $\Gamma = \emptyset$ ou $\Delta = \emptyset$ si Γ ou Δ respectivement est vide.

On définit l'α-équivalence, la β-réduction et la β-équivalence (au sens du λ-calcul usuel) et la δ-réduction au sens de 1 (Definition 9.7.2 pour cette dernière, voir aussi en annexe 7). On notera $A \overset{\Delta}{=}_\beta B$ si dans le contexte Δ, on peut passer de A à B en une suite finie de substitution, dans les sous-termes de A, de $(\lambda(x : A).B)C$ par $B[x \coloneqq C]$ ou réciproquement, et de substitutions d'un terme M par une constante $[C]$ avec $(C \coloneqq M : N) \in \Delta$, ou réciproquement, et on ne considèrera plus les termes qu'à α-équivalence près.

Enfin, sur l'ensemble des jugements $\Delta; \Gamma \vdash A : B$ où A et B sont des termes, on définit l'ensemble des jugements bien formés, par les relations suivantes, exprimées dans le calcul des séquents. Si le jugement $\Delta; \Gamma \vdash A : B$ est dérivable, on dit que A est de type B dans le contexte $(\Gamma; \Delta)$. Les règles de dérivations qui suivent sont adaptées de 1, Appendix D.

Contexte vide (Empty) : $$\frac{}{\emptyset;\emptyset \vdash * : \square}$$

Conversion (Conv) : $$\frac{\Delta;\Gamma \vdash A : B \qquad \Delta;\Gamma \vdash B' : u}{\Delta;\Gamma \vdash A : B'}$$ si $B \overset{\Delta}{=}_\beta B'$ et $u \in s$

Variable (Var) : $$\frac{\Delta;\Gamma \vdash A : u}{\Delta;\Gamma, x : A \vdash x : A}$$ si $x \notin \Gamma$ et $u \in s$

Affaiblissement (Weak) : $$\frac{\Delta;\Gamma \vdash A : B \qquad \Delta;\Gamma \vdash C : u}{\Delta;\Gamma, x : C \vdash A : B}$$ si $x \notin \Gamma$ et $u \in s$

λ-abstraction (λ) : $$\frac{\Delta;\Gamma, x : A \vdash M : B \qquad \Delta;\Gamma \vdash \prod_{x:A} B : u}{\Delta;\Gamma \vdash \lambda(x : A).M : \prod_{x:A} B}$$ si $x \notin \Gamma$ et $u \in s$

Π-abstraction (Π) : $$\frac{\Delta;\Gamma \vdash A : u_1 \qquad \Delta;\Gamma, x : A \vdash B : u_2}{\Delta;\Gamma \vdash \prod_{x:A} B : u_2}$$ si $(u_1, u_2) \in s^2$

Application (App) : $$\frac{\Delta;\Gamma \vdash M : \prod_{x:A} B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash MN : B[x \coloneqq N]}$$

Définition (Def) : $$\frac{\Delta;\Gamma \vdash K : L \qquad \Delta;\Gamma \vdash M : N}{\Delta, C \coloneqq M : N;\Gamma \vdash K : L}$$ si $C \notin \Delta$

Axiome (Ax) : $$\frac{\Delta;\Gamma \vdash K : L \qquad \Delta;\Gamma \vdash N : u}{\Delta, C \coloneqq @ : N;\Gamma \vdash K : L}$$ si $C \notin \Delta$ et $u \in s$

Instanciation (Inst) : $$\frac{\Delta;\Gamma \vdash * : \square}{\Delta;\Gamma \vdash [C] : N}$$ si $C \coloneqq M : N \in \Delta$

Instanciation d'axiome (Ax-Inst) : $$\frac{\Delta;\Gamma \vdash * : \square}{\Delta;\Gamma \vdash [C] : N}$$ si $C \coloneqq @ : N \in \Delta$

3 Correspondance de Curry-Howard

L'intérêt de ce système formel réside dans ce qui se nomme la correspondance de Curry-Howard : on peut en fait faire correspondre le Calcul des Constructions et la logique intuitionniste : ainsi, une proposition logique intuitionniste est associable à un terme du Calcul des Constructions, et la proposition est vraie si et seulement si le terme associé est le type d'un autre, dans un certain contexte correspondant aux axiomes choisis. Mais contrairement à la logique intuitionniste, il est bien plus simple de vérifier si un terme du Calcul des Constructions est d'un type donné, car la forme de ce dernier s'approche d'un langage de programmation : vérifier un type revient en fait à un calcul (ici, les β-réductions successives).

On se place ici dans un contexte $(\emptyset, \Delta)$ tel que l'on dispose d'une preuve (V) de $\Delta;\Gamma \vdash * : \square$ et tel que Δ ne contienne pas d'axiome. C'est en effet suffisant pour que les théorèmes 1 et 2 soient vrais, ce sont qui eux donnent un intérêt au système formel. Ce n'est pas nécessaire, mais pour les autres contextes, il faut montrer séparément que ces théorèmes sont encore vrais. On verra un cas où le Théorème 1 1 tombent en défaut dans pour un certain contexte (Γ, Δ) ne respectant pas ces hypothèses.

Un terme du calcul des constructions qui peut se retrouver en position d'être un type (c'est-à-dire un terme N tel qu'il existe M tel que $\Delta;\emptyset \vdash M : N$ soit dérivable) peut-être associé à une proposition logique. Ces termes sont les variables, les constantes, $*$, $\square$ et la Π-abstraction. On peut d'ailleurs remarquer que le type d'un type est forcément $*$ ou $\square$, s'il en a un (voir 8).

Les éléments de $*$ sont des « types », les éléments dont le type est un type sont appelés « valeurs » et les éléments dont le type est $\square$ sont appelés « sortes ».

On peut alors associer à un type une proposition logique, et à un élément d'un type une preuve de la proposition associée au type. En effet, si un terme est bien typé, on peut déduire son type à partir de la structure du terme, on en donnera un algorithme après en 5.

On note $A \to B$ pour $\prod_{x:A} B$ si x n'apparaît pas comme variable libre dans B. On associe ce type à l'implication : en effet, si j'ai des termes $x : A$ et $f : A \to B$, alors j'ai un terme $f\ x : B$. Avec cette

correspondance, on a donc que si A est vrai (car on dispose d'une démonstration de A), et $A \to B$ aussi, alors B est vrai. Cela correspond bien à l'implication.

Une sorte est associée à une proposition logique du second ordre : une proposition logique qui quantifie sur les propositions. Dans les mathématiques usuelles, ce n'est pas utilisé : on ne peut pas énoncer le principe de récurrence sur $\mathbb{N}$, par exemple, comme cela : $\forall P$, si P est un prédicat unaire, $(P(0) \wedge (\forall n \in \mathbb{N}, P(n) \implies P(n+1))) \implies (\forall n \in \mathbb{N}, P(n))$, ce n'est pas un énoncé de la logique employée. C'est pourquoi on le reformule en passant par des parties de $\mathbb{N}$: $\forall A \subset \mathbb{N}, (0 \in A \wedge (\forall n \in \mathbb{N}, n \in A \implies n+1 \in A)) \implies A = \mathbb{N}$.

La logique associée au calcul des constructions permet de quantifier sur les propositions, et cela permet de définir, par exemple, le type $\bot \coloneqq \prod_{A:*} A : *$. Montrons d'abord que ce terme est bien typé dans le contexte $(\emptyset, \Delta)$:

$$\dfrac{\dfrac{}{\Delta;\emptyset \vdash * : \square}\,(\mathrm{V}) \qquad \dfrac{\dfrac{}{\Delta;\emptyset \vdash * : \square}\,(\mathrm{V})}{\Delta;\emptyset, A : * \vdash A : *}\,\mathrm{Var}}{\Delta;\emptyset \vdash \prod_{A:*} A : *}\,\Pi$$

$\bot$ est associé, par la correspondance de Curry-Howard, à la proposition fausse. En effet, supposons par l'absurde qu'on dispose d'une valeur $absurd : \bot$. Soit deux termes N et S tel que $\Delta;\emptyset \vdash N : S$ soit dérivable et que $\Delta;\emptyset \vdash M : N$ n'est pas dérivable quel que soit M (il en existe, d'après le Théorème 1 1). On note (A) une preuve que $\Delta;\emptyset \vdash absurd : \bot$ est valide, et (B) une preuve que $\Delta;\emptyset \vdash N : S$. Alors :

$$\dfrac{\dfrac{}{\Delta;\emptyset \vdash \prod_{A:*} A : *}\,(\mathrm{A}) \qquad \dfrac{}{\Delta;\emptyset \vdash N : S}\,(\mathrm{B})}{\Delta;\emptyset \vdash absurd\ N : N}\,\mathrm{App}$$

Ainsi $\Delta;\emptyset \vdash absurd\ N : N$ contredit notre hypothèse.

Donc il n'y a pas de termes M tels que $\Delta;\emptyset \vdash M : \bot$, donc par la correspondance, la proposition associée n'a pas de démonstration. De plus, si $\bot$ était non-vide, on aurait ainsi un élément de N pour chaque type N, donc pour chaque proposition P, une démonstration de P. C'est similaire à « si la proposition fausse est vraie, toutes les propositions sont vraies. » Ainsi $\bot$ représente bien la contradiction fausse.

On note alors $\neg A$ pour $A \to \bot$. En effet, $x : \neg A$ correspond alors à une démonstration de $A \implies Faux$, donc de la négation de la proposition associée à A. La logique utilisée ici est la logique intuitionniste : on ne dispose pas du tiers exclus. On ne peut donc pas déduire, notamment, que quelque soit A, $\neg\neg A \to A$. Mais on peut travailler dans la logique classique si on l'admet pour axiome : on travaille alors dans $(\emptyset; \Delta')$ tel que $\Delta' = \Delta, tiers\_exclus \coloneqq @ : \prod_{A:*}(\neg\neg A \to A)$. Ce contexte est bien valide car $\Delta;\emptyset \vdash \prod_{A:*}(\neg\neg A \to A) : \square$ et $\Delta;\emptyset \vdash * : \square$ est dérivable.

On peut également encoder $\wedge$ et $\vee$ dans le Calcul des Constructions, ainsi que les quantificateurs universels et existentiels, comme ceci :

Logique	**Calcul des Constructions**
Propositions	Types
Démonstrations	Termes
$A \Rightarrow B$	$A \to B$
$\bot$	$\prod_{a:*}\ a$
$\neg A$	$A \to \bot$
$\forall x \in S, P(x)$	$\prod_{x:S}\ P\ x$
$\exists x \in S, P(x)$	$\prod_{a:*}\ ((\prod_{x:S}(P\ x \to a)) \to a)$
$A \wedge B$	$\prod_{C:*}\ (A \to B \to C) \to C$
$A \vee B$	$\prod_{C:*}\ (A \to C) \to (B \to C) \to C$

TABLE 1 – Tableau récapitulatif de la correspondance de Curry-Howard

4 Théorèmes

On suppose avoir une preuve (V) de $\Delta;\Gamma \vdash * : \square$, ce qu'on désignera ensuite par « le contexte est valide ». On se placera dans ce cas pour la suite. On se place aussi dans un contexte où $\Gamma = \emptyset$ et où Δ ne contient pas d'axiome. Le premier théorème énonce que le système formel étudié « a un intérêt » : si tous les jugements étaient dérivablees, ce ne serait pas le cas. Le second permet d'assurer que l'algorithme donné en 5 termine et est correct.

Theorem 1 (Cohérence) *Il existe des termes N tel qu'il existe S tel que $\Delta;\emptyset \vdash N : S$ soit dérivable mais que $\Delta;\emptyset \vdash M : N$ n'est dérivable pour aucun terme M.*

Ainsi notre théorie est dite **cohérente** : on ne peut tout y démontrer. On en fournit une démonstration en 8.

On a donc restreint le contexte à Δ qui ne contient pas d'axiome et à $\Gamma = \emptyset$ car dans un contexte avec l'axiome $absurd \coloneqq @ : \bot$ ou un contexte contenant $(absurd : \bot)$, on peut dériver $[absurd]\ N : N$ ou $absurd\ N : N$ quel que soit le terme (lui-même dérivable) N, ce qui contredirait ce théorème.

Theorem 2 (Forte normalisation) *Le calcul des constructions est fortement normalisant : si un terme M est bien typé dans un contexte valide $(\emptyset, \Delta)$ où Δ ne contient pas d'axiome (donc si on peut dériver $\Delta;\Gamma \vdash M : N$ pour un certain N), alors N est unique et M est **fortement normalisable** : on peut appliquer n'importe quelle suite de β-réduction à M, cette suite sera forcément finie et le terme M' obtenu ne dépendra pas de cette suite.*

Voir Theoreme 10.5.2 de 1. La démonstration ne sera pas donnée ici.

Un contre exemple de terme fortement normalisable est donné via le point fixe de Curry : on note $Y \coloneqq \lambda(f : *).(\lambda(x : *).f(xx))(\lambda(x : *).f(xx))$, et on se donne un terme a, par exemple $a = *$ (cela fonctionne quelque soit a).

Alors $Ya \to_\beta Z$ où $Z = (\lambda(x : *).a(xx))(\lambda(x : *).a(xx))$ et
$Z \to_\beta a((\lambda(x : *).a(xx))(\lambda(x : *).a(xx))) = aZ$.
Donc $Ya \to_\beta Z \to_\beta aZ \to_\beta a(aZ) \to_\beta a(a(aZ)) \to_\beta \cdots$, cette suite est infinie.

5 Algorithme

On présente dans cette section un algorithme `type` pour déterminer, dans le contexte (Γ, Δ) et si le terme est typable, son type, ainsi qu'un autre qui détermine si $\Delta;\Gamma \vdash M : N$.

Pour le typage, si le terme est typable et le contexte valide, le type se déduit de la structure même du terme, ainsi que du contexte pour les variables et définitions. Pour la vérification, à chaque fois qu'un terme est vérifié, on applique toutes les β-réductions possibles dans le terme. On sait que cela termine d'après le théorème 2, et que le terme obtenu est $\overset{\Delta}{=}_\beta$ au précédent, donc on peut le lui substituer partout. Enfin, il ne reste qu'à appliquer les règles. Comme $Conv$ n'est pas employée, la seule règle où il peut y avoir ambiguïté est $Weak$, sinon on a toujours au plus une règle applicable. Ainsi, on applique $Weak$ seulement si aucune autre règle n'est applicable.

On définit ensuite `type` et `check` par filtrage de motifs, et ce sont des fonctions récursives. Chaque motif est préfixé du cas correspondant : on filtre sur la forme du terme dans `type`, et sur la règle à appliqué dans `check`. En cas d'ambiguïté, c'est le premier motif reconnu qui est appliqué, dans l'ordre de haut en bas.

- Variable : $\texttt{type}\ \Delta\ \Gamma\ x = A$ où $(x : A) \in \Gamma$
- Univers des types : $\texttt{type}\ \Delta\ \Gamma\ * = \square$
- λ-abstraction : $\texttt{type}\ \Delta\ \Gamma\ (\lambda(x : A).B) : \prod_{x:A}(\texttt{type}\ \Delta\ \Gamma\ B)$
- Π-abstraction : $\texttt{type}\ \Delta\ \Gamma\ (\prod_{x:A} B) = \texttt{type}\ \Delta\ \Gamma\ B$
- Application : $\texttt{type}\ \Delta\ \Gamma(A\ B) = N[x \coloneqq B]$ où $\texttt{type}\ \Delta\ \Gamma\ A = \prod_{x:M} N$
- Constante : $\texttt{type}\ \Delta\ \Gamma\ [C] = N$ où $(C \coloneqq M : N) \in \Delta$ ou $([C] \coloneqq @ : N) \in \Delta$
- Sinon : $\texttt{type}\ \Delta\ \Gamma\ M$ n'est pas défini.

— Empty : $\texttt{check}\ \Delta\ \emptyset\ *\ \Box = \texttt{True}$
— Var : $\texttt{check}\ \Delta\ (\Gamma, x : A)\ x\ A = \texttt{check}\ \Gamma\ \Delta\ u\ \&\&\ u \in \texttt{s}$ où $u = \texttt{type}\ \Gamma\ \Delta\ A$
— λ : $\texttt{check}\ \Gamma\ \Delta\ (\lambda(x : A).M)\ (\prod_{x:A} B) = \texttt{check}\ \Delta\ (\Gamma, x : A)\ M\ B\ \&\&\ \texttt{check}\ \Gamma\ \Delta\ (\prod_{x:A} B)\ u\ \&\&\ u \in \texttt{s}$ où $u = \texttt{type}\ \Gamma\ \Delta\ B$ et en renommant la variable x si $x \in \Gamma$
— Π : $\texttt{check}\ \Gamma\ \Delta\ (\prod_{x:A} B)\ N = \texttt{check}\ \Gamma\ (\Delta, x : A)BN\ \&\&\ \texttt{check}\ \Gamma\ \Delta\ A\ u\ \&\&\ (N, u) \in \texttt{s}^2$ où $u = \texttt{type}\ \Gamma\ \Delta A$ et en renommant la variable x si $x \in \Gamma$
— App : $\texttt{check}\ \Gamma\ \Delta\ (MN)\ C = \texttt{check}\ \Gamma\ \Delta\ M\ (\prod_{x:A} B)\ \&\&\ \texttt{check}\ \Gamma\ \Delta\ NA\ \&\&\ C = B[x \coloneqq N]$ si $\texttt{type}\ \Gamma\ \Delta\ M = \prod_{x:A} B$
— Inst : $\texttt{check}\ \Gamma\ \Delta\ [C] : N = \texttt{True}$ si $(C \coloneqq @ : N) \in \Delta$ ou s'il existe M tel que $(C \coloneqq M : N) \in \Delta$
— Weak : $\texttt{check}\ \Delta\ (\Gamma, x : C)\ A\ B = \texttt{check}\ \Delta\ \Gamma\ A\ B\ \&\&\ \texttt{check}\ \Delta\ \Gamma\ C\ u\ \&\&\ u \in \texttt{s}$ où $u = \texttt{type}\ \Gamma\ \Delta\ C$
— Sinon : $\texttt{check}\ \Gamma\ \Delta\ A\ B = \texttt{False}$

J'ai implémenté cet algorithme dans le programme OCAML fourni avec ce rapport. Pour l'utiliser, il faut compiler le programme (`dune build`) puis lancer l'exécutable obtenu, en passant en paramètre de ligne de commande le nom d'un fichier au format suivant : Chaque ligne est soit vide, soit de la forme `def` $C \coloneqq M : N$;, soit de la forme `def` $C \coloneqq @ : N$;, où M et N sont des termes. Le second cas correspond à un axiome. C désigne le nom donné à cette définition ; il doit ne pas être déjà utilisé. La syntaxe est la même que celle employée ici, sauf que les constantes commencent par des majuscules au lieu d'être mises entre crochets, et les variables commencent par des minuscules. On note $\backslash(x : A).B$ pour $\lambda(x : A).B$, $/\backslash(x : A).B$ pour $\prod_{x:A} B$ et ¤ pour $\Box$. Pour écrire ce programme, j'ai employé les bibliothèques `ocamllex` et `menhir` pour définir la grammaire et convertir le fichier texte en une représentation informatique des termes.

Le programme analyse ensuite les lignes une après l'autre : lorsqu'une ligne non vide est rencontrée, elle est vérifiée par l'algorithme précédent, puis ajoutée dans le contexte, et le programme passe à la ligne suivante. Ainsi, si le programme ne renvoie rien, toutes les définitions sont correctes. Si une vérification échoue, le programme renvoie une erreur. J'ai inclus les fichiers `test.tipe` et `test_rate.tipe` (dont la vérification échoue) qui servent d'exemple.

De plus, j'ai employé dans mon programme des indices de De Bruijn pour représenter les variables. C'est une technique qui permet de se passer d'α-réductions, chaque variable liée est représentée par le nombre de symboles « liants » (λ ou Π) dans le terme avant celui qui l'introduit, et les variables libres sont représentées par le nombre de symboles liants avant elle auquel on ajoute le « numéro » de la variable libre, qui est son ordre d'apparition dans le contexte Γ. Les règles de dérivation restent pratiquement les mêmes, mais les indices de De Bruijn en rendent la compréhension plus complexe, aussi je ne les ai pas présentés directement ici. Enfin, pour faciliter l'utilisation du programme, le fichier à analyser est lui écrit avec des variables sous forme de lettres, j'ai donc implémenter un algorithme qui transforme un terme avec de telles variables en terme avec indice de De Bruijn.

6 Références

1. Nederpelt, R., & Geuvers, H. (2014). Acknowledgements. In *Type Theory and Formal Proof : An Introduction*. Cambridge : Cambridge University Press. `https://doi.org/10.1017/CBO9781139567725`

7 Annexe : δ-équivalence

Soit Δ une liste de définitions ou d'axiomes.
La relation de δ-réduction $\to_\delta$ se définit comme suit, par induction sur la structure des termes :
— $[C] \to_\delta M$ s'il existe N tel que $C \coloneqq M : N \in \Delta$
— $\lambda(x : A).B \to_\delta \lambda(x : A').B$ si $A \to_\delta A'$
— $\lambda(x : A).B \to_\delta \lambda(x : A).B'$ si $B \to_\delta B'$
— $\prod_{x:A}.B \to_\delta \prod_{x:A'}.B$ si $A \to_\delta A'$
— $\prod_{x:A}.B \to_\delta \prod_{x:A}.B'$ si $B \to_\delta B'$
— $AB \to_\delta A'B$ si $A \to_\delta A'$
— $AB' \to_\delta AB'$ si $B \to_\delta B'$

La relation $\overset{\Delta}{=}_\beta$ est la clôture réflexive, symétrique et transitive de la réunion des relations de β-réduction et de δ-réduction, la β-réduction étant définie au sens usuel du λ-calcul.

8 Annexe : Preuve du Théorème 1

Montrons ce résultat :

Pour ce faire, plaçons nous dans un tel contexte $(\Delta, \emptyset)$ où Δ ne contient pas d'axiome, et vérifions que $\lambda(x : *).x : \prod_{x:*} *$.

$$\dfrac{\dfrac{\dfrac{(V)}{\Delta;\emptyset \vdash * : \square}}{\Delta, x : * \vdash x : *}\,\text{Var} \qquad \dfrac{\dfrac{(V)}{\Delta;\emptyset \vdash * : \square} \qquad \dfrac{\dfrac{(V)}{\Delta;\emptyset \vdash * : \square} \qquad \dfrac{(V)}{\Delta;\emptyset \vdash * : \square}}{\Delta; x : * \vdash * : \square}\,\text{Weak}}{\Delta;\emptyset \vdash \prod_{x:*} * : \square}\,\Pi}{\Delta;\emptyset \vdash \lambda(x : *).x : \prod_{x:*} *}\,\lambda$$

Cela fournit également une preuve que $\Delta;\emptyset \vdash \prod_{x:*} * : \square$.

Ainsi, il existe des termes A tel qu'il existe des termes B et C tels que $A : B$ et $B : C$, car on a $\lambda(x : *).x : \prod_{x:*} *$ et $\prod_{x:*} * : \square$.

Montrons que si $A : B$ et $B : C$ alors $C \in$ s. Donc on ne peut pas dériver de terme $M : \lambda(x : *).x$, ce qui démontrera notre théorème. Sans perte de généralité, nous supposerons $\Delta = \emptyset$, car on peut substituer les $[C]$ dans les termes par M quand $C \coloneqq M : N$ est dans Δ, et insérer la preuve de $M : N$ dans la preuve originale aux endroits où la règle *Inst* était appliquée.

Montrons le résultat par récurrence sur la hauteur d'un arbre de preuve. On suppose qu'on a une démonstration, de hauteur n, de $\emptyset;\emptyset \vdash A : B$.

Si $n = 0$, l'unique règle appliquée est *Empty*, donc $A = *$, $B = \square$. Or il n'existe pas de C tel que $\square : C$, c'est absurde.

Supposons $n \geq 1$ et que pour tout k tels que $0 \leq k < n$, le résultat soit acquis.

On procède par induction sur la structure d'arbre de démonstration.

- Empty : Cette règle ne peut s'appliquer en premier car la démonstration étudiée est de hauteur $n \geq 1$.
- Conv : Dans ce cas, $C \in$ s, par unicité du type.
- Var : Dans ce cas, $C \in$ s, par unicité du type.
- Weak : Cette règle ne peut s'appliquer en premier car $\Gamma = \emptyset$
- λ : Dans ce cas, $C \in$ s, par unicité du type.
- Π : Dans ce cas, $B \in$ s, or $B : C$ et $\square$ n'a pas de type donc $B = *$. Par unicité du type, $C = \square \in$ s.
- App : Dans ce cas, on note $A = MN$, et soient K et L tels que $\emptyset;\emptyset \vdash M : \prod_{x:K} L$, $B = L[x \coloneqq N]$. La démonstration de $\emptyset;\emptyset \vdash M : \prod_{x:K} L$ est de hauteur $k < n$, et nécessite, une démonstration de $\emptyset; x : K \vdash L : I$ avec I un terme dans s, car la seule règle qui introduit un Π nécessite $I \in$ s. Donc $\emptyset;\emptyset \vdash L[x \coloneqq N] : I[x \coloneqq N]$ (Lemma 10.4.11 de 1) et $I[x \coloneqq N] = I$ car $I \in$ s. Or $B = L[x \coloneqq N]$. Donc $C = I \in$ s par unicité du type.

Les autres règles ne sont pas employées car $\Delta = \emptyset$. Donc $C \in s$.

In fine, le résultat vaut quelque soit n.

9 Annexe : Code

Pour lancer ce code, placez tous les fichiers dans un même dossier, installez `ocaml`, `dune` et `menhir` sur votre machine (`opam` peut être utile pour installer ces derniers). Je n'ai testé le programme que sur Linux et WSL dans Windows. Lancez ensuite `dune build` dans le répertoire où vous avez copié les fichiers. Cela produit un exécutable nommé `main.exe` dans `./_dune/default/`. Lancez ensuite l'exécutable en lui passant en ligne de commande le nom du fichier de preuve à vérifier. J'ai inclus `test.tipe` et `test_rate.tipe` à cet effet.

9.1 dune

```
(ocamllex "lexer")

(menhir
 (modules "parser"))

(executable
 (name main)
 (libraries)
 (flags
  (-w @1..3@5..28@30..39@43@46..47@49..57@61..62-40 -strict-sequence
    -strict-formats -short-paths -keep-locs))
 (ocamlc_flags (-g))
 (ocamlopt_flags (-g))
 (modules
  ("main" "lambda" "lexer" "parser" "syntax")))
```

9.2 dune-project

```
(lang dune 2.8)

(name TIPE)

(version 0.1.0)

(using menhir 2.1)
```

9.3 lambda.ml

```
type sort = Type | Kind (* * | □ *)

type term =
  Var of int (* de Bruijn indices, starts at 1 *)
  | Sort of sort
  | Apply of (term * term) (* M N *)
  (* the type lives in a context with one variable less than the body *)
  | Abstraction of (term * term) (* (Var (x, A), m) = \x : A. m; m : * *)
  | ProductType of (term * term) (* (Var (x, A), B) = /\x : A. B; B : □ *)
  | Constant of int (* definition order *)

let rec subst s = function
| Var n -> s n
| Apply (a, b) -> Apply (subst s a, subst s b)
| Abstraction (a, b) -> Abstraction (subst s a, subst (function
  | 1 -> Var 1
  | n -> incr (s (n - 1))
  ) b)
| ProductType (a, b) -> ProductType (subst s a, subst (function
  | 1 -> Var 1
  | n -> incr (s (n - 1))
  ) b)
| t -> t
and incr t = subst (fun k -> Var (k + 1)) t
```

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

(* it preserves well-typing *)
let rec reduce = function
| Abstraction (a, b) -> Abstraction (reduce a, reduce b)
| ProductType (a, b) -> ProductType (reduce a, reduce b)
| Apply (a, b) -> begin match reduce a with
  | Abstraction (_, t) -> substitute 1 t (reduce b)
  | ProductType (_, t) -> substitute 1 t (reduce b)
  | t -> Apply (reduce t, reduce b)
end
| t -> t

type definition = {
  body : term option; (* None stands for an axiom. It is written @ in the grammar *)
  ty : term;
}

type state = {
  defs : definition array; (* [| D_1, ..., D_n |] array of previously verified definitions,
    D_k matches Constant k *)
  vars : term list (* list A_n, ..., A_i, ..., A_1 of the type A_i of the i-th free var *)
}

let rec show_term = function
| Var n -> string_of_int n
| Sort Type -> "*"
| Sort Kind -> "¤"
| Apply (a, b) -> "(" ^ show_term a ^ " " ^ show_term b ^ ")"
| Abstraction (a, b) -> "( \\ " ^ show_term a ^ " . " ^ show_term b ^ ")"
| ProductType (a, b) -> "( /\\ " ^ show_term a ^ " . " ^ show_term b ^ ")"
| Constant n -> "[" ^ string_of_int n ^ "]"

let show_def state =
  let s = ref "" in
  for i = 0 to Array.length state.defs - 1 do
    let def = state.defs.(i) in
    s := !s ^ "def [" ^ string_of_int i ^ "] := " ^ (match def.body with Some t ->
        show_term t | None -> "@") ^ " : " ^ show_term def.ty ^ ";\n"
  done;
  !s

let show state term ty =
  print_newline ();
  print_string (show_def state);
  List.iter (fun t -> print_endline (show_term t ^ ";")) (List.rev state.vars);
  print_endline ("|- " ^ (match term with Some t -> show_term t | None -> "@") ^
    " :" ^ show_term ty ^ ";");
  print_newline ()

type error =
```

```
  | NotTypable of term * string (* t, explanation *)
  | IsNotASort of term * term (* t, ty where t : ty and ty should be a sort *)
  | ConstantOutOfBound of int (* k with Constant k which don't exists *)
  | TypesDoNotMatch of term * term * string (* t1, t2, explanation where t1 <> t2 *)
  | NotClosedTerm of int (* k with Var k which isn't binded *)
exception Error of error

(* unfold all non-axiom constant definitions *)
let rec unfold state = function
| Apply (a, b) -> Apply (unfold state a, unfold state b)
| Abstraction (a, b) -> Abstraction (unfold state a, unfold state b)
| ProductType (a, b) -> ProductType (unfold state a, unfold state b)
| Constant k -> begin match state.defs.(k).body with
  | Some t -> unfold state t
  | None -> Constant k
  end
| t -> t

let push_var t state = {state with vars = t :: state.vars};;
let pop_var state = match state.vars with
  | h :: t -> (h, {state with vars = t})
  | [] -> failwith "pop from empty context"

(* we can massively use reduce because we know the terms are well-typed *)
let rec typing state t = begin match reduce 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
  (* we retrieve the type of t *)
  let ty = List.nth state.vars (k - 1) in
  (* we adapt the type to the current context *)
  (* let vars = Array.of_list (
    List.filteri (fun i _ -> i >= k) state.vars) in *)
  incr_by_n k ty
| Sort Type -> Sort Kind
| Sort Kind -> raise (Error (NotTypable (Sort Kind,
  "[This state shouldn't be reached] term Kind does not have a type")))
| Apply (a, b) -> begin match typing state a with
  | ProductType (x, y) -> Apply (ProductType (x, y), b) (* it is reduced after *)
  | t -> raise (Error (NotTypable (Apply (a, b),
    "[This state shouldn't be reached] " ^
    "you can't apply a on b with a = " ^ show_term a ^ " : " ^ show_term t ^
    " and b = " ^ show_term b)))
end
| Abstraction (a, b) -> ProductType (a, typing (push_var a state) b)
| ProductType (a, b) -> typing (push_var a state) b
| Constant k -> state.defs.(k).ty
end |> reduce


let fail_if_not_sort state t = match typing state t with
| Sort _ -> ()
```

```
| ty -> raise (Error (IsNotASort (t, ty)))

let rec check state = let n = List.length state.vars in function
| Var k when k <= n ->
  if k = 0
  then
    failwith "k = 0";

  let (ty, st) = pop_var state in (* safe because k >= 1*)
  check st ty;
  fail_if_not_sort st ty;

  if k = 1
  then begin
    let t1 = typing state (Var k)
    and t2 = reduce (incr ty) (* ty live in a context with one less variable *)
    in if t1 <> t2
    then raise (Error (TypesDoNotMatch (t1, t2, "Var " ^ string_of_int k ^
    " should be of type t2 = " ^ show_term t2 ^ ", not t1 = " ^ show_term t1)))
  end else
    check st (Var (k - 1))
| Var k (* k > n *) -> raise (Error (NotClosedTerm k)) (* the term is not closed *)
| Sort Type when state.vars = [] -> ()
| Sort Type ->
  let (ty, st) = pop_var state in
  check st ty;
  fail_if_not_sort st ty;
  check st (Sort Type);
| Sort Kind -> raise (Error (NotTypable (Sort Kind, "term Kind does not have a type")))
| Apply (a, b) ->
  check state a;
  check state b;
  let x = typing state b in
  begin match typing state a with
  | ProductType (y, _) when y = x -> ()
  | t -> raise (Error (NotTypable (Apply (a, b),
    "you can't apply a on b with a = " ^ show_term a ^ " : " ^ show_term t ^
    " and b = " ^ show_term b ^ " : " ^ show_term x)))
  end
| Abstraction (a, b) ->
  let new_state = push_var a state in
  check new_state b;
  let c = typing new_state b in
  check state (ProductType (a, c));
  fail_if_not_sort state (ProductType (a, c));
| ProductType (a, b) ->
  check state a;
  fail_if_not_sort state a;
  let new_state = push_var a state in
  check new_state b;
  fail_if_not_sort new_state b;
| Constant k -> if (k >= Array.length state.defs)
  then raise (Error (ConstantOutOfBound k))
```

9.4 lambda.mli

```
type sort = Type | Kind;; (* * | □ *)

type term =
  Var of int (* de Bruijn indices *)
  | Sort of sort
  | Apply of (term * term) (* M N *)
  | Abstraction of (term * term) (* (Var (x, A), m) = \x : A. m; m : * *)
  | ProductType of (term * term) (* (Var (x, A), B) = /\x : A. B; B : □ *)
  | Constant of int (* definition order *)


(* substitute n a b = a[n:=b] *)
val substitute : int -> term -> term -> term

(* preserves well-typedness, reduce application *)
val reduce : term -> term

type definition = {
  body : term option; (* None stands for an axiom. It is written @ in the grammar *)
  ty : term;
}

type state = {
  defs : definition array; (* [| D_1, ..., D_n |] array of previously verified definitions,
    D_k matches Constant k *)
  vars : term list (* list A_n, ..., A_i, ..., A_1 of the type A_i of the i-th free var *)
}

type error =
  | NotTypable of term * string (* t, explanation *)
  | IsNotASort of term * term (* (t, ty) where t : ty and ty should be a sort *)
  | ConstantOutOfBound of int (* k with Constant k which don't exists *)
  | TypesDoNotMatch of term * term * string (* (t1, t2, explanationn) where t1 <> t2 *)
  | NotClosedTerm of int (* k with Var k which isn't binded *)
exception Error of error

(* unfold all non-axiom constant definitions *)
val unfold : state -> term -> term

(* take a well-formed state and a well-typed term and returns its type *)
val typing : state -> term -> term

(* take a well-formed state and a term and checks its well-typedness *)
val check : state -> term -> unit

(* (Delta; Gamma) |- M or @ : N *)
val show : state -> term option -> term -> unit

val show_term : term -> string
```

9.5 lexer.mli

```
val token : Lexing.lexbuf -> Parser.token
```

9.6 lexer.mll

```
{

open Parser

}

rule token = parse
  | [' ' '\t' '\n' '\r'] { token lexbuf } (* skip whitespaces *)
  | "def" { DEF }
  | "*" { TYPE }
  | "¤" { KIND }
  | "@" { AXIOM }
  | "." { DOT }
  | ":=" { ASSIGN }
  | ":" { COLON }
  | "\\" { LAMBDA }
  | "/\\" { PRODUCT }
  | "(" { LPAR }
  | ")" { RPAR }
  | ";" { SEMICOLON }
  | ['a'-'z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']* as id { VAR_IDENT id }
  | ['A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']* as id { CST_IDENT id }
  | eof { EOF }
```

9.7 main.ml

```
let perform path f =
  let ic = open_in path in
  let res = f ic in
  close_in ic;
  res;;

let transform l =
  let rec aux cst_table i = function
  | [] -> []
  | def :: tl -> let (name, def) = Syntax.translate_def cst_table def in
    def :: aux (Syntax.AssocTable.add name i cst_table) (i + 1) tl
  in aux Syntax.AssocTable.empty 0 l;;

(* Return the reduced definition if it is correct, raise an error else *)
let check_def state {Lambda.body = b; Lambda.ty = ty} =
  (* Lambda.show state b ty; *)
  let ty = Lambda.unfold state ty in
  Lambda.check state ty;
  let ty = Lambda.reduce ty in
  let body = Option.map (fun term ->
    let term = Lambda.unfold state term in
    Lambda.check state term;
    let real_ty = Lambda.typing state term in
    if real_ty <> ty
    then raise (Lambda.Error (Lambda.TypesDoNotMatch (real_ty, ty,
      "term x = " ^ Lambda.show_term term ^ " : " ^ Lambda.show_term real_ty ^
      " is not of the provided type " ^ Lambda.show_term ty)))
```

```
  else Lambda.reduce term
  ) b in
  {Lambda.body = body; Lambda.ty = ty}

let check_defs defs =
  let rec aux ({Lambda.defs = defs; _} as state) = function
  | def :: tl -> let def = check_def state def in
    aux ({state with Lambda.defs = Array.append defs [|def|]}) tl
  | [] -> ()
  in aux {Lambda.defs = [||]; Lambda.vars = []} defs;;


let main () =
  let defs = Array.to_list Sys.argv |> List.tl |>
    List.map (fun p ->
      try
        let def_list = perform p (fun ic ->
          let lexbuf = Lexing.from_channel ic in
          Parser.parse_file Lexer.token lexbuf
        ) in def_list
      with
        | Failure s -> print_endline ("ERROR: " ^ s); []
    ) |> List.flatten |> transform in
    check_defs defs;;

main ();;
```

9.8 parser.mly

```
%{

open Syntax

%}

%token<string> VAR_IDENT (* alphanumerical ascii, starts with a lowercase letter *)
%token<string> CST_IDENT (* alphanumerical ascii, starts with an uppercase letter *)
%token DEF (* def *)
%token TYPE (* * *)
%token KIND (* ¤ *)
%token AXIOM (* @ *)
%token DOT (* . *)
%token COLON (* : *)
%token LAMBDA (* \ *)
%token PRODUCT (* /\ *)
%token ASSIGN (* := *)
%token LPAR (* ( *)
%token RPAR (* ) *)
%token SEMICOLON (* ; *)
%token EOF

%start<Syntax.tok_def list> parse_file

%right DOT
```

```
%nonassoc VAR_IDENT CST_IDENT TYPE KIND LAMBDA PRODUCT LPAR
%left APP

%%

expr:
  | LPAR ; e = expr; RPAR { e }
  | TYPE { Sort Lambda.Type }
  | KIND { Sort Lambda.Kind }
  | e1 = expr; e2 = expr; %prec APP { Apply (e1, e2) }
  | LAMBDA ; v = var_intro ; DOT ; e = expr %prec DOT
    { let (s, ty) = v in Abstraction (s, ty, e) }
  | PRODUCT ; v = var_intro; DOT ; e = expr %prec DOT
    { let (s, ty) = v in ProductType (s, ty, e) }
  | c = CST_IDENT { Constant c }
  | s = VAR_IDENT { Var s }

var_intro:
  LPAR ; s = VAR_IDENT ; COLON ; ty = expr ; RPAR { (s, ty) }

expr_or_axiom:
  | e = expr { Some e }
  | AXIOM { None }

def:
  DEF ; c = CST_IDENT ; ASSIGN ;
  e = expr_or_axiom ; COLON ; ty = expr ; SEMICOLON {{
      name = c;
      body = e;
      ty = ty;
  }}

parse_file:
  l = list(def); EOF { l }

%%
```

9.9 syntax.ml

```
module AssocTable = Map.Make(String)

type tok_term =
  | Var of string
  | Sort of Lambda.sort
  | Apply of tok_term * tok_term
  | Abstraction of string * tok_term * tok_term
  | ProductType of string * tok_term * tok_term
  | Constant of string

type tok_def = {
  name : string;
  body : tok_term option;
  ty : tok_term;
}
```

```
let translate_term cst_table t =
  (* n is the number of binders already in scope *)
  (* current is the next free binding index *)
  let rec aux n var_table = function
  | Var s ->
    begin try
      Lambda.Var (AssocTable.find s var_table) (* fail if the term is not closed *)
    with Not_found ->
      print_endline (s ^ " is not found.");
      raise Not_found
    end
  | Sort u -> Lambda.Sort u
  | Apply (a, b) ->
    let a = aux n var_table a in
    let b = aux n var_table b in
    Lambda.Apply (a, b)
  | Abstraction (x, a, b) ->
    let a = aux n var_table a in
    let var_table = var_table |> AssocTable.map (fun k -> k + 1)
    |> AssocTable.add x 1 in
    let b = aux (n + 1) var_table b in
    Lambda.Abstraction (a, b)
  | ProductType (x, a, b) ->
    let a = aux n var_table a in
    let var_table = var_table |> AssocTable.map (fun k -> k + 1)
    |> AssocTable.add x 1 in
    let b = aux (n + 1) var_table b in
    Lambda.ProductType (a, b)
  | Constant s -> Lambda.Constant (AssocTable.find s cst_table)
        (* fail if the constant is not previously defined *)
  in let t = aux 0 AssocTable.empty t
  in t

let rec show_syntax_term = function
| Var s -> s
| Sort u -> Lambda.show_term (Lambda.Sort u)
| Apply (a, b) -> show_syntax_term a ^ " " ^ show_syntax_term b
| Abstraction (x, a, b) ->
    "(\\(" ^ x ^ " : " ^ show_syntax_term a ^ ") . " ^ show_syntax_term b ^ ")"
| ProductType (x, a, b) ->
  "(/\\(" ^ x ^ " : " ^ show_syntax_term a ^ ") . " ^ show_syntax_term b ^ ")"
| Constant s -> "[" ^ s ^ "]"
let _ = show_syntax_term

let translate_def cst_table def =
  let ty = translate_term cst_table def.ty
  and body = Option.map (translate_term cst_table) def.body in
  (def.name, { Lambda.body = body; Lambda.ty = ty})
```

9.10 syntax.mli

```
module AssocTable: Map.S with type key = string
```

```
type tok_term =
  | Var of string
  | Sort of Lambda.sort
  | Apply of tok_term * tok_term
  | Abstraction of string * tok_term * tok_term
  | ProductType of string * tok_term * tok_term
  | Constant of string

type tok_def = {
  name : string;
  body : tok_term option;
  ty : tok_term;
}

(* the AssocTable contains the constant bindings *)
val translate_term : int AssocTable.t -> tok_term -> Lambda.term

(* idem *)
val translate_def : int AssocTable.t -> tok_def -> string * Lambda.definition
```

9.11 test_rate.tipe

```
def Erreur := \(x : *). x : *;
```

9.12 test.tipe

```
def N := @ : *;
def O := @ : N;
def Succ := @ : /\(n:N).N;

def Func := \(a : *).\(b : *)./\(x : a).b : /\(a : *)./\(b : *).*;
def Id := \(a : *).\(x : a).x : /\(a : *). /\(x : a).a;

def Empty := /\(a : *). a : *;
def Not := \(a : *). Func a Empty : /\(a : *).*;
```