Mathématiques inversées de Coq

Yoan Géran

Mathématiques inversées de Coq

L'exemple de GeoCoq

Yoan GÉRAN

M2 student in Computer Science
ENS Paris-Saclay
2020-2021

Stage INRIA Saclay - Équipe Deducteam

Sous la direction de Gilles DOWEK et
Olivier HERMANT

Du 15/03/21 au 10/08/21

La preuve formelle est un domaine important de l'informatique, et de nombreux outils, tels que les assistants de preuve, permettent de formaliser des raisonnements mathématiques et de vérifier qu'une preuve est correcte. Néanmoins, ces différents outils ont chacun leur formalisme logique, et il est compliqué de récupérer une preuve écrite dans un logiciel X dans un autre logiciel.

L'équipe Deducteam travaille entre autres sur Dedukti, un *framework* logique dans lequel on peut encoder des théories logiques. L'idée est alors d'utiliser Dedukti comme « passerelle de traduction » entre les outils de preuves.

Durant mon stage, j'ai traduit les preuves du Livre I des Éléments d'Euclide, formalisées en Coq, vers d'autres systèmes de preuves en passant par Dedukti. Ceci permet de montrer sur un exemple qui ne semble pas compliqué (les Éléments d'Euclide n'utilisent pas, *a priori*, beaucoup de fonctionnalités de Coq) les possibilités offertes par Dedukti pour le partage de preuves Coq.

1 Les mathématiques inversées et le partage de preuves

1.1 Les systèmes de preuve

Les assistants de preuve nous permettent d'utiliser les capacités des machines à manipuler rapidement de grosses expressions pour formaliser des raisonnements. Par exemple, le théorème des quatre couleurs, prouvé avec un assistant de preuve, a demandé l'étude d'un peu plus de mille cas !

De plus, il est plus facile de croire en une preuve vérifiée informatiquement, qu'en une preuve énoncée et écrite « normalement ». Ainsi, avoir une preuve formalisée dans un assistant de preuve permet également d'en être plus sûr.

Néanmoins, il faut « faire confiance à l'assistant de preuves » et le choix du formalisme logique de l'assistant est alors important : nous voulons pouvoir exprimer la preuve, et nous voulons le faire de manière simple et intuitive. Notons que cela demande d'étudier le formalisme, notamment pour des raisons de cohérence.

La correspondance de Curry-Howard et le λ-calcul typé offrent un cadre parmi d'autres qui correspond à cela. L'idée est la suivante : les preuves sont des objets, les propositions sont des types et une preuve de la proposition P et un objet p de type P (le type P) est habité.

Ainsi, une implication $P \to Q$ peut être vue comme une fonction du type P vers le type Q (à partir d'une preuve de P, on construit la preuve de Q). Ce point de vue qui rapproche la déduction et le calcul nous permet de considérer une preuve d'un théorème comme un programme qui est la preuve du théorème.

Par exemple, une preuve que si la distance de A à B est la même que celle de A à C, alors la distance de A à B est la même que celle de C à A aurait ce type.

```
(A: Point) -> (B: Point) -> (C: Point) ->
(H: d(A, B) = d(A, C)) -> d(A, B) = d(C, A).
```

Un élément de ce type est une fonction qui prend trois points et un élément `H` du type `d(A, B) = d(A, C)` (donc une preuve) et retourne un élément du type `d(A, B) = d(C, A)`. On peut également également ce type avec une implication.

```
(A: Point) -> (B: Point) -> (C: Point) ->
d(A, B) = d(A, C) => d(A, B) = d(C, A).
```

Un élément de ce typr est une fonction qui prend en paramètre trois points et retourne un élément du type `d(A, B) = d(A, C) => d(A, B) = d(C, A)`, donc une preuve que l'implication est vraie.

Cette approche se base sur du λ-calcul typé et il nous faut décider quelles abstractions sont autorisés ce qui offre plusieurs formalismes et plusieurs fonctionnalités. En λ-calcul simplement typé, la seule abstraction autorisée est celle qui permet de créer des termes qui dépendent de terme, par exemple, une fonction qui prend un point et retourne un point. Mais on peut par exemple autoriser un type à dépendre d'un type ! Ces différentes fonctionnalités sont représentés sur le λ-cube en figure 1.

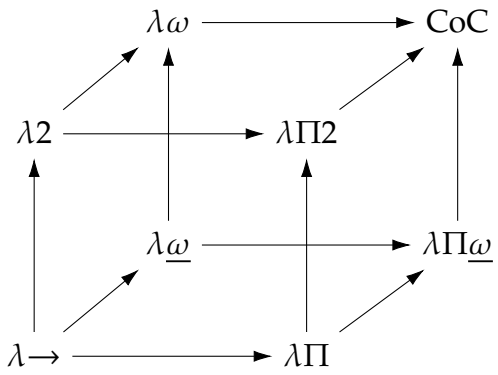


FIGURE 1 – Le lambda-cube.

— Sur l'axe des y, on a les termes qui peuvent dépendre de type, c'est-à-dire du *polymorphisme* : par exemple, l'identité est polymorphe, elle peut s'appliquer à un élément de n'importe quel type, donc est du type `forall A, A -> A`.
— Sur l'axe des x, on a les types qui peuvent dépendre de types, donc des *opérateurs de types* : par exemple, `pair` qui aurait le type `Type -> Type` et donc `pair Point` serait un `Type`, celui des paires de points.

— Sur l'axe des z, on a les types qui peuvent dépendre de termes ; ce sont des *types dépendants* : par exemple, `vector` aurait le type `nat -> Type` et donc `vector 4` serait un `Type`, celui des vecteurs à 4 éléments.

En autorisant toutes les abstractions, on obtient CoC, le Calcul des Constructions, où on a les types suivants.

— Les termes ont leur type (par exemple on a des types `Point`, `Prop`, etc.).
— Puisqu'on utilise les types comme des objets, ils ont également un type qui est `Type`. Ainsi, `Point` a le type `Type`.
— On peut abstraire sur des types ; on donne alors à `Type` le type `Kind` car pour des raisons de cohérence, il ne peut pas avoir le type `Type` (*cf.* [6, 3]).

Plus on autorise des fonctionnalités, plus on obtient un langage expressif. De plus, ces fonctionnalités semblent être utiles pour faire des preuves de manière simple et intuitive. Par exemple, nous voulons faire des preuves avec des polynômes d'un certain degré (types dépendants) ou sur des espaces vectoriels quelconques (opérateurs de types).

Pourtant, les assistants de preuve n'offrent pas tous ces fonctionnalités (par exemple, les systèmes basés sur HOL n'ont pas de types dépendants). Ceci rend compliqué le partage de preuves, alors que c'est quelque chose que nous devrions chercher pour plusieurs raisons.

— Gain de temps : ne pas refaire les preuves dans tous les systèmes.
— Gain de sûreté : car la preuve est vérifiée dans plusieurs systèmes.
— Gain de « popularité » : notamment auprès des mathématiciens.

1.2 L'assistant de preuve Coq

L'assistant de preuves Coq se base sur le calcul des constructions auquel il rajoute quelques éléments.

1.2.1 Les types inductifs

Nous pouvons créer le type des entiers naturels, ou encore un type des listes (qui mélange même type inductif et polymorphisme).

```
Inductive list (A : Type) :=
| nil : list A
| cons : A -> list A -> list A.
```

1.2.2 Les univers

En Coq, 0 a le type `nat`, qui a le type `Set`, qui lui-même a le type `Type`. Mais Coq permet d'abstraire sur n'importe quel type d'objets et nous pouvons écrire une fonction qui prend en paramètre un objet de type `Kind`. Pour des raisons de cohérence là-encore, cette fonction ne peut pas être de type `Kind` et il nous faut un nouveau type avec lequel on fait la même chose et ainsi de suite...

Ceci mène à une hiérarchie de types (appelés *univers* en Coq). $\texttt{Type}_0$ correspond à notre `Type` « classique » (le type de `Set`), $\texttt{Type}_1$ au type de $\texttt{Type}_0$, etc. On a alors deux règles pour les univers de Coq.

$$\forall i, \texttt{Type}_i \in \texttt{Type}_{i+1} .$$
$$\forall i < j, \texttt{Type}_i \in \texttt{Type}_j .$$

Ainsi, $\texttt{Type}_i$, et tout terme de type $\texttt{Type}_{i+1}$a le type $\texttt{Type}_j$ pour tout $j > i$ et n'a pas seulement le type $\texttt{Type}_{i+1}$. Il s'agit d'univers cumulatifs.

1.3 Les mathématiques inversées et le partage de preuves

Les *mathématiques inversées* sont au cœur de mon travail et cette expression apparaît d'ailleurs dans le titre de ce mémoire. Il s'agit de prendre un théorème, et de minimiser les concepts mathématiques nécessaires pour l'exprimer. Ce faisant, nous l'obtenons dans une théorie plus petite.

Dans notre cas, nous regardons non seulement le théorème, mais aussi sa preuve. Et s'ils ne nécessitent pas par exemple le principe du tiers exclu, alors ce dernier être retiré de la théorie dans laquelle on exprime notre théorème !

D'un point de vue théorique, connaître la théorie minimale dans laquelle on peut exprimer un théorème est très intéressant. Dans le cadre du partage de preuves, nous cherchons à connaître la théorie minimale pour pouvoir ensuite exporter la preuve facilement dans d'autres systèmes.

En effet, on voudrait se passer des fonctionnalités qui ne sont pas présentes dans certains systèmes (par exemple les types dépendants). En partant d'une preuve en Coq, il nous faudra donc également chercher à supprimer ce qui touche aux univers et aux types inductifs.

Attention —
Une théorie peut ne pas disposer d'une fonctionnalité, mais l'utiliser au niveau basique de ses axiomes. Par exemple, le quantificateur universel prend en paramètre un type T et un prédicat de T, donc semble nécessiter du polymorphisme. Mais ce n'est pas le cas, nous pouvons tout à fait autoriser le quantificateur universel dans une théorie tout en n'autorisant pas le polymorphisme.
En fait, nous pouvons nous dire que le quantificateur universel est défini par un *schéma d'axiomes* s'appliquant à un type, ce schéma étant défini dans la méta-théorie et pas dans la théorie.

1.4 Dedukti

Le partage de preuves est d'autant plus compliqué que le nombre de systèmes de preuve augmente. En effet, il faut un traducteur de Coq à Lean, de Lean à Coq, mais aussi de Coq à PVS, de PVS à Coq, etc. Le nombre de traducteurs est quadratique !

En fait, le manque de standards se fait fortement ressentir ; il manque une plateforme de partage, un système autour duquel le partage de preuves s'articulerait. L'outil Dedukti développé par Deducteam se propose pour cela.

Dedukti est une implémentation du $\lambda\Pi$-calcul Modulo Réécriture, noté $\lambda\Pi/\equiv$, c'est-à-dire du λ-calcul avec des types dépendants et des règles de réécritures (qui doivent être bien typées, ce qui est vérifié par Dedukti). Par exemple, on définit les entiers naturels et l'addition ainsi.

```
Nat: Type.
zero: Nat.
succ: Nat -> Nat.
def plus: Nat -> Nat -> Nat.

[ n ] plus zero n --> n
[ n ] plus n zero --> n
[ n, m ] plus (succ n) m --> succ (plus n m)
[ n, m ] plus n (succ m) --> succ (plus n m).
```

On définit quatre symboles `Nat`, `zero`, `succ` et `plus`, et des règles de réécriture qui définissent `plus`.

Notons que Dedukti n'est pas un assistant de preuve. Il s'agit d'un *framework logique* dans lequel nous allons *encoder des théories*. L'idée du partage de preuve d'un système S_1 de théorie T_1 vers un système S_2 de théorie T_2 est la suivante.

1. Nous encodons T_1 et T_2 dans Dedukti (notons les encodages $\texttt{Dedukti}[T_i]$).
2. Nous traduisons la preuve de S_1 dans $\texttt{Dedukti}[T_1]$.
3. Nous transformons cette traduction pour qu'elle soit dans $\texttt{Dedukti}[T_2]$.
4. Nous traduisons la preuve de $\texttt{Dedukti}[T_2]$ dans S_2.

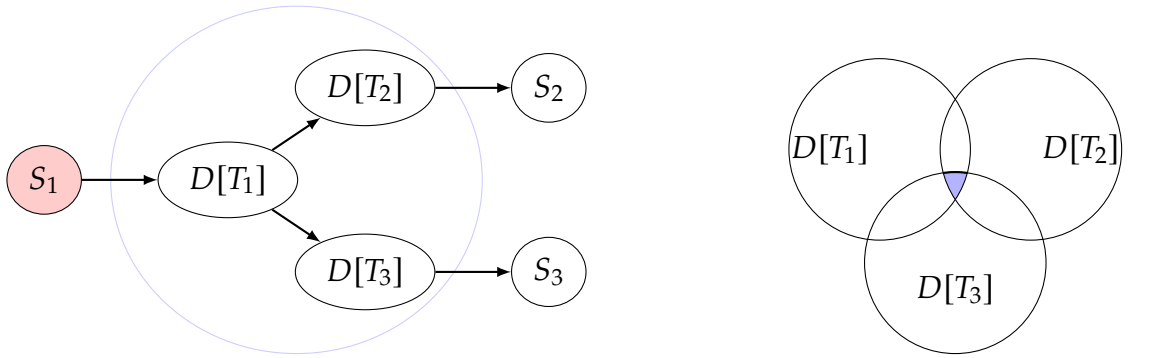
Les étapes compliquées sont la première (encoder les théories) et la troisième (passer d'un encodage à un autre).

Certains points de la première étape ont déjà été résolus. En effet, [4] montre que le Calcul des Constructions et plus généralement tous les *Pure Type System*) peuvent être encodés en $\lambda\Pi/\equiv$.

Les mathématiques inversées trouvent leur intérêt, vu ce que nous voulons faire. Après avoir traduit de T_1 vers $\texttt{Dedukti}[T_1]$, nous aimerions supprimer certaines fonctionnalités pour pouvoir ensuite passer facilement à $\texttt{Dedukti}[T_2]$.

De plus, pour faciliter le passage de $\texttt{Dedukti}[T_1]$ à $\texttt{Dedukti}[T_2]$, ce serait bien que les axiomes communs à T_1 et T_2 soient représentés de la même manière dans les deux encodages.

L'idée de la traduction est schématisée en figure 2, avec en figure 2b l'illustration d'encodages où les axiomes communs à plusieurs théories sont représentées de la même manière ; dans ce cas, notre but sera de transformer une preuve dans $\texttt{Dedukti}[T_1]$ en une preuve dans un sous-ensemble commun à l'encodage de plusieurs théories. L'idéal est d'arriver dans le sous-ensemble commun aux théories (en bleu sur la figure), d'où l'intérêt des mathématiques inversées.



(a) Le partage de preuve.

(b) La forme d'encodage voulue.

FIGURE 2 – Dedukti, plateforme d'échange.

QUESTION —
Mais que veut-on dire par « encodage d'une théorie » ?

Nous allons définir des symboles et des règles de réécriture qui permettront d'écrire des termes du système à encoder. Par exemple, voici un encodage des propositions, des preuves et de l'implication (*cf.*section 1.1 pour l'explication de la règle de réécriture).

```
Prop: Type.
imp: Prop -> Prop -> Prop.
Prf: Prop -> Type.

[p, q] Prf (imp p q) --> Prf p -> Prf q.
```

Listing 1 – Exemple d'encodage.

1.5 La théorie U

Nous avons dit que si une fonctionnalité est présente dans T_1 et T_2, alors ce serait bien que cette fonctionnalité soit encodée de la même manière dans `Dedukti`$[T_1]$ et dans `Dedukti`$[T_2]$.

La théorie U, introduite dans [1], est un très bon candidat pour cela. En effet, elle permet de représenter le Calcul des Constructions, et l'encodage présenté dans l'article a la bonne propriété de permettre de représenter facilement les sous-théories du Calcul des Constructions !

Pour cela, elle se base sur 38 axiomes (un axiome est un symbole de Dedukti) qu'on peut combiner pour obtenir plusieurs propriétés. Par exemple, on obtient la logique des prédicats minimale en ajoutant aux symboles et règles définis en liste 1 ceux qui suivent.

```
Set: Type.
El: Set -> Type.
forall : x : Set -> (((El x) -> Prop) -> Prop).

[x,p] Prf (forall x p) --> z : El x -> (Prf (p z)).
```

Avec `El nat`, on aurait l'ensemble des éléments de `nat`, et $\forall$ prend en paramètre un terme de type `Set` et une fonction de ce `Set` vers `Prop`, ce qui nous donne une nouvelle proposition. Par exemple, nous pouvons écrire la proposition $\forall n \in \mathbb{N}, n > 0$ comme suit (le symbole => est la λ-abstraction de Dedukti).

```
forall nat (n: nat => positive n)
```

Cela demande d'avoir défini le prédicat `positive` et un terme `nat` de type `Set`.

Attention (Une question d'encodages) —
Dans Dedukti, nous faisons bien des *encodages*. Il serait possible d'avoir deux encodages différents d'une même théorie. La théorie U se présente comme une théorie « universelle » pour Dedukti.

Par exemple, la théorie des types simples STT est une sous-théorie du Calcul des Constructions et est présente dans la théorie U en prenant les axiomes correspondants à STT. En fait, si nous avons une preuve dans une sous-théorie T de CoC (que ce soit STT, CoC ou la logique des prédicats), nous pouvons l'exprimer dans la théorie U en utilisant les axiomes de la théorie U nécessaires pour définir la théorie T.

1.6 Logipedia, un premier pas sur la Terre commune

François THIRÉ a déjà fait du partage de preuves. Pour cela, il a introduit une logique, STT∀, correspondant à la théorie des types simples avec du polymorphisme et des opérateurs de types. Logipedia, un programme permettant d'exporter des preuves de `Dedukti`[STT∀] vers plusieurs systèmes de preuves (Coq, HOL Light, Lean, Matita, Open Theory et PVS) a été écrit, ce qui a permis la traduction d'une bibliothèque d'arithmétique contenant une preuve du petit théorème de FERMAT (*cf.* [7, 8]).

Ce travail n'est pas basé sur la théorie U, mais montre que le partage de preuves peut bien se faire !

2 GeoCoq, de Coq aux autres systèmes

Mon travail était de partager GeoCoq [1], une bibliothèque Coq de géométrie euclidienne, et d'ainsi obtenir une formalisation du Livre I des Éléments d'Euclide (une partie de GeoCoq) dans d'autres systèmes de preuves.

Nous partons donc d'une preuve Coq, donc dans le Calcul des Constructions avec univers et types inductifs, et nous voulons l'encoder en Dedukti, la transformer pour l'avoir dans une logique très simple, si possible dans la théorie U, et l'exporter vers d'autres systèmes.

1. https://geocoq.github.io/GeoCoq/

Il semble raisonnable de se dire que si une preuve n'utilise que des opérations sur les entiers et ne quantifie jamais sur des univers, alors il n'y a aucune raison que des univers tels que $\texttt{Type}_3$ ou encore du polymorphisme y apparaissent. De même, le livre I des Éléments d'`Euclide` devrait pouvoir se faire sans ces concepts et est donc un bon test pour le partage de preuve.

2.1 Vodk, de Coq à Dedukti

Beaucoup de personnes ont travaillé sur un encodage de Coq en Dedukti, et un traducteur de Coq vers Dedukti, Vodk [2], a été écrit. Dans [2], Mathieu Boespflug et Guillaume Burel donnent notamment un encodage des types inductifs et Gaspard Férey montre comment gérer la les univers et leur cumulativité dans [5]. Je donne rapidement ici quelques points qui montrent intuitivement comment encoder les univers.

Le gros problème est qu'en Coq, un objet a plusieurs types (s'il est dans l'univers $\texttt{Type}_i$, alors il est dans l'univers $\texttt{Type}_j$ pour tous $j > i$), alors qu'en Dedukti, un objet a un seul type. Ainsi, si une fonction attend un élément de $\texttt{Type}_j$ et que nous voulons lui passer un élément de $\texttt{Type}_i$ avec $j > i$, ce sera possible en Coq et nous voulons rendre cela possible en Dedukti.

L'idée est d'avoir un opérateur `cast`, permettant de « traverser les univers ». Si t a le type $\texttt{Type}_i$, alors `cast` permet de construire un objet de type $\texttt{Type}_j$ pour tout j supérieur à i ; on a promu t !

Pour que cela reste correct, le `cast` demande une preuve de sous-typage (donc que j est bien supérieur à i) et cette preuve est faite à l'aide de règles de réécritures (on sait que t a le type $\texttt{Type}_i$ et qu'on veut le transformer en $\texttt{Type}_j$, donc on a accès à i et à j).

Donc, Vodk permet d'obtenir une preuve de Coq dans un encodage `Dedukti`[Coq] où est notamment présent cet opérateur `cast`.

2.2 Au contact de Vodk

En utilisant Vodk, j'ai pu traduire GeoCoq dans l'encodage `Dedukti`[Coq] de Gaspard Férey et commencer mon travail de minimisation qui se déroule en plusieurs étapes.

1. Supprimer les casts.
2. Utiliser les connecteurs de la théorie U.
3. Supprimer des égalités entre propositions.

2. https://github.com/Deducteam/CoqInE

Je me suis particulièrement servi de DKMeta [3], un outil de réécriture de termes Dedukti. Il permet, étant donné des règles de réécriture (appelées règles de méta-réécriture) et des fichiers Dedukti, de réécrire les fichiers Dedukti en appliquant les règles de méta-réécriture. Une première simplification (qui n'est pas dans les étapes décrites plus haut) s'est faite au niveau des fichiers Coq. En effet, les axiomes d'Euclide étaient définis dans une classe qu'il fallait traduire, ce qui complexifiait la traduction obtenue.

Placer les axiomes en dehors de la classe et la supprimer nous offrait déjà une certaine simplification : dans la traduction de la version avec classe, chaque fonction prenait en paramètre chaque élément de la classe, alors que ce n'était pas nécessaire. Par exemple, on passe de quarante à trois lignes pour la définition de l'égalité entre deux points.

```
def eqpoint :
  x:(C.T C.Type Point) -> y:(C.T C.Type Point) -> C.U prop
:= Coq_Logic.eq Point.
```

Ici, `C.U univers` encode l'univers `univers` et `C.T` correspond à l'encodage des termes, donc `C.T C.Type Point` correspond au types des points.

Mon travail est disponible sur mon Github [4]

2.3 Simplification des univers

La première chose que l'on remarque, c'est l'omniprésence de `cast`. La première étape de mon travail a donc été de chercher à les éliminer, les Éléments d'`Euclide` ne semblant pas nécessiter tous ces univers.

QUESTION —
Pourquoi ces casts apparaissent-ils ?

La plupart (quasiment tous) ces casts apparaissent parce que lors de la traduction (donc dans Vodk), on fait un cast dès qu'une fonction demande un élément d'un univers particulier. Mais on se rend compte que la majorité des `cast` sont inutiles et vont d'un univers au même univers. On a alors trois types de casts.

3. https://github.com/Deducteam/dkmeta
4. https://github.com/Karnaj/DGCE

2.3.1 Des casts triviaux

```
C.cast C.Type1 C.Type1 C.u0 C.u0 C.I C.prop
C.cast C.prop C.prop prop_ex prop_ex C.I prop_ex_proof
```

Il s'agit des `cast` qui vont d'un univers dans le même univers. Ici, le second cast par exemple nous transforme `prop_ex_proof` qui est un élément de `prop_ex` (comme dit en section 1.1, une proposition est un type dont les éléments sont ses preuves). Avec DKMeta, j'ai pu réécrire ces casts (en `prop_ex_proof` dans le deuxième cas).

2.3.2 Des casts avec le même type « caché »

Je me suis alors rendu compte que `cast` permettait de transformer des preuves d'une proposition P en preuve de P', mais où P et P' sont la même expression. Voici la forme générale de ces casts et un exemple.

```
C.cast C.prop C.prop prop_ex1 prop_ex2 C.I prop_ex1_proof

C.cast (C.rule C.prop C.prop) C.prop
       (C.prod C.prop C.prop (C.rule C.prop C.prop)
               C.I P (H: P => Coq__Init__Logic.False))
       (Coq__Init__Logic.not P)
       C.I
       proof
```

Ce type de cast est souvent présent avec comme propositions une fonction ,d'un côté, et sa définition de l'autre : ici, notre exemple permet de transformer une preuve de `P => False` (la définition de `not P`) en une preuve de `not P`.

Cet exemple me permet d'introduire `C.prod`, l'encodage du produit de Coq. Il prend en paramètre trois univers (celui de l'élément d'entrée, celui de l'élément d'arrivé, et celui de l'abstraction ainsi formée), une preuve qu'on peut bien construire un produit de cette manière, un élément de l'univers de départ et la fonction correspondant). L'univers d'un produit est « calculé » grâce à `C.rule` et de la réécriture : une règle permet de réécrire `C.rule s1 s2` en l'univers des abstractions de `s1` dans `s2`.

Par exemple, une règle réécrit `C.rule C.prop C.prop`en `C.prop`, c'est-à-dire qu'ici, on a une fonction de `C.prop` vers `C.prop` et qu'une telle fonction est un `C.prop`. En fait une telle fonction correspond à une implication, et on est bien en train de caster une preuve d'implication.

J'ai utilisé DKMeta pour simplifier les `rule`, puis pour supprimer ces `cast`.

INFORMATION —
Cet exemple nous montre que Vodk traduit les implications comme des produits et que lorsqu'on veut se servir d'une implication comme d'une proposition, on fait un cast inutile qui nous dit qu'un produit de `Prop` vers `Prop` peut être « converti » en `Prop`, c'est-à-dire qu'une implication est une proposition.
Notons alors que nous voulons également supprimer le plus de produits possibles et ne garder que les implications !

D'autres casts similaires étaient présents, par exemple pour transformer une implication (sous forme de produit) en proposition et pouvaient être supprimés car une preuve de `P => Q` *est* une fonction (dans le contexte où ces casts apparaissaient, c'est la preuve de la proposition et pas la proposition qui était utilisé). En fait, même le produit `prod` pouvait alors être supprimé !

2.3.3 Des problèmes de principe inductif

Certains `cast` étaient dûs aux principes inductifs de l'égalité et de certains connecteurs. Par exemple, pour l'égalité, nous avons `eq_rect` et `eq_ind`.

```
def eq__rect :
A:(C.U Type) -> x:(C.T Type A) ->
P:(__:(C.T Type A) -> C.U Type) ->
f:(C.T C.Type (P x)) -> y:(C.T C.Type A) ->
e:(C.T C.prop (Coq_Logic.eq A x y)) -> C.T C.Type (P y) := ...

def eq__ind :
A:(C.U Type) -> x:(C.T Type A) ->
P:(__:(C.T Type A) -> C.U C.prop) ->
f:(C.T C.prop (P x)) -> y:(C.T Type A) ->
e:(C.T C.prop (Coq_Logic.eq A x y)) -> C.T C.prop (P y) := ...
```

On donne un type A, un élément x de ce type, un prédicat P, une preuve de $P(x)$, un élément y de A, une preuve que $x = y$, et on obtient une preuve de $P(y)$. La différence entre les deux : `eq_rect` attend un prédicat qui renvoie un `Type`, `eq_ind` attend un « vrai » prédicat, donc qui renvoie une proposition.

On a de même `and__ind` et `and__rect`, etc. Seules les versions en `ind` sont nécessaires pour ce que l'on veut faire. Et pourtant, il se trouve que beaucoup de versions avec `rect` apparaissent dans nos preuves... Avec des `cast` pour transformer les prédicats (qui renvoient toujours des propositions), en des prédicats qui renvoient des `Type`.

Encore une fois, on peut simplifier cela et faire disparaître toutes les versions avec `rect` à l'aide de règles de méta-réécriture.

Finalement, à cette étape, on se rend compte que les seuls types dont on a besoin pour exprimer nos preuves sont les propositions, les points, et les cercles ! Puisqu'aucun autre type n'apparaît, on peut se contenter de deux univers U_0 et U_1. L'univers U_1 est celui des types (on retrouve `Prop`, `Circle` et `Point` dans cet univers), et U_0 correspond à l'univers des propositions `Prop`.

2.4 Vers la théorie U ?

Seul restait un dernier petit bastion d'univers et je pourrais alors essayer d'avoir les preuves dans l'encodage de la théorie U ! Ce dernier bastion était lié à la traduction des fichiers de Coq liés à la logique (dont tout ce qui définissait les connecteurs et l'égalité).

Il s'agissait d'une petite trentaine de définitions (les connecteurs, leurs règles d'élimination et d'introduction), etc. Et pour supprimer ces derniers restes d'univers, et se rapprocher de la théorie U, j'ai décidé de tout simplement les remplacer par les connecteurs de la théorie U !

Le point important ici est que ce qui nous intéresse avec un connecteur, ce n'est pas la manière dont il est défini, mais comment on en construit une preuve et comment on utilise cette preuve ! Dans la traduction de Vodk, le connecteur and était défini ainsi.

```
and: A: C.U C.prop -> B: C.U C.prop -> C.U C.prop.

conj: A: C.U C.prop -> B: C.U C.prop ->
      H1: C.T C.prop A -> H2: C.T C.prop B ->
      C.T C.prop (and A B).
```

Et on l'utilisait à travers ces deux fonctions.

```
def match____and:
  s: C.S -> A: C.U C.prop -> B: C.U C.prop ->
  P: (C.T C.prop (and A B) -> C.U s) ->
  case__conj: (H1: C.T C.prop A -> H2: C.T C.prop B ->
    C.T s (P (conj A B H1 H2))) ->
  H: C.T C.prop (and A B) -> C.T s (P H).

def and__ind:
  A: C.U C.prop -> B: C.U C.prop -> P: C.U C.prop ->
  f: (H1: C.T C.prop A -> H2: C.T C.prop B -> C.T C.prop P) ->
  H: C.T C.prop (and A B) -> C.T C.prop P := ...
```

`and__ind` est construite à partir de `match____and`, mais les deux nous permettent d'utiliser une preuve de `and A B`. Si on a une fonction qui à partir d'une preuve de `and A B` nous donne un élément dans l'univers `s` et qu'on a une preuve de `and A B`, alors on saura obtenir cet élément de `s`. La version `and__ind` est plus restrictive ; elle ne permet de montrer que des propositions.

En fait, seul `and__ind` est nécessaire ; on ne veut montrer que des propositions. Cela signifie que pour supprimer le `and` traduit par `vodk`, il nous suffit de savoir exprimer `and__ind` et `conj` à partir du `and` de la théorie U, ce qui est fait ci-dessous !

```
and: Prop -> Prop -> Prop.

[p2, p1] Prf (and p1 p2) -->
p:Prop -> ((Prf p1) -> (Prf p2) -> Prf p) -> Prf p.

def conj:
  A:Prop -> B:Prop -> H1:(Prf A) -> H2:(Prf B) ->
  Prf (and A B) :=
  A:Prop => B:Prop => H1:(Prf A) => H2:(Prf B) =>
  p:Prop => f:((Prf A) -> (Prf B) -> Prf p) => f H1 H2.

def and__ind:
  A:Prop -> B:Prop -> P:Prop ->
  f:(H1:(Prf A) -> H2:(Prf B) -> Prf P) ->
  H:(Prf (and A B)) ->
  Prf P :=
  A:Prop => B:Prop => P:Prop =>
  f:(H1:(Prf A) -> H2:(Prf B) -> Prf P) =>
  H:(Prf (and A B)) =>
  H P f.
```

On utilise le fait qu'une preuve de $A \wedge B$ est, par la règle de réécriture donnée, une fonction qui, étant donné une proposition P et une preuve de $A \longrightarrow B \longrightarrow P$, construit une preuve de P.

Information —
Les preuves du Livre I des Éléments d'Euclide n'utilisaient pas de type inductif autre que celui des booléens (que nous venons de remplacer en utilisant les éléments de logique de la théorie U).

Notons que j'ai profité de cette étape de ma traduction pour transformer mes preuves en utilisant les notations de la théorie U. En particulier, j'ai totalement supprimé tout

ce qui me restait comme référence aux univers.

— `C.U C.prop` devient `C.Prop` (l'univers des proposition).
— `C.T C.prop P` devient `C.Prf P` (un terme de `P` est une preuve de `P`).
— `C.T C._0 Circle` devient `C.El Circle`.

Et ça y est, nous n'avons plus aucun univers dans notre traduction ! Nous sommes donc, au pire des cas, dans le calcul des constructions, donc dans la théorie U !

À cette étape de la transformation, il me reste quelques produits et certains d'entre eux correspondent à des des implications ; je les ai alors remplacé par l'implication de la théorie U.

Ces produits apparaissaient par exemple avec le principe inductif de l'égalité qui prend en paramètre un prédicat qui peut être de la forme `X: Point => C.prod P Q`. Dans ce contexte, on veut un prédicat de `Point` vers `Prop` (et ce n'est pas la preuve de l'implication qui nous intéresse, mais bien l'implication), et donc nous ne pouvons pas supprimer le produit comme nous l'avons fait précédemment.

Ça y est, nous sommes, à quelques détails près dans l'encodage présenté de la théorie U. Le plus gros détail se situe au niveau des types des fonctions. Voici à quoi ressemblaient les types de mes objets à ce moment.

```
thm lemma__congruencesymmetric :
  A:(C.El Point) -> B:(C.El Point) ->
  C:(C.El Point) -> D:(C.El Point) ->
  H:(C.Prf (Axioms__euclidean__axioms.Cong B C A D)) ->
  C.Prf (Axioms__euclidean__axioms.Cong A D B C) := ...
```

Dans la théorie U, on voudrait plutôt ce genre de type.

```
thm lemma__congruencesymmetric :
C.Prf (
  C.forall Point (A:(C.El Point) =>
    C.forall Point (B:(C.El Point) =>
      C.forall Point (C:(C.El Point) =>
        C.forall Point (D:(C.El Point) =>
          C.imp (Axioms__euclidean__axioms.Cong B C A D)
                (Axioms__euclidean__axioms.Cong A D B C))))))) := ...
```

La notation précédente est moins verbeuse, mais elle a le désavantage de ne pas être dans l'encodage. Avec cette dernière notation, nous sommes certains d'être bien dans l'encodage !

Nous voudrions faire ce remplacement avec DKMeta en utilisant une règle de ce genre.

```
[T, Q] prod.prod (C.El T) (x => C.Prf (Q x)) -->
C.Prf (C.forall T (x: C.El T => (Q x))).
```

Qui permettrait de transformer `A:(C.El Point) -> C.Prf True.` en ce type.

```
C.Prf (C.forall Point (x: C.El Point => True)).
```

Mais en fait, on obtient le type suivant.

```
C.Prf (C.forall Point (x: C.El Point => (x: C.El Point => True) x).
```

C'est le même type, mais qui n'est pas bêta-réduit. En fait, DKMeta fait la bêta-réduction par défaut, mais si elle le fait, elle le fait pour tous les termes, ce que je ne voulais car cela faisait beaucoup augmenter la taille de certains termes et le temps qu'ils mettaient à être vérifiés.

Je ne pouvais donc pas utiliser DKMeta pour transformer mes types et j'ai finalement choisi de garder la première forme de type.

2.5 En logique des prédicats ?

Reste à savoir dans quel fragment de la théorie U sont les preuves à cette étape. Elles semblent être en théorie des types simples voire même en logique des prédicats, mais quelques petits points réfutent cette hypothèse.

En effet, les preuves contiennent des égalités de propositions, ce qu'il n'y a pas en logique des prédicats. Ce problème n'est pas, en théorie, compliqué à régler. Une égalité de deux propositions, c'est une équivalence ! Néanmoins, on peut se demander pourquoi de telles égalités sont présentes. J'ai conjecturé que c'est dû à certaines tactiques de Coq qui font qu'on se retrouve dans la situation suivante. On veut montrer que `P(c)`, sachant que `P(d)` est vraie.

— `f` renvoie une preuve que `P(d) = P(c)`. Pour cela, elle utilise une preuve que `d = c` construite par `h`.
— `g` utilise la preuve que `P(d) = P(c)` avec `eq__ind` pour obtenir une preuve de `P(c)`.

On se rend bien compte que c'est absurde et qu'on peut directement construire une preuve de `P(c)` si on sait construire une preuve de `d = c` et qu'on a une preuve de `P(d)`. Je transforme alors la preuve en faisant prendre en paramètre à `f` une preuve de `P(d)` et renvoyer directement une preuve de `P(c)`.

Pour automatiser cette transformation, j'ai écrit un outil qui détecte les fonctions `f` qui renvoient des preuves de `P = Q`, je les modifie pour qu'elles prennent en paramètre

une preuve de `P` et renvoient une preuve de `Q`, et il faut ensuite chercher les appels à cette fonction et les modifier. J'ai écrit un outil qui se charge de cela. Dans le cas de GeoCoq, il les remplace tous, mais en théorie, il me faudrait gérer plus de cas pour qu'il soit « universel », ce que je n'ai pas encore fait.

De plus, les preuves utilisent du polymorphisme (et donc on n'est même pas en théorie des types simples). En effet, l'égalité ne fait pas partie de la théorie U, et pour la définir , nous utilisons du polymorphisme (nous voulons une égalité pour les points, les cercles, etc.). Ce point s'arrange de deux manières.

— Créer deux égalités `eqPoint` et `eqCircle` et dupliquer chaque fonction de l'égalité pour avoir `eqPoint__ind` et `eqCircle__ind` (on instancie directement chaque fonction liés à l'égalité avec le type voulu).
— Rajouter l'égalité (polymorphe) à la théorie U (et avoir une théorie $U_=$).

J'ai choisi la deuxième option ; il ne semble pas aberrant de considérer l'égalité comme un objet basique de la théorie [5].

2.6 De la simplification

J'en ai également profité pour simplifier quelques preuves. Ces simplifications ne produisaient pas des termes dans une théorie plus petite, mais elles diminuaient leur taille, ce qui est toujours appréciable. Par exemple, des `eq_trans` inutiles étaient utilisées (pour prouver $A = B$, on montre $A = A$ et $A = B$) ou encore pour montrer $P(x)$ sachant que $P(y)$ est vrai et que $x = y$, on utilise `eq_ind` avec le prédicat

$$z \mapsto H_1 \implies \cdots \implies H_n \implies P(z) \implies H_{n+1} \implies \cdots \implies H_m$$

au lieu d'utiliser le prédicat $z \mapsto P(z)$. J'ai écrit un outil qui fait ces simplifications.

Une autre simplification que je n'ai pas totalement faite consiste à supprimer les arguments inutiles de certaines fonctions (des hypothèses inutiles ou encore des points et des cercles non utilisés dans le corps de la fonction). En effet, il n'est pas rare de voir dans le code des fonctions avec des dizaines, voire des centaines d'arguments (une simplification de ce type a permis d'enlever 3000 lignes à plusieurs fonctions).

Attention —
Bien sûr, je ne touche pas aux propositions et aux théorèmes du livre d'Euclide. Sur ce point je ne modifie que les lemmes introduits par Vodk lors de la traduction.

---

5. Depuis, l'égalité a été rajoutée à la théorie U

Néanmoins, ce n'est pas si simple à faire. Il faut par exemple se demander comment gérer les cas des fonctions partielles ou encore les $x \mapsto g(x,c)$ où $g(x,c) = f(a,x,c,d)$ et où $f(a,b,c,d)$ n'utilise pas b.

2.7 Vers STT∀ et vers les autres systèmes

Pour obtenir les preuves dans les autres systèmes, j'ai décidé de passer par STT∀ et Logipedia. J'avais ma preuve en logique des prédicats, donc il ne devait pas y avoir de gros problèmes pour utiliser STT∀ qui est, rappelons-le, de la théorie des types simples avec polymorphisme et opérateurs de types (et donc une extension de la logique des prédicats). En fait, il y a eu deux soucis majeurs.

2.7.1 Les connecteurs

Dans STT∀, les seuls connecteurs sont l'implication et le quatificateur universel. J'ai donc dû définir les autres connecteurs dans STT∀. Le problème est que, puisque je dois rester dans STT∀, je dois passer d'un symbole et de règles de réécriture à une définition.

Pour ce point, il faut garder en tête que ce qui nous intéresse, c'est comment se comporte une preuve de la proposition $A \wedge B$ par exemple, comportement donné par la règle de réécriture dans la théorie U. Ceci permet de savoir comment définir le connecteur dans STT∀.

```
(; dans la théorie U ;) [p1,p2] Prf (and p1 p2) -->
p: Prop -> (Prf p1 -> Prf p2 -> Prf p) -> Prf p.

(; dans STTFA ;) def and :
etap (p arrow bool (arrow bool bool))
:=
  x:(etap (p bool)) =>
  y :(etap (p bool)) =>
  forall bool
(z: (etap (p bool)) =>
    impl (impl x (impl y z)) z).
```

Ici, `etap (p type_ex)` correspond à un élément du type `type_ex`. On définit $x \wedge y$ comme la fonction qui à x et y associe la proposition $\forall y, (x \implies y \implies z) \implies z$ et le fait qu'une preuve de $P \implies Q$ soit une fonction des preuves de P dans les preuves de Q assure qu'une preuve de $x \wedge y$ dans STT∀ se comporte de la même manière qu'une preuve de $x \wedge y$ dans la théorie U.

2.7.2 Être *vraiment* dans STT∀

Logipedia ne traduit que des preuves qui sont *vraiment* dans son encodage de STT∀. On retombe alors sur le problème de types soulevé en section 2.4.

```
eta Point -> eta bool : KO.
eta arrow Point bool  : OK.
```

J'ai écrit un outil, basé sur des expressions régulières, pour faire la transformation. Ici, je ne travaille donc pas sur l'AST des preuves , mais sur le texte.

Et j'ai enfin pu traduire les fichiers vers les autres systèmes de preuve. Je donne dans le tableau tableau 1 les temps de traduction et les tailles des preuves générées. Pour comparaison, la première traduction en Dedukti faisait environ 500 Mo, celle obtenue en logique des prédicats environ 250 Mo, et celle en STT∀environ 300 Mo.

	Coq	HOL Light	Lean	Matita	Open Theory	PVS
Taille	100 Mo	250 Mo	150 Mo	100 Mo	1.9 Go	1.3 Go
Durée	13h50	14h10	13h20	12h10	14h40	15h20h

TABLE 1 – *Benchmark* des différentes traductions.

Conclusion et travaux futurs

Ce stage a été très enrichissant. Il a mené au partage d'une bibliothèque de preuves assez conséquente, ce qui en fait une très bonne *Proof Of Concept*, d'autant plus que les différentes étapes sont reproductibles avec d'autres preuves. Néanmoins, certaines traductions obtenues sont très grosses et je pense que certaines simplifications de preuves permettront de réduire considérablement la taille de certains termes.

Dans le futur, on voudrait la traduire vers d'autres systèmes, comme Isabelle. De plus, on remarque que nous avons traduit les preuves en logique des prédicats, pour ensuite l'obtenir dans STT∀qui en est une extension. On peut alors s'intéresser à des systèmes comme Mizar, basé sur la logique des prédicats.

Finalement, on voudrait s'intéresser à des fonctionnalités supplémentaires (par exemple, essayer de traduire des types dépendants à l'aide de types et de prédicats) et à l'*alignement des concepts* : il s'agit de faire en sorte que les preuves traduites vers Coq, PVS, ou n'importe quel autre systèmes utilisent les concepts de ces systèmes. Pour le moment, nous traduisons en définissant par exemple de nouveaux connecteurs, alors que nous aimerions utiliser les connecteurs natifs du système cible.

Références

[1] Frédéric Blanqui et al. “Some Axioms for Mathematics”. In : *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*. Sous la dir. de Naoki Kobayashi. T. 195. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 20 :1-20 :19. isbn : 978-3-95977-191-7. doi : 10.4230/LIPIcs.FSCD.2021.20. url : https://drops.dagstuhl.de/opus/volltexte/2021/14258.

[2] Mathieu Boespflug et Guillaume Burel. “CoqInE : Translating the Calculus of Inductive Constructions into the λΠ-calculus Modulo”. In : *in "Second International Workshop on Proof Exchange for Theorem Proving*. 2012.

[3] Thierry Coquand. “An Analysis of Girard’s Paradox”. In : *In Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1986, p. 227-236.

[4] Denis Cousineau et Gilles Dowek. “Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo”. In : juin 2007, p. 102-117. isbn : 978-3-540-73227-3. doi : 10.1007/978-3-540-73228-0_9.

[5] Gaspard Férey. “Higher-Order Confluence and Universe Embedding in the Logical Framework . (Confluence d’ordre supérieur et encodage d’univers dans le Logical Framework”. Thèse de doct. École normale supérieure Paris-Saclay, France, 2021. url : https://lmf.cnrs.fr/downloads/Perso/Ferey-thesis.pdf.

[6] J. Girard. “Interpretation fonctionelle et elimination des coupures dans l’aritmetique d’ordre superieur”. In : 1972.

[7] François Thiré. “Sharing a Library between Proof Assistants : Reaching out to the HOL Family”. In : *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages : Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018*. Sous la dir. de Frédéric Blanqui et Giselle Reis. T. 274. EPTCS. 2018, p. 57-71. doi : 10.4204/EPTCS.274.5. url : https://doi.org/10.4204/EPTCS.274.5.

[8] François Thiré. “Interoperability between proof systems using the logical framework Dedukti. (Interopérabilité entre systèmes de preuve en utilisant le cadre logique Dedukti)”. Thèse de doct. École normale supérieure Paris-Saclay, Cachan, France, 2020. url : https://tel.archives-ouvertes.fr/tel-03224039.