Modèles de la théorie des types donnés par traduction de programme

Simon Boulier, Pierre-Marie Pédrot, Nicolas Tabareau

Modèles de la théorie des types donnés par traduction de programme

Simon Boulier Pierre-Marie Pédrot Nicolas Tabareau

INRIA
prenom.nom@inria.fr

Résumé

Dans cet article, nous présentons quelques modèles du calcul des constructions avec univers (CC_ω) donnés par des traductions de programme. De tels modèles peuvent être vus comme la compilation d'une théorie des types compliquée vers une théorie des types plus simple. Nous les utilisons pour montrer simplement que certains axiomes ne sont pas dérivables dans CC_ω : l'extensionnalité fonctionnelle, l'extensionnalité propositionnelle et le fait que la bissimilarité implique l'égalité pour les *streams*. Ces modèles permettent également d'ajouter de nouveaux opérateurs dans la théorie source. Nous prendrons l'exemple du *pattern matching* sur un univers.

Les traductions de programme ont l'avantage sur les autres modèles de pouvoir être instrumentées sous forme de plug-ins d'un assistant de preuve (nous utiliserons Coq), ce qui permet d'ajouter de nouveaux axiomes à la volée, tout en préservant la cohérence de la théorie.

Introduction

« Peut-on prouver que $(\forall x.\ f\ x = g\ x) \to f = g$? » « Quand P et Q sont des propositions, quelle est la différence entre $P \leftrightarrow Q$ et $P = Q$? » « Peut-on prouver qu'un terme de $\forall A.\ A \to A$ est nécessairement l'identité ? »

Ces questions sont le pain quotidien de tout théoricien des types. Et en effet, elles ont rarement une réponse simple puisqu'elles nécessitent de montrer la consistance d'une théorie des types augmentée d'un axiome ou d'un principe logique. Traditionnellement, il existe deux principales voies pour montrer la consistance d'une théorie des types :

la voie syntaxique on montre que la théorie a de bonnes propriétés syntaxiques (la *subject reduction* et la terminaison forte notamment) et l'on en déduit la consistance,

la voie sémantique on construit un modèle de la théorie, par exemple dans une structure catégorique, et l'on montre que l'interprétation de l'absurde est vide.

Dans la voie sémantique, il existe, à notre connaissance, deux principales classes de modèles : les *modèles ensemblistes* qui reposent sur la théorie des ensembles (par exemple le modèle simplicial de Voevodsky [6]) et les *modèles syntaxiques* qui reposent sur une autre théorie des types (par exemple les modèles setoid de Hofmann [4] et Altenkirch [1]).

Dans cet article, nous proposons de mettre en lumière une classe particulière de modèles syntaxiques : les modèles syntaxiques obtenus par *traduction de programme*. Une traduction de programme est un modèle syntaxique qui respecte les structures de théorie des types. Un terme est ainsi traduit par un terme, un contexte par un contexte, etc. Mais surtout, la conversion est interprétée par la conversion et le calcul est donc préservé. Une traduction de programme n'est donc rien de moins que la compilation d'une théorie des types $\mathcal{S}$ vers une autre, plus simple, $\mathcal{T}$. Et elle permet de justifier la consistance de $\mathcal{S}$ à partir de celle de $\mathcal{T}$ à condition de traduire l'absurde par l'absurde.

Les traductions de programme ont plusieurs avantages sur la voie syntaxique et sur les autres modèles de la voie sémantique :

elles donnent des modèles simples Comme beaucoup de constructions sont interprétées de façon transparente (la conversion par la conversion, les fonctions par les fonctions, etc.), les traductions permettent de construire des premiers modèles à peu de frais. Ces modèles ne permettent par contre pas de justifier entièrement une théorie des types, mais plutôt d'y étudier l'addition de nouveaux principes.

elles n'utilisent que la théorie des types Comme elles n'utilisent que la théorie des types, elles permettent de rendre les modèles accessibles aux non-spécialistes de la théorie des ensembles ou des catégories (par exemple, un univers est interprété par un autre univers et non par un univers de Grothendieck). Il y a en outre un intérêt conceptuel : cela permet de se passer de la théorie des ensembles pour justifier la théorie des types considérée. À la place, on se repose sur une autre théorie des types déjà justifiée, et, ultimement, celle-ci l'est de façon syntaxique.

elles sont modulaires Les traductions se composent et s'étendent facilement, contrairement aux preuves de terminaison.

elles sont facilement implémentables Étant donné une traduction de $\mathcal{S}$ vers $\mathcal{T}$ et un assistant de preuve pour $\mathcal{T}$. On peut implémenter la traduction de manière effective et en déduire un assistant de preuve pour $\mathcal{S}$.

Les traductions de programme sont présentes depuis bien longtemps en théorie des types : elles puisent leurs racines dans les traductions logiques — traduction de Gödel, traduction CPS de Lafont-Streicher-Reus [7], etc. —; on en aperçoit des fragments dans de nombreux articles — par exemple le *subset model* de Hofmann [4] —; et, de façon encore plus explicite, dans des travaux récents sur le forcing [5] [8], Dialectica [10], ou encore la paramétricité [2]. Cependant, nous pensons qu'une présentation des traductions de programme en tant que telles (et non en tant que simples modèles) manque et qu'elle permettrait de populariser cette façon de construire des modèles. La présentation « traduction de programme » permet en outre de mettre en avant certaines bonnes propriétés, et notamment la préservation du calcul. Cet article se veut donc une première tentative de combler ce manque.

Nous nous focalisons sur des traductions du calcul des constructions avec univers CC_ω, c'est-à-dire une théorie des types avec des produits dépendants et une hiérarchie d'univers. Dans une première section, nous expliquons quel schéma suivent les traductions que nous considérons. Puis nous donnons des exemples de traductions de programme. Toutes utilisent le fait que les types négatifs (produits dépendants, types coinductifs, univers) sont sous-spécifiés en théorie des types intensionnelle, ce qui laisse de la liberté pour les rendre plus intensionnels. Ainsi, en section 2, nous présentons une traduction qui permet de nier l'extensionnalité fonctionnelle. Puis, à l'aide d'une technique similaire, nous décrivons des traductions qui permettent de montrer que la bissimilarité n'implique pas l'égalité pour les *streams* (on parlera d'extensionnalité des *streams*, section 3) et que l'extensionnalité propositionnelle n'est pas dérivable (section 4). Ensuite, nous montrons comment les traductions de programme permettent d'ajouter de nouveaux opérateurs en prenant l'exemple du *pattern matching* sur un univers (section 5). Une partie des résultats de ce papier ont été formalisés à l'aide de l'assistant de preuve Coq [12]. Nous décrivons notre formalisation dans la section 6. Enfin, nous montrons comment les traductions de programme du calcul des constructions peuvent être instrumentées sous forme d'un plug-in de Coq et proposons un prototype d'une telle implémentation pour les trois premières traductions de l'article.

Contributions Cet article propose les contributions suivantes :

— nous donnons une présentation des traductions de programme en théorie des types dépendants et montrons comment elles permettent d'intégrer de nouveaux principes logiques à une théorie,

— nous décrivons trois modèles qui nient l'extensionnalité fonctionnelle, l'extensionnalité des *streams* et l'extensionnalité propositionnelle,

— nous donnons un modèle permettant d'interpréter le *pattern matching* sur la forme normale d'un type,
— et enfin les trois premiers modèles sont formalisés en Coq et implémentés sous forme de plug-in de Coq.

Le code de la formalisation et des plug-ins accompagnant cet article sont disponible à l'adresse :

`https://github.com/CoqHott/Program-translations-CC-omega`

Cet article est la version alternative d'un article publié à *Certified Programs and Proofs (CPP) 2017*.

1. Cadre général

Les traductions de programme que nous allons présenter suivent toutes le même schéma. Étant donné une théorie source $\mathcal{S}$ et une théorie d'arrivée $\mathcal{T}$:

— un terme M de $\mathcal{S}$ est traduit vers un terme $[M]$ de $\mathcal{T}$,
— un type A de $\mathcal{S}$ est traduit vers un type $[\![A]\!]$ de $\mathcal{T}$,
— et un contexte Γ de $\mathcal{S}$ est traduit vers un contexte $[\![\Gamma]\!]$ de $\mathcal{T}$,
— où la traduction d'un type est donnée par une opération à spécifier ι :

$$[\![A]\!] := \iota\ [A]$$

— et les contextes sont traduits point à point :

$$[\![\,.\,]\!] := .$$
$$[\![\Gamma,\, x : A]\!] := [\![\Gamma]\!],\, x : [\![A]\!].$$

Les propriétés attendues d'une telle traduction sont alors :

préservation de la conversion si $M \equiv N$ alors $[M] \equiv [N]$,

préservation du typage si $\Gamma \vdash M : A$ alors $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$,

préservation de la consistance si $[\![\bot_{\mathcal{S}}]\!]$ est habité (dans le contexte vide) alors $\bot_{\mathcal{T}}$ l'est aussi, où $\bot_{\mathcal{S}}$ et $\bot_{\mathcal{T}}$ représentent l'absurde dans $\mathcal{S}$ et $\mathcal{T}$ (par exemple, $\bot = \Pi\, X : \square.\ X$).

Sous ces conditions, la consistance de $\mathcal{T}$ (le fait qu'il n'existe pas de terme clos habitant l'absurde) implique celle de $\mathcal{S}$. La théorie source $\mathcal{S}$ peut contenir plus de constructeurs de type ou de principes logiques que $\mathcal{T}$ du moment que ceux-ci ont une traduction dans $\mathcal{T}$. Dans la pratique, $\mathcal{S}$ est souvent de la forme $\mathcal{T}$ + axiome et la traduction assure que l'axiome peut être supposé sans briser la cohérence du système. Dans ce cas, la traduction triviale de $\mathcal{T}$ dans $\mathcal{T}$ + axiome (l'injection) donne l'équiconsistance entre les deux théories.

Remarque. Dans certaines traductions, on peut montrer la préservation de la réduction (si $M \to N$ alors $[M] \to^+ [N]$) ce qui donne la terminaison forte de $\mathcal{S}$ sous réserve de celle de $\mathcal{T}$.

Théories utilisées Le système de base que nous utiliserons dans cet article sera le calcul des constructions avec univers, CC_ω, c'est-à-dire une théorie des types avec seulement des fonctions dépendantes (Π-types) et une hiérarchie d'univers $\square_i$ (règles de typage en figure 1). Nous considérerons ensuite diverses extensions de CC_ω : un type égalité $=$, un univers imprédicatif $*$, des types Σ, et un type booléen $\mathbb{B}$ (règles de typage de chacune de ces extensions en figure 2). Nous considérerons aussi des types coinductifs en section 3 et des types inductifs-récursifs en section 5. On utilisera les notations suivantes : $\square$ pour désigner un univers quelconque ($\square_i$ ou $*$ en présence de l'univers imprédicatif), $\bot$ pour l'absurde $\Pi\, X : \square.\ X$ et les notations $\to$ et $\times$ pour les types produit et somme non dépendants.

Dans le cas des types Σ, nous requérons le *surjective pairing* ($(\pi_1\ M, \pi_2\ M) \equiv M$). Cela est nécessaire pour la traduction sur les *streams* (section 3).

$$A, B, M, N ::= \square_i \mid x \mid M\ N \mid \lambda x : A.\ M \mid \Pi x : A.\ B$$
$$\Gamma, \Delta ::= . \mid \Gamma, x : A$$

$$\frac{}{\vdash .} \qquad \frac{\Gamma \vdash A : \square}{\vdash \Gamma, x : A} \qquad \frac{\Gamma \vdash A : \square}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \square}{\Gamma, x : A \vdash M : B}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \square_i : \square_{i+1}} \qquad \frac{\Gamma \vdash A : \square_i \qquad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Pi x : A.\ B : \square_{\max(i\,j)}} \qquad \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\ B : \square}{\Gamma \vdash \lambda x : A.\ B : \Pi x : A.\ B}$$

$$\frac{\Gamma \vdash M : \Pi x : A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\{x := N\}} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \square \qquad A \equiv B}{\Gamma \vdash M : A}$$

$$(\lambda x : A.\ M)\ N \equiv M\{x := N\} \qquad \text{(règles de congruence omises)}$$

FIGURE 1 – Règles de typage de CC_ω

2. Nier l'extensionnalité fonctionnelle

La première traduction que nous considérons est une traduction qui nie l'extensionnalité fonctionnelle. Pour cela, une fonction est traduite par une paire d'une fonction et d'un booléen, et l'application de fonction « jette » le booléen. Le booléen ne peut donc pas être vu de manière externe car la seule façon d'observer une fonction est de l'appliquer. Le booléen sera toujours `true` en provenance de la théorie source.

Définition 1. *La traduction* $[\ ]_f$ *de* $\mathrm{CC}_\omega +=$ *vers* $\mathrm{CC}_\omega +=+\Sigma+\mathbb{B}$ *est définie par induction sur le terme par :*

$$\begin{aligned}
&[\square_i]_f && := \square_i \\
&[x]_f && := x \\
&[\Pi x : A.\ B]_f && := (\Pi x : [\![A]\!]_f.\ [\![B]\!]_f) \times \mathbb{B} \\
&[\lambda x : A.\ M]_f && := (\lambda x : [\![A]\!]_f.\ [M]_f, \mathtt{true}) \\
&[MN]_f && := \pi_1\ [M]_f\ [N]_f \\
&[M =_A N]_f && := [M]_f =_{[\![A]\!]_f} [N]_f \\
&[\mathtt{refl}\, M]_f && := \mathtt{refl}\, [M]_f \\
&[J(y.q.P, e, u)]_f && := J(y.q.[\![P]\!]_f, [e]_f, [u]_f) \\
&[\![A]\!]_f && := [A]_f
\end{aligned}$$

Dans cette traduction, $[\![A]\!]_f = [A]_f$ et les doubles crochets ne sont là que pour faire une distinction « morale » entre termes et types. Ceci n'est possible que parce que $[\square_i]_f = \square_i$.

On montre d'abord que la traduction vérifie un lemme de substitution :

Proposition 2. *Pour tous termes* M, N *et variable* x *on a :*

$$[M\{x := N\}]_f = [M]_f\{x := [N]_f\}$$

Grâce auquel on montre ensuite que la traduction préserve la conversion et le typage :

Univers Imprédicatif

$$A, B, M, N \quad ::= \quad \ldots \mid *$$

$$\frac{\vdash \Gamma}{\Gamma \vdash * : \Box_i} \qquad \frac{\Gamma \vdash A : \Box \qquad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi x : A.\, B : *}$$

Sommes Dépendantes

$$A, B, M, N \quad ::= \quad \ldots \mid \Sigma x : A.\, B \mid \pi_1\ M \mid \pi_2\ M \mid (M, N)$$

$$\frac{\Gamma \vdash A : \Box_i \qquad \Gamma, x : A \vdash B : \Box_j}{\Gamma \vdash \Sigma x : A.\, B : \Box_{\max(i,j)}} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B\{x := M\}}{\Gamma \vdash (M, N) : \Sigma x : A.\, B}$$

$$\frac{\Gamma \vdash M : \Sigma x : A.\, B}{\Gamma \vdash \pi_1\ M : A} \qquad \frac{\Gamma \vdash M : \Sigma x : A.\, B}{\Gamma \vdash \pi_2\ M : B\{x := \pi_1\ M\}}$$

$$\pi_1\ (M, N) \equiv M \quad \pi_2\ (M, N) \equiv N \qquad (\pi_1\ M, \pi_2\ M) \equiv M$$

Booléens

$$A, B, M, N ::= \ldots \mid \mathbb{B} \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ M\ \texttt{ret}\ P\ \texttt{then}\ N_1\ \texttt{else}\ N_2$$

$$\frac{\Gamma \vdash M : \mathbb{B} \qquad \Gamma \vdash P : \mathbb{B} \to \Box_i \qquad \Gamma \vdash N_1 : P\ \texttt{true} \qquad \Gamma \vdash N_2 : P\ \texttt{false}}{\Gamma \vdash \texttt{if}\ M\ \texttt{ret}\ P\ \texttt{then}\ N_1\ \texttt{else}\ N_2 : P\ M}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbb{B} : \Box_i} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \texttt{true} : \mathbb{B}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \texttt{false} : \mathbb{B}}$$

$$\texttt{if true ret}\ P\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \equiv N_1 \qquad \texttt{if false ret}\ P\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \equiv N_2$$

Égalité

$$A, B, M, N ::= \ldots \mid \texttt{refl}\, M \mid M =_A N \mid J(y.q.P,\, e\,,u)$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash A : \Box_i \qquad \Gamma \vdash M, N : A}{\Gamma \vdash M =_A N : \Box_i} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{refl}\, M : M =_A M}$$

$$\frac{\Gamma \vdash e : M =_A N \qquad \Gamma, y : A, q : M =_A y \vdash P : \Box_i \qquad \Gamma \vdash u : P\{y := M\}\{q :=\texttt{refl}\, M\}}{\Gamma \vdash J(y.q.P,\, e\,,u) : P\{y := N\}\{q := e\}}$$

$$J(y.q.P,\, \texttt{refl}\, M,\, u) \equiv u$$

FIGURE 2 – Extensions de CC_ω

Théorème 3. *Si* $M \equiv N$ *alors* $[M]_f \equiv [N]_f$.

Théorème 4. *Si* $\Gamma \vdash M : A$ *alors* $[\![\Gamma]\!]_f \vdash [M]_f : [\![A]\!]_f$.

Les preuves de ces théorèmes ne sont pas difficiles : le lemme de substitution se montre par induction sur le terme, et les deux théorèmes de préservation par induction sur les dérivations de conversion et de typage. Les cas à traiter sont néanmoins relativement nombreux, aussi nous fournissons une formalisation en Coq que nous présenterons dans la section 6.

Enfin, comme $[\![\Pi\, X : \Box.\ X]\!]_f = (\Pi\, X : \Box.\ X) \times \mathbb{B}$, on a :

Théorème 5. *La traduction préserve la consistance.*

Ce modèle est assez simple et pourtant il implique déjà strictement plus que CC_ω : grâce au booléen attaché aux fonctions on peut discriminer deux fonctions qui sont pourtant extensionnellement égales.

Théorème 6. *On définit* `funext` *par*

$$\texttt{funext} := \Pi(A : \Box)(B : \Box)(f\, g : A \to B).(\Pi\, x : A.\ f\ x = g\ x) \to f = g$$

Alors, il existe un terme clos de type

$$\vdash [\![\texttt{funext} \to \bot]\!]_f$$

Démonstration. On instancie A et B par un type X et f et g par $(\lambda\, x : X.\ x, \texttt{true})$ et $(\lambda\, x : X.\ x, \texttt{false})$. On a bien $[\![\Pi\, x : A.\ f\ x = g\ x]\!]_f$ car l'application n'observe pas le booléen. D'où $(\lambda\, x : X.\ x, \texttt{true}) = (\lambda\, x : X.\ x, \texttt{false})$. D'où $\texttt{true} = \texttt{false}$. □

Ce modèle montre donc que : si CC_ω est consistant, alors `funext` n'y est pas dérivable. Or `funext` est valable dans de nombreux autres modèles de CC_ω (si ce n'est tous), on en déduit donc qu'il est indépendant de CC_ω. On remarquera que l'on peut reproduire cette construction du produit avec un booléen dans un modèle ensembliste, mais notre approche en donne une présentation *synthétique*, indépendante du modèle.

3. La bissimilarité n'implique pas l'égalité des *streams*

Cette fois, on enrichit notre système avec des types coinductifs, et en l'occurrence des *streams*. Il y a deux notions d'égalité pour les *streams* : celle induite par le type identité et celle donnée par la bissimilarité. Deux *streams* sont bissimilaires si leurs têtes sont égales et que leurs queues sont bissimilaires. La traduction que l'on présente ici montre que la bissimilarité n'entraîne pas l'égalité de deux *streams*. Avec la même idée que pour la traduction précédente, un *stream* est traduit par une paire d'un *stream* et d'un booléen (toujours `true` en provenance de la théorie source). La fonction « tête » `hd` n'observe pas le booléen et la fonction « queue » `tl` le propage. Comme la bissimilarité observe tous les éléments à l'aide de `hd`, elle n'observe jamais le booléen et ne peut donc discriminer deux *streams* qui sont identiques mais ne portent pas le même booléen.

Le type des *streams* de A est noté `stream` A, on observe un *stream* à l'aide de `hd` et `tl` et on en construit à l'aide de `stream_corec`. La bissimilarité des *streams* M et N est notée `bisim` $A\ M\ N$, observée par $\texttt{hd}_\texttt{b}$ et $\texttt{tl}_\texttt{b}$, et construite par `bisim_corec`. Les règles de typage sont données en figure 3.

$$A, B, M, N \quad ::= \quad \ldots \mid \texttt{stream}\ A \mid \texttt{hd}\ M \mid \texttt{tl}\ M \mid \texttt{stream\_corec}\ S\ M_0\ M_1\ N \\ \mid \texttt{bisim}\ A\ M\ N \mid \texttt{hd}_\texttt{b}\ M \mid \texttt{tl}_\texttt{b}\ M \mid \texttt{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N$$

$$\frac{\Gamma \vdash A : \square_i}{\Gamma \vdash \texttt{stream}\ A : \square_i} \qquad \frac{\Gamma \vdash A : \square_i \qquad \Gamma \vdash M : \texttt{stream}\ A \qquad \Gamma \vdash N : \texttt{stream}\ A}{\Gamma \vdash \texttt{bisim}\ A\ M\ N : \square_i} \qquad \frac{\Gamma \vdash M : \texttt{stream}\ A}{\Gamma \vdash \texttt{hd}\ M : A}$$

$$\frac{\Gamma \vdash M : \texttt{stream}\ A}{\Gamma \vdash \texttt{tl}\ M : \texttt{stream}\ A} \qquad \frac{\Gamma \vdash M : \texttt{bisim}\ A\ N_1\ N_2}{\Gamma \vdash \texttt{hd}_\texttt{b}\ M : \texttt{hd}\ N_1 = \texttt{hd}\ N_2} \qquad \frac{\Gamma \vdash M : \texttt{bisim}\ A\ N_1\ N_2}{\Gamma \vdash \texttt{tl}_\texttt{b}\ M : \texttt{bisim}\ A\ (\texttt{tl}_\texttt{b}\ N_1)\ (\texttt{tl}_\texttt{b}\ N_2)}$$

$$\frac{\Gamma \vdash S : \square_i \qquad \Gamma \vdash M_0 : S \to A \qquad \Gamma \vdash M_1 : S \to S \qquad \Gamma \vdash N : S}{\Gamma \vdash \texttt{stream\_corec}\ S\ M_0\ M_1\ N : \texttt{stream}\ A}$$

$$\frac{\Gamma \vdash S : \texttt{stream}\ A \to \texttt{stream}\ A \to \square_i \qquad \Gamma \vdash M_0 : T_0 \qquad \Gamma \vdash M_1 : T_1 \qquad \Gamma \vdash N : S\ P_0\ P_1}{\Gamma \vdash \texttt{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N : \texttt{bisim}\ A\ P_0\ P_1}$$

$$\text{où} \quad \begin{array}{rcl} T_0 & := & \Pi(s_0\ s_1 : \texttt{stream}\ A).\ S\ s_0\ s_1 \to \texttt{hd}\ s_0 =_A \texttt{hd}\ s_1 \\ T_1 & := & \Pi(s_0\ s_1 : \texttt{stream}\ A).\ S\ s_0\ s_1 \to S\ (\texttt{tl}\ s_0)\ (\texttt{tl}\ s_1) \end{array}$$

$$\texttt{hd}\ (\texttt{stream\_corec}\ S\ M_0\ M_1\ N) \equiv M_0\ N$$

$$\texttt{tl}\ (\texttt{stream\_corec}\ S\ M_0\ M_1\ N) \equiv \texttt{stream\_corec}\ S\ M_0\ M_1\ (M_1\ N)$$

$$\texttt{hd}_\texttt{b}\ (\texttt{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N) \equiv M_0\ P_0\ P_1\ N$$

$$\texttt{tl}_\texttt{b}\ (\texttt{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N) \equiv \texttt{bisim\_corec}\ S\ M_0\ M_1\ (\texttt{tl}\ P_0)\ (\texttt{tl}\ P_1)\ (M_1\ P_0\ P_1\ N)$$

FIGURE 3 – Types coinductifs

Définition 7. *La traduction* $[\ ]_s$ *de* $\mathrm{CC}_\omega + = + \texttt{stream}$ *vers* $\mathrm{CC}_\omega + = + \texttt{stream} + \Sigma + \mathbb{B}$ *est définie par :*

$$\begin{array}{ll}
[\square_i]_s & := \square_i \\
[x]_s & := x \\
[\texttt{stream}\ A]_s & := (\texttt{stream}\ [\![A]\!]_s) \times \mathbb{B} \\
[\texttt{hd}\ M]_s & := \texttt{hd}\ (\pi_1\ [M]_s) \\
[\texttt{tl}\ M]_s & := (\texttt{tl}\ (\pi_1\ [M]_s),\ \pi_2\ [M]_s) \\
[\texttt{stream_corec}\ S\ M_0\ M_1\ N]_s & := (\texttt{stream_corec}\ [S]_s\ [M_0]_s\ [M_1]_s\ [N]_s,\ \texttt{true}) \\
[\texttt{bisim}\ A\ M\ N]_s & := \texttt{bisim}\ [\![A]\!]_s\ [M]_s\ [N]_s \\
[\texttt{hd}_\texttt{b}\ M]_s & := \texttt{hd}_\texttt{b}\ [M]_s \\
[\texttt{tl}_\texttt{b}\ M]_s & := \texttt{tl}_\texttt{b}\ [M]_s \\
[\texttt{bisim_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N]_s & := \texttt{bisim_corec}\ \big(\lambda s_0\ s_1.\ [S]_s\ (s_0, (\pi_2\ [P_0]_s))\ (s_1, (\pi_2\ [P_1]_s))\big) \\
 & \quad \big(\lambda s_0\ s_1\ s.\ [M_0]_s\ (s_0, (\pi_2\ [P_0]_s))\ (s_1, (\pi_2\ [P_1]_s))\ s\big) \\
 & \quad \big(\lambda s_0\ s_1\ s.\ [M_1]_s\ (s_0, (\pi_2\ [P_0]_s))\ (s_1, (\pi_2\ [P_1]_s))\ s\big) \\
 & \quad \big(\pi_1\ [P_0]_s\big)\ \big(\pi_1\ [P_1]_s\big)\ [N]_s \\
[\ldots]_s & := \ldots \\
[\![A]\!]_s & := [A]_s
\end{array}$$

Dans les cas non spécifiés, la traduction commute avec les constructeurs.

Comme pour la traduction précédente, $[\ ]_s$ vérifie le lemme de substitution et préserve la conversion. On en déduit la préservation du typage :

Théorème 8. *Si* $\Gamma \vdash M : A$ *alors* $[\![\Gamma]\!]_s \vdash [M]_s : [\![A]\!]_s$.

Remarque. Ce théorème n'est correct qu'en présence de *surjective pairing* définitionnel (règle η sur les types Σ). Elle est nécessaire pour montrer que la règle de typage de `bisim_corec` est bien préservée.

L'absurde est traduit par lui-même donc on a immédiatement :

Proposition 9. *La traduction préserve la consistance.*

Théorème 10. *On définit* `streamext` *par*

$$\texttt{streamext} := \Pi(A : \square)(s_1 s_2 : \texttt{stream}\ A).\ \texttt{bisim}\ A\ s_1\ s_2 \to s_1 = s_2$$

Alors il existe un terme clos de type

$$\vdash [\![\texttt{streamext} \to \bot]\!]_s$$

Par conséquent, si CC_ω *est consistant,* `streamext` *n'est pas dérivable dans* CC_ω.

Démonstration. Encore une, fois la bissimilarité n'observe pas le booléen contrairement à l'égalité. Étant donné un type X et $x : X$ on peut construire le *stream* $\mathsf{id}_x := (x, x, \dots)$. On instancie alors `streamext` avec $(\mathsf{id}_x, \texttt{true})$ et $(\mathsf{id}_x, \texttt{false})$. On en déduit $\texttt{true} = \texttt{false}$. □

4. Nier l'extensionnalité propositionnelle

Dans cette section, nous utilisons l'astuce du $. \times \mathbb{B}$ une dernière fois pour montrer que l'extensionnalité propositionnelle n'est pas dérivable dans CC_ω. Cette fois, c'est sur l'univers $\square$ que l'on travaille et un type est traduit par la paire de ce type et d'un booléen (toujours `true` en provenance de la théorie source). Quand il est vu en tant que terme, un type transporte un booléen. Par contre, quand il est considéré en tant que type, le booléen est oublié. Mais comme il n'y a pas de construction qui observe les types en tant que terme, deux propositions identiques qui n'ont pas le même booléen ne peuvent être distinguées. De la même façon, cette traduction nie aussi l'univalence sur tous les univers $\square_i$.

Définition 11. *La traduction* $[\ ]_t$ *de* $\mathrm{CC}_\omega + = + *$ *vers* $\mathrm{CC}_\omega + = + * + \Sigma + \mathbb{B}$ *est définie par :*

$$\begin{aligned}
[\square_i]_t &:= (\square_i \times \mathbb{B}, \texttt{true}) \\
[x]_t &:= x \\
[\Pi\, x : A.\ B]_t &:= (\Pi\, x : [\![A]\!]_t.\ [\![B]\!]_t, \texttt{true}) \\
[M =_A N]_t &:= ([M]_t =_{[\![A]\!]_t} [N]_t, \texttt{true}) \\
[*]_t &:= (* \times \mathbb{B}, \texttt{true}) \\
[\dots]_t &:= \dots \\
[\![A]\!]_t &:= \pi_1\ [A]_t
\end{aligned}$$

Dans les cas non spécifiés la traduction commute avec les constructeurs.

Théorème 12. *La traduction* $[\ ]_t$ *vérifie le lemme de substitution, préserve la conversion et le typage.*

Proposition 13. *La traduction* $[\ ]_t$ *préserve la consistance.*

Démonstration. En effet, $[\![\Pi\, X : \Box.\ X]\!]_t = \Pi\, X : \Box \times \mathbb{B}.\ \pi_1\ X$ qui est habité si et seulement si $\Pi\, X : \Box.\ X$ l'est. □

Théorème 14. *On définit* `propext` *par*

$$\texttt{propext} := \Pi\, A\ B : *.\ (A \to B) \to (B \to A) \to A = B$$

Il existe un terme clos de type

$$\vdash [\![\texttt{propext} \to \bot]\!]_s$$

Par conséquent, si CC_ω *est consistant,* `propext` *n'est pas dérivable dans* CC_ω.

Démonstration. Comme précédemment, en instanciant A et B par $(X, \texttt{true})$ et $(X, \texttt{false})$ avec $X : *$. □

5. Polymorphisme ad-hoc

La dernière traduction que nous présentons interprète une forme de polymorphisme ad-hoc. Cette fois, on n'implémente pas seulement un axiome mais un véritable opérateur (qui a des règles de réduction), à savoir le *pattern matching* sur un type. Cet opérateur, que l'on notera `univ_rec`, permet de faire une analyse de cas sur la forme normale d'un type. Grâce à lui, une fonction qui prend un type en argument peut avoir un comportement différent en fonction du type qui lui est passé. Cet opérateur peut donc être vu comme un opérateur de polymorphisme ad-hoc, ou encore comme une façon de faire du typage dynamique. Un tel opérateur pour $\Box_0$ est présenté dans [9, section 14.2].

L'idée de la traduction est d'interpréter un type par son code dans un univers interne `TYPE`. Cet univers interne est un type inductif-récursif [3] et l'on peut donc interpréter `univ_rec` par le *pattern matching* sur `TYPE`. Cette traduction fait une hypothèse de « monde clos », à savoir qu'on ne peut pas ajouter un nouveau type (un inductif par exemple) dans l'univers $\Box_i$ à la volée. En effet, contrairement aux précédentes traductions que l'on pouvait étendre à un nouveau type, celle-ci nécessite d'être redéfinie entièrement.

Pour que la traduction ne dépende que de la syntaxe du terme et pas de l'arbre de typage, nous avons besoin de garder trace du niveau d'univers des types considérés. Pour cela, on affaiblit légèrement CC_ω et on utilise le système CC^e_ω qui contient des annotations d'univers *e*xplicites. Habituellement, on écrit $\Gamma \vdash M : A : \Box_i$ pour signifier que les deux jugements $\Gamma \vdash M : A$ et $\Gamma \vdash A : \Box_i$ sont dérivables. Malheureusement, cela ne permet pas de faire de preuve par induction sur l'arbre de typage puisque l'on a deux arbres non liés. Dans CC^e_ω, on garde trace de la sorte tout au long du typage et $\Gamma \vdash M : A : \Box_i$ est un jugement à proprement parler. Nous ajoutons également des annotations d'univers sur les produits dépendants et dans le type de retour du `if` (cf. définition 16). Les règles de CC^e_ω sont données en figure 4. Nous omettrons les annotations d'univers quand celle-ci sont évidentes (notamment dans la notation $\to$).

Pour alléger la présentation, nous avons affaibli la règle de typage du produit : dans CC^e_ω, il n'est possible de typer $\Pi\, x : A.\ B$ avec $A : \Box_i$ et $B : \Box_j$ que si $i \leq j$. En toute rigueur, il faudrait considérer tous les produits. Cela nécessiterait d'ajouter des constructeurs aux inductifs $\texttt{TYPE}_i$ définis par la suite : dans $\texttt{TYPE}_k$, il faut un constructeur pour chaque règle (i, j, k) du PTS. Cela ne pose pas de problème particulier mais n'est pas très agréable à écrire.

Enfin, pour montrer que la traduction s'étend simplement, nous ajoutons les booléens dans la théorie source, la généralisation à d'autres constructeurs de type est directe.

Définition 15. *Notre théorie cible sera* CC_ω *enrichie avec des types inductifs-récursifs. Dans celle-ci, on définit les familles inductives-récursives* $(\mathtt{TYPE}_i, \mathtt{Elt}_i)$ *pour* $i \in \mathbb{N}$ *données par :*

$$
\begin{array}{l}
\mathtt{Inductive}\ \mathtt{TYPE}_0 : \square_1 := \\
\mid \mathtt{B} \ : \ \mathtt{TYPE}_0 \\
\mid \Pi_0 \ : \ \Pi(\mathtt{A} : \mathtt{TYPE}_0).\ (\mathtt{Elt}_0\ \mathtt{A} \to \mathtt{TYPE}_0) \to \mathtt{TYPE}_0 \\
\mathtt{with}\ \mathtt{Elt}_0 : \mathtt{TYPE}_0 \to \square_0 := \mathtt{fun} \\
\mid \mathtt{B} \quad\ \Rightarrow\ \mathbb{B} \\
\mid \Pi_0\ \mathtt{A}\ \mathtt{B} \ \Rightarrow\ \Pi(\mathtt{x} : \mathtt{Elt}_0\ \mathtt{A}).\ \mathtt{Elt}_0\ (\mathtt{B}\ \mathtt{x}).
\end{array}
$$

$$
\begin{array}{l}
\mathtt{Inductive}\ \mathtt{TYPE}_{i+1} : \square_{i+2} := \\
\mid \mathtt{U}_i \quad\ : \ \mathtt{TYPE}_{i+1} \\
\mid \Pi^0_{i+1} \ : \ \Pi(\mathtt{A} : \mathtt{TYPE}_0).\ (\mathtt{Elt}_0\ \mathtt{A} \to \mathtt{TYPE}_{i+1}) \to \mathtt{TYPE}_{i+1} \\
\mid \ldots \quad\ : \ \ldots \\
\mid \Pi^{i+1}_{i+1} \ : \ \Pi(\mathtt{A} : \mathtt{TYPE}_{i+1}).\ (\mathtt{Elt}_{i+1}\ \mathtt{A} \to \mathtt{TYPE}_{i+1}) \to \mathtt{TYPE}_{i+1} \\
\mathtt{with}\ \mathtt{Elt}_{i+1} : \mathtt{TYPE}_{i+1} \to \square_{i+1} := \mathtt{fun} \\
\mid \mathtt{U}_i \qquad\ \Rightarrow\ \mathtt{TYPE}_i \\
\mid \Pi^0_{i+1}\ \mathtt{A}\ \mathtt{B} \ \Rightarrow\ \Pi(\mathtt{x} : \mathtt{Elt}_0\ \mathtt{A}).\ \mathtt{Elt}_{i+1}\ (\mathtt{B}\ \mathtt{x}). \\
\mid \ldots \qquad\ \Rightarrow\ \ldots \\
\mid \Pi^{i+1}_{i+1}\ \mathtt{A}\ \mathtt{B} \ \Rightarrow\ \Pi(\mathtt{x} : \mathtt{Elt}_{i+1}\ \mathtt{A}).\ \mathtt{Elt}_{i+1}\ (\mathtt{B}\ \mathtt{x}).
\end{array}
$$

Les règles de typage et de réduction générées par ces définitions sont données formellement dans l'annexe A. Ces inductifs-récursifs vérifient la condition de stricte positivité (ils sont typés par Agda par exemple).

Définition 16. *On définit la traduction* $[\ \ ]_T$ *de* $\mathrm{CC}^e_\omega + \mathbb{B}$ *vers* $\mathrm{CC}_\omega + \mathbb{B} + (\mathtt{TYPE}_i)_{i\in\mathbb{N}}$ *par :*

$$
\begin{array}{ll}
[\square_i]_T & := \mathtt{U}_i \\
[x]_T & := x \\
[\Pi^j\, x : A : \square_i.\ B]_T & := \Pi^i_j\ [A]_T\ (\lambda\, x : [\![A]\!]^i_T.\ [B]_T) \\
[\lambda\, x : A : \square_i.\ M]_T & := \lambda\, x : [\![A]\!]^i_T.\ [M]_T \\
[M\ N]_T & := [M]_T\ [N]_T \\
[\mathbb{B}]_T & := \mathtt{B} \\
[\mathtt{true}]_T & := \mathtt{true} \\
[\mathtt{false}]_T & := \mathtt{false} \\
[\mathtt{if}\ M\ \mathtt{ret}^i\ P\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2]_T & := \mathtt{if}\ [M]_T\ \mathtt{ret}\ (\lambda\, b : \mathbb{B}.\ \mathtt{Elt}_i\ ([P]_T\ b))\ \mathtt{then}\ [N_1]_T\ \mathtt{else}\ [N_2]_T \\
[\![A]\!]^i_T & := \mathtt{Elt}_i\ [A]_T \\
[\![\,.\,]\!]_T & := \ . \\
[\![\Gamma,\, x : A : \square_i]\!]_T & := [\![\Gamma]\!]_T,\, x : [\![A]\!]^i_T
\end{array}
$$

La traduction vérifie le lemme de substitution et la préservation de la conversion. On en déduit :

Théorème 17. *La traduction préserve le typage :*

$$\textit{Si}\ \Gamma \vdash M : A : \square_i\ \ \textit{alors}\ [\![\Gamma]\!]_T \vdash [M]_T : [\![A]\!]^i_T.$$

On définit maintenant l'opérateur auquel la traduction donne accès.

$$A,\, B,\, M,\, N \;::=\; \Box_i \mid x \mid M\ N \mid \lambda\, x : A : \Box_i.\ M \mid \Pi^j\, x : A : \Box_i.\ B$$
$$\Gamma,\, \Delta \;::=\; . \mid \Gamma,\, x : A : \Box_i$$

$$\frac{}{\vdash .} \qquad \frac{\Gamma \vdash A : \Box_i : \Box_{i+1}}{\vdash \Gamma,\, x : A : \Box_i} \qquad \frac{\Gamma \vdash A : \Box_i : \Box_{i+1}}{\Gamma,\, x : A : \Box_i \vdash x : A : \Box_i} \qquad \frac{\Gamma \vdash M : B : \Box_i \qquad \Gamma \vdash A : \Box_i : \Box_{i+1}}{\Gamma,\, x : A : \Box_i \vdash M : B : \Box_i}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \Box_i : \Box_{i+1} : \Box_{i+2}} \qquad \frac{\Gamma \vdash A : \Box_i : \Box_{i+1} \qquad \Gamma,\, x : A : \Box_i \vdash B : \Box_j : \Box_{j+1} \qquad i \leq j}{\Gamma \vdash (\Pi^j\, x : A : \Box_i.\ B) : \Box_j : \Box_{j+1}}$$

$$\frac{\Gamma,\, x : A : \Box_i \vdash M : B : \Box_j \qquad \Gamma \vdash (\Pi^j\, x : A : \Box_i.\ B) : \Box_j : \Box_{j+1}}{\Gamma \vdash (\lambda\, x : A : \Box_i.\ B) : (\Pi^j\, x : A : \Box_i.\ B) : \Box_j}$$

$$\frac{\Gamma \vdash M : (\Pi^j\, x : A : \Box_i.\ B) : \Box_j \qquad \Gamma \vdash N : A : \Box_i}{\Gamma \vdash M\ N : B\,\{x := N\} : \Box_j}$$

$$\frac{\Gamma \vdash M : B : \Box_i \qquad \Gamma \vdash A : \Box_i : \Box_{i+1} \qquad A \equiv B}{\Gamma \vdash M : A : \Box_i} \qquad (\lambda\, x : A : \Box_i.\ M)\ N \equiv M\,\{x := N\}$$

FIGURE 4 – Règles de typage de CC^e_ω (règles de congruence omises)

Définition 18. *Soit* $(\texttt{univ\_rec}_i)_{i\in\mathbb{N}}$ *la famille d'opérateurs de* $\mathrm{CC}^e_\omega + \mathbb{B}$ *définie par :*

$$\frac{\Gamma \vdash P : \Box_0 \to \Box_0 : \Box_1 \qquad \Gamma \vdash p_{\mathbb{B}} : P\ \mathbb{B} : \Box_0 \qquad \Gamma \vdash p^0_\Pi : P\Pi_0 : \Box_0}{\Gamma \vdash \texttt{univ\_rec}_0(P,\, p_{\mathbb{B}},\, p^0_\Pi) : \Pi\, A : \Box_0.\ P\ A : \Box_1}$$

$$\frac{\begin{array}{c}\Gamma \vdash P : \Box_{i+1} \to \Box_{i+1} : \Box_{i+2} \qquad \Gamma \vdash p_{\Box_i} : P\ \Box_i : \Box_{i+1} \\ \Gamma \vdash p^0_\Pi : P\Pi^0_{i+1} : \Box_{i+1} \qquad \ldots \qquad \Gamma \vdash p^i_\Pi : P\Pi^i_{i+1} : \Box_{i+1} \qquad \Gamma \vdash p^{i+1}_\Pi : P\Pi_{i+1} : \Box_{i+1}\end{array}}{\Gamma \vdash \texttt{univ\_rec}_{i+1}(P,\, p_{\Box_i},\, p^0_\Pi,\, \ldots,\, p^{i+1}_\Pi) : \Pi\, A : \Box_{i+1}.\ P\ A : \Box_{i+2}}$$

$$\texttt{univ\_rec}_0(P,\, p_{\mathbb{B}},\, p^0_\Pi)\ \mathbb{B} \equiv p_{\mathbb{B}} \qquad\qquad \texttt{univ\_rec}_{i+1}(P,\, p_{\Box_i},\, p^0_\Pi,\, \ldots,\, p^{i+1}_\Pi)\ \Box_i \equiv p_{\Box_i}$$

$$\begin{array}{lcl} \textit{où} \quad P\Pi_i & := & \Pi\,(A : \Box_i)(B : A \to \Box_i : \Box_{i+1}).\ P\ A \to (\Pi^i\, x : A : \Box_i.\ P\ (B\ x)) \to P\ (\Pi^i\, x : A : \Box_i.\ B\ x) \\ \quad\ P\Pi^i_j & := & \Pi\,(A : \Box_i)(B : A \to \Box_j : \Box_{j+1}).\ (\Pi^j\, x : A : \Box_i.\ P\ (B\ x)) \to P\ (\Pi^j\, x : A : \Box_i.\ B\ x) \end{array}$$

$$\begin{array}{l} \texttt{univ\_rec}_j(P,\, p,\, p^0_\Pi,\, \ldots,\, p^j_\Pi)\ (\Pi^j\, x : A : \Box_j.\ B) \\ \quad \equiv p^j_\Pi\ A\ B\ (\texttt{univ\_rec}_j(P,\, p,\, p^0_\Pi,\, \ldots,\, p^j_\Pi)\ A)\ (\lambda\, x : A : \Box_j.\ \texttt{univ\_rec}_j(P,\, p,\, p^0_\Pi,\, \ldots,\, p^j_\Pi)\ (B\ x)) \end{array}$$

$$\begin{array}{l} \texttt{univ\_rec}_j(P,\, p,\, p^0_\Pi,\, \ldots,\, p^j_\Pi)\ (\Pi^j\, x : A : \Box_i.\ B) \\ \quad \equiv p^i_\Pi\ A\ B\ (\lambda\, x : A : \Box_i.\ \texttt{univ\_rec}_j(P,\, p,\, p^0_\Pi,\, \ldots,\, p^j_\Pi)\ (B\ x)) \qquad\qquad (i < j) \end{array}$$

L'opérateur $\texttt{univ\_rec}_i$ permet d'observer directement la forme normale d'un type dans $\Box_i$ et permet ainsi de définir des fonctions dont le résultat dépend du type de ses arguments. Par exemple, $\texttt{univ\_rec}_0$ permet de définir une fonction de type $\Pi\, A : \Box_0.\ A \to A$ qui vaut la négation sur $\mathbb{B}$ et l'identité sur les autres types.

Théorème 19. *La traduction* $[\ ]_T$ *donne un contenu calculatoire aux opérateurs* $\texttt{univ\_rec}_i$ $(i \in \mathbb{N})$.

Démonstration. On étend la traduction avec :

$$[\texttt{univ\_rec}_i(P, p, p^0_\Pi, \ldots, p^i_\Pi)]_T := \texttt{TYPE\_rec}_i\ (\lambda\, c : \texttt{TYPE}_i.\ \texttt{Elt}_i\ ([P]_T\ c))\ [p]_T\ [p^0_\Pi]_T\ \ldots\ [p^i_\Pi]_T$$

et on vérifie les règles de typage et de conversion de $\texttt{univ\_rec}_i$. □

Quand la théorie source a un type vide $\mathbf{0}$, on peut étendre la traduction en ajoutant un code pour $\mathbf{0}$ dans $\texttt{TYPE}_0$, la cohérence est alors préservée car $[\![\mathbf{0}]\!]^0_T = \mathbf{0}$. Par contre, quand ce n'est pas le cas, la préservation de la cohérence est moins claire. En effet, $[\![\Pi^i\, A : \square_i : \square_{i+1}.\ A]\!]_T = \Pi\, A : \texttt{TYPE}_i.\ \texttt{Elt}_i\ A$ et il n'y a pas de code évident à fournir à ce type qui permettrait de retrouver une inconsistance. Par exemple, si l'on essaie de donner le code de l'encodage imprédicatif du faux on retrouve le même type.

Théorème 20. *L'opérateur* $\texttt{univ\_rec}_i$ *est admissible (i.e. son ajout préserve la consistance) dans* $\mathrm{CC}^e_\omega + \mathbb{B} + \mathbf{0}$.

Les conséquences d'un tel opérateur sont diverses. Mais en particulier, en faisant dépendre le comportement des fonctions de leur type, cet opérateur va à l'encontre de la paramétricité :

Proposition 21. *Si* $\mathrm{CC}^e_\omega + \mathbb{B} + \mathbf{0}$ *est consistant, certains énoncés de paramétricité ne sont pas dérivables dans ce système.*

Démonstration. Par exemple la fonction qui vaut la négation sur $\mathbb{B}$ et l'identité sur les autres types permet de construire un terme de type :

$$\big(\Pi\,(f : \Pi\, A : \square_0.\ A)(A : \square_0)(\dot{A} : A \to \square_0)(x : A).\ \dot{A}\ x \to \dot{A}\ (f\ x)\big) \to \bot$$

qui est la négation de l'énoncé de paramétricité du type $\Pi\, A : \square_0.\ A \to A$. □

6. Formalisation

Nous avons formalisé le lemme de substitution, la préservation de la conversion et la préservation du typage pour les trois premières traductions ($[\ ]_f$, $[\ ]_s$ et $[\ ]_t$). Pour cela nous avons utilisé l'assistant de preuve Coq et nous sommes reposés sur une formalisation précédente des PTS de Siles et Herbelin [11]. Pour chaque théorie considérée, nous définissons d'abord le type `Term` des termes non typés en utilisant des indices de Bruijn (figure 5), puis les contextes, la substitution et la conversion. Et enfin, la bonne formation des contextes et la relation de typage à l'aide de familles inductives mutuelles. On peut alors définir la traduction sur les termes non typés. Par exemple, supposons que `S` soit le module dans lequel est défini $\mathrm{CC}_\omega +=$ et que `T` soit celui dans lequel est défini $\mathrm{CC}_\omega +=+\Sigma+\mathbb{B}$. Alors, la traduction sur les fonctions $[\ ]_f$ est implémentée par la fonction `tsl`, définie par induction sur `S.Term` (figure 5).

```
Inductive Term : Set :=
 | Var : ℕ → Term
 | Sort : Sorts → Term
 | Π : Term → Term → Term
 | λ : Term → Term → Term
 | App : Term → Term → Term
 | Eq : ∀ (A t₁ t₂ : Term), Term
 | refl : Term → Term
 | J : ∀ (A P t₁ u t₂ p : Term), Term.
```

```
Fixpoint tsl (t : S.Term) : T.Term :=
 match t with
  | S.Var v  ⇒  T.Var v
  | S.Sort s  ⇒  T.Sort s
  | S.Π A B  ⇒  T.Σ (Π A^t B^t) Bool
  | S.λ A M  ⇒  T.Pair (λ A^t M^t) true
  | S.App M N  ⇒  T.App (π₁ M^t) N^t
  | S.Eq A t₁ t₂  ⇒  T.Eq A^t t₁^t t₂^t
  | S.refl e  ⇒  T.refl e^t
  | S.J A P t₁ u t₂ p  ⇒  T.J A^t P^t t₁^t u^t t₂^t p^t
 end where "M^t" := (tsl M).
```

FIGURE 5 – Définition des termes de $\mathrm{CC}_\omega +=$ et de la traduction $[\ ]_f$

À l'aide de lemmes intermédiaires on en vient à montrer (par induction mutuelle) :

```
Theorem tsl_correctness :
  (∀ Γ,  Γ ⊣ → Γ^t ⊣) ∧ (∀ Γ M A, Γ ⊢ M : A →Γ^t  ⊢ M^t : A^t).
```

qui correspond à la préservation du typage de $[\ ]_f$.

Des développements similaires ont été menés pour $[\ ]_s$ et $[\ ]_t$. Les trois formalisations sont disponibles sur GitHub [†].

7. Instrumentation sous forme d'un plug-in

Il n'est pas facile de savoir quels axiomes réalisent une traduction. En particulier, les termes traduits deviennent vite gros et assez peu lisibles, et ne serait-ce qu'obtenir leur traduction à la main est pénible. Heureusement, les traductions que nous avons présentés reposent sur CC_ω qui est suffisamment proche de la théorie sous-tendant Coq pour que l'on puisse les adapter à Coq. Coq dispose d'un mécanisme de plug-in qui permet d'étendre ses fonctionnalités. Nous avons donc implémenté un plug-in par traduction.

Concrètement, un plug-in s'applique à une traduction de CC_ω dans lui-même et fournit une fonction qui traduit un terme. Mais plutôt que de ne travailler qu'à travers la traduction, on propose un mécanisme pour ajouter un axiome et vérifier que la traduction lui donne bien un sens. Cette méthode a l'avantage de ne demander d'habiter des termes traduits (et donc peu naturels) qu'au moment d'ajouter l'axiome, et non à chaque théorème que l'on veut démontrer.

La commande qui permet de faire cela est `Implement`. Supposons que l'on ait une traduction $[\ ]$ qui donne un contenu calculatoire à `PrincipeCool`. On fait alors appel à la commande

```
Implement ax : PrincipeCool.
```

Une preuve interactive est alors ouverte et un terme de type $[\![\texttt{PrincipeCool}]\!]$ est demandé. Une fois cela fait, le terme fournit est nommé $\texttt{ax}^\bullet$, l'axiome `ax` de type `PrincipeCool` est ajouté au contexte et la traduction est étendue à `ax` en $\texttt{ax}^\bullet$. On peut alors continuer de travailler dans la théorie enrichie avec le nouvel axiome et la correction de la traduction nous garantit l'équiconsistance.

Il est intéressant de pouvoir ajouter des axiomes à la volée. Par exemple si l'on imagine une traduction qui donnerait du sens à `funext` on pourrait vouloir, après avoir commencé à l'utiliser, prouver des égalités sur cet axiome lui-même (comme on fait fréquemment en théorie des types homotopiques par exemple [13]). C'est exactement ce que permet la commande `Implement`.

Nous avons implémenté des prototypes de plug-in pour les trois premières traductions ($[\ ]_f$, $[\ ]_s$ et $[\ ]_t$). Leurs codes ainsi que des exemples sont disponibles sur GitHub [†].

Travaux futurs

Les traductions de programme sont un domaine excitant dans lequel il reste beaucoup de choses à faire. En premier lieu, nous souhaiterions améliorer les plug-ins qui ne sont encore que des prototypes pour le moment. Cela nécessite de généraliser les traductions au langage de Coq en entier (inductifs et coinductifs notamment) et de gérer correctement les univers de Coq.

Pour l'implémentation du polymorphisme ad-hoc, cela nécessiterait la présence d'inductifs-récursifs dans Coq. C'est le sujet de recherches en cours et un prototype a déjà été implémenté par Mathieu Sozeau [‡]. Cependant, la condition de garde n'est pas encore implémentée.

†. `https://github.com/CoqHott/Program-translations-CC-omega`
‡. `https://github.com/mattam82/coq/tree/IR`

Pour l'instant notre implémentation des plug-ins est vraiment peu modulaire puisqu'elle nécessite un plug-in par traduction. À terme, nous souhaiterions proposer un framework dans lequel un utilisateur pourrait générer un plug-in pour sa propre traduction facilement – idéalement, en ne donnant la traduction que sur les termes non typés – ainsi que composer plusieurs traductions. Nous souhaiterions aussi pouvoir donner du sens à des règles de réduction, cela nécessiterait de pouvoir ajouter des règles de réduction dans le noyau à la volée, comme cela est possible en Agda par exemple.

Mais le plus important reste le développement de nombreuses traductions de programme pour réaliser des axiomes et des principes logiques. Par exemple, nous nous demandons si l'on peut présenter le modèle setoid de cette manière.

Références

[1] T. Altenkirch. Extensional equality in intensional type theory. In *Proceedings of LICS*, Trento, Italy, July 1999.

[2] J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. In *Proceedings of LICS*, Dubrovnik, Croatia, June 2012.

[3] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2) :525–549, Mar 2000.

[4] M. Hofmann. *Extensional constructs in intensional type theory.* CPHC/BCS distinguished dissertations. Springer, 1997.

[5] G. Jaber, G. Lewertoski, P.-M. Pédrot, N. Tabareau, and M. Sozeau. The Definitional Side of the Forcing. In *Proceedings of LICS*, New-York, USA, July 2016.

[6] C. Kapulkin, P. L. Lumsdaine, and V. Voevodsky. The simplicial model of univalent foundations. *arXiv preprint arXiv :1211.2851*, 2012.

[7] Y. Lafont, B. Reus, and T. Streicher. *Continuation semantics or expressing implication by negation.* Univ. München, Inst. für Informatik, 1993.

[8] A. Miquel. Forcing as a program transformation. In *Proceedings of LICS*, Toronto, Canada, June 2011.

[9] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's type theory*, volume 200. Oxford University Press Oxford, 1990.

[10] P.-M. Pédrot. A functional functional interpretation. In *Proceedings of LICS*, Vienna, Austria, July 2014.

[11] V. Siles and H. Herbelin. Pure type systems conversion is always typable. *Journal of Functional Programming*, 22(2) :153–180, Mar 2012.

[12] The Coq Development Team. *The Coq proof assistant reference manual.* 2015. Version 8.5.

[13] Univalent Foundations Project. *Homotopy Type Theory : Univalent Foundations for Mathematics.* `http://homotopytypetheory.org/book`, 2013.

A. Annexe : règles de typage des codes

$$A, B, M, N \ ::= \ \ldots \mid \mathtt{TYPE}_i \mid \mathtt{Elt}_i \mid \mathtt{U}_i \mid \Pi^i_j \mid \mathtt{TYPE\_rec}_i$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{TYPE}_i : \square_{i+1}} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathtt{Elt}_i : \mathtt{TYPE}_i \to \square_i} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathtt{B} : \mathtt{TYPE}_0} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathtt{U}_i : \mathtt{TYPE}_{i+1}}$$

$$\frac{\vdash \Gamma \qquad i \leq j}{\Gamma \vdash \Pi^i_j : \Pi\, A : \mathtt{TYPE}_i.\ (\mathtt{Elt}_i\ A \to \mathtt{TYPE}_j) \to \mathtt{TYPE}_j}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{TYPE\_rec}_0 : \Pi\, P : \mathtt{TYPE}_0 \to \square_j.\ P\ \mathtt{B} \to P\Pi_0 \to \Pi\, a : \mathtt{TYPE}_0.\ P\ a}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{TYPE\_rec}_{i+1} : \Pi\, P : \mathtt{TYPE}_{i+1} \to \square_j.\ P\ \mathtt{U}_i \to P\Pi^0_{i+1} \to \cdots \to P\Pi^i_{i+1} \to P\Pi_{i+1} \to \Pi\, a : \mathtt{TYPE}_{i+1}.\ P\ a}$$

où $P\Pi_i := \Pi\,(a : \mathtt{TYPE}_i)(b : \mathtt{Elt}_i\ a \to \mathtt{TYPE}_i).\ P\ a \to (\Pi\, x : \mathtt{Elt}_i\ a.\ P\ (b\ x)) \to P\ (\Pi^i_i\ a\ b)$

$P\Pi^i_j := \Pi\,(a : \mathtt{TYPE}_i)(b : \mathtt{Elt}_i\ a \to \mathtt{TYPE}_j).\ (\Pi\, x : \mathtt{Elt}_i\ a.\ P\ (b\ x)) \to P\ (\Pi^i_j\ a\ b)$

$$\begin{aligned}
&\mathtt{Elt}_0\ \mathtt{B} && \equiv \mathbb{B}\\
&\mathtt{Elt}_{i+1}\ \mathtt{U}_i && \equiv \mathtt{TYPE}_i\\
&\mathtt{Elt}_j\ (\Pi^i_j\ a\ b) && \equiv \Pi\, x : \mathtt{Elt}_i\ a.\ \mathtt{Elt}_j\ (b\ x)\\
&\mathtt{TYPE_rec}_0\ P\ p_{\mathbb{B}}\ p^0_\Pi\ \mathtt{B} && \equiv p_{\mathbb{B}}\\
&\mathtt{TYPE_rec}_{i+1}\ P\ p_{\mathtt{U}}\ p^0_\Pi\ \ldots\ p^{i+1}_\Pi\ \mathtt{U}_i && \equiv p_{\mathtt{U}}\\
&\mathtt{TYPE_rec}_j\ P\ p\ p^0_\Pi\ \ldots\ p^j_\Pi\ (\Pi^j_j\ a\ b) && \equiv p^j_\Pi\ a\ b\ (\mathtt{TYPE_rec}_j\ P\ p_{\mathbb{B}}\ p_\Pi\ a)\\
&&& \qquad (\lambda\, x : \mathtt{Elt}_j\ a.\ \mathtt{TYPE_rec}_j\ P\ p_{\mathbb{B}}\ p_\Pi\ (b\ x))\\
&\mathtt{TYPE_rec}_j\ P\ p\ p^0_\Pi\ \ldots\ p^j_\Pi\ (\Pi^i_j\ a\ b) && \equiv p^i_\Pi\ a\ b\ (\lambda\, x : \mathtt{Elt}_j\ a.\ \mathtt{TYPE_rec}_j\ P\ p_{\mathbb{B}}\ p_\Pi\ (b\ x)) \qquad (i < j)
\end{aligned}$$