Formalisation et implémentation des propositions strictes dans MetaCoq

Yann Leray

Stage M2
Rapport

Formalisation et implémentation des propositions strictes dans MetaCoq

Yann Leray
Département d'informatique
École Normale Supérieure

Encadré par
Nicolas Tabareau et Matthieu Sozeau

Gallinette, Inria Rennes - Bretagne Atlantique avec LS2N
2 Chem. de la Houssinière,
44300 Nantes

Mars - Juillet 2022

1 Introduction

Le contexte général

Les assistants de preuve sont des logiciels permettant de rédiger et prouver des propositions mathématiques. Ils sont principalement utilisés dans deux domaines : les mathématiques et la vérification de programmes. En mathématiques, ils permettent de formaliser des théories en posant les différentes définitions, axiomes, théorèmes et surtout en vérifiant les preuves des théorèmes. De manière similaire, l'utilisateur peut exprimer les spécifications d'un algorithme ou programme et rédiger des preuves vérifiées que le programme suit bien lesdites spécifications.

Coq [1] est l'un des plus importants assistants de preuve, développé par l'INRIA. Malgré son âge qui indique une forte robustesse, des preuves de `False` (une proposition sans preuve) sont régulièrement trouvées, avec un rythme d'environ une par an. Pour trouver les bugs qui créent ces preuves incohérentes, l'équipe Gallinette développe le projet MetaCoq [2, 3] dont le but est de formaliser la théorie de Coq et l'implémentation de son noyau de vérification, pour en établir et prouver les propriétés fondamentales, notamment la cohérence (impossibilité de prouver un théorème faux).

Le problème étudié

En 2019, l'équipe de développement de Coq y a ajouté une nouvelle fonctionnalité : SProp [4, 5]. Cette sorte permet de décrire des propositions dont toutes les preuves sont égales définitionnellement (ces notions seront définies plus bas). Un intérêt majeur est de pouvoir utiliser des objets mêlant des types usuels (entiers naturels, booléens) et des preuves sans s'empêcher d'utiliser l'égalité, ce que les propositions usuelles ne permettent pas toujours.

La contribution proposée

Pendant ce stage, j'ai tâché d'ajouter cette fonctionnalité au projet MetaCoq, avec toutes les règles et propriétés qu'elle requiert. Notamment, la théorie autour de SProp n'a jamais été finement décrite, il m'a donc fallu décrire certaines nouvelles propriétés et comportements pour pouvoir reprouver les théorèmes sur la théorie de Coq, avec SProp. Si les fondements de l'extension (marques de relevance) ont pu être ajoutés et largement prouvés corrects, je n'ai malheureusement pas pu finir de prouver les propriétés relatives à la nouvelle stratégie de réduction avec SProp.

Les arguments en faveur de sa validité

Dans la mesure où tout le développement est fait en Coq, le travail est toujours vérifié par l'assistant de preuve directement et on sait immédiatement si une preuve devient fausse. Alors, le fait de retrouver une preuve de la cohérence de la théorie donne une bonne confiance dans la cohérence de l'extension.

Le bilan et les perspectives

Certaines propriétés restent à démontrer afin de conclure quant à la cohérence de l'extension et la correction du noyau vis-à-vis de celle-ci. Les développements sur la version MetaCoq du noyau pourraient ensuite servir à améliorer le support actuel de SProp dans Coq, qui est pour le moment expérimental.

Une manière d'étendre ce développement par la suite est d'élargir l'extension SProp à d'autres sortes : le mécanisme central de marques de relevance est propice à être étendu pour porter d'autres informations de sortes ; il est même envisagé de commencer le développement du « polymorphisme de sortes » dans MetaCoq pour bien étudier et comprendre les comportements avant de les ajouter à Coq.

Table des matières

2 Coq et SProp

L'assistant de preuve Coq est fondé sur l'équivalence de Curry-Howard et une théorie des types nommée PCUIC (*Predicative Calculus of Cumulative Inductive Constructions*, théorie prédicative des constructions inductives cumulatives). Cette théorie des types contient à la fois des types dépendants, des types inductifs et une hiérarchie d'univers cumulative.

2.1 Équivalence de Curry-Howard

Dans PCUIC, les propositions sont des expressions construites notamment à partir d'atomes et combinateurs tels $=, \top, \bot, \neg, \implies, \wedge, \vee, \forall, \exists$. L'équivalence de Curry-Howard propose d'interpréter ces propositions comme des types dont les éléments forment les preuves de la proposition. Dans ce contexte, prouver une proposition revient à construire une expression (aussi appelé terme de preuve) dont le type sera cette proposition ; un type qui a au moins un élément correspond à une proposition prouvée, un type n'en ayant aucun est une proposition qui n'a pas de preuve.

On définira alors $\top$ comme un type non-vide en lui donnant un élément canonique, $\bot$ un type n'en ayant aucun. $P \wedge Q$ est un type qui doit contenir à la fois une preuve de P et de Q, c'est donc le type produit $P \times Q$; de même, $P \vee Q$ est le type somme $P + Q$. Ces types sont en pratique définis comme types inductifs (voir section 2.3), avec pour le produit et la somme des paramètres de type.
Enfin, le type de l'implication $P \implies Q$ doit pouvoir prouver Q en sachant P, c'est-à-dire trouver ou exprimer une preuve de Q en ayant une preuve de P. C'est donc le type des fonctions $P \to Q$.

2.2 Types dépendants

Dans PCUIC, il est possible de définir des types qui dépendent non seulement d'autres types, mais même d'expressions quelconques. En effet, les types sont considérés comme des termes comme les autres, et peuvent par exemple dépendre d'un test booléen via une construction `if then else`.

Le constructeur syntaxique central des types dépendants est le produit $\Pi(a : A), P[a]$ qui permet d'abstraire une variable dans une expression de type. Ainsi, $\Pi(a : A), P[a]$ est le type des fonctions qui prennent en argument une valeur de type A et renvoie un élément de type $P[a]$ (une expression qui peut dépendre de a). Il généralise donc le constructeur de type $\to$ en permettant au codomaine de dépendre de la valeur de l'argument dans le domaine.

En ceci, il offre une parfaite interprétation au constructeur logique $\forall$ qui demande de prouver une propriété pour tout argument, propriété qui dépend elle-même de la valeur de cet argument. Symétriquement, $\exists x, P[x]$ s'interprète naturellement en le type $\Sigma(a : A), P[a]$ des paires dépendantes (x, p) telles que $x : A$ et $p : P[x]$ (le constructeur Σ n'est lui pas primitif à PCUIC et est implémenté grâce à un type inductif).

2.3 Types inductifs

Les types inductifs de Coq peuvent être vus comme une généralisation des types sommes d'OCaml, ou encore comme la généralisation naturelle des GADT d'OCaml au fait que les types sont des termes en Coq.

```
Inductive sum (A : Type) (B : Type) : Type :=
  | left : A -> sum A B
  | right : B -> sum A B.

Inductive equal (A : Type) (a: A) : A -> Type :=
  | refl : equal A a a.

Definition test (arg : sum nat bool) : nat :=
  match arg with
  | left a => a
  | right b => 0
  end.
```

Fig. 1 : Exemples d'utilisation de types inductifs

La définition d'un type inductif contient les paramètres du type, les constructeurs du type et, pour chacun d'eux, les paramètres qu'il prend. Alors, par définition, les valeurs ayant pour type cet inductif (avec des paramètres donnés) sont les constructeurs avec les valeurs possibles de leurs paramètres. Réciproquement, les valeurs de ce type peuvent être inspectées par une construction de pattern-matching où l'on précise quoi faire pour chacun des constructeurs du type, en utilisant leurs paramètres.

Coq permet la construction de points fixes (fonctions avec appels récursifs) pour traverser les structures récursives. Cependant, la terminaison de tous les programmes est nécessaire à la cohérence de la logique et des points fixes libres permettent directement d'écrire du code qui ne termine pas. Pour empêcher ceci, les points fixes utilisent une garde sur un de leurs arguments appelé argument structurel et ayant un type inductif : comme les valeurs d'un type inductif sont constituées de constructeurs imbriqués en nombre fini, on exige des appels récursifs qu'ils soient faits sur des sous-termes stricts de cet argument.

De même que pour les GADT, il est possible pour les types inductifs d'avoir des paramètres réguliers (qui peuvent prendre toutes les valeurs possibles du type) et des indices (des paramètres qui ont des valeurs spécifiques pour chaque constructeur, et pour lesquels l'inductif n'est pas forcément surjectif).

Ainsi, en énumérant les différentes possibilités et en exigeant les paramètres corrects dans les constructeurs, on parvient à implémenter $\top, \bot, \times, +, \exists, =$ au moyen des types inductifs (avec éventuellement un inductif sans constructeur, ce qui n'est pas interdit).

2.4 Prop, Type et la hiérarchie cumulative de types

En théorie des types, on nomme sorte le type d'un type. Les types de données usuels de Coq appartiennent à la sorte `Type` alors que les propositions appartiennent à la sorte `Prop`. Cette distinction est notamment pertinente pour le mécanisme d'extraction de Coq qui exporte les programmes vers des langages de programmation comme OCaml, Haskell ou C : toutes les valeurs et types dans `Type` sont exportés le plus fidèlement possibles (la théorie des types du langage cible étant moins complexe) et les preuves et propositions sont écrasées vers le type unit sur le principe que les preuves ne font aucun calcul utile. Avec l'extraction, on a alors accès aux algorithmes définis dans `Type` sans s'encombrer de la structure des preuves de correction qui l'accompagnent.

Cela implique notamment une limitation dans le pattern-match de valeurs de sorte `Prop` ; en effet, les valeurs de sorte `Type` ne peuvent pas dépendre d'une valeur qui sera éventuellement écrasée vers l'élément du type unit. Ceci restreint donc le type retour d'un pattern-match sur une preuve à être lui-même dans `Prop`.

Pour construire et typer des valeurs qui dépendent de types, il faut utiliser la construction produit avec le type `Type` (ou `Prop`) comme domaine. Il faut pour cela que ces deux sortes soient en même temps des types et donc qu'elles aient des types. Si on peut donner le type `Type` à `Prop`, donner le type `Type` à `Type` mène à un paradoxe logique (paradoxe de Girard, analogue en théorie des types du paradoxe du barbier de Russel) et casse la cohérence du système.

La solution est de remplacer `Type` en une sorte avec une hiérarchie de niveaux d'univers (`Type@{n}`, $\mathtt{n} \in \mathbb{N}$). On pourra alors donner le type `Type@{n + 1}` à `Type@{n}`. Cette hiérarchie est cumulative en Coq car tout élément de `Type@{i}` l'est aussi de `Type@{j}` si $\mathtt{i} < \mathtt{j}$. Alors, pour simplifier, on se permet de continuer de noter les membres de cette hiérarchie `Type` et on laisse au compilateur le soin de gérer les niveaux d'indices (valeurs de `n`). En interne, les niveaux d'univers sont stockés de manière abstraite pour s'assurer de trouver une instanciation qui soit linéaire (au sens de l'ordre total sur $\mathbb{N}$), mais les détails techniques au-delà ne seront pas utiles pour ce rapport.

2.5 Règles de typage, règles de conversion

Pour la suite, il sera utile de regrouper en une figure l'ensemble des règles de typage de Coq (figure 4). Le prédicat de typage prend la forme $\Sigma; \Gamma \vdash t : T$ où t est le terme, T son type, Γ est le contexte local (liste des arguments abstraits ou lambdas et définitions locales ou let/in sous lesquels on se place) et Σ est l'environnement global (liste des constantes globales et types inductifs définis ainsi que des niveaux d'univers déclarés). Un deuxième prédicat, $\Sigma \vdash \Gamma\ w.f.$, affirme que le contexte Γ est bien formé. On notera s les sortes (`Type@{n}` ou `Prop`).

Bonne formation, variable, constantes

$$\frac{}{\Sigma \vdash [] \ w.f.} \ \varnothing\text{-CON} \qquad \frac{\Sigma \vdash \Gamma \ w.f. \qquad \Sigma; \Gamma \vdash T : s}{\Sigma \vdash \Gamma, (x : T) \ w.f.} \ \text{NEW-VAR}$$

$$\frac{\Sigma \vdash \Gamma \ w.f. \qquad \Sigma; \Gamma \vdash t : T}{\Sigma \vdash \Gamma, (x := t : T) \ w.f.} \ \text{DEF}$$

$$\frac{\Sigma \vdash \Gamma \ w.f. \qquad (x : A) \in^{\dagger} \Gamma}{\Sigma; \Gamma \vdash x : A} \ \text{VAR} \qquad \frac{\Sigma \vdash \Gamma \ w.f. \qquad (x : A) \in \Sigma}{\Sigma; \Gamma \vdash x : A} \ \text{CONST}$$

$$\frac{\Sigma \vdash \Gamma \ w.f.}{\Sigma; \Gamma \vdash \texttt{Type@\{i\}} : \texttt{Type@\{i + 1\}}} \ \text{TYPE} \qquad \frac{\Sigma \vdash \Gamma \ w.f.}{\Sigma; \Gamma \vdash \texttt{Prop} : \texttt{Type@\{1\}}} \ \text{PROP}$$

Règles sur les types

$$\frac{\Sigma; \Gamma \vdash A : s_1 \qquad \Sigma; \Gamma, x : A \vdash B : s_2}{\Sigma; \Gamma \vdash (\Pi x : A \cdot B) : s_3} \ \Pi \qquad (s_3 \text{ est fonction de } s_1 \text{ et } s_2)$$

$$\frac{\Sigma; \Gamma \vdash e : A \qquad \Sigma; \Gamma \vdash B : s \qquad A \leqslant B}{\Sigma; \Gamma \vdash e : B} \ \text{CONV}$$

Règles sur les lambda-termes

$$\frac{\Sigma \vdash \Gamma, (x : A) \ w.f. \qquad \Sigma; \Gamma, x : A \vdash e : B}{\Sigma; \Gamma \vdash (\lambda x : A \cdot e) : \Pi x : A \cdot B} \ \lambda$$

$$\frac{\Sigma \vdash \Gamma, (x := t : A) \ w.f. \qquad \Sigma; \Gamma, (x := t) : A \vdash e : B}{\Sigma; \Gamma \vdash (\texttt{let } x : A := t \texttt{ in } e) : \texttt{let } x : A := t \texttt{ in } B} \ \text{LET}$$

$$\frac{\Sigma; \Gamma \vdash e_1 : \Pi x : A \cdot B \qquad \Sigma; \Gamma \vdash e_2 : A}{\Sigma; \Gamma \vdash (e_1 \ e_2) : B\{x := e_2\}} \ \text{APP}$$

Règles sur les constructions inductives qui ne rentreront pas dans le rapport : IND (types inductifs), CONSTR (constructeurs), MATCH (pattern-matching), (CO)FIX (points fixes, co-points fixes)

† : en pratique, les termes utilisent des indices de de Bruijn pour référer au contexte ; j'utilise $\in^{\dagger}$ pour signifier que l'indice pointe bien vers une entrée (c'est-à-dire que le contexte est suffisamment rempli)

FIG. 2 : Règles de typage de PCUIC

Une règle cruciale est la règle CONV en ceci qu'elle invoque le prédicat de convertibilité (en fait de sous-convertibilité, cf section 3.4). Par convertibilité, on entend la clôture réflexive symétrique transitive contextuelle des cinq formes de réduction suivantes (congruence modulo ces cinq règles) :

$$\frac{}{\Sigma;\Gamma \vdash (\lambda x : A \cdot e)\ e' \equiv e'\{x := e\}}\ \beta \qquad \frac{}{\Sigma;\Gamma \vdash \texttt{let}\ x := t\ \texttt{in}\ e \equiv e\{x := t\}}\ \zeta$$

$$\frac{(c := t) \in \Sigma}{\Sigma;\Gamma \vdash c \equiv t}\ \delta$$

$$\frac{\begin{array}{c}n \text{ est le nombre d'arguments de } f \\ \text{L'argument structurel est bien un constructeur bien appliqué}\end{array}}{\Sigma;\Gamma \vdash (\texttt{fix}\ f\ x_1 \dots x_n := e)\ a_1 \dots a_n \equiv e\{f, x_{1,\dots,n} := (\texttt{fix} f \dots), a_{1,\dots,n}\}}\ \text{FIX}$$

$$\frac{n \text{ est le nombre d'arguments de } C}{\Sigma;\Gamma \vdash \texttt{match}\ C\ a_1 \dots a_n\ \texttt{with}\ C\ x_1 \dots x_n \Rightarrow e \mid \dots\ \texttt{end} \equiv e\{x_{1,\dots,n} := a_{1,\dots,n}\}}\ \iota$$

FIG. 3 : Règles de réduction pour la conversion de PCUIC

2.6 La sorte SProp

Comme expliqué en introduction, `SProp` est une nouvelle sorte ajoutée à Coq, avec une règle de conversion propre. Il peut être utile de considérer un exemple d'application pour cette nouvelle sorte : on va essayer de construire l'addition sur les entiers inférieurs à une borne k.

```
Definition boundednat (k : nat) : Type := { n : nat & n <= k }.
(* Une paire dépendante pour l'entier considéré et la preuve qu'il est inférieur à k *)

Coercion boundednat_to_nat : bounded_nat >-> nat.
(* On peut utiliser les entiers bornés comme des entiers normaux *)

Definition add {k} (n m : boundednat k) (e : n + m <= k) : boundednat k :=
  ( n + m ; e ). (* notation pour les paires dépendantes *)
(* On utilise l'addition des entiers sous-jacents et une preuve donnée en argument *)
```

Cherchons maintenant à prouver l'associativité de cette opération :

```
Lemma bounded_add_associativity k (n m p: boundednat k) e_1 e_2 e'_1 e'_2 :
  add (add n m e_1) p e_2 = add n (add m p e'_1) e'_2.
Proof.
  unfold add; simpl.
  (* À prouver : (n + m + p ; e2) = (n + (m + p); e2') *)
  rewrite Nat.add_assoc.
  reflexivity.
Qed.
```

Après la première étape de la preuve, on voit qu'on a deux égalités à prouver : la première découle directement de l'associativité de l'addition sur les entiers mais la deuxième requiert l'égalité des deux preuves (quelconques) que la somme est bornée. Si la proposition est dans Prop, il faut examiner attentivement les preuves possibles et cette égalité peut ne pas être vraie, alors que si la proposition est dans SProp, l'égalité devient vraie par définition (et la preuve présentée de l'associativité est correcte).

Revenons en maintenant aux propriétés abrstraites. Vis-à-vis des règles de typage, `SProp` se comporte comme `Prop`, sauf pour la règle d'élimination du pattern-match. Si tout examen d'une valeur de `Prop` ou `Type` peut sans souci renvoyer dans `SProp`, comme toutes les valeurs d'un type dans `SProp` sont censées être égales, un pattern-match sur une telle valeur ne peut pas décider d'une valeur dans `Prop` ou `Type`, donc on restreint la sorte de sortie à `SProp`.

Pour les conversions, la sorte `SProp` n'est convertible ou sous-type d'aucune autre sorte (alors qu'en général on autorise `Prop` à être sous-type de tous les `Type`). Enfin, on ajoute la règle de conversion spécifique aux valeurs de sorte `SProp` :

$$\frac{\Sigma;\Gamma \vdash t, u : T \qquad \Sigma;\Gamma \vdash T : \texttt{SProp}}{\Sigma;\Gamma \vdash t \equiv u}\ \textsc{SProp}$$

Il faut noter que cette règle prend en hypothèse des informations de typage sur les termes à comparer, alors que jusqu'à lors on pouvait exprimer la conversion de manière indépendante du typage alors que celui-ci se servait de la conversion. Naïvement, il faudrait maintenant définir mutuellement les deux notions, ce qui rendrait bien plus complexe le développement de preuves sur la théorie. Une autre approche a donc été décrite pour contourner ce problème.

3 Le projet MetaCoq

3.1 Présentation générale

MetaCoq est un projet démarré en 2014, développé par l'équipe Gallinette qui offre la possibilité d'examiner les entrailles du système de Coq, que ce soit pour transformer du code ou, pour ce qui nous intéresse ici, chercher à prouver des propriétés sur sa théorie et des spécifications sur le noyau inclus. Il se décompose en quatre modules [3] :

- `template-coq` manipule les termes avec l'AST (la syntaxe) exact de Coq, il fait l'interface avec Coq pour le mécanisme de *quotations* et *antiquotations* (réification et décodage, traductions de termes depuis et vers Coq) et pour les propriétés prouvées
- `pcuic` décrit la théorie du même nom, avec un AST idéalisé par rapport à template-coq (par exemple, les applications y sont binaires et non n-aires). Il contient toutes les propriétés et les preuves de la théorie
- `safechecker` contient une implémentation de noyau pour travailler sur l'AST et tout le développement pour montrer sa correction et complétude vis-à-vis de PCUIC. Ce noyau peut être extrait vers OCaml et vérifier des fichiers Coq
- `extraction` décrit et vérifie le mécanisme d'extraction

Dans la suite, on se concentrera très largement sur le module `pcuic`, dont l'organisation est la suivante :

3.2 AST et autres structures

Les premiers fichiers décrivent d'abord les sortes et niveaux d'univers, l'AST, les différentes constructions autour des types inductifs, les contextes et environnements. Pour les trois derniers, il y est également défini les différents prédicats de bonne formation de manière paramétrique en les prédicats de conversion et de typage, pas encore définis.

Il faut noter que les termes de Coq (et de MetaCoq) utilisent des indices de de Bruijn pour se référer au contexte : au lieu de stocker les variables par nom, elles le sont par profondeur en nombre de lambdas et let/in (abstractions de fonctions et définitions locales) à traverser. Par exemple, `fun a => let b := _ in b a` sera réifié en `Lambda(a, LetIn(b, _, App(Var 0, Var 1)))` car `b` a besoin de traverser 0 lambda et let/in pour trouver sa déclaration et `a` en traverse 1.

3.3 Opérations et prédicats sur les termes

Les quelques fichiers suivants décrivent les principaux prédicats et opérations spécifiques des termes qui utilisent des indices de de Bruijn : lift, substitution, prédicat de bon scope, mais aussi la substitution des niveaux d'univers. Pour simplifier les preuves des interactions de ces opérations avec le reste de la théorie, des versions idéalisées sont décrites en parallèle : renommage, instanciation, prédicat de scope quelconque. Par exemple, si la substitution remplace les indices de variables à partir de son argument n par le contenu de sa liste argument l, l'instanciation remplace toutes les variables par la valeur que prend la fonction argument en l'indice donné.

Déjà, des lemmes d'interactions entre toutes ces opérations sont établis et prouvés (notamment la simulation des opérations effectives par les versions idéalisées).

3.4 Conversion et cumulativité, spécificatives et algorithmiques

Vient ensuite la définition du prédicat de conversion/cumulativité. Précisons d'abord que la cumulativité est simplement la conversion où on autorise certains niveaux d'univers à être inférieurs et non simplement équivalents. Ensuite, ce prédicat est défini deux fois pour les mêmes raisons que les opérations du paragraphe précédent. On a en premier lieu la version spécificative, idéalisée, qui correspond à la description qui est faite en section 2.5, c'est notamment une congruence.

En deuxième lieu, la version algorithmique se restreint à une description décidable de la conversion, en utilisant les étapes telles que le noyau les applique : réduction et égalité. Ce qu'on appelle réduction est la clôture réflexive transitive des étapes de réduction (`red1`), qui sont elles la clôture par contexte syntaxique des règles listées en figure 3 (β, δ, ζ, ι, FIX). On cherche surtout à réduire les termes jusqu'à arriver à une forme normale, pour laquelle aucune règle de réduction n'est applicable, en s'appuyant sur une propriété souhaitée de confluence, qui établit que si plusieurs règles sont applicables, le choix ne change pas la valeur de la forme normale du terme.

Après avoir réduit les deux termes à comparer, on va regarder s'ils sont égaux pour l'égalité de termes, qui requiert que les termes aient la même forme d'AST et ne diffèrent que par les noms des variables utilisées (α-convertibilité) et par des niveaux d'univers, tant qu'ils sont aussi convertibles (ou inférieurs dans le cas du prédicat de cumulativité).

On observe alors que la version spécificative inclut toutes les conversions de la réduction et de l'égalité, elle est donc plus générale que la version algorithmique.

3.5 Typage, interactions

Une fois la cumulativité définie, on a tous les ingrédients pour définir les règles de typage ; la définition est fidèle à la description que j'en donne figure 4.

Plusieurs fichiers servent ensuite à décrire les interactions entre tous les concepts définis précédemment : comment interagissent d'un côté environnements (notamment leur affaiblissement, c'est-à-dire l'ajout de définitions), contextes (et leur affaiblissement, l'ajout de variables et définitions locales), les opérations et prédicats sur les termes, avec de l'autre réduction, égalité, cumulativité et typage.

On y introduit notamment des prédicats sur les termes à substituer et les contextes utilisés comme hypothèses pour que cumulativité et typage puissent être préservés (par exemple pour la substitution, une variable abstraite de relevance $*$ doit être substituée par un terme de relevance $*$).

3.6 Grands théorèmes

Le dernier (grand) groupe de fichiers prouve les différentes propriétés importantes de PCUIC. Le premier théorème est la confluence de la réduction, prouvée en passant par une définition de la réduction parallèle. Grâce à elle, on prouve ensuite un lemme d'inversion (du typage), puis la validité du typage (si $t : T$ alors il existe une sorte qui est le type de T).

Ensuite, on prouve *Subject Reduction*, c'est-à-dire que si $t : T$ et t se réduit en t', alors $t' : T$, et la principalité du typage qui affirme que si t est typable, il en existe un type T minimal pour la cumulativité parmi les types de t.

Le prochain grand théorème est la normalisation forte du système. Malheureusement, d'après des résultats relatifs au théorème d'incomplétude de Gödel, il est impossible de la prouver complètement en Coq. S'il peut exister des moyens de contourner l'impossibilité en utilisant plus de niveaux d'univers dans la preuve que le nombre de niveaux pour lequel on prouve le résultat, aucune preuve n'a été trouvée à ce jour, et le résultat est donc postulé en tant qu'axiome pour les résultats suivants. Le dernier théorème prouvé est *Progress*, qui dit que tout terme clos typable se réduit en une valeur (constructeur, inductif, lambda, produit, sorte ou (co-)point fixe), elle-même non réductible.

4 Marques de relevance, relevance de termes

4.1 Théorie initiale

Étudions à présent comment implémenter la sorte `SProp` et surtout sa règle de conversion de termes. Comme expliqué en section 2.6, on ne souhaite pas faire dépendre la définition de la conversion de celle du typage ; il nous faut donc une autre manière de caractériser les termes dont la sorte est `SProp`. L'approche proposée dans [5] utilise un critère syntaxique appelé relevance : on peut déterminer uniquement en examinant un terme si sa sorte est `SProp` ou non, tant qu'on conserve l'information pour les éléments du contexte. Observons le critère proposé pour un langage un peu moins expressif que celui de Coq, qu'on notera $\Sigma; \Gamma \vdash t *$ et qui se lit « t a la relevance $*$ ».

$$\frac{}{\Sigma \vdash [] \ w.f.} \ \varnothing\text{-CON} \qquad \frac{\Sigma \vdash \Gamma \ w.f. \qquad \Sigma; \Gamma \vdash T : \texttt{SProp}}{\Sigma \vdash \Gamma, (x : T \ \texttt{irrelevant}) \ w.f.} \ \text{NEW-IRR}$$

$$\frac{\Sigma \vdash \Gamma \ w.f. \qquad \Sigma; \Gamma \vdash T : s \qquad s \neq \texttt{SProp}}{\Sigma \vdash \Gamma, (x : T \ \texttt{relevant}) \ w.f.} \ \text{NEW-REL}$$

$$\frac{\Sigma \vdash \Gamma \ w.f. \qquad (x : A *) \in^{\dagger} \Gamma}{\Sigma; \Gamma \vdash x *} \ \text{VAR}$$

$$\frac{\Sigma \vdash \Gamma, (x : A \dagger) \ w.f. \qquad \Sigma; \Gamma, (x : A \dagger) \vdash t *}{\Sigma; \Gamma \vdash (\lambda (x : A \dagger) \cdot t) *} \ \lambda \qquad \frac{\Sigma; \Gamma \vdash t *}{\Sigma; \Gamma \vdash t \ t' *} \ \text{APP}$$

$$\frac{t \text{ est un produit, une sorte ou un inductif}}{\Sigma; \Gamma \vdash t \ \texttt{relevant}} \ \text{OTHER}$$

FIG. 4 : Règles de relevance minimales

La principale justification de la correction de ces règles est le fait qu'à la fois `SProp` et la classe des sortes *relevant* (l'ensemble {`Prop`, `Type`}) sont tous deux clos par produit (si T : `SProp`, alors $(\Pi(a : A) \cdot T)$: `SProp`) et par « destruction de produit » (réciproque de la proposition précédente), car ça justifie respectivement les règles λ et APP.

Ma première tâche a été d'étendre cette définition au reste de l'AST de Coq, ce qui se fait naturellement :

- Pour les définitions locales, on peut calculer syntaxiquement la relevance de la définition et calculer celle du corps dans le nouveau contexte enrichi
- Pour les constantes, on fait comme pour les variables en supposant connues les relevances dans l'environnement global
- Pour les constructeurs, on fait comme pour les constantes ; on peut noter que la relevance dépend uniquement de la sorte dans laquelle est située l'inductif construit, elle ne dépend donc pas du constructeur au sein d'un inductif
- Pour les points fixes, comme la fonction renvoyée a le même type que la fonction que le point fixe prend en argument et que cette dernière est dans le contexte, il suffit d'utiliser la relevance déjà présente dans le contexte
- Pour les pattern matchs, on n'a aucun moyen de savoir syntaxiquement la sorte de retour, donc on la suppose donnée

Cette approche repose donc en grande partie sur la collecte et conservation des marques de relevance pour toutes les entrées de contextes, des constantes de l'environnement, des inductifs pour leurs constructeurs, mais aussi la donnée par avance des relevance dans les arguments des lambdas et pour les pattern matchs.

4.2 Ajout des marques de relevance

Dans la suite, on dira que la relevance de sorte de s est `irrelevant` si $s =$ `SProp` et `relevant` sinon ; on dira que la relevance de type de $T : s$ est celle de sorte de s (qui ne dépend en fait pas du choix de s). Écrit en Coq, cela donne les définitions suivantes (qui sont dans MetaCoq) :

```
Definition relevance_of_sort (s: sort) : relevance :=
  match s with
  | SProp => Irrelevant
  | Prop | Type _ => Relevant
  end.

Definition isSortRel (s: sort) (r: relevance) := relevance_of_sort s = r.

Definition isTypeRel Σ Γ (T: term) (r: relevance) :=
  { s: sort | Σ ; Γ |- T : s × isSortRel s r }.
```

(Pour `isTypeRel`, j'ai repris la définition existante du prédicat `isType` que j'ai enrichi de l'hypothèse `isSortRel`). Je vais aussi reprendre l'exemple des entiers bornés pour illustrer le propos.

La première étape du développement que j'ai effectué est l'ajout et les preuves de conservation des informations sur les marques de relevance disséminées dans les structures et termes. On enrichit les différentes définitions de « bonne formation » des correspondances suivantes, qui dérivent de la définition de ce que la relevance représente :

- Dans un contexte contenant $(a : T\ *)$, $*$ est la relevance de type de T, c'est-à-dire qu'on remplace `isType Σ Γ T` par `isTypeRel Σ Γ T *` (de même si a a une définition, on se souciera de la relevance de terme de a dans un deuxième temps)
- Dans un environnement contenant $(c : T\ *)$, $*$ est la relevance de type de T (cf. ci-dessus)
- Dans un environnement où l'inductif I est de sorte s, la relevance stockée pour les constructeurs de I est égale à la relevance de sorte de s

De même, on enrichit des correspondances suivantes le prédicat de typage :

- Dans une abstraction $\lambda(a : T\ *) \cdot e$, $*$ est la relevance de type de T
- Dans une définition locale `let` $a : T\ * := e_1$ `in` e_2, $*$ est la relevance de type de T
- Dans la construction `match` c `return` $T\ *$ `with .. end`, $*$ est la relevance de type de T

Cela correspond dans notre exemple à ajouter et interpréter le fait que les preuves arguments e_1, e_2 que les entiers sont bornés sont bien irrelevant, que ce soit dans les abstractions à l'extérieur et dans le contexte quand on y fait passer les preuves (`intros.` dans une preuve).

Cette partie du développement s'est passée sans grand accroc vu que j'ai simplement ajouté des égalités qui interagissaient avec une petite partie du reste de MetaCoq. Les seules interactions notables ont menés aux deux lemmes suivants :

- ```
  Lemma isSortRel_univ_subst s r u : isSortRel s r -> isSortRel s@[u] r.
  ```

  (le calcul de relevance est compatible avec la substitution des niveaux d'univers, car ils ne changent pas la sorte)

- ```
  Lemma isSortRel_leq s s' r : sort_leq s s' ->
    isSortRel s r <-> isSortRel s' r.
  ```

  (le calcul de relevance est compatible avec la comparaison de sortes, car `SProp` est incomparable avec les autres sortes)

En parallèle, il m'a fallu faire un changement sur la définition de `sort_leq` sur la branche principale de MetaCoq pour rendre `isSortRel_leq` prouvable, sinon une option dans Coq le rendait potentiellement faux.

Enfin, j'ai pu ajouter les vérifications à faire dans le noyau pour vérifier les correspondances ajoutées : dans la mesure où les marques de relevance sont des entrées pour le typage, on n'a pas de garantie qu'elles soient correctes et il faut donc les vérifier.

4.3 Prédicat sur les termes

La deuxième étape a été d'ajouter le prédicat pour déterminer la relevance syntaxique des termes telle que présentée en section 4.1. En Coq, il sera noté `isTermRel Σ Γ t r` par symétrie avec les précédents. On peut alors l'ajouter au prédicat de bonne formation des contextes pour les définitions locales.

Cela correspond dans notre exemple à pouvoir calculer qu'un entier n est relevant, une preuve e est irrelevant et un entier borné $(n; e)$ est lui aussi relevant car l'inductif des paires dépendantes est dans `Type`.

Contrairement à la première étape, les calculs sur les termes interagissent plus largement avec les opérations dans MetaCoq, il faut donc prouver plus de lemmes de compatibilité :

- `isTermRel_univ_subst` (le prédicat n'examine jamais les niveaux d'univers)
- `isTermRel_extends` (compatibilité avec l'affaiblissement d'environnement)
- `isTermRel_rename` (sous une hypothèse de compatibilité de la fonction de renommage)
- `isTermRel_weaken` (compatibilité avec l'affaiblissement de contexte)
- `isTermRel_inst` (sous une hypothèse de compatibilité de la fonction d'instanciation)
- `isTermRel_compare` (compatibilité avec l'égalité de termes)
- `isTermRel_red` (compatibilité avec la réduction

Grâce à ces lemmes, j'ai pu prouver un théorème de correction et complétude du prédicat syntaxique vis-à-vis du critère sur les sortes (`wf` est le prédicat de bonne formation sur les environnements) :

```
Theorem isTermTypeRel Σ Γ (t T : term) (r : relevance) :
  wf Σ -> Σ ; Γ |- t : T ->
  isTermRel Σ Γ t r <-> isTypeRel Σ Γ T r.
```

Ceci permet de relier le prédicat syntaxique et la relevance de type, ce qui est utile dans le noyau. En effet, cela permet de rendre certains tests plus efficaces : pour les définitions locales, plus besoin de regarder la sorte du type, il suffit d'examiner le terme.

4.4 Nouvelle règle de conversion, nouveau prédicat de réduction

Après avoir mené à terme les deux premières vagues d'ajouts initiaux, il vint le temps d'ajouter la nouvelle règle de conversion :

```
conv_irrelevant :
  isTermRel Σ Γ t Irrelevant -> isTermRel Σ Γ u Irrelevant ->
  compare_term Σ Γ t u
```

C'est cet ajout qui permet de montrer l'égalité des preuves pour l'associativité dans notre exemple.

Cependant, pour prouver la confluence du système, MetaCoq s'appuie sur un lemme de commutation entre réduction et égalité de terme :

```
Lemma red1_eq_term Σ Γ (u v u' : term) :
  eq_term Σ Γ u u' -> red1 Σ Γ u v ->
  { v' : term & eq_term Σ Γ v v' × red1 Σ Γ u' v' }.
```

(La notation `{ v : term & P }` désigne le type des paires dépendantes d'un terme `v` qui vérifie la propriété P). Cependant, avec $u = (\lambda(x\ \texttt{Irrelevant}) \cdot x)\ a$ pour a une valeur irrelevant, on a d'un côté $u \equiv a$ car ils sont tous deux irrelevant, de l'autre $u \rightsquigarrow a$ par β-réduction. On ne peut donc pas trouver de v' qui soit égal à a et réduit de a, le lemme doit être corrigé.

La correction va consister à restreindre la réduction aux termes relevant, l'idée étant que les termes irrelevant seront pris en charge par la partie égalité de terme via la règle spécifique. On définit donc un nouveau prédicat de réduction qui vérifie que le rédex est relevant et qu'à tous les niveaux de contexte on est relevant. On ne peut pas simplement le vérifier pour le terme complet car on a le contre-exemple $u_2 = b\ u$ avec b relevant et u l'exemple ci-dessus (il se réduit et se compare comme u, même s'il est relevant).

Cela implique alors de corriger tous les lemmes à propos de l'égalité d'un côté (et de perdre ceux qui affirmaient que des termes égaux avaient forcément la même structure), corriger les lemmes à propos de la réduction de l'autre.

De plus, comme la réduction inspecte arbitrairement des sous-termes, il m'a semblé judicieux de renforcer le prédicat `isTermRel` pour qu'il passe au sous-terme, c'est-à-dire qu'il vérifie récursivement que tous les sous-termes acceptent une relevance (ce n'est pas le cas de ceux pour qui les marques de relevance sont fausses notamment). Il devient alors un prédicat intermédiaire entre le prédicat de bon scope et celui d'être typable, même si la deuxième implication nécessite `isTermTypeRel` qui a besoin des grands théorèmes pour être vrai. Il faut alors reprouver tous les lemmes de compatibilité du prédicat syntaxique.

Je me suis arrêté à la preuve de `red1_eq_term` qui s'avère particulièrement ardue.

4.5 Développements futurs

Après la preuve de la confluence du système, je ne m'attends pas à autant de difficulté pour avancer. Il s'agirait de reprendre les preuves des grands théorèmes en faisant probablement des transformations similaires à celles qui auront été faites pour la confluence,

à une moins grande échelle. À terme, l'objectif est de pousser le développement jusqu'au noyau et d'y ajouter la règle de conversion.

Ce développement peut aussi être généralisé en enrichissant l'information transmise dans les marques de relevance. Dans un objectif de développement du polymorphisme de sorte qui vise à ajouter de nouvelles sortes à Coq, la marque de relevance pourrait devenir une marque plus générale de sorte, avec tout le flux d'information et de preuves déjà établi.

5 Conclusion

Ce stage m'a permis d'explorer en profondeur la riche théorie des types de Coq et toutes ses propriétés, en les comparant au gré des discussions avec mes collègues à celles des assistants de preuve adjacents, notamment Agda et Lean. Ceci complémente mon stage de M1 sur des assistants de preuve avec des théories des types moins riches, Isabelle et Dedukti/Lambdapi.

Sur un autre plan, ce stage a été très pratique avec 18000 lignes changées d'après git ; j'ai sans cesse été confronté à Coq pour tenter de lui faire accepter les preuves que j'écrivais et j'ai grandement apprécié travailler de cette manière.

Enfin, sur le plan personnel, ces 4 mois à Nantes ont été particulièrement agréables, le cadre et l'équipe Gallinette ont permis un stage épanouissant et la tenue de la conférence TYPES 2022 sur place m'a ouvert à tout l'éventail de recherche dans le domaine petit mais en forte expansion de la théorie des types.

Références

[1] *Welcome ! | The Coq Proof Assistant - Coq.Inria.Fr/*. URL : https://coq.inria.fr/ (visité le 20/08/2022).

[2] *MetaCoq*. MetaCoq. URL : https://metacoq.github.io/ (visité le 20/08/2022).

[3] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau et Théo Winterhalter. « Coq Coq Correct ! Verification of Type Checking and Erasure for Coq, in Coq ». In : *Proceedings of the ACM on Programming Languages* 4 (POPL 20 déc. 2019), 8 :1-8 :28. DOI : 10.1145/3371076. URL : https://doi.org/10.1145/3371076 (visité le 20/08/2022).

[4] Gaëtan Gilbert. « A Type Theory with Definitional Proof-Irrelevance ». Theses. Ecole nationale supérieure Mines-Télécom Atlantique, déc. 2019. URL : https://tel.archives-ouvertes.fr/tel-03236271 (visité le 20/08/2022).

[5] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau et Nicolas Tabareau. « Definitional Proof-Irrelevance without K ». In : *Proceedings of the ACM on Programming Languages* 3 (POPL 2 jan. 2019), p. 1-28. ISSN : 2475-1421. DOI : 10.1145/3290316. URL : https://dl.acm.org/doi/10.1145/3290316 (visité le 20/08/2022).