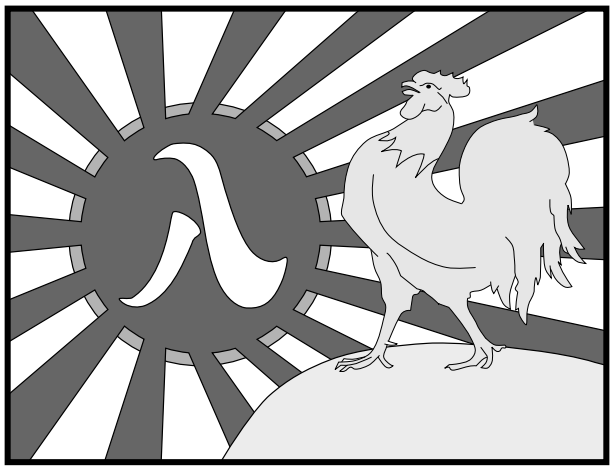
# Bisimulations d'un lambda-calcul polymorphe impératif en Coq

Pierre-Marie Pédrot

ENS Lyon

Stage de recherche de M1 : Juin–Juillet 2010

Université de Nagoya



Encadrant : Jacques Garrigue

— *Pourquoi le coq a-t-il traversé la route ?*
— `Admitted`.

Sacha Guitry à propos d'un célèbre
assistant à la preuve.

## Remerciements

Je tiens à remercier en premier lieu Jacques Garrigue, pour m'avoir trouvé un sujet de stage prenant, ainsi que pour avoir eu le courage de m'encadrer pendant deux mois. Je le remercie aussi vivement pour son soutien linguistique et culturel en milieu hostile, sa bienveillance allant toujours de pair avec la bonne humeur, sa sapience sans bornes en OCaml ainsi que sa capacité surnaturelle à trouver de la chartreuse à Nagoya.

Je remercie les divers combinis que j'ai fréquentés pour m'avoir fourni en doses astronomiques de thé, ainsi qu'en nourriture toute prête, ce qui représente toujours du temps de gagné quand on passe sa nuit devant un écran. Je remercie aussi les corbeaux tapageurs, les chats errants, les cigales géantes, et plus généralement tous les animaux qui m'ont distrait par leurs cris bruyants.

Enfin, je voudrais remercier tous ceux qui m'ont soutenu durant ce long voyage au Japon par voie épistolaire, comme la confraternelle liste M1IF, les collègues (et, pour certains, futurs colocataires) de la liste Coqtail, le perfide et démoniaque chat-garou, ainsi que bien entendu, la fameuse brochette d'incapables de la liste Ours en Jetpack.

# Introduction

Alors que nous vivons la protohistoire de la preuve assistée par ordinateur, nous sommes assurément à un tournant de l'histoire tant des mathématiques que de l'informatique. D'un côté, les mathématiciens commencent à numériser le vaste corpus de démonstrations que fournit leur discipline ; de l'autre, les informaticiens sont toujours plus nombreux à vouloir certifier leurs programmes via des preuves de correction vérifiées par la machine.

L'un des avantages de la preuve assistée par ordinateur vient du fait qu'elle permet de traiter automatiquement des objets suffisament énormes pour décourager le plus téméraire des mathématiciens. Témoin, par exemple, la désormais célèbre démonstration du non moins célèbre théorème des quatre couleurs en Coq, qui requiert l'étude exhaustive de plusieurs centaines de sous-cas. Même dans le cas de démonstrations traitant de structures plus petites, celles-ci peuvent se révéler assez grosses pour que l'auteur de la preuve laisse passer des points importants, voire écrive une démonstration fausse, en balayant nonchalament la difficulté sous couvert de *trivialité*.

Hélas, le prix à payer en est assez conséquent. De nombreux points qui sont réellement triviaux se retrouvent à représenter la majeure partie de la démonstration. Il faut aussi ajouter à cela que lors de la formalisation d'une démonstration papier, de nombreuses difficultés se cachent là où on s'y attend le moins.

L'objet de ce stage se trouve à mi-chemin entre une utilisation purement théorique et une autre plus pratique de l'assistant à la preuve bien connu, Coq [3]. Il s'agissait d'implémenter dans ce langage les techniques de bisimulations d'un langage plutôt riche décrites par Eijiro Sumii dans un article récent [7]. Le côté théorique du travail consistait à prouver que les démonstrations de cet article étaient correctes ; le côté pratique consistait à utiliser la bibliothèque ainsi écrite pour décrire des programmes et prouver leur équivalence par bisimulation. Faute de temps, nous avons surtout insisté sur le premier point [1].

En tant que telles, les bisimulations ne sont pas le sujet central de ce travail ; il s'attache principalement aux méthodes utilisées pour implémenter ces objets en Coq. Pour plus d'informations sur les techniques de preuve d'équivalence de programmes, le lecteur intéressé pourra se rapporter avantageusement à la bibliographie de l'article de Sumii.

On retrouve dans ce travail les avantages et inconvénients de la formalisation informatique cités précédemment, à savoir l'étude en profondeur de points pas si triviaux, habituellement mis de côté. Cette formalisation a permis de sortir de l'ombre de nombreux détails passés sous silence dans l'article original, par manque de place ; elle a aussi mis en exergue quelques erreurs d'inattention du même article, et révélé un problème légèrement plus sérieux qui n'avait jusque là pas été remarqué, sans doute à cause de la taille des définitions.

Le code produit en huit semaines atteint presque les 10.000 lignes [2]. On peut le trouver (et y contribuer) sur la plateforme de développement SourceForge, sous le nom de Simpou$\lambda$et [6]. La majorité du code est sous license WTFPL [3].

1. Mais Jacques Garrigue a contribué au second point, cf. le dossier `Examples`.
2. Il est cependant difficile de juger de la complexité d'une preuve en Coq par rapport à sa taille.
3. Plus connue sous son nom complet de « *Do What The Fuck You Want To Public License.* »

# 1 Généralités

## 1.1 Langage

Comme d'habitude, le langage étudié se base sur l'un des plus célèbres modèles de la sémantique, nommément le $\lambda$-calcul. Cependant, contrairement aux langages minimalistes généralement utilisés pour isoler les particularités de chaque extension du vénérable ancêtre, le langage décrit dans l'article de Sumii est particulièrement riche. Au vu de la taille d'icelui, nous ne faisons que donner ici une idée de la nature de ce langage ; pour une description légèrement plus détaillée, confère l'article original, et pour une description complète, se référer aux définitions du code Coq, qui a volontairement été écrit lisiblement pour cela.

Le but plus ou moins avoué est de pouvoir représenter les langages ML ; à ce titre, ce « $\lambda$-calcul polymorphe avec références générales » (appelé $\Lambda^{\Uparrow}$ par la suite) est un sur-ensemble de ML. À la différence de ML, il faut cependant noter qu'il est explicitement typé à la Church. Globalement, on peut le décrire comme suit :

- Un noyau de système $\mathbb{F}$ à la Church ;
- Des types produits à $n$ composantes, pour $n \geq 0$ ;
- Des types existentiels, pour encoder les modules de ML ;
- Des références ML-like, incluant la comparaison de locations [4] ;
- Les exemples font aussi abondamment usage de deux types de base, les booléens et les entiers [5], et du sucre syntaxique afférent.

$$\tau ::= \alpha \mid \sigma \to \tau \mid \forall \alpha.\tau \mid \exists \alpha.\tau \mid \tau_1 \times \ldots \times \tau_n \mid \tau \ \texttt{ref}$$

FIGURE 1 – Types de $\Lambda^{\Uparrow}$

Les jugements de typage sont de la forme $S, \Sigma, \Gamma \vdash t : \tau$ où $S$ est un ensemble de variables de types modélisant les variables de type libres apparaissant dans le reste du jugement, $\Sigma$ est un ensemble de paires $(\ell : \sigma)$ typant les locations de $t$ et $\Gamma$ est un ensemble de paires $(x : \sigma)$ typant les variables libres de $t$.

Du point de vue de la sémantique, $\Lambda^{\Uparrow}$ est en appel par valeur de gauche à droite, toujours dans la tradition des ML. Une remarque importante pour les bisimulations, l'allocation de mémoire est non-déterministe et il n'y a pas de *garbage collection*, c'est-à-dire que les locations, une fois allouées, survivent tout au long de l'exécution du programme.

Les programmes sont de la forme $s \rhd t$ où $t$ est le corps du programme et $s$ est un ensemble de paires $(\ell : v)$ représentant la mémoire allouée ($v$ dénote une valeur).

Pour des raisons pratiques, l'implémentation s'écarte un peu de cette description, et nous la nommerons $\Lambda^{\Uparrow}_{\mathtt{c}}$. Un premier changement mineur consiste à restreindre les types produits au cas $n = 0$ (`unit`) et $n = 2$ (paire), et à ne pas avoir de types de base. Cette modification est à peine cosmétique et ne change en rien l'expressivité du langage.

Une modification plus importante, mais qui simplifie grandement les preuves, consiste à effacer les types des termes, ce qui conduit à une syntaxe à mi-chemin entre les présentations à la Church

4. Similaire à l'opérateur == d'OCaml.
5. Qui sont de toute façon encodables via le codage imprédicatif de système $\mathbb{F}$.

| | $\Lambda^{\Uparrow}$ | $\Lambda_{\mathtt{c}}^{\Uparrow}$ |
|---|---|---|
| Abstraction | $\lambda x : \sigma \,.\, t$ | $\lambda x.t$ |
| Application de type | $t[\sigma]$ | $t[\cdot]$ |
| Abstraction existentielle | $\mathtt{pack}\ (t, \tau)\ \mathtt{as}\ \exists\alpha.\sigma$ | $\mathtt{pack}\ t$ |

FIGURE 2 – Effacement de type dans les termes

et à la Curry. Cela évite de faire des substitutions explicites[6] à de nombreux endroits. Cependant, ce changement n'est pas sans provoquer une conséquence fâcheuse : on perd alors la décidabilité du typage [4], ce qui complique les preuves par réflexivité pour l'utilisateur de la bibliothèque.

### 1.2 Bisimulations

La définition des relations de bisimulations décrites par Sumii est propre aux spécificités de $\Lambda^{\Uparrow}$. Tout comme pour la description de $\Lambda^{\Uparrow}$, nous ne faisons ici que d'insister sur les particularités de ces relations, et invitons le lecteur à lire l'article original pour avoir le détail complet.

Tout d'abord, $\Lambda^{\Uparrow}$ étant en appel par valeur, les relations de simulation sont bicéphales et sont définies par la donnée d'une relation sur les programmes $s \rhd t$ et d'une relation sur les états mémoire $s$, qui représente les programmes ayant fini de s'exécuter (le corps du programme $t$, en tant que valeur, n'est pas significatif, puisqu'il ne peut plus interagir dans la bisimulation).

Du point de vue de l'implémentation en Coq, cela implique de considérer les relations de bisimilarité comme des paires de relations, ce qui est pour le moins peu naturel et force quelques lourdeurs bureaucratiques dans les définitions.

Par ailleurs, les bisimulations sont implicitement typées pour garantir leur correction. La correction des types existentiels requiert de même de conserver dans la relation un ensemble d'association $\Delta$ entre les variables de types, représentant les types existentiels abstraits, et les types effectifs implémentant cette abstraction.

Cela s'est révélé source d'ennuis renouvelés dans l'implémentation, car la nécessité du typage n'est pas évidente selon les endroits. De même, le passage explicite de $\Delta$ en argument n'a pas semblé nécessaire dans un premier temps, et cela a permis de mettre à jour une erreur dans les définitions concernant les types existentiels.

Les relations $X$ se présentent donc comme un ensemble de tuples des formes suivantes :

$$\begin{array}{cc} (\Delta, \mathcal{R}, s_1 \rhd t_1, s_2 \rhd t_2, \tau) & \text{pour les programmes} \\ (\Delta, \mathcal{R}, s_1, s_2) & \text{pour les valeurs} \end{array}$$

où $\mathcal{R}$ est un ensemble de triplets $(v_1, v_2, \sigma)$ qui dénotent la connaissance du contexte sous la forme de valeurs équivalentes, avec un typage *ad-hoc*.

L'article original décrit alors les conditions pour que $X$ soit une bisimulation, ainsi que trois approximations :

6. Lire : pénibles.

1. bisimulation à réduction près
2. bisimulation à contexte près
3. bisimulation à allocation près

et les clôtures correspondantes. Les résultats du papier sont que ces clôtures sont correctes (i.e. si $X$ est une bisimulation à $p$ près, alors sa $p$-clôture est une bisimulation), et que d'autre part la notion de bisimulation choisie correspond à une certaine forme d'équivalence observationnelle.

Nous avons implémenté la preuve de correction du premier cas, et la preuve de correction du troisième cas devrait être assez directe, l'apparatus nécessaire étant déjà présent.

Par contre, la preuve de correction des bisimulations à contexte près s'est révélé être, conformément à nos attentes, un vrai supplice, et a soulevé des problèmes d'implémentation qui sont encore à ce jour ouverts. Ce sont en effet des définitions qui sont intrinsèquement difficiles à exprimer en Coq (ou dans n'importe quel autre assistant à la preuve, du reste) ; cf. les sections suivantes pour davantage d'explications. En l'état nous avons des morceaux de preuve.

# 2 Le $\lambda$-calcul en Coq

## 2.1 Problématique

Avant toute considération technique, il faut se rendre à l'évidence : le $\lambda$-calcul et ses dérivés, incluant notre $\Lambda^{\Uparrow}$, sont un outil fondamental dans certaines branches des mathématiques. Avoir une formalisation qui tienne la route est donc d'une importance primordiale.

Malheureusement, et de longue date, l'on sait que le $\lambda$-calcul traîne avec lui de nombreuses propriétés nécessaires sous-entendues, que l'on pourrait baptiser sous le nom générique de « cauchemar des variables liées. » En d'autres termes, on travaille toujours à $\alpha$-conversion près pour éviter la capture de variables libres lors de la substitution.

Le problème se pose doublement pour $\Lambda^{\Uparrow}_{\mathsf{c}}$, car en tant que langage du second ordre, les types peuvent aussi contenir des variables liées.

On pourrait par exemple s'astreindre à suivre la convention de Barendregt, mais Coq et les quotients ne font pas bon ménage [7] ; et quoi qu'en disent les zélotes, les sétoïdes sont d'une lourdeur bureaucratique conséquente.

**Définition 1.** Un terme $t$ suit la convention de Barendregt si pour tout sous-terme $u$ de $t$ :

$$\mathrm{fv}(u) \cap \mathrm{bv}(u) = \emptyset$$

Une autre voie, connue depuis longtemps, consiste à se débarrasser des variables en les remplaçant par les indices de De Bruijn.

**Définition 2.** Le $\lambda$-calcul en indices de De Bruijn est défini par la syntaxe suivante :

$$t ::= n \in \mathbb{N} \mid t(u) \mid \lambda.t$$

L'indice $n$ représente la variable liée du $n$-ième $\lambda$ rencontré en partant de la variable et en remontant dans le terme.

C'est cette technique que nous avons choisi d'utiliser, à quelques remaniements près, car la représentation de De Bruijn ne permet pas de manipuler aisément les termes ouverts, et méconnaît la notion de variable fraîche, voire de variable tout court.

7. Voir à ce propos la note instructive de Loïc Pottier sur le sujet.

## 2.2 Représentation localement anonyme cofiniment quantifiée (LNCFQR)

Cette représentation est relativement récente, et a été inventée en réponse à l'engouement naissant pour les assistants à la preuve. Quoiqu'elle ne soit pas fondamentalement innovante, elle a l'astuce de combiner plusieurs représentations du $\lambda$-calcul, et a surtout l'avantage de fonctionner plutôt agréablement en Coq. Pour une description complète de la chose, confère l'article non encore publié de Charguéraud [1].

Toujours pour des raisons de place, on présente ici le cas du $\lambda$-calcul simplement typé ; l'adaptation à $\Lambda_{\mathsf{c}}^{\Uparrow}$ est directe, et les lemmes énoncés ici y restent valides. Le principe de la LNCFQR peut se résumer ès deux points qui forment son nom :

**Anonymat local :** Les variables liées sont représentées en indices de De Bruijn, les variables libres par de véritables variables ;

**Quantification cofinie :** La notion de fraîcheur est locale à un sous-terme, et est relative à un ensemble fini $L$ hors duquel on peut choisir une variable fraîche.

Il devient alors nécessaire de séparer les prétermes des termes, ces derniers étant correctement formés par rapport à la syntaxe du $\lambda$-calcul.

**Définition 3.** Un *préterme* du $\lambda$-calcul est un élément généré par la syntaxe suivante :

$$t ::= n \in \mathbb{N} \mid x \in \mathcal{V} \mid t(u) \mid \lambda.t$$

soit, en Coq :

```
Inductive trm := bvar : nat → trm | fvar : var → trm | app : trm → trm → trm | abs : trm → trm.
```

Nous n'insistons pas ici sur la nature de l'ensemble des variables libres $\mathcal{V}$ ; cf. la section sur les choix d'implémentation. On suppose en outre disposer d'une opération $t\{x\}$ appelée « ouverture » qui remplace la variable liée de rang 0 par la variable libre $x$ dans le préterme $t$, ou, en termes équationnels :

$$(\lambda.t)(x) \to_\beta t\{x\}$$

**Définition 4.** On définit inductivement le prédicat $\mathcal{T}$ « être un terme » sur les prétermes de la façon suivante :

$$\frac{x \in \mathcal{V}}{\mathcal{T}(x)} \qquad \frac{\mathcal{T}(t) \qquad \mathcal{T}(u)}{\mathcal{T}(t(u))} \qquad \frac{\text{il existe } L \text{ fini t.q. } \mathcal{T}(t\{x\}) \text{ pour tout } x \notin L}{\mathcal{T}(\lambda.t)}$$

La quantification cofinie permet de choisir n'importe quelle variable fraîche pour ouvrir, sauf pour un nombre *fini* de cas problématiques, qui peuvent contenir entre autres les variables libres du terme considéré.

Cette définition alambiquée de $\mathcal{T}$ est pourtant équivalente à une propriété triviale énoncée dans le lemme ci-dessous. Cependant, cette présentation est rendue nécessaire pour rester en cohérence avec la notion de fraîcheur qui sera requise pour le typage.

**Lemme 1.** *Pour tout préterme $t$, $t$ est un terme ssi toutes les variables indicées de $t$ sont couvertes par un $\lambda$ (i.e. pour tout sous-terme de $t$, le degré des variables liées est strictement inférieur au nombre de $\lambda$ traversés).*

Grâce à ce lemme, on tire une caractérisation algorithmique du prédicat $\mathcal{T}$ qui n'apparaît pas à première vue dans la première définition. Elle s'est d'ailleurs révélée utile dans notre implémentation car elle permet des preuves par réflexivité et fournit un élégant principe d'induction. Il faut toutefois noter que ce lemme n'est pas trivial à démontrer en Coq et repose sur l'emploi d'un ordre bien fondé sur les termes.

**Définition 5.** La relation de typage $\Gamma \vdash t : \tau$ est définie inductivement comme suit, avec les conventions habituelles sur $\Gamma$ :

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash t(u) : \tau} \qquad \frac{\text{il existe } L \text{ fini t.q. } \Gamma, x : \sigma \vdash t\{x\} : \tau \text{ pour tout } x \notin L}{\Gamma \vdash \lambda.t : \sigma \to \tau}$$

Les mêmes remarques que pour le prédicat « être un terme » restent valides. La quantification cofinie décrit une notion de fraîcheur locale au terme considéré, comme le souligne le lemme suivant.

**Lemme 2.** *Les deux propriétés suivantes sont équivalentes :*

1. $\Gamma \vdash \lambda.t : \sigma \to \tau$
2. $\Gamma, x : \sigma \vdash t\{x\} : \tau$ *pour un* $x$ *localement frais, i.e.* $x \notin \mathrm{fv}(\Gamma) \cup \mathrm{fv}(t)$

Encore une fois, ceci garantit une caractérisation algorithmique qui permet de se passer de la quantification sur $L$ et $x$. Dans le cas de $\Lambda^{\Uparrow}_{\mathtt{c}}$, on perd la décidabilité du typage à cause de l'effacement des types, mais cela peut être résolu en fournissant un terme typé avec le terme nu à typer.

Le choix de la définition des termes transparaît plus clairement dans la preuve de l'énoncé suivant, qui est triviale avec nos définitions, mais compliquée avec la caractérisation algorithmique des termes.

**Lemme 3.** *Pour tout préterme* $t$*, si* $\Gamma \vdash t : \tau$*, alors* $t$ *est un terme.*

Ce lemme de cohérence se transpose à $\Lambda^{\Uparrow}_{\mathtt{c}}$ sur de nombreux points : à cause du second ordre, on distingue aussi entre prétypes et types, et la complexité de la relation de typage se traduit par de nombreuses propriétés de cohérence sur les environnements, les états mémoire, etc.

## 2.3 Avantages et inconvénients

La représentation localement anonyme dispose d'un sérieux avantage : elle permet de cacher sous le tapis toutes les hypothèses implicites sur la fraîcheur des variables dans un $L$ abstrait [8] qui accumule des ensembles que l'on ne voudrait pas avoir à écrire. Les preuves sont donc plus propres, et l'automatisation est plus efficace (nul besoin de vérifier la fraîcheur des variables dans une énorme union d'ensembles).

Les preuves qui doivent contourner cette représentation sont au contraire particulièrement atroces ; le lecteur courageux peut s'en assurer en regardant la chaîne de lemmes qui permet de prouver le résultat `typing_S_restrict` du fichier `Aux/Aux_subst.v`, ou encore les astuces qu'il a fallu déployer pour arriver aux résultats du dossier `Prog`.

Du côté programmatoire, le $L$ considéré est généralement évident, et les résultats d'équivalence algorithmique permettent quand même de calculer des propriétés sur $\Lambda^{\Uparrow}_{\mathtt{c}}$. Ainsi, si l'on voulait extraire des programmes de $\Lambda^{\Uparrow}_{\mathtt{c}}$ via un interpréteur écrit (partiellement [9]) en Coq, toutes ces hypothèses passeraient à la trappe et seraient recalculées à la volée.

8. Qui plus est, qui vit dans `Prop` et qui est effacé à l'extraction.
9. À cause des références, $\Lambda^{\Uparrow}_{\mathtt{c}}$ est Turing-complet.

Il est cependant à noter que la représentation localement anonyme n'est pas la panacée ; il y demeure des choses qui sont très dures à exprimer. En particulier, dès qu'il faut manipuler plusieurs variables liées, les démonstrations se compliquent sérieusement.

On peut notamment citer l'un des seuls points admis du développement de Simpou$\lambda$et, la clôture au contexte, qui a requis plusieurs jours de développement intensif restés vains.

**Définition 6.** On définit, dans cette section, la clôture au contexte d'un ensemble $T$ de termes par :

$$T^\star = \{C[\overline{x} \leftarrow \overline{t}] \mid \overline{t} \subseteq T\}$$

Il est clair pour un humain que $T^\star = T^{\star\star}$. Il suffit en effet de voir $C$ comme un arbre à trous représentés par les variables $\overline{x}$, et d'observer que la double clôture ne fait que de concaténer des sous-arbres. Cependant, cette opération de concaténation, qui nous paraît triviale, est passablement monstrueuse en Coq.

Soit on représente $C$ avec des variables libres $\overline{x}$, auquel cas il faut faire du renommage à la volée pour éviter les conflits entre sous-arbres dans la preuve de $T^{\star\star} \subseteq T^\star$. Cette manière de faire correspond à un point de vue impératif où l'on aurait un générateur de variables fraîches *absolues*, contrairement aux variables fraîches *locales* de la représentation co-finiment quantifiée. La concaténation est en effet alors intrinsèquement séquentielle, et le renommage ne peut pas se faire en parallèle, sous peine de conflit.

Soit on représente $C$ avec des indices de De Bruijn. Dans ce cas, l'opération de concaténation est particulièrement compliquée, puisqu'il faut aussi décaler les indices des sous-arbres suivants. Cela est tout aussi difficile, car le décalage se mélange mal avec les preuves de correction de la clôture (conservation du typage, entre autres).

En l'état, le programme de concaténation a été écrit, mais la preuve de sa correction a été admise. Il semble que ce problème soit connu, car Jacques Garrigue y a déjà été confronté dans un autre développement.

# 3 Implémentation

## 3.1 Structure du développement

L'implémentation use et abuse de découpage en modules modélisés par des dossiers. Cette manière de faire permet de séparer clairement les différents aspects du code ; outre ce découpage, il existe pour les bibliothèques un fichier du même nom que le dossier qui se contente d'exporter tous les modules de celui-ci.

Voici une brève description du contenu des dossiers.

`Lib` et `Meta` contiennent tous les prérequis au travail sur le $\lambda$-calcul en Coq : variables, ensembles finis, ensembles d'association... Le code de ces dossiers provient majoritairement d'un raccommodage des développements de Charguéraud et de `stdlib`, remis au goût du jour pour la version 8.3.

Le développement des ensembles d'association est original, quoiqu'il s'appuie sur celui des ensembles finis de Charguéraud et qu'il réutilise l'implémentation des `FMaps` de Coq. Il paraît en effet plus pratique d'utiliser des structures abstraites plutôt que des listes avec preuve de correction.

`Def` contient toutes les définitions relatives à $\Lambda_{\mathtt{c}}^{\Uparrow}$ : syntaxe, typage, réduction, opérations de base, etc. Ces fichiers sont purement descriptifs et ne contiennent pas de preuves.

`Aux` contient la plupart des lemmes relatifs à $\Lambda_{\mathtt{c}}^{\Uparrow}$ lui-même, lemmes qu'on considère habituellement comme allant de soi dans les preuves, mais qui constituent néanmoins un énorme corpus en Coq.

`Fact` contient les résultats sur $\Lambda_{\mathtt{c}}^{\Uparrow}$ qui sont dignes de mention, à savoir progrès, réduction sujette, pseudo-déterminisme de la réduction, ainsi que quelque lemmes qui en découlent.

`Prog` est un dossier qui traîte de toute la partie algorithmique des propriétés, et qui peut donc être utilisé en tant que bibliothèque pour écrire des tactiques réflexives pour $\Lambda_{\mathtt{c}}^{\Uparrow}$. Les programmes ainsi écrits sont d'ailleurs utilisés ponctuellement dans d'autres parties du développement pour simplifier les preuves grâce au calcul.

`Sim` est lui-même découpé en sous-dossiers analogues, et contient tout ce qui est relatif aux résultats de l'article de Sumii, des définitions aux preuves de correction, en passant par les lemmes intermédiaires.

Enfin, `Examples` contient des développements écrits par Jacques Garrigue qui utilisent la bibliothèque pour prouver l'équivalence de programmes-jouets (pour l'instant).

Remarquons au passage qu'un soin particulier a été pris à utiliser une nomenclature des lemmes qui soit cohérente [10]. Ceci permet de retrouver facilement un résultat sans avoir à parcourir l'ensemble assez conséquent de réponses fournies par une requête `SearchAbout`.

De même, l'emploi de caractères Unicode et de notations *ad hoc* est généralisé, ce qui permet une lecture aisée du code [11].

## 3.2 Méthodologie de la preuve en Coq

Nous avons insisté sur quelques points prophylactiques au cours de ce stage. La philosophie de Simpouλet pourrait être résumée ainsi :

- Limitation au maximum des axiomes indépendants de pCIC ;
- Utilisation des modules pour abstraire les structures de données ;
- Automatisation robuste des preuves.

Nous détaillons ci-après ces trois points.

### 3.2.1 Axiomes supplémentaires

Ce point est particulièrement important, car il est facile d'obtenir un système incohérent en rajoutant des axiomes à tort et à travers. Nous n'avons inclus que des axiomes sûrs et réellement indispensables. En effet, à plusieurs reprises la *proof-irrelevance* aurait trivialisé certaines propriétés, mais de légers remaniements ont permis de s'en passer.

Outre les rares points (triviaux sur le papier) admis faute de temps [12], les deux axiomes suivants sont déclarés :

1. Extensionalité (fonctionnelle) des renommages de locations

$$\forall(\pi\ \rho : \mathtt{loc} \to \mathtt{loc}), (\forall \ell, \pi(\ell) = \rho(\ell)) \to \pi = \rho$$

10. Au risque de répéter des propos proférés en tant que membre de l'équipe Coqtail, c'est loin d'être le cas pour la bibliothèque standard.
11. Sauf sur les documents générés par le `coqdoc` vanilla, qui est notoirement buggué.
12. Cf. section 2.3

2. Extensionalité (propositionnelle) des *value relations*

$$\forall(\mathcal{R}\ \mathcal{S} : \texttt{trm} \to \texttt{trm} \to \texttt{typ} \to \texttt{Prop}), (\forall t_1\ t_2\ \tau, \mathcal{R}\ t_1\ t_2\ \tau \leftrightarrow \mathcal{S}\ t_1\ t_2\ \tau) \to \mathcal{R} = \mathcal{S}$$

Le premier axiome aurait pu être évité au prix de longues circonvolutions dans les preuves, et le gain était suffisant pour en justifier. Le second axiome est au contraire fondamental, puisqu'il réfère aux propriétés d'extensionalité implicitement admises pour les relations décrites dans l'article de Sumii.

### 3.2.2 Modules et abstraction

Les modules de Coq, à l'instar de ceux de ML, permettent de camoufler l'implémentation d'une spécification à l'utilisateur d'une librairie. Cette séparation nette entre code et déclarations a de nombreux avantages et fait partie des bonnes pratiques du programmeur, quel que soit le langage de programmation. Elle a aussi des avantages particuliers à Coq, comme le fait d'éviter de surcharger la mémoire de termes inutiles car opacifiés, ou d'éviter de remplir l'espace de nommage de lemmes intermédiaires sans grand intérêt.

Dans le cas de $\Lambda^{\Uparrow}_{\texttt{c}}$, c'est une bonne chose, vu la taille des preuves, proportionnelle à la taille des définitions. Les détails d'implémentation (variables, ensembles, etc.) étant cachés à l'utilisateur, l'automatisation est autrement plus efficace que s'il avait fallu déplier toutes les définitions.

Cependant, les modules de Coq, hérités des modules ML, sont fondamentalement opposés à la nature même de pCIC.

Évidemment, Coq souffre de quelques défauts de jeunesse qui s'avèrent pénibles, et les modules ne font pas exception. On peut par exemple regretter l'absence d'équivalents aux fichiers `.mli`, ou encore l'incompatibilité de la commande `Print Assumptions` avec les modules (qui sont opaques).

Le vrai problème des modules, néanmoins, réside dans leur seul intérêt. Du fait que les modules abstraits cachent l'implémentation, ils interdisent tout usage même indirect de la règle de conversion de pCIC. En particulier, adieu tactiques réflexives !

Ce besoin de modules partiellement abstraits n'est pourtant pas incongrue. Citons par exemple la décision de $x \in s$ où $s$ est un ensemble calculatoire (type `FSet`). Le calcul pourrait renvoyer `true`, `false` ou bien un terme opaque si le calcul n'aboutit pas, tout en cachant l'implémentation à l'utilisateur et en restreignant l'espace de nommage à l'export.

On peut trouver un argument supplémentaire en faveur d'un tel mécanisme dans l'implémentation actuelle de Coq. En effet, pour une raison mystérieuse, le corps des modules abstraits est tout de même présent en mémoire, même s'il est absent de l'espace de nommage. Il suffirait donc d'ajouter un *flag* aux commandes de réduction comme `compute` qui l'autoriserait à déplier ces termes abstraits, et qui renverrait alors un terme concret si le calcul ne bloque pas, et un terme opaque sinon.

Nous avons exploré plusieurs contournements qui s'apparentent plus ou moins à des *hacks*.

1. Passer par des foncteurs. C'est le contournement le plus propre, mais il n'autorise le calcul que d'un point de vue extérieur à la bibliothèque, une fois le foncteur instancié. En outre, l'absence de sucre syntaxique rend cette construction excessivement lourde.
2. Utiliser des modules non-abstraits, opacifier les définitions à la main (`Global Opaque`), et les redéclarer transparentes dans les modules qui définissent des tactiques réflexives. Cette manière

de faire est en accord avec la philosophie de Coq. Le prix à payer est l'export inévitable [13] des définitions et lemmes intermédiaires.

3. Travailler avec des théories partiellement abstraites. La réflexivité passe alors par des AST, voire par des modèles concrets qui conservent la sémantique de la théorie. C'est la méthode généralement employée pour écrire des tactiques réflexives génériques (tactique `ring` par exemple). Tant que l'on reste dans la théorie, tout va bien ; malheureusement, l'ajout de structure supplémentaire nécessite de dupliquer le code. Accessoirement, c'est aussi une perte de temps : il faut réécrire la théorie que l'on a déjà implémenté.

Aucun de ces contournements n'est parfaitement satisfaisant. En l'état, les modules de Coq jurent avec le reste de ses fonctionalités. Il ne faut cependant pas tirer sur l'ambulance : se passer de modules relève du masochisme.

### 3.2.3 Automatisation

Simpouλet est d'une taille suffisament importante pour qu'on s'interdise de verser dans l'amateurisme. Au regard de la taille des définitions, aussi bien pour $\Lambda^{\Uparrow}_{\mathtt{c}}$ lui-même que pour les relations de bisimilarité, on ne peut pas se passer de tactiques pour traiter les centaines de cas qui apparaissent au détour d'une preuve.

En outre, on doit demander aux tactiques employées d'être efficaces (en terme de temps de calcul) et robustes (ne pas échouer lamentablement pour des modifications mineures).

L'utilisation de $L_{tac}$ et de tactiques comme `auto` est donc un passage incontournable ; cependant, ce sont des armes à double tranchant.

D'abord, parce que $L_{tac}$ paye sa puissance par une lenteur affligeante et une robustesse plus que douteuse. Des rapports de bugs récents l'accusent même d'être responsable de consommation mémoire excessive. Ensuite, parce qu'ajouter naïvement des théorèmes à la base de donnée de `auto` conduisent à une augmentation exponentielle du temps de calcul de cette tactique.

Pour ce qui est de la robustesse, quelques bonnes pratiques peuvent sauver la mise. Simpouλet s'interdit par exemple d'utiliser des hypothèses nommées automatiquement par l'interpréteur, typiquement générées par des `intros` ou `inversion`.

Par ailleurs, le code utilise à de nombreuses reprises des tactiques courtes écrites en dur dans la preuve, qui utilisent des astuces comme le remplacement du nommage explicite d'une hypothèse par un `match goal` (figure 3). Cela rend aussi la preuve plus lisible pour un humain qu'une vague commande anonyme.

| | |
|---|---|
| Concis mais abstrus | `inversion` $H42$ |
| Gloseux mais robuste | `match goal with` $[H$ : `typing` $?S$ $?\Sigma$ $?\Gamma$ $?t$ $?\tau \vdash \_\,] \Rightarrow$ `inversion` $H$ |

FIGURE 3 – Différentes manières de manipuler les hypothèses

Ces pratiques ont fait leurs preuves : une modification récente dans la définition du typage [14] n'a nécessité *aucune* modification dans des preuves qui en dépendent pourtant profondément (réduction sujette entre autres).

13. On pourrait s'en sortir par des sections et des définitions locales (`Let`), mais c'est plutôt lourd.
14. À savoir que le typage d'un *store* requiert que son contenu soit des valeurs.

Pour ce qui est de l'efficacité, plusieurs principes empiriques ont été adoptés.

L'emploi de tactiques réflexives sur les théories omniprésentes et gourmandes en calcul se sont révélées d'un grand secours. Les deux tactiques qui reviennent le plus souvent au cours du développement sont celles qui traitent les ensembles finis (`FSet` et `FMap`), qui sont omniprésents tant pour $\Lambda_{\mathtt{c}}^{\Uparrow}$ que pour les bisimulations.

Ces tactiques manipulent des AST représentant les prédicats (appartenance, association) et opérations de base (union, ajout, etc.) et les transposent en prédicats propositionnels, qui se résolvent à l'aide de la tactique `intuition`. Elles ne sont donc pas purement réflexives, mais sont cependant plus rapide que de la réécriture, et garantissent en outre une certaine robustesse par leur concision.

Pour éviter l'explosion exponentielle de la tactique `eauto`, les théorèmes sont ajoutés localement à la base de données principale en utilisant le mécanisme des sections. L'ajout de bases de données propres (commande `CreateHintDb`) a paru trop lourd pour l'usage qu'on voulait en faire.

Simpouλet fait aussi grand usage de la commande `Hint Extern` qui permet de rajouter des tactiques manuelles à la base de données de `auto`, plutôt que des théorèmes qui ne sont appliqués que si le but est de la même forme que la conclusion. De la sorte, l'automatisation est particulièrement contrôlée.

Enfin, il a fallu porter un intérêt particulier à la taille des preuves générées, car le typage final au moment du `Qed` avait, sur certaines preuves, tendance à faire planter l'IDE ; sans parler du temps de compilation. Certaines tactiques sont particulièrement responsables de l'explosion des termes de preuve ; vu la taille des définitions (17 constructeurs pour les termes, 25 pour la réduction), des tactiques comme `inversion` génèrent des termes gargantuesques.

Sur les points critiques pour la performance, des principes d'inversion opaques ont été utilisés, grâce à la commande `Derive Inversion`, ainsi que la création de sous-lemmes anonymes via la commande $L_{tac}$ `abstract`. On peut ainsi gagner jusqu'à un facteur $\simeq 4$ en temps de compilation.

On peut avoir une idée de l'automatisation poussée à l'extrême dans le dossier `Fact`. La preuve de déterminisme de $\Lambda_{\mathtt{c}}^{\Uparrow}$ génère plusieurs centaines de cas (la plupart absurdes), et l'automatisation les réduits à deux cas, en une dizaine de lignes de $L_{tac}$ et quelques déclarations préalables. La démonstration de la réduction sujette fait aussi un usage intensif de tactiques *ad hoc*.

On regrettera cependant que les preuves de correction des approximations de bisimilarité se prêtent peu à l'automatisation, car les définitions ne sont pas uniformes. Confère par exemple la preuve gloseuse du fichier `Sim/Sim_red.v`.

## 3.3 Structures de données

Coq n'étant après tout qu'un langage de programmation, le choix des structures de données utilisées peut changer drastiquement la complexité du développement. Ceci est encore plus vrai dès lors qu'on veut écrire des démonstrations.

### 3.3.1 Ensembles finis

Les ensembles finis (incluant les ensembles d'association) sont profondément liés au λ-calcul, il était donc nécessaire d'en avoir une axiomatisation facile à utiliser. Pour des raisons pratiques, nous

avons choisi d'utiliser des ensembles qui vérifiassent les spécifications `FSet` et `FMap` de la librairie standard [15].

Cependant, pour se conformer à la pratique mathématique, et pour simplifier grandement les preuves, notre implémentation est extensionnelle et dispose de nombreux principes d'inductions qui font cruellement défaut aux modules de la bibliothèque standard. On pourra trouver les spécifications dans le dossier `Lib`. L'extensionalité s'exprime comme suit [16] :

$$\forall s_1\ s_2 : \mathtt{t}, (\forall x, x \in s_1 \leftrightarrow x \in s_2) \rightarrow s_1 = s_2$$

Le principe d'induction le plus utilisé consiste à induire sur le nombre d'éléments :

$$\forall (P : \mathtt{t} \rightarrow \mathtt{Type}), P\ \emptyset \rightarrow (\forall x\ s, P\ s \rightarrow x \notin s \rightarrow P\ (s \oplus x)) \rightarrow \forall s, P\ s$$

où $\oplus : \mathtt{t} \rightarrow \mathtt{elt} \rightarrow \mathtt{t}$ est l'opération qui ajoute un élément à un ensemble fini.

Contrairement à ce qu'il semble, ces deux propriétés ne vont pas de soi pour les signatures décrites dans la bibliothèque standard. L'implémentation sous forme d'arbres binaires, par exemple, n'est pas extensionnelle (deux arbres distincts peuvent représenter le même ensemble), et le principe d'induction est faux : certains arbres peuvent être construits sans qu'ils soient le résultat d'une chaîne d'ajout d'éléments (en retirant des éléments, par exemple).

Pour obtenir l'extensionalité, il a fallu utiliser une implémentation simple et, en outre, ruser. Les ensembles finis de Simpou$\lambda$et sont donc simplement des listes triées équipées d'une preuve qu'elles sont bien formées [17]. Une telle représentation se fait au détriment de l'efficacité, mais comme *a priori* on ne calculera pas (éventuellement pour la réflexivité) avec ces ensembles, ce n'est pas une préoccupation majeure.

Un problème se pose ici, à savoir qu'en l'absence de l'axiome de *proof-irrelevance*, les preuves de bonne formation ne sont pas forcément uniques. Cependant, une astuce de CIC permet dans ce cas de se passer d'un axiome surnuméraire.

**Lemme 4.** *Dans CIC, les preuves de réflexivité sur un domaine décidable admettent au plus un habitant. En termes mathématiques :*

$$\forall (A : \mathtt{Type})(x : A)(p\ q : x = x), (\forall (x_1\ x_2 : A), x_1 = x_2 \vee x_1 \neq x_2) \rightarrow p = q$$

Cette propriété somme toute anodine nous permet pourtant de nous sortir d'affaire. En effet, en instanciant $A$ par `bool`, on a l'unicité des preuves de $\mathtt{true} = \mathtt{true}$.

Soit $P : A \rightarrow \mathtt{Prop}$ une propriété décidable avec pour fonction de décision associée $f_P : A \rightarrow \mathtt{bool}$. Il suffit alors de remplacer toutes les occurrences de $(P\ x)$ par $(f_P\ x = \mathtt{true})$, qui lui est logiquement équivalent, mais qui admet au plus un habitant.

Le prédicat « être bien formé » pour une liste étant décidable, on tire gratuitement l'extensionalité des ensembles, sans avoir à faire appel à la *proof-irrelevance*.

Cette technique [18] était déjà utilisée par Arthur Charguéraud pour les ensembles finis, mais étonnament, ses différentes implémentations utilisent des listes en clair pour les ensembles d'association ; nous avons adapté le même processus à cette structure.

15. Ces définitions sont d'ailleurs dépréciées à partir de la version 8.3 qui définit des typeclasses en remplacement.
16. `t` est ici le type des ensembles ; les énoncés sont analogues pour les ensembles d'association.
17. Explicitement : triées sans redondance.
18. Relativement connue chez les théoriciens de Coq, mais peu utilisée dans le domaine du génie logiciel, où l'on préfère admettre la *proof-irrelevance* dès le départ.

Les principes d'induction semblent, eux, relativement originaux, même si on en trouve des embryons dans `stdlib`.

### 3.3.2 Détails sur $\Lambda_{\mathtt{c}}^{\Uparrow}$

Comme nous l'avons déjà dit, $\Lambda_{\mathtt{c}}^{\Uparrow}$ utilise la représentation localement anonyme cofiniment quantifiée, aussi bien pour les termes que pour les types. Nous livrons ici quelques détails d'implémentation.

Pour ce qui est des variables, le cahier des charges est assez léger. Il suffit en effet d'avoir :

- Une structure sur laquelle l'égalité est décidable, et, pour avoir l'extensionalité des ensembles, qui soit totalement ordonnée ;
- Une fonction `fresh` : `FSet.t` $\rightarrow$ `var` qui étant donné un ensemble fini de variables, génère une variable qui ne s'y trouve pas.

L'implémentation actuelle, faisant fi de l'efficacité calculatoire, utilise tout simplement les entiers unaires `nat`. Les définitions de ce module sont d'ailleurs partagées aussi bien pour les variables de terme que de type, et aussi pour les locations.

Notons aussi qu'à un certain moment, il a semblé nécessaire de pouvoir générer des variables fraîches hors d'un ensemble infini (pour des fonctions partielles), ce qui aurait sérieusement posé problème, même d'un point de vue mathématique. Heureusement, il s'est avéré que l'on pouvait considérer que les ensembles en question étaient finis.

### 3.3.3 Bisimulations

L'implémentation en Coq des définitions des relations de bisimilarité a nécessité un travail préliminaire non-négligeable. En effet, il a fallu traduire les structures ensemblistes de l'article original en structures (co-)inductives, qui sont la manière naturelle de décrire les choses en Coq.

Ainsi, le code foisonne de définitions à base de *records* qui regroupent toutes les conditions (fort nombreuses !) énumérées dans les définitions, et les clôtures sont définies inductivement. À cause de la taille de ces définitions, nous conseillons au lecteur d'aller jeter un œil aux fichiers du dossier `Sim/SDef`, plutôt que de les recopier ici.

Il est ponctuellement besoin d'utiliser des structures co-inductives, pour décrire par exemple les programmes qui divergent. Ceux-ci sont représentés par une chaîne infinie de réductions : cette structure est nécessairement co-inductive.

Quoi que nous ne soyons pas arrivé jusqu'à ce point, remarquons qu'il semble normal de représenter l'équivalence observationnelle de la section 6 de l'article de Sumii via des structures co-inductives. Il s'agit en effet de « la plus grande relation » qui vérifie certaines conditions.

D'autre part, il est fait une distinction nette entre d'une part les définitions ensemblistes qui décrivent en fait des relations (typiquement les bisimulations et les *value relations*), et des définitions recouvrant des ensembles finis (clôture au contexte, abstraction existentielle). Les premiers sont décrits comme des relations (i.e. des produits à valeur dans `Prop`) tandis que les seconds restent implémentés comme des `FSet`.

On peut justifier cet état de fait par le fait que ces ensembles finis restent essentiellement calculatoires. Dans le cas de la clôture au contexte (cf. section 2.3), on est précisément en train de travailler sur un algorithme de concaténation d'arbres. Faire la même chose avec des relations aurait été pour le moins pénible, puisqu'il aurait fallu exprimer le résultat sous forme existentielle (dans `Prop`).

Inversement, la bisimilarité n'est pas du tout calculatoire, étant donné qu'elle n'est même pas décidable, à cause de la Turing-complétude de $\Lambda^{\Uparrow}_{\mathtt{c}}$.

Nous avons rencontré un autre problème majeur, incarné par le typage implicite des relations de bisimulation. En effet, non seulement, les environnements de typage sont quantifiés existentiellement dans les définitions de l'article (cf. définition 4 par exemple), mais en plus, les endroits où le typage doit avoir lieu sont complètement passés sous silence.

Pourtant, ces hypothèses de typage sont très loin d'être triviales. À de nombreuses reprises dans les preuves, il a fallu exhiber des propriétés de bonne conduite qui découlaient très subtilement du fait qu'on supposait la typabilité des termes.

Il a donc fallu rajouter ces hypothèses de manière parfaitement empirique. Dans le cas des clôtures, par exemple, le typage est inhérent à la définition. Dans le cas des bisimulations, une clause en sus de la définition de l'article requiert la typabilité de la relation.

Enfin, et comme d'habitude en Coq, la difficulté ne se trouve pas toujours là où on l'attend. En effet, les preuves de correction insistent particulièrement sur les deux critères qui font des relations définies des bisimulations ; les autres points sont considérés triviaux. Pourtant, ces points triviaux ont nécessité autant, sinon plus d'efforts que les points « compliqués. »

Par ailleurs, cette mise à plat a permis de mettre à jour une erreur dans un des cas triviaux ; ainsi la clause 2c (ouverture de l'existentielle) des bisimulations à contexte près (définition 9) ne permet pas d'assurer la correction de la clôture.

### 3.3.4 Réflexivité

Nous avons à plusieurs endroits fait usage de réflexivité pour accélerer les démonstrations et améliorer la robustesse des scripts de preuve. On rappelle le principe de la réflexivité dans la figure 4.

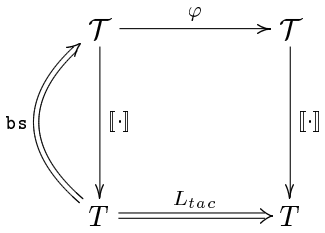



FIGURE 4 – Tactiques réflexives

Le but de la manœuvre est de remplacer de nombreuses applications de $L_{tac}$ par la règle de conversion de CIC, c'est-à-dire du calcul certifié. Le schéma générique d'une tactique réflexive se compose des éléments suivants :

1. Une théorie concrète $T$ (typiquement dans `Prop`)
2. Une théorie abstraite $\mathcal{T}$ (généralement représentée par des AST)
3. Une fonction d'évaluation $[\![\cdot]\!] : \mathcal{T} \to T$
4. Une fonction de réduction $\varphi : \mathcal{T} \to \mathcal{T}$ qui conserve la sémantique : $[\![x]\!] \equiv_T [\![\varphi(x)]\!]$
5. Une tactique de *bootstrap* `bs` qui à partir d'un terme concret construit le terme abstrait correspondant [19] : $[\![\mathtt{bs}(x)]\!] = x$

19. Le symbole = dénote l'égalité syntaxique.

On remplace alors l'invocation d'une tactique écrite en $L_{tac}$ par un simple calcul de $[\![\cdot]\!] \circ \varphi$ sur une instance de $\mathcal{T}$ générée par `bs`. La tactique résultante est alors plus efficace, et l'on peut même utiliser la machine virtuelle pour faire le calcul, et elle est prouvée en Coq, le seul morceau de $L_{tac}$ nécessaire étant le *bootstrap*.

Simpou$\lambda$et a deux manières d'utiliser la réflexivité, du fait qu'il peut être vu à la fois comme un développement *per se* ou bien comme une bibliothèque pour prouver l'équivalence de programmes.

En tant que développement indépendant, il utilise quelques tactiques réflexives très simples, en particulier, deux tactiques manipulant la théorie des ensembles et ensembles d'association finis. On peut trouver leur code dans le dossier `Lib` : il est particulièrement simple, mais cependant très utile.

En tant que bibliothèque, le dossier `Prog` contient quelques caractérisations algorithmiques des propriétés que l'on a à démontrer à répétition quand on manipule des programmes. Certaines preuves du développement utilisent d'ailleurs cette caractérisation, qui permet de contourner les complexes principes d'inductions dérivant de la représentation co-finiment quantifiée. On peut en effet se contenter d'induire sur les prétermes, et utiliser les preuves de correction et de complétude des fonctions de décision pour remplacer respectivement les hypothèses et la conclusion d'une propriété abstraite par une propriété algorithmique [20].

Il est à déplorer qu'il a fallu sacrifier à la simplicité d'utilisation de cette bibliothèque en faveur du développement propre.

En effet, le choix d'effacement des types de $\Lambda_{\mathtt{c}}^{\Uparrow}$ n'est pas innocent, puisqu'on perd la décidabilité du typage. En particulier, on ne peut plus prouver qu'un terme est typable par simple réflexivité. Jacques Garrigue s'est lancé dans l'écriture d'une tactique qui prend en argument un terme avec type, et qui génère un terme nu et une preuve de typage de ce terme. Ce problème pourrait être résolu, s'il est possible de développer suffisament d'efforts pour refaire les preuves avec termes typés.

20. Typiquement, remplacer `term` $t \to$ `term` $u$ par `term_dec_F` $t =$ `true` $\to$ `term_dec_F` $u =$ `true`.

## Conclusion

Au bout de deux mois de développement, Simpou$\lambda$et commence à représenter un corpus de code non-négligeable. Sa dizaine de milliers de lignes peut en témoigner, ainsi que le domaine couvert par les lemmes intermédiaires, qui, espérons-le, pourront servir à manipuler des preuves d'équivalence de programmes.

Des résultats importants sont déjà présents, et l'infrastructure nécessaire à d'éventuelles extensions est très étendue, ne serait-ce parce que $\Lambda^{\Uparrow}$ est un langage plutôt vaste.

Il faut cependant remarquer que de nombreux points restent incomplets, voire vierges, comme par exemple la preuve de correction et de complétude des bisimulations vis à vis de l'équivalence observationnelle. La programmation en Coq est en effet extrêmement chronophage, et même à plus de 50 heures par semaine, et en y passant ses nuits, deux mois de développement ne sont pas suffisants pour implémenter un article dense d'une bonne dizaine de pages.

Travailler à la formalisation de théories est assez peu compatible avec une autre activité, sauf à le faire sur des périodes de temps très longues[21]. À l'autre bout du spectre de Curry-Howard, le métier de développeur existe bien en lui-même !

D'autre part, certains résultats sont fondamentalement difficiles à encoder, et nécessiteraient sans doute des développements dédiés. On regrettera au passage l'absence d'outils au sens large pour développer en Coq (IDE, bibliothèques, documentation[22], scripts, etc.). Cette absence peut se justifier par la relative jeunesse et la confidentialité des assistants à la preuve en général.

Le travail sur Simpou$\lambda$et s'est montré très instructif, aussi bien d'un point d'un point de vue théorique (bisimulations, formalisation du $\lambda$-calcul), que des connaissances pratiques acquises par effet de bord (techniques de preuves, exploration de l'implémentation de Coq).

Le développement de Simpou$\lambda$et étant ouvert, nous espérons pouvoir continuer à travailler dessus, et voir d'éventuelles personnes intéressées par le sujet venir contribuer au projet.

## Références


[1] Arthur Charguéraud. « The Locally Nameless Representation ». 2009.

[2] Adam Chlipala. *Certified Programming with Dependent Types*. 2010.

[3] *Coq*. URL : `http://coq.inria.fr/`.

[4] Ken-etsu Fujita et Aleksy Schubert. « The Undecidability of Type Related Problems in Type-free Style System F ». Dans : *RTA*. 2010, p. 103–118.

[5] Adam Koprowski. « A Formalization of the Simply Typed Lambda Calculus in Coq ». 2007.

[6] *Simpou$\lambda$et*. URL : `http://sourceforge.net/projects/simpoulet/`.

[7] Eijiro Sumii. « A Complete Characterization of Observational Equivalence in Polymorphic Lambda Calculus with General References ». Dans : *CSL*. 2009, p. 455–469.


21. Afin de laisser aux développeurs Coq le temps de casser la compatibilité ascendante.
22. Que celui qui possède une description détaillée de la sémantique de $L_{tac}$ me jette la première pierre !