Cloisonnement des contenus calculatoire et logique du filtrage dépendant en théorie des types et application aux coupures commutatives

Pierre Boutillier
sous la direction de Hugo Herbelin
Équipe πr^2 - laboratoire PPS, 23 avenue d'Italie, Paris

Février - Juillet 2010

Table des matières

Fiche de synthèse

Le contexte général

La recherche en informatique a perpétuellement étendu durant les 40 dernières années la richesse des langages fonctionnels. Ils représentent aujourd'hui l'un des paradigmes les plus propices à l'élaboration de langages encore plus puissants et rigoureux de demain.

Dans ce cadre, le raisonnement par cas (ou **filtrage**) est une fonctionnalité centrale et bien connue. La force mais aussi la complexité de cette fonctionnalité augmente néanmoins fondamentalement dans le cadre de systèmes à types riches (ou types dépendants).

L'exploitation de cette richesse du point de vue logique est à la base de la théorie des types de Martin-Löf et les théories des types modernes l'intègrent très souvent.

Du point de vue de la programmation pure, tenter d'offrir un langage de programmation à types riches est fondamentalement conditionné par la maîtrise du filtrage dans cet environnement. L'implémentation d'Agda ou d'Epigram en Europe ont donc amené à une étude importante de ces questions.

Le problème étudié

Raisonner par cas, c'est fondamentalement raisonner avec l'égalité : égalité entre ce sur quoi le filtrage est réalisé et le cas que l'on traite à un instant t mais aussi égalité entre les arguments du cas traité et les arguments généraux de l'objet filtré.

Néanmoins, l'égalité n'est que l'identité, soit du bruit inutile, une fois le typage réalisé. Il faut par conséquent au maximum déplacer ces « calculs » indispensables pour la validité du typage en dehors du champ des réductions qui constitue le calcul réel, celui dont on tire de l'information lors de l'utilisation.

Actuelement, la réponse commune est de masquer intégralement ces calculs dit **logiques** (ceux à propos des égalités) pour ne faire apparaître que le contenu calculatoire. De tels systèmes infèrent, résolvent et exploitent silencieusement et automatiquement des preuves d'égalité toujours plus intelligentes mais nécessairement restreintes à un cadre décidable. Coq adopte lui un point de vue plus ancien mais toujours intéressant. Son noyau est par lui même faible mais il permet de poser et de réaliser explicitement des jugements à propos de l'égalité dans chacun des cas. Le calcul en est pollué mais la puissance du système est alors uniquement limitée par « l'intelligence du programmeur ».

Mon travail a été de trouver où et comment placer en Coq les preuves d'égalité que les autres systèmes devinent afin qu'elles restent explicites mais qu'elles ne gênent pas le calcul. Masquer en codant dans le noyau une théorie

décidable n'est pour moi pas recevable car la possibilité de fournir dans les cas complexes des preuves explicites est un atout primordial de Coq.

La contribution proposée

Je me suis aperçu qu'au-delà des différentes manières génériques de représenter les égalités que j'ai étudiées, le filtrage constituait en présence de constructeurs uniquement une manière d'écrire les égalités en lui même et plutôt que d'écrire `n = S m → T`, on pouvait écrire à la condition d'avoir un type `dummy` dont on connait un habitant trivial (`True` et `I` par exemple), `match n with |S m => T |_ => dummy end.`. Cette astuce mise au bon endroit permet de plus d'économiser la réécriture de `n` en `S m` dans `T` !

De même, plutôt que de relier les arguments d'un constructeur au type d'une valeur filtrée, il est plus judicieux de généraliser tout ce qui dépend du terme sur lequel le filtrage va avoir lieu. On peut ainsi travailler uniquement et directement dans l'environnement spécifique à chacune des branches.

Exploiter cette idée m'a amené à redéfinir une condition de garde structurelle sur les points fixes fortement normalisants à la fois plus permissive et plus rigoureuse que celle qui était implémentée dans Coq.

Les arguments en faveur de sa validité

A partir de cette idée, j'ai réécrit une implémentation de la bibliothèque de Coq `listn` (les listes dont la longueur est stockée dans le type) dont les limites sont les mêmes que les implémentations des autres langages à types dépendants.

J'ai ainsi montré une manière d'égaler les autres systèmes sans modifier ni la taille du noyau ni l'ampleur de l'expressivité de Coq.

La nouvelle condition de garde s'exprime sous la forme d'un calcul des séquents. En plus d'être plus large, elle est donc plus à même d'offrir des opportunités de formalisation certifiant sa correction.

Le bilan et les perspectives

Mon idée ne s'applique qu'à des objets dont l'injectivité est primitive (les constructeurs d'inductif). Comment exploiter l'injectivité prouvée par l'utilisateur d'une fonction définie par un terme quelconque ?

Le problème est même double car il faut sur le plan pratique savoir comment offrir des outils à l'utilisateur afin que ces preuves soient automatiquement exploitées et sur le plan voir comment les placer dans la théorie afin qu'elles n'altèrent pas le calcul. C'est-à-dire vraisemblablement que la réduction les efface sans les calculer.

De même, et c'est un problème général en la matière aujourd'hui, si `(S n) + m = S (n + m)` calculatoirement, qu'en est il de `n + (S m)` ? Le raisonnement modulo théorie est un enjeu des futurs assistants de preuves.

Introduction

Le logiciel Coq développé au sein de l'INRIA est un assistant de preuve. Son premier rôle est de fournir une infrastructure pour écrire et vérifier des preuves mathématiques.

Néanmoins, l'une des découvertes les plus fondamentales du siècle dernier nous dit que dans nombre de cas, écrire des preuves ou écrire des programmes fonctionnels n'est qu'une seule et même opération. Coq réalise la vérification des preuves justement à l'aide de cet isomorphisme. Il écrit des programmes dont il vérifie la validité par leur typage. Rien ne nous empêche donc d'écrire nous aussi des programmes. Nous y gagnerons une extraordinaire puissance pour attester que nos programmes sont corrects.

Le système de type de Coq s'appelle le **Calcul des Constructions Inductives**. Il est puissant mais très aride pour l'utilisateur. Il y a de ce fait plusieurs étapes entre les entrées de l'utilisateur et le vérificateur de type. Concrètement, outre des bibliothèques boites à outils, l'architecture du logiciel est la suivante :

noyau : à la base de l'édifice se trouve le vérificateur de type. C'est la pierre angulaire de l'édifice, là se trouve le code qui garantit la validité des preuves, la partie à laquelle « il faut faire confiance ». Il est compliqué mais doit rester le plus petit et le plus générique possible afin que un spécialiste du domaine soit théoriquement capable d'en approuver le fonctionnement à la simple lecture du code.

inférence de type : au-dessus du noyau se trouve un système d'inférence de type. Pour maintenir le noyau certifiable, son comportement est uniquement de vérifier, il lui faut donc absolument toutes les informations. Certaines informations peuvent en fait être devinées et toutes n'ont donc pas à être écrites, le système complétera les blancs « faciles ».

interpréteur : en plus de contenir des blancs, les termes peuvent être fournis d'une manière plus concise ou plus naturelle que celle que comprend le noyau. Cette remise en forme est la troisième couche du logiciel.

C'est à ce niveau que l'utilisateur fournit des termes explicitement au système le cas échéant. Cela revient à voir Coq comme un langage de programmation avec types riches. Cette opportunité que nous adopterons dans toute la suite du rapport n'est pas historiquement la force de Coq. Nous nous attellerons à en augmenter la puissance et la facilité.

engin de preuve : Coq dispose ensuite d'un système pour convertir l'écriture de terme en la recherche de preuve. Le tour de force est de permettre d'écrire des théorèmes comme un mathématicien alors que ce sont en réalité des programmes d'informaticiens qui sont générés.

tactiques : ces preuves de mathématiciens sont écrites grâce à des briques représentant des « schémas de termes ». Elles exploitent la structure des

hypothèses et de la conclusion recherchée pour fournir plus ou moins automatiquement le témoin de véracité des théorèmes.

lecture/écriture : enfin, interagir avec un humain suppose de disposer d'outils pour lire et écrire de manière courte et simplement compréhensible par l'utilisateur.

Maintenant que notre terrain de jeux est décrit, voyons successivement quel système est mis en place pour écrire des programmes, de quoi nous avons besoin pour en tirer profit, comment l'exploitation se met en œuvre et pour finir quelle règle du jeu il faut aménager pour nous permettre d'avoir les mains libres ...

1 Types inductifs

1.1 Définition

Les programmeurs de langages fonctionnels s'appuient sur deux principes fondamentaux. L'idée fondatrice est d'utiliser une fonction comme n'importe quelle autre valeur. Pour les logiciens, une implication est aussi une proposition utilisable partout. Mais écrire des programmes clairs passe aussi par définir et utiliser ses propres structures pour représenter ses données. Logiquement aussi, cette idée est porteuse et régit la théorie des types de Martin-Löf.

Les paragraphes suivants introduirons les objets et la terminologie mise en place pour permettre cette deuxième fonctionnalité.

Les structures de données sont introduites avec des **constructeurs** (`vrai` et `faux` par exemple pour les booléens) et détruites par le mécanisme de **filtrage** (dont la syntaxe varie mais dont le principe est le raisonnement par cas. Dans notre exemple : « que faire si on a `vrai` »/ « que faire si on a `faux` »).

Il est extrêmement réducteur de s'arrêter aux listes de constantes, les constructeurs peuvant avoir des **arguments** qui stockent de l'information. Ces arguments peuvent être eux-mêmes des éléments de la même structure. Ils servent alors à hiérarchiser l'information.

Pour être concret, prenons le cas des arbres cher aux algorithmiciens. Les arbres sont représentés en pratique comme en théorie par 2 constructeurs : la `feuille` qui prend une valeur comme argument et le `nœud` qui stocke une ou plusieurs valeurs mais aussi un ou plusieurs arbres (ses fils ou sous-arbres) comme arguments.

La boite à outils n'est pourtant toujours pas complète. En effet, derrière des constructions bien fondées dépendantes d'elles-mêmes se cache systématiquement la notion de **point fixe**. Or, il faut pouvoir éliminer tout ce qui est construit. Les **fonctions récursives** sont ces points fixes.

Du point de vue logique aussi, l'introduction de telles constructions augmente sensiblement l'expressivité du système de travail. Néanmoins, maintenir la consistance du système demande des restrictions sur les structures admissibles. Sinon, nous écririons des programme tel que

Fixpoint magic (x : True) : False := magic I.

or magic I est une preuve de Faux !

Précisément, lors de la construction, il faut assurer la **stricte positivité** des potentiels points fixes (ce point ne sera pas étudié en détail dans la suite). Lors de la destruction, rester consistant impose de certifier que les fonctions récursives écrites ne pourront jamais induire une infinité de réductions. Ce critère de correction est nommé **condition de garde** ; il fera l'objet d'une étude particulière ultérieurement. Les habitants restants sont appelés **inductifs** s'ils représentent un plus petit point fixe et **co-inductif** sinon.

Enfin, l'introduction de types dépendants des termes ou **types riches** ajoute l'enjeu supplémentaire d'exploiter pleinement les informations issues du typage. L'apport et les contraintes induits par un typage précis est le cœur de ce rapport.

Remarque de formalisation : Définir un inductif est une opération qui doit être réalisée avant d'écrire des programmes les utilisant. Le nom de la structure, de ses constructeurs et les arguments attendus doivent être donnés au système « globalement » afin qu'il contrôle la validité des programmes. Cette opération n'a aucun lien avec l'écriture d'un programme à proprement parler. Cet uniformité des tâches et l'aspect global et mouvant de l'ensemble des inductifs définis compliquent énormément les tentatives de définition des « termes avec inductifs bien formés » dans un langage à type dépendant.

1.2 Typage

Les inductifs peuvent avoir des paramètres. Une `liste`, par exemple, est un cadre générique pour représenter les listes de caractères, d'images, de fonctions sur les entiers... Elle ne doit par contre pas mélanger carottes et choux-fleurs. On écrit donc les listes d'éléments de type α où α est une variable instanciée par l'utilisation concrète qui sera faite des listes (`caractere`, `tableau`, ...).

Dans le cas dépendant, à la différence du cas simplement typé, les inductifs ont aussi des arguments qui changent en fonction des constructeurs. Pour être concret, introduisons dès maintenant ce qui sera un leitmotiv de ce rapport, la liste à n éléments : listn.

```
Coq < Inductive listn (A : Type) : nat -> Type :=
Coq < |nil : listn A 0
```

```
Coq < |cons : forall (h : A) (n : nat) (t : listn A n),
Coq <   listn A (S n).
```

L'entier que stocke le type représente ici le nombre d'éléments que contient la liste. Il est donc différent selon que la liste est vide ou constituée d'une tête et d'une queue.

Maintenant, lorsque nous allons détruire un élément de type listn, l'entier donnant la longueur communiquera de l'information dans les 2 sens.

- D'une part, dans la branche nil, nous saurons que cette taille est 0, ce qui affinera par conséquent le type des autres hypothèses et/ou du but.
- D'autre part, si nous savons que cette longueur est le successeur d'un nombre entier, nous saurons qu'il s'agit forcément d'un cons.

Dans ce formalisme et grâce au typage, il nous sera possible de définir la fonction qui retourne la queue d'une liste uniquement dans le cas où celle-ci est non vide. Nul besoin d'une valeur par défaut ou d'un mécanisme d'exception pour le cas vide. Écrivons :

```
Coq < Definition tail (A : Type) (n : nat) (v : listn A (S n)) :=
Coq < match v with
Coq <   |cons h n t => t
Coq < end.
tail is defined
```

Mais comment établir une règle de typage générique pour gérer tant d'information ? Coq utilise pour cela un **prédicat de retour**, c'est-à-dire une fonction qui pour toute valeur des arguments et du terme filtré spécifie le type du terme attendu en retour. Afin d'alléger le travail du programmeur, lorsqu'il n'est pas donné et qu'il est « simple », ce prédicat fait l'objet d'inférence. La fonction tail est en réalité :

```
Coq < Definition true_tail A (n : nat) (v : listn A (S n)) :=
Coq < match v as v0 in (listn _ n0)
Coq <   return match n0 with | 0 => ID | S n1 => listn A n1 end
Coq < with
Coq <   |nil => id
Coq <   |cons h n t => t
Coq < end.
```

Les termes ainsi écrits sont durs à comprendre au premier coup d'œil. En effet, si le prédicat de retour est bien une abstraction selon le terme filtré et les arguments de son type, afin d'être au final plus lisible, il ne s'écrit pas sous la forme fun pattern => ret. Le nom donné à la variable qui représente le terme est introduit après le mot clé as. Ceux donnés aux arguments sont définis en utilisant la construction : mot clé in, le nom de l'inductif, des _ pour les paramètres et enfin des identifiants pour les arguments. Le corps du prédicat suit pour finir le mot clé return.

Env. 1 : Pattern-matching rule

$$\frac{\Gamma \vdash \mathrm{w} \in \mathrm{I}\ u_1 \dots u_n\ \textbf{(1)} \quad \overrightarrow{\Gamma \vdash \mathrm{u} \in \mathrm{T}\ \textbf{(3)}}^{n} \quad \overrightarrow{\Gamma \vdash \mathrm{C}\ y_1 \dots y_m \in \mathrm{I}\ v_1 \dots v_n\ \textbf{(7)}}^{k} \quad \Gamma \vdash \lambda(x:T)_1 \dots (x:T)_n\ (\mathrm{z}{:}\mathrm{I}\ x_1 \dots x_n).\ \mathrm{P} \in \mathrm{Type}\ \textbf{(10)}}{\overrightarrow{\Gamma \vdash \mathrm{t} \in (\lambda(x:T)_1 \dots (x:T)_n\ (\mathrm{z}{:}\mathrm{I}\ x_1 \dots x_n).\ \mathrm{P})\ v_1 \dots v_n\ (\mathrm{C}\ y_1 \dots y_m)\ \textbf{(12)}}^{k}}$$

$$\frac{}{\Gamma \vdash \mathrm{match\ w\ as\ z\ in\ I}\ x_1 \dots x_n\ \mathrm{return\ P\ with}\ \overrightarrow{|\mathrm{C}\ y_1 \dots y_m \Rightarrow \mathrm{t}}^{k}\ \mathrm{end} \in (\lambda(x:T)_1 \dots (x:T)_n\ (\mathrm{z}{:}\mathrm{I}\ x_1 \dots x_n).\ \mathrm{P})\ u_1 \dots u_n\ \mathrm{w}\ \textbf{(15)}}$$

Ce prédicat de retour est instancié par le terme filtré et les arguments de son type à l'extérieur du filtrage (ligne 15). Il est, à l'intérieur des branches, instancié par la valeur du constructeur courant (prémisse 12). Mais il doit aussi avoir un sens par lui même indépendamment de ses différentes instanciations (prémisse 10). Nous verrons ce que cela induit dans la partie 2.2.

Associés à de judicieuses généralisations de terme, les usages de ce prédicat (étudiés plus concrètement dans la partie 3) sont de trois types. Il joue un rôle de cœrcion qu'exploite perpétuellement la tactique destruct. Il permet d'obtenir l'égalité entre les arguments hors du filtrage et ceux à l'intérieur de chacune des branches afin d'encoder les règles de filtrage des langages à types dépendants modernes tels que Agda ou Beluga. Le plugin Program fournit un cadre pour écrire plus simplement de tels termes en Coq. Enfin, il offre un espace dans lequel inverser des égalités (c'est-à-dire obtenir des égalités sur les arguments d'un constructeur à partir d'égalités sur les constructeurs) sans polluer le calcul.

1.3 Coupures commutatives

La sémantique d'une élimination d'un inductif est assez naturelle. Elle consiste à remplacer $\mathrm{match\ C}_i\ u_1 \dots u_j\ \mathrm{with}\ \overrightarrow{|\mathrm{C}\ x_1 \dots x_k \Rightarrow \mathrm{t}}^{l}\ \mathrm{end}$ par $(\lambda x_1 \dots x_j.\ \mathrm{t}_i)\ u_1 \dots u_j$.

En introduisant le concept de **contexte** F[∘], c'est-à-dire de terme contenant un « trou ∘ » où va être inséré un terme donné en argument, il apparaît clairement que $\mathrm{F[match\ u\ with}\ \overrightarrow{|\mathrm{C}\ x_1 \dots x_k \Rightarrow \mathrm{t}}^{l}\ \mathrm{end]}$ et $\mathrm{match\ u\ with}\ \overrightarrow{|\mathrm{C}\ x_1 \dots x_k \Rightarrow \mathrm{F[t]}}^{l}\ \mathrm{end}$ sont identiques après l'élimination de l'inductif (si u se réduit sur un terme commençant par un constructeur). Sinon (si u est un terme débutant par une variable), il est peut être possible de réaliser des réductions dans les branches une fois le terme mis sous la deuxième forme. On obtient ainsi des termes plus concis.

Ces transformations s'appellent les **coupures commutatives** et sont exploitables par les langages fonctionnels pour optimiser la taille du code généré.

Par exemple, la double négation d'un booléen devient

```
match b with
  |Vrai -> Vrai
  |Faux -> Faux
end.
```
au lieu de
```
match (match b with
         |Vrai -> Faux
         |Faux -> Vrai
       end)
with
  |Vrai -> Faux
  | Faux -> Vrai
end.
```
en utilisant le contexte `match ∘ with |Vrai → Faux |Faux -> Vrai end.`.

Néanmoins, dans notre description, nous ne nous sommes jamais demandé si $F[t_l]$ avait un sens. En effet, quand les types ne dépendent pas de termes, les constructeurs n'ont pas la possibilité d'influer sur le type du résultat du calcul. Néanmoins, la prémisse 12 de la règle de typage nous indique qu'en Coq, il en est totalement autrement et concrètement utiliser le contexte S ∘ dans l'expression S (tail l) aboutirait à
```
match v (..) with
  |nil => S (id)
  |cons _ _ t => S t
end
```
qui n'a aucun sens (mat typé).

Avoir des types différents dans les différentes branches d'une élimination est exactement le canal de transmission à chacun des cas des informations que contiennent les types dépendants. Loin de tenter d'uniformiser ces types afin de permettre les transformations, notre démarche va au contraire être de préciser au maximum le contexte dans lequel nous nous trouvons hors de l'élimination afin d'exclure les branches qui ne respectent pas exactement ce contexte.

Dans tous les cas impossibles car imparfaitement compatibles, nous fournirons une réponse systématique et détachée du contexte extérieur (id ou True par exemple). A l'extérieur du filtrage, notre réponse n'aura donc absolument aucun sens. (Quel est le successeur de fun (A : Type) (x : a) => x ?)

Nous verrons dans la partie 3 que raisonner par cas en Coq passe par un processus de

1. généralisation des hypothèses dépendantes
2. filtrage
3. réintroduction de ces hypothèses après cœrcion dans chacun des cas.

Une telle démarche est absolument nécessaire au typage et réduire les coupures commutatives représenterait un effacement pur et simple de ces arguments de correction.

2 Différentes égalités et les manières d'en tirer de l'information

2.1 Egalité dépendante, hétérogène et imprédicativité

Si l'algorithme suivant lequel l'addition est définie dit exactement 0 + n = n, n + 0 va aussi pour tout n concret être exactement n après calcul, mais rien ne le dit quand n est abstrait (l'addition travaille par cas sur le premier nombre). Programmer avec des types dépendants nécessite une relation d'équivalence sur les termes.

Il faut souvent, pour faire aboutir une preuve, prouver qu'il est possible de donner un terme de type a alors que le système attend un terme de type b. Il est alors nécessaire de posséder une manière de dire que là où a est demandé, b convient. Ceci s'exprime formellement par

Definition eq A (x y : A) := forall P : A → Type, P x → P y

dans un cadre imprédicatif.

Symétriquement, une égalité valide niée ou une égalité absurde en hypothèse permet d'éliminer les cas qui ne sont pas pertinents pour l'étude et évite de fournir des preuves impossibles (nul besoin d'étudier le cas de la liste vide pour l si l'on suppose que length l = 0 → False).

Néanmoins, la définition inductive de la propriété a = b (dont le principe d'induction est le terme sus-cité) se réduit à « un terme est égal à lui-même », soit :

Inductive eq A (x : A) : A → Prop := eq_refl : eq A x x.

. Il apparaît alors clairement pourquoi les lemmes à propos d'égalité n'ont pas d'intérêt pour le calcul à proprement parler : en présence de termes clos, calculer une égalité se résume à déduire eq_refl par une succession sans surprise de réductions issue de eq_refl. Qui imaginerait écrire en CaML des programmes purement fonctionnels n'exploitant que le type `unit` ?

Le cas des termes clos dont la réduction est plus lente n'est pas important au regard de la réduction de termes ouverts. Réduire des termes avec des variables libres est absolument indispensable pour la conversion de type donc pour les preuves. Or, la réduction est purement et simplement bloquée là où des preuves d'égalités peuvent être des variables, les réductions intéressantes qui suivent les égalités sont masquées. L'équivalence entre des types pourtant égaux à des coercions prêts n'apparaît pas.

```
Coq < Fixpoint rev_append A n m (i : listn A n) (o : listn A m)
Coq <   : listn A (n + m) :=
Coq < match i in listn _ n0 return listn A (n0 + m) with
Coq <   |nil => o
Coq <   |cons h n' t => eq_rect (n' + S m) (listn A)
Coq <       (rev_append A n' (S m) t (cons A h m o))
Coq <       (S n' + m) (eq_sym (plus_n_Sm n' m))
```

```
Coq < end.
rev_append is recursively defined (decreasing on 4th argument)
```

en est une illustration probante. Une cœrcion à l'aide de eq_rect de (eq_sym (plus_n_Sm n' m)) est requise. La réponse dans le cas cons est en effet (rev_append A n' (S m) t (cons A h m o) de type listn A (n' + (S m)) alors que le système attend un terme de type listn A ((S n') + m). La contrepartie est que rev_append (0 : : t) l ne se réduit pas à rev_append t (0 : : l) car plus_n_Sm ne fait qu'une réduction sur les n nécessaires à obtenir eq_refl donc eq_sym et eq_rect (qui sont elles aussi au niveau des termes des fonctions qui à eq_refl associe eq_refl) ne se réduisent pas.

Mais les ennuis ne s'arrêtent pas là car les types des termes peuvent eux aussi être dépendants. Si l1 @ l2 représente le terme « concaténation de l1 et l2 » de type listn (n1 + n2), comment exprimer l'associativité de @ ? En effet, l1 @ (l2 @ l3) a le type listn (n1 + (n2 + n3)) et (l1 @ l2) @ l3, listn ((n1 + n2) + n3) donc (l1 @ l2) @ l3 = l1 @ (l2 @ l3) n'est pas typable.

Plusieurs solutions se présentent :

égalité hétérogène : dire que tous les termes peuvent être comparés même s'ils ne sont pas forcément dans le même type Inductive JMeq A (x : A) : forall B, B → Prop tout en conservant à l'esprit qu'en réalité l'égalité n'existe qu'au sein d'un même type := JMeq_refl : A x A x. Cette solution est flexible mais partielle. L'égalité (l1 @ l2) @ l3 = l1 @ (l2 @ l3) ne permet pas d'obtenir simplement ((n1 + n2) + n3) = (n1 + (n2 + n3)).

égalité dépendante : ne pas séparer le terme du type qui le surveille et parler de l'égalité de la paire (dépendances des types, terme de ce type) au moyen d'un prédicat. Ici (((n1 + n2) + n3, (l1 @ l2) @ l3) = (n1 + (n2 + n3), l1 @ (l2 @ l3)) avec le prédicat fun n => listn n.eq_dep A a B b est la même chose que JMeq A a B b avec le prédicat fun X => X et plus fort sinon car elle contient l'information que les dépendances de types sont égales explicitement. Elle répète par contre très régulièrement de l'information à plusieurs endroit générant ainsi des termes inutilement importants.

égalité de télescope : un **télescope** est une suite finie de termes dont le n-ième peut dépendre des (n - 1) précédents. C'est-à-dire, une construction de type forall (a : A) (b : B a) (c : C a b) . . . . Ils peuvent être vus comme des généralisations de

```
Coq < Print sigT.
Inductive sigT (A : Type) (P : A -> Type) : Type :=
    existT : forall x : A, P x -> sigT P
```

qui permet d'ailleurs de les encoder et représentent les hypothèses d'un énoncé (lemme/constructeur/. . .).

Le type d'un télescope peut de plus être affiné si moins de dépendance apparaît. Par exemple, le télescope des arguments de cons devient alors

de type (A B : Type) (C : A → B → Type) et non pas (A : Type) (B A → Type) (forall (A0 : A), B A0 → Type) car il sera habité par (h : A) (n : nat) (t : vector A n).

C'est la forme d'égalité la moins facile à mettre en place car la moins générique mais la plus précise et souple à utiliser.

Reste que, comme nous l'avons vu, une égalité n'a aucun intérêt en elle même. C'est son utilisation (sa destruction) qui compte. Or, comme nous allons le voir tout de suite, la destruction de ces affirmations n'est ni aisée ni intuitive.

2.2 Axiome K de Streicher

Il parait extrêmement naturel à l'esprit que si x et y sont de même type, toutes les égalités citées précédemment à leur propos sont définies et équivalentes. Pourtant, le typage nous empêche d'écrire cette équivalence. Elle reste néanmoins compatible avec le calcul des constructions inductives dans lequel elle est souvent ajoutée sous le nom d'axiome **Streicher-K** (du nom du mathématicien qui a mis en évidence l'indépendance de la propriété avec le reste du système).

Pour comprendre en pratique où se situe le problème, retournons à l'étude de la règle 10 (Env. 1) qui impose que la clause de retour dans sa généralité ait un sens. Or, quand bien même dans match eq (pour eq : JMeq A x A y) et dans with JMeq_refl => ... end., A est commun, il faudrait que in JMeq _ _ A' y' return x = y' ait un sens (c'est-à-dire y' : A).

Les termes appliqués ici à l'inductif à la fois hors et dans le filtrage sont moins généraux que ceux de sa définition. Les solutions pour restreindre l'espace dans lequel la clause de retour est typée reviennent à inférer dans le noyau cette spécification afin de pouvoir l'exploiter. Bien que ce soit la mouvance actuelle, plusieurs raisons s'opposent à cette modification. En effet, l'inférence est une opération coûteuse. Plus fondamentalement, quand bien même les capacités de calcul augmentent, puisque inférence implique algorithme, elle doit être décidable et est donc forcément limitée. Que faire des cas où le système ne sait pas si l'humain ne peut pas lui dire ? Enfin, placer un tel système dans le noyau augmente sensiblement la taille du code auquel il faut faire confiance et fait perdre la dimension « terme de preuve »/témoin qui permet un contrôle externe sur la preuve.

De plus, bien que cette simplification soit commode pour l'esprit humain qui bataille pour réaliser des preuves (a fortiori quand cet intellect tente de mettre en œuvre des procédés automatiques de démonstration), il est naturel, si nous revenons à l'essence de l'égalité : « un contexte où j'ai a et un contexte où j'ai b sont en définitive le même terrain de jeu », de considérer une substitution simultanée des termes et de ce dont ils dépendent.

Le principe d'induction sur les télescopes affinés (partie 2.1)

forall (A : Type) (B : A → Type) (P : forall (a : A) (ty : B a), Type), P A x → P B y

est donc suffisant.

Remarque de formalisation : Il serait sur le papier possible de construire une clause de retour valable s'il était possible de dire

fun eq : JMeq A x A y => match eq in JMeq _ _ B b return "if B = A then x = b else True" with eq_refl => eq_refl end.

. Or il arrive que l'égalité entre 2 types dépendent d'une égalité décidable (typiquement l'égalité sur les entiers et listn). On retrouve dans ce cas un terme qui prouve cette instance de K.

2.3 "No confusion", injectivité, discrimination, acyclicité

Il arrive couramment, a fortiori lorsqu'il s'agit d'égalités découlant de l'instantiation d'une clause de retour dans une branche (ce que fait le plugin Program en Coq), d'obtenir des égalités de la forme f $x_1 \dots x_i$ = g $y_1 \dots y_j$ dont il est souhaitable d'obtenir soit une absurdité soit des relations entre les x et les y.

Les résultats peuvent être de 3 types :

l'injectivité f x = f y → x = y

la discrimination f a = g b → False

l'acyclicité/point fixe f x = x → False or f x = x → x = ?

Le mécanisme du filtrage impose que les constructeurs aient ces propriétés. Ainsi lorsque f et g sont des constructeurs, il existe des preuves génériques à l'aide d'une réécriture et d'un filtrage.

Pour d'autres fonctions pour lesquelles l'utilisateur aurait fait les preuves, il serait possible de réaliser des tables associatives dont les clés seraient le nom des constantes et dont l'exploitation étendrait avec un coût en calcul quasi nul la puissance des tactiques sur les égalités.

3 Utiliser les dépendances dans le filtrage

3.1 A partir du travail à propos de l'égalité

En présence de type dépendant, filtrer suppose de mettre en correspondance les termes du cas général avec ceux des cas particuliers.

Par exemple :

```
Coq < Fixpoint Vbinary A n (v w : listn A n) := match v,w with
Coq <   |nil,nil => (* n = 0 *) I
Coq <   |cons hv nv tv, nil => (* n = S nv, n = 0 *) I
```

```
Coq <   |nil, cons hw nw tw => (* n = 0, n = S nw *) I
Coq <   |cons hv nv tv, cons hw nw tw => (* n = S nv, n = S nw *) I
Coq < end.
```

Coq n'offre cette possibilité que de manière détournée par le biais de généralisation. Quand un terme dépend d'une variable, la généraliser suffit à ce que que la cœrcion soit faite par la clause de retour :

```
Coq < Definition map A m n (H : n = m) (l : listn A n) :=
Coq < match l in listn _ n0 return n0 = m -> _ with
Coq <   |nil => fun H' : 0 = m => I
Coq <   |cons h n t => fun H' : S n = m => I
Coq < end H.
```

Quand les dépendances sont issues de termes complexes (avec une fonction en tête) ou de variables déjà utilisées, la généralisation de l'égalité du terme à lui même (eq_refl de ce terme) permet de conserver l'information . C'est exactement ce que réalise Program.

```
Coq < Require Import Program.
```

```
Coq < Program Fixpoint nth A p q (l : listn A (S p + q)) :=
Coq < match l with
Coq < |nil => !
Coq < |cons h n t => match p with
Coq <   |0 => h
Coq <   |S p' => nth A p' q t
Coq < end end.
```

```
Coq < Print Term nth.
nth =
fix nth (A : Type) (p q : nat) (l : listn A (S p + q))
   {struct p} : A := match l as l0 in (listn _ n)
    return (n = S p + q -> l0 ~= l -> A)
  with
  | nil =>
     fun (Heq_anonymous : 0 = S p + q) (Heq_l : nil A ~= l) => !
  | cons h n t =>
      fun (Heq_anonymous : S n = S p + q)
        (Heq_l : cons A h n t ~= l) =>
      match p as p0 return (p0 = p -> A) with
      | 0 => fun _ : 0 = p => h
      | S p' =>
          fun Heq_p : S p' = p =>
          nth A p' q
            (eq_rect n (fun H : nat => listn A H) t
               (S p' + q)
               (nth_obligation_2 A p q l h n t Heq_anonymous
                  Heq_l p' Heq_p))
      end eq_refl
  end eq_refl JMeq_refl
     : forall (A : Type) (p q : nat), listn A (S p + q) -> A
```

Cette idée fonctionne extrêmement bien pour écrire des programmes complexes. Elle génère des termes qui se réduisent convenablement sur des entrées closes. Mais comme vu précédemment, les preuves sur l'égalité ajoutent des déchets pour le calcul qui restent et bloquent les réductions dans les preuves.

3.2 A partir de coupures commutatives et d'encore plus de filtrage

Quand le terme en tête de l'argument hors du filtrage est un constructeur, inverser l'égalité (c'est-à-dire déduire les égalités sur les arguments du constructeur) peut être fait directement dans la clause de retour. En effet, un constructeur est primitivement injectif (par définition du filtrage justement). Par conséquent, prouver n = S m → P m revient exactement à prouver match n with |S m → P m |_ → ID end.. L'avantage est énorme du fait que la clause de retour est complètement ignorée lors de l'élimination des inductifs. Les informations de typage ne gênent alors absolument pas le cours du calcul.

Remarque de formalisation : ID est le type de l'identité polymorphe (forall A : Type, A → A) et son habitant est id := fun A x => x. En réalité, n'importe quel type dont on aurait trivialement un habitant (comme True avec I) conviendrait pour le cas des branches de type incompatible. Le choix découle en réalité du sous-typage en Coq qui fait que ID : Type est plus général que True : Prop.

Prenons d'abord le cas des variables libres. Imaginons un environnement comprenant x de type P n et t un inductif de type I n sur lequel un filtrage est réalisé. Comme nous l'avons vu, la clause de retour va « généraliser » n dans I n avant de l'appliquer dans le cas des constructeurs. La variable x aura donc dans les branches un type déconnecté des constructeurs et sera inutilisable. La solution revient à réaliser une cœrcion sur le type de x par match t in I n0 return P n0 → ? with |constructeurs => fun x' => ? end x.

Si ensuite nous n'avons pas affaire à une variable libre mais à un terme constitué de variables et de constructeurs, la combinaison des systèmes donne satisfaction.

```
Coq < Inductive TyVar : nat -> Type :=
Coq < | ZTYVAR : forall u, TyVar (S u)
Coq < | STYVAR : forall u, TyVar u -> TyVar (S u).

Coq < Definition RTyL u w (r : TyVar u -> TyVar w) :
Coq <    TyVar (S u) -> TyVar (S w) :=
Coq <     fun var : TyVar (S u) => match var in TyVar n
Coq <       return match n with
Coq <         |0 => ID
Coq <         |S x => (TyVar x -> TyVar w) -> TyVar (S w) end
Coq <       with
Coq <       | ZTYVAR _ => fun _ => (ZTYVAR _)
```

```
Coq <       | STYVAR _ var' => fun r' => STYVAR _ (r' var')
Coq <   end r.
RTyL is defined
```

Pour poursuivre, si d'autres éléments dépendent du terme filtré en présence d'arguments avec constructeur, il faut généraliser le terme filtré dans le filtrage de la clause de retour. Voici le terme obtenu :

```
Coq < Inductive dummy : nat -> Prop := constr : dummy 0.
```

```
Coq < Definition failure : forall (x : dummy 0), x = constr :=
Coq < fun x => match x as x' in dummy n
Coq <   return match n as n' return dummy n' -> Prop with
Coq <      |0 => fun x0 => x0 = constr
Coq <      |S _ => fun _ => True
Coq <   end x' with
Coq <   constr => @eq_refl (dummy 0) constr end.
failure is defined
```

Enfin, si le type d'un constructeur est plus général que celui du terme hors du filtrage, un filtrage sur le bon argument du constructeur est nécessaire dans la branche en question.

```
Coq < Inductive UnitVector : nat -> Set :=
Coq < |UNil : UnitVector 0
Coq < |UBoxed : forall n, UnitVector n -> UnitVector n
Coq < |UCons : forall n, UnitVector n -> UnitVector (S n).
```

```
Coq < Fixpoint example n (v : UnitVector (S n)) : UnitVector n :=
Coq < match v as v' in UnitVector n'
Coq <  return match n' with
Coq <    |S n0 => UnitVector n0
Coq <    |_ => ID end with
Coq <  |UNil => @id
Coq <  |UBoxed n0 v => match n0 as n'
Coq <    return UnitVector n' -> match n' with
Coq <      |0 => ID
Coq <      |S n0 => UnitVector n0 end with
Coq <    |0 => fun _ => @id
Coq <    |S n0 => fun v' => example  n0 v' end v
Coq <  |UCons n' v => v
Coq < end.
Error : Cannot guess decreasing argument of fix.
```

Cet example ne passe pas la condition de garde de la version 8.3 de Coq car v' est une copie de v à laquelle a été appliquée une cœrcion de type et que les coupures commutatives ne sont pas prises en compte. D'où la nécessite de modifier l'algorithme de cette garde.

4 Une condition de garde structurelle sans réduction explicite

4.1 Spécifications/Cahier des charges

Comme signalé dans l'introduction, l'exploitation des valeurs inductives est réalisée par des points fixes. Comme tout terme récursif, ces termes sont des représentations finies d'objets potentiellement infinis. Il est essentiel pour la consistance du système que la réponse à une entrée close finie reste finie. C'est cela une condition de garde.

Tout d'abord, pour éviter que le système ne décrète que

```
fix lenght l:= match l
    with
  |nil => 0
  |cons _ l' => 1 +
    length l'
end.
```

=

```
fix lenght l := match l with
  |nil => 0
  |cons _ l' => 1 + (match l' with
     |nil => 0
     |cons _ l'' => 1 + length l'
  end)
end.
```

= ..., il est établi qu'un point fixe est remplacé par le corps de sa définition (se déplie) si et seulement si un constructeur se trouve en position récursive. Cette restriction n'est pas absolument suffisante et quand bien même la définition serait aussi correcte en théorie, il est inenvisageable d'accepter des termes de la forme de

```
fix lenght l := match l with |nil => 0 |cons _ l' => (match l' with |nil =>
                1 |cons h t => 1 + length (cons h t) end) end.
```

Plus profondément et fondamentalement, ne doivent être définis que des points fixes dont la réduction termine pour les termes clos. Pour certifier la terminaison, la méthode générique est d'exhiber une fonction qui à un terme associe une grandeur dans un treillis, grandeur choisie de telle manière que la valeur associée à l'appel récursif engendré par un terme soit strictement plus petite que la valeur associée au terme lui même.

En Coq, cette fonction est « le nombre de constructeurs en tête » du nième argument du point fixe. Cet argument est donc nécessairement de type inductif et ce n est donné par l'utilisateur (ou le système d'inférence de l'assistant). L'ordre utilisé est évidemment l'ordre sur les entiers naturels.

Remarque de formalisation :

1. Plutôt que de demander à l'utilisateur de préciser l'argument « décroissant », il serait possible sans perdre de généralité de considérer qu'il s'agit du dernier (car rien n'empêche de retourner une fonction à un point fixe). Par contre, la présence de type dépendant empêche d'introduire un lieur d'une seule variable. En effet, un inductif dépend de ses arguments et une listn n'a par exemple pas de sens sans sa taille.

2. Il arrive que plusieurs choix soient possibles et choisir l'argument que l'on qualifie de récursif induit de nombreuses conséquences sur le comportement calculatoire du terme ouvert (lors de preuves). Réaliser ce choix est donc un sujet essentiel d'ingénierie pour le programmeur avec types dépendants.

4.2 Fonctionnement

La première idée pour implémenter un algorithme qui garantit la garde est de réaliser une analyse syntaxique du terme dont le principe est :

Dans un premier temps, à chaque variable (argument du point fixe ou d'un lambda) est associée la spécification *non_sous_terme* hormis l'argument récursif pour laquelle elle est *sous_terme_large*. Lorsqu'un filtrage est réalisé sur une variable *sous_terme*, les arguments des constructeurs ont la spécification *sous_terme_strict*. Pour tout filtrage sur autre chose, les arguments des constructeurs sont aussi *non_sous_terme*.

Dans un second temps, la position récursive de tous les appels récursifs réalisés doit être occupée par une variable de spécification *sous_terme_strict*.

Ce jugement est extrêmement limité car il empêche toute coupure donc toute factorisation de code. Par exemple, tail l est sous terme strict de l mais l'algorithme ne faisant pas de réduction l'ignore.

La deuxième idée est alors de mettre le corps du point fixe sous forme normale avant d'appliquer l'algorithme ci-dessus. Coq implémente actuellement cette solution mais elle entraîne une perte d'information et ne garantit plus la normalisation forte comme l'illustre

```
Coq < Fixpoint f (n : nat) : nat := (fun x => 3) (f n).
f is recursively defined (decreasing on 1st argument)
```

Aucune solution ne s'est encore imposée mais l'un des axes les plus creusés actuellement est de définir des jugements de typage pour associer une « taille » aux termes afin d'être plus rigoureux, général et modulaire.

En attendant que ces belles solutions soient implémentées, il est possible d'écrire un jeu de règles d'inférences qui réalisent implicitement des β-réductions mais en les contrôlant avant. En plus, cette idée permet de réduire les rédex séparés par une coupure commutative qui ne sont pas réduits par les réductions explicites. Typiquement, dire que x et y ont la même spécification dans (match ...with |...=> fun y => ...end) x

Concrètement, nos jugements seront toujours dirigés par la syntaxe des termes et contiendront

- f une table qui à chaque point fixe associe le numéro de son argument récursif.
- Γ un contexte où seront stockées la valeur et la spécification de chacune des variables libres du terme étudié

– s une pile de termes qui représente les arguments appliqués au terme étudié
– t le terme étudié lui même.

Cette fois, les spécifications sont

non_sous_terme pour les arguments du point fixe et les arguments des constructeurs issus de la destruction d'un terme *non_sous_terme*

sous_terme_large pour l'argument récursif du point fixe,

sous_terme_strict pour les produits de la destruction d'un sous terme.

Enfin, sont associés à la variable d'un lambda la spécification et la valeur du terme en tête de pile si celle-ci est non vide et *neutre* sinon.

Et les appels récursifs dans le corps du point fixe sont contrôlés de la manière suivante :

– Lorsque nous rencontrerons une application, nous vérifierons que l'argument ne réalise des appels que sur des termes de spécification *sous_terme_strict* OU *neutre* puis empilerons l'argument pour contrôler le terme en tête.
– un lambda déplacera un terme de la pile à l'environnement.
– les variables seront remplacées par leur définition dans l'environnement.
– dans les autres cas et pour les termes restant à la fin dans la pile, nous imposerons que tous les appels récursifs soient fait sur des termes de spécification *sous_terme_strict*.

Un problème persiste, le système présenté ici refuse à tort

```
Fixpoint f (n: nat): nat :=
match n with
 |0 => 0
 |S n' => match S n' with
    |0 => 12
    |S n0 => f n0
 end
end.
```

car dans l'état actuel des choses (S n') a la spécification *non_sous_terme*. Il faudrait trouver un moyen d'encoder les ι-réductions.

Conclusion

Au final, ce stage fut l'occasion de récapituler les forces et les limites de Coq en tant que langage de programmation avec types dépendants. Sans être capable de donner aujourd'hui la prochaine règle de typage du filtrage qu'il sera souhaitable d'adopter, nous sommes en mesure de bien mieux décrire son cahier des charges. De même, des idées d'uniformisation de la syntaxe afin de faciliter l'écriture directe de programme ont émergé.

Surtout,

– Coq dispose maintenant d'une nouvelle règle pour assurer la terminaison des points fixes, règle plus uniforme, plus rigoureuse et dont la certification formelle (en cours) est plus aisée.
– La librairie standard contient des bibliothèques pour fin n (ensembles à n éléments) et vector A n (listes de A de taille n) dont les points fixes se déplient suffisamment sur des termes ouverts pour être exploitables dans des preuves. Ces structures sont les prémisses du passage de structures de données munies de spécification à des structures de données à invariants très forts.
– L'algorithme qui devine des clauses de retour gagne en généralité. La puissance de certaines tactiques pourra très facilement être augmenté dans le futur en se basant sur ce travail.
– Enfin, bien que d'autres directions aient été suivies ensuite, un début d'infrastructure a été pensé pour étendre les capacités de inversion. Son apport sera certain vu la voie que prennent les impléméntations modernes de filtrage dépendant.

Références

[1] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

[2] Bruno Barras, Pierre Corbineau, Benjamin Grégoire, Hugo Herbelin, and Jorge Luis Sacchini. A new elimination rule for the calculus of inductive constructions. *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*, 5497 :32–48, 2008.

[3] Gilles Barthe, Benjamin Grégoire, and Colin Riba. A tutorial on type-based termination. *Book chapter, LERNET 2008 Summer School*, 2008.

[4] The Agda Community. *The Agda Wiki*.

[5] Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages : Theory and Practice (LFMTP'08)*, Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, June 2008.

[6] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *TYPES*, pages 39–59, 1994.

[7] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1) :69–111, 2004.

[8] Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volume 2277 of *LNCS*. Springer-Verlag, 2002.

[9] The Coq Team. *Coq Reference Manual*.