Calcul des Constructions Inductives
D.E.A. Programmation: Sémantique, Preuves et Langages
2003-2004

Calcul des Constructions Inductives

Christine Paulin-Mohring, Benjamin Werner,

Bruno Barras, Hugo Herbelin,

Jean-Christophe Filliâtre, Claude Marché

Table des matières

Chapitre 6

Preuve de programmes fonctionnels

Dans ce cours, nous nous intéressons à la spécification et à la preuve de programmes purement fonctionnels. Nous montrons comment `Coq` peut être utilisé pour produire du code ML certifié. Ce chapitre est illustré par des programmes manipulant des arbres binaires de recherche, qui sont représentatifs des programmes purement fonctionnels à la fois complexes et très utiles en pratique.

Dans la suite, nous dénommons *informatif* tout ce qui se trouve dans la sorte `Set` et *logique* tout ce qui se trouve dans la sorte `Prop`. Cette distinction de sorte est exploitée par le mécanisme d'*extraction* [PM89a, PM89b, Let03a, Let03b] fourni par `Coq`. Ce mécanisme extrait le contenu informatif d'un terme `Coq` sous la forme d'un programme ML. Les parties logiques disparaissent (ou ne subsistent que sous la forme d'une valeur dégénérée sans aucun calcul associé). Les fondements théoriques de l'extraction sont exposés dans l'annexe A.

6.1 Méthode directe

La façon la plus simple de certifier un programme purement fonctionnel consiste à l'écrire comme une fonction dans `Coq` puis à prouver des propriétés de cette fonction. C'est ce qui a été fait par exemple dès le début de ce cours avec la `plus` sur les entiers de Peano. De manière générale, un grand nombre de programmes ML purement fonctionnels peuvent être écrits directement dans le Calcul des Constructions.

D'une manière générale, on commence par définir dans `Coq` une fonction « pure », c'est-à-dire avec un type à la ML (un type du système F) purement informatif. Supposons ici une fonction prenant un seul argument :

$$f \quad : \quad \tau_1 \rightarrow \tau_2$$

On montre alors que cette fonction réalise une certaine spécification $S : \tau_1 \rightarrow \tau_2 \rightarrow \texttt{Prop}$ par un théorème de la forme

$$\forall x.\ (S\ x\ (f\ x))$$

La preuve de ce théorème se fait en suivant la définition de f.

Exemple. On souhaite développer et certifier formellement une bibliothèque d'ensembles finis codés à l'aide d'arbres binaires de recherche.

On se donne un type d'arbres binaires contenant des entiers

```
Coq <   Inductive tree : Set :=
Coq <   | Empty
Coq <   | Node : tree -> Z -> tree -> tree.
```

et une relation d'appartenance `In` indiquant qu'un élément apparaît dans un arbre (indépendamment de tout choix de rangement des éléments dans les arbres) :

```
Coq <   Inductive In (x:Z) : tree -> Prop :=
Coq <   | In_left : forall l r y, (In x l) -> (In x (Node l y r))
Coq <   | In_right : forall l r y, (In x r) -> (In x (Node l y r))
Coq <   | Is_root : forall l r, (In x (Node l x r)).
```

Une fonction de test de l'ensemble vide peut alors s'écrire

```
Coq <   Definition is_empty (s:tree) : bool := match s with
Coq <   | Empty => true
Coq <   | _ => false end.
```

et sa preuve de correction s'énonce ainsi

```
Coq <   Theorem is_empty_correct :
Coq <   forall s, (is_empty s)=true <-> (forall x, ~(In x s)).
```

La preuve suit la définition de `is_empty` et tient en trois lignes :

```
Coq < Proof.
Coq <   destruct s; simpl; intuition.
Coq <   inversion_clear H0.
Coq <   elim H with z; auto.
Coq < Qed.
```

Venons-en au test d'occurrence dans un arbre binaire de recherche. On commence par se donner une relation d'ordre ternaire sur les entiers relatifs :

```
Coq <   Inductive order : Set := Lt | Eq | Gt.
Coq <   Hypothesis compare : Z -> Z -> order.
```

Puis on définit une fonction `mem` de recherche dans un arbre `supposé` être un arbre de recherche :

```
Coq <   Fixpoint mem (x:Z) (s:tree) {struct s} : bool := match s with
Coq <   | Empty =>
Coq <      false
Coq <   | Node l y r => match compare x y with
Coq <      | Lt => mem x l
Coq <      | Eq => true
Coq <      | Gt => mem x r
Coq <     end
Coq <   end.
```

La preuve de correction de cette fonction nécessite de définir la notion d'arbre binaire de recherche, sous la forme du prédicat inductif `bst` suivant :

```
Coq <   Inductive bst : tree -> Prop :=
Coq <   | bst_empty :
Coq <      (bst Empty)
Coq <   | bst_node :
Coq <      forall x (l r : tree),
Coq <      bst l -> bst r ->
Coq <      (forall y, In y l -> y < x) ->
Coq <      (forall y, In y r -> x < y) -> bst (Node l x r).
```

La correction de la fonction `mem` peut alors s'écrire ainsi :

```
Coq <   Theorem mem_correct :
Coq <     forall x s, (bst s) -> (mem x s=true <-> In x s).
```

On voit sur cet exemple que la spécification S prend la forme $P\ x \rightarrow Q\ x\ (f\ x)$. P est ce que l'on appelle une *précondition* et Q une *postcondition*.

Modularité. Si l'on cherche à faire la preuve de `mem_correct` on commence par `induction s; simpl` pour suivre la définition de `mem`. Le premier cas (`Empty`) est trivial. Avec le second (`Node s1 z s2`) on tombe alors sur un terme de la forme `match compare x z with ...` qui ne permet pas d'aller plus avant. En effet, on ne sait rien de la fonction `compare` utilisée ici par la fonction `mem`. Il nous faut la spécifier, par exemple sous la forme d'un axiome

```
Coq <   Hypothesis compare_spec :
Coq <     forall x y, match compare x y with
Coq <     | Lt => x<y
Coq <     | Eq => x=y
Coq <     | Gt => x>y
Coq <     end.
```

On peut alors utiliser cette spécification de la manière suivante :

```
Coq <    generalize (compare_spec x z); destruct (compare x z).
```

La preuve se poursuit alors sans difficulté.

Note. Pour des fonctions purement informatives telles que `is_empty` ou `mem`, le programme extrait est identique au terme `Coq`. Ainsi la commande `Extraction mem` donne-t-elle le code ocaml

```
let rec mem x = function
  | Empty -> false
  | Node (l, y, r) ->
      (match compare x y with
         | Lt -> mem x l
         | Eq -> true
         | Gt -> mem x r)
```

6.1.1 Cas des fonctions partielles

Une première difficulté apparaît lorsque la fonction est partielle, i.e. a un type de la forme

$$f : \forall x : \tau_1.\ (P\ x) \rightarrow \tau_2$$

Le cas typique est celui d'une fonction de division qui attend un diviseur non nul.

Dans notre exemple, on peut souhaiter définir une fonction `min_elt` retournant le plus petit élément d'un ensemble supposé non vide (c'est-à-dire l'élément rangé le plus à gauche dans l'arbre binaire de recherche). On peut donner à cette fonction le type suivant :

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \rightarrow \texttt{Z} \tag{6.1}$$

où apparaît la précondition $\neg s = \texttt{Empty}$. La spécification de `min_elt` peut alors s'écrire

$$\forall s.\ \forall h : \neg s = \texttt{Empty}.\ \texttt{bst}\ s \rightarrow \texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s \wedge \forall x.\ \texttt{In}\ x\ s \rightarrow \texttt{min\_elt}\ s\ h \leq x$$

avec une précondition reprenant naturellement celle de la fonction (hypothèse h) et ajoutant `bst` s. La présence de h est même nécessaire pour pouvoir appliquer la fonction `min_elt`. On voit que l'utilisation d'une fonction partielle en `Coq` n'est pas simple : il faut passer en argument des termes de preuve, souvent difficiles à construire.

La définition même d'une fonction partielle est souvent délicate. Écrivons une fonction `min_elt` ayant le type (6.1). Le code ML que l'on a en tête est le suivant :

```
let rec min_elt = function
  | Empty -> assert false
  | Node (Empty, x, _) -> x
  | Node (l, _, _) -> min_elt l
```

Malheureusement la définition en `Coq` est plus difficile. D'une part l'`assert false` dans le premier cas de filtrage correspond à un cas absurde (on a supposé l'arbre non vide). La définition en `Coq` exprime cette absurdité à l'aide du récurseur `False_rec` appliqué à une preuve de l'absurde. Il faut donc construire une telle preuve à partir de la précondition. De même le troisième cas de filtrage fait un appel récursif à `min_elt` et pour cela on doit construire une preuve que l'argument `l` n'est pas vide. C'est ici une conséquence du filtrage qui a éliminé le cas où `l` est `Empty`. Dans ces deux cas la nécessité de construire ces termes de preuve complique le filtrage, qui doit être dépendant. On obtient la définition suivante :

```
Coq < Fixpoint min_elt (s:tree) (h:~s=Empty) { struct s } : Z :=
Coq <   match s return ~s=Empty -> Z with
Coq <   | Empty =>
Coq <      (fun h => False_rec _ (h (refl_equal Empty)))
Coq <   | Node l x _ =>
Coq <      (fun h => match l as a return a=l -> Z with
Coq <                | Empty => (fun _ => x)
Coq <                | _ => (fun h => min_elt l
Coq <                                  (Node_not_empty _ _ _ _ h))
Coq <                end (refl_equal l))
Coq <   end h.
```

La première preuve (argument de `False_rec`) est construite directement. La seconde fait appel au lemme suivant :

```
Coq < Lemma Node_not_empty : forall l x r s, Node l x r=s -> ~s=Empty.
```

On peut alors prouver la correction de cette fonction :

```
Coq < Theorem min_elt_correct :
Coq <   forall s (h:~s=Empty), bst s ->
Coq <     In (min_elt s h) s /\
Coq <     forall x, In x s -> min_elt s h <= x.
```

Là encore la preuve se fait en suivant la définition de la fonction et ne pose pas de problème.

On peut vérifier que le code extrait est bien celui que l'on avait en tête. `Extraction min_elt` donne en effet :

```
let rec min_elt = function
  | Empty -> assert false (* absurd case *)
  | Node (l, x, t) ->
      (match l with
         | Empty -> x
         | Node (t0, z0, t1) -> min_elt l)
```

Plusieurs points sont à noter : d'une part l'utilisation de `False_rec` est extraite en `assert false`, ce qui est exactement le comportement souhaité (on a fait une preuve que ce point de programme est inatteignable, il est donc légitime de dire qu'arriver là est absurde i.e. une « preuve » de `false`) ; d'autre part on voit que les arguments logiques liés à la précondition qui compliquaient la définition ont disparu dans le code extrait (ils étaient dans la sorte `Prop`).

Une autre solution consiste à définir la fonction `min_elt` par une preuve plutôt que par une définition. Il est alors facile de construire les termes de preuve (on est dans l'éditeur de preuves). Ici la définition-preuve est relativement simple :

```
Coq < Definition min_elt : forall s, ~s=Empty -> Z.

Coq < Proof.

Coq < induction s; intro h.

Coq < elim h; auto.

Coq < destruct s1.

Coq < exact z.

Coq < apply IHs1; discriminate.

Coq < Defined.
```

Mais il est plus difficile de se persuader que l'on construit la bonne fonction (tant que l'on n'a pas montré la correction de cette fonction). Il faut en particulier prendre bien soin de ne pas utiliser à tort de tactiques automatiques telles que `auto` qui pourraient avoir pour effet de construire une fonction autre que celle que l'on a en tête ; sur cet exemple `auto` n'est utilisé que sur un but logique (`Empty=Empty`).

Une façon de se persuader que le code sous-jacent est bien le bon consiste à examiner le code extrait. Ici on retrouve exactement le même que précédemment.

La tactique `refine` aide à la définition de fonction partielle (mais pas seulement). Elle permet de donner partiellement un terme de preuve, certaines parties étant omises (dénotées par `_`) et donnant lieu à des sous-buts. Ainsi on peut redéfinir la fonction `min_elt` de la façon suivante :

```
Coq < Definition min_elt : forall s, ~s=Empty -> Z.

Coq < refine
Coq <   (fix min (s:tree) (h:~s=Empty) { struct s } : Z :=
Coq <   match s return ~s=Empty -> Z with
Coq <   | Empty =>
Coq <       (fun h => _)
Coq <   | Node l x _ =>
Coq <       (fun h => match l as a return a=l -> Z with
Coq <                 | Empty => (fun _ => x)
Coq <                 | _ => (fun h => min_elt l _)
Coq <                 end _)
Coq <   end h).
```

On obtient alors deux sous-buts qu'il est aisé de prouver. On remarque tout de même que l'on n'échappe pas à un filtrage dépendant.

Enfin une dernière solution consiste à rendre la fonction totale en la complétant de manière arbitraire en dehors de son domaine de définition. Ici on peut choisir de retourner la valeur 0 lorsque l'ensemble est vide. On évite ainsi l'argument logique $\neg s =$ `Empty` et ses fâcheuses conséquences. Le type de `min_elt` « redevient» `tree` → `Z` et sa définition est très simple :

```
Coq < Fixpoint min_elt (s:tree) : Z := match s with
Coq <   | Empty => 0
Coq <   | Node Empty z _ => z
Coq <   | Node l _ _ => min_elt l
Coq < end.
```

Le théorème de correction reste le même, en revanche :

```
Coq < Theorem min_elt_correct :
Coq <   forall s, ~s=Empty -> bst s ->
Coq <     In (min_elt s) s /\
Coq <     forall x, In x s -> min_elt s <= x.
```

L'énoncé garde la précondition $\neg s =$ `Empty`, sans quoi il ne serait pas possible de montrer l'appartenance de `min_elt` s à s.

Note. La fonction de division sur les entiers relatifs, `Zdiv`, est ainsi définie comme une fonction totale mais ces propriétés ne sont établies que sous l'hypothèse que le diviseur est non nul.

Note. Une autre façon de rendre totale la fonction `min_elt`, plus générale, eût été de lui faire retourner un résultat de type `option Z`, c'est-à-dire `None` lorsque l'ensemble est vide, et `Some` m lorsqu'il existe un plus petit élément m. Mais on change alors légèrement le code ML sous-jacent (et l'énoncé du théorème de correction).

6.1.2 Cas des fonctions non structurellement récursives

Le problème de la définition (et de l'utilisation) d'une fonction partielle se retrouve également lorsque l'on cherche à définir (et à prouver) une fonction récursive mais dont la récurrence n'est pas structurelle.

En effet, une solution pour définir une telle fonction consiste à utiliser un principe de récurrence bien fondée, tel que `well_founded_induction` :

```
Coq < well_founded_induction
Coq <      : forall (A : Set) (R : A -> A -> Prop),
Coq <        well_founded R ->
Coq <        forall P : A -> Set,
Coq <        (forall x : A, (forall y : A, R y x -> P y) -> P x) ->
Coq <        forall a : A, P a
```

Mais alors la définition nécessite de construire des preuves de `R y x` pour chaque appel récursif sur `y` ; on retrouve les difficultés — mais aussi les solutions — mentionnées dans la section précédente.

Supposons que l'on souhaite écrire une fonction `subset` qui teste l'inclusion sur nos ensembles comme arbres binaires de recherche. Un code ML possible est le suivant :

```
let rec subset s1 s2 = match (s1, s2) with
  | Empty, _ ->
      true
  | _, Empty ->
      false
  | Node (l1, v1, r1), (Node (l2, v2, r2) as t2) ->
      let c = compare v1 v2 in
      if c = 0 then
        subset l1 l2 && subset r1 r2
      else if c < 0 then
        subset (Node (l1, v1, Empty)) l2 && subset r1 t2
      else
        subset (Node (Empty, v1, r1)) r2 && subset l1 t2
```

On voit que les appels récursifs se font sur des arbres qui ne sont pas toujours des sous-termes des arguments initiaux (sans compter la difficulté supplémentaire d'une récurrence simultanée sur les deux arguments). Il existe cependant un critère simple de terminaison, à savoir le nombre total d'éléments dans les deux arbres.

On commence donc par établir un principe de récurrence bien fondée sur deux arbres basé sur la somme de leur nombre d'éléments :

```
Coq < Fixpoint cardinal_tree (s:tree) : nat := match s with
Coq <   | Empty => 0
Coq <   | Node l _ r => (S (plus (cardinal_tree l) (cardinal_tree r)))
Coq < end.

Coq <
Coq < Lemma cardinal_rec2 :
Coq <   forall (P:tree->tree->Set),
Coq <   (forall (x x':tree),
Coq <      (forall (y y':tree),
Coq <        (lt (plus (cardinal_tree y) (cardinal_tree y'))
Coq <            (plus (cardinal_tree x) (cardinal_tree x'))) -> (P y y'))
Coq <      -> (P x x')) ->
Coq <   forall (x x':tree), (P x x').
```

La preuve est facile : on se ramène à un principe de récurrence bien fondée sur le type `nat`, fourni dans la bibliothèque de `Coq`, à savoir `well_founded_induction_type_2`, et l'on prouve aisément que la relation est bien fondée car elle est de la forme `lt` $(f\ y\ y')$ $(f\ x\ x')$ et que `lt` est une relation bien fondée sur `nat` (autre résultat fourni par la bibliothèque) :

```
Coq < Proof.

Coq <   intros P H x x'.

Coq <   apply well_founded_induction_type_2
Coq <       with (R:=fun (yy' xx':tree*tree) =>
Coq <               (lt (plus (cardinal_tree (fst yy')) (cardinal_tree (snd yy')))
Coq <                   (plus (cardinal_tree (fst xx')) (cardinal_tree (snd xx'))))));
Coq <     auto.

Coq <     apply (Wf_nat.well_founded_ltof _
Coq <            (fun (xx':tree*tree) =>
Coq <                 (plus (cardinal_tree (fst xx')) (cardinal_tree (snd xx'))))).

Coq < Save.
```

On peut alors définir la fonction `subset` par une définition-preuve utilisant la tactique `refine` :

```
Coq < Definition subset : tree -> tree -> bool.

Coq < Proof.
```

On commence par appliquer le principe de récurrence `cardinal_rec2` :

```
Coq <   intros s1 s2; pattern s1, s2; apply cardinal_rec2.
```

Puis on filtre sur `x` et `x'` les deux cas `Empty` étant triviaux :

```
Coq <   destruct x.

Coq <   (* x=Empty *)
Coq <   intros; exact true.

Coq <   (* x = Node x1 z x2 *)
Coq <   destruct x'.

Coq <   (* x'=Empty *)
Coq <   intros; exact false.
```

On procède ensuite par cas sur le résultat de `compare z z0` :

```
Coq <   (* x' = Node x'1 z0 x'2 *)
Coq <   intros; case (compare z z0).
```

Dans chacun des trois cas, les appels récursifs (hypothèse `H`) se font à l'aide de la tactique `refine` : on a alors une obligation de montrer la décroissance du nombre d'éléments, ce qui est automatiquement prouvé par `simpl; omega` (`simpl` est nécessaire pour déplier la définition de `cardinal_tree`) :

```
Coq <   (* z < z0 *)
Coq <   refine (andb (H (Node x1 z Empty) x'2 _)
Coq <                (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
```

```
Coq <   (* z = z0 *)
Coq <   refine (andb (H x1 x’1 _) (H x2 x’2 _)); simpl; omega.

Coq <   (* z > z0 *)
Coq <   refine (andb (H (Node Empty z x2) x’2 _)
Coq <                (H x1 (Node x’1 z0 x’2) _)); simpl; omega.

Coq < Defined.
```

Note. On aurait pu également faire une définition à l'aide d'un unique `refine`.

Note. Il est intéressant d'examiner le code extrait d'une fonction définie à l'aide d'un récurseur tel que `well_founded_induction`. On peut commencer par regarder directement le code extrait pour `well_founded_induction` et l'on reconnaît un opérateur de point-fixe :

```
let rec well_founded_induction x a =
  x a (fun y _ -> well_founded_induction x y)
```

En dépliant cet opérateur et deux autres constantes

```
Coq < Extraction NoInline andb.

Coq < Extraction Inline cardinal_rec2 Acc_iter_2 well_founded_induction_type_2.

Coq < Extraction subset.
```

on obtient exactement le code ML souhaité :

```
let rec subset x x’ =
  match x with
    | Empty -> True
    | Node (x1, z0, x2) ->
        (match x’ with
           | Empty -> False
           | Node (x’1, z1, x’2) ->
               (match compare z0 z1 with
                  | Lt ->
                      andb (subset (Node (x1, z0, Empty)) x’2)
                        (subset x2 (Node (x’1, z1, x’2)))
                  | Eq -> andb (subset x1 x’1) (subset x2 x’2)
                  | Gt ->
                      andb (subset (Node (Empty, z0, x2)) x’2)
                        (subset x1 (Node (x’1, z1, x’2)))))
```

6.2 Utilisation de types dépendants

Une autre approche de la preuve de programmes fonctionnels dans `Coq` consiste à utiliser la richesse du système de types du Calcul des Constructions Inductives pour exprimer la spécification de la fonction dans son type même. En fait, un type *est* une spécification. Dans le cas de ML, c'est juste une spécification très pauvre — une fonction attend un entier et retourne un entier — mais dans `Coq` on peut exprimer qu'une fonction attend un entier positif et retourne un entier premier :

$$f : \{n : Z \mid n \geq 0\} \to \{p : Z \mid \texttt{premier}\ p\}$$

Nous allons montrer comment.

6.2.1 Type sous-ensemble `sig`

La notation `Coq` $\{x : A \mid P\}$ désigne le « sous-type de A des valeurs vérifiant la propriété P » ou, dans un vocabulaire plus théorie des ensembles, le « sous-ensemble de A des éléments vérifiant P ». La notation $\{x : A \mid P\}$ désigne l'application `sig` A (`fun` $x \Rightarrow P$) où `sig` est l'inductif suivant :

```
Coq < Inductive sig (A : Set) (P : A -> Prop) : Set :=
Coq <   exist : forall x:A, P x -> sig P
```

Cet inductif est identique à l'existentielle `ex`, si ce n'est sa sorte, `Set` au lieu de `Prop` (on souhaite définir une fonction et donc que ses arguments et résultats soient informatifs).

En pratique, on souhaite lier l'argument au résultat par une postcondition Q et on préfère donc la forme plus générale suivante :

$$f : \forall\ (x : \tau_1),\ P\ x \to \{y : \tau_2 \mid Q\ x\ y\}$$

Si l'on reprend l'exemple de la fonction `min_elt`, sa spécification peut être la suivante :

```
Coq < Definition min_elt :
Coq <   forall s, ~s=Empty -> bst s ->
Coq <     { m:Z | In m s /\ forall x, In x s -> m <= x }.
```

On a toujours les difficultés de définition directe mentionnées dans la section précédente et l'on préfère donc en général la définition par preuve (avec toujours la même mise en garde vis-à-vis des méthodes automatiques).

Note. Le déplacement de la propriété `bst s` vers la précondition n'est pas nécessaire ; c'est juste plus naturel.

Note. L'extraction de `sig` A Q oublie l'annotation logique Q et se réduit donc à l'extraction du type A. Dit autrement, le type `sig` peut disparaître à l'extraction ; de fait on a

```
Coq < Extraction sig.
type ’a sig = ’a
  (* singleton inductive, whose constructor was exist *)
```

6.2.2 Variantes de `sig`

On peut définir d'autres types similaires à `sig`. Ainsi si l'on souhaite écrire une fonction retournant deux entiers, telle que par exemple une division euclidienne, on a envie d'emboîter deux utilisation de `sig` de la même manière que l'on peut le faire pour deux existentielles `ex` :

$$\texttt{div} : \forall a\ b,\ b > 0 \to \{q \mid \{r \mid a = bq + r \wedge 0 \leq r < b\}\}$$

Mais la seconde utilisation de `sig` a pour sorte `Set` et non `Prop`, ce qui rend cette écriture incorrecte. `Coq` introduit pour cela une variante de `sig`, `sigS` :

```
Coq < Inductive sigS (A : Set) (P : A -> Set) : Set :=
Coq <   existS : forall x:A, P x -> sig P
```

où la seule différence est la sorte de `P` (`Set` au lieu de `Prop`). `sigS` A (`fun` $x \Rightarrow P$) se note $\{x : A \ \&\ P\}$, ce qui permet d'écrire

$$\texttt{div} : \forall a\ b,\ b > 0 \rightarrow \{q\ \&\ \{r \mid a = bq + r \wedge 0 \leq r < b\}\}$$

L'extraction de `sigS` est naturellement une paire :

```
Coq < Extraction sigS.
type (’a, ’p) sigS =
  | ExistS of ’a * ’p
```

De même si l'on souhaite une spécification de la forme

$$\{x : A \mid P\ x \wedge Q\ x\}$$

il existe un inductif « sur mesure », `sig2`, défini par

```
Coq < Inductive sig2 (A : Set) (P : A -> Prop) (Q : A -> Prop) : Set :=
Coq <   exist2 : forall x : A, P x -> Q x -> sig2 P Q
```

Son extraction est identique à celle de `sig`.

6.2.3 Spécification d'une fonction booléenne : `sumbool`

Un type de spécification[1] qui revient très souvent est celui de la spécification d'une fonction booléenne. Dans ce cas, on souhaite exprimer quelles sont les deux propriétés établies lorsque la fonction retourne `false` et `true` respectivement. Coq introduit un type inductif pour cela, `sumbool`, défini par

```
Coq < Inductive sumbool (A : Prop) (B : Prop) : Set :=
Coq <   | left : A -> sumbool A B
Coq <   | right : B -> sumbool A B
```

C'est un type semblable au type `bool` mais dont chaque constructeur contient une preuve, de `A` et de `B` respectivement. `sumbool A B` se note `{A}+{B}`. Une fonction de test de l'ensemble vide pourra se spécifier ainsi :

$$\texttt{is\_empty} : \forall s,\ \{s = \texttt{Empty}\} + \{\neg s = \texttt{Empty}\}$$

Un cas plus général, et très fréquent, est celui d'une égalité décidable. En effet, si un type `A` est muni d'une égalité `eq` : `A` → `A` → `Prop`, on peut spécifier une fonction de test de cette égalité sous la forme

$$\texttt{A\_eq\_dec} : \forall x\ y,\ \{\texttt{eq}\ x\ y\} + \{\neg(\texttt{eq}\ x\ y)\}$$

C'est presque la même chose que donner une preuve de

$$\forall x\ y,\ (\texttt{eq}\ x\ y) \vee \neg(\texttt{eq}\ x\ y)$$

si ce n'est que la sorte n'est pas la même. Dans ce dernier cas, on a une disjonction dans `Prop` (un tiers-exclu pour le prédicat `eq`) alors que dans le précédent on a une « disjonction » dans `Set`, c'est-à-dire un programme décidant de l'égalité.

L'extraction de `sumbool` est un type isomorphe à `bool` :

---

[1] C'est le cas de le dire !

```
Coq < Extraction sumbool.
type sumbool =
  | Left
  | Right
```

En pratique on peut indiquer à l'extraction de Coq d'utiliser directement les booléens de ML au lieu de `Left` et `Right` (permet notamment d'utiliser `if-then-else` dans le code extrait).

Variante `sumor`

Il existe une variante à `sumbool` où les sortes ne sont pas les mêmes à gauche et à droite :

```
Coq < Inductive sumor (A : Set) (B : Prop) : Set :=
Coq <   | inleft : A -> A + {B}
Coq <   | inright : B -> A + {B}
```

Cet inductif permet de spécifier une fonction ML qui retourne une valeur du type α `option` : le constructeur `inright` représente le cas `None` et lui associe la propriété `B`, et le constructeur `inleft` représente le cas `Some` et lui associe la spécification `A`. De fait, l'extraction de `sumor` est isomorphe au type `option` de ML :

```
Coq < Extraction sumor.
type ’a sumor =
  | Inleft of ’a
  | Inright
```

On peut ainsi combiner `sumor` et `sig` pour spécifier la fonction `min_elt` de la manière suivante :

```
Coq < Definition min_elt :
Coq <   forall s, bst s ->
Coq <   { m:Z | In m s /\ forall x, In x s -> m <= x } + { s=Empty }.
```

Il s'agit là de la version correspondant à une fonction ML rendue totale avec un type `option`.

On peut de même combiner `sumor` et `sumbool` pour spécifier notre fonction de comparaison ternaire :

```
Coq < Hypothesis compare : forall x y, {x<y} + {x=y} + {x>y}.
```

On note que maintenant cette seule hypothèse remplace à elle seule l'inductif `order` et les deux hypothèses `compare` et `compare_spec`.

Reprenons l'exemple de la fonction de test d'appartenance dans un arbre binaire de recherche, `mem`. On peut maintenant la spécifier à l'aide d'un type dépendant :

```
Coq < Definition mem :
Coq <   forall x s, bst s -> { In x s }+{ ~(In x s) }.
```

La définition-preuve commence par une induction sur `s`.

```
Coq < Proof.

Coq <   induction s; intros.

Coq <   (* s = Empty *)
Coq <   right; intro h; inversion_clear h.
```

Le cas `s=Empty` est trivial. Dans le cas `s=Node s1 z s2`, il s'agit de procéder par cas sur le résultat de `compare x z`. C'est maintenant plus simple qu'avec la méthode précédente : plus besoin de faire appel au lemme `compare_spec`, car `compare x z` contient sa spécification dans son type.

```
Coq <   (* s = Node s1 z s2 *)
Coq <   case (compare x z); intro.
```

De même chaque hypothèse de récurrence (sur `s1` et `s2`) est une fonction contenant sa spécification. On l'utilise, le cas échéant, en lui appliquant la tactique `case`. Le reste de la preuve est aisé.

Note. Il est possible de retrouver la fonction pure comme projection de la fonction spécifiée à l'aide d'un type dépendant :

```
Coq < Definition mem_bool x s (h:bst s) := match mem x s h with
Coq <   | left _ => true
Coq <   | right _ => false
Coq <   end.
```

Il est alors aisé de montrer la correction de cette fonction pure (car la preuve est contenue dans le type de la fonction d'origine) :

```
Coq < Theorem mem_bool_correct :
Coq <   forall x s, forall (h:bst s),
Coq <   (mem_bool x s h)=true <-> In x s.

Coq < Proof.

Coq <   intros.

Coq <   unfold mem_bool; simpl; case (mem x s h); intuition.

Coq <   discriminate H.

Coq < Qed.
```

Mais cette projection a peu d'intérêt en pratique.

Note. Il est important de noter que chaque fonction se voit maintenant donner sa spécification dès sa définition : il n'est plus aussi facile de montrer plusieurs propriétés d'une même fonction que dans le cas d'une fonction pure.

6.2.4 Spécification dans les types de données

L'ajout de spécification dans les types ML peut également s'appliquer aux types récursifs. Ainsi on peut introduire le type dépendant des arbres ayant la propriété d'être des arbres binaires de recherche :

```
Coq < Inductive bst_tree : Set :=
Coq <   | Bst_tree : forall t, bst t -> bst_tree.
```

Il s'agit là d'un couple dépendant constitué d'un arbre `t` (dans la sorte `Set`) et d'une preuve de `bst t` (dans la sorte `Prop`). Un tel inductif a un constructeur est un *record* (type enregistrement) :

```
Coq < Record bst_tree : Set := {
Coq <   t     :> tree;
Coq <   bst_t :  bst t
Coq < }.
```

Note. La notation `:>` introduit une coercion du type `bst_tree` vers le type `tree` (la première projection). Ceci permet par exemple d'appliquer directement le prédicat `In` a une valeur du type `bst_tree`.

6.3 Modules et foncteurs

L'adéquation de Coq comme formalisme de spécification et de preuve de programmes ML purement fonctionnels s'étend jusqu'au système de modules. En effet, Coq est depuis peu équipé d'un système de modules inspiré de celui d'Objective Caml [Ler00, Chr03a, Chr03b]. De même que les types de fonction Coq peuvent enrichir ceux de ML par des annotations logiques, les modules de Coq peuvent enrichir ceux de ML.

Ainsi, si l'on souhaite écrire notre bibliothèque d'ensembles finis sous la forme d'un foncteur prenant en argument un type quelconque (et non plus `Z` comme jusqu'à présent) équipé d'un ordre total, on commence par définir une signature pour cet argument. On y met un type `t`, une égalité `eq` et une relation d'ordre `lt` sur ce type :

```
Coq < Module Type OrderedType.

Coq <   Parameter t : Set.

Coq <   Parameter eq : t -> t -> Prop.

Coq <   Parameter lt : t -> t -> Prop.
```

ainsi qu'un résultat de décidabilité de `lt` et `eq` :

```
Coq <   Parameter compare : forall x y, {lt x y}+{eq x y}+{lt y x}.
```

Il faut également fournir quelques propriétés sur `eq` (relation d'équivalence) et `lt` (relation d'ordre incompatible avec `eq`) sans lesquelles les fonctions sur les arbres binaires de recherche ne peuvent être correctes :

```
Coq <   Axiom eq_refl : forall x, (eq x x).
Coq <   Axiom eq_sym : forall x y, (eq x y) -> (eq y x).
Coq <   Axiom eq_trans : forall x y z, (eq x y) -> (eq y z) -> (eq x z).
Coq <
Coq <   Axiom lt_trans : forall x y z, (lt x y) -> (lt y z) -> (lt x z).
Coq <   Axiom lt_not_eq : forall x y, (lt x y) -> ~(eq x y).
```

Enfin, on peut ajouter à la signature des commandes `Hint` pour la tactique `auto` —- et elles seront ainsi disponibles automatiquement dans le corps du foncteur :

```
Coq <   Hint Immediate eq_sym.
Coq <   Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
Coq < End OrderedType.
```

On peut alors écrire notre bibliothèque d'ensembles sous la forme d'un foncteur prenant un argument `X` de type `OrderedType` :

```
Coq < Module ABR (X: OrderedType).
Coq <
Coq <   Inductive tree : Set :=
Coq <   | Empty
Coq <   | Node : tree -> X.t -> tree -> tree.
Coq <
Coq <   Fixpoint mem (x:X.t) (s:tree) {struct s} : bool := ...
Coq <
Coq <   Inductive In (x:X.t) : tree -> Prop := ...
Coq <   Hint Constructors In.
Coq <
Coq <   Inductive bst : tree -> Prop :=
Coq <   | bst_empty : (bst Empty)
Coq <   | bst_node :
Coq <      forall x (l r : tree),
Coq <      bst l -> bst r ->
Coq <      (forall y, In y l -> X.lt y x) ->
Coq <      (forall y, In y r -> X.lt x y) -> bst (Node l x r).
Coq <
Coq <   (* etc. *)
```

Note. Le langage Objective Caml fournit une bibliothèque d'ensembles finis codés par des arbres binaires de recherche équilibrés (des AVL [AVL62]), sous la forme d'un foncteur prenant en argument un type ordonnée. Cette bibliothèque implante toutes les opérations habituelles sur les ensembles (union, intersection, différence, cardinal, plus petit élément, etc.), des itérateurs (`map`, `fold`, `iter`) et également une fonction d'ordre total sur les ensembles permettant d'obtenir des ensembles d'ensembles par une seconde application du même foncteur (et ainsi de suite). Cette bibliothèque a été certifiée à l'aide de Coq par Pierre Letouzey et Jean-Christophe Filliâtre [FL04]. Cette preuve a permis de découvrir un bug dans le ré-équilibrage des arbres effectué par certaines fonctions ; le code a été corrigé dans la dernière version d'ocaml (3.07).

Annexe A

Extraction de programmes et réalisabilité

Introduction

Dans ce cours nous étudions le caractère constructif de la logique sous-jacente au calcul des constructions inductives. Nous montrons comment construire à partir d'une preuve, un programme qui "réalise" la propriété montrée. Nous expliquerons la distinction entre les sortes `Prop` et `Set` dans le système Coq.

A.1 Interprétation constructive des preuves

A.1.1 Logique classique versus logique intuitionniste

La logique de Coq est intuitionniste, aucun axiome ne permet de dériver $A \vee \neg A$ ou bien $\neg\neg A \Rightarrow A$ pour une formule A arbitraire.

Est-ce embêtant ?

On se place par exemple dans l'arithmétique du premier ordre ou d'ordre supérieur. On notera $\vdash_C A$ lorsque A est prouvable de manière classique et $\vdash_I A$ lorsque A est prouvable de manière intuitionniste.

Il existe de nombreux résultats sur les liens entre les preuves classiques et intuitionnistes :

- Une propriété vraie de manière intuitionniste l'est aussi de manière classique.

$$\text{si } \Gamma \vdash_I A \text{ alors } \Gamma \vdash_C A$$

- Une propriété vraie de manière classique peut être prouvée de manière intuitionniste en ajoutant le schéma d'axiome du tiers exclu.

$$\text{si } \Gamma \vdash_C A \text{ alors } (C_i \vee \neg C_i)_i, \Gamma \vdash_I A$$

- En logique classique:

$$\vdash_C A \vee B \Leftrightarrow \neg(\neg A \wedge \neg B) \qquad \vdash_C \exists n : nat.P(n) \Leftrightarrow \neg\forall n.\neg P(n)$$

En logique intuitionniste l'équivalence ne peut être prouvée que dans un seul sens:

$$\vdash_I A \vee B \Rightarrow \neg(\neg A \wedge \neg B) \qquad \vdash_I \exists n : nat.P(n) \Rightarrow \neg\forall n.\neg P(n)$$

- Toute formule A peut être transformée en une formule $A*$ classiquement équivalente (par exemple en éliminant les $\exists$ et $\vee$ et en remplaçant les formules atomiques par leur double négation) telle que
$$\vdash_C A \Leftrightarrow A^* \quad \text{et si } \vdash_C A \text{ alors } \vdash_I A^*$$
- Les formules $\forall n.\exists m.Q(n,m)$ avec Q sans quantificateur (formules dites Π^0_2) sont démontrables de manière intuitionniste si et seulement elles sont démontrables de manière classique.
$$\vdash_C \forall n.\exists m.Q(n,m) \text{ si et seulement si } \vdash_I \forall n.\exists m.Q(n,m)$$

Les propriétés spécifiques des preuves intuitionnistes

- Les preuves intuitionnistes vérifient les *propriétés de la disjonction et du témoin*:
$$\text{si } \vdash_I A \vee B \text{ alors } \vdash_I A \text{ ou } \vdash_I B$$
$$\text{si } \vdash_I \exists x.P(x) \text{ alors il existe } t \text{ tel que} \vdash_I P(t)$$
- Les preuves intuitionnistes vérifient l'*axiome du choix*:
$$\text{si } \vdash_I \forall x.P(x) \vee \neg P(x) \text{ alors il existe } f \text{ fonction récursive tel que } \vdash_I f(x) = \mathtt{true} \Leftrightarrow P(x)$$
$$\text{si } \vdash_I \forall x.\exists y.P(x,y) \text{ alors il existe } f \text{ fonction récursive telle que } \vdash_I P(x, f(x))$$
- Les preuves classiques peuvent donner des résultats non conformes à l'intuition de la "vérité".

Les propriétés spécifiques des preuves classiques

- Les preuves classiques ont un contenu calculatoire: le schéma $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ (loi de Peirce) a pour contenu calculatoire l'opérateur de contrôle `call-cc` (l'opérateur *call with current continuation* que l'on peut trouver dans Scheme et SML).
- Toute preuve classique de $\vdash \exists n.P(n)$ avec $P(n)$ atomique s'évalue en une preuve intuitionniste de $\vdash \exists n.P(n)$ qui dévoile un terme t tel que $\vdash P(t)$.
- La réalisation de l'axiome du choix
$$(\forall x : A.\exists y : B.P(x,y)) \Rightarrow \exists f.\forall x.P(x, f(x))$$
pose problème en présence de la logique classique (considérer par exemple la preuve classique de $\forall x.\exists b.b = \mathtt{true} \Leftrightarrow P(x)$). Réaliser par une fonction calculable faisant intervenir `call-cc` entraîne que le domaine de quantification B soit dégénéré. Reste alors la réalisation par une fonction non calculable (c.-à-d. par un oracle décidant la vérité a priori de toute proposition) ce qui pousse à refuser un contenu calculatoire aux éléments du domaine de quantification et donc aux $\forall$ et $\exists$. On peut alors réaliser l'axiome du choix par l'identité de $P(x,y)$ dans $P(x, f(x))$ pour un certain f non calculatoire (c'est une voie très différente de celle suivie dans la suite de ce chapitre).

Preuves intuitionnistes et récursivité Un avantage de la logique intuitionniste est qu'elle permet de parler de la décidabilité de propriétés de manière implicite sans faire appel à une théorie de la récursivité. Pour montrer qu'un prédicat est décidable ou qu'une relation fonctionnelle est récursive, il suffit d'exhiber une preuve d'une formule disjonctive ou existentielle. Cependant on ne capture pas ainsi toutes les fonctions récursives en effet il existe des relations fonctionnelles correspondant à des fonctions récursives et pour lesquelles la formule disant que la relation est fonctionnelle n'est pas prouvable. Par exemple, il est possible de coder les termes du calcul comme des entiers et de définir la relation de réduction sur les termes. La preuve de totalité de la fonction de normalisation permet de montrer la cohérence logique du système et ne peut donc, du fait des théorèmes d'incomplétude de Gödel, être montrée dans le système lui-même.

A.1.2 Constructivité du Calcul des Constructions Inductives

Pour montrer le caractère constructif de la logique de `Coq`, on s'appuie sur l'isomorphisme de Curry-Howard qui permet de représenter les preuves par des λ-termes fortement normalisables et la propriété syntaxique suivante qui caractérise les objets normaux clos dans les types inductifs:

Propriété Un terme normal clos dont le type est une instance d'une définition inductive est forcément de la forme $(c\ t_1 \dots\ t_n)$ avec c un des constructeurs du type inductif et t_i des termes clos normaux.

Preuve En effet tout terme t s'écrit de manière unique $(c\ t_1 \dots\ t_n)$ avec c qui n'est pas une application. c peut donc être soit une abstraction, soit une variable, soit un constructeur, soit une sorte, soit un produit, soit un `Case`, soit un point fixe. On procède par récurrence sur la structure du terme et on examine chaque cas:

- c ne peut pas être une abstraction car t est normal et dans le cas $n = 0$ le type de t serait un produit,
- c ne peut pas être une variable car t est clos
- c ne peut pas être une sorte ou un produit car on aurait $n = 0$ et le type de t serait une sorte,
- c ne peut pas être un `Case` car l'argument principal du `Case` est un terme clos dans un type inductif et donc par hypothèse de récurrence commencerait par un constructeur et t ne serait pas normal
- de même c ne peut pas être un point fixe, en effet un point fixe a pour type $(x_1 : A_1) \dots (x_p : A_p)B$ avec x_p l'argument de décroissance dont le type est inductif, c serait au moins appliqué à p arguments (sinon le type est un produit) donc $p \leq n$, et t_p clos et normal commencerait par un constructeur et t ne serait pas normal.
- donc c est un constructeur dont le type est $(x_1 : A_1) \dots (x_p : A_p)I$ avec I une instance d'un type inductif. On a $p = n$, car c ne peut être appliqué au plus qu'à p arguments, et doit au moins être appliqué à p arguments pour que son type soit un type inductif.

Justification de la constructivité

- Si $A \vee B$ est prouvable sans hypothèse alors il existe une preuve de $A \vee B$ donc un terme clos de type $A \vee B$ qui est un type inductif à deux constructeurs:
  ```
  Inductive or [A,B:Set] : Set :=
     or_introl : A -> (or A B)
   | or_intror : B -> (or A B).
  ```
  En normalisant cette preuve, on obtient soit un terme (`or_introl` a) avec a clos de type A et donc A est prouvable, soit (`or_intror` b) avec b clos de type B et donc B est prouvable.
- De même la définition inductive de $\exists x : A.P(x)$ est
  ```
  Inductive ex [A:Set;P:A->Set] : Set :=
     ex_intro : (x:A)(P x) -> (ex A P) .
  ```
  Une preuve normale close de (`ex` A P) est de la forme (`ex_intro` t p) avec t terme clos de type A et p une preuve de $(P\ t)$.
- Si on a une preuve close de $\forall x : A.\exists y : B.P(x,y)$ alors il existe un terme F du type correspondant. Pour tout terme clos a de type A, on peut appliquer F à a ce qui nous donne un terme clos de type $\exists y : B.P(x,y)$ que l'on peut normaliser ce qui nous donne un terme de la forme (`ex_intro` t p) avec t terme clos de type A et p une preuve de $(P\ a\ t)$. Il y a donc une fonction récursive f qui transforme a en t et telle que $P(a, f(a))$ est montrable pour tout a. Cependant cette méthode ne nous dit pas si f peut être représentée dans le langage du système, ni si la formule $\forall x : A.P(x, f(x))$ est démontrable.

- Le même raisonnement s'applique à montrer que s'il existe une preuve de $\forall x : A.P(x) \vee \neg P(x)$ alors il existe un prédicat récursif p tel que $p(a) = \mathtt{true}$ si et seulement si $P(a)$ est démontrable et $p(a) = \mathtt{false}$ si et seulement si $\neg P(a)$ est démontrable.

Exemple 1 La propriété suivante qui dit que toute fonction sur les entiers admet un minimum est vraie de manière classique mais pas de manière intuitionniste même si on se limite aux fonctions récursives :

$$\forall f.\exists n.\forall m.f(m) \geq f(n)$$

En effet il n'existe pas de fonction récursive qui pour une fonction quelconque calcule son minimum. Sinon on pourrait décider pour toute fonction si elle prend la valeur nulle et donc on pourrait décider du problème de l'arrêt.

Exemple 2 On peut montrer de manière classique l'existence de deux nombre x et y irrationnels tels que x^y soit rationnel. On suppose établi que $\sqrt{2}$ est irrationnel.

- si ${\sqrt{2}}^{\sqrt{2}}$ est rationnel alors on prend $x = y = \sqrt{2}$
- si ${\sqrt{2}}^{\sqrt{2}}$ est irrationnel alors on prend $x = {\sqrt{2}}^{\sqrt{2}}$, $y = \sqrt{2}$ on a $x^y = {\sqrt{2}}^{\sqrt{2}\times\sqrt{2}} = {\sqrt{2}}^2 = 2$ est rationnel.

Cette démonstration ne permet pas d'exhiber une solution.

Exemple 3 Les résultats précédents permettent d'établir que $A \vee \neg A$ n'est pas démontrable en général. En effet supposons que ce soit le cas. On utilise le prédicat T de Kleene, $T(n, m, p)$ signifie que la fonction récursive de code n s'exécute sur l'entrée de code m pour effectuer un calcul de code p. C'est un résultat bien connu que le prédicat $P(n) = \exists p.T(n, n, p)$ n'est pas récursif car sinon $\neg P(n)$ le serait aussi et donc il existerait une fonction récursive de code q qui converge exactement lorsque $\neg P(n)$ est vérifiée c'est-à-dire pour tout n, $\exists p.T(q, n, p) \Leftrightarrow \neg\exists p.T(n, n, p)$ Il suffit de prendre $n = q$ pour aboutir à une contradiction.

Maintenant, en prenant pour A la formule $P(x) = \exists p.T(x, x, p)$, si $P(x) \vee \neg P(x)$ est montrable il en est de même de $\forall x.P(x) \vee \neg P(x)$ et on aurait la décidabilité de $P(x)$ qui est contradictoire.

A.1.3 Les limites de l'isomorphisme de Curry-Howard

L'isomorphisme de Curry-Howard permet de montrer la constructivité de la logique intuitionniste. Cependant, il ne permet pas de justifier certains principes comme celui de l'indépendance des prémisses.

Information logique

Soit une preuve du théorème de divison euclidienne:

$$\forall a, b.b > 0 \Rightarrow \exists q.\exists r.a = b \times q + r \wedge b < r$$

Pour calculer effectivement le quotient et le reste, il faut fournir les entrées a et b mais aussi une justification de $b > 0$ et le programme calculera le quotient et le reste mais aussi une preuve de correction. On voit que d'une part on doit fournir une information qui n'est pas utile pour le calcul (mais pour la terminaison et la correction du programme) d'autre part on calcule effectivement la preuve de correction du résultat ce qui est a priori inutile. La méthode proposée est donc inefficace.

Remarque L'isomorphisme de Curry-Howard n'est pas satisfaisant lorsqu'il s'agit de mettre en évidence les fonctions récursives sous-jacentes aux preuves. En effet, il ne permet pas de traiter le cas des preuves sous axiome, car alors les preuves ne sont plus closes et le calcul peut dépendre de manière essentielle de l'hypothèse.

Le problème est donc de savoir si étant donné une preuve de $\forall x.P(x) \Rightarrow \exists y.Q(x, y)$ il est possible de construire un programme f tel que $\forall x.P(x) \Rightarrow Q(x, f(x))$.

Ce n'est en général pas vrai pour toutes les propriétés $P(x)$ en effet la preuve de $P(x)$ peut transporter une information servant au calcul du témoin y.

Par exemple si on prouve $\forall n, m.n \leq m \Rightarrow \exists p.n + p = m$ par induction sur la preuve de $n \leq m$, alors le calcul de la différence entre n et m dépendra de cette preuve. Elle ne pourra plus être considérée comme non calculatoire et devra être donnée en argument au programme.

On s'intéressera à caractériser certaines formules $P(x)$ dont les preuves ne contiennent pas d'information calculatoire intéressante. C'est le cas par exemple des *formules de Harrop* qui sont des formules n'ayant pas de disjonction ou de quantificateur existentiel en partie strictement positive (par exemple toutes les formules $\neg P$).

Nous allons nous intéresser maintenant à des méthodes pour obtenir à partir d'une preuve intuitionniste de $\forall x.P(x) \Rightarrow \exists y.Q(x, y)$ effectivement un programme f et une preuve de correction $\forall x.P(x) \Rightarrow Q(x, f(x))$.

A.2 Réalisabilité

A.2.1 Principes généraux

La réalisabilité a été introduite par Kleene en 1952. C'est une interprétation sémantique des propositions en logique intuitionniste.

On se donne un ensemble de *réalisations* qui représentent des programmes.

Chaque proposition P est interprétée comme un ensemble de réalisations qui est défini en général par récurrence sur la structure de la formule P. Cet ensemble est défini intensionnellement par une propriété de réalisabilité "$x\ \mathbf{r}\ P$" dans lequel x est une nouvelle variable libre représentant une réalisation.

Une formule P dont l'interprétation est non vide sera dite *réalisable.*

L'idée de base des définitions est la suivante:

- l'absurde est interprété par l'ensemble vide,
- $A \Rightarrow B$ est interprété comme l'ensemble des réalisations représentant des fonctions des réalisations de A dans les réalisations de B,
- $A \wedge B$ est interprété comme l'ensemble des réalisations représentant des couples formés d'une réalisation de A et d'une réalisation de B,
- $\exists x.P(x)$ sera interprété comme l'ensemble des réalisations représentant des couples formés d'un objet t et d'une réalisation de $P(t)$ ...

Une fois l'interprétation définie, on montre que si une formule est prouvable alors son interprétation est non vide et que de plus il est possible de construire par récurrence sur la structure de la preuve, une réalisation particulière.

L'intérêt de cette méthode est qu'elle s'étend aux preuves sous contextes, c'est-à-dire que si une formule est prouvable sous hypothèses et que chaque hypothèse a une interprétation non vide alors il en est de même de la conclusion. Une conséquence de cette propriété est que toute formule dont l'interprétation est non vide est cohérente avec la théorie. En effet si on pouvait dériver l'absurde à partir de cette proposition alors l'interprétation de l'absurde serait non vide.

La réalisabilité peut servir également à montrer que certaines formules ne sont pas démontrables comme conséquence du fait qu'elles ne sont pas réalisables.

A.2.2 Différentes notions de réalisabilité

Il y a de très nombreuses notions de réalisabilité adaptées aux propriétés que l'on cherche à montrer. Elles se distinguent par la nature du langage de réalisation, on peut en effet prendre des entiers représentant des codes de fonction, ou bien un lambda-calcul pur ou typé ou tout autre langage. Ensuite on peut mettre différents ingrédients dans les formules de réalisabilité. Par exemple f appartient à l'interprétation de $A \Rightarrow B$ peut être défini comme pour tout a dans l'interprétation de a, $f(a)$ termine et est dans l'interprétation de B, ou bien on peut de plus demander que B soit démontrable …Les preuves de normalisation par les méthodes de candidat de réductibilité peuvent se voir comme des cas particuliers de méthode de réalisabilité.

On peut définir la réalisabilité de manière sémantique, ou au contraire de manière syntaxique (on parle de réalisabilité abstraite), la propriété $x\ \mathbf{r}\ P$ qui dit qu'une réalisation x est dans l'interprétation d'une formule P est définie comme une formule du système logique lui-même, définie en général dans un fragment plus faible (on élimine les connecteurs intuitionnistes $\exists$ et $\vee$). Plus d'information sur la réalisabilité peut se trouver dans les livres de Troelstra [Tro73], Troelstra et van Dalen [TvD88] et Beeson [Bee85].

Nous décrivons trois notions de réalisabilité abstraite. Pour cela nous nous plaçons dans une arithmétique fonctionnelle du premier ordre qui est l'arithmétique que l'on étend de manière à pouvoir parler d'objets fonctionnels représentés par des λ-termes, on considère également la possibilité de former la paire de deux objets. Les objets de base sont les entiers. Un prédicat particulier $x \in \mathtt{nat}$ permet de distinguer les objets entiers. De manière usuelle, on écrira $\forall x \in \mathtt{nat}.P$ pour $\forall x.x \in \mathtt{nat} \Rightarrow P$ et $\exists x \in \mathtt{nat}.P$ pour $\exists x.x \in \mathtt{nat} \wedge P$ et

La disjonction $A \vee B$ est définie comme $\exists b \in \mathtt{nat}.(b = 0 \Rightarrow A) \wedge (b \neq 0 \Rightarrow B)$.

Réalisabilité récursive La réalisabilité récursive a été introduite par Kleene [Kle52]. L'ensemble des réalisations est l'ensemble des fonctions récursives partielles. Une réalisation est forcément un objet qui a une valeur, soit un entier (qui peut représenter le code d'une fonction récursive) dans le cas de l'arithmétique du premier ordre, soit un entier ou une abstraction dans le cas de l'arithmétique fonctionnelle. On distingue un prédicat $t\Downarrow$ qui est vrai lorsque le terme t a une valeur. Les quantifications portent implicitement sur les termes qui ont une valeur.

La définition de la réalisabilité récursive est décrite dans la figure A.1.

$$\begin{array}{lll}
x\ \mathbf{r}\ A & = x = 0 \wedge A & A \text{ atomique} \\
x\ \mathbf{r}\ A \wedge B & = \mathtt{fst}(x)\ \mathbf{r}\ A\ \wedge\ \mathtt{snd}(x)\ \mathbf{r}\ B & \\
f\ \mathbf{r}\ A \Rightarrow B & = \forall x.\ x\ \mathbf{r}\ A\ \Rightarrow\ f(x)\Downarrow\ \wedge f(x)\ \mathbf{r}\ B & \\
x\ \mathbf{r}\ \exists y.B & = \mathtt{snd}(x)\ \mathbf{r}\ B\{y := \mathtt{fst}(x)\} & \\
f\ \mathbf{r}\ \forall x.B & = \forall x.\ f(x)\Downarrow\ \wedge\ f(x)\ \mathbf{r}\ B &
\end{array}$$

Fig. A.1 – Réalisabilité récursive

On montre que si une formule A est démontrable, alors on peut trouver un terme t tel que $t\Downarrow \wedge t\ \mathbf{r}\ A$ soit démontrable.

Principe de Markov La réalisabilité récursive sert à justifier par exemple le principe de Markov :

$$(\forall x \in \mathtt{nat}.P(x) \vee \neg P(x)) \Rightarrow \neg\neg\exists x \in \mathtt{nat}.P(x) \Rightarrow \exists x \in \mathtt{nat}.P(x)$$

Indépendance des prémisses La réalisabilité récursive sert à justifier que le principe d'indépendance des prémisses n'est pas démontrable en logique intuitionniste :

$$(\neg A \Rightarrow \exists x.P(x)) \Rightarrow \exists x.\neg A \Rightarrow P(x)$$

Évaluation Tous les objets intermédiaires manipulés dans le programme ont une valeur ce qui permet d'assurer qu'un programme t qui réalise une formule pourra se calculer par une stratégie d'appel par valeur mais sans évaluer les termes sous les abstractions.

Réalisabilité modifiée La réalisabilité modifiée (typée) a été introduite par Kreisel. Il s'agit d'assurer la terminaison des interprétations par une condition de bon typage. Les objets manipulés par la logique sont des termes typés dans un λ-calcul avec des types simples, un produit et des entiers. On notera explicitement le type des variables apparaissant dans les quantificateurs $\exists y : \sigma.P$ ou $\forall y : \sigma.P$.

À chaque formule A est associé le type $\mathtt{t}(A)$ de ses réalisations. Dans la formule $x\ \mathbf{r}\ A$, x représente une variable de type $\mathtt{t}(A)$.

$$\begin{array}{ll|lll}
\mathtt{t}(A) & = \mathtt{nat} & x\ \mathbf{r}\ A & = x = 0 \wedge A & A \text{ atomique} \\
\mathtt{t}(A \wedge B) & = \mathtt{t}(A) \times \mathtt{t}(B) & x\ \mathbf{r}\ A \wedge B & = \mathtt{fst}(x)\ \mathbf{r}\ A\ \wedge\ \mathtt{snd}(x)\ \mathbf{r}\ B & \\
\mathtt{t}(A \Rightarrow B) & = \mathtt{t}(A) \rightarrow \mathtt{t}(B) & f\ \mathbf{r}\ A \Rightarrow B & = \forall x{:}\mathtt{t}(A).\ x\ \mathbf{r}\ A\ \Rightarrow\ f(x)\ \mathbf{r}\ B & \\
\mathtt{t}(\exists y{:}\sigma.B) & = \sigma \times \mathtt{t}(B) & x\ \mathbf{r}\ \exists y{:}\sigma.B & = \mathtt{snd}(x)\ \mathbf{r}\ B\{y := \mathtt{fst}(x)\} & \\
\mathtt{t}(\forall y{:}\sigma.B) & = \sigma \rightarrow \mathtt{t}(B) & f\ \mathbf{r}\ \forall x{:}\sigma.B & = \forall x{:}\sigma.\ f(x)\ \mathbf{r}\ B &
\end{array}$$

Fig. A.2 – Réalisabilité modifiée

Une formule A est réalisable s'il existe un terme t de type $\mathtt{t}(A)$ tel que $t\ \mathbf{r}\ A$ soit prouvable.

Une manière alternative de présenter la réalisabilité modifiée est de prendre des suites finies de programmes pour les réalisations, on évite ainsi l'utilisation de produits, on peut de plus éliminer les réalisations des formules de Harrop.

Indépendance des prémisses La réalisabilité modifiée sert à justifier le principe d'indépendance des prémisses :

$$(\neg A \Rightarrow \exists x : \sigma.P(x)) \Rightarrow \exists x : \sigma.\neg A \Rightarrow P(x)$$

Principe de Markov La réalisabilité récursive sert à justifier que le principe de Markov n'est pas démontrable en logique intuitionniste :

$$(\forall x : \mathtt{nat}.P(x) \vee \neg P(x)) \Rightarrow \neg\neg\exists x : \mathtt{nat}.P(x) \Rightarrow \exists x : \mathtt{nat}.P(x)$$

Évaluation Dans le cas de la réalisabilité récursive, les réalisations sont des programmes typés fortement normalisables.

Réalisabilité modifiée non typée La réalisabilité modifiée non typée se place dans une logique où la quantification porte sur tous les objets qu'ils représentent ou non des programmes qui terminent. La définition est identique à celle de la réalisabilité modifiée décrite dans la figure A.2. Simplement la condition pour qu'un programme t réalise une formule P est simplement que $t\ \mathbf{r}\ P$ et ne comporte plus de condition de bon typage de t.

Pour assurer la terminaison, il sera nécessaire que la formule à montrer explicite cette condition de terminaison. Par exemple, l'interprétation du prédicat de base $x \in \mathtt{nat}$ pourra être telle que l'existence d'une preuve de $t \in \mathtt{nat}$ assure que t est normalisable.

Une telle notion de réalisabilité est utilisée dans le système AF_2 de J.-L. Krivine. Les formules utilisées pour garantir la terminaison des programmes ne permettent de garantir le calcul des valeurs que dans une stratégie paresseuse.

Réalisabilité et ordre supérieur Lorsque la logique manipule des variables du second ordre, on ne sait pas a priori par quelle formule cette variable sera instanciée. On réalise donc cette variable par un prédicat unaire arbitraire.

$$\begin{array}{ll} x\ \mathbf{r}\ \forall X.A & = \forall P_X.\, x\ \mathbf{r}\ A \\ x\ \mathbf{r}\ X & = P_X(x) \end{array}$$

Formules auto-réalisées Les formules auto-réalisées sont des formules pour lesquelles on connait a priori une réalisation. Plus formellement, une formule P est auto-réalisée s'il existe un objet t tel que si P est réalisable alors $t\ \mathbf{r}\ P$ est vérifié.

Les formules de Harrop sont des formules auto-réalisées.

A.3 Réalisabilité dans le Calcul des Constructions

Par rapport à l'interprétation des preuves comme programmes par l'isomorphisme de Curry-Howard, l'introduction d'une notion de réalisabilité dans le Calcul des Constructions a pour but l'obtention de programmes plus efficaces ne conservant que la partie de la preuve utile pour le calcul des témoins. Elle sert également à justifier certaines propriétés qui ne sont pas démontrables.

A.3.1 Oubli des types dépendants

On considère le Calcul des Constructions pur (sans univers et sans élimination forte sur les types inductifs) que nous noterons CC. On peut montrer que tout λ-terme pur typable dans CC est également typable dans le système F_ω (avec types inductifs).

Cette propriété est simple à montrer. On associe à chaque terme t ou type de CC un terme ou un type $\mathbf{E}(t)$ dans F_ω en oubliant les dépendances des types par rapport aux preuves et on montre que si $\vdash_{CC} t : P$ alors $\vdash_{\mathsf{F}_\omega} \mathbf{E}(t) : \mathbf{E}(P)$. La définition de la fonction d'oubli $\mathbf{E}(\_)$ est donnée dans la figure A.3.

$$\begin{array}{lll}
\mathbf{E}(\mathtt{Set}) & = \mathtt{Set} & \\
\mathbf{E}(X) & = \mathrm{X} & X : A : \mathtt{Type} \\
\mathbf{E}(x) & = \mathrm{x} & x : A : \mathtt{Set} \\
\mathbf{E}((x : A)B) & = \mathbf{E}(A) \to \mathbf{E}(B) & A, B : \mathtt{Set} \\
\mathbf{E}((X : A)B) & = (X : \mathbf{E}(A))\mathbf{E}(B) & A : \mathtt{Type}, B : \mathtt{Set} \\
\mathbf{E}((x : A)B) & = \mathbf{E}(B) & A : \mathtt{Set}, B : \mathtt{Type} \\
\mathbf{E}((X : A)B) & = \mathbf{E}(A) \to \mathbf{E}(B) & A, B : \mathtt{Type} \\
\mathbf{E}([x : A]t) & = [x : \mathbf{E}(A)]\mathbf{E}(t) & t : B : \mathtt{Set} \text{ ou } A : \mathtt{Type} \\
\mathbf{E}([x : A]t) & = \mathbf{E}(t) & t : B : \mathtt{Type} \text{ et } A : \mathtt{Set} \\
\mathbf{E}((t\ u)) & = (\mathbf{E}(t)\ \mathbf{E}(u)) & t : A : \mathtt{Set} \text{ ou } u : B : \mathtt{Type} \\
\mathbf{E}((t\ u)) & = \mathbf{E}(t) & t : A : \mathtt{Type} \text{ et } u : B : \mathtt{Set}
\end{array}$$

FIG. A.3 – Traduction de CC ves F_ω

Cette traduction permet de montrer que $0 \neq 1$ n'est pas prouvable dans CC. En effet $0 \neq 1$ est une abréviation pour

$$((P : \mathtt{nat} \to \mathtt{Set})(P\ 0) \to (P\ 1)) \to (C : \mathtt{Set})C$$

S'il existait un terme de ce type, il y aurait également un terme de F_ω de type $\mathbf{E}(0 \neq 1)$ c'est-à-dire :

$$((P : \mathtt{Set})(P \to P)) \to (C : \mathtt{Set})C$$

Mais comme il existe un terme de type $(P : \mathtt{Set})(P \to P)$ on aboutit à l'existence d'un terme de type $(C : \mathtt{Set})C$ ce qui est absurde.

Réalisabilité modifiée Une notion de réalisabilité modifiée naturelle est de demander de réaliser toute formule de P du Calcul des Constructions par un terme t de type $\mathbf{E}(P)$ qui satisfait de plus une certaine propriété $x\ \mathbf{r}\ P$ définie par récurrence sur P dans la figure A.4. On commence par définir la formule $x\ \mathbf{r}\ P$ pour $P : \mathtt{Set}$ on aura également besoin de définir $X\ \mathbf{r}\ A$ lorsque $A : \mathtt{Type}$ ce qui est fait dans la figure A.5. Dans le cas $A : \mathtt{Type}$, $X\ \mathbf{r}\ A$ sera également une arité de type `Type` qui représente le type des prédicats de réalisabilité pour les objets d'arité A. Comme une proposition peut être formée par abstraction et application, nous définissons également dans la figure A.6 une transformation $\mathbf{R}(P)$ qui s'applique à tous les objets $P : A$ avec $A : \mathtt{Type}$ et qui coincide avec la définition du prédicat $\lambda x.x\ \mathbf{r}\ A$ quand A est `Set`(en particulier $\mathbf{R}(A)$ a alors le type $\mathbf{E}(A) \to \mathtt{Prop}$).

$$\begin{array}{lll}
x\ \mathbf{r}\ P & = (\mathbf{R}(P)\ x) & P : \mathtt{Set} \text{ et } P \text{ n'est pas un produit} \\
f\ \mathbf{r}\ (x : A)B & = (x : \mathbf{E}(A))x\ \mathbf{r}\ A \to (f\ x)\ \mathbf{r}\ B & A, B : \mathtt{Set} \\
f\ \mathbf{r}\ (X : A)B & = (X : \mathbf{E}(A))(X_r : X\ \mathbf{r}\ A)(f\ X)\ \mathbf{r}\ B & A : \mathtt{Type}, B : \mathtt{Set}
\end{array}$$

FIG. A.4 – Réalisabilité dans CC: $x\ \mathbf{r}\ P$ pour $P : \mathtt{Set}$

$$\begin{array}{lll}
X\ \mathbf{r}\ \mathtt{Set} & = X \to \mathtt{Prop} & \\
F\ \mathbf{r}\ (x : A)B & = (x : \mathbf{E}(A))F\ \mathbf{r}\ B & A : \mathtt{Set}, B : \mathtt{Type} \\
F\ \mathbf{r}\ (X : A)B & = (X : \mathbf{E}(A))(X_r : X\ \mathbf{r}\ A)(F\ X)\ \mathbf{r}\ B & A, B : \mathtt{Type}
\end{array}$$

FIG. A.5 – Réalisabilité dans CC: $x\ \mathbf{r}\ P$ pour $P : \mathtt{Type}$

$$\begin{array}{lll}
\mathbf{R}(X) & = X_r & X \text{ variable de prédicat} \\
\mathbf{R}((P\ t)) & = (\mathbf{R}(P)\ \mathbf{E}(t)) & t : A : \mathtt{Set} \\
\mathbf{R}((P\ T)) & = (\mathbf{R}(P)\ \mathbf{E}(T)\ \mathbf{R}(T)) & T : A : \mathtt{Type} \\
\mathbf{R}([x : A]P) & = [x : \mathbf{E}(A)]\mathbf{R}(P) & A : \mathtt{Set} \\
\mathbf{R}([X : A]P) & = [X : \mathbf{E}(A)][X_r : X\ \mathbf{r}\ A]\mathbf{R}(P) & A : \mathtt{Type} \\
\mathbf{R}(P) & = [x : \mathbf{E}(P)]x\ \mathbf{r}\ P & \text{si } P \text{ est un produit}
\end{array}$$

FIG. A.6 – Réalisabilité dans CC : $\mathbf{R}(P)$ pour $P : B : \mathtt{Type}$

Exercice A quelle condition un terme p réalise-t-il une preuve de l'égalité définie avec $A : \mathtt{Set}$ par :

$$(P : A \to \mathtt{Set})(P\ t) \to (P\ u)$$

A.3.2 Distinction entre `Prop` et `Set`

L'oubli des types dépendants n'est pas suffisant, il est important de pouvoir éliminer les parties de la preuve qui ne servent pas au calcul du résultat. La notion de formule de Harrop n'est pas naturelle dans un cadre d'ordre supérieur où il n'y a pas de formule atomique autre que les variables de prédicats.

Pour remédier à cela, `Coq` propose une distinction entre deux sortes `Prop` et `Set`. Les deux sortes sont imprédicatives, ce qui justifie les bonnes propriétés du système (il suffit d'identifier `Prop` et `Set` pour retrouver le calcul initial).

L'interprétation de $t : A$ lorsque $A : \mathtt{Prop}$ est que A est vérifié et que la preuve t de A ne sert pas au calcul. L'interprétation de $t : A$ lorsque $A : \mathtt{Set}$ est que t est une preuve calculatoire de A dont il est possible d'extraire une programme.

Les règles de typage permettent d'assurer qu'aucun objet *logique* de sorte `Prop` ne sera utilisé pour la construction d'un objet *calculatoire* dans la sorte `Set`. Ceci permet de retirer de manière cohérente tout sous terme logique d'un terme calculatoire en préservant le résultat du calcul.

D'un point de vue technique, il faut étendre les notions d'extraction et de réalisabilité précédemment définies. La fonction d'extraction ne s'applique toujours qu'à des objets de type `Set` ou `Type`. Elle est définie dans la figure A.7. La fonction de réalisabilité $f \ \mathbf{r} \ P$ s'applique à tous les objets de `Set` et `Type` et est définie figures A.8, A.9 et A.10.

On dira que $A : \mathtt{Type}_P$ lorsque A est une suite de produits se terminant par `Prop`. La définition de la réalisabilité passe par la définition d'une transformation $\mathbf{L}(P)$ sur les objets de `Prop` ou $\mathtt{Type}_P$ qui est définie figure A.11. Le rôle de cette transformation est d'ajuster les types des objets de `Set` ou `Type` apparaissant dans P en y appliquant la fonction d'extraction $\mathbf{E}(B)$. En particulier, si A est dans `Prop`, $\mathbf{L}(A)$ est aussi dans `Prop`.

$\mathbf{E}((x : A)B)$	$= \mathbf{E}(B)$	$A : \mathtt{Prop}$ ou $A : \mathtt{Type}_P$
$\mathbf{E}([x : A]t)$	$= \mathbf{E}(t)$	$A : \mathtt{Prop}$ ou $A : \mathtt{Type}_P$
$\mathbf{E}((t \ u))$	$= \mathbf{E}(t)$	$u : B : \mathtt{Prop}$ ou $u : B : \mathtt{Type}_P$

FIG. A.7 – Extraction étendue à `Prop`

$f \ \mathbf{r} \ (x : A)B = (x : \mathbf{L}(A))f \ \mathbf{r} \ B$	($A : \mathtt{Prop}$ ou $A : \mathtt{Type}_P$) et $B : \mathtt{Set}$
$f \ \mathbf{r} \ (x : A)B = (x : \mathbf{L}(A))f \ \mathbf{r} \ B$	$A : \mathtt{Prop}$ et $B : \mathtt{Type}$
$f \ \mathbf{r} \ (x : A)B = (x : \mathbf{L}(A))f \ \mathbf{r} \ B$	$A : \mathtt{Type}_P$ et $B : \mathtt{Type}$

FIG. A.8 – Réalisabilité étendue à `Prop`: $x \ \mathbf{r} \ P$ avec $P : \mathtt{Set}$

$F \ \mathbf{r} \ (x : A)B \ = F \ \mathbf{r} \ B$	$B : \mathtt{Type}$ et $A : \mathtt{Prop}$
$F \ \mathbf{r} \ (X : A)B = (X : \mathbf{L}(A))F \ \mathbf{r} \ P$	$B : \mathtt{Type}$ et $A : \mathtt{Type}_P$

FIG. A.9 – Réalisabilité étendue à `Prop`: $x \ \mathbf{r} \ P$ pour $P : \mathtt{Type}$

Exemple Pour comprendre les interactions entre `Prop` et `Set` dans la réalisabilité, on peut prendre l'exemple de la définition imprédicative d'une famille `list` de listes d'objets de type A vérifiant un prédicat P. La famille `list` a pour type $(A : \mathtt{Set})(A \to \mathtt{Prop}) \to \mathtt{Set}$ et est défini par :

$$\begin{aligned} \mathtt{list} \equiv \ & [A : \mathtt{Set}][P : A \to \mathtt{Prop}] \\ & (X : \mathtt{Set})X \to ((a : A)(P \ a) \to X \to X) \to X \end{aligned}$$

$\mathbf{R}((P \ t))$	$= \mathbf{R}(P)$	$P : B : \mathtt{Type}$ et $t : A : \mathtt{Prop}$
$\mathbf{R}((P \ T))$	$= (\mathbf{R}(P) \ \mathbf{L}(T))$	$P : B : \mathtt{Type}$ et $T : A : \mathtt{Type}_P$
$\mathbf{R}([x : A]P)$	$= \mathbf{R}(P)$	$P : B : \mathtt{Type}$ et $A : \mathtt{Prop}$
$\mathbf{R}([X : A]P)$	$= [X : \mathbf{L}(A)]\mathbf{R}(P)$	$P : B : \mathtt{Type}$ et $A : \mathtt{Type}_P$

FIG. A.10 – Réalisabilité étendue à `Prop`: $\mathbf{R}(P)$ pour $P : B : \mathtt{Type}$

$\mathbf{L}(\mathtt{Prop})$	$= \mathtt{Prop}$	
$\mathbf{L}(X)$	$= X$	X variable de type ($B : \mathtt{Type}_P$ ou $B : \mathtt{Prop}$
$\mathbf{L}((x : A)B)$	$= (x : \mathbf{E}(A))\mathbf{L}(B)$	($B : \mathtt{Type}_P$ ou $B : \mathtt{Prop}$) et $A : \mathtt{Set}$
$\mathbf{L}((X : A)B)$	$= (X : \mathbf{E}(A))(X_r : X \ \mathbf{r} \ A)\mathbf{L}(B)$	($B : \mathtt{Type}_P$ ou $B : \mathtt{Prop}$) et $A : \mathtt{Type}$
$\mathbf{L}((x : A)B)$	$= (x : \mathbf{L}(A))\mathbf{L}(B)$	($B : \mathtt{Type}_P$ ou $B : \mathtt{Prop}$) et $A : \mathtt{Type}_P$
$\mathbf{L}((x : A)B)$	$= (x : \mathbf{L}(A))\mathbf{L}(B)$	($B : \mathtt{Type}_P$ ou $B : \mathtt{Prop}$) et $A : \mathtt{Prop}$
$\mathbf{L}((P \ t))$	$= (\mathbf{L}(P) \ \mathbf{E}(t))$	$P : B : \mathtt{Type}_P$ et $t : A : \mathtt{Set}$
$\mathbf{L}((P \ T))$	$= (\mathbf{L}(P) \ \mathbf{E}(T) \ \mathbf{R}(T))$	$P : B : \mathtt{Type}_P$ et $T : A : \mathtt{Type}$
$\mathbf{L}((P \ T))$	$= (\mathbf{L}(P) \ \mathbf{L}(T))$	($P : B : \mathtt{Type}_P$ ou $P : B : \mathtt{Prop}$) et $T : A : \mathtt{Type}_P$
$\mathbf{L}((P \ T))$	$= (\mathbf{L}(P) \ \mathbf{L}(T))$	($P : B : \mathtt{Type}_P$ ou $P : B : \mathtt{Prop}$) et $T : A : \mathtt{Prop}$
$\mathbf{L}([x : A]P)$	$= [x : \mathbf{E}(A)]\mathbf{L}(P)$	$P : B : \mathtt{Type}_P$ et $A : \mathtt{Set}$
$\mathbf{L}([X : A]P)$	$= [X : \mathbf{E}(A)][X_r : X \ \mathbf{r} \ A]\mathbf{L}(P)$	$P : B : \mathtt{Type}_P$ et $A : \mathtt{Type}$
$\mathbf{L}([X : A]P)$	$= [X : \mathbf{L}(A)]\mathbf{L}(P)$	($P : B : \mathtt{Type}_P$ ou $P : B : \mathtt{Prop}$) et $A : \mathtt{Type}_P$
$\mathbf{L}([X : A]P)$	$= [X : \mathbf{L}(A)]\mathbf{L}(P)$	($P : B : \mathtt{Type}_P$ ou $P : B : \mathtt{Prop}$) et $A : \mathtt{Prop}$

FIG. A.11 – Réalisabilité étendue à `Prop`: $\mathbf{L}(P)$

Le terme extrait de `list` est juste la définition usuelle des listes polymorphes. La propriété $\mathbf{R}(\mathtt{list})$ a pour type $(A : \mathtt{Set})(A_r : A \to \mathtt{Prop})(A \to \mathtt{Prop}) \to \mathtt{Prop}$ et est définie par :

$$\begin{aligned} \mathbf{R}(\mathtt{list}) \equiv \ & [A : \mathtt{Set}][A_r : A \to \mathtt{Prop}][P : A \to \mathtt{Prop}][l : \mathbf{E}(\mathtt{list})] \\ & (X : \mathtt{Set})(X_r : X \to \mathtt{Prop}) \\ & (x : X)(X_r \ x) \\ & \to (f : A \to X \to X)((a : A)(A_r \ a) \to (P \ a) \to (x : X)(X_r \ x) \to (X_r \ (f \ a \ x))) \\ & \to (X_r \ (l \ X \ x \ f)) \end{aligned}$$

On aurait pu choisir une autre notion d'extraction du contenu logique $\mathbf{L}(A)$ d'une proposition A. Par exemple, on aurait pu complétement ignorer les dépendances par rapport aux preuves logiques en modifiant la définition comme suit.

$\mathbf{L}((x : A)B)$	$= \mathbf{L}(B)$	$B : \mathtt{Type}_P$ et $A : \mathtt{Prop}$
$\mathbf{L}((P \ t))$	$= \mathbf{L}(P)$	$P : B : \mathtt{Type}_P$ et $t : A : \mathtt{Prop}$
$\mathbf{L}([x : A]P)$	$= \mathbf{L}(P)$	$P : B : \mathtt{Type}_P$ et $A : \mathtt{Prop}$

Dans un tel modèle, la propriété PI

$$(A : \mathtt{Prop})(p, q : A)(P : A \to \mathtt{Prop})(P \ p) \to (P \ q)$$

qui dit que deux preuves dans un même type $A : \mathtt{Prop}$ sont égales aurait été vraie, c'est-à-dire que $\mathbf{L}(PI)$ qui se réduit à :

$$(A : \mathtt{Prop})(p, q : A)(P : \mathtt{Prop})P \to P$$

est démontrable.

`Prop` est inclus dans `Set` Les rôles de `Prop` et `Set` sont disymétriques du fait de l'interprétation de réalisabilité. On peut plonger `Prop` dans un sous-ensemble de `Set` en faisant correspondre à $A : \texttt{Prop}$ un ensemble $\tilde{A}$ (il suffit de prendre $(C : \texttt{Set})(A \to C) \to C)$) qui a au plus un élement et qui est habité lorsque A est vérifié. On pourra montrer $A \leftrightarrow \tilde{A}$. La partie $\tilde{A} \to A$ peut se justifier dans CC par la réalisabilité. On peut également intégrer cet aspect directement dans le calcul par exemple par l'intermédiaire du sous-typage $\texttt{Prop} \leq \texttt{Set}$.

Cacher le contenu calculatoire des preuves Réciproquement, étant donné un type $A : \texttt{Set}$ on peut cacher le contenu calculatoire des preuves de A en construisant $\hat{A} : \texttt{Prop}$ tel que $A \to \hat{A}$ et $(C : \texttt{Prop})(A \to C) \to \hat{A} \to C$ (il suffit de prendre $(C : \texttt{Prop})(A \to C) \to C$). On n'a évidemment pas $\hat{A} \to A$ mais $\hat{A}$ et A sont équivalents dès lors qu'il s'agit de montrer des propriétés logiques de type `Prop`.

Types inductifs Dans `Coq` les règles d'élimination des types inductifs permettent de faire la distinction entre `Prop` et `Set`. Lorsque $c : I$ avec I instance d'une définition inductive alors :

- si $t : I : \texttt{Set}$, il est possible par la construction `Cases` t de construire à la fois des objets dans `Prop` et dans `Set`;
- si $I : \texttt{Prop}$, il est seulement possible de faire une construction `Cases` t pour construire des objets dans `Prop`;

Quelques cas spéciaux On traite de manière un peu particulière les types $I : A$ à un seul constructeur $c : C$ tel que si on avait $A : \texttt{Type}$ alors on aurait $\mathbf{E}(A) = \texttt{Set}$ et $\mathbf{E}(C) = \mathbf{E}(I)$. C'est le cas en particulier des conjonctions sur des objets dans `Prop` ou bien de l'égalité. Dans ces cas-là, on peut justifier l'équivalence pour la réalisabilité de la définition inductive de $I : A$ ou de $I : A'$ où A' est obtenu en remplaçant `Set` par `Prop` à la fin de l'arité A. En pratique on traite ce cas en autorisant la construction par cas d'objets de type `Set` même si $t : I : \texttt{Prop}$. On étend la notion d'extraction pour que l'extraction de

$$\texttt{Cases}\ t\ \texttt{of}\ (\texttt{c}\ x_1 \ldots x_p) \Rightarrow p\ \texttt{end}$$

soit simplement l'extraction de p ce qui est possible car p a le bon type (les x_i logiques, n'apparaissent pas dans P).

Limitations La notion d'extraction et de réalisabilité s'étend mal au cas des univers et de l'élimination forte sur les types inductifs. En effet la propriété d'oubli des dépendances ne s'applique plus. D'autre part les inclusions $\texttt{Prop} \leq \texttt{Type}$ et $\texttt{Set} \leq \texttt{Type}$ qui sont essentielles pour un développement harmonieux sont incompatibles avec une extraction incrémentale, car il existe alors des types $B : \texttt{Type}(n)$ (p.ex. $(U : Type(1))U \to U$) dont des instances sont dans `Type` et d'autres sont dans $\texttt{Type}_P$. Il faut connaître le développement complet avant de décider si l'on doit extraire ou cacher les objets de B.

Le mécanisme d'extraction de Coq (à partir de la version 7) s'éloigne d'ailleurs du schéma présenté ici. La distinction $\texttt{Type}_P$ et `Type` est abandonnée, et c'est une analyse ultérieure sur le programme qui permet de s'affranchir localement des occurrences de $b : B : \texttt{Type}$ qui s'avèrent purement logique.

Intérêt sémantique à la la distinction entre `Prop` et `Set` La distinction entre `Prop` et `Set` est utile sur le plan sémantique. En effet, certaines extensions (logique classique, extensionnalité,...) interagissent mal avec les aspects constructifs du Calcul des Constructions Inductives. Les prendre en compte simplement sur la sorte `Prop` permet de ne pas détruire la cohérence du système.

A.3.3 Autres méthodes d'analyse

Tous les assistants de preuves permettant de manipuler des programmes intègrent une certaine notion d'objets logiques. Dans le système Nuprl, on utilise un opérateur pour cacher le contenu calculatoire des preuves analogue à notre construction $\hat{A}$. D'autres systèmes privilégient la notion de sous-ensemble, ou bien donnent un statut logique à des propriétés particulières comme l'égalité. Toutes ses méthodes sont héritées de la notion logique de formule non-calculatoire.

Le défaut de cette méthode est qu'elle ne permet pas de supprimer du programme certains arguments inutiles. C'est le cas des listes de longueur n, le constructeur `cons` aura pour type $(n : \texttt{nat})A \to (\texttt{list}\ n) \to (\texttt{list}\ (\texttt{S}\ n))$ ce qui laisse un terme extrait de type $\texttt{nat} \to A \to \texttt{list} \to \texttt{list}$ dans lequel chaque constructeur est appliqué à un argument de type entier représentant la longueur de la liste. Même si on le souhaite, les méthodes reposant sur la réalisabilié ne permettent pas aisément de supprimer cette composante.

Pour régler ce problème, on peut déplacer les marques des arités vers les quantifications. On introduit une quantification logique $\forall x : A.B$ avec

$$\mathbf{E}(\forall x : A.B) = \mathbf{E}(B) \quad \texttt{et} \quad f\ \mathbf{r}\ \forall x : A.B = \forall x : A.f\ \mathbf{r}\ B$$

De tels systèmes ont été étudiés par Takayama [Tak91], Hayashi, ...

Une autre possibilité explorée plus récemment est de s'appuyer sur des méthodes d'analyse de code mort. Ces techniques ont été étudiées par Berardi [Ber96], Boerio [BB95], Damiani et Prost [DP98].

L'extraction a lieu a posteriori, le terme initial et le terme dans lequel on a retiré des parties de code mort sont prouvablement égaux pour une égalité extensionnelle (Berardi utilise une propriété générale qui dit que si deux termes du λ-calcul simplement typé ont le même type et que l'un est obtenu par élagage de certains sous-termes de l'autre alors les deux termes sont extensionnellement égaux).

Une des difficultés est qu'une même constante peut apparaître à plusieurs endroits avec des contenus calculatoires différents, d'où l'idée d'utiliser du sous-typage ou des types conjonctifs pour prendre en compte ce polymorphisme.

A.4 Développement de programmes certifiés dans Coq

A.4.1 Spécifications

La distinction entre `Prop` et `Set` amène à introduire plusieurs variantes pour les connecteurs constructifs d'existentielle et de disjonction.

Existentielles

On se donne $A : \texttt{Set}$.

$(\texttt{EX}\ x : A \mid (P\ x))$ Soit $P : A \to \texttt{Prop}$, la formule $(\texttt{EX}\ x : A \mid (P\ x))$ est de type `Prop` et donc sans contenu calculatoire, il n'est pas possible d'extraire le témoin, ni de démontrer l'axiome du choix.

$\{x : A \mid (P\ x)\}$ Toujours avec $P : A \to \texttt{Prop}$, la formule $\{x : A \mid (Px)\}$ est de type `Set`. Les preuves de $\{x : A \mid (Px)\}$ sont formées de couples (a, p) avec $a : A$ et $p : (P\ a)$. À l'extraction la preuve p disparaît. Le singleton (a) est optimisé et c'est simplement a qui est extrait.

Cette construction est utilisée pour exprimer la post-condition d'un programme. Soit $B : \texttt{Set}$ et $Q : A \to B \to \texttt{Prop}$, les réalisations de $(x : A)\{y : B \mid (Q\ x\ y)\}$ seront des fonctions f qui à une réalisation a de A associent une réalisation b de B tel que $(\mathbf{R}(Q)\ a\ b)$.

Cette construction permet de démontrer l'axiome du choix :

$$((x : A)\{y : B \mid (Q\ x\ y)\}) \to \{f : A \to B \mid (x : A)(Q\ x\ (f\ x))\}$$

On peut obtenir ainsi dans `Coq` un objet de type $A \to B$ mais qui est en général beaucoup plus gros que le terme extrait de la preuve de $(x : A)\{y : B \mid (Q\ x\ y)\}$.

$\{x : A\ \&\ (R\ x)\}$ Soit $R : A \to$ `Set`, la formule $\{x : A\ \&\ (R\ x)\}$ est de type `Set`. Les preuves de $\{x : A\ \&\ (Rx)\}$ sont extraites en des couples (a, b) avec a une réalisation de A et b une réalisation de $(R\ a)$. Cette construction est souvent utilisée pour enchaîner deux existentielles. Par exemple la spécification de la division euclidienne peut s'écrire :

$$(a, b : \texttt{nat})(b > 0) \to \{q : \texttt{nat}\ \&\ \{r : \texttt{nat} \mid a = bq + r \land r < b\}$$

Autres existentielles Il existe des variantes de ces existentielles lorsqu'on demande que deux conditions P et Q soient vérifiées. L'existentielle logique est également définie pour A : `Type` au lieu de A : `Set`. La disjonction $A \land B$ de deux propositions de type `Prop` ou le produit $A * B$ de deux spécifications de type `Set` peuvent également se voir comme des existentielles non dépendantes.

Disjonction

De même, il existe plusieurs définitions de conjonctions suivant si les arguments et le résultat sont dans `Prop` ou `Set`. Nous décrivons les plus utilisées :

$A \lor B$ Si A, B : `Prop` alors $A \lor B$: `Prop` représente la disjonction logique.

$\{A\} + \{B\}$ Si A, B : `Prop` alors $\{A\} + \{B\}$: `Set` spécifie un booléen qui vaut vrai si A est démontrable et faux si B est démontrable. Ceci montre en particulier que A ou B est vrai mais fournit, de plus, un moyen de décider lequel des deux est satisfait.

On peut d'ailleurs démontrer l'analogue de l'axiome du choix dans ce cas :

$$(A, B : \texttt{Prop})(\{A\} + \{B\}) \to \{b : \texttt{bool} \mid b = \texttt{true} \to A \land\ b = \texttt{false} \to B\}$$

Le terme extrait d'une preuve de $\{A\} + \{B\}$ est dans un type isomorphe au type des booléens (constructeurs `left` et `right`).

Cette forme est utilisée pour les preuves de décidabilité de prédicats logiques.

$A + \{B\}$ Lorsque l'on modélise des fonctions partielles ou des exceptions, soit une valeur est rendue, soit on est dans une situation particulière où une certaine propriété est vérifiée.

Il est alors commode de construire une spécification qui est une disjonction entre A : `Set` (le cas où une valeur est rendue) et une propriété logique B : `Prop` qui spécifie le cas exceptionnel. Cette disjonction est notée $A + \{B\}$, le terme extrait d'une preuve de $A + \{B\}$ est d'un type isomorphe au type A `option` de CAML.

Par exemple, la spécification d'un programme de recherche de l'entier minimum vérifiant une propriété P s'écrira :

$$(P : \texttt{nat} \to \texttt{Prop})(n : \texttt{nat})\{p : \texttt{nat} \mid (\texttt{min}\ P\ p)\} + \{(q : \texttt{nat})(q \leq n) \to \neg(P\ q)\}$$

Un autre exemple est la décision de l'existence d'un chemin dans un graphe, lorsqu'on souhaite explicitement calculer le chemin.

$A + B$ On peut évidemment faire la somme disjointe de deux spécifications A, B de type `Set`.

A.4.2 Exemple

Nous développons ici un exemple de preuves de programme, il s'agit de la fonction de calcul d'une puissance.

Nous en donnons une première version récursive qui servira à la spécification

```
Coq <     Require Import Arith.

Coq <     Fixpoint power (x n:nat) {struct n} : nat :=
Coq <       match n with
Coq <       | O => 1
Coq <       | S n' => x * power x n'
Coq <       end.
```

On peut écrire une version plus efficace qui utilise une méthode dichotomique. On justifie tout d'abord le principe de récurrence:

```
Coq <     Require Import Div2.

Coq <     Require Import Wf_nat.

Coq <     Require Import Compare_dec.

Coq <
Coq <     Definition dicho_rec :
Coq <       forall P:nat -> Set,
Coq <         P 0 ->
Coq <         (forall n:nat, 0 < n -> P (div2 n) -> P n) -> forall n:nat, P n.

Coq <     Proof.

Coq <     intros; apply well_founded_induction with (R := lt).

Coq <     exact lt_wf.

Coq <     intro x.

Coq <     case (zerop x).

Coq <     intro H1; rewrite H1; auto.

Coq <     intros; apply H0; auto with arith.

Coq <     Defined.
```

```
Coq <     Require Import Even.

Coq <     Definition square (n: nat) := n * n.

Coq <     Definition power2 (x: nat) : nat -> nat :=
Coq <       dicho_rec (fun _ => nat) 1
Coq <         (fun k kpos y =>
Coq <            if even_odd_dec k
Coq <            then fun _ => square y
Coq <            else fun _ => x * square y).
```

Maintenant on pourrait essayer de montrer que les fonctions `power` et `power2` coïncident, mais cette preuve est très laborieuse. Une meilleure solution est de construire directement une preuve de $(n, m : nat)\{p : nat \mid p = (\texttt{power}\ n\ m)\}$ dont le programme extrait soit le programme `power2` :

```
Coq <     Load "exp".

Coq <     Definition power2_proof :
Coq <       forall x n:nat, {p : nat | p = power x n}.

Coq <     Proof.

Coq <     intros x n.
```

```
Coq <     apply dicho_rec with (P := fun n:nat => {p : nat | p = power x n}).
2 subgoals

  x : nat
  n : nat
  ============================
   {p : nat | p = power x 0}
subgoal 2 is:
 forall n0 : nat,
 0 < n0 ->
 {p : nat | p = power x (div2 n0)} -> {p : nat | p = power x n0}

Coq <     exists 1; trivial.
1 subgoal

  x : nat
  n : nat
  ============================
   forall n0 : nat,
   0 < n0 ->
   {p : nat | p = power x (div2 n0)} -> {p : nat | p = power x n0}

Coq <     intro k; simple destruct 2; intros y Hy; case (even_odd_dec k); intros.
2 subgoals

  x : nat
  n : nat
  k : nat
  H : 0 < k
  H0 : {p : nat | p = power x (div2 k)}
  y : nat
  Hy : y = power x (div2 k)
  e : even k
  ============================
   {p : nat | p = power x k}
subgoal 2 is:
 {p : nat | p = power x k}

Coq <     exists (square y); rewrite Hy; auto with arith.
1 subgoal

  x : nat
  n : nat
  k : nat
```

```
  H : 0 < k
  H0 : {p : nat | p = power x (div2 k)}
  y : nat
  Hy : y = power x (div2 k)
  o : odd k
  ============================
   {p : nat | p = power x k}

Coq <     exists (x * square y); rewrite Hy; auto with arith.
Proof completed.

Coq <     Defined.
```

A.4.3 Extraction de programmes CAML

Il y a deux phases :

- Suppression des preuves de $p : P$: Prop
- Traduction vers un programme ML

```
Coq <     Definition x := 2.

Coq <     Definition n := 10.

Coq <     Extraction "power.ml" x n power power2 power2_proof.
```

Programme CAML

```
> ocaml
        Objective Caml version 3.04

# #use "power.ml";;
type prop = unit
val prop : unit = ()
type arity = unit
val arity : unit = ()
type nat = O | S of nat
val x : nat = S (S O)
val n : nat = S (S (S (S (S (S (S (S (S (S O)))))))))
val plus : nat -> nat -> nat = <fun>
val mult : nat -> nat -> nat = <fun>
val power : nat -> nat -> nat = <fun>
val acc_rec : ('a -> unit -> ('a -> 'b -> 'c) -> 'c) -> 'a -> unit -> 'c =
  <fun>
type sumbool = Left | Right
val div2 : nat -> nat = <fun>
val dicho_rec : 'a -> (nat -> unit -> 'a -> 'a) -> nat -> 'a = <fun>
val even_odd_dec : nat -> sumbool = <fun>
val square : nat -> nat = <fun>
val power2 : nat -> nat -> nat = <fun>
type 'a sig0 = 'a
val power2_proof : nat -> nat -> nat = <fun>
```

Test On peut lier les entiers de `Coq` aux entiers de CAML afin de tester les fonctions.

```
let rec int_of_nat = function O -> 0 | (S n) -> 1+(int_of_nat n);;
int_of_nat (power x n);;
int_of_nat (power2 x n);;
int_of_nat (power2_proof x n);;
- : int = 1024
```

Termes extraits On constate que les fonctions `power2` et `power2_proof` ont exactement le même code extrait.

A.4.4 D'un programme fonctionnel à la preuve

Il est parfois difficile de contrôler la forme du programme extrait à partir de la preuve. Des tactiques facilitent cette tâche.

`Refine` La tactique `Refine` permet de donner un terme de preuve partiel pour résoudre un but.

```
Coq <     Notation sol := (exist _).

Coq <     Definition power2_refine :
Coq <       forall x n:nat, {p : nat | p = power x n}.

Coq <
Coq <     refine
Coq <      (fun x =>
Coq <        dicho_rec (fun n:nat => {p : nat | p = power x n}) (
Coq <          sol 1 _)
Coq <          (fun k kpos r =>
Coq <             let (y, _) := r in
Coq <             if even_odd_dec k
Coq <             then fun _ => sol (square y) _
Coq <             else fun _ => sol (x * square y) _)).
```

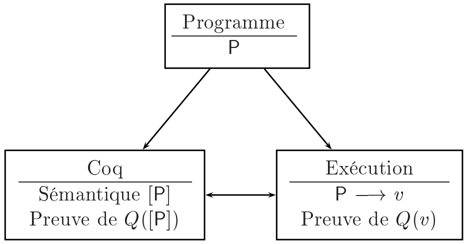
`Program` La tactique `Program` développée par C. Parent permettait d'associer à un but un programme puis d'interpréter ce programme comme l'extraction d'une preuve et de reconstruire les obligations de preuve. Cette tactique n'est plus accessible dans la version courante de Coq. Ce qu'elle produisait est mis en commentaire.

```
Definition power2_program : (x,n:nat){p:nat|p=(power x n)}.
Realizer power2.
Repeat Program.
(*
3 subgoals

  x : nat
  n : nat
  ============================
   (1)=(power x (0))

subgoal 2 is:
 (mult y0 y0)=(power x k)
```

```
subgoal 3 is:
 (mult x (mult y0 y0))=(power x k)
*)
```

A.4.5 Programmes impératifs

Récemment, J.-C. Filliâtre a étendu ce paradigme aux preuves de programmes non fonctionnels [Fil99, Fil03]. Le programme de départ est interprété dans `Coq` comme un programme purement fonctionnel (ce n'est rien d'autre que de la sémantique dénotationnelle), et l'on s'appuie sur un résultat de correction de la traduction. Le schéma est le suivant :



Cette méthode est mise en œuvre dans l'outil `Why` [Fil02], un outil produisant notamment des obligations de preuve pour `Coq` mais développé indépendamment de `Coq` (son langage logique d'entrée n'est pas celui de `Coq` et ses sorties ne sont pas limitées à `Coq`).

Conclusion

Les techniques d'extraction de programmes peuvent être utiles pour tester des spécifications, pour construire des versions de référence de logiciels critiques. Des exemples conséquents ont été développés dans le cadre du calcul formel, des vérificateurs de preuves, ou des compilateurs.

La construction de programmes certifiés, s'appuie sur la notion de réalisabilité mais peut-être grandement facilitée par la construction d'outils dédiés tels que les tactiques `Program` ou `Correctness`.

Bibliographie

[AVL62] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5):1259–1263, September 1962.

[BB95] S. Berardi and L. Boerio. Using subtyping in program optimization. In *Typed Lambda Calculus and Applications*, 1995.

[Bee85] M.J. Beeson. *Foundations of Constructive Mathematics, Metamathematical Studies*. Springer-Verlag, 1985.

[Ber96] S. Berardi. Pruning simply typed lambda terms. *Journal of Logic and Computation*, 6(5):663–681, 1996.

[Chr03a] Jacek Chrząszcz. Implementing modules in the system Coq. In *16th International Conference on Theorem Proving in Higher Order Logics*, University of Rome III, September 2003.

[Chr03b] Jacek Chrząszcz. *Modules in Type Theory with generative definitions*. PhD thesis, Warsaw University and Université Paris-Sud, 2003. To be defended.

[DP98] F. Damiani and F. Prost. Detecting and removing dead-code using rank 2 intersection. In *Proceedings TYPES'96*, LNCS, 1998.

[Fil99] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.

[Fil02] Jean-Christophe Filliâtre. The why verification tool, 2002. `http://why.lri.fr/`.

[Fil03] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003. English translation of [Fil99].

[FL04] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *Proceedings of The European Symposium on Programming*, Barcelona, Spain, March 29-April 2 2004. Voir aussi `http://www.lri.fr/~filliatr/fsets/`.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. Bibliotheca Mathematica. North-Holland, 1952.

[Ler00] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[Let03a] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Let03b] Pierre Letouzey. *Programmation fonctionnelle certifiée en Coq*. PhD thesis, Université Paris Sud, 2003. To be defended.

[PM89a] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In Association for Computing Machinery, editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989.

[PM89b] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, January 1989.

[Tak91] Y. Takayama. Extraction of redundancy-free programs from constructive natural deduction proofs. *Journal of Symbolic Computation*, 12:29–69, 1991.

[Tro73] A.S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. LNM 344. Springer-Verlag, 1973.

[TvD88] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, an introduction*. Studies in Logic and the foundations of Mathematics, volumes 121 and 123. North-Holland, 1988.