Master Parisien de Recherche en Informatique 2005–2006

cours 2-7-2

Assistants de preuve

Christine Paulin-Mohring, Benjamin Werner,

Bruno Barras, Hugo Herbelin,

Jean-Christophe Filliâtre, Claude Marché

Table des matières

Chapitre 1

Introduction au Calcul des Constructions Inductives

1.1 Motivations

Ce cours traite de la preuve formelle, comme discipline de l'informatique. Il prolonge donc le cours de tronc commun « Preuves Constructives », mais en mettant l'accent sur la construction effective de preuves formelles et de leur vérification sur ordinateur.

Si la logique mathématique, c'est-à-dire la formalisation complète du raisonnement est une discipline déjà relativement ancienne, elle a été relativement bouleversée par l'arrivée de l'ordinateur: la capacité de la machine à manipuler rapidement de grosses expressions symboliques permet de formaliser entièrement des raisonnements non-triviaux. La question que se pose le logicien n'est alors plus tant « existe-t-il une formalisation de tel raisonnement ? » que de construire et d'exhiber cette formalisation.

C'est avec cette observation à l'esprit que nous chercherons à traiter des points suivants:

Formalismes

Le choix du formalisme est important pour la pratique des mathématiques formelles. Non seulement il doit être capable d'exprimer la preuve conçue par le mathématicien, mais il doit permettre de le faire de la manière la plus facile et la plus intuitive possible. De fait, les évolutions récentes du Calcul des Constructions Inductives ont presque toujours été motivées par la pratique et sont donc postérieures aux premières implémentations de CoC puis Coq.

Modélisation

Plus encore qu'en mathématiques usuelles, savoir bien énoncer les propositions que l'on espère prouver est essentiel si l'on veut ultimement aboutir à une preuve formelle. Si l'on fait l'analogie entre les activités de prouver et programmer, alors le choix de la modélisation correspond à celui de la structure de représentation des données, avec les conséquences immédiates et évidentes sur l'architecture du logiciel obtenu.

Plus généralement, il nous semble que les analogies entre les activités de preuves et de programmation sont nombreuses, et s'il existe un « art » de la bonne programmation, il en va de même pour les preuves formelles. De plus, à chaque couple formalisme/problème, correspond un, ou plusieurs, style de bonne preuve, tout comme il existe de bons styles de programmation pour un problème et un langage de programmation donné.

Nous nous attacherons particulièrement à la modélisation et la formalisation de problèmes informatiques où la sûreté est importante, et à la preuve de correction de programmes.

Architecture des systèmes

Enfin nous chercherons à décrire les grands principes d'un logiciel d'assistant à la démonstration. Poursuivant encore l'analogie avec la programmation, ceci correspondrait à la description d'un compilateur. Nous nous intéresserons principalement au système de preuves Coq.

1.2 Quelques rappels sur les théories de types

Aujourd'hui, on désigne généralement par « théorie des types » un formalisme logique dont les objets sont des λ-termes typés. Il existe plusieurs formalismes rentrant dans cette catégorie, et ils se distinguent essentiellement par le système de types plus ou moins riche des objets, ainsi que par la logique pour parler de ces objets. On peut citer en particulier la logique d'ordre supérieur de Church, la Théorie des Types de Martin-Löf, la logique du système PVS et le Calcul des Constructions Inductives (CCI) dont les variantes implantées par Coq seront l'objet et le support premier de ce cours.

Les objets de la logique d'ordre supérieur de Church sont les λ-termes simplement typés. Les trois autres formalismes utilisent essentiellement des variantes (ou plutôt des fragments) du langage de programmation ML. Ils se distinguent par leur logique:

- PVS utilise un calcul des prédicats classique, sous forme de calcul des séquents; ses objets sont un fragment fortement terminant de ML, enrichi par une notion de sous-type. Les preuves en revanche ne sont pas un objet du formalisme.
- La théorie des types de Martin-Löf est une logique prédicative. Les preuves sont elles-mêmes des λ-termes et l'accent est mis sur la constructivité.
- Le Calcul des Constructions Inductives est lui une extension de la logique d'ordre supérieur et autorise la quantification imprédicative sur toutes les propositions.

Sur tous ces points, nous renvoyons bien sûr au cours de tronc commun. La plupart seront par ailleurs illustrés tout au long du cours.

Un point commun essentiel de CCI et de la théorie de Martin-Löf est bien sûr qu'ils sont construits sur l'isomorphisme de Curry-Howard. Rappelons que cela signifie que:

- les preuves sont des objets,
- les propositions sont des types,
- une preuve d'une proposition P est un objet p de type P.

En simplifiant, on peut dire que les avantages informatiques de cette approche sont:

- Une plus grande homogénéité preuves/objets, qui simplifie l'implémentation du système de preuves.
- Un statut bien compris des preuves comme objets du formalisme; en conséquence, elles ont également une représentation claire dans l'implémentation.
- Une articulation entre calcul et déduction qui permet des preuves particulièrement concises dans certains cas (depuis l'exemple $2 + 2 = 4$ ci-après aux preuves par réflexion comme la tactique `ring`).

1.3 Un premier contact avec Coq: Curry-Howard en action

Le système Coq est un système de traitement de preuves pour une version prédicative du Calcul des Constructions Inductives. Les composantes essentielles du système Coq sont:

- un noyau de vérification de types et de construction d'environnements bien typés
- un langage de développement de théories mathématiques: *Gallina*
- un outillage d'aide à la construction interactive de preuves par des *tactiques de preuve*

1.3.1 Syntaxe de base du formalisme

Le formalisme implanté par Coq sera donc couramment appelé CCI. Il s'agit pour l'essentiel d'une extension du Calcul de Constructions (CC) traité en tronc commun. Il nous faut dès maintenant signaler quelques différences de notations avec la syntaxe employée jusqu'ici.

Tout d'abord, Coq est conçu pour pouvoir être utilisé par une interface en mode texte (ASCII ou Unicode). Les notations de Coq pour $\lambda x : A.t$ et $\Pi x : A.B$ sont:

- La λ-abstraction est notée `fun x:A => t`. Cela désigne la fonction qui à un objet `x` de type `A` associe l'objet `t`.
- La quantification universelle est notée `forall x:A, B`. Cet objet est le type des fonctions qui à un objet `x` de type `A` associent un objet de type `B`. Comme en tronc commun, on utilisera la flèche pour simplifier l'écriture de ce type lorsque `x` n'apparaîtra pas dans `B`. En ASCII cela donne `A->B`.

Par ailleurs, essentiellement pour des raisons historiques, les *sortes* portent d'autres noms en Coq que dans le cours de tronc commun. Rappelons que les sortes sont des constantes particulières, qui sont les types des types. Dans le cours de tronc commun, on utilisait une sorte *Type* qui est le type des types, et une sorte *Kind* qui était le type de *Type*. Cette dernière servant essentiellement à énoncer les règles correspondant au *polymorphisme*.

Dans les versions de Coq antérieures à la version 8.0, *Type* est remplacée par deux sortes « jumelles », `Prop` et `Set`. Intuitivement, `Set` contient les types de données (objets « calculatoires » qui sont pris en compte par le processus d'extraction de programmes de Coq – cf le chapitre consacré) et `Prop` les énoncés « logiques » (qui sont « oubliés » par le processus d'extraction).

Depuis la version 8.0 de Coq, `Prop` garde le rôle joué par *Type* dans le Calcul des Constructions traité dans le cours de tronc commun. Il reste une sorte nommée `Set` mais qui est une version prédicative de *Type*. Typiquement, le type `forall A:Set, A->A` ne peut pas s'appliquer à lui-même en Coq version 8.0.

Pour ne rien arranger, la sorte *Kind* se trouve renommée `Type` dans Coq. Au lecteur qui serait tenté de voir là une source de confusion, rappelons simplement que l'informatique est sans doute la science de la bureaucratie et d'une vision un peu tyrannique de l'ordre et particulièrement du renommage…

1.3.2 Vérification et inférence de types

A partir de là, nous pouvons taper nos premières commandes pour utiliser le vérificateur de type. En syntaxe Coq, l'identité polymorphe est donc notée `fun (A:Set) (a:A) => a`. On demande au système de vérifier la bonne-formation d'un objet par la commande `Check`; les commandes Coq étant toujours terminées par un point, cela donne:

```
Coq < Check (fun (A:Set) (a:A) => a).
fun (A : Set) (a : A) => a
     : forall A : Set, A -> A
```

Toutes les opérations de typage de Coq ont lieu dans un *environnement* global. Cet environnement correspond au contexte Γ des jugements de typage $\Gamma \vdash t : T$. Il est donc possible de pousser de nouvelles variables dans l'environnement:

```
Coq < Variable A : Prop.
A is assumed

Coq < Variable a : A.
a is assumed
```

De plus, l'environnement de Coq peut également contenir des *constantes*, ou abréviations. Pour cela, on associe un terme à un nom; de plus, l'environnement mémorise aussi un des types du corps de la constante[1].

```
Coq < Definition Id (B:Set) (b:B) := b.
Id is defined

Coq < Check Id.
Id
     : forall B : Set, B -> B

Coq < Print Id.
Id = fun (B : Set) (b : B) => b
     : forall B : Set, B -> B
Argument scopes are [type_scope _]
```

1.3.3 Deux tactiques de base

Construire une preuve revient, dans notre formalisme, à exhiber un λ-terme du type attendu. L'utilisateur dispose pour cela d'un mode de preuve interactif. Les commandes de ce mode interactif sont appelées *tactiques* de preuve. Voici une illustration sommaire de leur principe, sur un exemple simple; on se donne trois variables propositionnelles `A`, `B` et `C`, puis l'on cherche à prouver la tautologie suivante:

```
Coq < Variables A B C : Prop.
A is assumed
B is assumed
C is assumed

Coq < Lemma exemp1 : ((A -> B) -> C) -> B -> C.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   ((A -> B) -> C) -> B -> C
```

On a alors le lemme comme seul but courant, sous la double barre. On peut commencer à construire la preuve:

```
Coq < intros c b.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  c : (A -> B) -> C
  b : B
  ============================
   C
```

[1]Rappelons qu'en raison de la règle de *conversion*, un terme bien-formé peut avoir plusieurs types dans CC ou CCI.

On voit que `(A->B)->C` et `B` ont été poussées comme hypothèses dans le contexte local, au-dessus de la double barre. Elles ont été nommées comme demandé.

Du point de vue de la construction de la preuve, cette tactique correspond à la λ-abstraction comme on peut le voir en affichant la preuve partielle construite.

```
Coq < Show Proof.
LOC:
Subgoals
1 -> ((A -> B) -> C) -> B -> C
Proof: fun (c : (A -> B) -> C) (b : B) => ?1 c b
```

Une preuve partielle peut comporter plusieurs points d'interrogations, c'est-à-dire que l'on peut avoir plusieurs sous-buts simultanément, chacun avec son contexte local.

La tactique `apply` correspond à l'application:

```
Coq < apply c.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  c : (A -> B) -> C
  b : B
  ============================
   A -> B
```

On est alors passé au terme de preuve partiel `fun (c:(A->B)->C) (b:B) => c ?1` où le but courant est maintenant `A->C`. On finit donc la preuve par

```
Coq < intros a.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  c : (A -> B) -> C
  b : B
  a : A
  ============================
   B

Coq < exact b.
Proof completed.
```

La commande `Qed` (ou `Save`) permet alors d'ajouter à l'environnement global le terme `exemp1` ainsi crée:

```
Coq < Qed.
intros c b.
apply c.
intros a.
exact b.
```

```
exemp1 is defined

Coq < Print exemp1.
exemp1 =
fun (c : (A -> B) -> C) (b : B) => c (fun _ : A => b)
     : ((A -> B) -> C) -> B -> C
```

Rassurons enfin le lecteur: une preuve aussi simple peut, heureusement, également être trouvée automatiquement par les tactiques de preuve plus évoluées dont dispose le système. Plus généralement, ce document n'est d'ailleurs pas un manuel d'utilisateur, ni un cours de Coq. Nous renvoyons pour cela à la documentation standard de Coq [ea99].

1.4 Les Entiers Naturels dans CCI

1.4.1 Définition

Le type des entiers naturels est le plus petit type contenant 0 et clos par le successeur S; une telle définition par *plus petit point fixe* est appelée une *définition inductive*.

La syntaxe en Coq d'une telle définition est:

```
Coq < Inductive nat : Set :=
Coq < | O : nat
Coq < | S : nat -> nat.
```

Cette commande ajoute à l'environnement du système les objets suivants:

– le type `nat : Set`
– les deux objets `O : nat` et `S : nat -> nat` appelés *constructeurs* de `nat`.

En général, cette définition (ainsi que la plupart de celles qui suivent) sont déjà présentes dans l'environnement du système au lancement.

```
Coq < Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat

Coq < Check nat.
nat
     : Set

Coq < Check O.
O
     : nat

Coq < Check S.
S
     : nat -> nat
```

Remarque Informellemment, le plus petit type ainsi défini est celui dont les habitants sont `O`, `(S O)`, `(S (S O))`, etc. On a donc bien une représentation de la notion mathématique d'entier naturel.

Remarque La vision informatique est que cette définition est la version Coq du type concret ML bien connu:

```
# type nat = O | S of nat;;
```

On peut remarquer au passage, qu'en Coq le constructeur `S` est fonctionnel et peut donc exister sans son argument. Il s'agit là d'un point syntaxique, d'importance marginale.

Remarque D'un point de vue ensembliste, les entiers naturels sont le plus petit point fixe de l'opérateur suivant:

$$F(X) \equiv \{0\} \cup \{(S\ x), x \in X\}.$$

Cet opérateur est monotone, et admet donc un plus petit point fixe. En effet, l'univers des ensembles est un treillis complet par rapport à l'ordre d'inclusion.

Cette possibilité de voir la définition comme un plus petit point fixe sera commune à toutes les définitions inductives. Par ailleurs on verra plus tard, comment ces points fixes apparaissent également dans la sémantique de la théorie.

Dans ce chapitre, nous n'expliciterons pas les règles de typage qui sous-tendent les définitions inductives. Elles seront explicitées en partie lors du cours 3.

1.4.2 Syntaxe alternative

Par défaut, les entiers de `nat` sont représentés en Coq via la notation standard en base 10. Par exemple, la notation `3` désigne l'entier naturel `(S(S(S O)))`. Qu'il soit clair que ces deux écritures désignent le même terme et sont représentées de manière identique dans le système.

On verra plus loin que les symboles infixes +, *, <=, ... désignent aussi par défaut en Coq les opérations arithmétiques sur `nat`.

1.4.3 Fonctions sur les entiers

Deux exemples simples

Les entiers sont l'archétype du type de données récursif. Tout comme en ML, l'on calcule sur ces entiers grâce à des fonctions définies par deux mécanismes fondamentaux:

- le filtrage
- la récursion.

Le premier exemple d'une telle fonction est en général l'addition. Voici sa définition en ML:

```
let rec plus n m =
 match n with
  O -> m
| (S p) -> S(plus p m);;
```

et l'équivalent en syntaxe Coq:

```
Coq < Fixpoint plus (n m:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | O => m
Coq <   | S p => S (plus p m)
Coq <   end.
```

Et voici, de même, une définition de la multiplication:

```
Coq < Fixpoint mult (n m:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | O => O
Coq <   | S p => plus m (mult p m)
Coq <   end.
```

La notion de récursion structurelle

On comprend bien que la construction `Fixpoint` correspond à la définition d'une fonction récursive, au même titre que le `let rec` de ML. On note en revanche que le premier des deux arguments de chacune des deux fonctions est syntaxiquement distingué par l'emploi du mot-clé `struct`. La raison en est simple: lorsque la théorie des types, comme ici, est utilisée en tant que formalisme logique, la cohérence du formalisme est essentiellement assurée par la propriété de *normalisation*, c'est-à-dire de la terminaison des calculs. Une sommaire justification informelle pourrait être donnée ainsi: supposons qu'il soit possible de définir la fonction suivante.

```
Coq < Fixpoint non_sens (n: nat) : nat := non_sens n.
```

Il est alors clair que l'objet `(non_sens O)` ne correspond mathématiquement pas à un entier naturel, et ne saurait être accepté dans le formalisme.

On reviendra par la suite à l'étude de la propriété de normalisation, et renverra, pour l'instant, au cours de tronc commun. Retenons pour l'instant que:

> Toutes les fonctions récursives acceptées par le système doivent terminer.

Pour pouvoir définir des règles de typage, il importe donc d'isoler une classe de fonctions récursives terminantes. Pour ce faire, on généralise la classe des fonctions définissables dans le système T de Gödel:

Définition 1 *On considère un terme de type* `nat` *de la forme* `(S n)`*. Sont considérés comme* structurellement plus petit *que* `(S n)` *les termes suivants:*
- `n`
- *tout terme structurellement plus petit que* `n`*.*

Une fonction est structurellement récursive si l'on peut distinguer l'un de ses arguments, qui décroit structurellement à chaque appel récursif.

En Coq, seules des fonctions structurellement récursives peuvent être définies par `Fixpoint`.

Les fonctions `plus` et `mult` définies ci-dessus sont structurellement récursives par rapport à leur premier argument. Bien sûr, la fonction `non_sens` est rejetée par le système.

Voici une définition alternative de l'addition, qui décroit par rapport à son second argument.

```
Coq < Fixpoint plus' (n m:nat) {struct m} : nat :=
Coq <   match m with
Coq <   | O => n
Coq <   | S p => S (plus' n p)
Coq <   end.
```

Fonctions plus complexes

Dans le système T, seul `n` est considéré comme structurellement plus petit que `(S n)`; la définition ci-dessus est plus souple. Par exemple voici une définition possible du quotient de la division entière par deux:

```
Coq < Fixpoint div2 (n:nat) : nat :=
Coq <  match n with
Coq <  | O => O
Coq <  | 1 => O
Coq <  | (S (S p)) => (S (div2 p))
Coq <  end.
```

Il faut noter que cette fonction pourrait également être définie dans le système T, mais de manière un peu plus lourde. Il s'agit donc là d'un aménagement du formalisme qui n'étend pas l'expressivité du formalisme, mais juste son confort d'utilisation.

Dans un registre différent, le mécanisme de récursion structurelle permet la définition de fonctions logiquement complexes, c'est-à-dire qui croissent très vite. L'exemple le plus connu est la fonction due à Ackermann, dont voici la définition ML:

```
let rec ack = function
   O,m -> S(m)
 | S(n),O -> ack(n,(S O))
 | S(n),S(m) -> ack(n,ack(S(n),m));;
```

La terminaison de cette définition est assurée par une décroissance de la paire d'arguments vis-à-vis de l'ordre lexicographique. En terme de récursion structurelle, ceci est codé par l'utilisation de deux récursions emboîtées. En Coq, la syntaxe est alors un peu plus complexe; la commande `fix` jouant le rôle d'un `let rec...in`:

```
Coq < Fixpoint ack (n:nat) : nat -> nat :=
Coq <   match n with
Coq <   | O => fun m:nat => S m
Coq <   | S n' =>
Coq <      (fix aux (m:nat) : nat :=
Coq <         match m with
Coq <         | O => ack n' (S O)
Coq <         | S m' => ack n' (aux m')
Coq <         end)
Coq <   end.
```

1.4.4 Calculer pour raisonner

Donnons un exemple simple (et classique) d'utilisation de la règle de conversion; il s'agit de prouver $2+2=4$ où $+$ est la notation infixe de Coq pour `plus` et `2` et `4` les représentations de `S (S O)` et `S (S (S (S O)))`. Ici, cette proposition s'énonce:

```
Coq < Lemma deux_et_deux : 2 + 2 = 4.
1 subgoal

  ============================
   2 + 2 = 4
```

Il suffit d'observer que le terme correspondant à $2+2$ se réduit effectivement vers `4`, et donc que la proposition est logiquement identifiée à `4=4`:

```
Coq < simpl.
1 subgoal

  ============================
   4 = 4

Coq < reflexivity.
Proof completed.
```

Remarquons que la tactique `simpl`, qui procède donc à la β-normalisation du but courant, *ne construit pas de terme-preuve.* Cela est dû à la forme de la règle de conversion qui, rappelons-le est:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash t : B}\ Conv$$

On voit bien que les termes-preuves de la conclusion et de la prémisse principale sont identiques.

En conséquence, l'on pourrait dans l'exemple précédent, se passer complètement de `simpl` et utiliser juste `reflexivity`, qui construit la preuve correspondant à la réflexivité de l'égalité. La définition exacte du prédicat d'égalité sera détaillée plus tard.

Exercice 1 *Prouver à partir de là, la commutativité de l'addition.*

1.4.5 Le schéma de récurrence des entiers naturels

Le fait que `nat` est bien le plus petit type clos par ses deux constructeurs est exprimé par le *schéma de récurrence.* Il s'agit d'une propriété logique qui s'énonce ainsi: soit P un prédicat sur les entiers; pour que la proposition $P(n)$ soit vraie pour tout entier n, il suffit que les deux conditions suffisantes soient vérifiées:
- $P(0)$ est vrai,
- pour tout entier n, si $P(n)$ est vraie, alors $P(S(n))$ l'est aussi.

Dans une logique d'ordre supérieur, ce schéma peut être exprimé comme une proposition. En syntaxe Coq:

```
forall (P: nat -> Prop),
       (P O) ->
       (forall (m:nat), (P m)->(P (S m)))->
          forall (n:nat), (P n)
```

De fait, la définition d'un type inductif génère automatiquement une preuve du schéma de récurrence correspondant. Si le type inductif est nommé `I`, la preuve du schéma de récurrence s'appellera `I_ind`; par exemple:

```
Coq < Check nat_ind.
nat_ind
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

L'utilisation du schéma de récurrence en Coq se fait grâce aux tactiques `elim` *et* `induction`.

1.4.6 Une preuve par récurrence

Voici un autre exemple simple, combinant le calcul et le raisonnement par récurrence. Il s'agit de prouver une première étape vers la commutativité de l'addition, à savoir:

```
Coq < Lemma comm_0 : forall n:nat, n = n + 0.
1 subgoal

  ============================
   forall n : nat, n = n + 0
```

Cette preuve se fait bien sûr par récurrence sur `n`; le cas de base est trivial:

```
Coq < simple induction n.
2 subgoals

  n : nat
  ============================
   0 = 0 + 0
subgoal 2 is:
 forall n0 : nat, n0 = n0 + 0 -> S n0 = S n0 + 0

Coq < reflexivity.
1 subgoal

  n : nat
  ============================
   forall n0 : nat, n0 = n0 + 0 -> S n0 = S n0 + 0
```

Pour le cas par récurrence, on peut procéder à une étape de réduction:

```
Coq < simpl; intros p HR.
1 subgoal

  n : nat
  p : nat
  HR : p = p + 0
  ============================
   S p = S (p + 0)
```

Il suffit alors d'utiliser l'hypothèse de récurrence pour remplacer dans le but courant, `p+0` par `p`; cela peut se faire par:

```
Coq < rewrite <- HR.
1 subgoal

  n : nat
  p : nat
  HR : p = p + 0
  ============================
   S p = S p

Coq < reflexivity.
Proof completed.
```

1.5 D'autres types de données courants

Comme en ML, le principe de définitions inductives permet la construction de types de données plus complexes que les entiers. Suivent quelques exemples courants.

1.5.1 Listes d'entiers

Il est sans doute inutile de rappeler la structure des listes, avec leurs deux constructeurs `nil` et `cons`. La commande définissant les listes d'entiers naturels est bien sûr:

```
Coq < Inductive list : Set :=
Coq < | nil : list
Coq < | cons : nat -> list -> list.
```

Une première remarque, marginale, est que contrairement à ML, les constructeurs peuvent avoir plusieurs arguments: ici `cons` est curryfié.

1.5.2 Listes paramétrées

Il est préférable de remplacer la définition précédente par une autre où le type des éléments des listes est un paramètre. La bonne définition des listes est:

```
Coq < Inductive list (A:Set) : Set :=
Coq < | nil : list A
Coq < | cons : A -> list A -> list A.
```

Il s'agit de la même définition, mais paramétrée par le type `A`. Cette abstraction est possible grâce aux types fonctionnels:

```
Coq < Check list.
list
     : Set -> Set

Coq < Check nil.
nil
     : forall A : Set, list A

Coq < Check cons.
cons
     : forall A : Set, A -> list A -> list A
```

A titre d'exemple, on écrira ainsi l'équivalent de l'objet ML $[1;2;3]$:

```
Coq < Check (cons nat 1 (cons nat 2 (cons nat 3 (nil nat)))).
cons nat 1 (cons nat 2 (cons nat 3 (nil nat)))
     : list nat
```

À condition de recourir à la définition définie dans le module `List` de la bibliothèque standard de Coq, on peut aussi utiliser la notation suivante:

```
Coq < Require Import List.

Coq < Check (1 :: 2 :: 3 :: nil).
1 :: 2 :: 3 :: nil
     : list nat
```

Le schéma de récurrence structurelle sur les listes est:

```
Coq < Check list_ind.
list_ind
     : forall (A : Set) (P : list A -> Prop),
       P (nil A) ->
       (forall (a : A) (l : list A), P l -> P (cons A a l)) ->
       forall l : list A, P l
```

On reconnaît la même structure que pour les entiers, avec toutefois l'argument supplémentaire de `cons` et la paramétrisation par rapport à `A`.

1.5.3 Types somme et produit

Un exemple courant de type paramétré est le type somme:

```
Coq < Inductive sum (A B:Set) : Set :=
Coq < | inl : A -> sum A B
Coq < | inr : B -> sum A B.
```

Son schéma d'élimination est plus simple, puisque le type est non récursif; il s'agit juste d'exprimer que tout élément de `(sum A B)` ne peut être construit qu'à partir de l'un des deux constructeurs:

```
Coq < Check sum_ind.
sum_ind
     : forall (A B : Set) (P : sum A B -> Prop),
       (forall a : A, P (inl A B a)) ->
       (forall b : B, P (inr A B b)) -> forall s : sum A B, P s
```

1.6 Types inductifs plus complexes

Dans tous les exemples de types récursifs vus jusqu'ici, l'ordre correspondant à la récursion (et la récurrence) structurelle se confondait avec la relation de sous-terme. Le mécanisme des types inductifs autorise toutefois des constructions plus générales.

1.6.1 Ordinaux

La définition qui suit est souvent appelé « type des ordinaux » par abus de langage. Il s'agit en fait d'une notation ordinale, qui ne permet que la représentation d'un fragment des ordinaux constructible dans CCI. Il s'agit d'une copie du type des entiers naturels, étendue par un nouveau constructeur correspondant à la limite ordinale:

```
Coq < Inductive Ord : Set :=
Coq < | Oo : Ord
Coq < | So : Ord -> Ord
Coq < | lim : (nat -> Ord) -> Ord.
```

On remarque que le constructeur `lim` est récursif, mais que son argument est une suite entière d'ordinaux.

L'ordre de la récursion structurelle est alors généralisé de la manière suivante: quel que soit les termes n de type nat et f de type nat $\rightarrow$ ord, $(f\ n)$ est structurellement plus petit que (limit f).

Voici une définition légale de fonction sur ce type des ordinaux:

Cette vision de la récursion structurelle est également reflétée dans l'énoncé du schéma de récurrence du type:

```
Coq < Check Ord_ind.
Ord_ind
     : forall P : Ord -> Prop,
       P Oo ->
       (forall o : Ord, P o -> P (So o)) ->
       (forall o : nat -> Ord, (forall n : nat, P (o n)) -> P (lim o)) ->
       forall o : Ord, P o
```

C'est-à-dire que pour appliquer le schéma spécialisé à un prédicat P, il faut vérifier que si:

– étant donnée f de type nat $\rightarrow$ ord telle que
– pour tout entier n, $(f\ n)$ vérifie P

alors (lim f) vérifie également P.

1.6.2 Arbres arbitrairement branchants

On peut utiliser l'idée précédente pour définir un type d'arbre très général: en utilisant le polymorphisme on s'autorise à indexer les fils de chaque nœud par un type arbitraire.

```
Coq < Inductive Inf_tree : Type :=
Coq <   Node : forall A:Set, (A -> Inf_tree) -> Inf_tree.
Inf_tree is defined
Inf_tree_rect is defined
Inf_tree_ind is defined
Inf_tree_rec is defined
```

Ce type est très peu intuitif. Il utilise et combine toutes les ressources du formalismes et permet ainsi la construction de très nombreux éléments. De fait, le logicien anglais Peter Aczel a prouvé qu'il était possible d'encoder les éléments de la théorie des ensembles dans ce type.

1.7 Prédicats inductifs

Nous avons vu comment construire des objets concrets. Le mécanisme de définitions inductives permet également la définition d'objets plus logiques, et en particulier de prédicats. En particulier, ce sera en général la manière la plus commode d'isoler une partie des éléments d'un type inductif.

1.7.1 Entiers pairs

En théorie des ensembles, une définition possible de l'ensemble des entiers pairs est de dire que c'est le plus petit ensemble tel que:

– 0 est pair
– pour tout entier n, si n est pair, alors $n + 2$ est pair.

La même définition est possible dans CCI. Le prédicat even, « être pair », étant un objet de type nat $\rightarrow$ Prop. Les deux clauses de la définition ci-dessus correspondant aux deux constructeurs du *prédicat inductif*. En Coq:

```
Coq < Inductive even : nat -> Prop :=
Coq < | even0 : even 0
Coq < | evenS : forall n:nat, even n -> even (S (S n)).
```

On voit bien que la structure d'une preuve de parité est récursive, à l'image de la structure d'un terme de type `nat`. Ceci est reflété dans le schéma de récurrence qui permet de prouver des propriétés d'entiers pairs:

```
Coq < Check even_ind.
even_ind
     : forall P : nat -> Prop,
       P 0 ->
       (forall n : nat, even n -> P n -> P (S (S n))) ->
       forall n : nat, even n -> P n
```

1.7.2 L'ordre sur les entiers

Un exemple essentiel est l'ordre sur les entiers:

```
Coq < Inductive le (n:nat) : nat -> Prop :=
Coq < | le_n : le n n
Coq < | le_S : forall m:nat, le n m -> le n (S m).
```

Son principe de récurrence:

```
Coq < Check le_ind.
le_ind
     : forall (n : nat) (P : nat -> Prop),
       P n ->
       (forall m : nat, le n m -> P m -> P (S m)) ->
       forall n0 : nat, le n n0 -> P n0
```

Exercice Prouver:

```
forall (n m:nat), (le n m)->(le (S n) (S m))
forall (n:nat), (le O n)
forall (n m:nat), (le n m)->(le m p)->(le n p)
```

1.7.3 Un exemple dangereux

Voici un exemple pour illustrer les subtilités propres aux mathématiques formelles. On peut proposer une définition alternative à `le`:

```
Coq < Inductive le_a : nat -> nat -> Prop :=
Coq < | le_a0 : forall n:nat, le_a 0 n
Coq < | le_aS : forall n m:nat, le_a n m -> le_a (S n) (S m).
```

Or cette définition, qui semble raisonnable et est mathématiquement saine, est peu praticable telle quelle. En particulier la preuve de la transitivité est très pénible; on peut utiliser cette définition, mais pour certaines propriétés, il vaut mieux commencer par prouver d'abord l'équivalence avec la définition précédente. L'on risque sinon de s'ensabler rapidement.

Chapitre 2

Exemple de développement Gallina : Sémantique d'un mini-langage

Le but de ce cours est d'illustrer sur un exemple les fonctionnalités du langage de spécification Gallina associé au Calcul des Constructions Inductives et implanté dans l'assistant à la démonstration Coq.

L'exemple choisi traite de la sémantique d'un mini-langage de programmation impératif (typage, évaluation et sémantique axiomatique). Plusieurs solutions à la modélisation des différentes notions sont proposées. Les différentes constructions utilisées dans ce chapitre seront expliquées plus en détail dans les prochains cours.

2.1 Introduction

Gallina est le nom donné au langage de spécification de l'assistant Coq. Il permet de définir:

– des types de données structurés,
– des fonctions qui peuvent être récursives sur la structure des données,
– des relations spécifiées inductivement par un ensemble de propriétés de fermeture,
– des formules du calcul des prédicats d'ordre supérieur.

Ces caractéristiques du langage le rendent particulièrement adapté à la formalisation des mathématiques et notamment de l'informatique théorique.

On pourra notamment se faire une opinion sur ce slogan en consultant sur la page de Coq la liste des développements réalisés par les utilisateurs.

Les définitions intervenant en sémantique des langages se représentent particulièrement bien en Gallina:

– les arbres de syntaxe abstraite se codent naturellement comme des types de données structurés,
– l'ordre supérieur permettra de manipuler aisément les mémoires qui pourront être représentées par des fonctions ou bien les assertions qui sont des prédicats sur la mémoire,
– les définitions sémantiques telles que le typage ou l'évaluation, lorsqu'elles sont présentées par un ensemble de règles d'inférence se traduisent immédiatement en définitions inductives.

Nous détaillons ici la formalisation de la sémantique d'un petit langage de programmation dans le style impératif.

Ce genre de formalisation (aussi appelé *plongement profond*) constitue une alternative à la preuve de programme impératif. Une autre alternative est le plongement simple (aussi appelé *plongement superficiel*) qui simule les propriétés des programmes impératifs à partir de leur interprétation dans un langage purement fonctionnel. La sémantique est alors implicite dans la traduction. Cette question sera évoquée dans le chapitre traitant de la preuve de programmes impératifs.

2.2 Présentation du problème

Notre langage comprend les commandes suivantes, à partir d'un ensemble de variables X :

$$
\begin{array}{rl}
C := & \mathtt{skip} \\
| & X \mathtt{:=} E \\
| & C_1 ; C_2 \\
| & \mathtt{if}\ E\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \\
| & \mathtt{while}\ E\ \mathtt{do}\ C
\end{array}
$$

FIG. 2.1 – Syntaxe des commandes

Les expressions sont formées de constructions entières et booléennes simples et sont données par la syntaxe de la figure 2.2.

$$
\begin{array}{lll}
E := & X^s & \textit{Variables sortées} \\
| & \mathtt{true} \mid \mathtt{false} & \textit{Constantes booléennes} \\
| & E_1\ \mathtt{xor}\ E_2 & \textit{Ou exclusif} \\
| & n & \textit{Constantes entières} \\
| & \mathtt{null}\ E & \textit{Teste si un entier est nul} \\
| & E_1\ \mathtt{op}\ E_2 & \textit{Opération binaire sur les entiers} \\
s := & \mathsf{nat} \mid \mathsf{bool} & \\
\mathtt{op} := & + \mid - \mid * \mid \ldots &
\end{array}
$$

FIG. 2.2 – Syntaxe des expressions

On cherche à définir des sémantiques statiques, opérationnelles naturelles et axiomatiques pour ce langage. Toutes sont représentées en sémantique naturelle par la définition d'une relation entre un programme et des "valeurs". Cette relation sera décrite à l'aide des règles d'inférence.

2.2.1 Sémantique statique

Il s'agit de déterminer de manière statique sans l'exécuter qu'un programme est bien formé. Ici il faut vérifier que dans les expressions conditionnelles ou de boucle, les tests sont faits sur des expressions booléennes. Cela nous amène à définir une relation de typage sur les expressions. Les deux valeurs seront les deux sortes nat et bool.

La relation de typage $E : s$ est définie par les axiomes et règles de la figure 2.3.

$$
X^s : s \qquad \mathtt{true} : \mathsf{bool} \qquad \mathtt{false} : \mathsf{bool} \qquad n : \mathsf{nat}
$$

$$
\frac{E_1 : \mathsf{bool} \quad E_2 : \mathsf{bool}}{E_1\ \mathtt{xor}\ E_2 : \mathsf{bool}} \qquad \frac{E : \mathsf{nat}}{\mathtt{null}\ E : \mathsf{bool}} \qquad \frac{E_1 : \mathsf{nat} \quad E_2 : \mathsf{nat}}{E_1\ \mathtt{op}\ E_2 : \mathsf{nat}}
$$

FIG. 2.3 – Sémantique statique des expressions

Pour les commandes, on définit la relation $C : \mathsf{ok}$ par les règles de la figure 2.4.

$$\texttt{skip} : \ \mathsf{ok} \qquad \frac{E : s}{X \texttt{:=} E : \ \mathsf{ok}} \qquad \frac{C_1 \ \mathsf{ok} \quad C_2 \ \mathsf{ok}}{C_1 ; C_2 : \ \mathsf{ok}}$$

$$\frac{E : \mathsf{bool} \quad C_1 : \ \mathsf{ok} \quad C_2 : \ \mathsf{ok}}{\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 : \ \mathsf{ok}} \qquad \frac{E : \mathsf{bool} \quad C : \ \mathsf{ok}}{\texttt{while } E \texttt{ do } C : \mathsf{ok}}$$

FIG. 2.4 – Sémantique statique des commandes

2.2.2 Sémantique opérationnelle

La sémantique opérationnelle définit le programme comme une transformation de l'état de la mémoire. Cette mémoire associe à chaque variable et sorte une valeur qui sera une constante entière n ou booléenne b. Les deux opérations utiles sur la mémoire sont la lecture et la mise à jour: si x est une variable et s une sorte alors $m(x, s)$ représente la valeur de la mémoire pour la variable x et la sorte s, si v est une valeur alors $m[x \leftarrow v]$ représente la mémoire où la variable x a été mise à jour par la valeur v. On a choisi de ne pas indiquer explicitement la sorte de la variable affectée, celle-ci sera déduite de la sorte de la valeur v.

Sémantique des expressions

On a besoin de la relation qui associe à chaque mémoire m et expression E une valeur v représentant le résultat de l'évaluation de E dans la mémoire m. On note cette relation $m \vdash E \triangleright v$. On utilisera des constantes et des fonctions sur le domaine des valeurs réalisant les opérations booléennes et arithmétiques. Elles seront notées en italique en utilisant le même identificateur que dans la syntaxe: ainsi, à la construction syntaxique `xor` correspond la fonction sémantique *xor*.

$$m \vdash X^s \triangleright m(X, s) \qquad m \vdash \texttt{true} \triangleright \mathit{true} \qquad m \vdash \texttt{false} \triangleright \mathit{false} \qquad m \vdash n \triangleright n$$

$$\frac{m \vdash E_1 \triangleright b_1 \quad m \vdash E_2 \triangleright b_2}{m \vdash E_1 \texttt{ xor } E_2 \triangleright b_1 \ \mathit{xor} \ b_2} \qquad \frac{m \vdash E \triangleright n}{m \vdash \texttt{null } E \triangleright \mathit{null} \ n} \qquad \frac{m \vdash E_1 \triangleright n_1 \quad m \vdash E_2 \triangleright n_2}{m \vdash E_1 \texttt{ op } E_2 \triangleright n_1 \ \mathit{op} \ n_2}$$

FIG. 2.5 – Sémantique des expressions

Sémantique des commandes

La relation $m \vdash C \triangleright m'$ exprime que la commande C s'évalue dans une mémoire m qu'elle transforme en une mémoire m'. On la définit par les règles d'inférence de la figure 2.6.

$$m \vdash \texttt{skip} \triangleright m \qquad \frac{m \vdash E \triangleright v}{m \vdash X \texttt{:=} E \triangleright m[X \leftarrow v]} \qquad \frac{m \vdash C_1 \triangleright m_1 \quad m_1 \vdash C_2 \triangleright m'}{m \vdash C_1 ; C_2 \triangleright m'}$$

$$\frac{m \vdash E \triangleright \mathit{true} \quad m \vdash C_1 \triangleright m'}{m \vdash \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 \triangleright m'} \qquad \frac{m \vdash E \triangleright \mathit{false} \quad m \vdash C_2 \triangleright m'}{m \vdash \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 \triangleright m'}$$

$$\frac{m \vdash E \triangleright \mathit{true} \quad m \vdash C \triangleright m_1 \quad m_1 \vdash \texttt{while } E \texttt{ do } C \triangleright m'}{m \vdash \texttt{while } E \texttt{ do } C \triangleright m'} \qquad \frac{m \vdash E \triangleright \mathit{false}}{m \vdash \texttt{while } E \texttt{ do } C \triangleright m}$$

FIG. 2.6 – Sémantique des commandes

2.2.3 Sémantique axiomatique

Il s'agit d'interpréter les commandes comme des transformations de prédicats, ces prédicats spécifiant des propriétés de la mémoire. Si P et Q sont deux tels prédicats et C une commande, alors on définit une relation $\{P\}C\{Q\}$ dont l'interprétation est : l'exécution de C à partir d'une mémoire vérifiant P conduit à une mémoire vérifiant Q.

Nous aurons besoin de transformateurs de prédicats particuliers. La conjonction et l'implication de deux prédicats seront notés par $P \wedge Q$ et $P \Rightarrow Q$, si x est une variable et E une expression alors $P[x \leftarrow E]$ représente le prédicat qui à toute mémoire m associe $P(m[x \leftarrow v])$ où v est la valeur telle que $m \vdash E \triangleright v$. Si w est une valeur alors $E = w$ représente le prédicat qui à tout m associe la propriété $v = w$ où v est la valeur telle que $m \vdash E \triangleright v$.

La définition de cette relation est donnée par les règles d'inférence de la figure 2.7.

$$\{P\}\texttt{skip}\{P\} \qquad \{P[X \leftarrow E]\}X\texttt{:=}E\{P\} \qquad \frac{\{P\}C_1\{P_1\} \quad \{P_1\}C_2\{Q\}}{\{P\}C_1;C_2\{Q\}}$$

$$\frac{\{P \wedge (E = \mathit{true})\}C_1\{Q\} \qquad \{P \wedge (E = \mathit{false})\}C_2\{Q\}}{\{P\}\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2\{Q\}}$$

$$\frac{\{P \wedge (E = \mathit{true})\}C\{P\}}{\{P\}\texttt{while } E \texttt{ do } C\{P \wedge (E = \mathit{false})\}}$$

$$\frac{P \Rightarrow P_1 \quad \{P_1\}C\{Q_1\} \quad Q_1 \Rightarrow Q}{\{P\}C\{Q\}}$$

FIG. 2.7 – Sémantique axiomatique des commandes

On remarque que toutes ces définitions font intervenir des notions définies dans le méta-langage telles que la mémoire, les domaines sémantiques, les opérations sur ces domaines,...

2.2.4 Quelques propriétés

Parmi les propriétés intéressantes à montrer on a:
- Décidabilité de la correction par rapport à la sémantique statique (algorithme de typage).
- Toute expression correctement typée admet une valeur.
- Correction de la sémantique axiomatique. Si $\{P\}C\{Q\}$ est vérifié alors toute évaluation de C dans une mémoire qui vérifie P conduit à une mémoire qui vérifie Q.

2.3 Spécification Gallina

Nous montrons comment représenter la théorie précédente en Gallina.

2.3.1 Les expressions

Définitions

On choisit de représenter les variables et les opérateurs binaires par des ensembles paramétriques qui pourront être instanciés dans un second temps. Par contre, les fonctions booléennes sont représentées de manière concrète par un constructeur du type de données. Ce choix permet d'illustrer deux traitements possibles des opérations du langage que l'on cherche à modéliser.

```
Coq < Parameter name : Set.

Coq < Inductive sort : Set :=
```

```
Coq < | Sbool : sort
Coq < | Snat : sort.

Coq < Parameter op : Set.

Coq < Inductive expr : Set :=
Coq < | Var : name -> sort -> expr
Coq < | Tr : expr
Coq < | Fa : expr
Coq < | Xor : expr -> expr -> expr
Coq < | Num : nat -> expr
Coq < | Null : expr -> expr
Coq < | Opn : op -> expr -> expr -> expr.
```

Expressions correctes

Le prédicat `exprcorrect` traduit la relation décrite dans la figure 2.3. Chaque règle d'inférence définissant la relation est traduite en un constructeur de la définition inductive. Les variables libres des définitions deviennent des variables quantifiées universellement dans les constructeurs.

```
Coq < Inductive exprcorrect : sort -> expr -> Prop :=
Coq < | corvar : forall (n:name) (s:sort), exprcorrect s (Var n s)
Coq < | cortr : exprcorrect Sbool Tr
Coq < | corfa : exprcorrect Sbool Fa
Coq < | corxor :
Coq <       forall b c:expr,
Coq <         exprcorrect Sbool b ->
Coq <         exprcorrect Sbool c -> exprcorrect Sbool (Xor b c)
Coq < | cornum : forall n:nat, exprcorrect Snat (Num n)
Coq < | cornull :
Coq <       forall e:expr, exprcorrect Snat e -> exprcorrect Sbool (Null e)
Coq < | corop :
Coq <       forall (o:op) (e f:expr),
Coq <         exprcorrect Snat e ->
Coq <         exprcorrect Snat f -> exprcorrect Snat (Opn o e f).
```

Domaines sémantiques

Il s'agit de représenter le domaine sémantique des variables, à chaque sorte de variable correspond un ensemble au sens mathématique qui sera représenté par un type de données `Coq` ici le type des booléens ou des entiers. Le domaine sémantique des valeurs `semval` est représenté par la somme disjointe des deux types.

```
Coq < Inductive semval : Set :=
Coq < | Bool : bool -> semval
Coq < | Nat : nat -> semval.
```

Interprétation des fonctions

Les fonctions sémantiques correspondant aux opérateurs concrets peuvent être explicitement programmées. Par contre la sémantique des opérations arithmétiques (qui sont vues de manière paramétrique) doit être fournie comme un paramètre.

```
Coq < Definition boolxor (b1 b2:bool) : bool :=
Coq <   if b1 then if b2 then false else true else b2.

Coq < Definition iszero (n:nat) : bool :=
Coq <   match n with
Coq <   | O => true
Coq <   | S n => false
Coq <   end.

Coq < Parameter semop : op -> nat -> nat -> nat.
```

Compatibilité entre sortes et valeurs

Il nous faudra relier la sorte des expressions et le type de leur interprétation sémantique. Pour cela, on définit `sort_val` une fonction des domaines sémantiques dans les sortes qui à chaque valeur sémantique associe la sorte correspondante.

```
Coq < Definition sort_val (v:semval) : sort :=
Coq <   match v with
Coq <   | Bool b => Sbool
Coq <   | Nat n => Snat
Coq <   end.
```

On utilisera également une relation `compat` entre les valeurs et les sortes telle que (`compat Sbool` v) soit équivalent à $\exists b:$ `bool`$.v =$ (`Bool` b) (défini par le prédicat `valbool`) et (`compat Snat` v) soit équivalent à $\exists n:$ `nat`$.v =$ (`Nat` n) (défini par le prédicat `valnat`).

```
Coq < Inductive valbool : semval -> Prop :=
Coq <     valbool_intro : forall b:bool, valbool (Bool b).

Coq < Inductive valnat : semval -> Prop :=
Coq <     valnat_intro : forall n:nat, valnat (Nat n).

Coq < Definition compat (s:sort) (v:semval) : Prop :=
Coq <   match s with
Coq <   | Sbool => valbool v
Coq <   | Snat => valnat v
Coq <   end.
```

On remarque que le définition inductive de `valbool` est un codage inductif direct de la proposition `fun` x`:semval => exists` b`:bool,` x`=Bool` b. On peut aisément montrer la correspondance entre `compat` et `sort_val`.

```
Coq < Theorem compat_sort_val :
Coq <  forall (s:sort) (v:semval), compat s v -> s = sort_val v.
```

Une alternative est de représenter la notion de compatibilité par un prédicat inductif :

```
Coq < Inductive compat1 : sort -> semval -> Prop :=
Coq < | compat_bool : forall b:bool, compat1 Sbool (Bool b)
Coq < | compat_nat : forall n:nat, compat1 Snat (Nat n).
```

Les deux définitions sont équivalentes, remarquons que dans le second cas la propriété

$$(\texttt{compat1 Snat } v) \rightarrow \exists n : \texttt{nat}. v = (\texttt{Nat n})$$

nécessite une inversion du prédicat inductif alors que dans le cas de `compat` c'est une simple conséquence du calcul de l'expression `Cases` et de la définition de `valnat`.

La mémoire

La mémoire est représentée comme une fonction qui à toute variable et sorte associe une valeur sémantique, il faudra de plus supposer que cette valeur est compatible avec la sorte.

```
Coq < Definition memory := name -> sort -> semval.
```

Valeur d'une expression

Pour construire la relation entre une expression et sa valeur, on peut se donner une mémoire comme paramètre dans une "Section", lorsque la section est fermée, les notions introduites seront automatiquement abstraites par rapport aux paramètres dont elles dépendent.

```
Coq < Section Valexpr.

Coq < Variable memstate : memory.

Coq < Hypothesis memstate_correct :
Coq <     forall (n:name) (s:sort), compat s (memstate n s).
```

La définition `exprval` formalise la sémantique des expressions telle qu'elle est présentée à la figure 2.5.

```
Coq < Inductive exprval : expr -> semval -> Prop :=
Coq < | valvar : forall (n:name) (s:sort), exprval (Var n s) (memstate n s)
Coq < | valtr : exprval Tr (Bool true)
Coq < | valfa : exprval Fa (Bool false)
Coq < | valxor :
Coq <       forall (f g:expr) (bf bg:bool),
Coq <         exprval f (Bool bf) ->
Coq <         exprval g (Bool bg) -> exprval (Xor f g) (Bool (boolxor bf bg))
Coq < | valnum : forall n:nat, exprval (Num n) (Nat n)
Coq < | valtzero :
Coq <       forall (f:expr) (nf:nat),
Coq <         exprval f (Nat nf) -> exprval (Null f) (Bool (iszero nf))
Coq < | valopn :
Coq <       forall (o:op) (f g:expr) (nf ng:nat),
Coq <         exprval f (Nat nf) ->
Coq <         exprval g (Nat ng) -> exprval (Opn o f g) (Nat (semop o nf ng)).
```

On peut maintenant énoncer le théorème qui dit que toute expression correctement typée admet une valeur.

```
Coq < Theorem expr_val_cor : forall (E:expr) (s:sort),
Coq <    exprcorrect s E ->  exists v : semval, exprval E v.
```

La preuve par récurrence nécessite un lemme plus fort qui dit que la valeur calculée est compatible avec la sorte de l'expression.

```
Coq < Lemma expr_val_cor_dom : forall (E:expr) (s:sort),
Coq <    exprcorrect s E ->  exists2 v : semval, compat s v & exprval E v.
```

Représentation de la mémoire à l'aide de types dépendants

Le traitement de la relation entre la sorte de l'expression et le domaine de la valeur sémantique est lourd. On peut profiter de l'expressivité du langage de spécification de `Coq` pour utiliser d'autres formalisations. On construit une famille dépendante `sval` (dont le type est `sort` $\rightarrow$ `Set`) qui à la sorte `Sbool` associe le type `bool` et à `Snat` associe le type `nat`. Le domaine sémantique `semval` pourrait être représenté par un couple formé d'une sorte s et d'un objet de type (`sval` s), ceci revient juste à coder une somme disjointe explicitement à l'aide d'un sélecteur s et d'un champ dont le type varie suivant le sélecteur.

On peut tirer partie de cette représentation pour simplifier la formalisation en rendant implicite dans la définition de la mémoire la notion de compatibilité.

```
Coq < Definition sval (s:sort) : Set :=
Coq <   match s with
Coq <   | Sbool => bool
Coq <   | Snat => nat
Coq <   end.

Coq < Definition memory1 := name -> forall s:sort, sval s.
```

On définit alors:

```
Coq < Parameter memstate1 : memory1.

Coq < Inductive exprval1 : expr -> forall s:sort, sval s -> Prop :=
Coq < | valvar1 :
Coq <       forall (n:name) (s:sort), exprval1 (Var n s) s (memstate1 n s)
Coq < | valtr1 : exprval1 Tr Sbool true
Coq < | valfa1 : exprval1 Fa Sbool false
Coq < | valxor1 :
Coq <       forall (f g:expr) (bf bg:bool),
Coq <         exprval1 f Sbool bf ->
Coq <         exprval1 g Sbool bg -> exprval1 (Xor f g) Sbool (boolxor bf bg)
Coq < | valnum1 : forall n:nat, exprval1 (Num n) Snat n
Coq < | valtzero1 :
Coq <       forall (f:expr) (nf:nat),
Coq <         exprval1 f Snat nf -> exprval1 (Null f) Sbool (iszero nf)
Coq < | valopn1 :
Coq <       forall (o:op) (f g:expr) (nf ng:nat),
Coq <         exprval1 f Snat nf ->
Coq <         exprval1 g Snat ng -> exprval1 (Opn o f g) Snat (semop o nf ng).
```

Le lemme de correction de l'évaluation s'énonce alors simplement :

```
Coq < Theorem expr_val_cor1 : forall (E:expr) (s:sort),
Coq <    exprcorrect s E ->  exists v : sval s, exprval1 E s v.
```

La formalisation et la preuve sont plus simples cependant l'usage de types dépendants rendra la formalisation de la fonction d'écriture sur la mémoire plus complexe en effet on doit avoir avec `n:name,s:sort,v:(sval s),mem:memory1`

```
Coq < Definition update : forall (m:nat) (s':sort), sval s'.
```

Le simple fait de savoir que $s = s'$ ne suffit pas à assurer que le terme v de type (`sval` s) est aussi de type (`sval` s'). Il faudra utiliser une analyse par cas suivant les valeurs de s et s'. Une telle analyse n'aurait pas été possible si l'ensemble des sortes était resté paramétrique.

De manière générale, en présence de types dépendants, l'usage de l'égalité devient délicat. Il n'est pas possible par exemple d'écrire $v = v'$ avec v : (`sval` s) et v' : (`sval` s') sauf lorsque s et s' sont convertibles, la présence d'une hypothèse $s = s'$ est insuffisante. Il faudra utiliser des notions plus complexes d'égalité.

2.3.2 Vérification du type et évaluation constructive

On pourrait définir le fait qu'une expression e est mal formée comme la propriété :

$$\forall s : \texttt{sort}.\neg(\texttt{exprcorrect}\ s\ e)$$

En fait il est plus aisé de manipuler des définitions positives et donc de définir une notion constructive d'expression erronée en énumérant les cas d'échec possibles. Si on s'intéresse à l'interprétation constructive des preuves (que nous verrons plus tard) on remarque qu'une preuve que l'expression erronée permet de retrouver exactement la nature de l'erreur à savoir dans quel sous terme un opérateur a été appliqué à une expression d'une sorte non adaptée.

```
Coq < Inductive exprerr : expr -> Prop :=
Coq < | errxorl : forall b c:expr, exprcorrect Snat b -> exprerr (Xor b c)
Coq < | errxorr : forall b c:expr, exprcorrect Snat c -> exprerr (Xor b c)
Coq < | errtzero : forall b:expr, exprcorrect Sbool b -> exprerr (Null b)
Coq < | erropl :
Coq <       forall (op:op) (b c:expr),
Coq <         exprcorrect Sbool b -> exprerr (Opn op b c)
Coq < | erropr :
Coq <       forall (op:op) (b c:expr),
Coq <         exprcorrect Sbool c -> exprerr (Opn op b c)
Coq < | errcongxorl : forall b c:expr, exprerr b -> exprerr (Xor b c)
Coq < | errcongxorr : forall b c:expr, exprerr c -> exprerr (Xor b c)
Coq < | errcongtzero : forall b:expr, exprerr b -> exprerr (Null b)
Coq < | errcongopl :
Coq <       forall (op:op) (b c:expr), exprerr b -> exprerr (Opn op b c)
Coq < | errcongopr :
Coq <       forall (op:op) (b c:expr), exprerr c -> exprerr (Opn op b c).
```

Le théorème exprimant la décidabilité du typage et de l'évaluation est juste une preuve du fait que pour toute expression E, soit il est possible de construire une valeur v de l'expression dont la sorte est la sorte de l'expression soit il est possible de prouver que l'expression E est mal formée. Le fait d'utiliser une interprétation constructive de la disjonction et de l'existentielle assure l'existence d'un algorithme permettant de décider dans quel cas on est et de calculer effectivement la valeur. On établit un résultat de décidabilité sans avoir à représenter une notion de calculabilité.

```
Coq < Theorem expr_val_check_proof : forall E:expr,
Coq <    {v : semval | exprval E v &  exprcorrect (sort_val v) E} + {exprerr E}.
```

Si on ne s'intéresse qu'à la partie calcul de cette preuve alors on a une fonction `eval_prog` qui à toute expression associe une valeur ou rien du tout. Cette fonction peut également se représenter dans Gallina en utilisant le type `option` de Coq.

```
Coq < Print option.
Inductive option (A : Set) : Set :=
    Some : A -> option A | None : option A
For Some: Argument A is implicit
For None: Argument A is implicit
For option: Argument scope is [type_scope]
For Some: Argument scopes are [type_scope _]
For None: Argument scope is [type_scope]

Coq < Implicit Arguments Some [A].
```

La fonction `eval_prog` se calcule par point fixe structurel sur l'expression.

```
Coq < Fixpoint eval_prog (e:expr) : option semval :=
Coq <   match e with
Coq <   | Var n s => Some (memstate n s)
Coq <   | Tr => Some (Bool true)
Coq <   | Fa => Some (Bool false)
Coq <   | Xor f g =>
Coq <       match eval_prog f, eval_prog g with
Coq <       | Some (Bool bf), Some (Bool bg) => Some (Bool (boolxor bf bg))
Coq <       | _, _ => None (A:=semval)
Coq <       end
Coq <   | Num n => Some (Nat n)
Coq <   | Null f =>
Coq <       match eval_prog f with
Coq <       | Some (Nat nf) => Some (Bool (iszero nf))
Coq <       | _ => None (A:=semval)
Coq <       end
Coq <   | Opn o f g =>
Coq <       match eval_prog f, eval_prog g with
Coq <       | Some (Nat nf), Some (Nat ng) => Some (Nat (semop o nf ng))
Coq <       | _, _ => None (A:=semval)
Coq <       end
Coq <   end.
```

La section `Valexpr`, ouverte page 27 est alors fermée ce qui a pour effet d'abstraire les définitions effectuées dans la section par rapport à `memstate` et à l'hypothèse `memstate_correct` lorsqu'elle est utilisée.

```
Coq < End Valexpr.

Coq < Check eval_prog.
eval_prog
     : memory -> expr -> option semval
```

2.3.3 Les commandes

La représentation de la syntaxe et de la sémantique statique des commandes suit les définitions des figures 2.1 et 2.4.

Syntaxe

```
Coq < Inductive com : Set :=
Coq <   | skip : com
Coq <   | aff : name -> expr -> com
Coq <   | seq : com -> com -> com
Coq <   | cond : expr -> com -> com -> com
Coq <   | while : expr -> com -> com.
```

Sémantique statique

```
Coq < Inductive comcorrect : com -> Prop :=
Coq < | corskip : comcorrect skip
Coq < | coraff :
Coq <       forall (n:name) (e:expr) (s:sort),
Coq <         exprcorrect s e -> comcorrect (aff n e)
Coq < | corseq :
Coq <       forall c d:com,
Coq <         comcorrect c -> comcorrect d -> comcorrect (seq c d)
Coq < | corcond :
Coq <       forall (b:expr) (c d:com),
Coq <         exprcorrect Sbool b ->
Coq <         comcorrect c -> comcorrect d -> comcorrect (cond b c d)
Coq < | corwhile :
Coq <       forall (b:expr) (c:com),
Coq <         exprcorrect Sbool b -> comcorrect c -> comcorrect (while b c).
```

2.3.4 Mise à jour de la mémoire

Nous définissons maintenant la fonction d'écriture dans la mémoire. Elle utilise la décidabilité de l'égalité sur les variables, qui vient de la décidabilité de l'égalité sur les noms (prise comme axiome puisque l'ensemble des noms n'est pas spécifié) et de celle sur les sortes qui peut explicitement être construite.
Pour montrer la décidabilité de l'égalité sur un ensemble A, on peut construire une fonction booléenne f de type $A \to A \to \mathsf{bool}$ et démontrer que c'est la fonction caractéristique de l'égalité $(f\ x\ y) = \mathsf{true} \Leftrightarrow x = y$. On choisit une autre solution qui est de construire un terme `fdec` de type $\forall x, y : A, \{x = y\} + \{\neg(x = y)\}$ qui à tous x et y associe soit un objet (`left` h) avec h une preuve de $x = y$ soit un objet (`right` h) avec h une preuve de $\neg(x = y)$.
Une expression "`if` $a = b$ `then` p `else` q" s'écrira dans Coq:

```
match (fdec a b) with left _ => p | right _ => q end.
```

Ou de manière équivalente :

```
ifdec (fdec a b) p q
```

On oublie l'information de preuve pour construire l'expression mais celle-ci pourra être facilement utilisée lorsqu'il s'agira de montrer des propriétés de cette expression.

Nous commençons par poser en axiome la décidabilité de l'égalité sur l'ensemble des noms et nous prouvons la décidabilité de l'égalité sur l'ensemble des sortes.

```
Coq < Parameter eq_name_dec : forall n m:name, {n = m} + {n <> m}.

Coq < Lemma eq_sort_dec : forall s s':sort, {s = s'} + {s <> s'}.

Coq < decide equality.

Coq < Defined.
```

Nous pouvons définir maintenant l'opération `update` de mise à jour de la mémoire qui se fait dans une section introduisant le nom de la variable x, la valeur à affecter à cette variable v et la mémoire m. La sorte de la valeur est localement nommée s.

```
Coq < Section Update.

Coq < Variable x : name.

Coq < Variable v : semval.

Coq < Variable mem : memory.

Coq < Let s := sort_val v.

Coq < Definition update : memory :=
Coq <   fun (m:name) (s':sort) =>
Coq <     ifdec (eq_sort_dec s s') (ifdec (eq_name_dec x m) v (mem m s'))
Coq <       (mem m s').
```

On montre ensuite les propriétés de base de cette fonction :

```
Coq < Theorem update_eq : v = update x s.

Coq < Theorem update_diff_name :
Coq <  forall (m:name) (s':sort), x <> m -> mem m s' = update m s'.

Coq < Theorem update_diff_sort :
Coq <  forall (m:name) (s':sort), s <> s' -> mem m s' = update m s'.
```

Après avoir prouvé les lemmes, la section peut être refermée, les constructions sont alors paramétrées par x, v et m.

```
Coq < End Update.
```

2.3.5 Sémantique opérationnelle

La déclaration suivante implémente la relation décrivant la sémantique opérationnelle du langage de commande telle qu'elle est décrite dans la figure 2.6.

```
Coq < Inductive semcom : memory -> com -> memory -> Prop :=
Coq < | semskip : forall m:memory, semcom m skip m
Coq < | semaff :
```

```
Coq <       forall (m:memory) (n:name) (v:semval) (e:expr),
Coq <         exprval m e v -> semcom m (aff n e) (update n v m)
Coq < | semseq :
Coq <       forall (m m1 m':memory) (c d:com),
Coq <         semcom m c m1 -> semcom m1 d m' -> semcom m (seq c d) m'
Coq < | semcondtr :
Coq <       forall (m m':memory) (e:expr) (c d:com),
Coq <         exprval m e (Bool true) ->
Coq <         semcom m c m' -> semcom m (cond e c d) m'
Coq < | semcondfa :
Coq <       forall (m m':memory) (e:expr) (c d:com),
Coq <         exprval m e (Bool false) ->
Coq <         semcom m d m' -> semcom m (cond e c d) m'
Coq < | semwhiletr :
Coq <       forall (m m':memory) (e:expr) (c:com),
Coq <         exprval m e (Bool true) ->
Coq <         forall m1:memory,
Coq <           semcom m c m1 ->
Coq <           semcom m1 (while e c) m' -> semcom m (while e c) m'
Coq < | semwhilefa :
Coq <       forall (m:memory) (e:expr) (c:com),
Coq <         exprval m e (Bool false) -> semcom m (while e c) m.
```

2.3.6 Sémantique axiomatique

Nous nous intéressons finalement à la sémantique axiomatique. Une assertion est une propriété de la mémoire, représentée par un prédicat unaire. En logique de Hoare, ce prédicat unaire sera défini concrètement par une formule logique utilisant les variables du programme pour représenter les valeurs correspondantes de la mémoire.

```
Coq < Definition Assertion := memory -> Prop.
```

Transformations de prédicats

On étend les opérations usuelles de la logique à des transformations d'assertion:

```
Coq < Definition Istrue (E:expr) : Assertion :=
Coq <   fun m:memory => exprval m E (Bool true).

Coq < Definition Isfalse (E:expr) : Assertion :=
Coq <   fun m:memory => exprval m E (Bool false).

Coq < Inductive AndAss (P Q:Assertion) (m:memory) : Prop :=
Coq <     Conjass : P m -> Q m -> AndAss P Q m.

Coq < Definition ImplAss (P Q:Assertion) : Prop := forall m:memory, P m -> Q m.
```

Le transformateur suivant correspond à ce que nous avons noté $P[X \leftarrow E]$. Le terme (`memupdate` x E P) représente le prédicat qui est vrai en m si $P(m[X \leftarrow v])$ est vérifié avec v la valeur de l'expression E dans la mémoire m.

```
Coq < Definition memupdate (x:name) (e:expr) (P:Assertion) : Assertion :=
Coq <   fun m:memory => forall v:semval, exprval m e v -> P (update x v m).
```

Définition de $\{P\}c\{Q\}$

On définit le prédicat (`trueform P c Q`) correspondant à $\{P\}C\{Q\}$ telle qu'il est décrit dans la figure 2.7.

```
Coq < Inductive trueform : Assertion -> com -> Assertion -> Prop :=
Coq < | trueskip : forall P:Assertion, trueform P skip P
Coq < | trueaff :
Coq <       forall (P:Assertion) (n:name) (e:expr),
Coq <         trueform (memupdate n e P) (aff n e) P
Coq < | trueseq :
Coq <       forall (P Q R:Assertion) (c d:com),
Coq <         trueform P c R -> trueform R d Q -> trueform P (seq c d) Q
Coq < | truecond :
Coq <       forall (P Q:Assertion) (e:expr) (c d:com),
Coq <         trueform (AndAss P (Istrue e)) c Q ->
Coq <         trueform (AndAss P (Isfalse e)) d Q -> trueform P (cond e c d) Q
Coq < | truewhile :
Coq <       forall (P:Assertion) (e:expr) (c:com),
Coq <         trueform (AndAss P (Istrue e)) c P ->
Coq <         trueform P (while e c) (AndAss P (Isfalse e))
Coq < | truecons :
Coq <       forall (P P1 Q Q1:Assertion) (c:com),
Coq <         ImplAss P P1 ->
Coq <         trueform P1 c Q1 -> ImplAss Q1 Q -> trueform P c Q.
```

Lemme de correction

Le théorème suivant énonce la correction de la relation donnée $\{P\}c\{Q\}$ par rapport à la sémantique.

```
Coq < Theorem truecorrect : forall (P Q:Assertion) (c:com),
Coq <    trueform P c Q -> forall m1 m2:memory, semcom m1 c m2 -> P m1 -> Q m2.
```

2.4 Pour en savoir plus

2.4.1 Sémantique des langages et compilateurs

Les logiques d'ordre supérieur comme le Calcul des Constructions Inductives se prêtent bien aux formalisations de notions sémantiques et logiques. Une des premières preuves de cette nature a été effectuée par Samuel Boutin. Il s'agissait du schéma de compilation d'un mini-ml vers la CAM (Categorical Abstract Machine) tel qu'il était décrit dans l'article [CDDK86]. Yves Bertot a étudié la preuve du compilateur d'un langage impératif vers un langage assembleur [Ber98].

Des preuves de compilateurs plus conséquents ont ensuite été entreprises: Delphine Terrasse [Ter92] a ébauché la preuve d'un compilateur Esterel, Pablo Argon [Arg98] a extrait le noyau du compilateur du langage Electre exprimé comme l'exécution des règles de la sémantique, des équipes de recherche de Dassault et Aérospatiale ont étudié la formalisation d'un compilateur pour le langage Lustre tel qu'il est implanté dans l'outil Scade, une partie de ces développements est disponible domme contribution au système Coq. On trouvera également dans les contributions des formalisations de plusieurs calculs de processus, en particulier le π-calcul ainsi que des modélisations de logique temporelle.

2.4.2 Logique de Hoare

La formalisation de la logique de Hoare a été effectuée par Thomas Kleymann-Schreiber [Sch97, Kle98] dans l'assistant LEGO dont le langage s'apparente au Calcul des Constructions Inductives. Une formalisation dans l'outil HOL a également été réalisée par Peter Homeier, l'objectif principal de ces développements est de justifier les propriétés fondamentales de la sémantique axiomatique comme la correction et la complétude, ce qui n'est pas évident dès que le langage comporte des constructions avancées comme des appels de procédure.

2.4.3 Preuve de programmes Java

Un domaine actif de recherche est actuellement l'étude des propriétés de sécurité des programmes Java, qu'ils soient utilisés sur l'internet ou les cartes à puce. Pour garantir de telles propriétés, il est essentiel d'avoir une description précise de la sémantique de ce langage, au niveau du code source ou du byte-code, que ce soit la sémantique statique, dynamique ou axiomatique. Le projet Bali `http://isabelle.in.tum.de/Bali`http://isabelle.in.tum.de/Bali à l'université de Munich formalise ces notions dans l'outil Isabelle/HOL. Des développements analogues sont réalisés à l'aide de Coq en France, dans le projet Lemme à l'INRIA Sophia-Antipolis, dans le projet Lande à l'IRISA ou par la société Trusted Logic. Les définitions inductives sont utilisées de manière intensive dans ces développements.

2.4.4 Plongement superficiel ou profond

Lorsqu'on veut étudier les programmes d'un langage X, on peut procéder de deux manières. La première, appelée plongement profond (deep embedding en anglais), consiste à introduire un type concret dans Coq pour représenter les arbres de syntaxe abstraite du langage X. La seconde appelée plongement superficiel (shallow embedding en anglais), consiste à représenter directement un programme de X par sa sémantique exprimée dans le langage mathématique du système Coq.

Plongement profond Dans le plongement profond, on dispose d'un type concret Y qui représente les arbres de syntaxe abstraite du langage X. On peut ensuite construire des fonctions et des relations sur ce type. On pourra ainsi parler des expressions bien formées, construire des algorithmes de typage, représenter la sémantique du langage. Un programme écrit dans le langage X pourra être représenté par un objet Coq de type Y, les propriétés de cet objet seront établies en utilisant différentes sémantiques.

Dans notre exemple, les programmes sont représentés par un plongement profond.

Plongement superficiel Dans le plongement superficiel, un programme est directement traduit en sa sémantique. Par exemple, on pourrait commencer par introduire une notion de mémoire et identifier un programme à une fonction de transformation des mémoires. La séquence de deux programmes pourrait être définie comme la composition des mémoires les représentant. Les relations sur les programmes comme les sémantiques opérationnelle ou axiomatique peuvent être définies en faisant référence à la sémantique des programmes. Les règles d'inférence deviennent alors des théorèmes qui peuvent être utilisés pour prouver des propriétés de programmes. La représentation des propriétés dans la sémantique axiomatique utilise un plongement superficiel (on n'introduit pas la syntaxe des formules).

Cette représentation peut permettre de raisonner rapidement sur les propriétés de programmes particuliers. Par contre, elle ne permet pas de construire des programmes ou de faire des preuves par récurrence sur la structure des programmes. Elle n'est donc pas adaptée à l'étude méta-théorique des propriétés du langage.

Chapitre 3

Types inductifs

Ce chapitre décrit la théorie des types inductifs. Dans une première partie, nous nous intéressons aux définitions non récursives puis nous aborderons les définitions récursives. Dans ce document, pour simplifier les notations, les symboles $\forall$ et λ seront utilisés à la place des mots-clé de Coq `forall` et `fun`.

3.1 Généralités

3.1.1 Forme générale

Une déclaration inductive dans Coq a la forme suivante :

$$\begin{array}{l} \mathsf{Inductive}\ \mathtt{nom}\ (z_1 : P_1) \dots (z_k : P_k) : \forall (a_1 : A_1) \dots (a_l : A_l), s \\ := \mathtt{co}_1 : \forall (x^1 : C_1^1) \dots (x^{n_1} : C_1^{n_1}), (\mathtt{nom}\ z_1\ \dots\ z_k\ t_1^1 \dots t_1^l) \\ \quad \vdots \\ \mid \mathtt{co}_p : \forall (x^1 : C_p^1) \dots (x^{n_p} : C_p^{n_p}), (\mathtt{nom}\ z_1\ \dots\ z_k\ t_p^1 \dots t_p^l). \end{array}$$

Vocabulaire On introduit les notations suivantes :
- On appelle *paramètres* les variables : $z_1 : P_1; \dots; z_k : P_k$
- On notera $A \equiv \forall (a_1 : A_1) \dots (a_l : A_l), s$, ce type est appelé *l'arité* de la définition inductive et s est la *sorte* de la définition inductive
- $\mathtt{co}_1, \dots, \mathtt{co}_p$ sont les noms des p constructeurs de la déclaration, on peut avoir $p = 0$. On notera $C_m \equiv \forall (x^1 : C_m^1) \dots (x^{n_m} : C_m^{n_m}), (\mathtt{nom}\ z_1\ \dots\ z_k\ t_m^1 \dots t_m^l)$ Ce terme est appelé le type du m-ème constructeur de la déclaration `nom`. On appellera *type d'argument de constructeur* les types C_m^n. On appelle *arité du constructeur* le nombre n_m d'arguments du constructeur.

Déclaration récursive On dira que la déclaration est *récursive* si `nom` apparaît dans l'un des types d'argument de constructeur C_m^n. On dira qu'elle est *non-récursive* sinon.

3.1.2 Forme abstraite

Une déclaration inductive dans Coq introduit de nouveaux noms. Sur le plan théorique, il est parfois plus commode d'avoir une représentation abstraite des définitions inductives.

On ne garde alors que les éléments essentiels : l'arité A de la déclaration inductive et les types des constructeurs C_m. Dans le cas où il n'y a pas de paramètre, une notation possible pour la définition inductive est :

$$\mathsf{Ind}(\mathtt{nom} : A)\{C_1; \dots; C_p\}$$

et pour le k-ème constructeur :

$$\mathsf{Constr}(k, \mathsf{Ind}(\texttt{nom} : A)\{C_1; \dots; C_p\})$$

`nom` peut être vu comme un lieur dans la déclaration de l'inductif ce qui permet d'identifier des déclarations inductives isomorphes (même arité, même nombre de constructeurs avec les mêmes types d'argument).

Dans cette approche, si toutes les occurrences de `nom` dans les types d'arguments sont appliquées aux paramètres $z_1, \dots, z_p$ alors les paramètres peuvent être vus comme des abstractions. On construit de nouveaux types de constructeur C'_k en remplaçant dans chaque type de constructeur C_k le terme ($\texttt{nom}\ z_1 \dots z_p$) par le nouvel identificateur `nom'` et on retrouve la définition inductive générale à l'aide des définitions suivantes.

$$\begin{array}{ll} \texttt{nom} & := \lambda(z_1 : P_1) \dots (z_k : P_k) \Rightarrow \mathsf{Ind}(\texttt{nom'} : A)\{C'_1; \dots; C'_p\} \\ \texttt{co}_k & := \lambda(z_1 : P_1) \dots (z_k : P_k) \Rightarrow \mathsf{Constr}(k, \mathsf{Ind}(\texttt{nom'} : A)\{C'_1; \dots; C'_p\}) \end{array}$$

Une déclaration inductive avec paramètres peut se voir comme une famille de définitions inductives.

3.2 Les déclarations non-récursives

Beaucoup des difficultés concernant les définitions inductives apparaissent déjà au niveau des définitions non-récursives que nous étudions en premier. Dans un premier temps, nous ne précisons pas la sorte de déclaration de la définition inductive. Nous considérons également des définitions inductives sans paramètres car ceux-ci ne jouent pas de rôle essentiel.

3.2.1 Les déclarations de base

Les définitions de base non récursives sont :

- La déclaration vide `Zero` (arité s, pas de constructeur)
- La déclaration unitaire `Un` (arité s, un constructeur de type `Un`)
- Les types sommes (aussi appelés Σ-type) $\Sigma x : A.B$ (arité s, un constructeur de type $\forall(x : A), B \to \Sigma x : A.B$)
- Les sommes disjointes $A{+}B$: (arité s, deux constructeurs de type $A \to A{+}B$ et $B \to A{+}B$)
- Égalité $x =_A y$: (un paramètre $x : A$, arité $A \to s$, un constructeur de type $x =_A x$)

Si on suppose que l'on a un Σ-type : $\Sigma x : A.B$ avec deux projections : $\pi_1 : \Sigma x : A.B \to A$ et $\pi_2 : (p : \Sigma x : A.B)B\{x := (\pi_1\ p)\}$ alors il est aisé de définir une somme n-aire :

$$\Sigma(x_1 : A_1; \dots; x_n : A_n)$$

avec un constructeur de type $\forall(x_1 : A_1) \dots (x_n : A_n), \Sigma(x_1 : A_1; \dots; x_n : A_n)$ et n projections π_k : $\forall p : \Sigma(x_1 : A_1; \dots; x_n : A_n), (A_k\{x_1, \dots, x_{k-1} := (\pi_1\ p), \dots, (\pi_{k-1}\ p)\}$. On notera $(a_1, \dots, a_n)$ l'élément de $\Sigma(x_1 : A_1; \dots; x_n : A_n)$ défini à l'aide du constructeur. Cette somme se définit par récurrence sur n. On pose $\Sigma() \equiv \texttt{Un}$, $\Sigma(x : A) \equiv A$, et $\Sigma(x : A; x_1 : A_1; \dots; x_n : A_n) \equiv \Sigma x : A.\Sigma(x_1 : A_1; \dots; x_n : A_n)$.

De même on peut définir une disjonction n-aire $A_1 + \cdots + A_n$ comme étant `Zero` si $n = 0$, A_1 si $n = 1$ et dans le cas $n > 1$ $A + A_1 + \cdots A_n \equiv A + (A_1 + \cdots A_n)$.

À partir de ces constructions de base, on peut trouver un équivalent à toute définition inductive non récursive. En reprenant les notations données en 3.1.1 et en introduisant la notation ΣA pour $\Sigma(a_1 : A_1; \dots; a_n : A_n)$ avec $A \equiv \forall(a_1 : A_1) \dots (a_n : A_n)$, s l'arité de la définition inductive :

$$\begin{array}{l} \texttt{nom} := \lambda(a1 : A_1) \dots (a_l : A_l) \Rightarrow \\ \quad \Sigma(x^1 : C_1^1; \dots; x^{n_1} : C_1^{n_1}; (a_1, \dots, a_l) =_{\Sigma A} (t_1^1, \dots, t_1^l)) \\ + \dots + \\ \quad \Sigma(x^1 : C_p^1; \dots; x^{n_p} : C_p^{n_p}; (a_1, \dots, a_l) =_{\Sigma A} (t_p^1, \dots, t_p^l)) \end{array}$$

Exemple 1 *Le type des booléens est équivalent à* $\texttt{Un} + \texttt{Un}$.

Opérateurs primitifs, vs schéma général On peut choisir d'introduire dans un formalisme les constructions de base (c'est le cas d'un système comme NuPrl ou dans des présentations catégoriques) ou bien introduire un schéma général de définition inductive. Le second choix permet d'utiliser l'uniformité des principes d'introduction et d'élimination des différents opérateurs. Il permet également de représenter de manière concise des propriétés qui nécessiteraient une imbrication profonde de connecteurs. L'introduction ou l'élimination de ces propriétés peuvent alors se faire en une seule étape ce qui permet d'avoir des preuves plus directes. Par contre, la généralité du schéma complique les raisonnements par cas sur la forme des définitions inductives : on ne peut pas juste traiter les cas `Zero`, `Un`, Σ-type, somme disjointe et égalité, il faut raisonner sur la structure interne des définitions inductives et traiter de manière générale des suites finies de termes ou de types; cela introduit une lourdeur à la fois au niveau de la programmation et de la présentation théorique.

Exemple 2 *On suppose donné un type* $U : \texttt{Set}$ *et un prédicat* $P : U \to \texttt{Prop}$. *On se propose de définir une relation* `dec` *portant sur un booléen* b *tel que* $(\texttt{dec}\ b)$ *soit vérifié si* $b = \texttt{true}$ *et* $\forall x : U, (P\ x)$ *est prouvable ou bien si* $b = \texttt{false}$ *et* $\exists x : U, \neg(P\ x)$ *est prouvable.*

On peut bien sûr introduire `dec` *comme :*

$$\begin{array}{l}\lambda(U : \texttt{Set})(P : U \to \texttt{Prop})(b : \texttt{bool}) \Rightarrow \\ ((\forall x : U, (P\ x)) \wedge b = \texttt{true}) + ((\exists x : U, \neg(P\ x)) \wedge b = \texttt{false})\end{array}$$

Mais on peut également introduire une définition inductive avec U *et* P *comme paramètres.*

```
Inductive dec (U:Set) (P:U→Prop) : bool → Prop
    isTrue : (∀ x:U,(P x)) → (dec U P true)
 |  isFalse : ∀ x:U,~(P x) → (dec U P false).
```

3.2.2 Règles de formation et d'introduction

Une déclaration inductive va introduire de nouveaux objets dans la théorie. Dans un cadre de déduction naturelle, on va trouver des règles d'introduction (une par constructeur) et des règles d'élimination. Il faut ajouter une règle de bonne formation pour la définition elle-même.

Règle de formation

En reprenant les notations de 3.1.1, la règle de formation donne le typage de la définition inductive. La condition est que chaque type de constructeur soit bien formé dans un environnement comportant les paramètres :

$$\frac{\Gamma, \texttt{nom} : A \vdash C_m : s \quad (m \in 1..p) \qquad s \text{ sorte de l'arité } A}{\Gamma \vdash \texttt{nom} : A}$$

Cette règle impose un lien entre la sorte de l'inductif et celles des types d'argument de constructeur dans le cas où la sorte s est prédicative.

Exemple 3 *On peut introduire* $\Sigma X : \texttt{Prop}, X$ *de type* `Prop` *qui représente la propriété* $\exists X.X$ *(quantification existentielle du second ordre). Par contre* $\Sigma X : \texttt{Type}_i, X$ *représente le type des types non vides et sera bien typé de type* $\texttt{Type}_{i+1}$. *Finalement dans un système avec* `Set` *prédicatif, le type* $\Sigma X : \texttt{Set}, X$ *est forcément de type* `Type` *alors qu'il peut être de type* `Set`, *si cette sorte est imprédicative.*

Règle d'introduction

Il y a une règle d'introduction par constructeur (donc pas de règle d'introduction pour le type inductif sans constructeur), la précondition est juste la bonne formation des types de constructeurs.

$$\frac{\Gamma, \texttt{nom} : A \vdash C_m : s \quad (m \in 1..p) \qquad s \text{ sorte de l'arité } A}{\Gamma \vdash \texttt{co}_m : C_m}$$

3.2.3 Schémas d'élimination

Les règles d'introduction nous disent comment former des objets dans une définition inductive. Les règles d'élimination indiquent comment utiliser un objet $x : (\texttt{nom}\ u_1 \dots u_l)$. Il y a plusieurs manières d'exprimer cette propriété.

Le schéma d'élimination minimal

L'interprétation usuelle des définitions inductives est que les valeurs (objets clos normaux) dans une instance de la définition inductive sont exactement ceux formés à partir des constructeurs.

Donc si on a un objet x dans la définition inductive $(\texttt{nom}\ u_1 \dots u_l)$ et que l'on veut montrer une propriété C, il suffit de regarder les cas où $x = (\texttt{co}_m\ x_1 \dots x_{n_m})$ avec x_i quelconque du type approprié. Il suffit donc de montrer pour chaque m :

$$\forall (x^1 : C_m^1) \dots (x^{n_m} : C_m^{n_m}), (u_1, \dots, u_l, x) = (t_m^1 \dots t_m^l, (\texttt{co}_m\ x^1 \dots x^{n_m})) \to C$$

Cependant faire intervenir des n-uplets et l'égalité qui ne sont pas des notions primitives parait peu intuitif (il faudrait commencer par préciser les règles pour ces deux types inductifs). Pour éviter cela, on peut intégrer le raisonnement égalitaire dans le schéma d'élimination.

On suppose que l'on a une propriété P de type $\forall (a_1 : A_1) \dots (a_l : A_l), (\texttt{nom}\ a_1 \dots\ a_l) \to s'$. Pour prouver $\forall (a_1 : A_1) \dots (a_l : A_l), (x : (\texttt{nom}\ a_1 \dots\ a_l))(P\ a_1 \dots\ a_l\ x)$, il suffit de montrer pour chaque constructeur $\texttt{co}_m$, la propriété :

$$\forall (x^1 : C_m^1) \dots (x^{n_m} : C_m^{n_m}), (P\ t_m^1 \dots t_m^l\ (\texttt{co}_m\ x^1 \dots x^{n_m}))$$

On obtient donc le schéma d'élimination suivant qui est paramétré par la sorte s' de la propriété à prouver :

$$\frac{\begin{array}{c}\Gamma \vdash x : (\texttt{nom}\ u_1 \dots\ u_l) \qquad \Gamma \vdash P : \forall (a_1 : A_1) \dots (a_l : A_l), (\texttt{nom}\ a_1 \dots\ a_l) \to s' \\ \Gamma \vdash f_m : \forall (x^1 : C_m^1) \dots (x^{n_m} : C_m^{n_m}), (P\ t_m^1 \dots t_m^l\ (\texttt{co}_m\ x^1 \dots x^{n_m})) \quad (m \in 1..p)\end{array}}{\Gamma \vdash \mathsf{Case}(P, x, f_1, \dots, f_p) : (P\ u_1 \dots\ u_l\ x)}$$

Où Case est un nouveau constructeur de terme.

Réduction Comme toute règle d'élimination, celle-ci se combine avec les règles d'introduction pour former une réduction. Si la fonction d'élimination est appliquée au m-ième constructeur alors elle se réduit en la m-ème branche instanciée par les arguments du constructeur. On vérifie que le type est préservé. Cette réduction est appelée la ι-réduction et s'écrit :

$$\mathsf{Case}(P, (\texttt{co}_m\ x^1 \dots x^{n_m}), f_1, \dots, f_p) \longrightarrow_\iota (f_m\ x^1 \dots x^{n_m})$$

On remarque que cette construction est analogue à une déclaration par filtrage, dans laquelle on examine dans l'ordre les constructeurs et on n'a que des motifs de la forme $(\texttt{co}_m\ x^1 \dots x^{n_m})$ avec

$\mathtt{co}_m$ un constructeur et x^k une variable. On peut utiliser la notation :

$$\begin{array}{l}
\lambda(a_1 : A_1)\dots(a_l : A_l)(x : (\mathtt{nom}\ a_1 \dots a_l)) \Rightarrow \\
\mathsf{match}\ x\ \mathsf{with} \quad \begin{array}{lcl} (\mathtt{co}_1\ x^1 \dots x^{n_1}) & \Rightarrow & (f_1\ x^1 \dots x^{n_1}) \\ \mid \dots & & \\ \mid (\mathtt{co}_p\ x^1 \dots x^{n_p}) & \Rightarrow & (f_m\ x^1 \dots x^{n_p}) \end{array} \\
\mathsf{end}
\end{array}$$

Élimination non dépendante Un cas particulier souvent utilisé de l'élimination est celui où le prédicat P ne dépend pas de l'objet éliminé. On parle alors d'élimination *non-dépendante*. La règle devient :

$$\frac{\Gamma \vdash x : (\mathtt{nom}\ a_1 \dots\ a_l) \qquad \Gamma \vdash P : \forall (a_1 : A_1)\dots(a_l : A_l), s' \qquad \Gamma \vdash f_m : \forall (x^1 : C_m^1)\dots(x^{n_m} : C_m^{n_m}), (P\ t_m^1 \dots t_m^l) \quad (m \in 1..p)}{\Gamma \vdash \mathsf{Case}(P, x, f_1, \dots, f_p) : (P\ a_1 \dots\ a_l)}$$

Codage imprédicatif Les définitions inductives peuvent être vues comme de nouveaux objets ajoutés à la théorie, ou bien on peut essayer de les coder, c'est-à-dire, étant donné une théorie fixée, et une déclaration inductive de la forme 3.1.1, peut-on trouver des termes $\mathtt{nom}$, $\mathtt{co}_m$ et Case qui ont les types appropriés et satisfont les règles de réduction.

Un des attraits des logiques d'ordre supérieur est qu'elles sont assez puissantes pour coder les définitions inductives. C'est d'ailleurs ce qui est fait dans les systèmes HOL, où les définitions inductives sont juste des packages plus ou moins puissants automatisant la définition de propriétés.

Dans le Calcul des Constructions Pures, il est possible de coder les définitions inductives. On introduit les définitions suivantes :

$$\begin{array}{lll}
\mathtt{nom} & := & \lambda(a_1 : A_1)\dots(a_l : A_l) \Rightarrow \\
& & \forall P : A, C_1\{\mathtt{nom} := P\} \to \dots \to C_n\{\mathtt{nom} := P\} \to (P\ a_1 \dots a_l) \\
\mathtt{co}_m & := & \lambda(x_1 : C_m^1)\dots(x_{n_m} : C_m^{n_m}) \\
& & (P : A)(f_1 : C_1\{\mathtt{nom} := P\})\dots(f_n : C_n\{\mathtt{nom} := P\}) \Rightarrow (f_m\ x_1 \dots x_{n_m}) \\
\mathtt{Case} & := & \lambda(P : A)(a_1 : A_1)\dots(a_l : A_l)(x : (\mathtt{nom}\ a_1 \dots a_l)) \\
& & (f_1 : C_1\{\mathtt{nom} := P\})\dots(f_n : C_n\{\mathtt{nom} := P\}) \Rightarrow (x\ P\ f_1\ \dots f_n)
\end{array}$$

On vérifie que les objets introduits sont bien typés du type attendu sauf dans le cas du schéma d'élimination Case où le type est plus faible que celui que nous avons proposé précédemment. En effet on intègre dans le schéma le fait que dans chaque branche $(a_1, \dots, a_l)$ est égal à $(t_m^1, \dots t_m^l)$ mais pas le fait que x lui-même est égal à $(\mathtt{co}_m\ x^1 \dots x^{n_m})$. On appelle le schéma ainsi obtenu le *schéma itératif*. Dans le cas non récursif, ce shéma est le même que le schéma non-dépendant. On remarque également que le schéma d'élimination que nous avons codé a une sorte d'élimination qui est la même que la sorte de la définition inductive.

Cette représentation imprédicative a l'avantage de ne pas nécessiter d'extension de la théorie et donc de ne pas poser de problème théorique. Cependant, la faiblesse du schéma d'élimination fait que certaines propriétés attendues ne sont pas prouvables.

On peut construire le type des booléens : $\mathtt{bool} := \forall C : \mathtt{Set}, C \to C \to C$ avec $\mathtt{true} := \lambda(C : \mathtt{Set})(t\ f : C) \Rightarrow t$ et $\mathtt{false} := \lambda(C : \mathtt{Set})(t\ f : C) \Rightarrow f$ mais il n'est pas possible de prouver $\mathtt{true} \neq \mathtt{false}$ et il n'est pas possible de montrer :

$$\forall b : \mathtt{bool}, ((b = \mathtt{true}) \vee (b = \mathtt{false}))$$

Il devient alors difficile de raisonner sur les booléens sauf à ajouter ces propriétés comme axiome. cependant, on perd alors la correspondance entre le calcul et le raisonnement.

Une autre difficulté du codage imprédicatif est qu'il y a plus de termes normaux clos dans le codage du type inductif que ceux construits à partir des constructeurs.

L'exemple le plus simple illustrant ce phénomène consiste à prendre le type inductif I des singletons avec un seul constructeur $c : (\texttt{Un} \to \texttt{Un}) \to I$. Le codage imprédicatif donne :

$$\begin{array}{ll} \texttt{Un} & := \forall C : \texttt{Set}, C \to C \\ \texttt{I} & := \forall C : \texttt{Set}, ((\texttt{Un} \to \texttt{Un}) \to C) \to C \\ \texttt{c} & := \lambda(f : \texttt{Un} \to \texttt{Un})(C : \texttt{Set})(h : (\texttt{Un} \to \texttt{Un}) \to C) \Rightarrow (h\ f) \end{array}$$

Il n'est pas difficile de construire un terme normal clos de type I qui ne peut s'écrire sous la forme $(\texttt{c}\ f)$ avec f un terme normal clos de type $\texttt{Un} \to \texttt{Un}$.

On distingue une classe de types appelés "types de données" qui correspondent à des codages de définitions inductives dans lesquelles tous les types d'arguments des constructeurs sont eux-mêmes récursivement des "types de données" (il est surtout essentiel qu'il n'y ait pas de quantification d'ordre supérieur en position négative). Pour les types de données, il est possible d'établir un théorème de représentation à savoir que tous les termes clos normaux sont β-équivalent à un constructeur appliqué à des arguments clos du bon type. Cependant, même pour les types de données, le schéma qui dit que les seuls objets dans le type sont ceux obtenus via les constructeurs n'est pas démontrable.

Définition de types par cas/Élimination forte

On appelle schéma d'élimination forte, le schéma d'élimination d'un inductif d'une sorte imprédicative (par exemple `Prop` mais aussi `Set` dans le calcul des constructions avec `Set` imprédicatif) vers des prédicats de sorte `Type`.

Si on reprend l'exemple des booléens, le schéma d'élimination forte a la forme suivante :

$$\frac{b : \texttt{bool} \quad P : \texttt{bool} \to \texttt{Type} \quad f : (P\ \texttt{true}) \quad g : (P\ \texttt{false})}{\mathsf{Case}(P, b, f, g) : (P\ b)}$$

On peut en particulier instancier ce schéma avec $P := \lambda b : \texttt{bool} \Rightarrow \texttt{Prop}$, on obtient

$$\frac{f : \texttt{Prop} \quad g : \texttt{Prop}}{\lambda b\ : \texttt{bool} \Rightarrow \mathsf{Case}(P, b.f, g) : \texttt{bool} \to \texttt{Prop}}$$

En prenant pour f la propriété toujours vrai `True` et pour g la propriété toujours fausse `False` on obtient une propriété ϕ de type $\texttt{bool} \to \texttt{Prop}$ telle que $(\phi\ \texttt{true})$ est équivalent à `True` et $(\phi\ \texttt{false})$ équivalent à `False`. Cette propriété P nous permet de réfuter le fait que $\texttt{true} = \texttt{false}$.

On peut également prendre $P := \lambda b : \texttt{bool} \Rightarrow \texttt{Set}$, $f := \texttt{nat}$ et $g := \texttt{bool}$, on pourra alors définir un type ψ de type $\texttt{bool} \to \texttt{Set}$ tel que $(\phi\ \texttt{true})$ est équivalent à `nat` et $(\phi\ \texttt{false})$ équivalent à `false`. Pour construire un objet dans ce type, il est encore nécessaire d'utiliser le schéma d'élimination de `bool` en prenant cette fois-ci $P := \psi, f : \texttt{nat}$ et $g : \texttt{bool}$. On a alors :

$$\mathsf{Case}(\psi, b, f, g) : (\psi\ b)$$

Ces fonctions sortent du cadre de ce qui peut être typé dans un langage à la ML, elles sont pourtant très utiles dans les techniques de preuve à base de réflexion pour interpréter les objets d'un type concret de proposition vers des propriétés `Coq`.

Nous verrons dans la partie 3.2.4 que l'élimination forte ne peut être autorisée pour tous les types inductifs sans risque de rendre le système incohérent.

Types dépendants

Les définitions inductives permettent de définir des familles de type $I : \forall(a_1 : A_1)\ldots(a_n : A_n), s$. Par exemple, on peut définir une relation `neq` sur les booléens par :

$$\begin{array}{l}\mathsf{Inductive}\ \mathtt{neq}\ : \mathtt{bool} \to \mathtt{bool} \to \mathtt{Prop}\\ := \mathtt{neq}_1 : (\mathtt{neq}\ \mathtt{true}\ \mathtt{false})\\ \ \ \mid \mathtt{neq}_2 : (\mathtt{neq}\ \mathtt{false}\ \mathtt{true}).\end{array}$$

Le schéma d'élimination permet de montrer des propriétés de la forme :

$$\forall(x_1\ x_2 : \mathtt{bool}), (\mathtt{neq}\ x_1\ x_2) \Rightarrow (P\ x_1\ x_2)$$

Il est étrangement plus complexe de construire une preuve de $(\mathtt{neq}\ b_1\ b_2) \Rightarrow P$ lorsque b_1 et b_2 sont des termes et plus seulement des variables distinctes. C'est le cas par exemple si on veut montrer $(\mathtt{neq}\ \mathtt{true}\ \mathtt{true}) \Rightarrow \bot$ ou $(b : \mathtt{bool})(\mathtt{neq}\ b\ b) \Rightarrow \bot$.

En fait pour pouvoir utiliser le schéma d'élimination il faut pouvoir généraliser la propriété à montrer en : $\forall(x_1\ x_2 : \mathtt{bool}), (\mathtt{neq}\ x_1\ x_2) \Rightarrow (Q\ x_1\ x_2)$. puis l'appliquer à b_1 et b_2. Les exemples ci-dessus montrent qu'une généralisation naïve ne fonctionne pas en général. Une manière systématique de résoudre ce problème est de renforcer la propriété à prouver en :

$$\forall(x_1\ x_2 : \mathtt{bool}), (\mathtt{neq}\ x_1\ x_2) \Rightarrow x_1 = b_1 \Rightarrow x_2 = b_2 \Rightarrow P$$

puis d'utiliser l'élimination standard et de simplifier les égalités. Certaines correspondent à des hypothèses absurdes, d'autres vont donner lieu à des simplifications, on obtiendra de nouvelles égalités qui pourront être propagées.

Tactiques d'inversion Le travail décrit ci-dessus est (partiellement) automatisé par les tactiques d'inversion. Cependant la preuve engendrée est loin d'être atomique. Il faut donc éviter autant que possible d'avoir recours aux schémas d'inversion. Pour cela, il est utile de mettre explicitement en paramètre tout argument qui n'est pas instancié dans les constructeurs (juste une variable liée). L'ordre dans lequel on réalise les éliminations, et des généralisations appropriées permettent parfois d'éviter l'utilisation de l'inversion. Finalement, il peut être utile d'engendrer des principes d'élimination ad-hoc pour certaines instances de définition inductive, ce qui permet d'éviter l'utilisation multiple de l'inversion.

Égalité

On suppose fixé un type $A : \mathtt{Set}$.

L'égalité inductive est définie par

$$\mathsf{Inductive}\ \mathtt{eq}\ (x : A) : A \to \mathtt{Prop} := \mathtt{refl} : (\mathtt{eq}\ x\ x)$$

On note $x = y$ le terme $(\mathtt{eq}\ x\ y)$. Le schéma d'élimination exprime que toute preuve de $x = y$ est de la forme $(\mathtt{refl}\ x)$, il a la forme suivante :

$$\frac{e : (x = y) \qquad P : \forall(y : A), x = y \to s \qquad p : (P\ x\ (\mathtt{refl}\ x))}{\mathsf{Case}(P, e, p) : (P\ y\ e)}$$

Il vérifie la règle de réduction :

$$\mathsf{Case}(P, (\mathtt{refl}\ x), p) \longrightarrow_\iota p$$

Cette égalité permet de comparer deux objets du même type. Cependant, on est parfois amenés à devoir comparer des objets dans des types différents.

Un exemple est la définition du type (récursif) des listes de longueur n :

$$\begin{array}{l}\textsf{Inductive}\ \texttt{list}\ : \texttt{nat} \rightarrow \texttt{Prop} \\ \quad := \texttt{nil} : (\texttt{list}\ 0) \mid \texttt{cons} : \forall n : \texttt{nat}, A \rightarrow (\texttt{list}\ n) \rightarrow (\texttt{list}\ (\texttt{S}\ n)).\end{array}$$

Une propriété attendue est $\forall l : (\texttt{list}\ 0), l = \texttt{nil}$. Pour la montrer, en utilisant la généralisation pour l'inversion, on voudrait se ramener à montrer : $\forall (n : \texttt{nat})(l : (\texttt{list}\ n)), n = 0 \rightarrow l = \texttt{nil}$. Malheureusement, cette propriété ne peut être énoncée car la propriété $l = \texttt{nil}$ est mal formée puisque $l : (\texttt{list}\ n)$ et $\texttt{nil} : (\texttt{list}\ 0)$ sont de type différents non convertibles. Le fait qu'il existe une preuve e de $n = 0$ dans le contexte permet juste d'appliquer une transformation à l pour en faire un objet de type $(\texttt{list}\ 0)$. Mais l'énoncé devient

$$\forall (n : \texttt{nat})(l : (\texttt{list}\ n))(e : n = 0), \textsf{Case}(\texttt{list}, e, l) = \texttt{nil}$$

On peut ensuite faire une preuve par cas sur l, cependant on se retrouve à devoir montrer :

$$\forall (e : 0 = 0), \textsf{Case}(\texttt{list}, e, \texttt{nil}) = \texttt{nil}$$

On souhaiterait alors utiliser le fait que toute preuve de $0 = 0$ est de la forme $(\texttt{refl}\ 0)$, en remplaçant e par cette valeur, on doit montrer

$$\textsf{Case}(\texttt{list}, (\texttt{refl}\ 0), \texttt{nil}) = \texttt{nil}$$

En utilisant la ι réduction, on aboutit à la trivialité $\texttt{nil} = \texttt{nil}$. Malheureusement, le remplacement de $e : 0 = 0$ par $(\texttt{refl}\ 0)$ ne peut se faire. En effet, il faut utiliser le schéma d'élimination qui demande de généraliser le but sous la forme d'un prédicat de type $\forall y : \texttt{nat}, (0 = y) \rightarrow \texttt{Prop}$ or les dépendances nous empêchent de généraliser comme on le souhaiterait :

$$\lambda (y : \texttt{nat})(e : 0 = y) \Rightarrow \textsf{Case}(\texttt{list}, e, \texttt{nil}) = \texttt{nil}$$

n'est pas un terme bien typé.

Cette "pathologie" a été mise en évidence par Thierry Coquand. Thomas Streicher et Martin Hoffman ont exhibé des modèles de la théorie des types, pour lesquels il y a des preuves de $x = x$ qui ne sont pas convertibles à $(\texttt{refl}\ x)$. Cependant, s'il existe une égalité décidable sur A (ce qui est le cas de tous les types de données usuels), alors cette propriété est démontrable mais la construction est complexe (l'idée originale est dûe à Michael Hedberg, elle a été codée dans LEGO par Thomas Kleymann et reprise dans Coq par Bruno Barras, cf le fichier `theories/Logic/Eqdep_dec.v`).

L'axiome K de Streicher Thomas Streicher a proposé d'ajouter un principe d'élimination plus puissant pour l'égalité qui capture exactement que toute preuve de $x = x$ est une preuve par réflexivité :

$$\frac{e : (x = x) \quad P : x = x \rightarrow s \quad p : (P\ (\texttt{refl}\ x))}{\textsf{Case}_K(P, e, p) : (P\ e)}$$

Ce nouvel opérateur satisfait un règle de ι-réduction :

$$\textsf{Case}_K(P, (\texttt{refl}\ x), p) \longrightarrow_\iota p$$

Il existe de nombreuses formes équivalentes de cet axiome. La théorie EqDep de Coq introduit un axiome qui dit que :

$$\forall (e : x = x)(p : (P\ x)), \textsf{Case}(P, e, p) = p$$

qui ne dit pas que e est égal à $(\texttt{refl}\ x)$ mais qu'il se comporte calculatoirement de manière identique.

Égalité de John Major C. Mc Bride a introduit une égalité qui permet de comparer deux objets dans des types quelconques. De manière inductive, cette égalité se définit par :

$$\textsf{Inductive}\ \texttt{eq}\ (A:\texttt{Set})(x:A):\forall B:\texttt{Set}, B\to\texttt{Prop} := \texttt{refl}:(\texttt{eq}\ A\ A\ x\ x).$$

On peut facilement prouver que cette égalité est réflexive. symétrique et transitive en utilisant le schéma général des définitions inductives. Par contre, ce schéma impose une généralisation du but sous la forme d'un prédicat de type $\forall B:\texttt{Set}, B\to\texttt{Prop}$, ce qui est en général malaisé.

C. Mc Bride propose d'introduire un schéma d'élimination renforcé, analogue (et prouvablement équivalent) à l'axiome K de Streicher. Ce schéma d'élimination dit que si deux objets *de même type* sont égaux selon l'égalité de John Major, alors on peut remplacer l'un par l'autre (ie ils sont égaux au sens de l'égalité de Leibniz).

$$\frac{P:A\to s \quad e:(\texttt{eq}\ A\ x\ A\ y) \quad q:(P\ x)}{\textsf{Case}(P,e,q):(P\ y)}$$

Cette égalité est commode car si on a $n:nat$ et $l:(\texttt{list}\ n)$ ainsi que $n':nat$ et $l':(\texttt{list}\ n')$, on pourra simplement écrire $(\texttt{eq}\ \texttt{nat}\ n\ \texttt{nat}\ n')$ et $(\texttt{eq}\ (\texttt{list}\ n)\ l\ (\texttt{list}\ n')\ l')$. Comme n et n' sont du même type, on pourra remplacer n par n' en particulier dans le type de l, ensuite l et l' seront du même type $(\texttt{list}\ n')$ et on pourra remplacer l par l'. Cette égalité évite de passer par un codage de paires assez lourd.

3.2.4 Types inductifs et sortes

Nous n'avons pour l'instant pas préciser les sortes pour lesquelles l'élimination d'une définition inductive était possible.

Les types de données prédicatifs

Les types de données prédicatifs sont ceux pour lesquels les types des arguments des constructeurs sont dans la même sorte que l'inductif. C'est forcément le cas si le type inductif est dans `Type` ou bien dans `Set` dans le cas où cette sorte est aussi prédicative.

Les types de données imprédicatifs

Un type de données est imprédicatif s'il est défini dans `Prop` ou `Set` (imprédicatif) et si au moins un des types d'argument est dans la sorte `Type`. C'est le cas de la définition :

$$\textsf{Inductive}\ \texttt{prop}\ :\texttt{Prop} := \texttt{in}:\texttt{Prop}\to\texttt{prop}.$$

Si on autorisait, pour cette définition, une élimination forte alors on pourrait définir :

$$\texttt{out}(p:\texttt{prop}):\texttt{Prop} := \textsf{match}\ p\ \textsf{with}\ (\texttt{in}\ P)\Rightarrow P\ \textsf{end}$$

On vérifie de plus que $(\texttt{out}\ (\texttt{in}\ P))\simeq P$. On a donc `prop`:`Prop` qui est isomorphe à `Prop` : `Type` ce qui introduit deux niveaux d'imprédicativité et conduit à un paradoxe.

Une possibilité est d'interdire les définitions imprédicatives, celles-ci peuvent être représentées par un codage à l'ordre supérieur. Dans `Coq`, de telles définitions sont autorisées, on peut utiliser le schéma d'élimination usuel sur la sorte imprédicative de la définition inductive par contre l'élimination forte (sur un prédicat dont la sorte est `Type`) ne peut pas leur être appliquée.

De telles définitions inductives sont utiles pour coder des existentielles du second ordre qui servent elle-même à coder des types abstraits : on sait qu'il existe un type avec certaines opérations sur ce type mais on ne peut pas accéder à l'implantation de ce type. Un exemple de telle utilisation est la définition `nu` dans `contribs/Lyon/GFP.v` qui code un plus grand point-fixe d'un opérateur monotone sur les types.

La distinction entre `Prop` et `Set`

Les sortes `Prop` et `Set` sont toutes les deux de type `Type`. Leur interprétation diffère vis-à-vis de l'extraction de programmes à partir de preuves. Un terme de preuve est dit *logique* s'il est de type P : `Prop`; le typage garantit qu'il ne servira pas de manière calculatoire pour construire un terme de preuve *calculatoire* de type S : `Set`. Ceci permet d'éliminer les termes de preuve logique qui sont toujours en position de code mort dans les termes calculatoires.

Si un type inductif est de type `Prop` alors on ne peut pas en général autoriser une élimination sur la sorte `Set`. En effet la règle d'élimination est :

$$\frac{\begin{array}{c}\Gamma \vdash x : (\texttt{nom}\ a_1 \ldots\ a_l) \qquad \Gamma \vdash P : \forall(a_1 : A_1)\ldots(a_l : A_l), (\texttt{nom}\ a_1 \ldots\ a_l) \to s' \\ \Gamma \vdash f_m : \forall(x^1 : C_m^1)\ldots(x^{n_m} : C_m^{n_m}), (P\ t_m^1 \ldots t_m^l\ (\texttt{co}_m\ x^1 \ldots x^{n_m}))\quad (m \in 1..p)\end{array}}{\Gamma \vdash \mathsf{Case}(P, x, f_1, \ldots, f_p) : (P\ a_1 \ldots\ a_l\ x)}$$

Si x est logique, il doit disparaître à l'extraction, par contre si s' est de type `Set` on doit être en mesure de fournir une réalisation de $(P\ a_1 \ldots\ a_l\ x)$. On a beau pouvoir extraire chaque branche f_i, en l'absence de x, il est difficile de les combiner.

On aboutit à la règle suivante : un inductif de sorte `Prop` ne peut s'éliminer sur un prédicat de sorte `Set`. Il ne peut pas non plus s'éliminer sur un prédicat de sorte `Type`, car tout objet dans `Set` est également un objet de la sorte `Type`, donc l'élimination sur `Type` implique l'élimination sur `Set`.

De plus l'élimination sur `Type` permet de montrer qu'il existe une propriété A : `Prop` et $a, b : A$ tel que $a \neq b$ est prouvable (par analogie avec la preuve de $\texttt{true} \neq \texttt{false}$). Or du fait de notre interprétation non calculatoire des propositions logiques, supposer que toutes les preuves d'une même proposition logique sont égales est parfois utile. C'est également une propriété que l'on dérive à partir d'autres axiomes tel que celui de la logique classique $\forall A : \texttt{Prop}, A \vee \neg A$ ou de l'extensionalité $\forall A\ B : \texttt{Prop}, (A \leftrightarrow B) \to A = B$.

Les types singletons Il existe deux cas particuliers d'inductif définis dans `Prop` pour lesquels l'élimination sur la sorte `Set` ne pose pas de difficulté. Il s'agit tout d'abord de la définition inductive vide. Dans une situation ou l'absurde est prouvable, n'importe quel objet est un programme correct par rapport à n'importe quelle définition inductive.

Le second cas est plus intéressant. Nous appelons *type singleton*, un type qui n'a qu'un constructeur dont tous les types d'arguments sont de type `Prop`. L'élimination ne comporte donc qu'une seule branche et on montre aisément que si la propriété est vérifiée alors le programme extrait de cette branche est correct par rapport à la propriété de la conclusion. L'extraction de la définition par cas est alors définie comme l'extraction de l'unique branche, l'élément sur lequel se fait le Case n'intervient donc pas dans le calcul. On peut montrer que pour de tels types, dès que l'on a l'élimination vers `Set` et que ces types sont prédicatifs, alors l'élimination vers `Type` peut également être simulée et est donc valide.

Les types conjonctions de propriétés logiques sont des types singletons, bien que définis dans `Prop` ils admettent une élimination pour toutes les sortes, il en est de même du prédicat d'égalité que nous avons défini plus haut.

La condition que tous les types d'arguments de constructeur sont logiques est essentielle. Un exemple de type à un seul constructeur qui ne vérifie pas cette condition est un type `I` avec un seul constructeur `c` avec un argument de type `bool`. La règle d'élimination :

$$\frac{x : I \qquad P : I \to \texttt{Set} \qquad f : \forall b : \texttt{bool}, (P\ (c\ b))}{\mathsf{Case}(P, x, f) : (P\ x)}$$

Sur le plan calculatoire, la preuve x est essentielle pour décider l'instance de la branche à choisir (f `true`) ou bien (f `false`).

3.3 Les types inductifs récursifs

3.3.1 Exemples

La définition inductive récursive de base est bien entendu celle des entiers :

$$\mathsf{Inductive}\ \mathtt{nat} : \mathtt{Set} := \mathtt{0} : \mathtt{nat} \mid \mathtt{S} : \mathtt{nat} \to \mathtt{nat}.$$

À peine plus compliquée est celle des listes, qui peut être paramétrée par le type des éléments :

$$\mathsf{Inductive}\ \mathtt{list}(A : \mathtt{Set}) : \mathtt{Set} := \mathtt{nil} : (\mathtt{list}\ A) \mid \mathtt{cons} : A \to (\mathtt{list}\ A) \to (\mathtt{list}\ A).$$

On construit aisément sur le même modèle le type des arbres ou de manière plus générale de toute structure de terme algébrique.

Un exemple un peu plus sophistiqué est celui des notations ordinales (du second ordre). Il se construit comme le type des entiers, mais on ajoute un constructeur de limite qui correspond à un branchement infini paramétré par des entiers. Cette structure infinie se représente de manière finie par une fonction des entiers vers les ordinaux.

$$\mathsf{Inductive}\ \mathtt{ord} : \mathtt{Set} := \mathtt{0} : \mathtt{ord} \mid \mathtt{S} : \mathtt{ord} \to \mathtt{ord} \mid \mathtt{lim} : (\mathtt{nat} \to \mathtt{ord}) \to \mathtt{ord}.$$

En suivant le même modèle, on peut définir un type générique d'arbre où les branchements sont indicés par un premier type de données A (il y a autant de type de branchements possibles que d'éléments dans A) et où l'arité de chaque branchement est donnée par un type B qui dépend de l'indice du branchement : Ce type a été initialement introduit par Per Martin-Löf et est traditionnellement écrit W. Il se définit en Coq par :

$$\mathsf{Inductive}\ \mathtt{W}(A : Set)(B : A \to \mathtt{Set}) : \mathtt{Set} := \mathtt{node} : \forall x : A, ((B\ x) \to (\mathtt{W}\ A\ B)) \to (\mathtt{W}\ A\ B).$$

Ce type est suffisant pour coder les autres définitions inductives. Par exemple, pour représenter les entiers, on remarque qu'il nous faut deux types de branchement (l'un pour $\mathtt{0}$ qui est d'arité nulle et l'autre pour $\mathtt{S}$ qui est d'arité un). Il suffit donc de prendre $A \equiv \mathtt{bool}$ et B définie par $(B\ \mathtt{true}) \equiv \mathtt{False}$ et $(B\ \mathtt{false}) \equiv \mathtt{True}$. On pose alors : $\mathtt{nat} \equiv (\mathtt{W}\ A\ B)$ On peut alors introduire $\mathtt{0} \equiv (\mathtt{node}\ \mathtt{true}\ \lambda t : \mathtt{False} \Rightarrow \mathsf{Case}(\mathtt{nat}, t))$ et $\mathtt{S} \equiv \lambda n : \mathtt{nat} \Rightarrow (\mathtt{node}\ \mathtt{false}\ \lambda t : \mathtt{True} \Rightarrow n)$ dont on vérifie qu'ils ont le bon type.

Un autre type récursif intéressant est la définition d'un élément bien fondé $x : A$ pour une relation $R : A \to A \to \mathtt{Prop}$. On dit que x est bien fondé si tous les éléments en relation avec x sont eux-mêmes bien fondés. Cela s'écrit :

$$\begin{array}{l}\mathsf{Inductive}\ \mathtt{wf}\ (A : Set)(R : A \to A \to \mathtt{Prop}) : A \to \mathtt{Prop} \\ \quad := \mathtt{wf\_intro} : \forall x : A, (\forall y : A, (R\ y\ x) \to (\mathtt{wf}\ A\ R\ y)) \to (\mathtt{wf}\ A\ R\ x)\end{array}$$

3.3.2 Condition de positivité

Positivité large

Un type inductif qui comporterait une occurrence récursive négative, permet de construire des objets qui bouclent sans utiliser de récursivité. Supposons que l'on puisse définir :

$$\mathsf{Inductive}\ \mathtt{L} : \mathtt{Set} := \mathtt{lam} : (\mathtt{L} \to \mathtt{L}) \to \mathtt{L}.$$

et que l'élimination sur la sorte $\mathtt{Set}$ soit permise. Ce type est habité puisqu'il contient au moins le terme $(\mathtt{lam}\ [x : \mathtt{L}]x)$ On peut également définir :

$$\mathsf{Definition}\ \mathsf{app}\ (l_1\ l_2 : \mathtt{L}) : \mathtt{L} := \mathsf{match}\ l_1\ \mathsf{with}\ (\mathtt{lam}\ f) \Rightarrow (f\ l_2)\ \mathsf{end}$$

La réduction suivante est satisfaite :

$$(\mathtt{app}\ (\mathtt{lam}\ f)\ l) \longrightarrow_\iota (f\ l)$$

On construit alors :

$$\mathsf{Definition}\ \delta : \mathtt{L} := (\mathtt{lam}\ \lambda x : \mathtt{L} \Rightarrow (\mathtt{app}\ x\ x))$$

On vérifie alors que le terme $(\mathtt{app}\ \delta\ \delta)$ se réduit en une étape de ι réduction vers lui-même et donc que ce terme n'est pas normalisable.

Cet exemple peut être aisément codé en ML. Par contre, il explique pourquoi les définitions inductives doivent imposer une condition de positivité.

Une définition inductive I est positive si les seules occurrences de I dans des types d'arguments de constructeur se font de manière positive, c'est-à-dire à la gauche d'un nombre pair (éventuellement nul) de produits avec la définition suivante : Dans le terme $\forall x : A, B$, le terme B est à gauche de 0 produit, un sous-terme C de B qui est à gauche de n produits dans B est également à gauche de n produits dans $\forall x : A, B$. Le terme A est à gauche d'un produit dans $\forall x : A, B$ et un sous-terme C de A qui est à gauche de n produits dans A est également à gauche de $n+1$ produits dans $\forall x : A, B$.

Des exemples de type de constructeur positif pour I sont :
- $A \to I$ avec I qui n'apparaît pas dans A
- $(I \to A) \to B$ avec I qui n'apparaît pas dans A mais peut apparaître positivement dans B

Un type d'argument C qui dépend de I de manière positive satisfait une condition de monotonicité, c'est-à-dire que si $I \subseteq J$ (ie il existe un terme de type $\forall (a_1 : A_1) \dots (a_l : A_l), (I\ a_1 \dots a_l) \to (J\ a_1 \dots a_l)$) alors on peut construire un terme de type $C \to C\{I := J\}$. Cette condition de monotonicité suffit à garantir l'existence d'un plus petit point fixe qui peut être codé de manière imprédicative.

Par contre, cette positivité au sens large ne convient pas lorsqu'il s'agit de définir un type inductif au niveau prédicatif `Type`. Th. Coquand [CPM90] a montré que accepter la définition inductive suivante conduisait à un paradoxe :

$$\mathsf{Inductive}\ \mathtt{X} : \mathtt{Type} := \mathtt{in} : ((\mathtt{X} \to \mathtt{Prop}) \to \mathtt{Prop}) \to \mathtt{X}.$$

En effet, pour tout Y, il existe une application ψ de Y dans $Y \to \mathtt{Prop}$ qui à $y : Y$ associe le prédicat $\lambda y' \Rightarrow y = y'$. On en déduit une application ϕ de $X \to \mathtt{Prop}$ dans X qui à P de type $X \to \mathtt{Prop}$ associe $(\mathtt{in}\ \lambda P'.P = P')$. On vérifie que ϕ est une injection : $\phi(P) = \phi(P') \to P = P'$. Il suffit alors de considérer le prédicat $P_0 = \lambda x.\exists P, \phi(P) = x \wedge \neg(P\ x)$ et de prendre $x_0 = \phi(P_0)$. On vérifie alors aisément que $(P_0\ x_0)$ est equivalent à $\neg(P_0\ x_0)$ d'où une incohérence.

Positivité stricte

Les définitions inductives de Coq ne permettent pas de la positivité large (c'est-à-dire une occurrence de la définition inductive à la gauche d'au moins un produit).

Le schéma général est que si la définition inductive I apparaît dans un type d'argument d'un constructeur alors ce type d'argument à la forme :

$$\forall (z_1 : D_1) \dots (z_m : D_m), (I\ u_1 \dots u_l)$$

et I n'apparaît ni dans les D_i ni dans les u_i.

On vérifiera que toutes les définitions inductives données en exemple satisfont cette condition de positivité.

Récursivité emboîtée La positivité stricte n'interdit pas a priori que l'argument récursif se trouve comme paramètre d'une autre définition inductive. De fait, si un type de constructeur de la définition inductive I a pour type $(A*I) \to I$ ou $A*I$ est la définition inductive pour le produit de A et de I alors, I est équivalente à une définition J où le type de constructeur serait $A \to J \to J$ et donc strictement positif. De même si le constructeur est $(A+I) \to I$ alors on peut le remplacer par deux constructeurs strictement positifs de type $A \to J$ et $J \to J$. Cela fonctionne même si la définition imbriquée est récursive, il faut alors définir deux types mutuellement récursifs. Si le type de constructeur est $(\texttt{list}\ I) \to I$ on remplace I par une définition J définie de manière mutuellement récursive avec L qui représente $(\texttt{list}\ I)$. Le constructeur de type $(\texttt{list}\ I) \to I$ devient un constructeur de type $L \to J$, les constructeurs de L sont ceux de $(\texttt{list}\ I)$ soit $\texttt{nil}_L : L$ et $\texttt{cons}_L : J \to L \to L$.

Cette transformation nous montre qu'il est possible d'autoriser une occurrence récursive de l'inductif I comme paramètre d'une autre définition inductive J. Cependant certaines conditions doivent être vérifiée. L'occurrence de I doit apparaître strictement positivement en position de paramètre de la définition inductive J, et ce paramètre doit lui-même apparaître strictement positivement dans les types d'arguments de J. On peut donc avoir un type de constructeur de I de la forme $(\texttt{list}\ (A+I)) \to I$ par contre si on introduit

$$\textsf{Inductive}\ \texttt{neg}\ (X : \texttt{Prop}) : \texttt{Prop} := \texttt{in} : (X \to \texttt{False}) \to (\texttt{neg}\ X).$$

Alors définir un inductif I dont un type de constructeur est $(\texttt{neg}\ I) \to I$ est non valide.

3.3.3 Schéma d'élimination récursif primitif

Supposons que l'on définisse un inductif récursif qui vérifie la condition de stricte positivité sans imbrication. Alors le schéma d'élimination peut être renforcé pour prendre en compte le fait que la propriété que l'on cherche à montrer est récursivement satisfaite pour les sous-termes du type approprié.

On peut donc renforcer le schéma d'élimination en prenant sa forme récursive :

$$\frac{\begin{array}{c}\Gamma \vdash x : (\texttt{nom}\ a_1 \ldots\ a_l) \qquad \Gamma \vdash P : \forall (a_1 : A_1) \ldots (a_l : A_l), (\texttt{nom}\ a_1 \ldots\ a_l) \to s' \\ (m \in 1..p) \quad C_m^{i_1}, \ldots, C_m^{i_{r_m}} \text{ types d'arguments récursifs} \\ \Gamma \vdash f_m : \forall (x^1 : C_m^1) \ldots (x^{n_m} : C_m^{n_m}), C_m^{i_1}[P, x^{i_1}] \to \cdots C_m^{i_{r_m}}[P, x^{i_{r_m}}] \to (P\ t_m^1 \ldots t_m^l\ (\texttt{co}_m\ x^1 \ldots x^{n_m}))\end{array}}{\Gamma \vdash \textsf{Rec}(P, x, f_1, \ldots, f_p) : (P\ a_1 \ldots\ a_l\ x)}$$

Si C est un type d'argument récursif strictement positif et sans imbrication alors il est de la forme :

$$\forall (z_1 : D_1) \ldots (z_n : D_n), (\texttt{nom}\ u_1 \ldots u_l)$$

si $x : C$ on définit :

$$C[P, x] \equiv \forall (z_1 : D_1) \ldots (z_n : D_n), (P\ u_1 \ldots u_l\ (x\ z_1\ \ldots z_n))$$

On voit que ce schéma d'élimination est plus général que le schéma non récursif préalablement introduit. Il implique en particulier que l'ordre structurel sur le type inductif est bien fondé. Nos objets dans le type inductif ne représentent que des structures qui peuvent être infinies mais dont toutes les branches récursives sont finies.

Les schémas d'élimination récursive sont engendrés automatiquement par Coq au moment de la définition inductive. Dans le cas d'une définition de sorte Prop, le schéma déclaré correspond à une élimination non dépendante. Les schémas dépendants peuvent être introduits à l'aide de la commande de vernaculaire Scheme.

Dans le cas de définition inductive imbriquée, la forme du schéma d'élimination récursive n'est pas aussi simple à formuler. Coq ne fournit pas de facilité pour les introduire, c'est à l'utilisateur de concevoir un schéma adapté et de le démontrer à l'aide de l'élimination récursive et des constructions par point fixe que nous allons maintenant introduire.

3.3.4 Condition de garde

Le schéma d'élimination récursive était proposé par P. Martin-Löf comme la construction d'élimination de base des définitions inductives. Il permet de capturer la notion de fonctionnelle définie de manière primitive récursive ainsi que la notion de preuve par récurrence structurelle.

Cependant, Th. Coquand a suggéré une approche alternative, où comme dans les langages de programmation fonctionnelle, les notions primitives sont l'élimination non récursive (qui correspond à l'analyse par cas) et les définitions par point fixe. Évidemment, on ne peut autoriser n'importe quel point fixe sous peine de construire des termes non normalisables et d'aboutir à une incohérence. Une condition syntaxique de garde permet d'accepter les définitions qui suivent un schéma primitif récursif et de préserver la terminaison.

Pour cela, une fonction f peut être définie par point fixe si un de ses arguments x a pour type une définition inductive et si dans le corps de f tous les appels récursifs à f se font avec en place de x un terme t que l'on reconnaît syntaxiquement comme étant plus petit que x.

La règle principale pour être structurellement plus petit que x est d'apparaître dans une branche d'un Case sur x et d'être l'un des sous-termes de x. Mais cette définition peut être étendue. Par exemple un Case est reconnu plus petit que x si chacune de ses branches est plus petite que x (en particulier s'il n'y a aucune branche dans le cas de l'élimination d'une condition absurde). De même être plus petit est une opération transitive.

Les points fixes de Coq sont représentés de manière anonyme par un terme :

$$\mathsf{Fix}(f/n : T := t)$$

dans lequel f est une variable liée et n est un entier positif ou nul. La règle de typage est :

$$\frac{\Gamma, f : T \vdash t : T \quad f \text{ est gardée dans } t \text{ par le } n+1\text{ème argument}}{\Gamma \vdash \mathsf{Fix}(f/n : T := t) : T}$$

La règle de réduction de point fixe est :

$$\begin{array}{l}\text{si } a_{n+1} \text{ commence par un constructeur alors :}\\ (\mathsf{Fix}(f/n : T := t)\ a_1 \ldots\ a_{n+1}) \longrightarrow_\iota (t\{f := \mathsf{Fix}(f/n : T := t)\}\ a_1 \ldots\ a_{n+1})\end{array}$$

La règle de réduction est également gardée par la condition que l'argument de décroissance commence par un constructeur. Ceci permet d'éviter de poursuivre la réduction du point fixe à l'intérieur du terme ce qui violerait la terminaison forte.

On définit aisément à l'aide du point fixe la fonction qui calcule le minimum de deux entiers :

$$\mathsf{Fix}(\mathtt{min}/0 : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat} := [n,m]\mathsf{Case}(\mathtt{nat}, n, 0, [n']\mathsf{Case}(\mathtt{nat}, m, 0, [m'](\mathtt{S}\ (\mathtt{min}\ n'\ m')))))$$

Par contre la fonction d'Ackermann spécifiée par les équations :

$$(\mathtt{ack}\ 0\ m) = (\mathtt{S}\ m) \quad (\mathtt{ack}\ (\mathtt{S}\ n')\ 0) = (\mathtt{ack}\ n'\ (\mathtt{S}\ 0)) \quad (\mathtt{ack}\ (\mathtt{S}\ n')\ (\mathtt{S}\ m')) = (\mathtt{ack}\ n'\ (\mathtt{ack}\ (\mathtt{S}\ n')\ m')$$

nécessite l'utilisation de deux points fixe imbriqués :

$$\begin{array}{l}\mathsf{Fix}(\mathtt{ack}/0 : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat} :=\\ \quad [n]\mathsf{Case}(\mathtt{nat},\ n,\\ \qquad\qquad \mathtt{S},\\ \qquad\qquad [n']\mathsf{Fix}(\mathtt{ackn}/0 : \mathtt{nat} \to \mathtt{nat} :=\\ \qquad\qquad\qquad [m]\mathsf{Case}(\mathtt{nat}, m, (\mathtt{ack}\ n'\ (\mathtt{S}\ 0)), [m'](\mathtt{ack}\ n'\ (\mathtt{ackn}\ m'))))\\ \quad))\end{array}$$

La condition de choisir un argument de décroissance inductif est essentiel. En effet, du fait de la présence d'imprédicativité, on peut construire des types pour lesquels certains sous-arbres sont plus "grands" que l'objet initial. Ainsi si on introduit un type `I` avec un constructeur (non-récursif) `c` de type $(\forall A : \texttt{Set}, A \to A) \to I$. On introduit l'objet $t \equiv \lambda(A : \texttt{Set})(x : A) \Rightarrow x$ de type $\forall A : \texttt{Set}, A \to A$. L'objet $(\texttt{c}\ t)$ est de type `I`. Soit une fonction f de type $\texttt{I} \to \texttt{I}$ définie par

$$(f\ (\texttt{c}\ x)) = (f\ (x\ I\ (\texttt{c}\ t)))$$

On pourrait penser que l'appel récursif qui se fait sur un sous-terme x de $(\texttt{c}\ x)$ est bien fondé. Ce n'est pas le cas, on a En effet $(f\ (\texttt{c}\ t))$ se réduit en $(f\ (t\ I\ (\texttt{c}\ t)))$ qui se réduit en $(f\ (\texttt{c}\ t))$ et donc se terme ne se normalise pas.

3.3.5 Récurrence structurelle versus récurrence bien fondée

La définition primitive par point fixe permet de capturer certaines fonctions récursives mais évidemment toutes les fonctions ne suivent pas ce critère de décroissance structurel. Un exemple classique est la fonction `quicksort` sur les listes qui s'appelle récursivement sur deux sous-listes qui sont de longueur plus petite. Évidemment, on sait ramener cette décroissance à un point-fixe structurel sur un argument supplémentaire représentant la longueur de la liste. Mais on voudrait également pouvoir justifier la définition par point fixe, en utilisant uniquement un argument de bonne fondation de la relation *avoir une longueur strictement plus petite.*

Ceci est possible en Coq en utilisant notre définition d'être bien fondé.

On suppose que l'on dispose d'une fonction $\texttt{split} : A \to \texttt{list} \to \texttt{list} \times \texttt{list}$ qui sépare une liste l suivant un pivot a en deux listes l_1 et l_2 telle que la longueur de l_i est inférieure ou égale à la longueur de l. On notera $|l|$ la longueur de l, qui vérifie les propriétés :

$$|\texttt{nil}| = 0 \qquad |(\texttt{cons}\ a\ l)| = (\texttt{S}\ |l|)$$

On peut alors définir une fonction de tri $\texttt{quick} : \forall(l : \texttt{list})(n : \texttt{nat}), (|l| \leq n) \to \texttt{list}$ de manière structurelle sur n. La définition de `quick` suit le schéma suivant :

$$\begin{array}{ll}
(\texttt{quick}\ \texttt{nil}\ n) & = \lambda h : |\texttt{nil}| \leq n \Rightarrow \texttt{nil} \\
(\texttt{quick}\ (\texttt{cons}\ a\ l)\ 0) & = \lambda h : |(\texttt{cons}\ a\ l)| \leq 0 \Rightarrow \texttt{absurd}\ h_0 \\
(\texttt{quick}\ (\texttt{cons}\ a\ l)\ (\texttt{S}\ n)) & = \lambda h : |(\texttt{cons}\ a\ l)| \leq (\texttt{S}\ n) \Rightarrow \\
& \quad \mathsf{let}\ (l_1, l_2) = (\texttt{split}\ a\ l) \\
& \quad \mathsf{in}\ (\texttt{append}\ (\texttt{quick}\ l_1\ n\ h_1)\ (\texttt{cons}\ a\ (\texttt{quick}\ l_2\ n\ h_2)))
\end{array}$$

Avec h_0, h_1 et h_2 des preuves de $\bot$, $|l_1| \leq n$ et $|l_2| \leq n$. Pour obtenir une fonction de tri de type $\texttt{list} \to \texttt{list}$ il suffit de prendre une valeur initiale de n égale à la longueur de l. Cependant, d'un point de vue calculatoire, l'argument entier n qui a été ajouté apparaît superflu.

L'alternative consiste à utiliser une preuve du fait que la relation $R \equiv \lambda l, m : \texttt{list} \Rightarrow |l| < |m|$ est bien fondée.

Le programme se construit alors comme un terme `quickwf` de type :
$\forall l : \texttt{list}, (\texttt{wf}\ \texttt{list}\ R\ l) \to \texttt{list}$. Cette preuve suit le schéma suivant :

$$\begin{array}{ll}
(\texttt{quickwf}\ \texttt{nil}) & = \lambda(h : (\texttt{wf}\ \texttt{list}\ R\ \texttt{nil})) \Rightarrow \texttt{nil} \\
(\texttt{quickwf}\ (\texttt{cons}\ a\ l)) & = \lambda(h : (\texttt{wf}\ \texttt{list}\ R\ (\texttt{cons}\ a\ l))) \Rightarrow \\
& \quad \mathsf{let}\ (l_1, l_2) = (\texttt{split}\ a\ l) \\
& \quad \mathsf{in}\ (\texttt{append}\ (\texttt{quickwf}\ l_1\ h_1)\ (\texttt{cons}\ a\ (\texttt{quickwf}\ l_2\ h_2)))
\end{array}$$

Avec h_1 et h_2 des preuves de $(\texttt{wf}\ \texttt{list}\ R\ l_1)$ et $(\texttt{wf}\ \texttt{list}\ R\ l_2)$ qui sont structurellement plus petites que h. Ces preuves sont construites de la manière suivante. Il est possible de définir wf_inv de type $(A:\texttt{Set})(R:A\rightarrow A\rightarrow \texttt{Prop})(x:A)(\texttt{wf}\ A\ R\ x)\rightarrow (y:A)(R\ y\ x)\rightarrow (\texttt{wf}\ A\ R\ y)$ par :

$$(\texttt{wf\_inv}\ A\ R\ x\ (\texttt{wf\_intro}\ A\ R\ x\ h)) = h$$

On remarque que $(\texttt{wf\_inv}\ A\ R\ x\ H)$ est structurellement plus petit que H car obtenu en prenant un sous-terme structurel de H.

Pour obtenir une fonction de type list→list il suffit d'utiliser une preuve que R est bien fondée, c'est-à-dire :$(l:\texttt{list})(\texttt{wf list}\ R\ l)$.

L'avantage de cette définition est que la propriété wf peut être définie dans la sorte Prop. Donc le comportement calculatoire de la fonction quickwf à l'extraction est le comportement attendu. Par contre, il est compliqué de réduire la fonction quickwf dans Coq car cette réduction ne peut avoir lieu que si la preuve h de $(\texttt{wf list}\ R\ l)$ commence par un constructeur. Pour raisonner sur de telles fonctions, il est préférable d'établir des équations de point fixe. Cependant ces équations ne sont pas immédiates à établir. Ce problème a été étudié en détail dans la thèse de Antonia Baala. Des théorèmes dans la bibliothèque Init/Wf permettent de faciliter les constructions de fonctions par point fixe.

3.3.6 Définitions mutuellement inductives

Coq accepte de manière primitive les définitions mutuellement récursives. En fait de telles définitions peuvent simplement être codées en faisant de la définition inductive une famille de définitions inductives. On introduit un type A qui fait la somme disjointe des arités des différentes définitions inductives. Les définitions mutuellement inductives I_k seront remplacées par une famille inductive d'arité $A\rightarrow s$ dans laquelle s est la sorte des définitions inductives. On remplace dans les types de constructeur chaque mention à une des définitions mutuellement inductives $(I_k\ a_1 \dots\ a_l)$ par la référence à l'instance appropriée de I, c'est-à-dire $(I\ (\texttt{in}_k\ a_1 \dots a_l))$. Le schéma d'élimination mutuellement récursif des définitions I_k peut également se déduire du schéma général pour I.

Exemple 4
```
Inductive arbre (A:Set) : Set :=
  | node : A → (foret A) → (arbre A)
with foret (A:Set) : Set :=
  | vide : (foret A)
  | add : (arbre A) → (foret A) → (foret A).
```

est équivalent à

```
Inductive arbre_foret (A:Set) : bool → Set :=
  | node : A → (arbre_foret A false) → (arbre_foret A true)
  | vide : (arbre_foret A false)
  | add : (arbre_foret A true) → (arbre_foret A false)
           → (arbre_foret A false).
Definition arbre (A:Set) := arbre_foret A true.
Definition foret (A:Set) := arbre_foret A false.
```

Dans le cas de définitions mutuellement récursives, Coq engendre automatiquement un principe d'élimination récursive pour chaque type qui ne tient pas compte des appels aux autres inductifs de la famille. Par contre la commande Scheme permet d'engendrer automatiquement les schémas d'élimination mutuellement récursifs dans le cas de positivité non imbriquée. Ces schémas peuvent ensuite être utilisés par la tactique

Elim *term* using *theorem* with *instances* en instanciant de manière appropriée les propriétés à montrer pour les types auxiliaires.

Dans le cas de l'exemple des arbres et des forêts, la commande `Inductive` introduit en particulier le schéma suivant qui n'utilise pas la structure récursive de `foret` :

```
arbre_ind
     : ∀(A:Set)(P:(arbre A)→Prop),
       (∀(a:A)(f:foret A), P (node A a f))→∀a:(arbre A), P a
```

Pour définir `arbre_foret_ind` qui permet de prouver une propriété P sur les arbres en utilisant une propriété Q sur les forêts prouvée de manière mutuellement récursive, on utilise :

```
Scheme arbre_foret_rec := Induction for arbre Sort Prop
with foret_arbre_rec := Induction for foret Sort Prop.
```

On obtient alors :

```
arbre_foret_ind :
  ∀(A:Set)(P:(arbre A)→Prop)(Q:(foret A)→Prop),
     (∀(a:A)(f:(foret A)),(Q f)→ P (node A a f))
   →(P0 (vide A))
   →(∀a:(arbre A),(P a)→∀f:(foret A),(Q f)→Q (add A a f))
→∀a:(arbre A), P a
```

3.4 Extensions

Les définitions inductives du Calcul des Constructions Inductives permettent une représentation directe des types de données concrets et des relations inductives définies comme les plus petites propriétés satisfaisant un ensemble de conditions de stabilité.

Cependant, cette notion est insuffisante pour représenter certaines structures qui apparaissent naturellement dans les développements mathématiques et informatiques.

3.4.1 Structures infinies

Il s'agit en particulier des structures potentiellement infinies comme les streams (suites infinies), les séries mathématiques ou les expressions de processus. Il est possible d'utiliser des fonctions pour représenter de tels objets mais on aimerait une représentation plus proche de la structure concrète de ces objets. Les définitions co-inductives ont une structure duale des définitions inductives. Elles ont été ajoutées au Calcul des Constructions Inductives et intégrées à `Coq` par Eduardo Giménez. L'approche suivie avait été suggérée par Th. Coquand. Les définitions co-inductives admettent comme schéma d'élimination la définition par cas et comme règle d'introduction, en complément des constructeurs, des définitions par point fixe.

3.4.2 Structures quotients

Les définitions (co)-inductives sont des structures libres, il est donc possible de prouver que deux constructeurs d'un type inductif dans `Set` ou `Type` ont des images disjointes. Supposer qu'une équation entre constructeurs est satisfaite conduit à une théorie inconsistante. Pourtant les structures quotients sont très utilisées (représentation des rationnels, des réels ou des lambda-termes). Plusieurs solutions ont été proposées pour ajouter des types quotients à une théorie des types, en particulier par R. Backhouse, M. Hoffman, S. Boutin, G. Barthe et plus récemment par P. Courtieu. A part NuPrl, les assistants de preuve ne proposent pas de type inductif quotient. C'est à l'utilisateur de gérer à la main une égalité ad-hoc et des conditions de compatibilité.

3.4.3 Réductions généralisées

Les types inductifs de Coq permettent de définir aisément une addition sur les entiers qui vérifie les axiomes de Peano $0+x=x$ et $(S\ y)+x=(S(x+y))$ comme des règles de conversion. Par contre la propriété $x+0=x$ ou l'associativité de l'addition sont prouvables mais ne sont pas des conversions et doivent donc être traités manuellement alors que les systèmes de réécriture savent parfaitement traiter ses égalités de manière automatique. Pour pallier cette difficulté, on étudie depuis quelques années des systèmes qui combinent la réécriture et le lambda-calcul typé. Une difficulté est de garantir la confluence et terminaison du système résultant. Pour cela, il faut bien entendu restreindre la forme des réécritures applicables. Les travaux récents de F. Blanqui dans la lignée de ceux de J.-P. Jouannaud et M. Okada proposent un cadre qui capture un très large sous-ensemble des définitions inductives de Coq et permet de définir des fonctions par des systèmes de réécriture vérifiant un certain schéma. De manière alternative, J.-P. Jouannaud, A. Rubio et D. Walukiewicz-Chrząszcz proposent un ordre RPO applicable aux termes du Calcul des Constructions qui garantit la préservation par ajout de règles de réécriture.

Chapitre 4

Architecture des assistants à la démonstration

4.1 Architecture de Coq

Le système Coq repose sur un noyau de vérification du Calcul des Constructions Inductives. Les opérations de base de ce noyau consistent à ajouter une déclaration (variable, définition, déclaration inductive, module) dans l'environnement. Cela nécessite d'effectuer le typage des termes en particulier de contrôler des contraintes d'univers et de pouvoir tester la convertibilité de deux termes, ce qui passe par des étapes de réduction.

Au dessus de ce noyau, est construit un langage de description de haut niveau qui est compilé vers le CCI. Ce langage permet de laisser certaines informations implicites, par exemple d'utiliser des coercions entre différents types de données, d'utiliser les dépendances entre types pour omettre certains arguments de fonction qui seront calculés par unification. Le mécanisme de définition par filtrage est également compilé vers le filtrage atomique du CCI. On pourrait avoir des mécanismes de définition récursive de fonctions proches des langages de programmation et écrire des termes incomplets qui nécessitent des preuves complémentaires (par exemple pour traiter agréablement des fonctions partielles). La distinction de ces deux niveaux permet une souplesse d'expression sans modifier la théorie de base (en particulier sans risque de la rendre incohérente). Par contre elle introduit une distance entre ce que l'utilisateur écrit et ce que la machine prouve. Techniquement, cela complexifie l'interaction avec l'utilisateur (affichage, détection des erreurs).

Un second aspect de l'architecture de Coq est le mécanisme de construction de preuves qui se fait à l'aide de tactiques. Partant d'une propriété à montrer, on cherche à construire une preuve (dans le cas de `Coq`, un terme de ce type). Cela se fait à l'aide de tactiques qui travaillent sur un arbre de preuve dont la racine correspond à la propriété P à établir et les feuilles un ensemble de propriétés suffisantes pour prouver P. Les tactiques peuvent implanter des procédures arbitrairement complexes. Dans `Coq`, lorsque l'arbre n'a plus de feuilles, un terme de preuve est reconstruit puis vérifié par le noyau. On peut imaginer différents langages pour exprimer les étapes de preuve.

4.2 Critères de classification

Il est d'usage de classer les systèmes d'aide (ou assistants) à la démonstration selon les critères suivants [Bar81, Wieb, Wiea].

– Critère de de Bruijn
– Logique ou méta-logique
– Principe de Poincaré

– La représentation des preuves
– Développement interactif des preuves
– Degré d'automatisation

Dans la suite, on décrira ces différents critères et on analysera divers systèmes de développements de démonstration (ACL2, PVS, HOL, Isabelle, MetaPrl - anciennement NuPrl -, Mizar et Coq) à travers ces critères.

Le critère de de Bruijn Le critère de de Bruijn caractérise les systèmes dont la part dédiée à la certification de la correction des preuves est petite et bien délimitée.

Chacun des systèmes HOL, Isabelle et Coq a un « noyau » consacré à la certification des preuves et en ce sens vérifie le critère de de Bruijn. Toutefois, autant les noyaux de HOL et Isabelle restent assez petits, autant celui de Coq est assez conséquent (en particulier en raison de la gestion des types inductifs et de la réduction).

En revanche, ni Mizar, ni PVS n'ont une notion de « noyau » bien délimité. En particulier, de nouvelles méthodes de preuves peuvent être ajoutées à ces systèmes et la correction des preuves nouvellement obtenues ne dépendra que de la correction de l'implantation de la nouvelle méthode, pas d'un « noyau » stable et préalablement bien circonscrit.

Logique ou méta-logique ? Le choix des systèmes Isabelle et MetaPrl est de fournir non pas une logique mais une méta-logique (« logical framework ») permettant de déclarer les signatures et règles d'inférences de logiques arbitraires. Dans la pratique, compte tenu du développement d'une bibliothèque minimale nécessaire à toute formalisation conséquente, seules peu de logiques sont effectivement implantées dans un système basé sur une méta-logique. Ainsi, Isabelle par exemple, offre essentiellement des bibliothèques pour la logique d'ordre supérieur de Church (HOL) et la théorie des ensembles de Zermelo-Fraenkel (ZF).

Le principe de Poincaré Le principe de Poincaré [Poi02] caractérise les systèmes qui distinguent entre simples vérifications calculatoires et étapes de preuve. Poincaré prend l'exemple de la propriété $2+2=4$ qui ne se justifie pas par une preuve mais plutôt par une vérification par calcul. Des systèmes comme Coq, Isabelle, MetaPrl, PVS et HOL utilisent une *règle de conversion* qui identifie des propriétés équivalentes modulo certaines règles de calcul. Dans HOL, cette règle de conversion ne prend en compte que la β-réduction dans un lambda-calcul simplement typé alors que dans Coq, on identifie les termes modulo la ι-réduction qui permet de calculer une grande classe de fonctions récursives. Mizar par contre n'intègre pas de notion de calcul.

Le principe de Poincaré peut être implanté à des degrés très divers. Par exemple, dans le Calcul des Constructions Inductives, une preuve de $0+n=n$ relève de la simple vérification (c'est la règle de conversion) tandis qu'une preuve de $n+0=n$ nécessite une étape d'induction et des étapes de réécriture. Autrement dit, dans le Calcul des Constructions Inductives, seul un quotient relativement à une évaluation séquentielle des programmes est introduit dans la logique. On pourrait ainsi imaginer d'étendre la règle de conversion à un quotient relativement à une évaluation non déterministe des programmes. C'est ce que se propose de faire les extensions du calcul des constructions avec des règles de réécriture.

Toutefois, il existe une manière de ramener toute procédure de simplification décidable à une application de la règle de conversion et une étape de réécriture. C'est le mécanisme de *réflexion*.

La représentation des preuves Un système de preuve peut soit valider une preuve sans garder aucune trace de la vérification autre que le source de la preuve. C'est le cas de PVS et ACL2. Ceci est forcément le cas lorsqu'aucun « noyau » ne vérifie les preuves. Toutefois, un système peut vérifier le critère de de Bruijn et ne pas garder de trace des preuves. C'est le cas de

HOL qui déduit de nouveaux théorèmes à partir de règles d'inférence bien définies sans garder de trace de quelles règles ont été appliquées pour valider tel ou tel théorème.

Un système peut valider les preuves sous la forme d'un terme de preuve. C'est le cas de Coq et Isabelle.

Le développement interactif des preuves Tous les systèmes n'offrent pas un mécanisme de développement interactif de preuves. Par exemple, Mizar et ACL2 ne peuvent que vérifier une preuve dans sa globalité. En cas de non validation, l'utilisateur doit apporter des modifications globales à la preuve.

De l'autre côté, les systèmes HOL, Coq, Isabelle et PVS, héritent tous de la notion de tactique introduite dans le système LCF et offrent ainsi la possibilité de développement de preuve interactifs.

Le degré d'automatisation Le degré d'automatisation est un facteur clé dans la diffusion des assistants de preuve en dehors du milieu des logiciens. Souvent, l'automatisation, car elle applique des méthodes « de force brute » s'oppose au souci de lisibilité des preuves : un énoncé même simple aura facilement une preuve complexe, et en tout cas non directe. Une réponse à ce souci est une nouvelle fois le mécanisme de preuve par réflexion qui isole la méthode de preuve en une étape élémentaire (à l'échelle humaine) de démonstration.

PVS et ACL2 sont actuellement les systèmes les plus automatisés parmi ceux mentionnés ci-dessus. Viennent ensuite Isabelle, puis HOL et Coq.

4.3 Autres systèmes

On peut mentionner de nombreux autres systèmes dont les bibliothèques et la base d'utilisateur sont moins développées.

Lego [Pol] est une implémentation du Calcul des Constructions enrichie par des déclarations de types inductifs. Le système Plastic [Gro] est une continuation de Lego expérimentant divers mécanismes de coercions. Ces systèmes sont développés à Edimbourg et Durham au Royaume-Uni.

Alfa/Agda [Coq] est un système expérimental de développement de preuve « comme un programme »: les preuves sont des λ-termes qui sont saisis interactivement via une interface graphique. Ces systèmes sont développés à l'université de Chalmers en Suède.

PhoX [Raf] est un système basé sur l'arithmétique d'ordre 2 manipulant des programmes ML avec inférence de type polymorphe à la Milner. Son utilisation est importante en milieu enseignant. Il est développé par C. Raffali à l'université de Savoie.

Twelf [PS] est une méta-logique dotée d'un mécanisme d'unification d'ordre supérieur. Twelf est utilisé pour modéliser la sémantique de langages de programmation. Ce système est développé à Carnegie Mellon University dans l'équipe de Frank Pfenning.

4.4 Preuves par réflexion

La réflexion est une technique permettant de remplacer les étapes de preuves associées à une procédure de simplification ou de décision en la combinaison d'une unique étape de preuve et du calcul. En ce sens, la technique de réflexion suit le principe de Poincaré qui consiste à sortir du langage de preuve les méthodes de preuves qui se ramènent à un simple calcul.

4.4.1 Utilisation de preuves de décidabilité

Supposons qu'une propriété A sur un ensemble U soit décidable. On a alors une preuve `dec` de la propriété $\forall x : U.\{A\ x\} + \{\neg(A\ x)\}$. On en déduit aisément une fonction de décision booléenne : $\texttt{dec\_bool} : U \rightarrow \texttt{bool}$ et sa propriété de correction :
$\texttt{correct} : \forall x : U.(\texttt{dec\_bool}\ u) = \texttt{true} \rightarrow A\ u$.

Supposons que l'on veuille prouver la propriété $(A\ u)$ pour un terme u clos. Le terme $(\texttt{dec\_bool}\ u)$ est clos et de type `bool`, il doit s'évaluer vers la valeur `true`. Une preuve possible de $(A\ u)$ est donc :

$$\texttt{correct}\ u\ (\texttt{refl}\ \texttt{true})$$

La vérification que ce terme est de type $(A\ u)$ nécessite de typer u puis de typer $(\texttt{refl}\ \texttt{true})$ de type $(\texttt{dec\_bool}\ u) = \texttt{true}$ ce qui revient à réduire $(\texttt{dec\_bool}\ u)$ en `true`, ce qui peut nécessiter un calcul complexe.

4.4.2 Utilisation d'une structure abstraite

En général, on souhaite montrer des propriétés sur des termes qui ne sont pas forcément clos et sur lesquels il n'est pas forcément simple de construire des procédures de décision. On introduit alors une structure abstraite intermédiaire qui va représenter la syntaxe des expressions à manipuler et qui va permettre des manipulations symboliques. On a alors besoin d'une fonction d'interprétation de la syntaxe vers les propriétés à montrer.

La technique de réflexion procède comme suit:

- Définition d'une structure abstraite S pour le type de problème auquel s'adresse la méthode de décision ou de simplification considérée
- Définition d'une fonction d'interprétation $x : S \mapsto |x|$ des objets de cette structure abstraite S en une expression concrète (un terme ou une formule) du système logique
- Définition de la fonction ϕ de décision ou de simplification par calcul sur les objets de la structure abstraite
- Preuve que la méthode est valide, c'est-à-dire que $\forall s : S, |s| = |\phi(s)|$ pour une procédure de simplification sur les termes, $\forall s : S, |s| \leftrightarrow |\phi(s)|$ pour une procédure de simplification sur les propositions (sachant qu'une procédure de décision peut être vue comme une procédure de simplification renvoyant soit la proposition vraie soit la proposition fausse)

Ces bases étant posées, la simplification d'un énoncé de la forme $\psi(t)$ en l'énoncé $\psi(\phi(t))$ où t a été simplifié procède par une simple application du lemme de validité de ϕ. En effet, par applicabilité de la méthode de décision, il existe un $s : S$ tel que $|s|$ est convertible avec t, qui par le lemme de validité est égal à $|\phi(s)|$ qui lui-même est convertible en la forme simplifiée de t.

En fait, plus généralement, on considère des structures abstraites avec des variables et des fonctions d'interprétation paramétrées par une substitution de ces variables par des sous-termes non traitables par la méthode de simplification. On a alors un lemme de validité qui a la forme $\forall s : S, \forall \sigma : Var \rightarrow Term, |s|_\sigma = |\phi(s)|_\sigma$.

4.4.3 Un exemple en Coq: l'associativité de l'addition sur les entiers naturels

On considère des expressions construites à partir de l'addition (`plus`, notée $+$) sur les entiers naturels que l'on souhaite normaliser sous une forme associative à droite en supprimant les zéros (par exemple, la normalisation de `(x+u)+((y*t)+0)` est `x+(u+(y*t))`).

Construction de la structure abstraite représentant les expressions construites à partir de $+$ et 0 On prendra les entiers naturels eux-mêmes pour dénoter les variables.

```
Coq < Definition index := nat.

Coq < Inductive expr : Set :=
Coq <       | Plus : expr -> expr -> expr
Coq <       | Zero : expr
Coq <       | Var : index -> expr.
```

Construction de la fonction d'interprétation

```
Coq <     Require Import List.

Coq <     Require Import Plus.

Coq < (* Valeur par défaut de nth si la substitution n'était pas de bonne longueur *)
Coq <     Definition default := 0.

Coq < (* Fonction d'interprétation *)
Coq <     Fixpoint interp (s:list nat) (e:expr) {struct e} : nat :=
Coq <       match e with
Coq <       | Plus e1 e2 => interp s e1 + interp s e2
Coq <       | Zero => 0
Coq <       | Var i => nth i s default
Coq <       end.
```

Construction de la fonction de simplification

```
Coq < Fixpoint insere (e1 e:expr) {struct e1} : expr :=
Coq <       match e1 with
Coq <       | Plus e1 e2 => insere e1 (insere e2 e)
Coq <       | Zero => e
Coq <       | Var i => Plus e1 e
Coq <       end.

Coq < Fixpoint norm (e:expr) : expr :=
Coq <       match e with
Coq <       | Plus e1 Zero => norm e1
Coq <       | Plus e1 e2 => insere e1 (norm e2)
Coq <       | x => x
Coq <       end.
```

Construction de la preuve de correction de la simplification

```
Coq < Lemma validite_insere :
Coq <      forall (s:list nat) (e1 e2:expr),
Coq <        interp s (insere e1 e2) = interp s (Plus e1 e2).

Coq < Proof.

Coq < induction e1; intro e2; simpl; auto.

Coq < rewrite plus_assoc_reverse.

Coq < change (interp s e1_2 + interp s e2) with (interp s (Plus e1_2 e2)).

Coq < rewrite <- IHe1_2.
```

```
Coq < rewrite IHe1_1; trivial.

Coq < Qed.
```

```
Coq < Theorem validite :
Coq <      forall (s:list nat) (e:expr), interp s (norm e) = interp s e.

Coq < Proof.

Coq < induction e; simpl; auto.

Coq < destruct e2 as [e e0| | i].

Coq < rewrite validite_insere.

Coq < rewrite <- IHe2; trivial.

Coq < simpl interp; rewrite IHe1; auto.

Coq < rewrite validite_insere.

Coq < rewrite <- IHe2; trivial.

Coq < Qed.
```

Après ce travail préalable, chaque application de la méthode de simplification se déroule comme suit.

```
Coq < Variable P : nat -> Prop.

Coq < Lemma exemple : forall x y t u:nat, P (0 + x + (u + y * t)).

Coq < intros.
1 subgoal

  x : nat
  y : nat
  t : nat
  u : nat
  ============================
   P (0 + x + (u + y * t))

Coq < pose (sigma := x :: u :: y * t :: nil);
Coq < change (P (interp sigma (Plus (Plus Zero (Var 0)) (Plus (Var 1) (Var 2))))).
1 subgoal

  x : nat
  y : nat
  t : nat
  u : nat
  sigma := x :: u :: y * t :: nil : list nat
  ============================
   P (interp sigma (Plus (Plus Zero (Var 0)) (Plus (Var 1) (Var 2))))

Coq < rewrite <- validite.
1 subgoal

  x : nat
  y : nat
  t : nat
```

```
  u : nat
  sigma := x :: u :: y * t :: nil : list nat
  ============================
   P
     (interp sigma
        (norm (Plus (Plus Zero (Var 0)) (Plus (Var 1) (Var 2)))))

Coq < simpl interp; simpl norm; clear sigma.
1 subgoal

  x : nat
  y : nat
  t : nat
  u : nat
  ============================
   P (x + (u + y * t))
```

Idéalement le travail consistant à trouver s tel que $|s| = t$ devrait être automatisé. Cela peut se faire en ML ou en utilisant le langage de tactique.

Un exemple plus consistant de tactique par réflexion disponible dans Coq est la tactique `Ring`. Un model-checker utilisant des BDDs a également été construit selon ce modèle [VGLPAK00].

Il n'est pas toujours commode ni très efficace de programmer dans le langage de Coq des procédures de recherche de preuve complexes. Les outils de preuve automatique peuvent souvent être adaptés pour produire une trace de preuve. On peut alors simplement internaliser dans Coq la notion de trace et la preuve de correction de cette trace. Ainsi la preuve vérifiée par Coq effectue un calcul sur la trace et sa taille est proportionnelle à cette taille. Cette approche a été utilisée pour construire une interface entre Coq et le système de réécriture Elan, ou dans une version réflexive de la tactique de preuve en arithmétique `Omega`.

Chapitre 5

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.

5.1 Interprétation constructive des preuves

5.1.1 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
$$\vdash_C A \Leftrightarrow A^* \text{ et si } \vdash_C A \text{ alors } \vdash_I A^*$$
– 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* :
$$\text{si } \vdash_I \forall x.P(x) \vee \neg P(x) \text{ alors il existe } f \text{ fonction récursive telle que } \vdash_I f(x) = \mathtt{true} \Leftrightarrow P(x)$$
$$\text{si } \vdash_I \forall x.\exists y.P(x,y) \text{ alors il existe } f \text{ fonction récursive telle que } \vdash_I P(x,f(x))$$

Les propriétés spécifiques des preuves classiques

– Les preuves classiques peuvent donner des résultats non conformes à l'intuition de la « vérité » (tel que le célèbre théorème des buveurs : « Dans chaque bar il y a une personne telle que si cette personne boit alors tout le monde boit »).
– 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.

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.

5.1.2 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 $(c\ t_1 \ldots\ t_n)$ 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 $(c\ t_1 \ldots\ t_n)$ 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 $p(a) =$ `true` si et seulement si $P(a)$ est démontrable et $p(a) =$ `false` si et seulement si $\neg P(a)$ est démontrable.

5.1.3 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 division 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 donnée 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 $\forall x.P(x) \Rightarrow Q(x, f(x))$.

5.2 Réalisabilité

5.2.1 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 intentionnellement par une propriété de réalisabilité "x **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.

5.2.2 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, etc. 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 $\exists x.x \in \mathtt{nat} \wedge P$. 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 5.1.

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

FIG. 5.1 – 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 :

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

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 $\mathbf{t}(A)$ de ses réalisations. Dans la formule $x\ \mathbf{r}\ A$, x représente une variable de type $\mathbf{t}(A)$.

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

FIG. 5.2 – Réalisabilité modifiée

Une formule A est réalisable s'il existe un terme t de type $\mathbf{t}(A)$ 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 modifiée sert à justifier que le principe de Markov n'est pas démontrable en logique intuitionniste :

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

Évaluation Dans le cas de la réalisabilité modifiée, 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 5.2. 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 connaît 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.

5.3 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.

5.3.1 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 5.3.

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.

$\mathbf{E}(\texttt{Set})$	$= \texttt{Set}$	
$\mathbf{E}(X)$	$= X$	$X : A : \texttt{Type}$
$\mathbf{E}(x)$	$= x$	$x : A : \texttt{Set}$
$\mathbf{E}((x : A)B)$	$= \mathbf{E}(A) \to \mathbf{E}(B)$	$A, B : \texttt{Set}$
$\mathbf{E}((X : A)B)$	$= (X : \mathbf{E}(A))\mathbf{E}(B)$	$A : \texttt{Type}, B : \texttt{Set}$
$\mathbf{E}((x : A)B)$	$= \mathbf{E}(B)$	$A : \texttt{Set}, B : \texttt{Type}$
$\mathbf{E}((X : A)B)$	$= \mathbf{E}(A) \to \mathbf{E}(B)$	$A, B : \texttt{Type}$
$\mathbf{E}([x : A]t)$	$= [x : \mathbf{E}(A)]\mathbf{E}(t)$	$t : B : \texttt{Set}$ ou $A : \texttt{Type}$
$\mathbf{E}([x : A]t)$	$= \mathbf{E}(t)$	$t : B : \texttt{Type}$ et $A : \texttt{Set}$
$\mathbf{E}((t\ u))$	$= (\mathbf{E}(t)\ \mathbf{E}(u))$	$t : A : \texttt{Set}$ ou $u : B : \texttt{Type}$
$\mathbf{E}((t\ u))$	$= \mathbf{E}(t)$	$t : A : \texttt{Type}$ et $u : B : \texttt{Set}$

FIG. 5.3 – Traduction de CC vers F_ω

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 5.4. On commence par définir la formule $x\ \mathbf{r}\ P$ pour $P : \texttt{Set}$ on aura également besoin de définir $X\ \mathbf{r}\ A$ lorsque $A : \texttt{Type}$ ce qui est fait dans la figure 5.5. Dans le cas $A : \texttt{Type}$, $X\ \mathbf{r}\ A$ sera également une arité de type $\texttt{Type}$ 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 5.6 une transformation $\mathbf{R}(P)$ qui s'applique à tous les objets $P : A$ avec $A : \texttt{Type}$ et qui coïncide avec la définition du prédicat $\lambda x.x\ \mathbf{r}\ A$ quand A est $\texttt{Set}$(en particulier $\mathbf{R}(A)$ a alors le type $\mathbf{E}(A) \to \texttt{Prop}$).

$x\ \mathbf{r}\ P$	$= (\mathbf{R}(P)\ x)$	$P : \texttt{Set}$ et P n'est pas un produit
$f\ \mathbf{r}\ (x : A)B$	$= (x : \mathbf{E}(A))x\ \mathbf{r}\ A \to (f\ x)\ \mathbf{r}\ B$	$A, B : \texttt{Set}$
$f\ \mathbf{r}\ (X : A)B$	$= (X : \mathbf{E}(A))(X_r : X\ \mathbf{r}\ A)(f\ X)\ \mathbf{r}\ B$	$A : \texttt{Type}, B : \texttt{Set}$

FIG. 5.4 – Réalisabilité dans CC : $x\ \mathbf{r}\ P$ pour $P : \texttt{Set}$

$X\ \mathbf{r}\ \texttt{Set}$	$= X \to \texttt{Prop}$	
$F\ \mathbf{r}\ (x : A)B$	$= (x : \mathbf{E}(A))F\ \mathbf{r}\ B$	$A : \texttt{Set}, B : \texttt{Type}$
$F\ \mathbf{r}\ (X : A)B$	$= (X : \mathbf{E}(A))(X_r : X\ \mathbf{r}\ A)(F\ X)\ \mathbf{r}\ B$	$A, B : \texttt{Type}$

FIG. 5.5 – Réalisabilité dans CC : $x\ \mathbf{r}\ P$ pour $P : \texttt{Type}$

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

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

5.3.2 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.

$\mathbf{R}(X)$	$= X_r$	X variable de prédicat
$\mathbf{R}((P\ t))$	$= (\mathbf{R}(P)\ \mathbf{E}(t))$	$t : A : \texttt{Set}$
$\mathbf{R}((P\ T))$	$= (\mathbf{R}(P)\ \mathbf{E}(T)\ \mathbf{R}(T))$	$T : A : \texttt{Type}$
$\mathbf{R}([x : A]P)$	$= [x : \mathbf{E}(A)]\mathbf{R}(P)$	$A : \texttt{Set}$
$\mathbf{R}([X : A]P)$	$= [X : \mathbf{E}(A)][X_r : X\ \mathbf{r}\ A]\mathbf{R}(P)$	$A : \texttt{Type}$
$\mathbf{R}(P)$	$= [x : \mathbf{E}(P)]x\ \mathbf{r}\ P$	si P est un produit

FIG. 5.6 – Réalisabilité dans CC : $\mathbf{R}(P)$ pour $P : B : \texttt{Type}$

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 $A : \texttt{Prop}$ est que A est vérifiée et que la preuve t de A ne sert pas au calcul. L'interprétation de $t : A$ lorsque $A : \texttt{Set}$ est que t est une preuve calculatoire de A dont il est possible d'extraire un 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 5.7. La fonction de réalisabilité $f\ \mathbf{r}\ P$ s'applique à tous les objets de `Set` et `Type` et est définie figures 5.8, 5.9 et 5.10.

On dira que $A : \texttt{Type}_P$ 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 $\texttt{Type}_P$ qui est définie figure 5.11. 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`.

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

FIG. 5.7 – Extraction étendue à `Prop`

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

FIG. 5.8 – Réalisabilité étendue à `Prop` : $x\ \mathbf{r}\ P$ avec $P : \texttt{Set}$

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 $(A : \texttt{Set})(A \to \texttt{Prop}) \to \texttt{Set}$ et est définie par :

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

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

FIG. 5.9 – Réalisabilité étendue à Prop : $x\ \mathbf{r}\ P$ pour $P : \texttt{Type}$

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

FIG. 5.10 – Réalisabilité étendue à Prop : $\mathbf{R}(P)$ pour $P : B : \texttt{Type}$

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

FIG. 5.11 – Réalisabilité étendue à Prop : $\mathbf{L}(P)$

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

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

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

$$\begin{array}{lll}\mathbf{L}((x:A)B)=\mathbf{L}(B) & B:\texttt{Type}_P\text{ et }A:\texttt{Prop}\\ \mathbf{L}((P\ t))\quad=\mathbf{L}(P) & P:B:\texttt{Type}_P\text{ et }t:A:\texttt{Prop}\\ \mathbf{L}([x:A]P)\ =\mathbf{L}(P) & P:B:\texttt{Type}_P\text{ et }A:\texttt{Prop}\end{array}$$

Dans un tel modèle, la propriété PI

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

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

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

est démontrable.

`Prop` est inclus dans `Set` Les rôles de `Prop` et `Set` sont dissymétriques 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 élément 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 `match` 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 `match` 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 : `Type` alors on aurait $\mathbf{E}(A) =$ `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 $t : I :$ `Prop`. On étend la notion d'extraction pour que l'extraction de

$$\texttt{match}\ t\ \texttt{with}\ (\texttt{c}\ x_1 \ldots x_p) \Rightarrow p\ \texttt{end}$$

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 `Prop` $\leq$ `Type` et `Set` $\leq$ `Type` qui sont essentielles pour un développement harmonieux sont incompatibles avec une extraction incrémentale, car il existe alors des types $B :$ `Type`(n) (p.ex. $(U : \mathit{Type}(1))U \to U$) 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 $b : B :$ `Type` qui s'avèrent purement logique.

Intérêt sémantique à 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.

5.3.3 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 réalisabilité 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

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

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.

5.4 L'extraction en pratique

Le mécanisme d'extraction de `Coq` implante la fonction d'extraction $\mathbf{E}(\_)$. Il se présente sous la forme d'une commande `Extraction` qui produit du code pour plusieurs langages de programmation fonctionnels (Ocaml, Haskell et Scheme). Cette commande peut être utilisée directement dans la boucle d'interaction de `Coq` pour afficher le code extrait :

```
Coq < Extraction plus.
(** val plus : nat -> nat -> nat **)
let rec plus n m =
  match n with
    | O -> m
    | S p -> S (plus p m)
```

La commande `Recursive Extraction` aura pour effet d'extraire récursivement tout ce qui est nécessaire. On peut également utiliser la commande `Extraction` pour écrire dans un fichier tout le code correspondant à un ensemble d'objets `Coq` (le caractère récursif est alors implicite) :

```
Coq < Extraction "arith.ml" plus mult.
```

Dans le cas d'Ocaml, une interface est également créée (fichier `.mli`).

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 ont été exposés au chapitre précédent.

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. 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 $S : \tau_1 \rightarrow \tau_2 \rightarrow$ `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 $\texttt{bst}\ s$. 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` $\rightarrow$ `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 => O
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)))))
```

De nombreuses autres techniques pour définir des fonctions récursives non structurelles dans Coq sont décrites dans le chapitre 15 de l'ouvrage *Interactive Theorem Proving and Program Development* [BC04].

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 :

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

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` $\rightarrow$ `A` $\rightarrow$ `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

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

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

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` :

```
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` :

```
Module ABR (X: OrderedType).

  Inductive tree : Set :=
  | Empty
  | Node : tree -> X.t -> tree -> tree.

  Fixpoint mem (x:X.t) (s:tree) {struct s} : bool := ...

  Inductive In (x:X.t) : tree -> Prop := ...
  Hint Constructors In.

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

  (* 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).

Chapitre 7

Preuve de programmes impératifs

7.1 Logique de Hoare classique

On considère un langage PASCAL très simplifié, avec des variables globales entières, des expressions entières et booléennes, et les instructions d'affectation, de test et de boucle while :

$$\begin{array}{rcl} e & ::= & n \mid x \mid e \; op \; e \\ op & ::= & + \mid - \mid * \mid / \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid and \mid or \\ i & ::= & skip \mid x := e \mid i ; i \mid if \; e \; then \; i \; else \; i \mid while \; e \; do \; i \; done \end{array}$$

Exemple 5 *Appelons ISQRT le programme suivant, sur trois variables n, count et sum :*

```
count := 0; sum := 1;
while sum <= n do count := count + 1; sum := sum + 2 * count + 1 done
```

Affirmation : à la fin de l'exécution de ce programme, count est la racine carré de n, arrondie à l'entier inférieur.

7.1.1 Sémantique opérationnelle

Un *état* d'un programme est une table d'association E qui à chaque variable x du programme associe sa valeur courante $E(x)$. La valeur d'une expression bien typée e dans un état E est définie par

$$\begin{array}{rcl} E(n) & = & n \\ E(x) & = & E(x) \\ E(e_1 \; op \; e_2) & = & E(e_1) \; op \; E(e_2) \end{array}$$

La sémantique opérationnelle de ce langage est définie par les règles de transition (sur toute instruction bien typée) :

$$\begin{array}{rcl} E & \underset{x:=e}{\longrightarrow} & E\{x = E(e)\} \\ E_1 & \underset{i_1;i_2}{\longrightarrow} & E_3 \text{ si } E_1 \underset{i_1}{\longrightarrow} E_2 \text{ et } E_2 \underset{i_2}{\longrightarrow} E_3 \\ E_1 & \underset{if\ e\ then\ i_1\ else\ i_2}{\longrightarrow} & E_2 \text{ si } E_1(e) = true \text{ et } E_1 \underset{i_1}{\longrightarrow} E_2 \\ E_1 & \underset{if\ e\ then\ i_1\ else\ i_2}{\longrightarrow} & E_2 \text{ si } E_1(e) = false \text{ et } E_1 \underset{i_2}{\longrightarrow} E_2 \\ E_1 & \underset{while\ e\ do\ i}{\longrightarrow} & E_3 \text{ si } E_1(e) = true, E_1 \underset{i}{\longrightarrow} E_2 \text{ et } E_2 \underset{while\ e\ do\ i}{\longrightarrow} E_3 \\ E & \underset{while\ e\ do\ i}{\longrightarrow} & E \text{ si } E(e) = false \end{array}$$

7.1.2 Logique de Hoare

Un triplet de Hoare est un triplet noté $\{P\}i\{Q\}$ où P et Q sont des assertions logiques et i une instruction. Ces assertions logiques sont des formules du premier ordre, avec comme formules atomiques les expressions de notre langage. Remarque importante : il y a ainsi identification entre les variables du programme et les variables de la logique.

On dit qu'un triplet de Hoare $\{P\}i\{Q\}$ est valide si pour tous états E_1 et E_2 tels que $E_1 \underset{i}{\longrightarrow} E_2$ et P est vraie dans E_1, Q est vraie dans E_2.

Exemples de triplets valides : $\{x = 1\}x := x + 2\{x = 3\}$, $\{x = y\}x := x + y\{x = 2y\}$.

Exemple 6 *Sur le programme $ISQRT$, on souhaiterait montrer la validité du triplet*

$$\{n \geq 0\}ISQRT\{count * count \leq n \wedge n < (count + 1) * (count + 1)\}$$

Logique de Hoare : ensemble de règles de déduction sur les triplets :

$$\frac{}{\{P\}skip\{P\}} \qquad \frac{\{P \wedge e = true\}i_1\{Q\} \qquad \{P \wedge e = false\}i_2\{Q\}}{\{P\}if\ e\ then\ i_1\ else\ i_2\{Q\}}$$

$$\frac{}{\{P[x \leftarrow e]\}x := e\{P\}} \qquad \frac{\{I \wedge e = true\}i\{I\}}{\{I\}while\ e\ do\ i\{I \wedge e = false\}}$$

$$\frac{\{P\}i_1\{Q\} \qquad \{Q\}i_2\{R\}}{\{P\}i_1; i_2\{R\}} \qquad \frac{\{P'\}i\{Q'\} \qquad P \to P' \qquad Q' \to Q}{\{P\}i\{Q\}}$$

Proposition 1 *Cet ensemble de règles est correct : tout triplet dérivable est valide.*

Preuve : pas de difficulté (cf chapitre 2).

Difficulté : prouver un triplet à partir de ces règles demande de « deviner » les bonnes annotations intermédiaires, par exemple pour la séquence, mais aussi pour la règle d'affaiblissement. Ainsi, on ne peut pas prouver le programme de la racine carrée sans réfléchir : il faut en particulier trouver un invariant de boucle adéquat. L'équivalent, du point de vue théorique, de cette difficulté est : a-t-on complétude de la logique de Hoare, c.-à-d. peut-on prouver tous les triplets valides ?

7.1.3 Complétude, et calcul de plus faible précondition

Pour i et Q fixés, l'ensemble des P tels que $\{P\}i\{Q\}$ est valide, s'il est non vide, possède-t-il un élément *minimal* P_0 au sens où pour tout P tels que $\{P\}i\{Q\}$ est valide, P implique P_0.

Calcul de WP :

$$\begin{array}{rcl}
WP(x := e, Q) & = & Q\{x \leftarrow e\} \\
WP(i_1; i_2, Q) & = & WP(i_1, WP(i_2, Q)) \\
WP(if\ e\ then\ i_1\ else\ i_2, Q) & = & (e = true \to WP(i_1, Q)) \wedge (e = false \to WP(i_2, Q)) \\
WP(while\ e\ do\ i, Q) & = & \text{pas de formule simple !}
\end{array}$$

Exemple 7 $WP(x := x + y, x = 2y) \equiv x + y = 2y$

Proposition 2 *L'ensemble des règles de logique de Hoare est* relativement *complet : Tout triplet valide $\{P\}i\{Q\}$ est dérivable (en particulier, on peut trouver des invariants pour les boucles while de i).*

Preuve : le *relativement* exprime ici une hypothèse qui est que la logique dans laquelle on exprime les annotations est suffisamment expressive, en particulier pour exprimer les invariants de boucle nécessaires à l'aide de point-fixe [Cou90].

```
let isqrt = fun (n : int) ->
  { n >= 0 }     (* pré-condition *)
  begin
    let count = ref 0 in
    let sum = ref 1 in
    begin
    while !sum <= n do
      { invariant count >= 0 and n >= count*count and sum = (count+1)*(count+1)
        variant   n - sum }  (* invariant et variant de boucle *)
      count := !count + 1;
      sum := !sum + 2 * !count + 1
    done;
   !count
    end
  end
  { result >= 0 and result * result <= n and n < ( result+1)*( result+1) }
    (* post-condition *)
```

FIG. 7.1 – Calcul de la racine carrée en Why

7.1.4 Difficultés

De nombreux travaux ont fait suite à la logique de Hoare originale [Cou90], pour étendre le formalisme et résoudre des difficultés, par exemple :

- Référer, dans une post-condition Q de $\{P\}i\{Q\}$, à la valeur d'une variable avant l'exécution de i.
- Avoir des effets de bord dans les expressions.
- Traiter l'appel de sous-programmes.
- Prouver la terminaison des programmes (terminaison des boucles while).
- Supporter les `break`, `continue`, les exceptions.
- Avoir des structures de données complexes : tableaux, structures, pointeurs, objets, etc.
- Travailler sur des entiers bornés, et vérifier le non-débordement des opérations sur les entiers.

7.2 Transformation fonctionnelle : la méthode Why

Objectif général : établir la validité d'un triplet de Hoare, non pas avec les règles de déduction précédente, mais avec une technique fondée sur le calcul des constructions inductives. Cet objectif a été développé par Jean-Christophe Filliâtre [Fil03a] et implantée dans un outil logiciel appelée Why [Fil03b]. Les programmes traités par Why ne sont pas en PASCAL, ni en C ou autre langage de programmation existant, mais dans une syntaxe spécifique, qui a été conçue pour la preuve de ces programmes.

En Why, toutes les difficultés mentionnées ci-dessus sont traitées, exceptés la possibilité d'avoir des structures de données complexes et les entiers bornés. Dans la section suivante, nous verrons à la fois comment traiter les structures de données complexes, et comment faire des preuves sur des programmes écrits dans des langages de programmation impératifs standards : Java ou C en l'occurrence.

La figure 7.1 donne de nouveau le programme qui calcule la racine carrée, écrit cette fois sous forme d'une fonction Why. On remarque que :

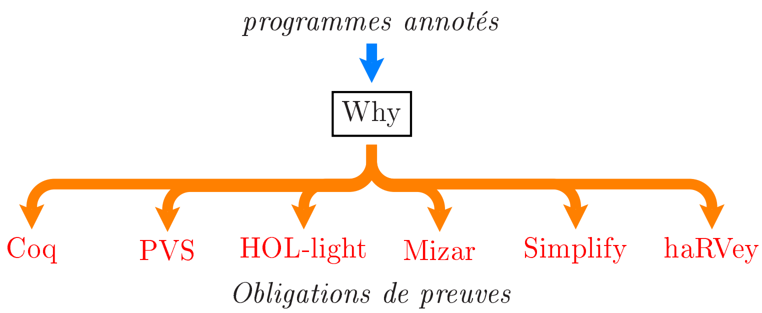

FIG. 7.2 – Approche multi-prouveur de Why

- Le langage Why est un langage proche de Caml
- En particulier, il n'y a pas de distinction entre instruction et expression. Les triplets à la Hoare s'applique sur des expressions, et dans une post-condition on peut utiliser le mot clé **result** pour parler du résultat de l'expression.
- L'ajout d'un variant, pour garantir la terminaison

Le travail de l'outil Why consiste à produire, à partir d'un programme annoté, des *obligations de preuves* dont la validité assure la correction du programme. Avec Why, ces obligations de preuves sont des formules en logique du premier ordre, exprimable dans la syntaxe de différents démonstrateurs existants, aussi bien des démonstrateurs interactifs comme Coq, PVS, HOL-light ou Mizar, que des démonstrateurs automatiques comme Simplify ou haRVey.

Néanmoins, la sortie pour Coq possède une particularité supplémentaire : la *validation*, qui est un programme fonctionnel Coq équivalent au programme de départ. Il s'agit là de l'idée directrice de l'approche Why : un programme impératif peut être traduit en un programme fonctionnel par ajout de paramètre. Sur l'exemple de ISQRT, on écrirait quelque chose comme

```
isqrt(n) = isqrt2(n,0,1)
isqrt2(n,count,sum) =
  if sum<=n then isqrt2(n,count+1,sum + 2 * (count+1)+1) else count
```

On est ramené alors à la preuve d'un programme fonctionnel. Il s'agit là d'une instance de l'approche par plongement superficiel (*shallow embedding* en anglais) opposé à plongement profond (*deep embedding*), tel qu'expliqué dans le chapitre 2.

7.2.1 Le langage Why

C'est un langage fonctionnel auquel sont rajoutés quelques traits impératifs. Les types de base sont : `int`, `bool`, `float` et `unit` (habité par la constante `void`). Comme d'habitude avec les langages fonctionnels, il n'y a pas de distinction syntaxiques entre expressions et instructions : les instructions sont les expressions de type `unit`.

Le noyau fonctionnel est constitué des expressions suivantes :

- les constantes entières, booléennes, flottantes et `void` ;
- les variables ;
- l'application, notée sous forme curryfiée ($e\ e_1\ \ldots\ e_n$) ;
- les opérations primitives (unaires et binaires) $+$, -, etc. ;
- les définitions locales `let` $v = e_1$ `in` e_2 ;
- les conditionnelles `if` e_1 `then` e_2 `else` e_3 ;
- l'abstraction notée `fun` $(x_1 : t_1)\ \cdots\ (x_n : t_n) \rightarrow e$.

Les traits impératifs sont introduits par

- les définitions de références locales `let` $v =$ `ref` e_1 `in` e_2 ;

- la déréférenciation $!v$;
- l'affectation : $v := e$;
- la séquence $e_1; ...; e_n$;
- la boucle `while` e_1 `do` e_2 `done` ;

Remarque : pour simplifier on ne traite pas les définitions récursives ici, ni la levée et le rattrapage d'exception. Le langage complet est décrit dans le manuel utilisateur.

En Why, toute expression e peut être annotée, avec la notation des triplets de Hoare $\{pre\}\ e\ \{post\}$. Les boucles *while* doivent être annotée par

while e do { invariant inv variant var } e done

pour spécifier un invariant de boucle, ainsi qu'un *variant*, une expression censée décroitre à chaque itération de la boucle, assurant ainsi la terminaison. Les annotations *pre*, *post* et *inv* sont des formules de logique du premier ordre, leur variables sont les variables du programme. De plus, dans les *post* on peut utiliser le mot-clé result pour référer à la valeur résultat de l'expression, et v@ pour désigner la valeur de v avant l'exécution de l'expression.

7.2.2 Typage avec effets

Comme un langage de programmation classique, le langage Why possède des règles de typage, et un programme doit être correctement typé pour que la génération des obligations de preuves puissent se faire. Comme ce langage est proche de Caml, le type en est assez proche aussi, mais on lui ajoute une spécificité qui est le typage des effets, qui consiste à préciser quelles sont les références qui sont lues et celles qui sont écrites.

Voici un exemple de programme très simple, qui va nous servir pour illustrer les mécanismes de Why :

```
parameter montant : int ref

let crediter = fun (s:int) ->
  { s >= 0 }
  montant := !montant + s
  { montant = s + montant@ }
```

En tant que programme « Caml », on sait très bien inférer que `crediter` a le type `int -> unit`. Ce qu'on va faire en plus c'est de calculer ses effets, pour déterminer dans cet exemple que la référence `montant` est à la fois lue et écrite.

Un *type avec effet* est un triplet (type, variables lues, variables écrites). Voici un extrait des règles de typage de Why (cf [Fil03a] pour le reste).

$$\frac{x : t \in \Gamma \qquad t \text{ non ref}}{\gamma \vdash x : (t, \emptyset, \emptyset)} \qquad \frac{x : t\ ref \in \Gamma}{\Gamma \vdash !x : (t, \{x\}, \emptyset)}$$

$$\frac{\Gamma \vdash e_1 : (t_2 \rightarrow (t_1, R, W), R_1, W_1) \qquad \Gamma \vdash e_2 : (t_2, R_2, W_2) \qquad t_2 \text{ non ref}}{\Gamma \vdash (e_1\ e_2) : (t_1, R_1 \cup R_2 \cup R, W_1 \cup W_2 \cup W)}$$

$$\frac{\Gamma, x : t \vdash e : (t', R, W)}{\Gamma \vdash fun(x : t) \rightarrow e : (t \rightarrow (t', R, W), \emptyset, \emptyset)}$$

$$\frac{x : t\ ref \in \Gamma \qquad \Gamma \vdash e : (t, R, W)}{\Gamma \vdash x := e : (unit, \{x\} \cup R, \{x\} \cup W)}$$

La règle de typage de l'application ci-dessus interdit d'appliquer une fonction à une référence. Une règle spéciale existe dans ce cas :

$$\frac{\Gamma \vdash e_1 : (t_2\ ref \rightarrow (t_1, R, W), R_1, W_1) \qquad \Gamma \vdash r : (t_2\ ref, R_2, W_2) \qquad r \notin (t_1, R, W)}{\Gamma \vdash (e_1\ r) : (t_1[x \leftarrow r], R_1 \cup R_2 \cup R[x \leftarrow r], W_1 \cup W_2 \cup W[x \leftarrow r])}$$

Il est fondamental de remarquer que cette règle interdit les *alias*, c.-à-d. de référencer la même chose avec deux noms différents. C'est une condition essentielle pour garantir la correction de la méthode Why. Voici un exemple typique :

```
let incr2 = fun (x:int ref) (y:int ref) ->
  { true } begin x := !x + 1; y := !y + 1 end { x = x@ + 1 and y = y@ + 1 }

parameter t : int ref

let test = { true } (incr2 t t) { t = t@ + 1 }
```

Why signale une erreur de typage sur l'application `(incr2 t t)`. Si c'était accepté, alors au vu de la post-condition de `incr2`, la post-condition `t = t@ + 1` serait prouvable, or elle est fausse, c'est `t=t@ + 2` qui est vrai.

Exercice 2 *Dériver le jugement de typage*

$$montant : int\ ref \vdash crediter : (int \rightarrow (unit, \{montant\}, \{montant\}), \emptyset, \emptyset))$$

On peut utiliser cette approche de typage avec effets pour définir des programmes Why de façon modulaire : un programme peut très faire appel à une autre fonction dont le code n'est pas donné, cette fonction doit seulement être spécifiée par son type avec effets, ainsi qu'avec un pré et une post-condition. Par exemple on peut écrire :

```
parameter montant : int ref

parameter crediter : s:int ->
  { s >= 0 }
  unit reads montant writes montant
  { montant = s + montant@ }

let test = fun (tt:unit) ->
  { true }
  begin
    (crediter 50); (crediter 80)
  end
  { montant = montant@ + 130 }
```

7.2.3 Calcul de plus faible précondition

Avec l'inférence de type avec effets, on est capable d'associer à chaque sous-expression d'un programme un type avec effets. L'étape suivante consiste à leur associer aussi une pré et une post-condition : de deux choses l'une : soit l'expression considérée est déjà annotée (une boucle while, un identificateur de fonction, ou une expression explicitement annotée par l'utilisateur), soit on lui détermine une annotation par calcul de plus faible pré-condition, d'une manière très similaire à celui de la logique de Hoare classique, mais adaptée au langage Why.

Voici un extrait des formules de calcul de WP, les règles complètes sont dans le manuel de Why [Fil02] et dans [Fil99] .

$$\begin{array}{rcl}
WP(x, Q) & = & Q[result \leftarrow x] \\
WP(!x, Q) & = & Q[result \leftarrow x] \\
WP(x := e, Q) & = & WP(e, Q[result \leftarrow tt, x \leftarrow result, x@ \leftarrow x]) \\
WP(e_1; e_2, Q) & = & WP(e_1, WP(e_2, Q)) \\
WP(if\ e_1\ then\ e_2\ else\ e_3, Q) & = & WP(e_1, if\ result\ then\ WP(e_2, Q)\ else\ WP(e_3, Q) \\
WP(let\ x = e_1\ in\ e_2, Q) & = & WP(e_1, WP(e_2, Q)[x \leftarrow result])
\end{array}$$

La règle pour l'application est l'une des plus complexe :

$$\begin{array}{rl}
WP((f\ e_1\ \cdots\ e_n), Q) = & WP(e_1, WP(e_2, \ldots, WP(e_n, \\
& (P_f \wedge \forall w_1, \ldots, w_k, result, Q_f \rightarrow Q)[x_n \leftarrow result]) \ldots)[x_1 \leftarrow result])
\end{array}$$

si f a le type avec effets annoté $x_1 : t_1 \cdots x_n : t_n \rightarrow \{P_f\}(t, R, W)\{Q_f\}$, et les expressions $e_1, \ldots, e_n$ sont pures, c.-à-d. ne modifient aucune variable. Cette dernière condition fait qu'il peut arriver que l'outil Why peut parfois ne pas parvenir à générer les obligations de preuves, auquel cas il faut simplifier l'application concernée en introduisant des `let`...`in`...

Exercice 3 *Annoter le code de* `crediter`.

7.2.4 Traduction fonctionnelle

Il s'agit maintenant du cœur même de la méthode Why. Nous en donnons ici un aperçu, les détails étant dans[Fil99, Fil03a].

Le but est d'engendrer un programme fonctionnel Coq équivalent au programme Why complètement annoté. Ce programme comportera des « trous » pour les preuves des annotations logiques : les obligations de preuve. Celle-ci peuvent naturellement être prouvées dans Coq, pour obtenir alors un programme fonctionnel Coq certifié, équivalent au programme de départ, que l'on appelle la *validation* Why.

Interprétation fonctionnelle des types avec effets

Tout type avec effets annoté $T = \{P\}(t, \vec{r}, \vec{w})\{Q\}$ est interprété en un type Coq $\overline{T}$:

$$\forall \vec{x}, P(\vec{x}) \rightarrow \exists \vec{y}, \exists r, Q(\vec{x}, \vec{y}, r)$$

La notation $P(\vec{x})$ désigne la formule P où les occurrences des variables lues $\vec{r}$ sont substituées par les $\vec{x}$, et la notation $Q(\vec{x}, \vec{y}, r)$ désigne la formule Q où, pour chaque variable v de $\vec{w}$, les occurrences de v sont remplacées par le y correspondant, les $v@$ par le x correspondant, pour chaque variable v de $\vec{r}$ qui n'est pas dans $\vec{w}$, chaque occurrence de v est remplacée par le x correspondant, et enfin `result` est remplacé par r. Il s'agit là d'exprimer le fait que les x désignent les anciennes valeurs des variables modifiables, alors que les y désignent les nouvelles valeurs.

En pratique, le $\exists$ est utilisé est celui dans Set, noté avec des accolades. Par exemple, le type avec effet de `crediter` :

$$s : int \rightarrow \{s >= 0\}(unit, \{montant\}, \{montant\})\{montant = s + montant@\}), \emptyset, \emptyset)$$

est interprété par le type Coq

```
forall (s: Z), forall (montant: Z), forall (H: s >= 0),
  { montant0: Z, result: unit | montant0 = s + montant }
```

Notons également que les types de base `int`, `bool` et `float` sont interprétés par leurs représentations mathématiques Coq (`Z`, `bool` et `R`) et donc en particulier les entiers ne sont pas bornés.

Interprétation fonctionnelle des programmes

On donne maintenant des règles de traduction d'une expression e de type avec effets $T = \{P\}(t, \{r_1, \ldots, r_k\}, \{w_1, \ldots, w_l\})\{Q\}$ dans un environnement Γ, en un terme Coq à trous $\overline{e}$ de type $\overline{T}$ dans l'environnement $\overline{\Gamma}$ obtenu en enlevant toutes les variables de type ref.

- si $e \equiv x$ variable :
$$\overline{e} = \lambda \vec{x_0}, \lambda p : P(\vec{x_0}), (\emptyset, x, ?_1)$$
où $?_1 : Q(\vec{x_0}, \emptyset, x)$
- si $e \equiv !x$:
$$\overline{e} = \lambda \vec{x_0}, \lambda p : P(\vec{x_0}).(\emptyset, x_{0,i}, ?_1)$$
où $x_{0,i}$ est la variable correspondant à x, et $?_1 : Q(\vec{x_0}, \emptyset, x_{0,i})$
- si $e \equiv x := e_1$, où e_1 a le type $\{P_1\}(t_1, \vec{r_1}, \vec{w_1})\{Q_1\}$
$$\overline{e} = \lambda \vec{x_0}, \lambda p : P(\vec{x_0}), let(\vec{x_1}, v, q_1) = (\overline{e_1}\ \vec{x_0}\ ?_1)\ in\ (\vec{x_1} \oplus \{x \leftarrow v\}, tt, ?_2)$$
où $?_1 : P_1(\vec{x_0})$ et $?_2 : Q(\vec{x_0}, \vec{x_1} \oplus \{x \leftarrow v\}, tt)$, où $\vec{x_1} \oplus \{x \leftarrow v\}$ désigne le vecteur de variables $\vec{x_1}$ où la variable $x_{1,i}$ correspondant à x est remplacée par v.
- si $e \equiv fun(x : t_1) \rightarrow e_1$ et t_1 n'est pas un type ref :
$$\overline{e} = \lambda \vec{x_0}, \lambda p : P(\vec{x_0}), (\emptyset, \lambda x.\overline{e_1}, ?_1)$$
où $?_1 : Q(\vec{x_0}, \emptyset, \lambda x.\overline{e_1})$.
- si $e \equiv (e_2\ e_1)$: alors e_2 a un type avec effets annoté de la forme
$$\{P_2\}(x : t_1 \rightarrow \{P'\}(t, R', W')\{Q'\}, R_2, W_2)\{Q_2\}$$
on distingue deux cas suivant e_1 :
  - t_1 non ref, e_1 a le type $\{P_1\}(t_1, R_1, W_1)\{Q_1\}$:
$$\begin{aligned}\overline{e} = \lambda \vec{x_0}, \lambda p : P(\vec{x}),\ & let\ (\vec{x_1}, a, q_1) = (\overline{e_1}\ \vec{x_0}\ ?_1)\ in \\ & let\ (\vec{x_2}, f, q_2) = (\overline{e_2}\ (\vec{x_0} \oplus \vec{x_1})\ ?_2)\ in \\ & let\ (\vec{x_3}, v, q') = (f\ a\ (\vec{x_0} \oplus \vec{x_1} \oplus \vec{x_2})\ ?_3)\ in \\ & (\vec{x_1} \oplus \vec{x_2} \oplus \vec{x_3}, v, ?_4)\end{aligned}$$
où $?_1 : P_1(\vec{x_0})$, $?_2 : P_2(\vec{x_0} \oplus \vec{x_1})$, $?_3 : P'(\vec{x_0} \oplus \vec{x_1} \oplus \vec{x_2})$ et $?_4 : Q(\vec{x_0}, \vec{x_1} \oplus \vec{x_2} \oplus \vec{x_3}, v)$, et la notation $\vec{x} \oplus \vec{y}$ désigne le vecteur de variables obtenus en ajoutant/surchargeant les variables de $\vec{y}$ aux variables de $\vec{x}$.
  - $t_1 = t_1'\ ref$, alors e_1 est une variable r et
$$\begin{aligned}\overline{e} = \lambda \vec{x_0}, \lambda p : P(\vec{x_0}),\ & let\ (\vec{x_1}, f, q_2) = (\overline{e_2}\ \vec{x_0}\ ?_1)\ in \\ & let\ (\vec{x_2}, v, q') = (f\ r\ (\vec{x_0} \oplus \vec{x_1})\ ?_2\ in \\ & (\vec{x_1} \oplus \vec{x_2}, v, ?_3)\end{aligned}$$
où $?_1 : P_2(\vec{x_0}$, $?_2 : P'(\vec{x_0} \oplus \vec{x_1})$ et $?_3 : Q(\vec{x_0}, \vec{x_1} \oplus \vec{x_2}, v)$.

Remarques : les interprétations de l'application évaluent les arguments de l'application de droite à gauche. L'interprétation des boucles while [Fil99, Fil03a] utilise `well_founded_induction` tel qu'expliqué au chapitre 6.

Exemple 8 *La validation à trous de* `crediter` *est*

```
Definition crediter :=
  (fun (s: Z) (montant: Z) (Pre: s >= 0) =>
    let (result1, Post) :=
      exist (fun (result: Z) => result = (s + montant))
            (montant + s) (PO1 s montant Pre)
      (* : { result: Z | result = s + montant } *)
```

```
    in
      exist_2 (fun (montant1: Z) (result0: unit) => montant1 = s + montant)
              result1 tt Post)
      (* : { result1: Z, result2:unit | result1 = s + montant } *)
```

où

```
PO1 (s:Z) (montant:Z) (Pre:s>=0) : montant + s = s + montant
```

Proposition 3 (Correction [Fil99, Fil03a]) *Pour toute expression e typée avec effets annotée, si les obligations de preuve de $\overline{e}$ sont prouvables, alors e vérifie sa spécification (c.à-d. que sa post-condition est valide quand sa pré-condition est satisfaite).*

Pour la preuve de cette proposition, il faut définir formellement la sémantique de Why [Fil99, Fil03a].

7.3 Traitement des structures données complexes et application à d'autres langages de programmation

Why ne gère pas explicitement de structures de données complexes. Néanmoins, il a une approche modulaire, dans le sens où de même qu'il accepte des programmes en paramètre avec uniquement leurs spécifications, il accepte des types abstraits, et des prédicats et fonctions logiques abstraites sur ces types.

7.3.1 Exemple d'un programme avec un tableau : le drapeau hollandais

Il s'agit d'un exemple célèbre : un programme de tri linéaire quand il n'y a que trois valeurs (comme les couleurs du drapeau hollandais !) dû à Dijkstra [Dij76].

On introduit un type abstrait Why avec des axiomes :

```
parameter BLUE, WHITE, RED : color
logic iscolor : color -> prop
axiom color_elim : forall c:color. iscolor(c) -> c=BLUE or c=WHITE or c=RED
```

puis un premier programme Why uniquement spécifié, pour le test d'égalité des couleurs :

```
parameter eq_color : c1:color -> c2:color ->
  { } bool { if result then c1=c2 else c1<>c2 }
```

On introduit ensuite une axiomatisation des tableaux : on les représente en logique comme des tableaux « fonctionnels », c.-à-c. où la mise à jour d'une case de tableau retourne un nouveau tableau :

```
logic acc : colorarray, int -> color  (* acc(t,i) represente t[i] *)
logic length : colorarray -> int      (* longueur d'un tableau    *)
logic update : colorarray, int, color -> colorarray (* mise à jour t[i] := c *)
axiom length_pos: forall t:colorarray. 0 <= length(t)

axiom length_up:
  forall t:colorarray.
    forall i:int. forall v:color.
       length(update(t,i,v)) = length (t)
```

```
axiom acc_up_eq :
  forall t:colorarray.
    forall i:int. forall v:color.
       acc(update(t,i,v),i) = v

axiom acc_up_neq :
  forall t:colorarray.
    forall i:int. forall j:int. forall v:color.
       i <> j -> acc(update(t,i,v),j) = acc(t,j)
```

et on a les programme WHY suivant sur les tableaux :

```
parameter length_ : t:colorarray -> { } int { result = length(t) }

parameter acc_ : t:colorarray -> i:int ->
  { 0 <= i < length(t) }
  color
  { result=acc(t,i) }

parameter update_ : t:colorarray ref -> i:int -> v:color ->
  { 0 <= i < length(t) }
  unit reads t writes t
  { t = update(t@,i,v) }
```

noter les préconditions sur les bornes des tableaux utilisés.

On peut maintenant commencer par un petit programme qui echange deux éléments d'un tableau :

```
let swap = fun (t : colorarray ref) (i:int) (j:int) ->
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let ti = (acc_ !t i) in
  let tj = (acc_ !t j) in
  begin
  (update_ t i tj);
  (update_ t j ti)
  end
  { length(t) = length(t@) and
    acc(t,i) = acc(t@,j) and
    acc(t,j) = acc(t@,i) and
    forall k:int. i <> k and j <> k -> acc(t,k) = acc(t@,k) }
```

Pour écrire le programme de tri de Dijkstra, on introduit un predicat `monochrome` :

```
logic monochrome : colorarray, int, int, color -> prop

axiom mon1 :
  forall t:colorarray.
    forall i:int. forall j:int. forall c:color.
      monochrome(t,i,j,c) -> (forall k:int. i <= k < j -> acc(t,k)=c)
axiom mon2 :
  forall t:colorarray.
    forall i:int. forall j:int. forall c:color.
      (forall k:int. i <= k < j -> acc(t,k)=c) -> monochrome(t,i,j,c)
```

et le programme principal est alors

```
let flag = fun (t : colorarray ref) ->
  { forall k:int. 0 <= k < length(t) -> iscolor(acc(t,k)) }
  begin
  let b = ref 0 in
  let i = ref 0 in
  let r = ref (length_ !t) in
  while !i < !r do
   { invariant
       (forall k:int. 0 <= k < length(t) -> iscolor(acc(t,k)))
       and 0 <= b and b <= i and i <= r and r <= length(t)
       and monochrome(t, 0, b, BLUE)
       and monochrome(t, b, i, WHITE)
       and monochrome(t, r, length(t), RED)
     variant r - i }
    let c = (acc_ !t !i) in
    if (eq_color c BLUE)
    then
      begin (swap t !b !i); b := !b + 1; i := !i + 1 end
    else
     if (eq_color c WHITE)
     then i := !i + 1
     else
       begin r := !r - 1; (swap t !r !i) end
   done
  end
  { exists r:int. exists b:int.
          monochrome(t, 0, b, BLUE)
       and monochrome(t, b, r, WHITE)
       and monochrome(t, r, length(t), RED) }
```

Notons que les onze obligations de preuves engendrées pour ces deux programmes (`swap` et `flag`) sont prouvées entièrement automatiquement par Simplify.

7.3.2 Programmes Java et C

L'approche ci-dessus pour les tableaux simples peut être étendue à toute structure de données complexes, ce qui permet de traiter des « vrais » langages comme Java ou C : pour ceux-ci, la structure de la mémoire est modélisée avec les opérations logiques abstraites et des axiomes. L'outil Krakatoa [MPMU04] suit ce principe, et permet de traduire automatiquement un programme Java en un programme Why avec une telle modélisation. L'outil Caduceus, en cours de développement, fait de même avec les programmes C.

Remarquons aussi que cette approche pour modéliser les structures de données complexes peut être aussi utilisée pour traiter les entiers bornés : il suffit d'interpréter les opérations arithmétiques comme l'addition par une autre fonction logique. On a même le choix de permettre les débordements et faire des calculs modulo 2^{32}, ou bien imposer le non-débordement, en introduisant une fonction d'addition sur les entiers non bornés mais avec précondition de la forme

```
parameter bounded_add : x:int -> y:int ->
  { -2^31 <= x+y < 2^31 } int { result = x+y }
```

Bibliographie

[Arg98] Pablo Argon. *Etude sur l'application de méthodes formelles à la compilation et à la validation de programmes* ELECTRE. Thèse de doctorat, École Centrale de Nantes, 1998.

[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.

[Bar81] H.P. Barendregt. *The Lambda Calculus its Syntax and Semantics*. North-Holland, 1981.

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

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004. `http://www.labri.fr/Perso/~casteran/CoqArt/index.html`.

[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.

[Ber98] Y. Bertot. A certified compiler for an imperative language. Rapport de Recherche RR-34-88, INRIA, 1998.

[CDDK86] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ml. Rapport de Recherche 529, INRIA, May 1986.

[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.

[Coq] C. Coquand. Agda. `http://www.cs.chalmers.se/~catarina/agda/`.

[Cou90] Patrick Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 841–993. North-Holland, 1990.

[CPM90] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*. Springer-Verlag, 1990. LNCS 417.

[Dij76] Edsger W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

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

[ea99] B. Barras et. al. The Coq Proof Assistant User's Guide Version 6.2. Rapport Technique draft, INRIA, January 1999.

[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/`.

[Fil03a] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[Fil03b] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003. `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.

[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/`.

[Gro] Computer Assisted Reasoning Group. The PLASTIC proof assistant. `http://www.dur.ac.uk/CARG/plastic.html`.

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

[Kle98] Th. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs.* PhD thesis, Edinburgh-LFCS-Technical Report ECS-LFCS-98-392, 1998.

[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.

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.

[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.

[Poi02] H. Poincaré. *La Science et l'Hypothèse.* Flammarion, 1902.

[Pol] R. Pollack. The LEGO proof assistant. `http://www.dcs.ed.ac.uk/home/lego/`.

[PS] F. Pfenning and C. Schürmann. The Twelf project. `http://www-2.cs.cmu.edu/~twelf/`.

[Raf] C. Raffalli. The PhoX proof assistant. `http://www.lama.univ-savoie.fr/sitelama/Membres/pages_web/RAFFALLI/af2.%html`.

[Sch97] Th. Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 697–711, 1997.

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

[Ter92] D. Terrasse. Traduction de TYPOL en COQ. Application à Mini ML. Rapport de dea, IARFA, September 1992.

[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.

[VGLPAK00] K. Neeraj Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN'2000*, volume 1961 of *LNCS*, pages 162–181, 2000.

[Wiea] F. Wiedijk. Comparing mathematical provers. `http://www.cs.kun.nl/~freek/comparison/index.html`.

[Wieb] F. Wiedijk. The fifteen provers of the world. `http://www.cs.kun.nl/~freek/comparison/index.html`.