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

Calcul des Constructions Inductives

Christine Paulin-Mohring, Benjamin Werner,

Bruno Barras, Hugo Herbelin,

Jean-Christophe Filliâtre, Claude Marché

Table des matières

Chapitre 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 <  | 0 => 0
Coq <  | 1 => 0
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 $\to$ 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 $\to$ 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 $\to$ 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 :

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

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.

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

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.

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

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

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 $\exists b:\texttt{bool}.v = (\texttt{Bool}\ b)$ (défini par le prédicat `valbool`) et (`compat Snat` v) soit équivalent à $\exists n:\texttt{nat}.v = (\texttt{Nat}\ n)$ (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 26 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 $A \to A \to$ bool et démontrer que c'est la fonction caractéristique de l'égalité $(f\ x\ y) =$ `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

Travaux pratiques
Le damier de Cachan

3.1 Le problème

On dispose d'un damier de 3 cases sur 3, sur lequel sont posés 9 jetons bicolores (une face blanche et une face noire). À chaque étape on peut soit retourner les pions d'une ligne, soit retourner les pions d'une colonne. On cherche à savoir si certaines configurations sont ou non atteignables à partir d'une configuration initiale.

3.2 Modélisation du damier

3.2.1 Les pions

- En s'inspirant du type `bool` (essayer `Print bool.`), définir un type `color` ayant exactement deux valeurs `White` et `Black`
- Définir par cas la fonction `turn_color` de type `color` → `color` qui inverse les couleurs.
- Tester la fonction définie :
  `Eval compute in (turn_color White)`

3.2.2 Triplets polymorphes

La modélisation du damier utilise un type polymorphe de triplets.

- Ouvrir une section (`Section` *nom_de_la_section*) dans laquelle on introduit une variable `M:Set`.
- Définir le type `triple` des triplets d'objets de type `M` avec un constructeur : `Triple` de type `M` → `M` → `M` → `triple`.
  On pourra s'inspirer du type prédéfini `prod` des produits binaires.
- La fonction `triple_x` qui à `x:M` associe le triplet formé de trois `x` s'écrit :
  `Definition triple_x (x:M) : triple := (Triple x x x).`

3.2.3 Fonctions sur les triplets

Pour accéder aux composantes du triplet on définit un type :
`Inductive pos : Set := A : pos | B : pos | C : pos.`

- Définir `proj_triple:pos` → `triple` → `M` qui renvoie la valeur du triplet en la position donnée.

On se donne maintenant une fonction `f` de type `M`→`M`.

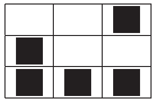
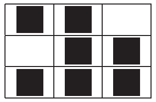
- Définir `triple_map:triple` → `triple` qui applique la fonction `f` à chaque composante du triplet.
- Définir `triple_map_select:pos` → `triple` → `triple` qui applique la fonction `f` à la composante désignée du triplet.

Fermer la section sur les triplets (`End` *nom_de_la_section*). Observer le type des objets définis.

3.2.4 Modéliser le damier

- Définir le type `damier` comme étant un triplet de triplets d'objets de type `color`.
- Définir `white_board` le damier dont tous les pions sont blancs.

On introduit deux configurations :

`start` =    `target` = 

3.3 Fonctions sur les damiers

3.3.1 Retournement

- Définir `proj_board:pos` → `pos` → `board` → `color` la fonction qui calcule la couleur d'un point du damier.
- Définir `turn_line`, `turn_col:pos` → `board` → `board` qui retournent respectivement une ligne et une colonne du damier.
- Tester ces fonctions sur des exemples.

3.3.2 Déplacement

- Définir une relation `move:board` → `board` → `Prop` telle que :
  `(move b1 b2)` est vrai lorsque `b2` est obtenu en retournant une ligne ou une colonne de `b1`.
  On pourra choisir une définition inductive ou bien utiliser les connecteurs logiques.
- Dans les deux cas on définira :
  `move_line`: `forall (b1:board) (p:pos), (move b1 (turn_line p b1))`
  `move_col`: `forall (b1:board) (p:pos),(move b1 (turn_col p b1)).`

3.3.3 Déplacements multiples

- Définir une relation `moves:board` → `board` → `Prop` telle que :
  `(moves b1 b2)` est vrai lorsque `b2` est obtenu par 0, 1 ou plusieurs déplacements à partir de `b1`.

On prouvera les lemmes suivants :

- `moves_init`: `forall (b1:board), (moves b1 b1)`
- `move_moves`: `forall (b1 b2:board), (move b1 b2)` → `(moves b1 b2)`
- `moves_trans`: `forall (b1 b2 b3:board),`
  `(moves b1 b2)`→`(moves b2 b3)`→`(moves b1 b3)`

3.4 Résultats

Établir les propriétés `(moves start target)` et `~(moves start white_board)`.
Pour établir que `~(moves b1 b2)` on montrera qu'il suffit de trouver un type `M` et une fonction `f:board`→`M` telle que :

- `forall (b1 b2:board), (moves b1 b2)→(f b1)=(f b2)`
- `(f b1)` n'est pas égal à `(f b2)`.

3.5 Décider de la relation `moves`

Le but de cette partie de l'exercice est de prouver la décidabilité de la relation `moves` sur les damiers.

Pour cela on utilise une fonction de normalisation des damiers en retournant les lignes et les colonnes de manière à mettre uniquement des blancs sur la première ligne et colonne.

3.5.1 Normalisation

Définir une fonction `force_white:board → board` qui retourne ligne et colonnes d'un damier pour mettre des blancs sur la première ligne et la première colonne.

3.5.2 Preuve de correction de `force_white`

Le but de cette partie est de montrer la correction de `force_white` à savoir :

`(force_white b1)=(force_white b2) ↔(moves b1 b2)`

Pour cela on pourra suivre les étapes suivantes.

Invariant

On appelle `invariant` une fonction `f` sur les damiers telle que
`(move b1 b2)→(f b1)=(f b2)`

Montrer que si `f` est un invariant et `~(f b1)=(f b2)`
alors `~(moves b1 b2)`.

Involution

On appelle `involution` une fonction telle que `(f (f x))=x`.
On pourra supposer sans démonstration que `turn_line` et `turn_col` sont des involutions.
En déduire que les relations `move` et `moves` sont symétriques.

Résultat

Établir : `(force_white b1)=(force_white b2) ↔(moves b1 b2)`

3.5.3 Décidabilité de `moves`

En utilisant la décidabilité de l'égalité sur les damiers, énoncer puis démontrer la décidabilité de la relation `moves`.

Construire à partir de cette preuve un programme :
`dec` de type `board → board → bool`.

Prouver les résultats suivants :
- si `(dec b1 b2)=true` alors `(moves b1 b2)`
- si `(dec b1 b2)=false` alors `~(moves b1 b2)`

En déduire des preuves "automatiques" pour les résultats établis dans la première partie.

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.

[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.

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

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

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

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