Calcul des Constructions Inductives
DEA Sémantique, Preuves et Programmation
2003-2004

Travaux Pratiques Coq

La section 1 met en œuvre progressivement les principales commandes permettant d'interagir avec l'assistant. Ceux qui ont déjà une pratique de Coq pourront s'en servir comme guide à la nouvelle syntaxe et passer directement à la section 4. La section 2 suggère des exercices pour développer des programmes certifiés dans Coq. La section 3 explique comment compiler ses développements et simplifier l'écriture des termes. La section 4 est un exercice complet permettant la manipulation avancée des définitions inductives.

1 Pratique de Coq

1.1 Démarrer avec Coq

- La commande `coqtop` lance le système Coq.
- Toutes les commandes se terminent par un point.
- La commande pour quitter Coq est `Quit`.
- Le manuel de référence de la version 8.0 de Coq est accessible à l'adresse : http ://coq.inria.fr/doc8/main.html.
- Coq est utilisable graphiquement en utilisant l'interface graphique intégrée CoqIde qui se lance par la commande `coqide`. On peut aussi utiliser le mode Proof General de Emacs (développé à l'université d'Edimbourg, cf http ://proofgeneral.inf.ed.ac.uk) ou le système Pcoq (développé à l'INRIA Sophia-Antipolis, cf http ://www-sop.inria.fr/lemme/pcoq).

1.2 La syntaxe des termes

Nous rappelons dans un tableau les principales constructions syntaxiques de Coq ainsi que leur interprétation.

Sortes	
`Prop`	A de type `Prop` est une formule logique
`Set`	A de type `Set` est un type de données
`Type`	type de `Prop` et de `Set`, des prédicats ...
Produits	
A`->`B	type des fonctions de A dans B lorsque A et B de type `Set`
A`->`B	propriété $A \Rightarrow B$ lorsque A et B de type `Prop`
A`->Prop`	type des prédicats unaires, attendant un argument de type A pour former une proposition.
A`->`B`->`C	même chose que A`->(`B`->`C`)`
`forall` $x : A$, B	quantification universelle de B par rapport à la variable x de type A.
`forall` $x : A$, B	$\forall x : A.B$ lorsque B de type `Prop` et A de type `Set` ou `Type`.
`forall` A : `Set`, A`->`A	type de la fonction identité polymorphe
`forall` A : `Prop`, A`->`A	tautologie $\forall A.A \Rightarrow A$
`forall` $x\ y : A$, B	équivalent à `forall` $x : A$, `forall` $y : A$, B
`forall` $(x : A)(y : A')$, B	équivalent à `forall` $x : A$, `forall` $y : A'$, B
A`->`B	`forall` $x : A$, B lorsque x est une variable sans occurrence libre dans B.
Abstractions	
`fun` $x : A$ `=>` t	abstraction du terme t par rapport à la variable x de type A
`fun` $x\ y : A$ `=>` B	`fun` $x : A$ `=>` `fun` $y : A$ `=>` B
`fun` $(x : A)(y : A')$ `=>` B	`fun` $x : A$ `=>` `fun` $y : A'$ `=>` B

Applications	
$f\ a$	application d'une fonction f à un argument a
$f\ a_1\ a_2 \ldots a_n$	$\ldots((f\ a_1)\ a_2)\ldots a_n$

1.3 L'environnement Coq

Quelques commandes interactives de base sont utiles avant de se lancer dans des preuves qui utilisent les bibliothèques Coq.

CoqIde Dans CoqIde on ne tape dans la fenêtre d'édition que des déclarations ou tactiques de Coq. Les commandes de recherche ou typage sont accessibles à partir du menu **Queries** de coqide ou de la **Query Window** du menu **Window**.
Les commandes `Reset` de suppression de déclarations d'environnement doivent impérativement s'effectuer à l'aide des flèches de navigation.

Recherche dans l'environnement	
`Inspect` *num*	type des *num* dernières déclarations de l'environnement.
`SearchAbout` *name*	imprime les déclarations *id : type* telles que *name* apparait dans *type*
`Search` *name*	imprime les déclarations d'objets *id : type* telles que *name* est la constante de tête de *type* ie *type* est de la forme `forall` $(x_1 : A_1)\ \ldots\ (x_n : A_n)$, `name` $t_1 \ldots t_p$.
`Print` *name*	imprime le corps de la constante *name* avec son type.
Vérification de type	
`Check` *term*	vérifie que *term* est typable et imprime son type. Si *term* est un identificateur, imprime le type de cette constante.
Chargement de modules	
`Require Import` *name*	vérifie si le module *name* est déjà présent dans l'environnement ; sinon cherche en suivant le chemin courant un fichier *name.vo* (une version compilée du module). Charge le module et rend son contenu visible.
`Require Import Arith`	charge le module `Arith` qui regroupe les modules `Le, Lt, Gt, Plus, Mult, Between`.
`Print Modules`	imprime les modules déjà chargés et le chemin de recherche
`Print LoadPath`	imprime le chemin de recherche, par défaut tous les sous-répertoires de la librairie de Coq.
Recherche dans l'environnement	
`Reset` *name*	remet le système dans l'état où il était avant la déclaration de *name*.
`Reset Initial`	remet le système dans l'état initial (lancement de la commande `coqtop`).

Exemples

```
Coq < Definition x := 0.

Coq < Inductive y : Set := C : y.

Coq < Inspect 5.
x : nat
Inductive y : Set :=  C : y
y_rect : forall P : y -> Type, P C -> forall y : y, P y
y_ind : forall P : y -> Prop, P C -> forall y : y, P y
y_rec : forall P : y -> Set, P C -> forall y : y, P y

Coq < Search not.
sym_not_eq : forall (A : Type) (x y : A), x <> y -> y <> x
sym_not_equal : forall (A : Type) (x y : A), x <> y -> y <> x
not_eq_S : forall n m : nat, n <> m -> S n <> S m
```

```
O_S : forall n : nat, 0 <> S n
n_Sn : forall n : nat, n <> S n

Coq < Check (fun x :nat => x + 0).
fun x : nat => x + 0
     : nat -> nat

Coq < Require Import Arith.
```

Exercice 1.3.1

`Search/SearchAbout` *Chercher tous les lemmes connus sur la conjonction (*`and`*), la disjonction (*`or`*), l'égalité (*`eq`*), l'ordre sur les entiers (*`le`*).*

`Print` *Chercher la définition du type* `nat`*, des ordres* `le`*,* `lt` *et les preuves des lemmes* `eq_S` *et* `f_equal`*. Cherchez les définitions des opérations* `plus` *et* `mult`*. Remarquons que celles-ci sont notées de manière infixe avec les symboles* `+` *et* `*`*.*

`Check` *Que prouve le lemme* `nat_case` *?*
Les termes suivants sont-ils bien formés : `(plus O (S O))`, `(plus true false)` *?*

`Print Modules` *Afficher le chemin de recherche des modules.*

`Require Import` *Charger l'un des modules arithmétiques et rechercher les théorèmes connus à propos par exemple de la relation* `le`*.*

1.4 Définitions dans Coq

1.4.1 Paramètres

On peut introduire les paramètres de la théorie que l'on veut développer. La syntaxe est :

`Variable` *name : type*

où *name* est le nom de l'hypothèse ou de la variable à introduire et *type* est son type.
On peut utiliser les mots-clés `Hypothesis`, `Axiom` et `Parameter` comme synonyme de `Variable`.

Plusieurs variables avec le même type peuvent être introduites en utilisant les mots-clés `Variables` ou `Hypotheses` et pour *name* une liste de noms séparés par des espaces.

Par exemple, pour introduire un type pour les éléments d'une liste et deux variables de ce type on écrit :

```
Coq < Variable A : Set.

Coq < Variables x y : A.
```

1.4.2 Définitions

La manière usuelle d'introduire une définition est :

`Definition` *name* `:=` *term* `:` *type*

L'identificateur *name* est alors une abréviation pour représenter le terme *term*. Le type *type* de la définition est optionnel.

Exemple. La fonction qui élève au carré (les deux définitions ci-dessous sont rigoureusement équivalentes).

```
Coq < Definition carre := fun x :nat => x * x.

Coq < Definition carre_bis (x :nat) : nat := x * x.
```

1.5 Déclarations inductives

1.5.1 Définir des structures de données : `Inductive` *name* `: Set`

Des types de base éventuellement récursifs peuvent être introduits directement en spécifiant un ensemble de constructeurs. À chaque constructeur est associé un type qui décrit son « arité ». Certaines restrictions syntaxiques s'appliquent sur ce type. La syntaxe de cette commande est :

`Inductive` *name* `:Set :=` $\mid c_1 : C_1 \mid \ldots \mid c_n : C_n$

où *name* est le nom du type à définir, c_i sont les noms des constructeurs du type et chaque C_i est le type du constructeur c_i.

Exemple. Le type des entiers naturels est déclaré par la commande suivante :

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

Noter que les noms des constructeurs doivent être des identificateurs en particulier `O` est la lettre majuscule et non pas le chiffre zéro. Toutefois, les entiers de `nat` sont par défaut représentés en `Coq` via la notation décimale usuelle. Ainsi, le chiffre `0` est un autre moyen de désigner la constante de nom `O`. De la même manière, la notation décimale, par exemple, `3`, désigne l'entier `S (S (S O))` de `nat`.

Exercice 1.5.1 *Suivre le même schéma pour définir un type pour la représentation des notions suivantes :*

Entiers relatifs : Les entiers relatifs $\mathbb{Z}$ peuvent être représentés comme une structure libre avec `zero`*, et deux injections* `pos` *et* `neg` *de* `nat` *dans $\mathbb{Z}$. Le terme* `(pos n)` *représente l'entier positif $n+1$ et* `(neg n)` *représente l'entier négatif $-n-1$.*

(Optionnel). Expressions arithmétiques correspondant à la description suivante :

$$\mathtt{expr} ::= 0 \mid 1 \mid \mathtt{expr} + \mathtt{expr} \mid \mathtt{expr} - \mathtt{expr}$$

(Optionnel). Les arbres binaires sur un type `A` *à déclarer.*

$$\mathtt{tree} ::= \mathsf{empty} \mid \mathsf{node}(A, \mathtt{tree}, \mathtt{tree})$$

1.5.2 Définir des relations inductives : `Inductive`

Les définitions inductives introduisent des relations spécifiées par un ensemble de propriétés de clôture qui sont un peu analogue à des clauses Prolog généralisées. À chaque clause est associé un nom, celui-ci sera vu comme un constructeur de la relation dont le type est la formule logique associée à la clause. La syntaxe de cette commande est :

`Inductive` *name* : *arity* `:=` $c_1 : C_1 \mid \ldots \mid c_n : C_n$

avec *name* le nom de la relation à définir, *arity* son type (par exemple `nat->nat->Prop` pour une relation binaire sur les entiers naturels) et comme précédemment c_i et C_i sont respectivement le nom et le type des constructeurs.

Exemple. La définition de la relation d'ordre sur les entiers naturels peut être spécifiée comme la plus petite relation vérifiant :

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

Dans `Coq`, on définit une telle relation de la manière suivante :

```
Coq < Inductive LE : nat -> nat -> Prop :=
Coq < | LE_O : forall n :nat, LE 0 n
Coq < | LE_S : forall n m :nat, LE n m -> LE (S n) (S m).
```

A la suite de cette définition, les identificateurs `LE, LE_O, LE_S` sont connus, chacun ayant pour type le terme correspondant de la déclaration inductive, de plus des opérations d'analyse par cas et de définitions récursives sont possibles avec les objets inductifs qui permettent de capturer la propriété de minimalité de la relation.

En fait la définition de l'ordre dans les bibliothèques de base est un peu différente et correspond à la définition :

```
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).
```

L'indication `(n :nat)` après `le` permet de factoriser la déclaration de la variable n dans toute la définition inductive. En contrepartie, le premier argument de `le` doit toujours être cette variable `n`, en particulier la variable n n'aurait pu être factorisée ainsi dans la définition précédente de `LE` puisque `LE` doit être appliqué dans la seconde clause à `(S n)`.

Exercice 1.5.2

- *Définir inductivement un prédicat `Natural` sur $\mathbb{Z}$ qui caractérise `zero` et les entiers positifs. Ce prédicat est défini par deux clauses :* $(\texttt{Natural zero})$ *et* $\forall n : \texttt{nat}, (\texttt{Natural}\ (\texttt{pos}\ n))$.
- *Définir une relation `Diff` telle que* $(\texttt{Diff}\ n\ m\ z)$ *signifie avec, n et m deux entiers naturels et z un entier relatif que la valeur de $n-m$ est z.*
  On peut définir `Diff` par les clauses :
  $(\texttt{Diff}\ O\ O\ \texttt{zero})$, $\forall n : \texttt{nat}, (\texttt{Diff}\ (S\ n)\ O\ (\texttt{pos}\ n))$, $\forall n : \texttt{nat}, (\texttt{Diff}\ O\ (S\ n)\ (\texttt{neg}\ n))$
  et $\forall n\ m : \texttt{nat}, \forall z : \mathbb{Z}, (\texttt{Diff}\ n\ m\ z) \Rightarrow (\texttt{Diff}\ (S\ n)\ (S\ m)\ z)$.
- *(Optionnel) Définir une relation* SubZ *qui spécifie la différence de deux entiers de $\mathbb{Z}$.*
  On peut définir SubZ *par les clauses :*
  $(\mathsf{SubZ}\ x\ \texttt{zero}\ x)$, $(\mathsf{SubZ}\ \texttt{zero}\ (\texttt{pos}\ n)\ (\texttt{neg}\ n))$, $(\mathsf{SubZ}\ \texttt{zero}\ (\texttt{neg}\ n)\ (\texttt{pos}\ n))$,
  $(\mathsf{SubZ}\ (\texttt{pos}\ n)\ (\texttt{neg}\ m)\ (\texttt{pos}\ (n+m+1)))$, $(\mathsf{SubZ}\ (\texttt{neg}\ n)\ (\texttt{pos}\ m)\ (\texttt{neg}\ (n+m+1)))$,
  $\forall n\ m : \texttt{nat}, (\texttt{Diff}\ n\ m\ z) \Rightarrow (\mathsf{SubZ}\ (\texttt{pos}\ n)\ (\texttt{pos}\ m)\ z)$
  et $\forall n\ m : \texttt{nat}, (\texttt{Diff}\ m\ n\ z) \Rightarrow (\mathsf{SubZ}\ (\texttt{neg}\ n)\ (\texttt{neg}\ m)\ z)$,
- *(Optionnel) Définir une relation* Sem *qui associe à chaque expression sa « sémantique » comme un entier de $\mathbb{Z}$.*

1.6 Définir des fonctions

1.6.1 Un opérateur élémentaire de définition par filtrage `match`

Lorsqu'un terme t appartient à un type inductif, on peut construire un nouvel objet par cas suivant les différents constructeurs qui peuvent apparaître en tête de t lorsque celui-ci est évalué. Ces définitions sont connues dans les langages fonctionnels comme des définitions par filtrage. La syntaxe est

$$\texttt{match}\ \textit{term}\ \texttt{with}\ C_1\ args_1\ \texttt{=>}\ \textit{term}_1\ \ldots\ C_n\ args_n\ \texttt{=>}\ \textit{term}_n\ \texttt{end}$$

Dans cette construction, l'expression *`term`* a pour type une définition inductive qui a n constructeurs C_1, ..., C_n. Le terme *`term`*$_i$ représente le calcul à faire dans le cas où l'évaluation de t produit le constructeur C_i. On peut aussi mentionner le type attendu *`type`* du résultat avec la construction :

$$\texttt{match}\ \textit{term}\ \texttt{return}\ \textit{type}\ \texttt{with}\ C_1\ args_1\ \texttt{=>}\ \textit{term}_1\ \ldots\ C_n\ args_n\ \texttt{=>}\ \textit{term}_n\ \texttt{end}$$

Entiers naturels. Si `n` a pour type `nat`, la fonction qui teste si un nombre est nul peut s'écrire

```
Coq < Check (fun n :nat => match n with
Coq <                      | O => true
Coq <                      | S x => false
Coq <                      end).
```

1.6.2 Définitions par filtrage généralisées

Plus généralement, les motifs peuvent filtrer simultanément plusieurs termes, être imbriqués, et contenir le caractère de soulignement _ qui filtre toutes les expressions. Ils sont examinés de manière séquentielle et doivent couvrir complètement le domaine de l'objet inductif. On pourra écrire par exemple :

```
Coq < Check (fun n m :nat =>
Coq <           match n, m with
Coq <           | O, _ => true
Coq <           | _, S O => false
Coq <           | _, _ => true
Coq <           end).
```

Toutefois, cette généralisation du filtrage n'est pas considérée comme élémentaire et sera *compilée* en motifs primitifs tels qu'indiqués plus haut.

1.6.3 Quelques formes équivalentes

`let` $(x_1,..,x_n)$ `=` t `in` u	`match` t `with` $c\ x_1..x_n$ `=>` u `end` lorsque le type de t est un inductif à un seul constructeur c
`if` t `then` u_1 `else` u_2	`match` t `of` c_1 `=>` u_1 `|` c_2 `=>` u_2 `end` lorsque le type de t est un inductif à deux constructeurs c_1 et c_2.

Exercice 1.6.1
- *Définir la fonction prédécesseur de type* `nat->nat` *ainsi qu'une fonction de type* $\mathbb{Z}$`->bool` *qui teste si son argument est zéro.*
- *Écrire la fonction* `or` *pour le « ou » sur les booléens.*
- *(Optionnel) Écrire l'injection canonique de* `nat` *dans* $\mathbb{Z}$*.*
- *(Optionnel) Écrire une fonction de* $\mathbb{Z}$ *dans* $\mathbb{Z}$ *qui calcul l'opposé d'un entier.*

1.6.4 Tester des égalités

On peut imprimer le résultat de l'évaluation d'un terme à l'aide de la commande `Eval compute in` *`term`*. Par exemple :

```
Coq < Eval compute in (fun x :nat => 0 + x).
     = fun x : nat => x
     : nat -> nat
```

Exercice 1.6.2 *Vérifier les propriétés* `or true b` *s'évalue en* `true`*, et* `or false b` *en* `b`.

1.6.5 Élimination dépendante

`Coq` permet à la fois de construire des programmes et de prouver leur propriétés.

Pour prouver $P\ n$ pour n un entier naturel quelconque on peut distinguer le cas $n = 0$ du cas $n =$ `S` p pour un p quelconque. Il suffit donc de prouver P `O` et `forall` p : `nat`, P (`S` p). Le terme qui représente cette preuve repose également sur une construction par filtrage qui utilise de manière interne l'opérateur `match`.

La facilité de définition par filtrage général `match` peut aussi être utilisée dans ce cas, cependant il est très souvent difficile de déduire de manière automatique le prédicat P sur lequel se fait l'analyse, aussi celui-ci doit-il être indiqué explicitement par l'utilisateur en recourant à la construction

`match` *term* `as` x `return` *type*(x) `with` C_1 $args_1$ `=>` *term*$_1$ `...` C_n $args_n$ `=>` *term*$_n$ `end`

. Si *term* est une variable x, il n'est pas nécessaire de mettre l'annotation `as` x.

Exercice 1.6.3 *(Difficile) Soit* `A :Set` *et* `P :A->Set` *définir la somme disjointe de type* `Set` *avec un constructeur :* `intro : forall x :A, P x ->sum`*. Définir ensuite les deux projections* `pi1` *de type* `sum->A` *et la seconde projection* `pi2` *de type* `forall s :sum, P (pi1 s)`*.*
Pour construire `pi2`*, on utilisera la construction* `match` s `return` P `(pi1` s`)` `with ...`*.*

1.6.6 Définitions par point-fixe

Pour coder des fonctions intéressantes sur les structures inductives il est nécessaire d'utiliser une certaine forme de récursion. La récursion générale ne peut être utilisée car elle conduit à des incohérences logiques. La récurrence autorisée est structurelle. Une fonction peut être définie par point fixe si l'un de ses arguments formels x a pour type une définition inductive et dans chaque appel récursif l'argument correspondant peut syntaxiquement être reconnu comme structurellement plus petit que x.

L'idée de base est que x est l'argument principal d'une expression `match` et l'appel récursif peut se faire dans les branches sur une variable d'un motif.

La construction `Fixpoint` La manière de définir une fonction par point fixe est :

`Fixpoint` *name* $(x_1$ `:`*type*$_1)$ `...` $(x_p$ `:`*type*$_p)$ `{struct` x_i`}` `:` *type*$_f$ `:=`*term*

La variable x_i (celle mentionnée après le mot-clé `struct`) est la variable de récursion. Son type *type*$_i$ doit être une instance d'une définition inductive.
Le type de *name* est `forall` $(x_1$ `:`*type*$_1)$`...`$(x_p$ `:`*type*$_p)$`,` *type*$_f$. Les occurrences de *name* dans *term* doivent apparaître appliquées à au moins i arguments et le i-ème doit être reconnu comme plus petit que x. Notons que le mot-clé `struct` peut être omis quand $p = 1$.

Exemples Les deux définitions suivantes de `plus` par récursion sur le premier et le second argument sont correctes.

```
Coq < Fixpoint plus1 (n m :nat) {struct n} : nat :=
Coq <       match n with
Coq <       | O => m
Coq <       | S p => S (plus1 p m)
Coq <       end.

Coq < Fixpoint plus2 (n m :nat) {struct m} : nat :=
Coq <       match m with
Coq <       | O => n
Coq <       | S p => S (plus2 n p)
Coq <       end.
```

Exercice 1.6.4
- *Écrire une fonction de type* `nat` $\rightarrow$ `nat` $\rightarrow \mathbb{Z}$ *qui calcule la différence entre deux entiers naturels, la spécification est :*

$$\mathtt{diff}\ 0\ 0 = \mathtt{zero} \qquad \mathtt{diff}\ 0\ (S\ n) = \mathtt{neg}\ n$$
$$\mathtt{diff}\ (S\ n)\ 0 = \mathtt{pos}\ n \qquad \mathtt{diff}\ (S\ n)\ (S\ m) = \mathtt{diff}\ n\ m$$

- *Utiliser la fonction* `diff` *pour définir la soustraction* `subZ` *de deux entiers relatifs comme une fonction de type* $\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$*.*
- *(Optionnel). Définir une fonction qui associe à chaque expression de type* `expr` *sa « sémantique » comme un élément de* $\mathbb{Z}$*.*

1.6.7 Combinateurs

Lors de l'introduction d'une définition inductive, le système construit à l'aide des primitives `match` et `fix` des principes d'analyse par cas ou récurrence pour le type introduit. Ces principes portent le nom de l'inductif avec pour suffixe `_ind` ou `_rec` ou `_rect` suivant si le principe s'applique à la construction de preuve ou d'objets. Ces principes sont analogues aux combinateurs d'élimination que l'on trouve dans le système T de Gödel ou bien la théorie des types de Martin-Löf.
Des principes supplémentaires peuvent aussi être engendrés à l'aide de la commande `Scheme`.

1.7 Preuves dans Coq

Pour prouver un théorème dans Coq, il faut d'abord énoncer la propriété à prouver. Ceci est fait par la déclaration suivante avec *name* le nom du théorème et *type* le type représentant l'énoncé :

`Lemma` *name : type*

La déclaration `Theorem` est équivalente à `Lemma`.

1.7.1 Raisonnement du premier ordre

La logique de Coq utilise des règles de déduction naturelle pour un calcul des prédicats d'ordre supérieur. À chaque connecteur correspondent des règles d'introduction et d'élimination. Pour la conjonction, les règles sont :

$$(\mathrm{intro})\frac{A \quad B}{A \wedge B} \qquad (\mathrm{elim}_1)\frac{A \wedge B}{A} \qquad (\mathrm{elim}_2)\frac{A \wedge B}{B}$$

Coq utilise une seule règle pour l'élimination équivalente aux deux précédentes.

$$(\mathrm{elim})\frac{A \wedge B \qquad A \Rightarrow B \Rightarrow C}{C}$$

À chaque règle d'inférence de la logique on peut faire correspondre de manière naturelle une tactique : partant d'un but qui est une instance de la conclusion de la règle d'inférence on engendre un sous-but pour chaque prémisse, les conditions de bord devant être par ailleurs vérifiées. Ainsi pour la conjonction, la tactique `split` qui correspond à la règle d'introduction transformera un but conjonctif $A \wedge B$ en deux buts A et B. Pour les tactiques d'élimination, l'information présente dans la conclusion (C) n'est pas suffisante pour instancier les prémisses (il faut connaître en plus B ou A ou même les deux), les tactiques pourront donc prendre en argument des informations supplémentaires. Typiquement les règles d'élimination de base prennent en argument une preuve de la prémisse principale qui est $A \wedge B$ dans le cas de la conjonction.

Dans la table suivante nous donnons les correspondances entre la syntaxe Coq, les connecteurs usuels et les noms des tactiques d'introduction et d'élimination.

Proposition (P)	Syntaxe Coq	Introduction	Élimination (H de type P)
$\perp$	`False`		`elim` H, `contradiction`
$\neg A$	`~A`	`intro`	`apply` H
$A \wedge B$	`A/\B`	`split`	`elim` H
$A \Rightarrow B$	`A->B`	`intro`	`apply` H
$A \vee B$	`A\/B`	`left, right`	`elim` H
$\forall x : A.P$	`forall x :A, P`	`intro`	`apply` H
$\exists x : A.P$	`exists x :A, P`	`exists` *witness*	`elim` H
$x =_A y$	`x=y`	`reflexivity, trivial`	`elim` H, `rewrite` H

Tactique `assumption`. La règle d'axiome est donnée par la tactique `assumption` qui cherche un élément du contexte qui satisfait le but courant.

Tactiques `apply with` et `elim with`. Dans les règles d'élimination, l'argument des tactiques `apply` ou `elim` est un terme qui prouve la prémisse principale.

En général on dispose d'un théorème ou d'une hypothèse T qui ne prouve pas exactement la prémisse principale P mais une généralisation, typiquement `forall` (x1 :$A1$)..(xn :An)`,` P'. Les tactiques tentent de trouver une instanciation adéquate de T qui sera éliminée et engendrera si nécessaire de nouveaux sous-buts à prouver. Si certains arguments ne peuvent être inférés, les tactiques `elim` et `apply` échouent. Dans ce cas les versions dérivées peuvent être utilisées, {`apply | elim`} H `with` $t_1 \dots t_k$.

Quelques commandes utiles Les commandes suivantes permettent de construire une preuve d'un théorème.

- `Proof` *term* : donne explicitement *term* comme un terme représentant la preuve du théorème.
- `Qed` : sauve dans l'environnement une preuve terminée (plus de sous-buts).

Les commandes suivantes s'utilisent dans le cas d'une tentative de preuve erronée. En Coqlde, on utilise à la place les commandes de navigation et d'édition.

- `Undo` : annule la dernière tactique.
- `Restart` : annule toute la preuve.
- `Abort` : annule toute la preuve ainsi que l'énoncé du théorème.

La manière usuelle de combiner les tactiques est `tac1 ; tac2` qui applique `tac1` au but courant puis `tac2` aux sous-buts engendrés.

Exercice 1.7.1 *Prouver les tautologies suivantes :*

$$\begin{array}{r|c} A \wedge (B \vee C) \Rightarrow (A \wedge B) \vee (A \wedge C) & \neg\neg\neg A \Rightarrow \neg A \\ A \vee (\forall x.(P\ x)) \Rightarrow \forall x.(A \vee (P\ x)) & \exists x.\forall y.(Q\ x\ y) \Rightarrow \forall y.\exists x.(Q\ x\ y) \end{array}$$

1.7.2 Raisonnement équationnel

But	tactique	sous-buts	
		Prouver des égalités	
$t = u$	`reflexivity`		t et u s'évaluent en la même expression
$t = u$	`symmetry`	$u = t$	
$t = u$	`transitivity` v	$t = v$, $v = u$	
		Utiliser des égalités pour réécrire	
$P(t)$	`rewrite` H	$P(u)$	H de type $t = u$
$P(t)$	`rewrite <-` H	$P(u)$	H de type $u = t$
$P(t)$	`replace` t `with` u	$P(u)$,u=t	H de type $t = u$
		Convertibilité	
G	`unfold` *name*	G'	remplace dans G la constante *name* par sa définition
G	`simpl`	G'	calcule certaines expressions dans G

Les tactiques `rewrite` ou `replace` remplacent en général toutes les occurrences du terme t à réecrire dans le but. Pour remplacer seulement certaines occurrences du terme t on utilise la tactique `pattern` t `at` *occs*, où *occs* est une liste des occurrences à réécrire avant d'utiliser les tactiques `elim`, `rewrite` ou `replace`. Les tactiques `simpl` ou `unfold` *name* réduisent le but en transformant un terme dans le but en un terme calculatoirement équivalent.

Exercice 1.7.2

- *Prouver la stabilité des constructeurs du type* $\mathbb{Z}$ *pour l'égalité* ($n = m \Rightarrow$ `neg` $n =$ `neg` m).
- *Prouver la direction opposée par exemple* `neg` $n =$ `neg` $m \Rightarrow n = m$.
  (On pourra définir une projection de $\mathbb{Z}$ *dans* `nat`*, ou tester la tactique* `injection`*)*
- *Prouver l'une des égalités définitionnelles de la fonction* `diff`*. Par exemple* `diff` $(S\ n)\ (S\ m) =$ `diff` $n\ m$.

1.7.3 Preuve par récurrence

La tactique `induction`. La tactique à utiliser pour une preuve par récurrence est `induction` *term* avec *term* une expression dont le type est une définition inductive. Cela peut être une récurrence sur un entier ou une liste mais aussi les règles usuelles d'élimination des connecteurs logiques ou bien les principes de minimalité des relations inductives. Plus précisément une élimination correspond à l'application d'un principe engendré au moment de la déclaration de la définition inductive.

La tactique `induction` peut s'utiliser sur des variables ou des hypothèses présentes dans le but. Pour faire référence à une hypothèse non nommée du but, il faut utiliser `induction` *num* où *num* dénote précisément la *num*-ième hypothèse non nommée du but courant.

La tactique `induction` généralise les hypothèses dépendant de l'objet sur lequel l'induction s'applique. Pour éviter cela, on peut soit effacer les hypothèses avec `clear`, soit utiliser les tactiques `elim` (qui attend un terme) et `simple induction` (qui attend une hypothèse quantifiée dans la conclusion) qui sont des variantes plus élémentaires de `induction`.

Récurrence sur les objets. Si *term* a pour type un type inductif alors la récurrence utilisée est la généralisation naturelle de la récurrence sur les entiers naturels. La difficulté principale est d'indiquer au système la propriété à montrer par récurrence.

Par défaut la propriété inférée est l'abstraction du but par rapport à toutes les occurrences de *term*.

Si seulement certaines occurrences doivent être abstraites alors la tactique `pattern` *term* `at` *occs* peut être préalablement utilisée.

Il est parfois nécessaire de généraliser le but avant la récurrence. Ceci peut être fait par la tactique `cut` *type* qui change le but G en *type* $\Rightarrow G$ et engendre le nouveau sous-but *type*. Si la généralisation concerne des hypothèses dans le contexte, on peut utiliser la tactique `generalize`. Si `x` est une variable de type A, la tactique `generalize x` change le but G en le nouveau but `forall x :`A`,` G.

Récurrence sur des preuves. Si *term* appartient à une relation inductive alors la tactique d'élimination correspond à l'utilisation de la minimalité de la relation inductive par rapport aux clauses de définition.

En général la propriété à prouver est $(I\ x_1 \dots x_n) \Rightarrow G$ avec I la relation inductive . Le but G est abstrait par rapport à $x_1 \dots x_n$ pour former la relation utilisée dans la récurrence. Cela fonctionne correctement quand $x_1 \dots x_n$ sont des variables. Dans le cas où $x_1 \dots x_n$ sont des termes structurés, le système peut ne pas trouver d'abstraction correctement typée ou trouver une propriété non prouvable. Dans ce cas la tactique `inversion` *term* au lieu de `elim` *term* combine des tactiques élémentaires de manière à exploiter le maximum d'information disponible dans *term*.

Exercice 1.7.3

- *Prouver les propriétés `Diff` n n `zero` et `Diff` $(S\ n)$ n (`pos` 0).*
- *Prouver que pour tous les entiers n et m si $m \leq n$ alors il existe un entier z tel que `Natural` z et `Diff` n m z.*
- *Prouver que pour tous les entiers naturels n et m la propriété `Diff` n m (`diff` n m) est vérifiée.*

2 Programmer dans Coq

Nous proposons quelques exercices permettant de s'initier à la programmation fonctionnelle certifiée dans Coq.

2.1 Programmation fonctionnelle dans Coq

On se place dans un environnement où un type `A` est spécifié ainsi qu'un ordre `inf` sur `A` de type `A` $\rightarrow$ `A` $\rightarrow$ `bool`.

1. **[Type `list`]** Charger la librairie `List` (`Require Import List`) et chercher à l'aide de la commande `Search list` les fonctions qui définissent des listes. La fonction `length` calcule la longueur d'une liste.

2. **[Fonction `nb_occ`]** Construire à partir de `inf` une fonction qui teste l'égalité de deux objets de type `A`. En déduire une fonction `nb_occ` calculant le nombre d'occurrences d'un élément dans une liste.
3. **[Insertion dans une liste triée]** On suppose que la liste est triée. Construire une fonction d'insertion `insert` d'un élément dans cette liste qui respecte l'ordre.

2.2 Définition de fonctions récursives

1. **[Fonction `split_list`]** Définir une fonction `split_list` qui prend comme argument un objet `a` de type `A` et une liste `l` et qui renvoie la liste des objets plus petits que `a` et celle des objets plus grands que `a`.
   Montrer que les deux listes obtenues ont une longueur plus petite que la liste `(cons a l)`.
2. **[Récurrence bien fondée]** Écrire une fonction qui prend comme argument une liste et renvoie une liste triée. On utilisera un algorithme qui commence par diviser la liste puis trie récursivement les sous-listes et les recompose.
   Pour cela on utilisera l'opérateur `well_founded_induction` et la librairie `Wf_nat` dans laquelle il est prouvé que l'ordre défini par une mesure à valeur dans les entiers est bien fondé.
3. **[Technique : Relations bien fondées]**
   Définir le type des arbres binaires. Écrire la fonction de parcours d'un arbre binaire en largeur. Identifier l'ordre de décroissance associé et justifier la terminaison de cet ordre en utilisant les modules de la bibliothèque `Wellfounded`.

2.3 Spécification et preuve de programmes

1. **[Prédicat `equiv`]** Introduire une spécification `equiv` de type `list→list→Prop` qui est satisfaite pour deux listes lorsque chaque élément apparaît le même nombre de fois dans chaque liste.
2. Prouver que le résultat de l'insertion de `a` dans une liste triée `l` donne une liste équivalente à `(cons a l)`.
3. **[Prédicat `all`]** On se donne une propriété arbitraire `R :A→Prop`. Spécifier pour une liste la propriété `all` qui dit que tous les objets de la liste satisfont `R`.
4. Montrer que si `(all t)` et `(R a)` alors `(all (insert a t))`.
5. **[Élimination sur les booléens]** Le principe de récurrence sur les booléens dit que pour montrer `(P b)` il suffit de montrer `(P true)` et `(P false)`.
   Démontrer un principe plus fort qui dit que pour montrer `(P b)` il suffit de montrer `b=true→(P true)` et `b=false→(P false)`.
6. **[Prédicat `sorted`]** Spécifier la propriété `sorted` pour une liste d'être triée.
   Montrer que le résultat de l'insertion dans une liste triée est encore une liste triée.

2.4 Programmer avec des Preuves

1. Montrer la propriété suivante par récurrence sur `l` :

   ```
   forall (a :A)(l :list), sorted l ->{m :list | equiv m (cons a l) & sorted m}
   ```

2. Est-il possible de faire une récurrence sur l'hypothèse `(sorted l)` ?
3. Montrer un principe de récurrence `sorted_rec` analogue à `sorted_ind` mais portant sur les prédicats `P` de type `list→Set`.
   Quelle est maintenant la réponse à la question 2 ?
4. Donner une spécification pour la fonction `split_list` et faire une preuve correspondante.
5. Montrer le théorème : `forall l :list, {m :list | sorted m & equiv l m}`

3 Utilisation pratique de Coq

3.1 Compiler ses développements

Il est sage de découper un développement en plusieurs fichiers, chacun pouvant être compilé puis recharger de manière très rapide.

Attention! Un fichier ne peut contenir une section qui n'a pas été refermée.

Un outil unix `coq_makefile` permet d'engendre un fichier `Makefile` qui aidera à la compilation et la maintenance des développements. Le plus simple est de créer un fichier `Make` qui contient les informations nécessaires à la compilation du développement comme le nom des fichiers Coq et ML à compiler, le nom des sous-répertoires à traiter, éventuellement les chemins à utiliser pour accéder aux bibliothèques qui ne sont pas installées en standart. L'utilisation la plus courante est :

```
unix% coq_makefile -f Make -o Makefile
```

La première fois que `coq_makefile` est appelé il est nécessaire de créer un fichier `.depend` (par exemple en utilisant la commande unix `touch .depend`) utile au calcul des dépendances et qui est automatiquement inclus dans le `Makefile`. Ce fichier est ensuite tenu à jour en utilisant la commande :

```
unix% make depend
```

Cette commande devra être refaite si un changement intervient dans les commandes `Require` ou lors d'ajout de nouveaux fichiers.

Les fichiers `.vo` peuvent ensuite être compilés individuellement (commande `make` *`file`*`.vo`) ou globalement (commande `make`).

3.2 Notations

3.2.1 Constantes à arguments implicites

Certaines informations de typage sont redondantes dans un terme. Par exemple l'opérateur de formation de paires a un type polymorphe :

```
Coq < Check pair.
pair
     : forall A B : Set, A -> B -> A * B
```

Pour former une paire de deux entiers, il faut construire le terme :

```
Coq < Check (pair 0 0).
(0, 0)
     : (nat * nat)%type
```

Un mode spécial permet de calculer pour chaque constante définie un ensemble d'arguments susceptibles d'être synthétisés. Si le type de la constante c est `forall` $(x_1 : \mathit{type}_1) \ldots (x_n : \mathit{type}_n)$, *type* alors l'argument x_i est classé implicite si x_i est une variable libre d'un des types type_j, et ce dans une position qui ne peut être effacée par réduction.

Ces arguments doivent ensuite être omis. Ce fonctionnement est activé par la commande :

```
Coq < Set Implicit Arguments.
```

On peut ensuite définir :

```
Coq < Definition pair3 (A B C :Set) (x :A) (y :B) (z :C) :
Coq <   A * (B * C) := pair x (pair y z).
pair3 is defined
```

Les positions d'arguments synthétisés sont visibles par la commande `Print Implicit` :

```
Coq < Print Implicit pair3.
pair3 : forall A B C : Set, A -> B -> C -> A * (B * C)
Arguments A, B, C are implicit
```

Si la constante est appliquée à un argument alors celui-ci est par défaut le premier argument non synthétisé de la constante. Une syntaxe spéciale `@pair3` permet de désigner la constante sans arguments synthétisés, on peut également fournir explicitement l'argument `t` associé au paramètre implicite `x` en écrivant `(x :=t)`. Les expressions suivantes seront licites :

```
Coq < Check (pair3 0 true 1).
pair3 0 true 1
     : (nat * (bool * nat))%type

Coq < Check (pair3 (A :=nat)).
pair3 (A :=nat)
     : forall B C : Set, nat -> B -> C -> nat * (B * C)

Coq < Check (pair3 (B :=bool) (C :=nat) 0).
pair3 (B :=bool) (C :=nat) 0
     : bool -> nat -> nat * (bool * nat)
```

La génération de constantes à arguments implicites est désactivée par la commande :

```
Coq < Unset Implicit Arguments.
```

Il est également possible de forcer les arguments implicites d'une constante. Par exemple, on peut ne garder que A comme argument implicite à `pair3` avec la commande :

```
Coq < Implicit Arguments pair3 [A].

Coq < Print Implicit pair3.
pair3 : forall A B C : Set, A -> B -> C -> A * (B * C)
Argument A is implicit
```

Depuis la version 8.0 de Coq, les théories de base de Coq sont définies avec le mécanisme d'automatisation des implicites activé.

3.2.2 Termes incomplets

Un sous-terme peut être remplacé par le signe `_` s'il peut être déduit par les conditions de bon typage du terme.

```
Coq < Type refl_equal.
refl_equal
     : forall (A : Type) (x : A), x = x

Coq < Check (refl_equal 0).
refl_equal 0
     : 0 = 0
```

Exercice 3.2.1 *Quel est le type de* `f_equal _`
Comment est interprété et quel est le type de `(f_equal S (refl_equal 0))`

3.2.3 Abstraction non typée

Si le type attendu d'un terme est connu alors on peut omettre le type dans les lieurs.

```
Coq < Definition istrue b := b = true.
istrue is defined

Coq < Definition istrue' := fun b => b = true.
istrue' is defined
```

4 Exercice : Le damier de Cachan

4.1 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.

4.2 Modélisation du damier

4.2.1 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)`

4.2.2 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.`

4.2.3 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`.

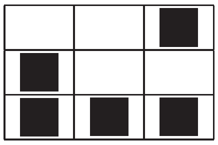
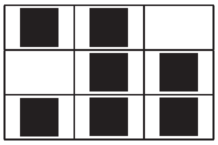
- Définir `triple_map` :`triple` → `triple` qui applique la fonction `f` à chaque composante du triplet.
- Définir `triple_map_select` :`pos` → `triple` → `triple` qui applique la fonction `f` à la composante désignée du triplet.

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

4.2.4 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` = 

4.3 Fonctions sur les damiers

4.3.1 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.

4.3.2 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)`.

4.3.3 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`

4.4 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`.

4.5 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.

4.5.1 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.

4.5.2 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)` $\leftrightarrow$`(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)`$\rightarrow$`(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)` $\leftrightarrow$`(moves b1 b2)`

4.5.3 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` $\rightarrow$ `board` $\rightarrow$ `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.

Table des matières