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 4

Types inductifs

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

4.1 Généralités

4.1.1 Forme générale

Une déclaration inductive dans Coq a la forme suivante :

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

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

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

4.1.2 Forme abstraite

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

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

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

et pour le k-ème constructeur :

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

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

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

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

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

4.2 Les déclarations non-récursives

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

4.2.1 Les déclarations de base

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

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

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

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

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

À partir de ces constructions de base, on peut trouver un équivalent à toute définition inductive non récursive. En reprenant les notations données en 4.1.1 et en introduisant la notation ΣA pour $\Sigma(a_1 : A_1; \ldots; a_n : A_n)$ avec $A \equiv \forall (a_1 : A_1) \ldots (a_n : A_n), s$ l'arité de la définition inductive :

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

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

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

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

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

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

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

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

4.2.2 Règles de formation et d'introduction

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

Règle de formation

En reprenant les notations de 4.1.1, la règle de formation donne le typage de la définition inductive. La condition est que chaque type de constructeur soit bien formé dans un environnement comportant les paramètres :

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

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

Règle d'introduction

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

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

4.2.3 Schémas d'élimination

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

Le schéma d'élimination minimal

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

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

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

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

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

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

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

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

Où Case est un nouveau constructeur de terme.

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

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

On remarque que cette construction est analogue à une déclaration par filtrage, dans lequel on examine dans l'ordre les constructeurs et on n'a que des motifs de la forme $(\mathsf{co}_m\ x^1 \dots x^{n_m})$ avec co_m un constructeur et x^k une variable. On peut utiliser la notation :

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

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

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

Codage imprédicatif Les définitions inductives peuvent être vues comme de nouveaux objets ajoutés à la théorie, ou bien on peut essayer de les coder, c'est-à-dire, étant donné une théorie fixée, et une déclaration inductive de la forme 4.1.1, peut-on trouver des termes $\mathtt{nom}$, $\mathtt{co}_m$ et Case qui ont les types appropriés et satisfont les règles de réduction.

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

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

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

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

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

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

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

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

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

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

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

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

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

Élimination forte

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

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

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

On peut en particulier instancier ce schéma avec $P := [b : \mathtt{bool}]\mathtt{Prop}$, on obtient

$$\frac{f : \mathtt{Prop} \quad g : \mathtt{Prop}}{[b\ : \mathtt{bool}]\mathsf{Case}(P, b.f, g) : \mathtt{bool} \to \mathtt{Prop}}$$

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

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

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

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

Nous verrons dans la partie 4.2.4 que l'élimination forte ne peut être autorisée pour tous les types inductifs sans risque de rendre le système incohérent.

Types dépendants

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

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

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

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

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

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

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

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

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

Égalité

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

L'égalité inductive est définie par

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

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

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

Il vérifie la règle de réduction :

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

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

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

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

On pourrait vouloir montrer que $(n : \texttt{nat})(l : (\texttt{list}\ n))n = 0 \to l = \texttt{nil}$ malheureusement, cette propriété ne peut être énoncée car la propriété $l = \texttt{nil}$ est mal formée puisque $l : (\texttt{list}\ n)$ et $\texttt{nil} : (\texttt{list}\ 0)$ sont de type différents non convertibles. Le fait qu'il existe une preuve e de $n = 0$ dans le contexte permet juste d'appliquer une transformation à l pour en faire un objet de type $(\texttt{list}\ 0)$. Mais l'énoncé devient

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

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

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

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

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

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

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

n'est pas un terme bien typé.

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

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

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

Ce nouvel opérateur satisfait un règle de ι-réduction :

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

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

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

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

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

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

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

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

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

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

4.2.4 Types inductifs et sortes

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

Les types de données prédicatifs

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

Les types de données imprédicatifs

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

$$\mathsf{Inductive}\ \mathtt{prop}\ : \mathtt{Prop} := \mathtt{in} : \mathtt{Prop} \to \mathtt{prop}.$$

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

$$\mathtt{out}(p : \mathtt{prop}) : \mathtt{Prop} := \mathsf{Case}(\lambda x : \mathtt{prop} \Rightarrow \mathtt{Prop}, p, \lambda X : \mathtt{Prop} \Rightarrow X)$$

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

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

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

La distinction entre `Prop` et `Set`

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

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

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

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

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

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

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

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

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

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

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

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

4.3 Les types inductifs récursifs

4.3.1 Exemples

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

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

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

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

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

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

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

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

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

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

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

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

4.3.2 Condition de positivité

Positivité large

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

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

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

$$\mathsf{Definition}\ \mathtt{app}\ (l_1\ l_2:\mathtt{L}):\mathtt{L}:=\mathsf{Case}(\mathtt{L},l1,\lambda f:\mathtt{L}\to\mathtt{L}\Rightarrow(f\ l_2))$$

La réduction suivante est satisfaite :

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

On construit alors :

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

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

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

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

Des exemples de type de constructeur positif pour I sont :

- $A\to I$ avec I qui n'apparaît pas dans A
- $(I\to A)\to B$ avec I qui n'apparaît pas dans A mais peut apparaître positivement dans B

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

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

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

Positivité stricte

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

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

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

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

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

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

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

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

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

4.3.3 Schéma d'élimination récursif primitif

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

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

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

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

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

si $x : C$ on définit :

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

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

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

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

4.3.4 Condition de garde

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

Cependant, Th. Coquand a suggéré une approche alternative, où comme dans les langages de programmation fonctionnelle, les notions primitives sont l'élimination non récursive (qui correspond à l'analyse par cas) et les définitions par point fixe. Évidemment, on ne peut autoriser n'importe quel point fixe sous peine de construire des termes non normalisables et d'aboutir à

une incohérence. Une condition syntaxique de garde permet d'accepter les définitions qui suivent un schéma primitif récursif et de préserver la terminaison.

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

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

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

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

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

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

La règle de réduction de point fixe est :

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

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

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

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

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

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

nécessite l'utilisation de deux points fixe imbriqués :

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

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

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

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

4.3.5 Récurrence structurelle versus récurrence bien fondée

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

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

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

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

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

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

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

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

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

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

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

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

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

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

L'avantage de cette définition est que la propriété `wf` peut être définie dans la sorte `Prop`. Donc le comportement calculatoire de la fonction `quickwf` à l'extraction est le comportement attendu. Par contre, il est compliqué de réduire la fonction `quickwf` dans Coq car cette réduction ne peut avoir lieu que si la preuve h de $(\mathtt{wf}\ \mathtt{list}\ R\ l)$ commence par un constructeur. Pour raisonner sur de telles fonctions, il est préférable d'établir des équations de point fixe. Cependant ces équations

ne sont pas immédiates à établir. Ce problème a été étudié en détail dans la thèse de Antonia Baala. Des théorèmes dans la bibliothèque `Init/Wf` permettent de faciliter les constructions de fonctions par point fixe.

4.3.6 Définitions mutuellement inductives

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

Exemple 3
```
Inductive arbre (A:Set) : Set :=
  | node : A → (foret A) → (arbre A)
with foret (A:Set) : Set :=
  | vide : (foret A)
  | add : (arbre A) → (foret A) → (foret A).
```

est équivalent à

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

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

Elim *term* using *theorem* with *instances*

en instanciant de manière appropriée les propriétés à montrer pour les types auxiliaires.

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

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

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

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

On obtient alors :

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

4.4 Extensions

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

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

4.4.1 Structures infinies

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

4.4.2 Structures quotients

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

4.4.3 Réductions généralisées

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

systèmes de réécriture vérifiant un certain schéma. De manière alternative, J.-P. Jouannaud, A. Rubio et D. Walukiewicz-Chrząszcz proposent un ordre RPO applicable aux termes du Calcul des Constructions qui garantit la préservation par ajout de règles de réécriture.

Chapitre 5

Types coinductifs dans Coq

5.1 Introduction

5.1.1 Types concrets

On appellera *type concret* un type I spécifié par la donnée d'un ensemble fini de constructeurs qui sont des constantes dont le type a pour conclusion une instance de I.

On attend d'un type concret la propriété suivante : tout terme clos de ce type se réduit en un terme commençant par un constructeur.

5.1.2 Types récursifs positifs

Lorsque le type I apparait dans un type d'un argument d'un constructeur alors le type est dit *récursif*.

Si de plus les occurrences de I sont positives dans les arguments des constructeurs alors on dira que le type concret est *positif*.

Un modèle de ces types est donné par des arbres dont les nœuds sont indicés par les constructeurs et dont certaines branches peuvent être des fonctions renvoyant des familles d'arbres de même nature ce qui permet de représenter par un terme fini un branchement infini. C'est le cas par exemple des notations ordinales introduites dans le paragraphe 4.3.1.

On peut vouloir se restreindre aux structures dont chaque branche est finie (même si du fait des branchements infinis, la structure elle-même n'est pas finie). On parlera alors d'objet *bien fondé* dans le type positif. Lorsqu'on se restreint à ces objets bien fondés, on peut disposer de principes de récurrence pour le type, soit sous la forme d'une récurrence structurelle généralisée, soit en disant que l'ordre sous-terme est bien fondé.

Mais il peut être intéressant de disposer aussi d'un type qui pourrait contenir des arbres dont les branches sont potentiellement infinies. Il ne s'agit pas alors de se donner un principe de récurrence pour raisonner sur ces objets mais au contraire des opérateurs pour construire de tels objets infinis, en effet l'application des seuls constructeurs ne construira que des objets finis.

5.2 Exemple des listes infinies

L'exemple de base qui est donné pour de tels types infinis est celui des listes [1]:

[1] on pourra trouver la définition de `Streams` dans la bibliothèque `theories/Lists`


```
Coq < Variable A : Set.
Coq < CoInductive ListI : Set := NilI : ListI  | ConsI : A -> ListI -> ListI.
Coq < CoInductive Stream :  Set := Cons : A -> Stream -> Stream.
```

Dans le cas de `ListI` les listes peuvent être finies ou infinies, dans le cas de `Stream`, il n'y a que des listes infinies dans ce type.

Exercice: Montrer que si on définit le type `Stream` de manière inductive et pas coinductive alors ce type est vide, c'est-à-dire que l'on a une preuve de `Stream` $\rightarrow$ `False`.

5.2.1 Principe de destructivité

Il faut garder à l'esprit que l'on cherche à préserver la propriété des types concrets que chaque terme clos se réduira vers un terme qui commence par un constructeur.

Dans le cas des streams, cela revient à dire que tout terme de type `Stream` est égal à (`Cons` a l) pour un certain $a : A$ et l : `Stream` ce qui peut s'écrire:

```
Coq < Lemma Stream_destr : forall P:Stream -> Set,
Coq <       (forall (a:A) (l:Stream), P (Cons a l)) -> forall s:Stream, P s.
```

D'un point de vue plus opérationnel, si toute valeur de type `Stream` peut être réduite vers un terme construit avec `Cons` alors on peut utiliser un opérateur de définition par filtrage:

```
Coq < Variable C : Set.
Coq < Variable f : A -> Stream -> C.
Coq < Check (fun s:Stream => match s with
Coq <                          | Cons a l => f a l
Coq <                          end).
fun s : Stream => match s with
                  | Cons a l => f a l
                  end
     : Stream -> C
```

et on a la réduction suivante:

```
Coq < Eval compute in
Coq <       (fun (b:A) (s:Stream) => match Cons b s with
Coq <                                | Cons a l => f a l
Coq <                                end).
     = fun (b : A) (s : Stream) => f b s
     : A -> Stream -> C
```

Ceci permet de définir les fonctions d'accès au premier élément de la liste ainsi qu'à sa queue:

```
Coq < Definition head (x:Stream) := match x with
Coq <                                 | Cons a _ => a
Coq <                                 end.
Coq < Definition tail (x:Stream) := match x with
```

```
Coq <                                  | Cons _ s => s
Coq <                                  end.
```

En combinant ces deux fonctions et par récurrence structurelle sur n il est possible de définir la fonction d'accès au n-ème élément d'une liste.

```
Coq < Fixpoint nth (n:nat) (s:Stream) {struct n} : A :=
Coq <       match n with
Coq <       | O => head s
Coq <       | S m => nth m (tail s)
Coq <       end.
```

5.2.2 Principe de co-itération

Tout comme les branchements infinis étaient en fait représentés par des fonctions qui sont une notation finie pour représenter une infinité de valeurs, les branches infinies seront représentées intentionnellement par des programmes permettant de les développer aussi loin que nécessaire.

Le principe dit de co-itération a le type suivant:

```
Coq < Definition Stream_coiter : forall X:Set, (X -> A) -> (X -> X) -> X -> Stream.
```

Le principe de base pour représenter un liste infinie par un objet fini est de se donner un type X quelconque, une fonction `out` de type $X \to A$ une fonction `tran` de type $X \to X$ et un objet x de type X. Ces composantes permettent de construire un processus dont l'état est composé d'un registre de type X dont la valeur initiale est x et des deux fonctions `out` et `tran`. On obtient alors une stream (`Stream_coiter out tran` x) que l'on notera également $(X, \texttt{out}, \texttt{tran}, x)$. On peut interpréter cette stream comme un processus : A chaque étape ce processus a une valeur x dans son registre il peut émettre une valeur de type A donnée par (`out` x) et transformer son registre en la valeur (`tran` x). Si on appelle p la fonction qui prend en entrée la valeur x du registre et renvoie la stream $(X, \texttt{out}, \texttt{tran}, x)$ alors on a :

$$(p\ x) = (\texttt{Cons}\ (\texttt{out}\ x)\ (p\ (\texttt{tran}\ x))) \tag{5.1}$$

Tout comme il existe plusieurs algorithmes qui implantent la même fonction, la même liste infinie pourra être engendrée par des processus très différents.

Par exemple on peut engendrer la suite des puissances successives de n : $1\ n\ n^2\ \ldots n^k \ldots$ des deux manières suivantes en prenant dans les deux cas un registre de type entier:

- Pour `tran` la fonction successeur: $(\texttt{tran}\ k) = k + 1$, pour `out` la fonction $(\texttt{out}\ k) = n^k$ et pour valeur initiale 0.
- Pour fonction `tran` la fonction $(\texttt{tran}\ k) = k \times n$ pour `out` la fonction identité: $(\texttt{out}\ k) = k$ et pour valeur initiale 1.

S'il faut calculer un segment de cette suite alors la seconde méthode sera plus efficace.

En général si on se donne une fonction $f : \texttt{nat} \to A$ alors celle-ci peut être implantée par une stream de registre entier initialisée à 0 de fonction `out` égale à f et de fonction `tran` égale à la fonction successeur. Si cette stream permet de calculer toutes les valeurs successives de f, elle ne tient pas compte des calculs précédents pour le faire.

Les streams définies par co-itération peuvent véritablement se voir comme des petits circuits séquentiels.

5.2.3 Principe de co-récursion

Dans le principe de co-itération, le processus fournit des valeurs successives en mettant à jour le registre mais ne modifie jamais sa structure interne (type du registre, ou fonctions de sortie ou de transition).

Ce n'est pas théoriquement un problème. En effet supposons que l'on veuille faire évoluer un processus $(X_1, \texttt{out}_1, \texttt{tran}_1, x_1)$ vers un processus $(X_2, \texttt{out}_2, \texttt{tran}_2, x_2)$ il suffit de le prévoir en avance et d'appliquer le principe de coitération à un registre de type la somme disjointe $X_1 + X_2$ de X_1 et X_2, la fonction de sortie qui utilise $\texttt{out}_1$ lorsque le registre est dans l'état X_1 et $\texttt{out}_2$ lorsque le registre est dans l'état X_2, quant à la fonction de transition elle peut choisir de laisser le processus dans l'état X_1 ou X_2 ou au contraire d'effectuer une transition d'un état X_1 dans un état X_2 ou inversement.

Il y a une manière systématique de capturer cette possibilité de transformation dans l'implantation d'une stream, c'est le principe de co-récursion:

```
Coq < Definition Stream_corec :
Coq <       forall X:Set, (X -> A) -> (X -> X + Stream) -> X -> Stream.
```

La fonction de transition peut soit choisir de renvoyer un processus de même implantation, soit choisir de transformer le processus en un nouvel objet de type `Stream` pouvant avoir une implantation complètement différente.

Exercice: Montrer que la propriété de corécursion `Stream_corec` se déduit de la coitération `Stream_coiter`.

La propriété attendue pour l'opérateur de co-récursion est la suivante :

$$(p\ x) = (\texttt{Cons}\ (\texttt{out}\ x)\ \texttt{Cases}\ (\texttt{tran}\ x)\ \texttt{of}\ (\texttt{inl}\ x') \Rightarrow (p\ x')\ |\ (\texttt{inr}\ s) \Rightarrow s\ \texttt{end}) \tag{5.2}$$

5.2.4 Définitions par points fixes

Dans les langages de programmation fonctionnels qui manipulent des structures infinies (par exemple les langages de la famille ML paresseux tels que Haskell), il n'est pas question d'opérateur de co-itération, on utilise simplement le point fixe général du langage et la paresse de l'évaluation pour construire des structures infinies. En reprenant les équations définissant les propriétés des streams définies par coitération ou corécursion, on voit que ces streams auraient pu être directement définies par les équations récursives 5.1 ou 5.2.

Cependant dans notre approche, nous avons supposé a priori que toute liste était infinie et en particulier possédait un objet de tête. Il ne sera donc pas possible d'accepter n'importe quelle définition de point fixe.

L'exemple classique de liste problématique est la fonction qui filtre une liste infinie en ne gardant que les objets qui vérifient une condition P. Dans un langage de programmation on écrirait comme corps de la définition de (`filtre` P s):

$$\texttt{Cases}\ s\ \texttt{of}\ (\texttt{Cons}\ a\ l) \Rightarrow \texttt{if}\ (P\ a)\ \texttt{then}\ (\texttt{Cons}\ a\ (\texttt{filtre}\ P\ l))\ \texttt{else}\ (\texttt{filtre}\ P\ l)\ \texttt{end}$$

Cette définition n'est pas correcte sans information supplémentaire, en effet si la liste ne possède plus à partir d'un certain rang d'éléments qui vérifie P alors la liste (`filtre` P s) n'est pas infinie. Renvoyer le résultat dans le type des listes finies ou infinies ne solutionne pas le problème puisqu'on ne sais pas décider si la liste s'arrête ou pas en ayant examiné un fragment initial. Le même problème se pose d'ailleurs si les listes infinies sont représentées par des fonctions totales.

Pour contourner cette difficulté on peut adopter plusieurs approches comme par exemple introduire un constructeur silencieux pour les listes résultats qui sera introduit dans le cas où l'élément ne vérifie pas P. On peut aussi ajouter un élément de preuve qui va garantir que P est vrai infiniment souvent dans s.

5.3 Définition des types co-inductifs dans Coq

Nous montrons maintenant les différentes constructions disponibles dans Coq pour manipuler les structures infinies.

5.3.1 Types de données infinis

Ceux-ci se spécifient à l'aide d'une construction analogue à celle des définitions inductives mais utilisant le mot clé `CoInductive`. On a vu la définition du type des listes.

L'opéateur de destruction par filtrage `Cases` traite de la même manière les définitions inductives et coinductives. Il exprime juste qu'une valeur dans le type concret est formé à partir de l'un des constructeurs du type.

5.3.2 Conditions de gardes

Les définitions d'objets infinis se font à l'aide d'un point fixe gardé par des constructeurs, il n'y a pas besoin de spécifier d'argument de décroissance comme dans le cas de définition inductive.

Exemples de streams sur les entiers

```
Coq < CoFixpoint zeros  : Stream nat := Cons 0 zeros.
zeros is corecursively defined

Coq < CoFixpoint from (n:nat) : Stream nat := Cons n (from (S n)).
from is corecursively defined

Coq < Parameter n : nat.
n is assumed

Coq < CoFixpoint puis (nk:nat) : Stream nat := Cons nk (puis (nk * n)).
puis is corecursively defined
```

Définitions des opérateurs de co-itération et co-récursion

```
Coq < Variables A X : Set.
A is assumed
X is assumed

Coq < Variable out : X -> A.
out is assumed

Coq < Variable tran : X -> X.
tran is assumed

Coq < CoFixpoint Stream_coiter (x:X) : Stream A :=
Coq <       Cons (out x) (Stream_coiter (tran x)).
```



```
Stream_coiter is corecursively defined

Coq < Variable tran_rec : X -> X + Stream A.
tran_rec is assumed

Coq < CoFixpoint Stream_corec (x:X) : Stream A :=
Coq <       Cons (out x)
Coq <         match tran_rec x with
Coq <          | inl x' => Stream_corec x'
Coq <          | inr s => s
Coq <         end.
Stream_corec is corecursively defined
```

La condition de garde stipule qu'un appel récursif ne peut avoir lieu que sous au moins un constructeur (et il doit alors être en position d'argument récursif) et sous uniquement des constructeurs, en particulier un appel récursif ne doit pas se trouver en position d'argument d'un symbole de fonction ou dans la partie principale d'un filtrage, par contre il peut être dans la branche d'une définition par cas. Par exemple pour définir la fonction `map` sur les streams.

```
Coq < Variable B : Set.
B is assumed

Coq < Variable f : A -> B.
f is assumed

Coq < CoFixpoint map (s:Stream A) : Stream B :=
Coq <       match s with
Coq <       | Cons a l => Cons (f a) (map l)
Coq <       end.
map is corecursively defined
```

Exemples de définitions mal formées On vérifie que la fonction `filter` n'est pas acceptée.

```
Coq < Parameter P : A -> bool.
P is assumed

Coq < CoFixpoint filtre (s:Stream A) : Stream A :=
Coq <       match s with
Coq <       | Cons a l => if P a then Cons a (filtre l) else filtre l
Coq <       end.
Error:
Recursive definition of filtre is ill-formed.
In environment
A : Set
X : Set
out : X -> A
tran : X -> X
tran_rec : X -> X + Stream A
B : Set
f : A -> B
filtre : Stream A -> Stream A
s : Stream A
a : A
l : Stream A
unguarded recursive call in filtre l
```

Cependant, certaines définitions qui sont sémantiquement productives ne sont pas acceptées par ce critère essentiellement syntaxique :

```
Coq < CoFixpoint bad  : Stream nat := Cons 0 (map S bad).
Error:
Recursive definition of bad is ill-formed.
In environment
bad : Stream nat
unguarded recursive call in
(cofix map (s : Stream nat) : Stream nat :=
   match s with
   | Cons a l => Cons (S a) (map l)
   end)
```

D'autres systèmes [Gim97] permettent une tolérance plus grande vis-à-vis de ces définitions.

5.3.3 Réduction

Comme dans le cas des définitions co-inductives, il est nécessaire de "garder" également la réduction afin de ne pas perdre la normalisation forte.

Un point fixe est une forme normale :

```
Coq <     Eval compute in zeros.
     = (cofix zeros  : Stream nat := Cons 0 zeros)
     : Stream nat

Coq <     Eval compute in (from 0).
     = (cofix from (n : nat) : Stream nat := Cons n (from (S n))) 0
     : Stream nat
```

Si f est définie comme le co-fixpoint de la fonctionnelle F par $f := \lambda x \Rightarrow (F\ f\ x)$ alors la réduction $(f\ x) \longrightarrow_\iota (F\ f\ x)$ ne se produira que si f se trouve en position de tête dans une définition par cas. La ι-réduction a donc pour forme :

$$\mathtt{match}\ (f\ x)\ \mathtt{with} \ldots \mathtt{end} \longrightarrow_\iota \mathtt{match}\ (F\ f\ x)\ \mathtt{with} \ldots \mathtt{end}$$

La preuve de normalisation forte peut se trouver dans [Gim96b].

Cette propriété de réduction est suffisante pour calculer n'importe quelle valeur d'une stream :

```
Coq < Eval compute in (head (from 0)).
     = 0
     : nat

Coq < Eval compute in (tail (from 1)).
     = (cofix from (n : nat) : Stream nat := Cons n (from (S n))) 2
     : Stream nat

Coq < Eval compute in (nth 6 (from 0)).
     = 6
     : nat
```

En pratique, il est possible de faire une preuve de :

$$(f\ x) = (F\ f\ x)$$

dans laquelle l'égalité est celle de Leibniz, alors que $(f\ x)$ et $(F\ f\ x)$ ne sont pas directement convertibles. Pour cela il faut passer par une étape intermédiaire analogue à une η-expansion que nous allons expliquer maintenant :

Soit I un type inductif avec n constructeurs $c_1, \ldots, c_n$. Il est facile de montrer par cas sur x la propriété suivante que nous appelerons η :

$$x = \mathsf{match}\ x\ \mathsf{with}\ (c_1\ \vec{x}) \Rightarrow (c_1\ \vec{x}) \mid \ldots (c_n\ \vec{x}) \Rightarrow (c_n\ \vec{x})\ \mathsf{end}$$

On procède ensuite de la manière suivante :

$$\begin{array}{rcll} (f\ x) & = & \mathsf{match}\ \ (f\ x)\ \ \mathsf{with}\ (c_1\ \vec{x}) \Rightarrow (c_1\ \vec{x}) \mid \ldots (c_n\ \vec{x}) \Rightarrow (c_n\ \vec{x})\ \mathsf{end} & (\eta) \\ & = & \mathsf{match}\ (F\ f\ x)\ \mathsf{with}\ (c_1\ \vec{x}) \Rightarrow (c_1\ \vec{x}) \mid \ldots (c_n\ \vec{x}) \Rightarrow (c_n\ \vec{x})\ \mathsf{end} & (\iota) \\ & = & (F\ f\ x) & (\eta) \end{array}$$

Cas des Streams Dans le cas des streams, le lemme de η-expansion aura la forme suivante :

```
Coq < Lemma Stream_eta : forall (A:Set) (x:Stream A),
Coq <       x = (match x with Cons a l => Cons a l end.)
Coq <  destruct x; trivial.

Coq < Qed.

Coq < Hint Resolve Stream_eta.
```

On montre ensuite les propriétés attendues :

```
Coq < Lemma puis_eq : forall k:nat, puis k = Cons k (puis (k * n)).

Coq < intros; transitivity (Stream_eta (puis k)); simpl in |- *;
Coq < trivial.
```

5.3.4 Familles coinductives

Le schéma de définition de familles inductives suit le schéma de définition des types inductifs. Il en est de même pour les définitions de familles co-inductives.

Une famille définie co-inductivement est une propriété dont les preuves peuvent être des objets infinis.

Supposons que l'on veuille justifier que de streams (potentiellement infinies) sont extensionnellement égales, c'est-à-dire que les valeurs de leurs différents éléments sont les mêmes. On pourrait le faire en utilisant un entier n et l'accès à la n-ème valeur d'une stream. Cependant, si le type co-inductif devient plus compliqué, alors l'accès aux composantes va nécessiter l'introduction de types de données assez complexes. Il est plus naturel d'utiliser une définitions structurelle de l'égalité. Deux objets co-inductifs sont égaux si ils commencent par le même constructeur et que les composantes sont égales. Évidemment, dans le cas d'objets infinis, on ne peut utiliser une définition inductive de l'égalité. Il faut avoir recours à une définition co-inductive pour capturer l'égalité d'objets infinis.

Dans le cas des streams, cela donne :

```
Coq < Parameter A : Set.
<Your Tactic Text here>
Error: Attempt to save an incomplete proof
```

```
Coq < CoInductive EqStream : Stream A -> Stream A -> Prop :=
Coq <   EqStream_intro : forall s1 s2:Stream A, head s1 = head s2 ->
Coq <                    EqStream (tail s1) (tail s2) -> EqStream s1 s2.
A is assumed
```

Une variante équivalente est :

```
Coq < CoInductive EqStream2 : Stream A -> Stream A -> Prop :=
Coq <   EqStream2_intro : forall (a:A) (s1 s2:Stream A),
Coq <                     EqStream2 s1 s2 -> EqStream2 (Cons a s1) (Cons a s2).
EqStream is defined
```

Les preuves de définitions de relations co-inductives suivent les même règles que les types coinductifs. On peut leur appliquer des analyses par cas, des inversions ou bien construire des preuves par co-itération, co-récursion ou plus généralement par point fixe gardé. Cette dernière méthode s'avère la plus souple. Cependant sa manipulation pour la construction de preuves peut être parfois délicate. En effet les preuves sont construites interactivement par des tactiques. Pour les constructions par point fixe, une tactique $\mathbf{t}(Cofix)$ existe. Elle introduit dans le but une hypothèse identique à la propriété à prouver mais qui ne pourra être utilisée que de manière gardée. Vérifier que la condition de garde est toujours satisfaite après chaque application de tactique est trop couteûx, aussi cette vérification n'est faite qu'au moment de la sauvegarde de la preuve finale ou bien à la demande explicite de l'utilisateur.

Exemples de preuves par point fixe Nou pouvons montrer que l'égalite sur les streams est réflexive, symétrique et transitive :

```
Coq < Lemma EqStreamRefl : forall x:Stream A, EqStream x x.

Coq <      cofix; constructor; trivial.
1 subgoal

  ============================
   forall x : Stream A, EqStream x x
```

```
Coq < Lemma EqStreamSym : forall x y:Stream A, EqStream x y -> EqStream y x.

Coq <      cofix; intros.
1 subgoal

  ============================
   forall x y : Stream A, EqStream x y -> EqStream y x

Coq <      inversion_clear H; constructor; auto.
1 subgoal

  EqStreamSym : forall x y : Stream A, EqStream x y -> EqStream y x
  x : Stream A
  y : Stream A
  H : EqStream x y
  ============================
   EqStream y x
```



```
Coq < Lemma EqStreamTrans :
Coq <      forall x y z:Stream A, EqStream x y -> EqStream y z -> EqStream x z.

Coq <      cofix; intros.
puis_eq < 1 subgoal

  ============================
   forall x y z : Stream A,
   EqStream x y -> EqStream y z -> EqStream x z

Coq <      inversion_clear H; inversion_clear H0; constructor.
1 subgoal

  EqStreamTrans : forall x y z : Stream A,
                  EqStream x y -> EqStream y z -> EqStream x z
  x : Stream A
  y : Stream A
  z : Stream A
  H : EqStream x y
  H0 : EqStream y z
  ============================
   EqStream x z

Coq <      transitivity (head y); auto.
2 subgoals

  EqStreamTrans : forall x y z : Stream A,
                  EqStream x y -> EqStream y z -> EqStream x z
  x : Stream A
  y : Stream A
  z : Stream A
  H1 : head x = head y
  H2 : EqStream (tail x) (tail y)
  H : head y = head z
  H3 : EqStream (tail y) (tail z)
  ============================
   head x = head z
subgoal 2 is:
 EqStream (tail x) (tail z)

Coq <      apply EqStreamTrans with (tail y); trivial.
1 subgoal

  EqStreamTrans : forall x y z : Stream A,
                  EqStream x y -> EqStream y z -> EqStream x z
  x : Stream A
  y : Stream A
  z : Stream A
  H1 : head x = head y
  H2 : EqStream (tail x) (tail y)
  H : head y = head z
  H3 : EqStream (tail y) (tail z)
  ============================
   EqStream (tail x) (tail z)
```

On peut également montrer que l'égalité ainsi définie est bien la plus grande relation R qui vérifie : $(R\ s_1\ s_2) \Rightarrow (\mathtt{head}\ s_1) = (\mathtt{head}\ s_2) \wedge (R\ (\mathtt{tail}\ s_1)\ (\mathtt{tail}\ s_2))$. Il suffit de montrer que si

R est une telle relation alors : (R s_1 s_2) $\Rightarrow$ (`EqStream` s_1 s_2). Cette preuve se fait aisément en utilisant un point fixe est est laissée en exercice.

5.4 Applications

5.4.1 Calcul de processus

Les définitions co-inductives sont utiles à la modélisation de processus parallèles. En effet une définition co-inductive d'une algèbre de processus permet de pouvoir définir des processus récursifs sans avoir explicitement à manipuler un opérateur de récursion. D'autres part les définitions de bisimulation entre processus sont naturellement des définitions co-inductives. Dans les contributions du système `Coq`, on trouvera des modélisations de plusieurs algèbres de processus (CCS, CBS) ainsi que du π-calcul.

5.4.2 Logique temporelle

Les définitions co-inductives permettent également de définir naturellement certains opérateurs de la logique temporelle. Si on suppose donnée un système de transitions R sur des états de type X.

Les formules de la logique temporelle ont pour interprétation in ensemble d'états. On peut définir les propriétés de la logique temporelle, en les représentant directement par leur semantique.

Il sera alors très simple de définir de manière co-inductive les opérateurs `All` et `Ex` sur les formules. La sémantique de `All` (resp. `Ex`) est que soit P une formule et x un état, pour tout chemin (resp. pour un chemin) issu de x, la formule P est vérifiée.

```
Coq < Section Logique_temporelle.

Coq < Variable X : Set.

Coq < Variable R : X -> X -> Prop.

Coq < Variable P : X -> Prop.

Coq < CoInductive All : X -> Prop :=
Coq <   All_intro : forall x:X, P x -> (forall y:X, R x y -> All y) -> All x.

Coq < CoInductive Ex : X -> Prop :=
Coq <   Ex_intro :  forall x:X, P x -> ( exists y : X, R x y /\ Ex y) -> Ex x.

Coq < End Logique_temporelle.
```

5.4.3 Autres applications

Les définitions co-inductives peuvent également être utilisées pour représenter des suites infinies, par exemple des suites de rationnels permettant de représenter des entiers.

Bibliographie

[Coq94] Th. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1994.

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

[Gim95] E. Giménez. Codifying guarded definitions with recursive schemes. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs, TYPES'94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer-Verlag, 1995.

[Gim96a] E. Giménez. An application of co-inductive types in Coq : Verification of the alternating bit protocol. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, 1996.

[Gim96b] E. Giménez. *Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants*. Thèse d'université, Ecole Normale Supérieure de Lyon, December 1996.

[Gim97] E. Giménez. A certification of Petersson's algorithm for managing mutual exclusion. Coq's Contributions, 1997.

[Gim98] E. Giménez. A tutorial on recursive types in coq. Rapport Technique 0221, INRIA, May 1998.