République **A**lgérienne **D**émocratique et **P**opulaire
Ministère de l'**E**nseignement **S**upérieur et de la **R**echerche **S**cientifique

UNIVERSITE **M**OULOUD **M**AMMERI DE **T**IZI-**O**UZOU

FACULTE DE **G**ENIE **E**LECTRIQUE ET D'INFORMATIQUE
DEPARTEMENT D'INFORMATIQUE

Mémoire de Fin d'Etudes de MASTER ACADEMIQUE

Domaine : **Mathématiques et Informatique**
Filière : **Informatique**

Spécialité : **Conduite de projet informatique**

Présenté par
Faiza Aliouat
Saloua Benallel

Thème

Utilisation du système coq pour spécifier un système de transition, modelisant un système physique ainsi que ses propriétés d'accessibilité et de vivacité.

Mémoire soutenu publiquement le 03/07/ 2016 devant le jury composé de :

Président : Mr Chabane Hemdani.
Encadreur : Mr Salem Khemliche.
Examinateur : Mr Mohamed Said Habet.
Examinateur : Mr Youcef Yacine.

République **A**lgérienne **D**émocratique et **P**opulaire
Ministère de l'**E**nseignement **S**upérieur et de la **R**echerche **S**cientifique

UNIVERSITE **M**OULOUD **M**AMMERI DE **T**IZI-**O**UZOU

FACULTE DE **G**ENIE **E**LECTRIQUE ET D'INFORMATIQUE
DEPARTEMENT D'INFORMATIQUE

Mémoire de Fin d'Etudes de MASTER ACADEMIQUE

Domaine : **Mathématiques et Informatique**
Filière : **Informatique**

Spécialité : **Conduite de projet informatique**

Présenté par
Faiza Aliouat
Saloua Benallel

Thème

Utilisation du système coq pour spécifier un système de transition, modelisant un système physique ainsi que ses propriétés d'accessibilité et de vivacité.

Mémoire soutenu publiquement le 03/07/ 2016 devant le jury composé de :

Président : Mr Chabane Hemdani.
Encadreur : Mr Salem Khemliche.
Examinateur : Mr Mohamed Said Habet.
Examinateur : Mr Youcef Yacine.

Remerciements

Louange à notre créateur qui nous incité à acquérir le savoir et nous a donné la volonté et le courage pour y arriver. C'est à lui que nous adressons toute notre gratitude en premier lieu.

Nous tenons à exprimer notre profonde gratitude à notre promoteur, monsieur Khemliche Salem pour nous avoir encadrés durant cette année, Nous tenons, également, à lui exprimer notre profonde reconnaissance pour le temps précieux qu'il nous a consacré ainsi que pour ses conseils judicieux et généreusement prodigués.

Que les membres du jury trouvent ici nos plus vifs remerciements pour avoir accepter d'honorer par leur jugement notre travail.

Un grand merci aussi à toute personne qui de prés ou de loin a contribué à ce que ce modeste travail voit le jour.

Saloua et Faiza.

DEDICACES

Je dédie ce modeste travail :

A Mes chers parents qui m'ont toujours soutenu.
A Mes frères: Nacer, Djamel, Ahmed, Hakim.
A Mes sœurs : Zina, Ouiza, Nassima.

A Ma tante Sadia

A mon binôme Faiza et à toute sa famille.
A toute mes amies sans exception.
A tous mes cousins et cousines.
A toutes les personnes qui pensent à moi de prés ou de loin.

B.Saloua

Je dédie ce modeste travail :

A mes chers parents qui par leurs aides et leurs soutient j'ai pu surmonter tous les obstacles.

A ma sœur la plus cher et unique que j'aime beaucoup.

A mon proche entourage famille et amies.

A tous ceux et celles qui m'ont aidés à réaliser ce travail.

A. Faiza

Résume

Les méthodes formelles ont considérablement évolué, depuis les études de Floyd, Hoare et plusieurs autres pionniers, une tendance de cette évolution fut l'étude de méthodes formelles pour la spécification de systèmes physique de plus en plus complexe .Soit de la vérification de programmes vers la spécification de systèmes.

Dans ce mémoire nous allons utiliser les méthodes formelles pour spécifier un système de transition, modélisant un système physique, ainsi que ses propriétés d'accessibilité et de vivacite, en utilisant l'assistant de preuve Coq.

Pour pouvoir spécifier ce système dans l'assistant de preuve coq, il est impératif d'étudier le fonctionnement de base de ce dernier, dans un premier temps nous avons commencé par étudier la logique du lambda calcul, le calcul des constructions inductives qui est un formalisme intéressant pour servir de base à la programmation fonctionnelle ainsi qu'aux assistants de preuve.

Mots clés

Lambda calcul, Calcul des constructions inductives, Système Coq, Système de transition, Spécification formelle.

Tables des matières

Chapitre 3 : Spécification de système

Chapitre 4 : Etude de cas

Liste des figures

Introduction générale

Aujourd'hui, l'informatique est appliquée à la plupart des activités humaines, y compris ceux dont sa sécurité en dépend, Elle est utilisée dans différents domaines, comme, l'automatisation du transport ferroviaire, les centrales nucléaires, la gestion de paiements électroniques. L'automatisation des tâches touchant aux vies humaines conduisent à des systèmes dits **critiques**, Dans ce genre d'application les enjeux sont énormes, une erreur dans le logiciel peut être très coûteuse, L'absence de validation ou une validation partielle, conduit souvent à des ratages, Parmi ces ratages, nous pouvons citer le Terac 25 et la fusée Ariane V.

- Terac 25 [31] : est un accélérateur de particules utilisé dans le traitement de la radiothérapie. Il est contrôlé par un programme informatique non validé, ce qui a provoqué des pertes en vie humaines.
- Ariane V [30] : est une fusée qui s'est cassée en deux quelques secondes juste après son décollage, pour avoir utilisé pour son contrôle automatique, un fragment de code informatique qui n'est pas validé pour cette fusée.

Pour parer à de tels scénarios catastrophes, il est plus que nécessaire de mettre en oeuvre des méthodes destinées à la spécification et la vérification de logiciel.

Historiquement, la validation se faisait par le biais des **jeux d'essais** ou de la **simulation**. Ces approches **non exhaustives** ne garantissent pas l'absence d'erreur. Il faut donc faire appel à des approches plus rigoureuses qui garantissent de façon absolue l'absence de toute défaillance dans certains logiciels. C'est ce que proposent les méthodes formelles qui se basent sur des théories mathématiques valides.

Il existe essentiellement deux approches dans le domaine de la vérification formelle :

1. La technique de vérification par modèle ;
2. La technique de preuve formelle.

- La vérification par modèle : Elle consiste à :
    1. Modéliser le système par un automate, ou un système de transition, ou une structure de Kripke, ou encore un réseau de pétrie …etc.
    2. Spécifier les propriétés que le système doit vérifier (propriété de robustesse, de vivacité, d'accessibilité, d'équité, de sûreté…etc.) dans une logique temporelle (linéaire ou arborescente).
    3. Vérifier que le modèle vérifie la propriété en utilisant le modèle-Checking, et le raffinement.

L'inconvénient de cette approche est l'explosion combinatoire (le nombre d'exécutions – de cas à vérifier - étant souvent important, qu'on arrive très vite à la saturation de la mémoire).

- La technique de preuve : Elle se base sur la logique formelle, introduite par Hoare cette méthode consiste à spécifier et à démontrer les propriétés du système à l'aide d'outils logiciels appelés assistants de preuves tels que Coq, HOL, Isabelle, LEGO. Toute preuve est effectuée pas à pas par l'utilisateur mais sa validité est garantie par ce système. Cette approche a l'avantage d'être indépendante de la taille de l'espace d'états du système, et peut donc s'appliquer sur des modèles avec un très grand nombre d'états, potentiellement indéterminé ou infini.

Notre objectif est de combiner les deux techniques pour éviter l'explosion combinatoire.
Le but de notre travail est d'utiliser le système Coq pour :

1. Spécifier un système de transition modélisant un système physique réel ;
2. Spécifier les propriétés de vivacité et d'accessibilité ;
3. Montrer que le modèle vérifie ces propriétés.

Pour cela, nous avons structuré notre mémoire comme suit :

Dans le premier chapitre nous allons introduire la théorie du lambda calcul comme étant un modèle formel de calcul, ainsi qu'un aperçu sur le lambda calcul simplement typé et d'ordre supérieur.

Le deuxième chapitre est consacré à l'étude de l'assistant de preuve Coq, en mettant l'accent sur le calcul des constructions inductives qui est à la base du système d'assistance à la preuve Coq.

Le troisième chapitre introduit quelques notions sur les automates et les systèmes de transition, ainsi qu'un développement en Coq d'une spécification du système de transition et des propriétés d'accessibilité et de vivacité de manière générale.

Le quatrième chapitre est consacré à l'étude d'un cas réel, nous avons donc réalisé la spécification d'un système de transition modélisant un système physique qui est le digicode, ainsi que ses propriétés d'accessibilités et de vivacité, et aussi la validation de ces propriétés.

Chapitre 1

La théorie du lambda calcul.

Partie 1

λ –calcul

Introduction

λ -calcul est un calcul défini sur des termes appelés des λ-termes, il a été introduit par Church en 1930. L'idée est de pouvoir exprimer à la fois les données et les fonctions dans un même formalisme. Certes, ce calcul ça provient essentiellement de la simplicité de sa syntaxe et de sa capacité à programmer toute les fonctions calculables, on se basant sur le théorème de la calculabilité.

λ -calcul est donc un formalisme intéressant pour servir de base à la programmation fonctionnelle ainsi qu'aux assistants à la preuve, qui l'utilisent pour représenter les preuves de systèmes de déduction. Il a cependant certaines limitations. En effet, c'est un formalisme qui décrit essentiellement le sujet du calcul. Il donne une description relativement fine d'une procédure de calcul, grâce à laquelle on peut coder beaucoup d'algorithmes. Dans ces langages, un programme est d'abord une fonction. Il est donc très naturellement modélisé comme une expression du λ-calcul (exemple : (λx.a)).

1. La syntaxe du λ –calcul [2] [6]

Soit l'alphabet **A** des λ-termes, qui est composé de :

- Un symbole « λ ».
- Les parenthèses (et).
- Le point «.» et l'espace.
- Un ensemble V (infini dénombrable) de variable : V= {x, y, z, …}.
- Un ensemble Λ de symbole de termes : Λ= {M, N, …}.

On se basant sur l'alphabet **A**, plusieurs règles de construction sont applicables pour construire un λ-terme:

1- Tous les éléments de V sont des λ-termes appelés atomes.

2- Si **e1** et **e2** sont des λ-termes déjà construits alors **e1e2** (la juxtaposition des termes) est un λ-terme, appelé application.

3- Si **x** est une variable et **e** un λ-terme déjà construit, alors **λx.e** est un λ-terme, que nous appellerons abstraction ou fonction, qui à l'argument **x** est associé le terme **e.**

En particulier, une abstraction **λx.e** peut être vue comme l'équivalence en λ-calcul de la forme « **function** x->e » en CAML.

On donne habituellement la syntaxe du λ-calcul en disant que c'est l'ensemble des expressions **Λ** formées de la façon suivante :

$$\Lambda ::= V \mid \Lambda\ \Lambda \mid \lambda\ V \cdot \Lambda$$

On parle de termes plutôt que de λ-termes quand il n'y a pas d'ambiguïté .Les λ-termes sont aussi appelés combinateurs.

- **Convention**

On écrira M≡N pour signifier que M et N sont (syntaxiquement) les mêmes termes ou pour designer des abréviations comme la suppression de certaines parenthèses.

$$(MN) \equiv MN$$
$$((MN)\ P) \equiv MNP$$
$$(\ldots\ ((M_1M_2)\ M_3)\ldots Mn) \equiv M_1M_2M_3\ldots..M_n$$
$$(\lambda x.M\ ) \equiv \lambda x.M$$
$$(\lambda x_1.\ (\lambda x_2.\ (\ldots\ (\lambda xn.M)))) \equiv \lambda x_1\ \ldots\ x_n.M$$

Mais attention, il y'a des contextes dans lesquels les parenthèses sont nécessaires.

$$(M\ (N\ P)) \not\equiv M\ N\ P$$
$$\lambda x.(M\ N) \not\equiv \lambda x.M\ N$$

2. Curryfication

Notons qu'en λ-calcul, les fonctions sont écrites en notation de Curry, ou curryfiées, ce qui signifie qu'une fonction de n ≥ 2 variables est écrite comme une fonction de la première variable renvoyant une fonction des n − 1 restantes, et ce, de manière récursive. On peut définir des λ-termes à partir des constantes, qui jouent le rôle de valeurs et fonctions prédéfinies des langages de programmation.

Exemple 1

On a (x,y)->f(x,y), ça se traduit avec la curryfication en : λxy.fxy

3. Longueur d'un terme [3]

La longueur d'un terme noté Lg(t) est définie par induction comme suit :

- Pour toute variable x, Lg(x)=1
- Lg(M N)= Lg(M) + Lg (N)
- Lg(λx.M)=1+Lg(M)

4. Constantes prédéfinies [2]

On peut ajouter à l'alphabet initial 0, 1, 2, ..., +, ∗, if, etc., pour former des termes comme λx.(+ (∗ 2 x) 1), que l'on pourra écrire encore λx.(2 ∗ x + 1), en notation infixée habituelle avec les priorités d'opérateurs standard.

Cependant, la puissance du λ-calcul n'est pas augmentée par l'introduction de constantes car il permet sous sa forme basique de représenter toutes les structures de données et algorithmes imaginables, ce qui est habituellement appelé la thèse de Church.

5. Variable libres, liées

5.1 Variables libres

Dans un λ-terme, une variable est dite **libre** si elle n'est pas sous la portée d'un **λ**. Plus formellement, l'ensemble FV(M) des variables libres d'un λ-terme M est défini par induction sur la structure de M par les règles :

$$FV(x)=\{x\}$$
$$FV(\lambda x.M)=FV(M)\backslash\{x\}$$
$$FV(M\ N) = FV(M) \cup FV(N)$$

5.2 Variables liées

Une variable est dite **liée** si elle est sous la portée d'un **λ**. L'ensemble BV(M) des variables liées d'un λ-terme M se définit par induction avec les règles :

$$BV(x) = \emptyset$$
$$BV(\lambda x.M) = BV(M) \cup \{x\}$$
$$BV(M\ N) = BV(M) \cup BV(N)$$

Attention : une même variable peut être à la fois libre et liée dans un même λ-terme.

Par exemple,

Avec M= (x (λx.(x x) y)) , on a :

FV(M)={x,y}

BV(M)={x}

La variable x est donc à la fois libre et liée.

6. Terme clos

On dit que le terme M est **clos** ou encore **combinateur**, lorsque FV(M)=∅.

Par exemple,

λx.x

λfgx. (f (g x))

sont des termes clos qui représentent intuitivement l'identité et la composition de fonctions

7. Sous-termes (les occurrences)

Un sous-terme d'un λ-terme est un facteur de ce λ-terme qui est lui-même un λ-terme. L'ensemble ***Sub*(M)** des sous –termes d'un λ-terme M se définit inductivement par les règles :

$$Sub(x) = \{x\}$$

$$Sub(\lambda x.M) = Sub(M) \cup \{\lambda x.M\}$$

$$Sub(M\ N) = Sub(M) \cup Sub(N) \cup \{(M\ N)\}$$

Exemple 2

Avec M = (x (λx.(x x) y))on a :

Sub(M) = {x, y,(x x), λx.(x x),(λx.(x x) y),(x (λx.(x x) y))}.

8. Notion de substitution [3]

L'opération de substitution d'une variable par un terme N dans un terme M consiste à remplacer toutes les occurrences libres de la variable par le terme N. On note M[x :=N]le terme ainsi obtenu.

Cette opération peut être définie précisément inductivement par les règles suivantes :

1. x[x:=P] = P
2. **y**[x:=P] = y si y ≠ x
3. (λx. M) [x:=P] = λx. M
4. (λy. M) [x:=P] = (λy.(M [x:=P])
si y ≠ x, et y ∉FV(P) ou x∉FV(M)
5. (λy. M) [x:=P] = λz.(M [y:=z] [x:=P]) si y ≠ x
et x∈FV(M) et y ∈FV(P) et z est une nouvelle variable.
6. (M1 M2) [x:=P] = M1 [x:=P] M2 [x:=P]

Exemples 3

(λx.y x)[x:=Z] ≡ (λx.y x)
(x (λx.y x))[x:=Z] ≡(Z (λx.y x)) empathie susceptible

La **substitution** est un mécanisme externe au lambda-calcul, on dit aussi qu'il fait partie de la méta-théorie. À noter que certains travaux visent à introduire la substitution comme un mécanisme interne au lambda-calcul, conduisant à ce qu'on appelle les calculs de substitutions explicites.

8.1 α-conversion

La α-conversion identifie λy.v et λz.v[y := z]. Deux lambda-termes qui ne diffèrent que par un renommage (sans capture) de leurs variables liées sont dits **α-convertibles**. La α-conversion est une relation d'équivalence entre lambda-termes.

Exemples 4

- (λx.xy)[y := a] = λx.xa
- (λx.xy)[y := x] = λz.zx (et non λ x.xx, qui est totalement différent)

Remarque

La α-conversion doit être définie avec précaution avant la substitution. Ainsi dans le terme λx.λy.xy, on ne pourra pas renommer brutalement x en y (on obtiendrait λy.λy.yy) par contre on peut renommer x en z.

8.2 α-équivalence [3]

La α-équivalence notée (=α) est une relation binaire définie sur l'ensemble Λ par :

- P =α P pour tout λ-terme P.
- λx.P =α λy.P[x:=y] s i y ∉ FV (P).

9. Réduction [2] [3]

Une manière de voir les termes du lambda-calcul consiste à les concevoir comme des arbres ayant des nœuds binaires (les applications), des nœuds unaires (les λ- abstractions) et des feuilles (les variables). Les réductions ont pour but de modifier les termes, ou les arbres si on les voit ainsi ;

Exemple 5

la réduction de *(λx.xx)(λy.y)* donne *(λy.y)(λy.y)*.

9.1 β-redex (Radical)

Un terme de la forme (λx.M) N est un radical, appelé aussi redex .

9.2 β-contraction

Le terme M[x := N], qui désigne M où le terme N est substitué à chaque occurrence libre de la variable x, est le β-contracté de (λx.M) N .

On note cette β-contraction (λx.M) N $\rightarrow_B$ M[x := N]. Elle se généralise à p variables en (λx_1 . . . xp.M) N1 . . . Np$\rightarrow_B$ (. . . (M[x_1 := N_1]) . . .)[x_p := N_p].
La relation $\rightarrow_{\mathbf{B}}$est appelée contraction.

Exemple 6

(λx.xy) $\rightarrow_B$(xy)[x := a] = ay.

9.3 β-réduction

La β-réduction est la fermeture réflexive transitive de la relation binaire $\rightarrow_B$, notée comme de coutume $\rightarrow^*_{\mathbf{B}}$. Autrement dit, une réduction est une suite finie, éventuellement vide, de contractions.

La relation $\rightarrow^*_{\mathbf{B}}$ est appelée réduction.

9.4 β-conversion

La β-conversion est la fermeture réflexive transitive symétrique de la relation binaire $\rightarrow^*_B$, notée ↔ ou encore $=_B$. Appelé aussi β-équivalence. Peut être vue comme étant l'équivalence entre deux expressions qui semblent différentes, mais qui ont la même signification.

Exemple 7

(λx y. (x y x)) u v w ($\rightarrow_\beta$) (λy. (u y u)) v w ($\rightarrow_\beta$) (u v u) w.

9.5 β-Abstraction

La règle β peut être utilisée à l'envers afin de construire de nouvelles abstractions.

Exemple 8

(+ 4 1) ($_{\beta}\leftarrow$) (λx. + x 1) 4.

9.6 β-expansion

Le passage du terme du milieu au terme de droite est appelé β-expansion, c'est l'inverse d'une β-réduction.

9.7 βη-réduction

La βη-réduction est une relation réflexive, symétrique et transitive, Appelée êta-réduction, obtenu à partir de β-réduction. Elle est définie ainsi : $\mathbf{\lambda x.ux \rightarrow_{\eta} u}$, lorsque x n'apparait pas libre dans u.

Exemple 9

(λx. + 2 x) ($\rightarrow_{\beta\eta}$)(+ 2).

10. Forme normale

Le terme M est en forme normale, s'il ne contient aucun rédex comme sous terme. Ne peut plus être β-réduit, calcul terminé.

Par exemple,

λx.x

λx. (x y z x)

La propriété de Church Rosser nous assure que lorsqu'elle existe, cette forme est unique.

11. Terminaison

La syntaxe du λ-calcul est extraordinairement simple, et pour l'essentiel il y a une seule règle (la β-réduction) pour "exécuter" un programme. Un calcul se termine lorsqu'on obtient un terme irréductible : aucune β-réduction n'est plus possible, par absence de redex, c'est-à-dire de sous-terme de la forme (λx . M) N.

Chaque objets de base (entiers, opérateurs sur les entiers (addition, produit, etc.), les booléens et les opérateurs logiques) sont des termes irréductibles.

12. Terme normalisable

Un terme M est dit normalisable s'il existe une β-réduction $M \rightarrow_B^* N$, où N est normal.

- **terme fortement normalisable**

On dit que le terme M est fortement normalisable si et seulement si toutes les réductions partant de u sont finies.

- **terme faiblement normalisable**

On dit que le terme M est faiblement normalisable si et seulement si au moins une réduction partant de u est finie.

Exemple 10

- (λx.y) z est fortement normalisable en y.
- ((λx.x x) (λx.x x)) n'est pas normalisable.
- les termes normaux sont normalisables.
- (λx.y) est normalisable (en y), mais pas fortement normalisable.

Remarque

Tous les termes fortement normalisable sont faiblement normalisable, mais le contraire n'est pas vrai.

13. Stratégies de réduction

Un λ-terme peut contenir plusieurs rédex. Si l'on souhaite calculer la forme normale d'un terme u (si elle existe), on va pouvoir l'obtenir en choisissant un rédex dans u, en le contractant, et en répétant le processus jusqu'à ce qu'il n'y ait plus. Ce choix du redex à chaque étape de réduction est appelé une **stratégie de réduction**.

- **Stratégie interne**

Ca consiste à réduire d'abord les rédex les plus internes des termes. La stratégie particulière présentée préfère de plus les rédex de gauche à ceux de droite: elle ne réduit les rédex de droite que quand il n'y en a plus à gauche. Il s'agit d'une stratégie interne gauche. Une stratégie interne droite est évidemment envisageable, et se rapprocherait de la sémantique de OCaml.

La stratégie interne permet de ne procéder qu'une seule fois à l'évaluation des arguments des fonctions, puis de les associer déjà évalués aux paramètres formels de la fonction. Mais cette évaluation aura lieu même si les paramètres formels correspondants ne sont pas utilisés.

- **Stratégies externes**

Ca consiste à réduire les rédex les plus externes d'abord. Autrement dit, alors qu'une stratégie interne réduit (λx · u) v en normalisant λx·u vers λx·u', en normalisant v vers v', puis en réduisant u' [x := v'], une stratégie externe commence par contracter (λx · u)v en u[x := v], qu'il réduit à son tour. Elle procède aux appels de fonction sans en évaluer les arguments. Cette stratégie est parfois également qualifiée d'appel par nom.

14. Représentation des données en λ-calcul [4]

Des codages simulent les objets usuels de l'informatique dont les entiers naturels, les fonctions récursives et les machines de Turing. Réciproquement le lambda-calcul peut être simulé par une machine de Turing. Grâce à la thèse de Church on en déduit que le lambda-calcul est un modèle universel de calcul.

- **Booléens**

Pour représenter des programmes en lambda-calcul, il faut disposer d'abord d'une instruction conditionnelle qui est sous forme de (IF p THEN a ELSE b), qui vaut a si le booléen p est vérifié, et b sinon, et qu'on veut encoder.

- Il s'agit de choisir une représentation des booléens en λ-calcul qui rende cette forme naturelle (et qui existe !).

L'idée la plus naturelle est d'utiliser une application de fonctions. Ce qui revient à représenter les deux valeurs booléennes, "vrai" et "faux", par des fonctions à deux arguments.

$$\text{vrai} := \lambda ab.\ a$$
$$\text{faux} := \lambda ab.\ b$$

Cet encodage permet effectivement de représenter les booléens. On peut alors écrire avec les fonctions booléennes usuelles. La forme (si p a b), équivalente à (IF p THEN a ELSE b), est par construction particulièrement simple.

$$\text{si} := \lambda pab.\ p\ a\ b$$

On peut aussi représenter la disjonction, la conjonction et la négation.

$$\text{et} := \lambda p.\ \lambda q.\ p\ q\ \text{faux}.$$
$$\text{ou} := \lambda p.\ \lambda q.\ p\ \text{vrai}\ q.$$
$$\text{non} := \lambda p.\ \lambda a.\ \lambda b.\ p\ b\ a.$$

Il faut noter que cet encodage n'est pas unique : on pourrait par exemple inverser les définitions de vrai et faux, et on pourrait toujours définir ces opérations (il suffirait d'inverser « et » et « ou », ainsi que « si » et « non »).

- **Entiers**

On peut aussi représenter les entiers naturels. L'idée sous-jacente est de représenter l'entier n par l'opération d'itération n fois d'une fonction : $f \rightarrow f^n(x)$.

$$0 := \lambda fx.\ x$$
$$1 := \lambda fx.\ fx$$
$$2 := \lambda fx.\ f(fx)$$

Plus généralement, on peut définir une fonction **succ** qui prend en paramètre le code d'un entier, et renvoie le code de son successeur.

$$succ := \lambda n.\ \lambda fx.\ f(nfx)$$

On peut aussi coder un prédicat **iszero** qui renvoie vrai si l'entier fourni est nul, et faux sinon. Il suffit d'itérer n fois la fonction qui renvoie toujours faux, en lui donnant vrai comme valeur initiale : si n > 0, la fonction sera appliquée et renverra faux, sinon f^n est l'identité et renvoie vrai.

$$iszero := \lambda n.\ n(\lambda x.\ faux)\ vrai$$

Enfin, on peut coder facilement diverses fonctions numériques usuelles.

$$plus := \lambda nm.\ \lambda fx.\ nf(mfx)$$
$$mult := \lambda nm.\ \lambda f.\ n(mf)$$

Pour des raisons de lisibilité, on notera (a+b) et (a×b) au lieu de (plus a b) et (mult a b) respectivement, mais cela reste des termes du λ-calcul pur.

- **Les listes**

On peut remarquer qu'un entier est une liste qui ne contient pas de clé. En rajoutant une information on peut construire les listes d'une manière analogue aux entiers :

$[a_1 ; ... ; a_n] = \lambda gy.\ g\ a_1\ (...\ (g\ a_n\ y)...$

Ainsi :

$[] = \lambda gy.y$

$[a_1] = \lambda gy.g\ a_1\ y$

$[a_1 ; a_2] = \lambda gy.g\ a_1\ (g\ a_2\ y)$

En peu aussi introduire des itérations sur la liste avec : **liste_it := λlxm.l x m.**

- **Paires**

On voudrait pouvoir représenter des couples de λ-termes. On sait déjà représenter des entiers, donc on pourrait imaginer d'implémenter l'usuelle bijection de N^2 vers N, pour représenter un couple d'entier par un simple entier.

Pour généraliser à tout λ-terme, il suffit de choisir un Godel-codage sur les λ-termes (qui sont finis) pour les représenter par des entiers. Cependant, il faudrait alors pouvoir reconstruire le lambda-terme correspondant à son Godel-codage, soit en substance implémenter un méta-évaluateur (de λ-calcul en λ-calcul). Pour commencer, on ne sait même pas coder la fonction pred (qui à 0 associe 0, et à n associe n - 1).

Il existe une solution beaucoup plus simple et naturelle. Il suffit de représenter les couples comme une application partielle attendant les deux membres du couple comme arguments :

$$pair := \lambda xy.\ \lambda f.\ fxy.$$

(pair a b) construira donc un λ-terme qui prend en paramètre une fonction f et l'applique aux deux éléments du couple. En particulier, si l'on veut obtenir l'un des deux éléments, il suffit de lui passer une fonction de sélecteur qui renvoie soit le premier, soit le second de ses arguments, en utilisant ces fonctions, ce sont les booléens.

$$\text{fst} := \lambda p.\ p\ \text{vrai}.$$
$$\text{snd} := \lambda p.\ p\ \text{faux}.$$

Pour des raisons de lisibilité, on notera (a, b) au lieu de (pair a b).

Il devient alors possible de coder plutôt simplement la fonction **pred**. L'idée est d'associer par itérations un couple valant (pred n, n) à l'entier n : pour 0 il suffit de donner (0, 0), et ensuite d'itérer n fois l'opération (a, b) → (b, b + 1).

$$\text{succpair} := \lambda p.((\lambda n.(n,\text{succ } n))(\text{snd } p)).$$
$$\text{pred} := \lambda n.\ \text{fst}\ (n\ \text{succpair}\ (0, 0)).$$

On peut aussi généraliser le codage des couples à des dimensions supérieures. Pour n entier, on peut représenter le constructeur de n-uplet (couple à n membres) et les projections.

$$\text{prod}_n := \lambda f.\ \lambda x_1 \ldots x_n.\ f x_1 \ldots x_n$$
$$\text{proj}^n_i := \lambda p.\ p(\lambda x_1 \ldots x_n.\ x_i)$$

- **Récursions**

➢ **Fonctions récursives primitives**

Grâce au codage de Church des entiers naturels, on peut définir une notion de calculabilité des fonctions de N dans N selon le λ-calcul : une fonction f' : N → N est λ-calculable s'il existe un λ-terme f tel que pour tout entier naturel n' de code n, le résultat de l'évaluation (par β réductions successives) du terme (f n) est le codage de l'entier f' (n').

Les fonctions récursives primitives sont λ-calculables : on sait représenter les fonctions constantes, n-uplets, et projections. Il est clair que l'on peut écrire les projections de fonctions (pour les fonctions d'arité 1, f o g = λ x. f (g x)). Il reste à prouver que l'on peut représenter en λ-calcul la fonction h, définie par récursion primitive à partir de deux fonctions f et g :

$$h(0, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$$
$$h(n + 1, x_1, \ldots, x_n) = g(h(n, x_1, \ldots, x_n), x_1, \ldots, x_n)$$

Nous traiterons le cas des fonctions d'arité 1, la généralisation ne posant pas de difficulté. Si le terme **c** représente une constante entière (ou fonction d'arité 0), et g une fonction N×N → N λ-calculable, on voudrait un λ-terme **h** vérifiant :

$$h(0) = c \text{ et } h(n + 1) = f(h(n), n).$$

Si on avait h (n + 1) = f(h(n)), il s'agirait en fait de l'itérée de la fonction h, que l'on peut représenter facilement en λ -calcul. Il manque cependant le paramètre n supplémentaire. On va utiliser la même méthode que pour pred : On transmet le couple (h(n), n) à la fonction itérée.

$$succf := \lambda\, p.\ p(\lambda\, x\ n.\ (f\ x\ n.\ succ\ n))$$
$$h := \lambda\, n.\ fst\ (n\ (c, 1)\ succf).$$

- **Combinateur de point fixe**

On voudrait maintenant pouvoir définir des fonctions récursives. L'exemple le plus courant est la fonction factorielle :

$$fac =^{?} \lambda\, n.\ (iszero\ n\ 1\ (n \times fac(pred\ n)))).$$

Une telle définition ne serait pas correcte puisqu'elle utilise le terme fac qui n'est pas défini. On peut la modifier légèrement pour remplacer l'appel récursif de fac par l'appel d'une fonction f passée en paramètre.

$$fac' := \lambda\, f\ n.\ (iszero\ n\ 1\ (n \times f(pred\ n))).$$

Cette fonction fac' n'est pas la factorielle : pour obtenir une factorielle, il faudrait l'appeler en lui donnant « elle-même » en argument :

$$fac = fac'(fac) = fac'(fac'(fac)) = \ldots$$

Autrement dit, on cherche le point fixe de l'opération $f \rightarrow fac'\ f$. Plus généralement, on voudrait (afin de pouvoir écrire toute fonction récursive, et pas seulement fac) disposer d'un combinateur de point fixe, c'est-à-dire un λ-terme vérifiant l'équation suivante :

$$Y\ (f) = f\ (Y\ (f)).$$

Il se trouve qu'il est possible de définir ce terme en λ-calcul.

$$auto := \lambda\ x.\ x\ x$$
$$Y := \lambda\ f.\ auto\ (\lambda\ x.\ f(auto\ x))$$

Le terme vérifie bien l'équation demandée :

$$\begin{aligned} Y\ (f) &= (\lambda\ f.\ auto\ (\lambda\ x.\ f(auto\ x)))f \\ &=_\beta auto\ (\lambda\ x.\ f(auto\ x)) \\ &=_\beta f\ (auto\ (\lambda\ x.\ f(auto\ x))) \\ &= f\ (Y\ (f)) \end{aligned}$$

On peut alors définir fact.

$$fac := Y\ (fac')$$

Il est en fait possible de définir fac plus simplement, comme une fonction récursive primitive.

- **Les arbres binaires**

Le principe de construction des entiers, des couples et des listes se généralise aux arbres binaires :

- constructeur de feuille : feuille = λngy.y n
- constructeur de nœud : nœud = λbcgy.g (b g y) (c g y) (avec b et c des arbres binaires)
- itérateur : arbre_it = λaxm.a x m

Un arbre est soit une feuille, soit un nœud. Dans ce modèle, aucune information n'est stockée au niveau des nœuds, les données (ou clés) sont conservées au niveau des feuilles uniquement. On peut alors définir la fonction qui calcule le nombre de feuilles d'un arbre :

nb_feuilles = λa.arbre_it a (λbc.add b c) (λn.1),

Ou plus simplement:

nb_feuilles = λa.aadd (λn.1)

Le λ-calcul possède un pouvoir expressif qui lui permet de représenter tous les objets nécessaire pour un langage de programmation fonctionnel.

15. Les Théorème [2]

- **Théorème1** (théorème de Church-Rosser)

Si le terme M peut être réduit en M_1 d'une part, et en M_2 d'autre part, alors il existe un terme N tel qu'on puisse réduire à la fois M_1 et M_2 en N. Confluence ne signifie en aucun cas terminaison, c'est-à-dire normalisation forte pour tous les termes.

- **Théorème2** (Stratégie de normalisation)

La stratégie de réduction à gauche est normalisant, c'est-à-dire : pour tout M, si M est normalisable, on obtient la forme normale par réduction systématique répétée du radical le plus à gauche.

- **Théorème3** (Standardisation)

Si u est faiblement normalisable, alors la stratégie externe gauche calcule la forme normale de u par une réduction finie.

Autrement dit, la stratégie externe est peut-être lente, mais elle est sûre: elle ne se laisse pas embarquer dans des bouclages inutiles.

- **Théorème4** (point fixe)

Tout λ-terme F à un point fixe x, autrement dit un λ-terme x tel que $x =_\beta F\ x$. En fait, il existe un λ-terme Y sans variable libre tel que pour tout F, Y F soit un point fixe de F. Un tel Y est appelé un combinateur de point fixe.

- **Théorème5** (Les fonctions λ-représentables)

Les fonctions λ-représentables sont exactement les formes récursives. On dit alors que la fonction f de N^n dans N est représentable en λ-calcul par le terme clos F si et seulement si pour tous entiers $k_1, ..., k_n$ si $f(k_1, ..., k_n)$ n'est pas défini alors l'expression $Fk_1, ..., k_n$ n'est pas réductible sinon si $f(k_1, ..., k_n) = k$ alors l'expression $F\ k_1, ..., k_n$ se réduit en k.

On peut en effet construire toutes les fonctions récursives telles que :

$$SUC = \lambda n.\ \lambda f.\ \lambda x.\ ((n\ f)\ (f\ x))$$

En particulier :

$SUC\ n \rightarrow \lambda f.\ \lambda x.\ (((\lambda f.\ \lambda x.\ (f^n\ x))\ f)\ (f\ x)) \rightarrow \lambda f\ .\lambda x.((\lambda x.\ (f^n\ x))\ (f\ x)) \rightarrow \lambda f\ .\lambda x.\ (f^n\ (f\ x)) = \lambda f.\ \lambda x.\ (f^{n+1}\ x) = n+1.$

Et aussi :

$$Somme + = \lambda n.\ \lambda m\ .\lambda f\ .\lambda x.((n\ f)\ ((m\ f)\ x)).$$
$$produit * = \lambda n\ .\lambda m.\ \lambda f.\ \lambda x\ .((n\ (m\ f))\ x).$$

Exemple 11

$$+ 1\ 2 = \lambda n.\ \Lambda m.\ \lambda f\ .\lambda x.((n\ f)\ ((m\ f)\ x))\ (\lambda f.\lambda x.(f\ x))\ (\lambda f\ .\lambda x\ .(f\ (f\ x))) \rightarrow$$
$$\lambda f\ .\lambda x.\ (((\lambda f\ .\lambda x.\ (f\ x))\ f)\ ((\lambda f.\ \lambda x\ .(f\ (f\ x))\ f)\ x)) \rightarrow \lambda f.\ \lambda x.\ ((\lambda x.\ (f\ x))\ (f\ (f\ x)) \rightarrow$$
$$\lambda f\ .\lambda x.(f\ (f\ (fx))) = 3$$

16. Corollaire [2]

- **Corollaire 1** (Unicité)

Si M est normalisable, il existe un terme normal unique, noté M', tel que $M \twoheadrightarrow_B M'$. On dit aussi que M' est la forme canonique de M.

- **Corollaire 2**

Si la réduction à gauche d'un terme est infinie, ce terme est non normalisable.

- **Corollaire 3**

Le problème de normalisation est semi-décidable. Si M est normalisable, la stratégie de réduction à gauche donne la forme normale en un temps fini, sinon on peut toujours attendre.

17. Lambda calcul et les langages de programmation [3]

Par rapport à la vision habituelle des fonctions en mathématiques, le lambda-calcul offre l'avantage de mettre en évidence le « procédé » de calcul. La fonction est donc définie par son algorithme. C'est pour cette raison que ce langage est adapté à la représentation de programmes informatiques. Plusieurs langages informatiques dérivent directement du lambda-calcul : tout d'abord « LISP » et « Scheme » qui sont des langages non typés, mais aussi les langages de la famille « ML » et « Haskell » qui sont des langages typés.

Conclusion

Le Lambda-calcul est une théorie de la fonctionnalité, dit pur ou non typé. Il Consiste à appliqué une fonction à un argument et former une autre fonction par l'abstraction.

Malgré son laxisme au niveau de l'application, le λ-calcul est intéressent de plusieurs point de vue. Tout d'abord, du point de vue de la calculabilité, les termes du λ-calcul représentent exactement toutes les fonctions récursives partielles. Du point de vue de la programmation, le λ-calcul est un outil essentiel puisqu'il s'agit d'un système formel qui décrit la construction et l'évaluation des fonctions. Il apparait ainsi comme le prototype des langages de programmation dits fonctionnels (ou de la partie fonctionnelle des autres langages).

Le λ-calcul peut être vu comme un résultat positif vue qu'il est puissant que les autres modèles de calcul, mais aussi comme un résultat négatif puisqu'il existe des λ -termes non normalisables.

Il est possible d'éviter ce désagrément en enrichissant le langage d'un système de typage : on essaie d'attribuer un type aux λ-termes, de manière à ce que tous les termes typables soient normalisables (mais il existe des termes au typage non valide, entre autres les termes non normalisables).

Après avoir montré le bon fonctionnement de cette approche, nous allons nous intéresser, dans le chapitre suivant, à l'étude de λ-calcul typé : λ-calcul simplement typé et λ-calcul typé d'ordre supérieur.

Partie 2

λ –calcul typé d'ordre supérieur

Introduction

Le lambda-calcul, étant "trop expressif" (on peut appliquer n'importe quelle fonction, voire quelque chose qui n'est pas une fonction, à n'importe quoi), est une sorte de langage machine. Il est utilisable notamment pour modéliser des phénomènes de compilation. Mais il permet de simuler à bas niveau les types avec les opérations usuelles : les booléens, les couples, les unions de types et les entiers naturels.

Pour raisonner à plus haut niveau, on cherche à typer les λ-termes afin de déterminer le type d'un terme de façon à savoir d'une part s'il est correct, d'autre part dans quel contexte l'utiliser. Permet aussi d'éliminer des termes comme Δ dont la sémantique n à rien d'intuitif et de restreindre le langage à une certaine classe de fonctions (par exemple les fonctions totales).

Il existe en fait de nombreux λ -calculs typés, correspondant à de nombreux systèmes de typages différents, dont la complexité augmente avec le nombre de termes typables. On présentera dans ce chapitre le λ-calcul simplement typé, et puis le λ-calcul typé d'ordre supérieur.

1. λ –calcul simplement typé [2] [3] [4] [5]

On enrichit le langage du λ-calcul par une nouvelle classe d'objets, les types. Deux grandes familles de méthodes existent pour typer le lambda-calcul : à la Curry, en cherchant à déterminer le type sans que le terme soit annoté ou à la Church, en contraignant les variables par un type.

On considère un ensemble dénombrable V de types atomiques à partir desquels on va bâtir des types plus complexes.

1.1 Les types à la Curry (Types simples)

L'ensemble des types simples est le plus petit ensemble **T** tel que $V \subset T$ et, si $\alpha, \beta \in T$, alors $\alpha \rightarrow \beta \in T$. On abrège $\alpha \rightarrow (\beta \rightarrow \gamma)$ en $\alpha \rightarrow \beta \rightarrow \gamma$, en prenant comme convention l'associativité à droite pour →.

De manière équivalente, la génération des types simples symbolisés par le non terminal **α**, avec des types atomiques (variables de type) symbolisés par le terminal **v**, obéit à la grammaire : $\alpha ::= v \mid \alpha \rightarrow \alpha$.

Exemple 1

Pour l'arithmétique, on peut avoir $V =_{déf}$ {nat, bool}. On interprète les objets de type nat →nat comme des fonctions entières, et ceux de nat → (nat → bool), ou nat → nat → bool, comme des fonctions de comparaison d'entiers.

- **Règles de typage:**

$$(\text{var}) \quad \frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$(\text{abs}) \quad \frac{\Gamma, x : \sigma \vdash M : \beta}{\Gamma \vdash \lambda x.\, M : A \rightarrow \beta}$$

$$(\text{app}) \quad \frac{\Gamma \vdash M : \sigma \rightarrow \beta \quad \Gamma \vdash N : \sigma}{G \vdash MN : \beta}$$

1.2 Les types à la Church (λ -termes typés)

On modifie aussi légèrement la grammaire des λ-termes pour que le type des variables apparaisse explicitement dans les abstractions (on le note en exposant) :

$$E ::= v \mid E\ E \mid \lambda v^{T}.\ E$$

Pour typer les termes, on introduit pour tout type **σ** un ensemble X^{σ} de variables de type σ notées x^{σ}, y^{σ}, etc. Ainsi, le terme $\boldsymbol{\lambda x^{\sigma}.\ x}$ ($\lambda x :\sigma.\ \ x$**)** est de type $\sigma \rightarrow \sigma$: c'est l'identité sur les λ-termes de type σ.

Si l'on note T[x] le type du λ-terme clos x, on peut définir **T** récursivement :

– T [MN] = B si T [M] = A → B et T [N] = A, MN n'est pas typable sinon
– T [λx^{t} .E] est le type de E, sachant que les occurrences liées de x sont de type t
– T[v] est le type connu de v (puisque le terme initial était clos, l'occurrence v apparaît sous le champ d'une abstraction, et son type est donc connu)

On notera **x : σ** ou bien $\mathbf{x^{\sigma}}$ pour dire : x est de type σ.

Exemple 2

$M = \lambda x^{nat}.(z^{nat\to nat\to bool} x^{nat} y^{nat})$ est de type $nat \to bool$.

1.3 Typage avec contexte

On peut être beaucoup plus souple en ne fixant pas à priori le type des variables. Dans ce cas, le typage de λ-termes peut se définir à l'aide d'un système d'inférence utilisant une notion de contexte. Mais d'abord, rappelons quelques notions se rapportant aux systèmes d'inférence.

- **Système d'inférence**

Un système d'inférence (ou système formel) est un triplet $S =_{déf} (J, R, X)$ où :

(i) J est un langage appelé langage de jugements ;
(ii) R est un ensemble de règles d'inférence, ou de déduction, de la forme :

$$(R) \quad \frac{p_1 \cdots p_n}{q}$$

avec (R) le nom de la règle, $\mathbf{p_1, \ldots, p_n}$ des jugements de **J** appelés prémisses, et **q** un jugement de **J** appelé conclusion ;

(iii) X est un sous-ensemble de jugements de **J** appelés axiomes, chacun d'eux pouvant d'ailleurs être vu comme une règle d'inférence sans prémisse.

- **Contexte (environnement)**

On appelle contexte un ensemble d'association de types à des variables, noté Γ. Traditionnellement, pour simplifier les notations, un contexte Γ est écrit sous la forme d'une liste.

$$\Gamma = [\, x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n \,]$$

Aussi, $\Gamma \cup \{x : \sigma\}$ est-il noté $\Gamma, x : \sigma$, mais on écrit $x : \sigma \in \Gamma$ pour exprimer que $x : \sigma$ est un élément de Γ. De même, si Γ et Δ sont deux contextes, $\Gamma \cup \Delta$ est noté Γ, Δ.

- **Jugement**

Le système d'inférence pour le typage admet des jugements, appelés jugements de typage, de la forme $\Gamma \vdash M : \sigma$, ce qui signifie "dans le contexte Γ, M est du type σ".
Les règles d'inférence, ou de typage, sont une règle de contexte (C), l'introduction de λ notée (I_λ) et l´élimination de λ notée (E_λ) :

$$(C)\ \frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$(I_\lambda)\ \frac{\Gamma, x : \sigma \vdash M : \beta}{\Gamma \vdash \lambda\, x{:}\sigma\, .\, M : \sigma \to \beta}$$

$$(E_\lambda)\ \frac{\Gamma \vdash M : \sigma \to \beta \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \beta}$$

D'où les règles de typage à la Church.

Un système d'inférence sert à exprimer rigoureusement des preuves, dans notre cas pour typer des termes. Soit un ensemble $H \subset J$.

- **Déduction et théorème**

Une déduction (preuve ou démonstration) d'ensemble d'hypothèses H est une suite $j_1, \ldots, j_m$ de jugements tels que, pour tout $k = 1, \ldots, m$, soit j_k est une hypothèse de H,

soit j_k est la conclusion dans l'application d'une règle d'inférence dont les prémisses sont dans $j_1, \ldots, j_{k-1}$, ce qui se note $H \vdash_S j_m$.

Un théorème est le dernier jugement j_m d'une déduction avec un ensemble d'hypothèses vide, ce qui se note $\emptyset \vdash_S j_m$, ou simplement $\vdash_S j_m$.

Une déduction peut être présentée comme un arbre de raisonnement (ou de déduction ou de preuve), où les nœuds sont les jugements et les arcs correspondent aux barres horizontales étiquetées par les noms des règles appliquées. Les feuilles (en haut) correspondent à des axiomes et à des hypothèses, et la racine (en bas) au résultat.

1.4 Les constantes typées

On peut encore étendre l'ensemble des termes par des constantes typées, en théorie inutiles, mais souvent commodes. Il faut leur adjoindre des règles particulières de réduction (ou de calcul ou de réécriture), appelées δ-règles. La relation binaire correspondante est notée $\rightarrow_{\boldsymbol{\delta}}$.

Exemple 3

Pour l'arithmétique, on peut avoir 0^{nat}, 1^{nat},2^{nat}, succ $^{nat\rightarrow nat}$,$+^{nat\rightarrow nat\rightarrow nat}$, etc., avec des règles de calculs comme succ$0 \rightarrow_{\boldsymbol{\delta}} 1$, $1 + 0 \rightarrow_{\boldsymbol{\delta}} 1$, $1 + 1 \rightarrow_{\boldsymbol{\delta}} 2$, etc.

Mais ces règles compliquent la réduction, et la confluence de la β- réduction peut même être détruite. Les principales définitions du λ-calcul pur se transportent au cas typé : substitution, radical, β-réduction... .

1.5 Isomorphisme de Curry-Howard

Le λ-calcul est un formalisme qui permet de décrire des algorithmes. L'isomorphisme de Curry-Howard montre que les démonstrations et les algorithmes sont deux facettes d'un même concept et qu'ils peuvent se représenter de manière uniforme par des λ –termes :

Il met en évidence la similarité entre les preuves et les lambda-termes, notamment il est possible de “calculer” avec des preuves comme avec des lambda-termes. Par exemple une preuve de A -> B est considérée comme une fonction qui à toute preuve de A associe une preuve de B. De plus, on peut noter les preuves de la même façon.

Pour illustrer cela, nous allons prendre trois règles de déduction logique issues du système de déduction naturelle et les trois règles de typage de lambda calcul simplement typé correspondantes. Nous constaterons alors la correspondance existant entre les deux formalismes.

Lambda calcul simplement typé	**Déduction naturelle**
(var) $\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$	(Var) $\frac{}{\sigma}$ Si $\sigma \in \Gamma$
(Abs) $\frac{\Gamma, x : \sigma \vdash M : \beta}{\Gamma \vdash \lambda x{:}\sigma.\ M : \sigma \to \beta}$	(→intro) $\frac{\Gamma, \sigma \vdash \beta}{\Gamma \vdash \sigma \to \beta}$
(APP) $\frac{\Gamma \vdash M : \sigma \to \beta \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \beta}$	(→élim) $\frac{\Gamma \vdash \sigma \to \beta \quad \Gamma \vdash \sigma}{\Gamma \vdash \beta}$

Il est facile de constater la correspondance syntaxique entre les règles de la première et de la deuxième colonne. Les propriétés dans la deuxième colonne correspondent exactement aux types dans la première colonne. Cette correspondance va même plus loin. La β-réduction dans un λ-terme correspond à l'opération d'élimination des coupures dans une déduction.

On obtient le tableau de correspondance complète de Curry-Howard:

Types	propositions
Termes	preuves
Réduction	simplification des preuves

Dans $\Gamma \vdash M : \beta$,

– **M** est une annotation qui est le « le terme de preuve »,
– **β** peut être vu comme un type ou comme une proposition.

1.6 Terminaison des réductions typées

Si on observe la règle qui régit le typage de l'application d'une fonction on s'aperçoit que le type de la fonction est un terme strictement plus grand que le type de l'argument. Pour cette raison, il n'est pas possible de typer une expression de la forme **Δ**. L'expression $\Omega \equiv \lambda\ x.\ x\ x$ n'est pas typable et donc pas l'expression $\Omega \equiv \mathbf{\Delta\Delta}$, ni l'expression $Y = (\lambda\ zx.\ x(z\ z\ x))(\ \lambda\ zx.\ x(z\ z\ x))$.

Les expressions du λ -calcul pur qui présentent des dérivations infinies ne sont pas typables. En fait ceci se généralise en un théorème très important : toutes les expressions typables du λ -calcul simplement typé sont fortement normalisantes. Dit autrement, les exécutions de programmes écrits en λ-calcul simplement typé se terminent.

1.7 Types récursifs et récusions structurelle

Une autre approche permet d'introduire de la récursion sans perdre la propriété que les calculs se terminent toujours. Il s'agit d'introduire des structures de données représentant des arbres et de n'autoriser que les fonctions récursives qui calculent sur ces arbres en n'autorisant les appels récursifs que sur les sous-termes directs de leur argument initial.

On introduit donc simultanément un type de données récursif et une fonction qui décrit la récursion sur ce type. La contrainte que les appels récursifs ne sont autorisés que pour les sous-arbres peut être exprimée à l'aide du typage.

1.8 Types dépendants

La théorie des types ne devient vraiment intéressante qu'avec les types dépendants, qui autorisent à considérer des familles de types indexées par des données.

Avec cette extension du λ -calcul typé, on pourra exprimer que certaines fonctions ne peuvent être appliquée que sur des arguments qui vérifient certaines propriétés. Par exemple, une fonction de recherche dans une liste ne peut être appliquée qu'à une liste et un entier plus petit que la longueur de cette liste.

On note : **∏x: A. f x**

- **Types dépendants et types récursifs**

Il y a bien une interaction entre types dépendants et types récursifs. En particulier, les types dépendants permettent d'étendre la définition de fonctions récursives en maintenant la propriété de terminaison mais en relâchant la contrainte que les appels récursifs ne peuvent avoir lieu que sur des sous-termes directs.

Une bonne façon de les explorer et d'utiliser un système de preuve basé sur la théorie des types, comme le système Coq, qui utilise une théorie des types particulière : le calcul des constructions inductives.

1.9 Le λ ∏-calcul

- **Le type Type**

Les types sont des termes particuliers de type **Type**. Type est un type de base: c'est le type de tous les types.

Dans le jugement G⊢ t : T, t est terme de type T, dans le contexte G.
Cependant, dans G⊢ t : Type, t est un type.

- **Le type Kind**

Puisque (tab n) est un type, il aura le type Type. Il faut alors que tab ait le type nat→Type et n le type nat. Pour que l'on puisse former (nat→Type) : Type, il faut que l'on ait Type : Type, ce qui conduit à un paradoxe. Pour éviter cela, on introduit le type Kind comme type des familles de type.

- **Règle de typage de λ∏-calcul**

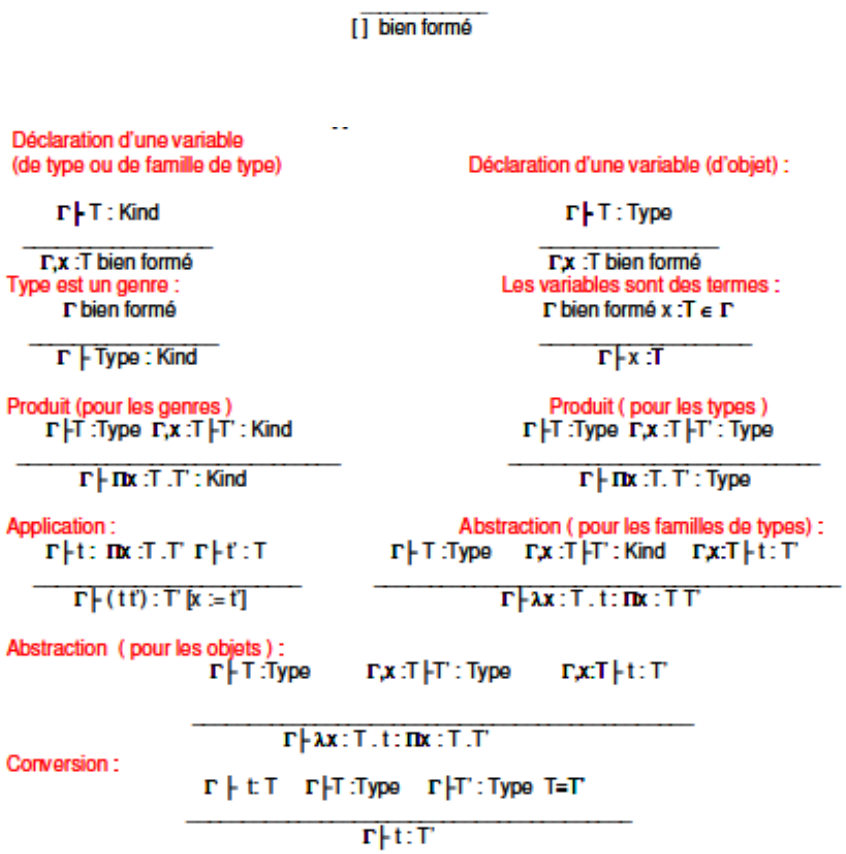


Fig.1. Règles de typage.

Un terme t de type T dans un contexte Γ

- Est un objet si Γ⊢T : Type.
- Est un type si T=Type.
- Est une famille de types si Γ⊢T :Kind.
- Est un genre si T=Kind.

1.10 Extension du λ∏-calcul: Le λ1 –Calcul

Dans le lambda-calcul typé, on peut simuler les types habituels, par exemple bool et nat, pour les booléens et les entiers naturels. Mais la construction réelle de ces types nécessite de passer à l'ordre supérieur. Dans le même ordre d'idée, on peut étendre → par des notations de types secondaires :

- **Produit de types :** on note $\alpha \times \beta$ le type des couples d'éléments de α et β, avec le constructeur de couples noté (,) et les projections fst et snd :

$$(,) : \alpha \to \beta \to \alpha \times \beta$$
$$\text{fst} : \alpha \times \beta \to \alpha \qquad \text{snd} : \alpha \times \beta \to \beta$$

- **Somme de types, ou union disjointe** : on note $\alpha + \beta$ le type des variantes d'éléments de α et β, avec les injections gauche et droite notées inl et inr, et l'alternative case :

$$\text{inl} : \alpha \to \alpha + \beta \qquad \text{inr} : \beta \to \alpha + \beta$$
$$\text{case} : (\alpha \to \gamma) \to (\beta \to \gamma) \to \alpha + \beta \to \gamma$$

Ce sont des types secondaires car ils peuvent être définis à partir de → et des variables de types. Pour typer des termes construits avec ces types secondaires, on peut utiliser les règles suivantes (avec contexte implicite) qui s'ajoutent à celles déjà vues :

$$(I_x)\ \frac{a : \alpha \quad b : \beta}{(a, b) : \alpha \times \beta} \qquad (E_x^1)\ \frac{t : \alpha \times \beta}{(\text{fst } t) : \alpha} \qquad (E_x^2)\ \frac{t : \alpha \times \beta}{(\text{snd } t) : \beta}$$

$$(I_+^1)\ \frac{a : \alpha}{(\text{inl } a) : \alpha + \beta} \qquad (I_+^2)\ \frac{b : \beta}{(\text{inr } b) : \alpha + \beta}$$

$$(E_+)\ \frac{\begin{matrix}[a : \alpha] & [b : \beta] \\ \vdots & \vdots \\ c : \gamma & c' : \gamma \end{matrix} \qquad t : \alpha + \beta}{(\text{case } \lambda a : \alpha.c \quad \lambda b : \beta.c'\ t) : \gamma}$$

En fait, on utilise ici des opérations inl, inr, case, etc., quels que soient les types, ce qui n'est pas correct. Il faudrait les spécialiser en inl $_{\beta}$, inr $_{\sigma}$, case $_{\sigma, \beta, \gamma}$, etc. Enfin, dans cette théorie des types simples, × et + ne sont que des notations. Pour en faire de véritables opérations sur les types, il faut passer à l'ordre supérieur.

Conclusion

Les types simples sont construits comme les types des fonctions, des fonctions de fonctions, des fonctions de fonctions de fonctions vers les fonctions, etc. Quoi qu'il puisse paraître, le pouvoir expressif de ce calcul est très limité (ainsi, l'exponentielle ne peut y être définie, ni même la fonction $n \to 2^n$).

Ainsi, le système de types simples n'est pas assez général. Il ne permet pas d'affecter un type unique à chaque terme. Le type d'un même terme peut dépendre de son contexte.

2. λ –calcul typé d'ordre supérieur [2]

Le lambda-calcul simplement typé est trop restrictif pour exprimer toutes les fonctions calculables dont on a besoin en mathématiques et donc dans un programme informatique. Un moyen de dépasser l'expressivité du lambda-calcul simplement typé consiste à autoriser des variables de type et à quantifier sur elles, comme cela est fait dans le système Fou le calcul des constructions.

Le Système T de Gödel qui fusionne la récursion primitive et le lambda-calcul simplement typé offre aussi, au prix d'un enrichissement, un système plus expressif. Dans ce système, on peut coder, grâce à l'ordre supérieur, de nouveaux algorithmes comme la fonction d'Ackermann qui est non primitive récursive.

2.1 Lambda-calcul du premier ordre : Système T

On va s'intéresser en premier lieu au système T, une extension simple du lambda-calcul classique formalisée par Gödel dans l'espoir de démonter la cohérence de l'arithmétique.

- **Syntaxe du système T :**

Dans le système T, on a deux types de base Int et Bool , à partir duquel on peut comme précédemment construire des types composés. On ajoute par ailleurs trois symboles de constantes O, T et F respectivement de type Int, Bool et Bool et trois opérateurs R, S et D.

S est tel que si t est de type Int, St est de type Int également. Intuitivement, S représente la fonction successeur, et O la constante 0.

R est tel que si u, v et t sont de types U, U->(Int-> U) et Int alors Ruvt est de type Int. (U est un type quelconque). Intuitivement, R est un opérateur de récursion.

D est tel que si u, v et t sont de types U, U, Bool alors Duvt est de type U. Intuitivement, D est un opérateur de branchement, T et F représentant respectivement la valeur Vrai et la valeur Faux.

On ajoute également 4 nouvelles règles de réduction, afin que nos opérateurs aient le sens que l'on veut leur donner.

- **Règles de réduction du système T :**

Ruv0 -> u

Ruv(St) -> v(Ruvt)

Duv T -> u

DuvF -> v

On va enfin coder l'addition et la multiplication qui sont deux opérateurs infixes binaires tels que si **u** et **v** sont de type **Int** alors u(+/*)v est de type Int. Pour ce faire on définit :

$x \underline{+} y = R\ x\ (\lambda n_{Int}.\ \lambda m_{Int}.\ Sn\ )yx \underline{*}\ y = R\ 0\ (\lambda n_{Int}.\ \lambda m_{Int}.(n + x)\ y$

2.2 Lambda-calcul du deuxième ordre : Système F

Ainsi, on passe au deuxième ordre en quantifiant non plus sur des variables d'objets élémentaires, mais sur des variables de propositions, ou de types. Ceci permet d'internaliser certaines définitions d'opérateurs de composition de propositions ou de types. Le système logique correspondant s'appelle système F, dit aussi polymorphisme complet. Il a été proposé à l'origine conjointement par Girard et Reynolds.

2.2.1 Syntaxe de F

- **Syntaxe des types :**

On considère un ensemble dénombrable de variables de type représentées par X. La syntaxe des types T du système F est définie par la grammaire :

$$T ::= X \mid T \rightarrow T \mid \forall X, T$$

Ainsi, l'expression ∀X, T est la quantification par la variable de type X de l'expression de type T. La définition d'un type (ou d'une proposition) par une telle expression est dite imprédicative. Dans la suite, on considère → prioritaire sur ∀X.

- **Syntaxe des termes**

On considère un ensemble dénombrable de variables d'objets élémentaires représentées par x. Le langage des termes t est défini par la grammaire :

$$t ::= x \mid \lambda x : T.t \mid (t\ t) \mid \lambda X.t \mid (t\ T)$$

On voit donc qu'on peut paramétrer les termes (d'objets) par des objets, ce qui était déjà le cas en λ-calcul typé, ou par des types, ce qui est nouveau.

Dans cette notation, λX.t est l'abstraction par la variable de type X du terme t et (t T) est l'application du terme t au type T.

2.3 Système Fω : Ordre supérieur

On passe à l'ordre supérieur à 2 en quantifiant sur des variables de fonctionnelles, c'est-à-dire d'opérateurs sur des propositions ou des types.

L'idée est ici de définir de manière interne les opérateurs, fonctionnelles ou constructeurs, de types, comme + ou × vus précédemment. Pour cela, il faut étendre le système F : en généralisant la sorte type à un système d'arités , construites à partir de type et de →, pour exprimer la fonctionnalité des opérateurs de types en introduisant l'abstraction d'opérateur de type, notée **λX : type →. . . → type**, dans des termes servant à en construire d'autres. On obtient ainsi le système logique appelé **Fω**.

2.3.1 Syntaxe de Fω

Dans cette extension de F qui internalise les définitions d'opérateurs de types, nous examinons successivement la syntaxe des arités, des opérateurs de types et des termes de base.

- **Syntaxe des arités**

Le langage des arités **a** est défini par :

$$a ::= type \mid a \rightarrow type$$

On remarquera que les types vus précédemment en système **F** sont juste ceux d'arité type.

- **Syntaxe des termes-opérateurs de types**

On considère un ensemble dénombrable de variables d'opérateurs de types notées X, Y, etc. D'après la remarque précédente, cet ensemble contient les variables de types, puisqu'elles sont d'arité type. Le langage S des termes-opérateurs de types, encore appelés schémas de types, qui étend le langage de types T du système F, est défini par la grammaire :

$$S ::= X \mid S \rightarrow S \mid \forall X : a, S \mid \lambda X : a.S \mid (S\ S)$$

L'expression **λX : a.S** est l'abstraction du terme-opérateur de type **S** par la variable d'opérateur de type **X** ayant l'arité **a**, et **(S U)** est l'application du terme-opérateur **S** au terme-opérateur **U**.

En réalité, il faut restreindre le langage ci-dessus pour avoir des termes operateurs toujours licites. C'est le rôle des règles de bonne formation qui vont suivre pour les opérateurs de types.

- **Syntaxe des termes**

On reprend celle des termes du système F. Mais, maintenant, on va pouvoir abstraire un terme, non seulement par une variable de type, mais aussi par rapport à une variable d'opérateur de types. On indiquera donc explicitement l'arité des variables d'opérateur de types dans les abstractions. La syntaxe des termes **t** de **Fω** est alors :

$$t ::= x \mid \lambda x : S.t \mid (t\ t) \mid \lambda X : a.t \mid (t\ S)$$

2.4 Calcul des prédicats d'ordre supérieur

On peut ajouter à **F** ou à **Fω** des objets et des quantificateurs du premier ordre, ainsi que des variables propositionnelles d'arité quelconque, pour passer à un calcul des prédicats d'ordre supérieur polymorphe.

Exemple 4

Généralisons un des exemples précédents, en supposant connu l'opérateur de types somme noté **P**, et le prédicat **between**. Un type intervalle d'entiers naturels paramétré par les bornes **a** et **b** est alors défini par :

$$\text{Intab} = \lambda a\text{: nat}.\lambda b\text{: nat}.\sum n \text{:nat, (between a n b)} : \text{nat} \rightarrow \text{nat} \rightarrow \text{type}$$

Ainsi, l'intervalle des entiers de 1 à 10, Int10, est simplement construit par :

$$\text{Int10} =_{\text{def}} (\text{Intab 1 10})$$

En fait, les mécanismes d'abstraction et les syntaxes aux niveaux objet et type se généralisent et s'unifient dans le calcul des constructions qui fait l'objet du chapitre suivant.

Conclusion

Comme langage de programmation, le λ-calcul pur a plusieurs défauts. Le premier défaut est que l'on peut trop facilement faire des erreurs de programmation. Le deuxième défaut est que l'on n'utilise pas intelligemment les structures de données fournies sur les ordinateurs, comme les nombres. Pour cela le λ-calcul a été étendu pour ajouter une notion de typage aux fonctions.

Cette notion de typage permet d'exprimer quelles sont les données attendues par chaque fonction et quelles sont les données retournées par les calculs. Ceci permet de réduire les erreurs que l'on peut faire à la programmation et d'inclure des fonctions qui peuvent prendre en compte les types natifs de l'ordinateur. Le λ-calcul typé permet de garantir que les calculs se terminent toujours.

Chapitre 2

Le CIC et Le Système Coq.

Partie1

Calcul des constructions inductives

Introduction

Le Calcul des Constructions a été présenté en 1985 par Thierry Coquand. Il s'agit d'un lambda-calcul typé qui, à travers l'isomorphisme dit de Curry-Howard, peut-être vu comme un formalisme logique. Ce système qui étend à la fois la logique d'ordre superieur de Church et les systèmes de Martin-Löf est particulièrement expressif du point de vue algorithmique et peut facilement être mis en oeuvre sur ordinateur.

Dans le Calcul des Constructions originel, les types de données (entiers, listes, sommes, etc) sont représentés dans le lambda-calcul à travers un codage imprédicatif. Cette solution est élégante mais conduit à un certain nombre de difficultés pratiques et théoriques. Pour y remédier, Thierry Coquand et Christine Paulin-Mohring ont proposé d'étendre le formalisme par un mécanisme générique de définitions inductives.

L'aspect le plus spectaculaire de l'extension par des types inductifs est la possibilité de définir de nouveaux types et de nouvelles propositions par récurrence structurelle (élimination forte).

En CoC, on exploite rationnellement l'isomorphisme de Curry-Howard [12] : preuve = λ-terme, proposition = type. Il n'y a alors qu'une seule notion pour les propositions et les types, et une seule notion pour les objets concrets et les preuves. Alors, on parle ici uniquement de types (et pas de propositions) et de preuves (et pas d'objets concrets).

1. Syntaxe de CoC [2]

- **Termes** : On a un ensemble dénombrable de variables x, y, etc., utilisées uniformément pour les preuves, les types et les arités. Alors, l'ensemble des termes de preuves, de types et d'arités est d´écrit de façon très générale par :

$$T ::= x \mid type \mid \lambda x : T.T \mid (T\ T) \mid \Pi\ x : T, T$$

- **Contexte** : Un contexte Γ est un ensemble de déclarations de la forme x : A. On peut comprendre une telle écriture de différentes manières :

1) x est une variable de preuve et A est un type ;
2) x est une variable de type, ou d'opérateur (ou fonctionnelle) de types,et A est une arite ;
3) x est une variable d'arité et A est le mot-cle arité.

Exemple 1

Dans le contexte Γ = { n : nat, T : type, op : type → type → type, a : arité}

1- n : nat, déclare la variable n du type nat, qui peut être construit lui-aussi en CoC ;
2- T : type déclare la variable T du type Type, c'est-à-dire un opérateur de types "constante", et op : type → type → type déclare op comme une variable d'opérateur binaire de types ;
3- a : arité déclare la variable a du type arité.

Dans la suite les contextes restent explicites, on appelle type et arité des sortes, et l'on désigne par S l'ensemble composé des mots-clés type et arité, c'est-à-dire :

S =déf {type, arité}.

2. Réduction en CoC

On a la β-réduction usuelle :

$(\lambda x : A.t\ u) \rightarrow_\beta t[x := u]$

D'où, comme d'habitude, $\overset{*}{\rightarrow}_\beta$ et la β-équivalence $=_\beta$.

3. Règles d'inférence ou de typage en CoC

Les jugements s'écrivent $\Gamma \vdash t : A$, où Γ est un contexte, t un terme et A un terme de type ou d'arité, ou encore le mot-clé arité. Avec S =déf {type, arité}, les règles, souvent avec pré condition (ou garde), sont alors très compactes (on y réintègre les contextes) :

$$(Axiome)\ \frac{}{type : arité}$$

$$(Hypothèse)\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \text{si } s \in S \text{ et } \neg(x \in \Gamma)$$

$$(Affaiblissement)\ \frac{\Gamma \vdash A : s \quad \Gamma \vdash t : B}{\Gamma, x : A \vdash t : B}\ \text{si } s \in S$$

$$(Produit)\ \frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : s'}{\Gamma \vdash \prod x : A, B : s'}\ \text{si } s, s' \in S$$

$$(Abstraction)\ \frac{\Gamma \vdash \prod x : A, B : s \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : \prod x : A, B}\ \text{si } s \in S$$

$$(Application)\ \frac{\Gamma \vdash t : \prod x : A, B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B[x := u]}$$

$$(Conversion)\ \frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s \quad A =_\beta B}{\Gamma \vdash t : B}\ \text{si } s \in S$$

Fig.2. Règles d'inférence en CoC.

Si l'on a le typage t : A, alors, comme pour les variables ci-dessus :

(1) t est un terme de preuve si A est un type, avec A : type ;

(2) t est un terme de type si A = type, et, plus généralement, un terme d'opérateur de types si A est une arité, avec A : arité ;

(3) t un terme d'arité si A = arité.

4. Typage et schémas de construction en CiC

4.1 Typage

En CiC le mot-clé type de CoC par Prop et la nouvelle sorte Set, et le mot-clé arité par Type. De manière pratique, en calcul des constructions inductives, ou CiC :

- Prop est le type des propositions telles (pair n), x = y, etc. ;
- Set est le type des types concrets tels nat, (liste nat) pour les listes d'entiers, (pile bool) pour les piles de booléens, etc., et fonctionnels tels nat -> nat, (liste nat) -> (liste nat) -> (liste nat) , etc. ;
- Type est le type des arités, telles que Prop, Set, Prop -> Prop, nat -> Prop, nat-> Set, etc.

Type est, en première approximation, le "type de tous les types". Ainsi, on a Prop : Type, Set : Type, Prop -> Prop : Type, Type -> nat : Type, etc. Mais, par convention, on a aussi Type:Type.

En réalité, pour éviter les paradoxes, des niveaux de types sont définis à l'aide d'indices. Ainsi, Prop :$Type_i$, Set :$Type_i$, pour tout $i \geq 0$ et Typei : Typej, pour $i < j$.

Mais, de manière pratique, la hiérarchie est aplatie, et les indices sont supprimés, au profit de Type : Type.

4.2 schémas de construction

Le CiC a trois schémas de construction de termes supplémentaires par rapport à CoC :

1) Des minimisateurs, qui sont des définitions de types ou de prédicats inductifs par cas selon une union disjointe ;
2) Des constructeurs qui fabriquent des termes de ces types ou des preuves de ces prédicats selon les différents cas ;
3) Des éliminateurs qui redéstructurent des termes construits selon leurs composants. Pour un type T de type Type, l'éliminateur est noté T_rect. De manière générale, il sert :

- au niveau Prop : à définir un schéma d'induction noté T_ind permettant le raisonnement par cas sur T ;
- au niveau Set : à un schéma de récursion noté T_rec permettant de définir par récurrence sur T des valeurs d'un autre type.

Pour chaque couple (éliminateur, constructeur), le CiC fournit une règle de calcul supplémentaire. Ainsi, la construction de nouveaux types ou prédicats inductifs s'accompagne d'une augmentation de la puissance de raisonnement permettant de les traiter et de les utiliser.

5. Minimisateurs en CiC

Un type concret ou un connecteur logique peuvent être définis de manière inductive, ou par minimisation, symbolisée par µ. Ainsi, en notation mathématique générale, on peut définir

$$nat \stackrel{déf}{=} \mu X : Set.[X \mid X \to X]$$

Comme un "plus petit type" X ayant deux formes de constructeurs, l'un de type X et l'autre de type X → X, bien sûr respectivement pour zéro et la fonction successeur. Ces constructeurs sont considérés comme indépendants et globalement injectifs, c'est-à-dire que des objets représentés par des termes (sans variable) distincts formés avec ces deux constructeurs sont traités comme distincts. Ainsi, les types X satisfaisant cette définition sont tous isomorphes entre eux, et l'on peut parler du " plus petit type" avec la propriété de minimisation ci-dessus.

6. Les constructeurs

Pour chacun des cas de la déclaration d'un type inductif, le constructeur d'objets correspondant est automatiquement engendré comme un terme. Pour bool, les deux constructeurs de booléens, indicés par 1 et 2, sont écrits formellement C (1, bool) : bool et C (2, bool) : bool. Il s'agit en fait des deux constantes booléennes notés habituellement true et false, de manière à les rendre plus lisibles. En réalité, dans le langage de spécification Gallina de Coq, on écrit la définition de bool :

```
Inductive bool : Set :=  true : bool | false : bool.
```

Ce type est d'ailleurs prédéfini exactement comme cela dans le système Coq. Pour nat les deux constructeurs, indicés par 1 et 2, sont C (1,nat) : nat et C(2,nat) : nat → nat, en fait pour zéro et la fonction de succession dans nat.. En Gallina, le type nat est en fait prédéfini comme:

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

7. Définition de fonctions par des schémas d'induction

Le calcul des constructions inductives permet de définir des fonctions totales sur un type inductif comme le plus petit point fixe d'une certaine fonctionnelle. Pour cela, sans avoir besoin d'expliciter la fonctionnelle, on utilise un schéma d'induction comme dans les langages fonctionnels, par exemple Caml ou OBJ3.

Par exemple, en supposant connue la multiplication dans les entiers naturels, on définit en Coq la factorielle fact par :

```
Fixpoint fact(n:nat){struct n}:nat:=
    match n with
       O => (S O)
     |(S n0) => (fact n0) * (S n0)
    end.
```

De même, on peut définir l'addition plus, avec le même type d'effets, par :

```
Fixpoint plus (n:nat) {struct n} : nat -> nat :=
  fun (m:nat) =>
  match n with
      O => m
    |(S n0) => (S (plus n0 m))
  end.
```

On note que l'argument sur lequel porte l'induction figure entre accolades derrière struct. C'est en fait nécessaire seulement s'il y a plus d'un paramètre.

Une telle fonction est ainsi totale sur cet argument et elle est à terminaison finie. En effet, sa définition reposant sur le principe d'induction structurelle, les appels récursifs de plus se font sur des termes entiers de plus en plus petits. Nous verrons par la suite des cas plus compliqués, par exemple avec une induction sur deux arguments, pris successivement ou conjointement.

8. Eliminateurs, récurseurs

Des types d'élimination pour conduire des raisonnements sont engendrés automatiquement sur les types inductifs. Sans entrer précisément dans leur mécanique, nous insistons ici sur une de leurs conséquences : la définition automatique d'éliminateurs et de récurseurs sur les types inductifs.

8.1 Eliminateurs

Pour chaque type inductif T, un éliminateur, noté T_rect, est automatiquement engendré. Cet éliminateur sert à "démonter" tout objet de type T pour retrouver et agir sur ses composants. Il sert notamment à réaliser des récurseurs, comme à la section suivante.

- Booléens : L'élimination des booléens de type bool, est réalisée en Coq par bool_rect prédéfini comme étant :

```
bool_rect =
 fun (P : bool->Type) (f:P true) (f0:P false) (b:bool) =>
     if b as b0 return (P b0) then f else f0
 : forall P : bool -> Type, P true -> P false ->
     forall b : bool, P b
```

Dans cette écriture, l'expression conditionnelle if b as b0 return (P b0) then f else f0 renvoie, selon les valeurs true ou false de b renommé b0, un objet f de type (P true) ou un objet f0 de type (P false). L'annotation return (P b0), où (P b0) est appelé type de retour, contribue à donner le type de toute l'expression conditionnelle.

C'est nécessaire ici car les types de f et f0 sont différents et dépendent des valeurs de b, ou b0. Alors, la conditionnelle est ici considérée comme du type dépendant (P b0). Tout se passe en fait comme si cette définition de bool rect engendrait une nouvelle règle d'élimination, pour tout P :bool → Type

$$(E_{\texttt{bool}})\ \frac{f : P\ true \quad f0 : P\ false}{(bool\_rect\ P\ f\ f0) : \forall b : bool, P\ b}$$

- Entiers naturels : L'élimination des entiers de type nat, est réalisée en Coq nat_rect prédéfini comme :

```
nat_rect =
  fun (P : nat -> Type) (f : P 0)
      (f0 : forall n : nat, P n -> P (S n)) =>
   (fix F (n : nat) : P n :=
     match n return P with
        | 0 => f
        | S n0 => f0 n0 (F n0)
     end)
 : forall P : nat -> Type,
   P 0 -> (forall n : nat, P n -> P (S n)) ->
     forall n : nat, P n
```

Dans cette écriture,

```
(fix F (n : nat) : P n :=
   match n return P with
      | 0 => f
      | S n0 => f0 n0 (F n0)
   end)
```

La définition récursive ”au vol” d’une fonction, nommée en interne F, d’un n : nat dans le type dépendant (P n). En outre, comme ci-dessus, l’expression match n return P with ... end comporte un type de retour P, qui contribue à lui donner le type dépendant (P n). Tout se passe comme si cette définition de nat_rect engendrait une nouvelle règle d’élimination, pour tout P : nat → Type :

$$(E_{\mathbf{nat}})\ \frac{f : P\ O \quad f0 : \forall n : nat, P\ n \rightarrow P\ (S\ n)}{(nat\_rect\ P\ f\ f0) : \forall n : nat, P\ n}$$

- Conjonction : L’élimination des conjonctions de type (and A B), où A, B :Prop est réalisée en Coq par and_rect prédéfini comme :

```
and_rect :=
  fun (A B : Prop) (P : Type)
      (f : A -> B -> P) (a : A /\ B) =>
    match a return P with
      | conj x x0 => f x x0
    end
  : forall (A B : Prop) (P : Type),
      (A -> B -> P) -> A /\ B -> P
```

Notons qu’ici, l’annotation return P est faite sur le type P : Type, qui n’est pas un type dépendant. La règle correspondante s’écrit, pour tout P : Type :

$$(E_{\mathbf{and}})\ \frac{f : A \rightarrow B \rightarrow C \quad a : A \wedge B}{(and\_rect\ A\ B\ P\ f\ a) : P}$$

En réalité, la plupart du temps, Coq n’admet pas que les types construits sur des propositions aient des éliminateurs généraux. Ainsi, bien que or soit défini inductivement, or_rect n’existe pas.

8.2 Récurseurs

Pour chaque type inductif T, des récurseurs, notés T_ind, pour l'induction dans des propositions de type Prop, et T_rec, pour la récursion dans des ensembles de type Set, sont donnés automatiquement sur les termes de T. Ils sont construits à partir de l'éliminateur T_rect quand il existe, ou bien directement sinon.

Ils servent à engendrer, pour Prop, des termes de preuves de propositions, et, pour Set, des témoins de types, notamment des fonctions.

- Booléens : Pour les booléens de type bool, on a le principe d'induction suivant, défini explicitement en Coq à partir de l'éliminateur général bool_rect vu auparavant par :

```
bool_ind :=
  fun P : bool -> Prop => bool_rect P
  : forall P : bool -> Prop, P true -> P false ->
      forall b : bool, P b
```

Noter que l'on applique bool_rect à P : bool -> Prop, alors qu'il a été défini sur bool -> Type. En fait, en Coq, un objet de type Prop (par exemple True ou False) ou de type Set (par exemple bool ou nat) est également considéré comme de type Type. Ceci correspond évidemment à une nouvelle règle d'élimination obtenue à partir de (E_{bool}) en prenant :

```
P : bool -> Prop :
```

$$(E_{\texttt{bool\_ind}})\ \frac{f : P\ true \quad f0 : P\ false}{(bool\_ind\ P\ f\ f0) : \forall b : bool, P\ b}$$

Cette règle permet les preuves par induction sur les booléens.

De manière analogue, on a le récurseur pour aller des booléens vers un type concret quelconque :

```
bool_rect =
 fun (P : bool->Type) (f:P true) (f0:P false) (b:bool) =>
     if b as b0 return (P b0) then f else f0
 : forall P : bool -> Type, P true -> P false ->
     forall b : bool, P b
```

Il est utilisé pour définir récursivement des fonctions sur les booléens. Ainsi, si l'on instancie P avec fun (b :bool) => nat, on veut prouver que forall b : bool, nat, c'est-à-dire bool -> nat (puisque nat ne dépend pas de b), est un type habité, ce que l'on obtient en exhibant un témoin de ce type, c'est-à-dire une fonction de bool dans nat, à l'aide du récurseur. On forme :

```
Definition Cas_rec := (bool_rec (fun (b:bool)=>nat)).
```

Exemple 2 (Codage de booléens)

Par exemple, quand on fixe deux entiers, disons 1 et 0, pour les valeurs booléennes true et false, alors on construit une fonction de codage cod des booléens en entiers par :

```
Definition cod := (Cas_rec 1 0).
```

On a bien sûr `(cod true)` $\xrightarrow{*}_{\beta}$ `1` et `(cod false)` $\xrightarrow{*}_{\beta}$ `0`.

- Entiers naturels : De manière analogue aux booléens, pour les entiers naturels de type nat, on a :

```
nat_ind :=
  fun P : nat -> Prop => nat_rect P
  : forall P : nat -> Prop,
    P 0 -> (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

Il correspond à la règle d'élimination, pour tout P : nat → Prop :

$$(E_{\texttt{nat\_ind}})\ \frac{f : P\ O \quad f0 : \forall n : nat, P\ n \rightarrow P\ (S\ n)}{(nat\_ind\ P\ f\ f0) : \forall n : nat, P\ n}$$

De manière analogue, on a le récurseur pour aller des entiers vers un type concret quelconque:

```
nat_rec :=
  fun P : nat -> Set => nat_rect P
  : forall P : nat -> Set,
    P 0 -> (forall n : nat, P n -> P (S n)) ->
        forall n : nat, P n
```

Ce récurseur est utilisé pour définir des fonctions totales sur les entiers. Ainsi, si l'on instancie P avec fun (n :nat)=>nat, on veut prouver que forall n :nat, nat, ou bien nat -> nat (puisque nat ne dépend pas de n), est un type habité, ce que l'on obtient en exhibant un témoin de ce type, c'est-à-dire une fonction de nat dans nat, à l'aide du récurseur. On forme :

```
Definition Prim_rec := (nat_rec (fun (n:nat)=>nat)).
```

Typé par

```
nat_rec :=
  fun P : nat -> Set => nat_rect P
  : forall P : nat -> Set,
    P 0 -> (forall n : nat, P n -> P (S n)) ->
        forall n : nat, P n
```

Ce type étant β-équivalent à :

```
nat -> (nat -> nat -> nat) -> nat -> nat
```

Prim_rec est l'opérateur de récursion primitive sur les entiers. Ainsi, dans une définition de fonction f par

```
Definition f := (Prim_rec m g)
```

m est l'entier correspondant à la valeur (f 0) et g est la fonction de succession de type nat -> nat -> nat telle que f (S p) = g p (f p). On peut aussi définir des opérations comme l'addition et le prédécesseur dans nat (où, par convention, 0 est prédécesseur de lui-même) par :

```
Definition add (n m:nat) :=
   (Prim_rec m (fun (p:nat)(r:nat)=>(S r)) n).

Definition pred (n:nat) :=
   (Prim_rec 0 (fun (p:nat)(r:nat)=>p) n).
```

Ici, r est le résultat de l'appel récursif.

Conclusion

Le choix du formalisme est important pour la pratique des mathématiques formelles. Non seulement il doit être capable d'exprimer la preuve conçue par le mathématicien, mais il doit permettre de le faire de la manière la plus facile et la plus intuitive possible.

Le Calcul des Constructions Inductives est lui une extension de la logique d'ordre supérieur, il est à la base du système d'assistance à la preuve Coq développé par l'INRIA. Ce formalisme qui étend à la fois la logique d'ordre supérieur de Church et les systèmes de Martin-Löf est particulièrement expressif du point de vue algorithmique et peut facilement être mis en œuvre sur ordinateur.

Partie2

Système Coq

Introduction

Pour introduire la théorie du système Coq, il serait intéressant de présenter un aperçu sur la théorie des types et la logique constructive, et aussi de donner une idée sur l'interprétation des λ-termes comme des preuves et les types assignés à ces termes comme des formules. Cette dernière est connue sous le nom de l'isomorphisme de Curry-Howard introduite par Howard [13] en se basent sur une idée de Curry [14].

Cette interprétation couplée avec le développement de l'informatique sera exploitée en utilisant la puissance de calcul des machines pour construire des preuves, de vérifier automatiquement ces preuves et aussi d'arriver à extraire, à partir de la preuve de la consistance de la spécification des programmes informatiques certifiés sans erreurs.

1. les systèmes de preuve

Ces dernières années, le domaine de recherche consacré à l'automatisation des preuves sur ordinateur a connu un développement important. Ceci a donné comme résultat l'apparition de plusieurs systèmes de preuves qui peuvent être classés en trois catégories :

1. Les vérificateurs de preuve (proof-checkers) : dans ces systèmes la preuve est construite par un utilisateur et sa validité est vérifiée par la machine. Automath [15] et Mizar [16] sont les systèmes les plus répandus dans cette classe de vérificateurs.

2. Les démonstrateurs Automatiques : la preuve est effectuée par la machine sans intervention du programmeur. Leurs domaines d'application sont réduits.

3. Les assistants de preuve sont des systèmes interactifs dans lesquels l'utilisateur guide la démonstration à l'aide des fonctions appelées les tactiques dont les parties faciles sont démontrées automatiquement. Parmi les premiers systèmes d'aide à la preuve le système LCF (Logic for Computable Function) [17] de Robin Milner qui introduisit les notions des tactiques. Pour écrire ces tactiques, Milner introduisit un meta-langage qui évolua par la suite pour donner naissance aux langages de la famille ML. Les autres systèmes comme HOL , PVS , Isabelle , Lego , Alf et Coq sont basés sur les mêmes concepts que ceux de LCF.

2. L'assistant de preuve Coq

Coq est un outil d'aide à la preuve, basé sur une logique d'ordre supérieure appelée calcul des constructions inductives (CIC). La première implémentation date de 1984 et fut réalisée en CAML par Gérard Huet et Thierry Coquand. Ce système peut être vu sous deux aspects :

- Langage de programmation : il permet en effet de générer des programmes qui sont conformes à leurs spécifications formelles.

- Assistant de preuve : il permet de construire des démonstrations de théorèmes.

Coq se compose essentiellement de 3 parties qui sont :

- Le langage logique : Appelé Gallina. Ce langage permet de définir un grand nombre d'objet mathématique tels que les prédicats, les définitions, … qui lui serviront à modéliser des théories ou spécifier des programmes.

- Le démonstrateur de théorèmes : Il permet de construire des démonstrations de manière assistée par la machine. La certification de programme consiste à valider l'adéquation entre un programme et sa spécification. En cas de succès, le programme résultant peut être vu comme la preuve de sa spécification.

- L'extracteur de programmes : Ce module de Coq permet d'extraire un programme sans erreur à partir de la preuve de la consistance de sa spécification.

2.1 Le langage de spécification Gallina

Le langage Gallina repose sur le CIC [18] [24] [29].Dans ce calcul tous les objets ont un type. Il y a des types pour les fonctions (ou les programmes), les données, les preuves et les types eux-mêmes possèdent un type. Par exemple, on ne permet pas la déclaration "pour tout x, P", on doit dire au lieu de cela : "Pour tout x appartenant à T, P". L'expression "x appartenant à T" est écrite "x:T ". On dit aussi : "x de type T". Pour manipuler l'environnement Coq, on dispose d'un ensemble de commandes qui permettent la spécification des différents objets.

En général, le langage de commande de Gallina est défini par la syntaxe décrite par la figure 3 :

Sentence ::= declaration
| definition
| statement
| inductive
| fixpoint
| statement proof

params ::= typed_idents ; ... ; typed_idents

declaration ::= Axiom ident : term .
| declaration_keyword params .

declaration_keyword ::= Parameter | Parameters
| Variable | Variables
| Hypothesis | Hypotheses

definition ::= Definition ident [: term] := term .

| Local ident [: term] := term .

inductive ::= [Mutual] Inductive ind_body with ... with ind_body .
| [Mutual] CoInductive ind_body with ... with ind_body .

ind_body ::= ident [[params]] : term := [constructor | ... | constructor]

constructor ::= ident : term

fixpoint ::= Fixpoint fix_body with ... with fix_body .
| CoFixpoint cofix_body with ... with cofix_body .

statement ::= Theorem ident : term .
| Lemma ident : term .
| Definition ident : term .

proof ::= Proof . ... Qed .
| Proof. ... Defined .

Fig.3 Langage de commande du Gallina.

2.1.1 Les termes

La notion de terme constitue une catégorie syntaxique très générale dans le langage de spécification Gallina [19]. Elle correspond à la notion intuitive d'expression bien formée prenant en compte les règles de construction du langage.

2.1.2 Les sortes

Les types sont vus comme des termes du langage et alors devraient appartenir à un autre type. Le type d'un type est toujours une constante du langage appelé une sorte. Les deux sortes de base dans le langage du CIC sont Prop et Set. Prop est le type des propositions logiques et Set est le type des spécifications qui inclut les programmes et les ensembles usuels comme les booléens, les naturels, et les listes etc.

Ces sortes eux-mêmes peuvent être manipulées comme des termes ordinaires. Par conséquent on devrait donner aussi un type aux sortes car la supposition que Set de type Set mène à une théorie inconsistante. Il y'a en plus de Set et Prop une hiérarchie des Type(i) pour n'importe quel nombre entier i. Nous appelons S l'ensemble des sortes défini par :

S = {Prop, Set, Type (i) | i /N}. Les sortes ont les propriétés suivantes: Prop: Type(0) et Type(i):Type (i+1).

2.1.3 Le langage

Un terme est un type ou une variable ou une constante de l'environnement. Comme d'habitude dans le λ-calcul, nous combinons des objets en utilisant l'abstraction et l'application.

Plus précisément le langage du Calcul des Constructions Inductives est construit à partir des règles suivantes :

1. Les sortes Set, Prop et Type sont des termes.

2. les constantes de l'environnement sont des termes.

3. Les variables sont des termes.

4. Si x est une variable et T, U des termes alors (x:T) U est un terme. Si x se produit dans U, (x:T) U est lu "pour tout x de type T, U". Si U dépend de x, on dit que (x:T) U est un produit dépendant. Si x n'appartient pas à U alors (x:T) U est lu "si T alors U ".Un produit non dépendant peut être écrit: T → U.

5. Si x est une variable et T, U sont des termes alors [x:T] U est un terme. C'est une notation pour β- abstraction.

6. Si T et U sont des termes alors (T U) est un terme. Le terme (T U) se lit "T appliqué à U".

La table suivante représente la syntaxe des termes dans le système Coq :

```
term          ::= ident
              | sort
              | term -> term
              | ( typed_idents ; ... ; typed_idents ) term
              | [ idents ; ... ; idents ] term
              | ( term ... term )
              | [annotation] Cases term of [equation | ... | equation] end
              | Fix ident { fix_body with ... with fix_body }
              | CoFix ident { cofix_body with ... with cofix_body }

sort          ::= Prop
              | Set
              | Type

annotation    ::= < term >

typed_idents  ::= ident , ... , ident : term
idents        ::= ident , ... , ident [: term]

fix_body      ::= ident [ typed_idents ; ... ; typed_idents ]: term := term
cofix_body    ::= ident : term := term

simple_pattern ::= ident
              | ( ident ... ident )

equation      ::= simple_pattern => term
```

Fig.4. Syntaxe du terme.

2.1.4 Le contexte

Un terme bien typé dépend d'un ensemble de déclarations et de variables appelé contexte. Un contexte Γ écrit [x1 : T1, x2 : T2,…, xn : Tn] où les x_i sont des variables distinctes et les T_i sont des termes.

2.1.5 Environnement

Parce que nous manipulons les constantes, nous devons aussi considérer un environnement E, qui est un ensemble de définitions des constantes.

2.2 Les types atomiques et les types composés

Les types peuvent être classés en deux catégories : les types atomiques et les types composés.

a) Types atomiques : Réduits à un seul identificateur comme nat, Z et bool.

b) Types composés (flèches) : Des types de la forme (A→B) où A et B sont deux types. Ces types forment un cas particulier d'une construction appelée produit dépendant.

2.3 Les règles

Les règles pour présenter des nouveaux objets dans l'environnement :

1. Les Sortes sont des termes

1) Prop a un type Type :

$$\frac{E[\Gamma]}{E[\Gamma] \vdash \text{Prop} : \text{Type}(p)}$$

En Coq :

```
Coq < Check Prop.
Prop
     :Type
```

2) Set est de type Type :

$$\frac{E[\Gamma]}{E[\Gamma] \vdash \text{Set} : \text{Type}(q)}$$

En Coq :

```
Coq < Check Set.
 Set
     : Type
```

3) Type est de type Type

$$\frac{E[\Gamma] \quad i<j}{E[\Gamma] \vdash \text{Type}(i) : \text{Type}(j)}$$

En Coq :

```
Coq < Check Type.
Type
     : Type.
```

2. Expressions réduites à un identificateur

La forme syntaxique la plus simple d'expression est une simple variable ou une constante x. Un tel terme ne peut être accepté que si x est déclarée dans le contexte ou l'environnement courant. Soit A le type de x dans cette déclaration, alors le terme réduit à x a pour type A.

$$Var \quad \frac{(x:T)\in\Gamma}{E[\Gamma] \vdash x : T}$$

$$Cons \quad \frac{(c:T) \in E}{E[\Gamma] \vdash c : T}$$

La règle Var se lit : si la variable x est spécifiée de type A dans le contexte Γ, alors le terme x a pour type A dans cet environnement et ce contexte.

Exemple 1

```
Coq < Check 0.
0
  : nat.
```

```
Coq < Check true
true
     : bool
```

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

3. Application

Soient un environnement E et un contexte Γ et deux expressions t et u de types respectifs T→ U et T dans E [Γ] alors l'application de t à u est le terme qui s'écrit (t u).Ce terme est de type U dans le contexte et l'environnement considérés.

$$App \quad \frac{E[\Gamma] \vdash t : T \text{->} U \quad E[\Gamma] \vdash u : T}{E[\Gamma] \vdash (t\ u) : U}$$

```
Coq < Check (plus O).
(plus O)

     : nat->nat
```

```
Coq < Check (plus O O).
(plus O O)

     : nat
```

4. Abstraction

L'abstraction est un moyen simple de construire des fonctions en associant un paramètre formel et une expression. La fonction qui à tout x de type T associe l'expression U se note [x:T]t en Gallina.

$$Lam \quad \frac{E[\Gamma::(x:T)] \vdash t : U}{E[\Gamma] \vdash [x:T]t : T \to U}$$

```
Coq < Check [n :nat] n .
[n:nat] n
        : nat->nat
```

5. Produit

Un produit dépendant est un type de la forme (v:T) U où T et U sont des types et v est une variable liée dont la portée est le type U. La variable v peut avoir des occurrences libres dans le type U. Le produit dépendant se lit pour tout v de type T, U.

2.4 Règles de conversion

Les conversions utilisées dans Coq sont de quatre sortes :

1. La ϒ-conversion (delta-réduction) : Soit t un terme et v un identificateur défini par t' dans l'environnement ou le contexte courant, alors la ϒ-conversion transforme le terme t en t {v/t'}.

2. La β-réduction (beta-réduction) : Permet de transformer un β-radical c'est à dire un terme de la forme ([v : T] t u) en le terme t{v/u}.

```
Coq < Definition one := (plus O (S O)).
one is defined
Coq < Eval Cbv  Delta   [one] in [n : nat] (S(plus (one) n)).
     = [n:nat](S (plus (plus O (S O)) n))
     : nat->nat
```

3. La ζ-réduction (zeta-réduction) : Consiste en l'élimination des liaisons locales, plus précisément, elle remplace une expression de la forme [v:=u]t par t{v/u}.

4. La ι-réduction :(ita-réduction) : Assure le mécanisme de filtrage au type défini par induction.

2.5 Les définitions et les déclarations

Comme dans la plupart des langages de programmation, il est d'usage de distinguer les notions de définition et déclaration, ainsi que leur portée locale ou globale. Une déclaration permet d'attacher un type à un identificateur sans lui donner de valeur. Comme dans les fichiers d'interface de C, Java ou ML, on peut déclarer par exemple la variable max_nat est de type nat , sans lui attacher de valeur précise.

```
Coq < Variable max_nat : nat.
max_nat is assumed
```

En revanche, une définition donne une valeur à un identificateur sous la forme d'un terme associé. Dans la mesure où l'on peut déterminer le type de ce terme, une définition précise à la fois le type et la valeur d'un identificateur.

```
Coq < Definition one := (plus O (S O)).
one is defined
Coq < Eval Cbv  Delta   [one] in [n : nat] (S(plus (one) n)).
     = [n:nat](S (plus (plus O (S O)) n))
     : nat
```

Par conséquent, toute définition peut aussi jouer le rôle d'une déclaration, et un énoncé concernant une déclaration quelconque s'applique donc à une définition en oubliant la valeur qu'elle précise. La portée d'une déclaration ou d'une définition peut être soit globale, c'est à dire tout le reste du développement, soit locale, c'est à dire restreinte à une sous-expression ou à une section (nom donné en Coq à un mécanisme similaire aux blocs des langages de programmation). À tout point d'un développement en Coq sont associés à la fois un environnement et un contexte dits courants.

2.6 Structures de données inductives

La définition de types inductifs en Gallina étend les différentes notions de définitions de types fournies dans les langages de programmation. On peut les comparer aux définitions de types récursifs dans les langages fonctionnels ML,Ocaml, Haskell . Mais la possibilité de mélanger types récursifs et produits dépendants rend les types inductifs de Gallina beaucoup plus précis et expressifs. À chaque type de données inductif correspond une structure de calcul, basée sur le filtrage et la récursion. Ces structures de calcul sont le noyau de la programmation récursive en Gallina.

2.6.1 Types sans récursion

Les types inductifs les plus simples sont les types énumérés, utilisés pour décrire des ensembles finis. L'exemple le plus fréquemment utilisé d'un tel ensemble fini est celui des valeurs booléennes, qui contient seulement deux éléments. On peut former la définition de ce type comme suit :

```
Inductive bool : Set := true : bool
                      | false : bool .
```

2.6.2 Types avec récursion

Les types inductifs sans récursion permettent de décrire toute sorte de données, mais toujours des données dont la taille est connue à l'avance. Il est nécessaire de pouvoir raisonner sur des structures de données dont la taille peut varier, par exemple des tableaux de données de taille non définie à l'avance.

La récursion fournit une solution extrêmement simple. On exprime que certaines données comportent des fragments qui sont de même nature que ces données elles-mêmes. Par exemple, si l'on considère les arbres binaires, ils peuvent être des feuilles ou des arbres composés. Si ce sont des arbres composés, alors ils contiennent deux fragments qui sont eux aussi des arbres binaires, donc des objets de même nature que l'arbre lui-même.

Ces types de données représentent des ensembles infinis, où la construction de chaque élément est faite en un nombre fini d'étapes. Cette caractéristique nous permettra de disposer d'un moyen de raisonnement et de calcul systématique pour chacun de ces types.

La preuve par récurrence (proof by induction) sera le moyen de raisonnement et la construction de fonctions récursives sera le moyen de calcul.

2.7 Le type des entiers naturels

La présentation formelle la plus naturelle des nombres naturels est inspirée des travaux de Peano. Tout nombre naturel peut être obtenu, soit en prenant le nombre 0, soit en appliquant la fonction successeur à un nombre déjà construit. Dans le système Coq, ceci va s'exprimer par la définition suivante:

```
Inductive nat : Set := O : nat
                     | S : nat -> nat.
 nat_ind is defined
 nat_rec is defined
 nat_rect is defined
 nat is defined
```

En effet, il n'existe que deux méthodes pour construire un nombre naturel. Soit on prend le nombre 0 (Zéro), soit on prend un nombre x déjà construit et l'on en construit un nouveau grâce à la fonction S.

Pour une propriété P donnée, si l'on arrive à démontrer que cette propriété est bien satisfaite par 0 et que si l'on prend un nombre x qui la satisfait alors le nombre construit (S x) la satisfait également alors, tout nombre naturel satisfait P. Ceci s'exprime à l'aide d'un principe de récurrence, dont l'énoncé mathématique est le suivant:

$$\forall P.(P\ (0)\ \wedge(\forall x.P\ (x)\ \rightarrow P\ (S(x)))\rightarrow \forall x.P\ (x)$$

Dans le système Coq, cet énoncé s'écrit de la manière suivante:

```
nat_ind :
     (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n))) ->(n:nat)(P n)
nat_rec :
     (P:(nat->Set))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
nat_rect :
     (P:(nat->Type))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

En pratique le système Coq engendre automatiquement le principe de récurrence nat_ind, nat_rec et nat_rect à partir de la définition inductive.

2.8 Types co-inductifs

Comme pour les types inductifs, un type co-inductif est spécifié par la signature (le type) de ses constructeurs. Les types co-inductifs se distinguent des types inductifs par le fait que leurs habitants contiennent une infinité de constructeurs, il faut signaler qu'à la suite de la définition d'un co-inductif, le système ne génère pas le principe d'induction et donc pour raisonner sur les habitants d'un tel type on disposera seulement l'analyse par cas.

```
Variable A : Set.
CoInductive Set Stream  := Cons : A->Stream->Stream.
```

2.9 Les fonctions récursives

Une fonction se définit généralement en indiquant la valeur qu'elle retourne pour un paramètre donné. On dit la fonction qui associe à x l'expression e. La variable x est autorisé à apparaître dans l'expression e, ce qui permet d'assurer que la fonction n'est pas constante. Pour une fonction récursive, la fonction f qui associe à x l'expression e, avec la possibilité que f apparaisse aussi dans l'expression e. En d'autres termes, on suppose que la fonction f est déjà partiellement définie lorsque l'on essaie de déterminer la valeur qu'elle retourne pour une nouvelle valeur du paramètre.

En Coq, l'utilisateur doit déterminer les valeurs prises par la fonction dans un ordre précis, qui suit l'ordre dans lequel on a construit les termes des types inductifs.

Pour les nombres naturels, on est obligé de construire O avant (S O), avant (S (S O)) et ainsi de suite. C'est cet ordre de construction des nombres qui est suivi pour la définition d'une fonction récursive f, de sorte que l'on s'autorise à utiliser la valeur (f O) pour définir la valeur (f (S O)), la valeur (f (S O)) pour définir la valeur (f (S (S O))) et ainsi de suite. De manière générale, on s'autorise à utiliser la valeur (f n) pour définir la valeur (f (S n)).

En pratique, ce procédé de construction est fourni dans le système Coq par la commande Fixpoint qui prend la forme suivante dans le cas des nombres naturels :

Fixpoint f [n:nat] : t := expr.

Dans cette définition, f est le nom de la fonction que l'on est entrain de définir, n est le nom de l'argument sur lequel la récursion s'organise, t est le type retourné et expr indique comment la valeur de (f n) est déterminée. Voici une fonction récursive simple sur les entiers naturels, qui multiplie son argument par 2.

```
Fixpoint mult2 [n:nat] : nat :=
Cases n of
O => O
| (S p) => (S (S (mult2 p)))
end.
```

Cette fonction fait bien apparaître un appel récursif dans la sous-expression (mult2 p). Par l'intermédiaire du traitement par cas sur n, cet appel récursif est utilisé lorsque l'on veut déterminer la valeur de mult2 sur l'argument (S p). Les fonctions récursives définies avec Fixpoint contiennent toujours un filtrage sur l'argument de la fonction, de sorte que l'on est amené à donner d'abord la valeur de la fonction lorsque cet argument est O, puis la valeur de la fonction lorsque l'argument est de la forme (S p) avec la possibilité d'utiliser la valeur de la même fonction en p.

3. Manipulation des preuves

Dans cette partie, nous abordons les techniques de raisonnement en Coq, en commençant par la syntaxe de présentation des théorèmes :

```
Theorem < nom_du_théorème > : < énoncé >.
Lemma < nom_du_lemme>     :       < énoncé >.
Remark < nom_de_la_remarque > : < énoncé >.
```

Cet énoncé est appelé le but initial, d'une manière générale un but est la donnée d'une formule à prouver sous certaines hypothèses, celles-ci forme le contexte local d'hypothèses du but. Dans le cas d'un but initial, le contexte initial est vide.

Le développement d'une preuve se fait d'une manière interactive par l'usage de commandes appelées tactiques.

1. La tactique

Une tactique est une fonction qui est appliquée à un sous-but :

- Soit réussit en produisant les deux informations suivantes :
    - Le résoudre c'est à dire terminer la preuve de sous-but courant et afficher le nouveau sous-but à prouver.
    - Le transformer en un autre sous-but avec un contexte différent.
    - Le transformer en plusieurs sous-buts, chacun avec son contexte.
- Soit échouer, en laissant le sous-but inchangé.

Les tactiques peuvent être combinées en utilisant des opérateurs appelés tacticals.

2. Les tactiques de base

Nous présentons trois tactiques simples associées aux règles de typage Var, App et Lam.

1. La tactique Assumption s'utilise pour résoudre un sous-but T lorsque le contexte H contient une déclaration (v : T) (en terme de raisonnement, on dirait que la propriété qu'on veut prouver fait déjà l'objet d'une hypothèse).

2. La tactique Intro : Associe à la règle de typage Lam qui permet d'introduire les hypothèses et des variables quantifiées dans le contexte du but courant

- Si le but courant est un produit dépendant (x :A)B, le nouveau but sera b et la variable x de type A sera poussée dans le contexte local.
- Si le but est un produit non dépendant A → B une nouvelle hypothèse Hi : A est introduite dans le contexte et le but courant devient B.

3. La tactique Apply : Réduit le but à prouver en d'autre plus élémentaire par l'application d'un axiome ou un théorème déjà prouvé. L'exemple 2 montre l'utilisation de ces tactiques. Construire une preuve revient à montrer un λ-terme du type attendu. Voici une illustration de leur principe, sur des exemples, on se donne trois variables propositionnelles A, B et C, puis l'on cherche à prouver la tautologie ((A→B) →C) →B→C.

Exemple 2

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

Coq < Lemma exemple5 : ((A->B)->C)->B->C.
1 subgoal

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

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

```
Coq < Intros c b.
1 subgoal

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

On voit que (A→B) →C et B ont été poussées comme hypothèses dans le contexte local, au-dessus de la double barre. Elles ont été nommées comme demandé. Du point de vue de la construction de la preuve, cette tactique correspond à la λ-abstraction. C'est- à dire que la preuve vide "?" a été remplacée par la preuve incomplète [c:(A→B)-→C][b:B]? .

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

```
Coq < Apply c.
1 subgoal

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

On est alors passé au terme de preuve partiel [c:(A→B) →C][b:B](c ?) où le but courant est maintenant A→B. On fini donc la preuve par :

```
Coq < Intros a.
1 subgoal

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

Coq < Exact b.
Subtree proved!
```

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

```
Coq < Qed.
Intros c b.
Apply c.
Intros a.
Exact b.
exemple5 is defined
Coq < Print exemple5
Exemple5 = [c:((A->B)->C); b:B](c [_:A]b)
     : ((A->B)->C)->B->C
```

3. Quelques autres tactiques

Split : Transforme le but A /\ B en les deux buts A et B.

Left : Transforme le but A \/ B en A.

Right : Transforme le but A /\ B en B.

Exists u : Transforme le but Ex ([x: M] N) en N{x/u} (remplacement de toutes les occurrences libre de x dans N par u).

Elim H : Permet de faire des preuves par induction, cette tactique applique le principe d'élimination associe à la définition inductive de H.

Unfold nom : Remplace dans le but les occurrences de nom par sa définition.

Exemple 3

```
Coq < Lemma et_est_commutative: (A /\ B) -> (B /\ A) .
1 subgoal

  ============================
   A/\B->B/\A

et_est_commutative < Intro.
1 subgoal

  H : A/\B
  ============================
   B/\A

et_est_commutative < Elim H.
1 subgoal

  H : A/\B
  ============================
   A->B->B/\A

et_est_commutative < Intros.
1 subgoal

  H : A/\B
  H0 : A
  H1 : B
  ============================
   B/\A

et_est_commutative < Split.
2 subgoals

  H : A/\B
  H0 : A
  H1 : B
  ============================
   B

subgoal 2 is:
 A

et_est_commutative < Assumption.
1 subgoal

  H : A/\B
  H0 : A
  H1 : B
  ============================
   A

et_est_commutative < Assumption.
Subtree proved!
```

4. L'extraction d'un programme ML

L'extraction des programmes est entièrement prise en charge par le système Coq. Elle s'effectue en deux étapes :

- Extraction du λ-terme correspondant à la preuve après la suppression de l'information logique.

- Traduction de ce λ-terme en un programme écrit en langage fonctionnel (Caml, Ocaml ou Haskell). En effet, la preuve de la consistance de la spécification contient à la fois l'information logique (qui nous assure que la démonstration est correcte) et l'information dite calculatoire (qui permet de construire la fonction ou le programme désiré). C'est cette dernière qui nous intéresse pour la génération du programme.

Apres le chargement du module de l'extraction par la commande

```
Require Extraction.
```

La génération du code de programme dans le fichier nom_du_fichier par la commande :

```
Write Caml File " nom_du_fichier " [Liste_des_constantes].
```

Liste_des_constantes est la liste des théorèmes et/ou des lemmes que l'on voudra extraire.

Conclusion

Coq est avant tout un système permettant de faire des preuves dans une logique très expressive, dite d'ordre supérieur. Ces preuves sont construites de façon interactive, assistée par des outils de recherche automatique de preuves dans des domaines ou cela est possible. Les domaines d'utilisation de Coq sont très varies : logique, automates, syntaxe et sémantique des langues naturelles, algorithmique, etc.

Coq fait partie d'une longue tradition de systèmes informatiques d'aide à la démonstration de théorèmes. Citons par exemple les systèmes Automath , Isabelle, Lego, HOL, PVS …etc

Un des points les plus remarquables de Coq est la possibilité de synthétiser des programmes certifiés à partir de preuves.

Chapitre 3

Spécification de système.

Chapitre 3

Spécification de Système

Introduction

Les systèmes de transitions (automates et réseaux de Petri) sont les principaux modèles utilisés pour spécifier le comportement de systèmes réactifs et/ou temps réel. Un **système de transitions** est un **automate**. Le terme système de transitions est surtout utilisé par ceux qui s'intéressent à la modélisation de systèmes. Le terme automate est plus général.

Certes, les automates constituent des techniques de base que l'informaticien se doit de connaître. Ils sont utilisés essentiellement dans la théorie des langages et dans la modélisation et vérification de systèmes communicants (qui englobent les systèmes temps réel). La notion d'automates est utilisée dans différentes méthodes de conception (UML, SDL et autres), avec des notations graphiques souvent différentes d'une méthode à l'autre.

Les automates offrent des mécanismes qui sont à la fois formels et simples à comprendre et à utiliser (surtout lorsqu'ils sont utilisés sous forme graphique). Ils servent aussi à vérifier formellement des propriétés sur les systèmes modélisés.

Afin de donner la spécification d'un système, on s'intéresse aux systèmes de transitions et aux automates.

1. Automate [7]

Un automate est une machine ayant un nombre fini d'état et de transition, une transition consiste en un passage d'un état à un. De manière simplifiée, un système se trouve dans un état initial et change d'état en fonction de l'arrivée des signaux (événements) qui lui parviennent de son environnement.

❖ Définitions

Un automate est un quadruplet A = (S, s_0, E, T, F), où :

- S = ensemble d'états.
- S_0 = ensemble des états initiaux de l'automate ($S_0 \subseteq S$).
- E = ensemble fini des étiquettes des transitions.
- T = ensemble des transitions ($T \subseteq S \times E \times S$).
- $F \subseteq S$ l'ensemble des états finaux.

Un automate est souvent représenté de manière graphique (c'est d'ailleurs l'un des atouts des automates) où les cercles désignent les états et les flèches les transitions.

Exemple 1

S = {A, B, C}

E = {a, b, c, d}

T = { (A, a , B), (A, b , C), (A, b , A), (A, c , A), (A, d , A), (B, c , C), (B, a, B), (B, b , B), (B, d , B), (C, d , A), (C, a , C), (C, b , C), (C, c, C)}

S_0 = {A}

F = {C}.

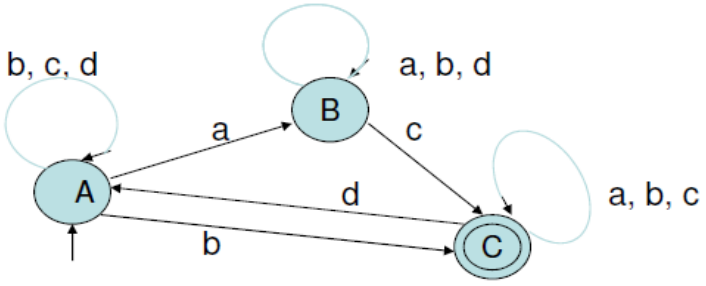


Fig.5. Graphe représentant un automate.

2. Système de transition [7]

Les systèmes de transitions sont un formalisme permettant de décrire l'évolution d'un système informatique dans le cadre du modèle de traces. C'est une machine abstraite construisant les séquences d'états possibles au cours d'une exécution. Il existe plusieurs types de système de transitions. On distingue deux types de systèmes de transition :

- Les systèmes de transition étiquetés ;
- Les systèmes de transition non étiquetés.

Ces systèmes sont souvent utilisés dans la vérification et la validation des propriétés de dispositifs physique réels.

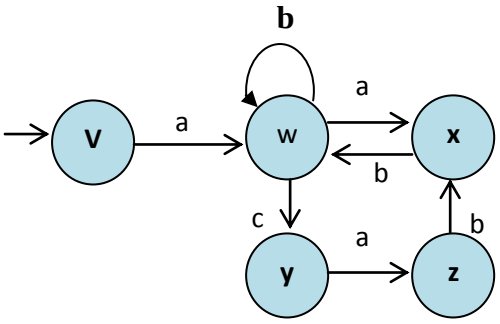
En effet, pour vérifier toutes les exécutions possibles du système étudié, on représente son modèle sous la forme d'un système de transitions à états, où chaque état correspond à un état du système à chaque pas d'exécution (chaque transition) le système évolue d'un état donné vers un autre, selon l'action exécutée par le système. Ce système de transition peut-être représenté par un graphe orienté ou automate, une machine de Turing ou bien un réseau de Pétri.

2.1 Système de transition étiqueté

Un système de transitions étiqueté sur transition est un quadruplet $\mathbf{S = < Q, \lambda, T, q_0 >}$ où:

- Q est un ensemble d'états,
- λ est un alphabet fini d'actions étiquetant les transitions,
- $T \subseteq Q \times \lambda \times Q$ est un ensemble de transitions reliant les états deux à deux,
- $Q_0 \in Q$ est un sous ensemble de Q non vide.

Exemple 2



Q = {v, w, x, y, z}

λ = {a, b, c}

T = {(v ,a, w),(w ,b ,w), (w ,a ,x),
(x ,b, w),(w ,c ,y) ,(y ,a ,z),(z ,b ,x)}

q_0 =v

Fig.6. Système de transition étiqueté.

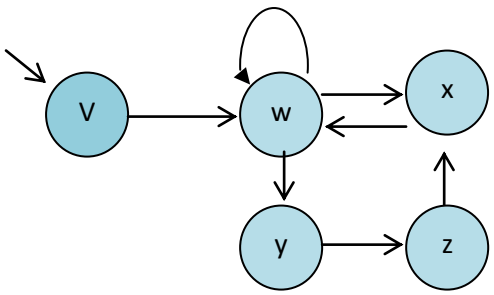
Une étiquette représente généralement l'entrée attendue pour passer d'un état à un autre, une condition qui doit être vraie ou le nom de l'action prise durant la transition.

2.2 Système de transition non étiqueté [10]

Un système de transitions est un triplet **S = < Q, T, q_0 >** où :

- Q est un ensemble d'états,
- T ⊆ Q x Q est un ensemble de transitions reliant les états deux à deux,
- Q_0 ϵ Q est un sous ensemble de Q non vide.

Exemple 3



Q = {v, w, x, y, z}

T = {(v, w), (w, w), (w, x), (x, w),
(W, y), (y, z), (z, x)}

q_0 =v

Fig.7. Système de transition non étiqueté.

3. Etat d'un système

Un état est parfois défini par une affectation des valeurs à chaque variable du système. L'ensemble des états est généralement défini comme un produit cartésien des états de différents sous-systèmes dans le cas d'un système complexe.

4. Séquence de transition

Soit **S = < Q, λ, T, q_0 >** un système de transitions étiqueté.

- Le franchissement d'une transition **< q, a, q'** > ϵ T est noté :

$$q \xrightarrow{a} q'$$

- Le franchissement d'une séquence de transitions **< q_i, a_i, q'_{i+1} >** ϵ T (avec $1 \leq i \leq n$) est noté :

$$q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots\ldots \xrightarrow{a_{n-2}} q_{n-1} \xrightarrow{a_{n-1}} q_n$$

- Soit **σ** = $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots\ldots \xrightarrow{a_{n-2}} q_{n-1} \xrightarrow{a_{n-1}} q_n$ une séquence de transitions. La séquence σ est dite le support du mot $a_1, a_2 \ldots, a_{n-1}$, appelé aussi **trace** .

Remarque

Soit Q un ensemble d'états.

- Q* : l'ensemble des séquences finies sur S.
- Q^{ω}: l'ensemble des séquences infinies sur S.
- σ_i : le i ème élément d'une séquence σ.

5. Propriétés des systèmes de transitions étiquetés

- Dans un système de transitions étiqueté les ensembles d'états Q et de transition sont finis.

- Un système de transitions étiqueté est **déterministe** si et seulement si

$$\forall\ < q, a, q' >, < q, b, q'' > \epsilon\ T, a = b \Rightarrow q' = q''.$$

- Un système de transitions étiqueté est **complet** si et seulement si

$$\forall\ q \epsilon\ Q, \forall\ a \epsilon\ \lambda, \exists\ q' \text{ tel que } < q, a, q' > \epsilon\ T.$$

6. Exécution d'un système de transitions [8]

Une exécution appelée aussi chemin d'exécution ou plus simplement chemin est une séquence de transitions.

Exemple 4

Soit l'exemple du digicode permettant l'ouverture d'une porte. Pour simplifier, nous supposons que 3 touches seulement sont possibles A, B et C et que la porte s'ouvre dès que l'on a tapé la suite ABB et cela même si on a commencé par se tromper.

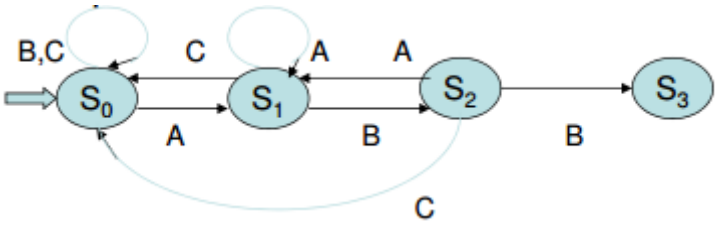


Fig.8. Modèle de digicode permettant d'ouvrir 1 fois une porte.

BAABABB, AABABB, CBACAABB sont des exécutions.

En général, les chemins sont soit des séquences d'états, soit des séquences de transitions. Par exemple S0, S1, S2, S3 est une séquence d'états qui ouvre la porte.

Notons que dans cet exemple, les exécutions qui ouvrent la porte sont finies alors que celles qui ne l'ouvrent pas sont infinies.

7. Equivalence d'un STE à un ST sans étiquette [9]

Un système de transitions étiqueté **S = < Q, λ, T, q_0 >** est équivalent au système sans étiquette **S' = < Q', T', q'_0 >** défini par :

- Q' = ({ ε} ∪ λ) × Q
- T' = {ε} × T
- q'_0 = { ((e, s), (e', s')) : (s, s') ∈ **q_0** & e, e' ∈ λ tq e' étiquette de l'arc (s, s')}

Exemple 5

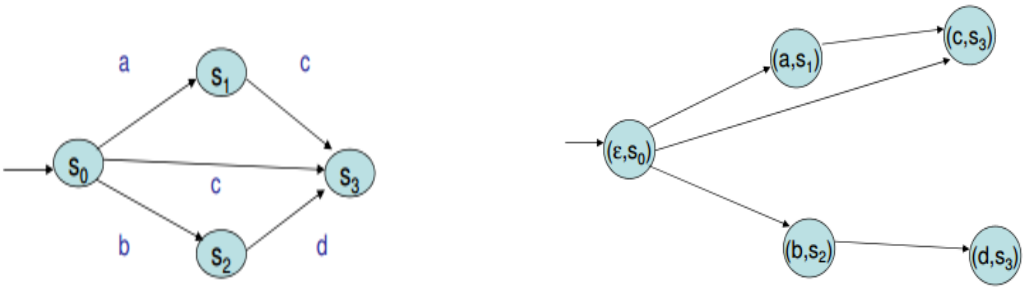


Fig.9. Equivalence d'un STE à un ST sans étiquette.

Les systèmes de transitions sont un formalisme permettant de décrire l'évolution d'un système informatique dans le cadre du modèle de traces. C'est une machine abstraite construisant les séquences d'états possibles au cours d'une exécution.

8. Propriétés liées à la dynamique des systèmes [27]

Les propriétés générales attendues concernant le comportement d'un système sont souvent regroupées en deux catégories :

- ✓ **Propriétés de sûreté** (safety) : le fonctionnement d'un système ne doit pas conduire à des situations catastrophiques ou dangereuses. ("safety = bad things do not happen on all the execution of a system"). Le non-blocage est un exemple de propriétés de sûreté.
- ✓ **Propriétés de vivacité** (liveness) : Propriété de vivacité (liveness property) Une propriété de vivacité exprime le fait que "quelque chose de bien arrivera finalement".

Il est parfois difficile, étant donné un système, de distinguer ce qui relève de la vivacité et ce qui relève de la sûreté. Des propriétés plus élémentaires (plus proches des automates) sont souvent vérifiées pour démontrer des propriétés plus globales. Il s'agit notamment des propriétés suivantes :

- ✓ **Accessibilité** : un état est accessible s'il existe un chemin l'atteignant depuis l'état initial.
- ✓ **Réinitialisabilité** : un automate est réinitialisable s'il existe un chemin depuis chaque état vers l'état initial.
- ✓ **Manque de blocage** : une situation de blocage (deadlock) indique qu'aucune transition n'est plus possible à un instant donné.
- ✓ **Manque de famine** : la famine est une situation où l'automate boucle sur une partie, ce qui empêche tout accès à d'autres parties de l'automate.

Dans ce qui suit nous allons étudier deux propriétés qui concernent le comportement d'un système de transition qui sont l'accessibilité et la vivacité.

9. Propriétés demandées par le modèle spécifié [26]

- Vivacité

La **vivacité** est le fait pour un système de garantir qu'une propriété sera vraie dans un système à partir d'une certaine étape de l'exécution.

Ce type de propriété va donc être évalué sur un ensemble d'états représentant un chemin d'exécution du système. Par exemple, "l'application se terminera finalement", "si un client envoie une requête, il recevra finalement une réponse" ou "chaque processus sera finalement en section critique" (exclusion mutuelle), etc. Dans ce cas, nous travaillons donc sur un temps infini contrairement à une propriété de sûreté. Pour formuler ce type de propriété, il est nécessaire d'utiliser un langage plus riche permettant d'exprimer le temps infini, comme la logique.

- Accessibilité

Un état est dit atteignable (ou accessible) s'il apparaît au moins une fois dans l'arbre d'exécution de l'automate. La notion d'atteignabilité est importante ; elle est souvent utilisée pour démonter des propriétés du système. En effet, le passage par un état (donc cet état est atteignable) peut correspondre à une propriété particulière du système ou bien ce passage est nécessaire pour que la propriété soit vérifiée. Par exemple, si dans un automate correspondant à la commande d'une porte, il y a un état Porte_Fermée, démonter la propriété « La porte peut être fermée » revient à démontrer que l'état Porte_fermée est atteignable.

10. Spécification du Système de transition

Les méthodes formelles peuvent être utilisées pour donner une spécification du système que l'on souhaite développer, au niveau de détails désiré. Une spécification formelle du système est basée sur un langage formel dont la sémantique est bien définie.

Cette description formelle du système peut être utilisée comme référence pendant le développement. De plus, elle peut être utilisée pour vérifier (formellement) que la réalisation finale du système (décrite dans un langage informatique dédié) respecte les attentes initiales (notamment en termes de propriétés).

10.1 Les approches de modélisation formelle [28]

Deux approches complémentaires de modélisation formelle :

1. Approche ascendante

- Programmation : programme + tests
- Spécification : propriétés du programme
- Validation : preuve de propriété

⇒ Coq "classique" : fonctions récursives → preuves inductives

2. Approche descendante

- Spécification : modèle formel d'un système
- Programmation : implantation de la spécification
- Validation : preuve que l'implantation respecte

⇒ Méthodes formelles : Z, VDM, B, Event B, ASM, etc.

Dans ce qui suit nous allons donc utiliser l'approche ascendante pour spécifier les propriétés de vivacité et d'accessibilité d'un système de transition dans l'environnement de Coq.

10.2 Automate

Un automate se définit par un type pour représenter des états, un type pour représenter des actions, un état initial et un ensemble de transitions, chacune étant définie par un état de départ, une action et un état d'arrivée. L'ensemble des transitions sera représenté par une fonction à valeurs booléennes.

Les enregistrements (records) de coq sont bien adaptés à cette représentation.

```
(* Automate *)

Record automate : Type  :=
   mk_auto {.
      etat :Set ;
      action : Set ;
      initial : etat;
     transition : etat -> action -> etat ->bool
}.
```

10.3 Traces et traces finies

Une trace est une suite d'action correspondant à une suite de transition dans un automate. Cette trace peut être finie ou infinie, et dans le premier cas mène à un état d'arrivée. Nous pouvant alors définir deux prédicats, l'un correspondant aux traces finies menant d'un état q à q', l'autre aux traces partiellement infinies issues de q :

➢ Trace finie

```
Inductive Trace_finie (A : automate) :
         (etat A) -> (LList (action A) ) -> (etat A ) -> Prop :=
   Vide_ftarce :
     forall (q : (etat A )),( Trace_finie A q LNil q)
| trans_ftrace :
  forall  (q q' q'':(etat A)) ( a : (action A)) (l : (LList (action A)))
   ,(transition A q a q') =true ->
   ( Trace_finie A q' l q'') ->
   (Trace_finie A q (LCons a l )q'').
```

➢ Trace

```
Inductive Trace (A : automate) :
          (etat A) -> (LList (action A)) -> Prop :=
   Vide_tarce :
     forall (q : (etat A )),( Trace A q (LNil) )
| trans_trace :
   forall (q q':(etat A)) ( a : (action A)) ( l : (LList (action A))),
   (transition A q a q')=true ->
   (Trace A q' l) ->
   (Trace A q (LCons  a  l )).
```

10.4 Propriétés d'accessibilité et de vivacité

L'intention des deux définitions suivantes est de définir un état accessible comme un état de l'automate atteignable par une suite finie de transitions à partir de l'état initial, et un état vivace comme un état duquel part au moins une suite infinie de transitions.

- **Accessibilité (Atteignable)**

```
Inductive Atteignable(A :automate) : (etat A) -> Set :=
      Atteignable_initial : (Atteignable A (initial A))
      |  Atteignable_trans : forall (q q':(etat A))(a : (action A)) ,
                                    (Atteignable A q) ->
                                    ( transition A q a q') =true ->
                                   (  Atteignable A q').
```

- **vivacité**

```
CoInductive Viv (A:automate):(etat A)-> Set :=
          Viv_trans :forall (q q':(etat A))(a:action A),
                     (Viv A (q'))->
                     ( transition A q a q') = true -> (Viv A (q)).
```

Conclusion

Dans ce chapitre nous avons donné quelques généralités sur les systèmes de transitions, ainsi que leurs spécifications formelles de manière générale.

Chapitre 4

Etude de Cas.

Chapitre 4

Etude de Cas

Introduction

Apres avoir fait une spécification d'un système de transition de manière générale dans le chapitre précédent, nous allons à présent vérifier les propriétés d'accessibilité et de vivacité sur un système physique qui est le digicode.

1. Modélisation du digicode

1.1 L'outil de modélisation

- Automata Editor :

Automata Editor est éditeur dédié au dessin des automates finis selon le format VauCanSon -G (Pachage de LaTeX) . L'éditeur prend en charge l'exportation vers plusieurs formats v par exemple des (EPS , SVG , VauCanSon -G , graphml , ...) ainsi que les formats bitmap par exemple (BMP , JPEG , ...) .

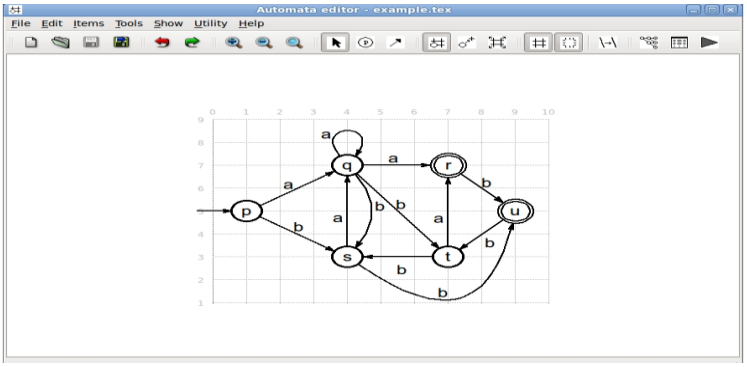
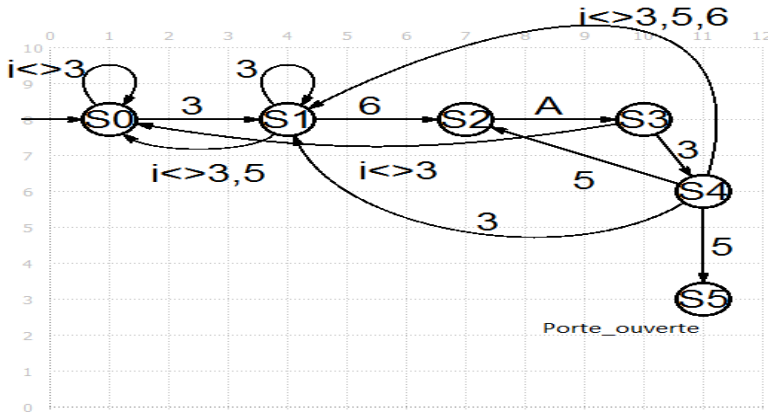


Fig.10. Interface graphique de l'outil Automata Editor.

1.2 Le digicode

Prenons l'exemple du digicode permettant l'ouverture d'une porte lorsqu'on introduit le code 36A35.



S= {S1, S2, S3, S4, S5}.

S_0= {S0}.

E= {1, 2, 3, 4, 5, 6, A, B}

Fig.11. Système de transition modélisant le digicode.

2. Spécification du digicode

2.1 L'outil de spécification

➢ L'assistant de preuve Coq :

Coq est un assistant de preuve utilisant le langage Gallina, développé par l'équipe PI.R2 d'Inria au sein du laboratoire PPS du CNRS. Il est fondé sur le calcul des constructions (CoC abrégé en anglais) introduit par Thierry Coquand. Dans la même veine, son langage est Gallina et Coq possède un wiki dédié, baptisé Cocorico.

Coq est fondé sur le calcul des constructions, une théorie des types d'ordre supérieur, et son langage de spécification est une forme de lambda-calcul typé. Le calcul des constructions utilisé dans Coq comprend directement les constructions inductives, d'où son nom de calcul des constructions inductives (CIC).

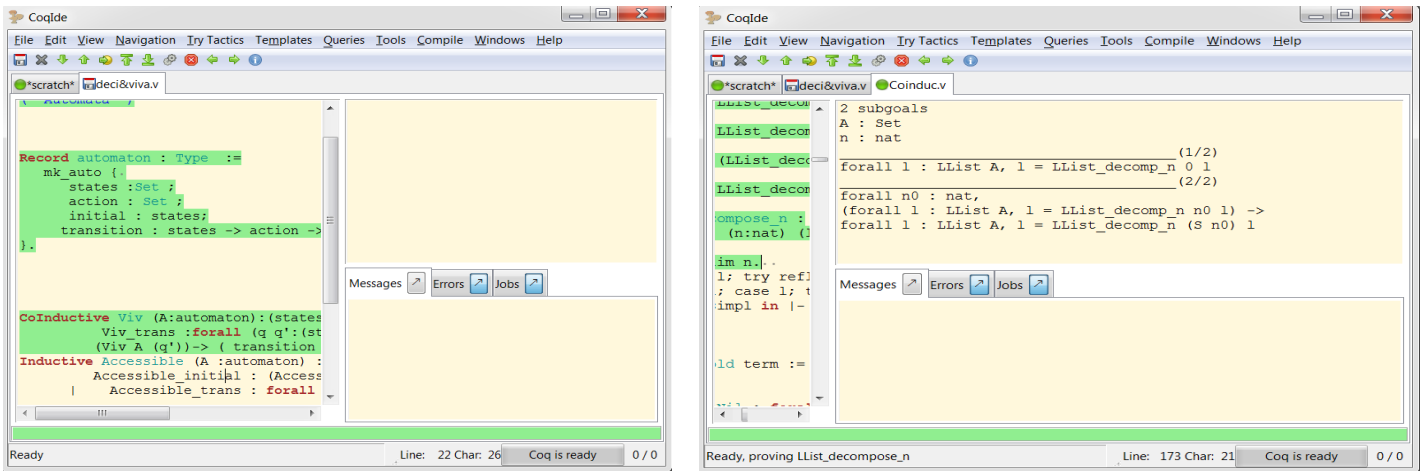


Fig.12. Interfaces graphiques de coq : Spécification (Gauche), Preuve (droite).

Plus particulièrement, Coq permet :

- ✓ De manipuler des assertions du calcul ;
- ✓ De vérifier mécaniquement des preuves de ces assertions ;
- ✓ D'aider à la recherche de preuves formelles ;
- ✓ De synthétiser des programmes certifiés à partir de preuves constructives de leurs spécifications.

2.2 Le programme de spécification

Dans ce qui suit nous allons spécifier le digicode dans l'environnement Coq.

```
Require Import List.
Set Implicit Arguments.

(* Declaration des listes  *)

CoInductive LList (A:Type) : Type :=
  | LNil : LList A
  | LCons : A -> LList A -> LList A.

Implicit Arguments LNil [A].

LList is defined

(* Automate *)

Record automate : Type  :=
   mk_auto {
      etat :Set ;
      action : Set ;
      initial : etat;
     transition : etat -> action ->list etat
}.

automate is defined
etat is defined
action is defined
initial is defined
transition is defined
```

```
(* Trace_fini *)

Inductive Trace_fini (A : automate) :
        (etat A) -> (LList (action A) ) -> (etat A ) -> Prop :=
   Trace_vide :
     forall (q : (etat A )),( Trace_fini A q LNil q)
| Trans_trace:
  forall  (q q' q":(etat A)) ( a : (action A)) (l : (LList (action A))) ,
  (transition A q a ) <> nil ->
   (Trace_fini A q' l q") ->
   (Trace_fini A q (LCons a l ) q").

Finite_trace is defined
Finite_trace_ind is defined

(* Trace *)

Inductive Trace (A : automate) :

   (etat A) -> (LList (action A)) -> Prop :=

   Trace_vide :

     forall (q : (etat A )),( Trace A q (LNil) )

| Trans_trace :

   forall (q q':(etat A)) ( a : (action A)) ( l : (LList (action A))),

   (transition A q a)<> nil->

   (Trace A q' l) ->

   (Trace A q (LCons  a  l )).

Trace is defined
Trace_ind is defined
```

```
(* Vivacite *)

Definition vivacite (A:automate) (q:etat A) :=

  forall a:action A, @transition A q a = nil.

Vivacite is define

(* Accessible*)

Inductive Accessible (A :automate) : (etat A) -> Set :=

     | Accessible_initial : (Accessible A (initial A))

     |    Accessible_trans : forall (q q':(etat A))(a : (action A)) ,

                                  ( Accessible A q) ->

                                  ( transition A q a )<> nil ->

                                   ( Accessible A q').

Accessible is defined
Accessible_rect is defined
Accessible_ind is defined
Accessible_rec is defined
```

2.3 Le programme de validation

```
Section Digicode.
(* etats *)
Inductive st : Set :=
  | S0 : st
  | S1 : st
  | S2 : st
  | S3 : st
  | S4 : st
  | S5 : st
  | S6:st.
```

```
(* actions *)
Inductive acts : Set:=
  |un:acts
  |deux :acts
  |trois : acts
  |quatre:acts
  |cinq :acts
  |six: acts
  |A:acts
  |B:acts.

(* transitions *)

Definition trans (q:st) (x:acts) : list st :=

  match q, x with

  | S0, trois => S1 :: nil
  | S1, six=> S2 ::nil
  | S1,trois=> S1::nil
  | S2, A => S3 :: nil
  | S3, trois=> S4 :: nil
  |S4,cinq=>S5::S2::nil
  |S4,trois=>S1::nil
  | _, _ => nil

  end.
```

```
(* automate *)
Definition A1 := mk_auto S0 trans.

(* Vavacite *)
Eval compute in Vivacite A1 S2.

  = forall a : acts,
      match a with
      | un => nil
      | deux => nil
      | trois => nil
      | quatre => nil
      | cinq => nil
      | six => nil
      | A => S3 :: nil
      | B => nil

Eval compute in Vivacite A1 S4.

  = forall a : acts,
       match a with
      | un => nil
      | deux => nil
      | trois => S1 :: nil
      | quatre => nil
      | cinq => S5 :: S2 :: nil
| six => nil
| A => nil
 | B => nil
```

```
(*Accessible*)

Eval compute in Accessible A1 S0 .

= Accessible

      {|

  etat := st;

  action := acts;

  initial := S0;

  transition := fun (q : st) (x : acts) =>

  match q with

   | S0 =>

      match x with

      | un => nil

      | deux => nil

      | trois => S1 :: nil

      | quatre => nil

      | cinq => nil

      | six => nil

      | A => nil

      | B => nil

      end


| S1 =>
```

```
            match x with
            | un => nil
            | deux => nil
            | trois => S1 :: nil
            | quatre => nil
            | cinq => nil
            | six => S2 :: nil
            | A => nil
            | B => nil
            end
        | S2 =>
            match x with
            | un => nil
            | deux => nil
            | trois => nil
            | quatre => nil
            | cinq => nil
            | six => nil
            | A => S3 :: nil
            | B => nil
            end

    | S3 =>
```

```
        match x with
        | un => nil
        | deux => nil
        | trois => S4 :: nil
        | quatre => nil
        | cinq => nil
        | six => nil
        | A => nil
        | B => nil
        end
    | S4 =>
        match x with
        | un => nil
        | deux => nil
        | trois => S1 :: nil
        | quatre => nil
        | cinq => S5 :: S2 :: nil
        | six => nil
        | A => nil
        | B => nil
        end

    | S5 => nil
```

```
| S6 => nil

 end |}
```

NB : En suivant la trace d'exécution marquée en rouge, on trouve que l'état S5 de l'ouverture de la porte est accessible à partir de l'état initial S0.

Conclusion

Dans ce dernier chapitre ,nous avons appliquer les notions de la théorie du lambda-calcul ainsi que le calcul des constructions inductives vues dans les deux premiers chapitres, Nous avons aussi appris à interagir avec l'assistant de preuve coq pour arriver enfin à spécifier un système de transition ,modélisant un système physique ainsi que ses propriétés d'accessibilité et de vivacité .En se basant sur l'approche formelle pour la spécification des systèmes matériels .

Conclusion générale

Le travail réalisé est inscrit autour de la richesse et la consistance de l'approche formelle. Il consiste à spécifier et valider un système de transition modélisant un système physique, ainsi que ses propriétés d'accessibilité et de vivacité dans l'assistant de preuve coq.

Dans Coq, programmer revient à écrire des spécifications et de démontrer des propriétés. Il s'agit d'une nouvelle méthodologie de la programmation basée sur les méthodes formelles, qui peut sembler déroutante au premier abord, mais elle offre un avantage important car elle garantit l'absence d'erreurs dans le programme obtenu.

Ce travail nous a permis de nous familiariser avec la théorie des assistants de preuves et de découvrir le principe de base du fonctionnement de ce dernier ,à travers l'étude qui a été faite dans les deux premiers chapitres qui se sont basés sur le théorie du lambda –calcul comme langage formelle et le calcul des constructions inductives qui est a la base du système d'assistance à la preuve Coq.

En fin, Nous pouvons dire que notre objectif à travers ce travail a été réalisé, en tenant compte d'une éventuelle amélioration du programme, afin d'obtenir une spécification plus optimal du point de vue affichage, une chose qui peut être inscrite dans les perspectives futurs de ce travail.

Plusieurs extensions sont envisageables pour ce travail, parmi les quelles on peut citer :

- Etendre ce travail par la prise en charge d'autres propriétés comme l'équité, la sureté, ..etc.
- Etudier d'autres modules de l'assistant de preuve coq, tel que l'extracteur de programme afin de pouvoir extraire un programme sans erreur à partir de la spécification.
- Effectuer le même travail sur d'autres assistants de preuves, et pouvoir ainsi faire une étude comparative sur les outils existants.

Bibliographie

[1] Dominique Méry. Modèles et algorithmes, sur le λ- calcul, 23 avril 2013,17p.

[2] Jean-François Dufourd. Spécification formelles, preuves et programmation. Notes de cours version 9 ; Université de Strasbourg, 18 août 2009, 241 p.

[3] S. Khemliche. Logique formelle .Notes de Cours, master 2 conduite de projet informatique .2015-2016, 71 p.

[4] Gabriel Scherer .Langages formels, calculabilité et complexité: Lambda calcul , université paris Diderot ,2008-2009 ,17p.

[5] Sylvain Baro. Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML. Thèse doctorat : informatique, Université Paris-Diderot - Paris VII, 10 juillet 2003, 234 p.

[6] Yves Bertot . Introduction au lambda-calcul pur, février 2006 ,12 p.

[7] Z. Mammeri .Système de transition, Automates a état finis .Université Paul sabatier, Toulouse III ,29 p.

[8] S. Khemliche .Validation Formelle, notes de Cours, master 2 SI .2015-2016, 50 p.

[9] Philippe Queinnec, Xavier Thirioux .Système de transition , Département Informatique et Mathématiques appliquées ENSEEIHT, 14 janvier 2016 ,45 p.

[10] Emmanuelle Encrenaz-Tiphène . Modélisation des Systèmes Répartis, Une modélisation par automates. Master d'Informatique de l'Université Paris VI, Spécialité SAR (20 p)

[11] Yves Bertot , Pierre Castéran . Le Coq Art. 18 février 2003, 447 p.

[12] Joël le Despeyroux. L'isomorphisme de Curry-Howard. INRIA Sophia-Antipolis Module Mécanisation des Preuves ; DEA d'informatique, Nice, janvier-mars 2000 ,10p.

[13] W.A. Howard the formulae-as-types notion of construction. In Hindley and seldin, editor, To H.B curry : Essays on Combinatory Logic, Lambda Calculus and Formalism,.Academic Press, London, 1980, pages 479-490.

[14] H.B Curry, R. Feys. Combinatory logic, North Holland publishing, 1958,417p.

[15] De bruijn. The mathematical language Automath,its usage and some of its extentions, Lecture notes in Mathematics, Spring-Verlag,1970,29-61p.

[16] Freek Wiedijk .Mizar: An Impression, 2002,42 p.

[17] M.J.Gordon,R Milner and C.P Wadsworth. Edinburgh LCF : a mechanised logic of computation. Lectures note in computer science, volume 78. Spring-Verlag, 1979.

[18] B.Werner. Une théorie des constructions inductives. Thèse de doctorat, Université Paris7, 1994 ,161p.

[19] The Coq Developement team , The Coq Proof Assistant Reference Manual Version 6.3.1. INRIA- Rocquencourt, December 1999,503p.

[20] Y. Bertot and P. Castéran. Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions. SpringerVerlag, 2004, 472p.

[21] Y. Bertot Coq in a Hurry Archive ouverte "cours en ligne", 2008. http: //cel.archives-ouvertes.fr/inria-00001173

[22] The Coq development team. The Coq proof Assistant Reference Manual, Ecole Polytechnique, INRIA, Université de Paris-Sud, 2004. http://coq. inria.fr/doc/main.html

[23] G. Huet, G. Kahn, C. Paulin-Mohring, The Coq proof Assistant, A Tutorial, Ecole Polytechnique, INRIA, Université de Paris-Sud, 2004. http://coq. inria.fr/V8.1/tutorial.html

[24] E. Giménez, P. Castéran, A Tutorial on Recursive Types in Coq, INRIA, Université de Bordeaux, 2006. http://www.labri.fr/Perso/~casteran/ RecTutorial.pdf.gz

[25] A. Miquel, Petit guide de survie en Coq, Université de Paris VII. http: //www.pps.jussieu.fr/~miquel/enseignement/mpri/guide.html

[26] Z.Mammeri. Système de transition .Support de cours, Université Paul Sébastian Toulouse. M2 Génie Logiciel, logiciels Répartis et Embarqués, 29p.

[27] Guthmulle Marion. Model checking et vérification de propriétés de vivacité dans SimGrid . Rapport de stage Master 2 Informatique parcours Recherche, Spécialité SSR (Sécurité, Services, Réseaux). Université Henri Poincaré, Nancy 1 Faculté des Sciences et Technologies, 2010-2011 ,39p.

[28] Frédéric Peschanski. Spécification et Validation de Programmes. Notes de cours, Master Informatique - M2 / STL .2014-2015 ,25p.

[29] C. Paulin-Mohring. Le système Coq. Thèse d'habilitation. ENS Lyon, January 1997.

---

[30] Ariane 5 Inquiry Board. ARIANE 5- Flight 501 Failure. Technical report, European Space Agency, 1996.

[31] Leveson, N., Turner, C.S. An Investigation of the Therac-25 Accidents. Reprinted from IEEE Computer, Vol.26, No.7,July 1993, pp.18-41. http://ei.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html.

Annexe

Introduction

COQ est un assistant de preuve, il permet la construction interactive des preuves formelles, et aussi la manipulation des programmes fonctionnels avec leurs spécifications. Il fonctionne sur de nombreuses architectures. Le présent document vise à présenter une spécification de base de l'environnement coq tels que les fonctions, les tactiques, et les commande dans le langage appelée Gallina, dans lequel des spécifications formels peuvent être développés et extraite sous forme de programme fonctionnels.

1. Démarrer le système

Pour utiliser Coq on dispose de trois commandes, une commande qui sert à compiler des fichiers (coqc), une commande qui fournit une boucle d'interaction sur un terminal textuel (coqtop), et une commande pour avoir une interface graphique (coqide). Nous conseillons d'utiliser coqide, mais les exemples traités dans ces notes de cours peuvent aussi être étudiés avec coqtop. Si vous utilisez coqide, vous voyez apparaitre une grande fenêtre divisée en trois parties.

La partie de gauche est prévue pour recevoir le texte que vous écrivez, les deux fenêtres de droite sont prévues pour afficher le résultat de vos commandes. Lorsque vous avez écrit une commande complète dans la fenêtre de gauche, vous pouvez en demander l'exécution en cliquant sur la flèche vers le bas qui apparait en haut de la fenêtre.

Les commandes exécutées changent de couleur pour indiquer qu'elles ont été exécutées, et il n'est plus possible de les modifier. Pour modifier une commande, il faut d'abord annuler son exécution, ce que l'on peut faire en cliquant sur la flèche vers le haut qui apparaît en haut. La commande revient alors dans la couleur de départ.

2. Apprendre le langage de programmation

Le système Coq contient un langage de programmation à la fois simple et compliqué. Dans cette première partie nous ne regarderons que la partie simple. Pour commencer, il vaut mieux utiliser des éléments du langage qui sont déjà définis. Pour cela on utilise une commande pour charger des modules de la librairie du système. Nous utiliserons la commande suivante:

```
Require Import ZArith List String Recdef.
```

Ainsi nous pourrons utiliser du calcul sur les nombres entiers relatifs (type Z, et sur les listes. Pour indiquer que “1 + 3” doit être compris comme l’addition de deux nombres dans Z, nous devons ensuite ajouter la commande suivante:

```
Open Scope Z_scope.
```

Nous pouvons maintenant définir des valeurs et des fonctions en utilisant le mot clef **Definition**:

```
Definition quatre := 4.

Definition polynome1 (x:Z) := x^2 + 3 * x + 1.
```

A tout moment on peut demander à Coq de vérifier si une expression est bien formée, à l’aide de la commande Check. Cette commande est donc bien utile pour faire un essai sans changer l’etat du système. Lorsque l’on vérifie le type d’une expression, on n’effectue pas le calcul décrit par cette expression.

```
Check polynome1 (polynome1 (polynome1 (polynome1 (polynome1 4)))).
```

On peut également demander à Coq d’exécuter une fonction ou de calculer une valeur:

```
Eval vm_compute in
  polynome1 (polynome1 (polynome1
      (polynome1 (polynome1 4)))).
```

Dans ce langage on peut aussi construire des fonctions sans leur donner de nom, et par exemple, mettre ces fonctions dans un couple:

```
Check (fun x => x * x, fun x => x ^ polynome1 x).
```

2.1 Les structures de données et le filtrage

Dans tout langage de programmation, on peut avoir besoin de réunir plusieurs données ensemble. Dans Coq, on peut utiliser les couples, mais il est également possible d'utiliser des listes: dans une liste tous les éléments doivent avoir le même type, mais la longueur peut être arbitraire. Il existe une notation infixe pour ajouter un élément devant une liste (::) et une constante pour représenter une liste vide: nil.

Le langage de Coq utilise une approche différente des langages de programmation de la famille de C, C++, ou Java pour observer le contenu de structures de données. Cette approche combine deux aspects: un aspect de test pour vérifier que les données ont la bonne forme, et un aspect d'accès aux structures. Cette approche est appelé filtrage. On écrit une expression de la forme suivante:

```
match p with
  (a, b) => ...
end
```

L'expression cachée dans les petits points peut faire référence à a pour parler de la première composante du couple p. Ainsi, la variable a est utilisée pour accéder à une sous-partie de la donnée p. C'est à peu prés la même chose pour les listes, sauf qu'il fait également dire ce que l'on fera si la liste que l'on observe est vide:

```
match p with
  a::v => ...
| nil =>  ...
end
```

On voit ici que la commande de filtrage effectue un test de la forme si-alors sinon, puisque la première zone de trois petits points correspond à ce que l'on fait si la liste est non vide et de la forme a::v et la deuxième zone de trois petits points correspond `a ce que l'on fait sinon. En d'autres termes, il s'agit donc d'un style de programmation o`u l'on décrit ce qui se passe dans chaque cas, et le langage nous oblige à prévoir tous les cas possibles. Par exemple, nous pouvons décrire une fonction qui reçoit une liste d'entiers,

retourne l'entier contenu dans cette liste s'il n'y en a qu'un, et retourne 0 dans les autres cas, de la façon suivante:

```
Definition list1_to_Z (l : list Z) : Z :=
  match l with
    a::nil => a
  | _ => 0
  end.
```

2.2 Polymorphisme et arguments implicites

Pour chaque type T dans Coq, il existe un type list T dont les éléments sont des listes qui contiennent des objets de type T. La valeur nil est particulière, car elle doit appartenir au type list T pour un certain T. En général, le système Coq est capable de deviner le type T grâce au contexte, mais dans certains cas, il n'y parvient pas et émet un message d'erreur sybillin.

```
Check nil.
Error: Cannot infer the implicit parameter A of nil
```

Dans ce cas, il souvent judicieux d'indiquer au système Coq que l'on veut parler de nil comme élément d'un certain type de listes, avec une notation de "forçage du type" (... : ...).

```
Check (nil: list Z).
```

Plusieurs fonctions sont fournies dans le système Coq pour manipuler des listes sans tenir compte du type des éléments. Par exemple, il existe une fonction de concaténation de listes, appelée **app** et notée ... ++ .... Elle ne peut être utilisée qu'en concaténant des listes de même type:

```
Check app (1::2::nil) (3::4::nil).
(1::2::nil) ++ 3 :: 4 :: nil : list Z
```

Les fonctions de cette forme sont appelées fonctions polymorphes. Lorsque l'on demande à Coq si une telle fonction est bien formée on voit apparaître une nouvelle notation, qui indique que **app** appartient à toute une famille de types.

```
Check app.
app : forall A : Type, list A -> list A -> list A
```

2.3 Définir son propre type de données

Nous pouvons également définir un nouveau type de données en décrivant chacun des cas possibles pour ce type de données. Par exemple, nous pouvons définir un type de données avec deux cas, le premier qui regroupe deux entiers et le deuxième qui ne contient qu'une chaîne de caractères:

```
Inductive ex_type : Type :=
  case1 (n1 n2 : Z)
| case2 (s : string).
```

Pour construire des éléments de ce type, on va pouvoir utiliser directement les fonctions case1 et case2, en leur donnant les arguments appropriés. Par exemple, voici deux définitions d'éléments du type ex type.

```
Definition vex1 : ex_type := case1 1 3.
Definition vex2 : ex_type := case2 "hello world!".
```

Les fonctions qui décrivent chacun des cas sont appelées des constructeurs. Dans le type ex type les constructeurs sont case1 et case2. Lorsque l'on écrit une fonction qui prend en argument un élément du type ex type pour faire un traitement par cas, on doit donc couvrir les cas donnés par tous les constructeurs, ce qui va s'écrire comme dans l'exemple suivant:

```
Definition ex_type_to_Z (v : ex_type) :=
  match v with
    case1 n1 n2 => n1 + n2
  | case2 s => 0
  end.
```

2.4 La récursion

Dans un type de données, il est autorisé que certains champs de l'un des constructeurs soient dans le type même que l'on est en train de définir. Dans ce cas, une valeur du type pour contenir une autre valeur du même type, et ce répétitivement, de telle sorte qu'une telle donnée peut être potentiellement très grosse. Ceci est illustré dans l'exemple suivant:

```
Inductive String_Z_tree : Type :=
  bin_Z (n1 : Z) (t1 t2 : String_Z_tree)
| one_string (s : string) (t : String_Z_tree)
| sz3.
```

L'expression suivante appartient à ce type:

```
Check bin_Z 0 (bin_Z 1 sz3 (one_string "hello" sz3))
      (bin_Z 2 (bin_Z 3 sz3 sz3) (bin_Z 4 sz3 sz3)).
```

Lorsque l'on définit une fonction sur un tel type de données, il est autorisé d'inclure des appels récursifs, mais seulement sur les éléments du type qui sont des champs de constructeurs observés par un traitement par cas sur la valeur initiale. Une telle fonction récursive doit nécessairement être définie à l'aide du mot-clef Fixpoint.

Par exemple, la fonction suivante est acceptée, cette fonction additionne toutes les valeurs entières qui apparaissent dans un terme de type String Z tree.

```
Fixpoint sum_sz_tree (t : String_Z_tree) : Z :=
  match t with
    bin_Z n t1 t2 => n + sum_sz_tree t1 + sum_sz_tree t2
  | one_string s t1 => sum_sz_tree t1
  | sz3 => 0
  end.
```

Ainsi, le système Coq autorise les fonctions récursives, mais seulement avec une discipline qui assure qu'aucune fonction récursive ne boucle indéfiniment. Lorsque l'on définit une fonction récursive à plusieurs arguments, il est parfois nécessaire d'indiquer quel est l'argument pour lequel la discipline de récursion doit être respectée, ceci se fait avec le mot-clef struct.

Par exemple, la fonction suivante remplace tous les sous-termes de la forme sz3 par un nouveau terme, cette fonction prend deux arguments, mais c'est le premier qui décroît à chaque appel récursif.

```
Fixpoint subst_sz3 (t t' : String_Z_tree) {struct t}
   : String_Z_tree :=
  match t with
    bin_Z n t1 t2 => bin_Z n (subst_sz3 t1 t')
                       (subst_sz3 t2 t')
  | one_string s t1 => one_string s (subst_sz3 t1 t')
  | sz3 => t'
  end.
```

Les listes fournies par Coq sont justement un exemple de type de données où l'un des constructeurs contient un champ du même type. La définition des listes peut être observée à l'aide de la commande suivante:

```
Print list.
```

```
Inductive list (A : Type) : Type :=
   nil : list A  | cons : A -> list A -> list A
For nil: Argument A is implicit
For cons: Argument A is implicit
For list: Argument scope is [type_scope]
For nil: Argument scope is [type_scope]
For cons: Argument scopes are [type_scope _ _]
```

Ce dialogue nous apprend que list est un type de données qui dépend d'un autre type, nommé A dans cette définition. Les deux constructeurs cons et nil prennent tous les deux un type en premier argument, mais celui-ci est implicite. Par ailleurs, la notation a::l' représente en fait l'expression cons a l' .

Du point de vue de la programmation récursive, le constructeur cons contient un champ qui est dans le type list: les appels récursifs sont autorisés sur les champs de cette forme. Voici par exemple une fonction qui prend en entrée une liste contenant les entiers a1, a2, . . . , a_n et retourne la liste a1 + a2, a3 + a4, . . . a2k + a2k+1.

```
Fixpoint sum_pairs (l:list Z) : list Z :=
  match l with
    a::b::tl => (a+b)::sum_pairs tl
  | _ => l
  end.
```

Il faut noter que dans cet exemple, l'argument de l'appel récursif **tl** est bien un sous terme de l'argument initial l, même si ce n'est pas un sous-terme direct. Il existe une façon de décrire les nombres entiers positifs ou nuls qui est particulièrement bien adaptée pour cette forme de récursion. C'est ce que l'on appelle les entiers de peano.

Il s'agit de dire que tout entier positif ou nul est soit 0 soit le successeur d'un autre entier positif ou nul. Ceci s'exprime avec le type de données **nat** qui s'écrit de la façon suivante:

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

Dans ce type de données, le nombre 3 peut s'écrire S (S (S O)). Comme nous avons écrit Open Scope Z scope au début, si l'on écrit 3, c'est un nombre de type Z qui est construit, mais si nous tapons 3%nat ou (3 + 4)%nat, ce sont des expressions de type **nat** qui sont construites. Avec ce type de données, les nombres ne sont pas représentés efficacement mais la programmation récursive est assez facile : pour définir une fonction récursive il suffit de savoir exprimer le résultat pour un nombre de la forme n+1 à partir du résultat de la même fonction pour le prédécesseur de ce nombre n. Par exemple, on peut décrire la fonction factorielle de la façon suivante:

```
Fixpoint fact (n:nat) : nat :=
  match n with O => O%nat | S p => (n * fact p)%nat end.
```

On peut également définir la fonction de Fibonacci de la façon suivante, où l'on calcule la valeur de la fonction pour n et n + 1 à la fois.

```
Fixpoint fibo (n:nat) : nat*nat :=
  match n with
    0 => (0,1)%nat
  | S p =>
    match fibo p with
      (vp, vsp) => (vsp, vp + vsp)%nat
    end
  end.
```

Malheureusement, la fonction **fibo** ainsi définie est très inefficace et ne permet pratiquement pas de calculer la valeur pour des entrées moyennes, comme 1000.

3. Preuves et démonstrations

3.1 Faire ses propres théorèmes

Pour démontrer de nouveaux théorèmes dans Coq, le système fournit une collection de commandes. La première est la commande **Lemma1**, qui permet de démarrer une preuve en donnant le nom du théorème et son énoncé. L'énoncé est écrit avec des connecteurs logiques :

- **forall** (quantification universelle, "quel que soit"),

- **->** (implication, "si . . . alors . . . "),

- **exists** (quantification existentielle, "il existe"),

- /\ (conjonction, "et"), \/ (disjonction, "ou"), ~ (négation, "non").

Les formules atomiques sont dans le type Prop et les connecteurs logiques prennent des formules dans le type Prop et construisent de nouvelles formules dans le type Prop. Parmi les notions connues, on dispose de la comparaison des nombres, l'égalité, etc. Par exemple, on peut définir la notion "être un nombre premier" de la façon suivante:

```
Definition prime (n:Z) : Prop :=
  1 < n /\ forall x, 1 < x < n -> ~exists k, n = x * k.
```

3.2 Preuve dirigée par les buts

Une fois que l'énoncé est accepté par Coq, il faut appliquer des commandes qui vont réduire cet énoncé en des énoncés plus simples jusqu'à qu'ils soient solubles trivialement ou solubles par un outil automatique. Nous aurons besoin de connaître deux outils automatiques:

(1) omega permet de prouver les comparaisons linéaires qui sont des conséquences des comparaisons linéaires déjà dans le contexte.

On appelle comparaison linéaire une formule de comparaison dont le membre de droite et le membre de gauches sont des sommes de produits entre une constante numérique et une variable;

(2) ring permet de prouver les égalités entre deux formules qui sont égales modulo l'associativité et la commutativité de l'addition et la multiplication et la distributivité. Un énoncé est soluble trivialement si la formule à prouver est déjà dans les hypothèses ou si c'est une égalité dont les deux membres sont égaux.

Pour simplifier un but, nous pouvons appliquer plusieurs commandes, adaptées pour chaque connecteur logique. Par exemple, si le but commence par une quantification universelle (forall) ou une implication →, il est souvent judicieux de commencer par une commande intros. Si une hypothèse est une conjonction, il est souvent judicieux de décomposer cette hypothèse pour récupérer séparément les deux sous-formules qu'elle contient. Si la formule à prouver est une conjonction, il est souvent judicieux de casser cette formule pour prouver séparément les deux sous formules qu'elle contient. Ces différentes commandes, appelées tactiques, sont récapitulées dans le tableau suivant:

	⇒	∀	∧	∨	∃
Hypothesis	`apply`	`apply`	`elim`	`elim`	`elim`
goal	`intros`	`intros`	`split`	`left` or `right`	`exists` v
	¬	=			
Hypothesis	`elim`	`rewrite` `injection` `discriminate`			
goal	`intro` `discriminate`	`reflexivity`			

La tactique **discriminate** n'est utilisable pour une négation que lorsque la formule niée est une égalité. La tactique **injection** est utilisable pour une hypothèse lorsque l'égalité dans cette hypothèse est une égalité entre deux formules qui commencent par le même constructeur d'un type inductif. La tactique **discriminate** sur une hypothèse est une égalité entre deux formules qui commence par des constructeurs différents. Il faut parfois faire attention à quelle commande est utilisée, par exemple, lorsque l'on doit prouver une disjonction, appliquer **left** ou **right** est un choix important, car cela permet de choisir celle des deux sous formules qui est prouvable. Pour faire avancer une preuve, il peut également être judicieux de faire apparaître une formule logique intermédiaire, que l'on prouve séparément, et que l'on peut ensuite réutiliser pour faire avancer la preuve principale.

Pour effectuer cette étape de raisonnement, il suffit d'utiliser la commande assert. Comme dans l'exemple suivant:

```
assert (nle10 : n <= 10).
```

Le système Coq ajoute la formule n <= 10 parmi les formules à prouver. Lorsque cette formule est prouvée, l'ancien but réapparaît avec l'hypothèse nle10 est ajoutée dans son contexte.

3.3 Démonstrations par récurrence

Lorsqu'un énoncé logique porte sur une variable d'un type de données inductif, il est possible de faire une preuve par récurrence suivant la structure de ce type de données.

La preuve demande alors de vérifier que la formule à prouver est bien satisfaite dans chacun des cas décrits par les constructeurs, en ajoutant une hypothèse de récurrence pour chacun des champs des constructeurs qui sont déjà dans le type inductif. Par exemple, pour les nombres naturels, nous pouvons faire une preuve par récurrence, nous allons montrer que pour n et m qui sont des nombres naturels, n est inférieur ou égal à n + m. Ce théorème est déjà connu de Coq, mais nous allons le redémontrer en utilisant seulement les deux propriétés suivantes:

```
le_n : forall n, n <= n
le_S : forall n m, n <= m -> n <= S m
```

Puisque nous voulons faire une preuve sur les nombres naturels, nous allons commencer par remettre en place l'interprétation des notations arithmétiques pour qu'elle concerne des expressions naturelles en priorité.

```
Close Scope Z_scope.
Lemma plus_le : forall n m, n <= m + n.
intros n m; induction m.
2 subgoals

  n : nat
  ============================
   n <= 0 + n

subgoal 2 is:
 n <= S m + n
```

L'étape de preuve engendre donc deux buts correspondant aux deux constructeurs du type nat. Nous ne voyons pas les hypothèses du second but pour l'instant. Concentrons-nous sur le premier. La définition de l'addition dans Coq fait que 0 + n est la même chose que n. Nous pouvons demander à coq de changer ceci avec la tactique suivante:

```
change (0 + n) with n.
2 subgoals

  n : nat
  ============================
   n <= n
```

Nous pouvons maintenant appliquer le théorème **le n**. La tactique pour cela est apply.

```
apply le_n.
1 subgoal


  m : nat
  IHm : n <= m + n
  ============================
   n <= S m + n
```

Nous voyons maintenant apparaître les hypothèses du second but. L'hypothèse IHm est une hypothèse de récurrence, qui indique que la propriété que nous voulons prouver est déjà satisfaite pour m.

La conclusion du but indique que nous voulons prouver cette propriété pour S m. Ici encore, la définition de l'addition pour les entiers naturels est telle que S m + n et S (m + n) sont considérés comme la même expression, nous pouvons demander `a Coq de faire apparaître cette similitude avec la tactique suivante:

```
change (S m + n) with (S (m + n)).
1 subgoal

  n : nat
  m : nat
  IHm : n <= m + n
  ============================
   n <= S (m + n)
```

Nous pouvons maintenant appliquer le théorème le S, ce qui requiert ensuite que nous prouvions l'hypothèse de ce théorème. C'est facile car elle apparaît dans les hypothèses.

```
apply le_S.
1 subgoal

  n : nat
  m : nat
  IHm : n <= m + n
  ============================
   n <= m + n

trivial.
Proof completed.
Qed.
```

Quand la preuve est complète, il faut la terminer avec la commande Qed.

3.5 raisonner sur le calcul des fonctions

Les fonctions que nous avons définies en Coq peuvent apparaître dans les buts. Parfois, nous aurons besoin d'exprimer que le calcul d'une certaine fonction retourne une certaine valeur, simplement parce que c'est la définition de cette fonction. La façon la plus systématique de faire ce genre de calcul est d'appeler 13 la tactique simpl.

Cette tactique parcourt toute la conclusion du but pour trouver des instances de fonctions récursives et faire progresser le calcul des ces fonctions récursives en respectant la définition de ces fonctions. Parfois, la fonction simpl ne fait pas ce que l'on veut.

Dans ce cas, il est possible d'être plus précis en utilisant la tactique change et en donnant précisément l'expression que l'on veut remplacer et l'expression que l'on veut mettre à la place. Il faut faire bien attention que le remplacement de l'une par l'autre doit bien être une conséquence directe de la définition des fonctions qui apparaissent dans les deux formules. Un exemple d'utilisation de la tactique change est déjà apparu dans la section 3.4.

4. Définir de nouveaux prédicats

On peut définir de nouvelles propriétés en indiquant par quelles règles ces propriétés peuvent être prouvées et en précisant que toute preuve d'une de ces propriétés doit ultimement reposer uniquement sur ces règles. Par exemple, on peut décrire l'ordre "inférieur ou égal" sur les entiers naturels en indiquant que tout nombre est inférieur à lui-même et que si un nombre n est déjà inférieur à m, alors il est également inférieur S m. Cette définition est exprimée de la façon suivante :

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

Les règles de raisonnement fournies dans la définition inductive sont encore appelées des constructeurs: ces constructeurs peuvent être utilisés comme des théorèmes. Ici les deux théorèmes fournis dans cette définition ont les énoncés suivants:

```
le_n : forall n:nat, le n n
le_S : forall n m:nat, le n m -> le n (S m)
```

Lorsque l'on a défini une propriété de cette manière, il est également possible de faire des preuves par récurrence sur la propriété: le système Coq engendre alors autant de buts que le nombre de constructeurs pour la définition inductive. Pour ceux des constructeurs qui ont en hypothèse une instance de la propriété définie dans cette définition inductive, le système Coq engendre aussi une hypothèse de récurrence. Par exemple, nous pouvons prouver la réciproque du théorème étudié dans la section 3.4

```
Lemma le_ex_plus : forall n m, le n m -> exists p, m = n + p.
Proof.
intros n m Hle.

 induction Hle.
 exists 0.
 rewrite plus_0_r.
 trivial.
```

La tactique induction Hle avait engendré 2 buts et les trois tactiques qui la suivent ont résolu le premier but. Le deuxième but a la forme suivante:

1 subgoal

n : nat
m : nat
Hle : n <= m
IHHle : exists p : nat, m = n + p
============================
exists p : nat, S m = n + p

L'hypothèse IHHle est une hypothèse de récurrence qui vient du fait que l'on suppose que la propriété recherchée est déjà satisfaite pour les nombres n et m, qui apparaissent dans l'hypothèse Hle. Nous pouvons utiliser cette hypothèse en suivant l'indication de la table donnée en section 3.2.

```
destruct IHHle as [x meqnplusx].
exists (S x).
rewrite meqnplusx.
rewrite plus_n_Sm.
trivial.
Qed.
```

NB :

Vous pourrez trouver plus d'information dans le livre [21] et le manuel de référence [23]. Il y a un tutoriel dans [25]. Il existe également quelques tutoriaux en anglais sur le web [24, 22].