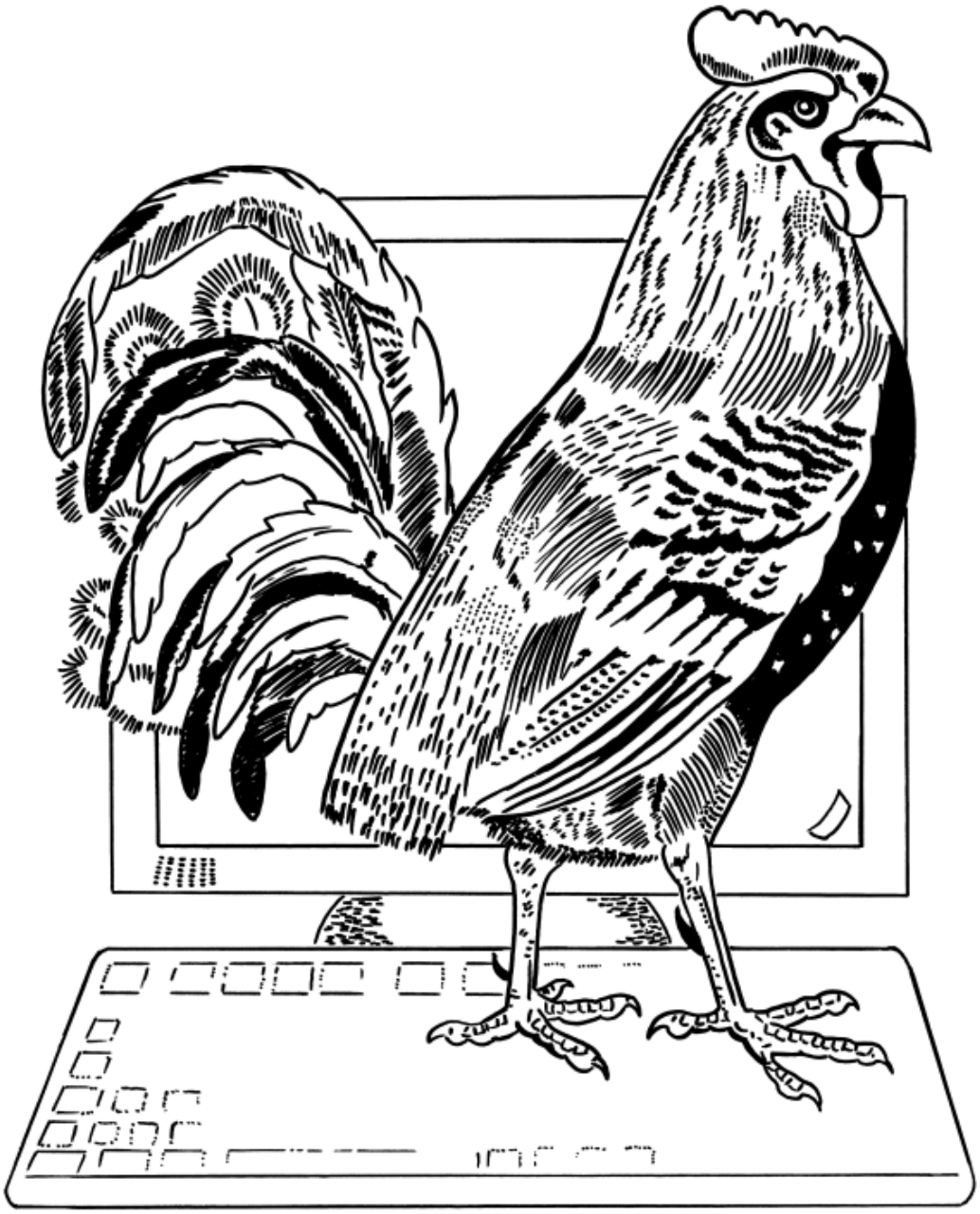


Formalisation de concepts mathématiques dans le système Coq

Mémoire de Master Recherche

par Vincent Demange
encadré par Loïc Colson

soutenu au LITA (EA 3097)
le 2 juillet 2008

paul verlaine université - metz
ufr mathématique informatique mécanique et automatique

École doctorale IAEM Lorraine
Master 2 recherche – modélisation par le logiciel

Résumé

Le but du stage est de se familiariser avec le système Coq de preuves assistées par ordinateur, développé par l'Inria, l'Université d'Orsay, l'école Polytechnique et le cnrs. La formalisation devant être conçue pour être lisible et compréhensible par une personne non experte du système Coq, en particulier par un mathématicien non informaticien.

Comme exemple de preuves, on démontre le théorème du plus petit et du plus grand point fixe de Knaster pour une fonction croissante d'un ensemble de parties dans lui-même. Puis, comme Tarski, on généralise à un treillis complet quelconque. Ensuite on généralise à une famille quelconque de fonctions monotones. On montre que l'ensemble des points fixes d'une fonction croissante est un treillis complet.

Enfin on s'intéresse à la formalisation d'un théorème de Damian Niwiński sur les points fixes diagonaux de fonctions à deux variables, puis sa réciproque par Loïc Colson.

Remerciements

> « Une cause très petite, qui nous échappe, détermine un effet considérable que nous ne pouvons pas ne pas voir, et alors nous disons que cet effet est dû au hasard. »
> Henri Poincaré

Cette remarque de M. Poincaré est tout à fait transposable à la difficulté d'accorder sa reconnaissance aux personnes impliquées, directement ou indirectement, à l'aboutissement d'un travail quelque soit sa nature. Je tiens donc à m'excuser de ne pouvoir citer tous ceux qui le méritent, et remercie par avance ceux qui se reconnaîtront.

Je tiens tout d'abord à remercier mon encadrant, Loïc Colson, qui par ses réponses pertinentes, ses attentes exigeantes et motivées, son suivi d'excellente qualité, ses réflexions d'ordre général et philosophique, m'ont toujours encouragé à me dépasser et à voir plus, savoir plus et comprendre mieux.

Mes remerciements aux collègues de pauses du bureau d'à côté, messieurs Dieter Kratsch, Mathieu Liedloff et mon camarade d'infortune Mathieu Chapelle, et aux discussions sérieuses et amusées que nous avons partagé.

Je n'oublie pas de remercier les doctorants Thomas Pietrzack, Thomas Veneziano et Julien Schleich pour m'avoir accueilli à leur table et dans leur bureau (avec canapé), ainsi que pour m'avoir fait profiter des *potins* de l'Université.

Je tiens à témoigner ma reconnaissance à tous les enseignants de cette année, pour m'avoir transmis leur connaissance, et pour leur sympathie.

Je remercie particulièrement l'artiste Magali Kaczmarski, pour avoir autant embelli la couverture de ce rapport, et pour avoir donné de son temps à la réalisation du dessin de ce magnifique et fier Coq.

Un énorme remerciement au logiciel Coq et à ses contributeurs, sans qui ce mémoire ne saurait exister.

Et je termine par une pensée à toute ma famille pour m'avoir offert *gîte et couvert* (entre autres), ainsi qu'à mes amis à qui je souhaite réussite pour leurs projets personnels.

Table des matières

Chapitre 1 Motivations

Avec la complexification des preuves mathématiques, la question de la confiance qu'on leur accorde prend une place majeure. Car l'apparente rigueur des mathématiques repose en fait sur des processus mentaux. Les systèmes formels aident à réduire l'acceptation de la validité des démonstrations à l'acceptation d'une succession d'applications de principes suffisamment « intuitifs » pour être admis en l'état. Plus ces principes sont fins, plus la taille des preuves augmente, et plus l'aspect opérationnel de ces systèmes peut être mis en doute.

Mais les récents progrès en informatique permettent d'implémenter ces systèmes formels, de les utiliser et surtout d'automatiser la vérification des preuves qu'ils produisent. C'est donc au niveau de la vérification qu'il suffit d'accorder sa confiance pour s'assurer de la validité d'une preuve. Cette validation par la machine est d'autant plus nécessaire que les mathématiques se sont elles-mêmes complexifiées au point d'obliger les mathématiciens à s'occuper de domaines de plus en plus spécialisés. Il n'est donc plus toujours possible de reconnaître une preuve sur l'accord de ses pairs.

L'utilisabilité opérationnelle de tels systèmes est assurée par l'application d'ensembles de règles, ce qui permet à l'utilisateur une productivité raisonnable. Des procédures de (semi-)décision pour des problèmes bien définis permettent d'éviter la frustration d'avoir à développer les étapes « triviales » d'une démonstration. Cependant, à ce niveau de granularité, les démonstrations restent très détaillées et parfois répétitives.

Pour espérer que les mathématiciens reconnaissent et utilisent ces outils, plusieurs conditions doivent être respectées (cf. [2]). En particulier, le *style* mathématique doit être conservé, autant dans la présentation des théorèmes que dans le développement des preuves. Ce sont ces deux points qui ont particulièrement retenu notre attention et motivé ce présent travail.

Nous avons donc choisi de nous placer dans la position toute particulière de développer des preuves mathématiques avec un outil d'aide à la démonstration. Il s'agit en fait d'un mélange de compétences : (a) il faut comprendre les théorèmes mathématiques qu'on cherche à démontrer, (b) il faut savoir organiser ses démonstrations pour qu'elles soient réutilisables et (c) il faut comprendre les limitations du logiciel qu'on utilise pour la formalisation ainsi que les mécanismes qui le gouvernent. En d'autres termes, il est nécessaire (a) d'être mathématicien, (b) de savoir correctement programmer et (c) d'avoir des connaissances et des compétences générales de logique.

1.1 Choix des théorèmes

Les mathématiciens travaillent dans des structures algébriques bien définies. Ils y établissent des propriétés et les démontrent. Nous avons choisi de travailler dans les treillis complets, une structure algébrique à la fois simple et puissante (cf. [4]), et de commencer

par y établir le fameux théorème de point fixe de Knaster-Tarski [15, 20] (dont la formalisation est bien connue) et ses développements (qui demandent plus de travail pour être formalisés).

Les treillis sont bien connus en informatique dans divers domaines : algorithmique, intelligence artificielle (représentation de concepts), logique (fondements de la théorie des ensembles), etc. On peut aussi observer l'utilisation des treillis en sciences sociales (hiérarchisation, liens sociaux). De plus, en informatique, on rencontre fréquemment des théorèmes de point fixe ; combinateurs de point fixe du λ-calcul pur, sémantique dénotationnelle, théorie des jeux, calculabilité, fondements des mathématiques, etc.

Ce choix est donc motivé par le domaine d'étude, l'informatique, et les liens qui l'unissent à ces sujets.

1.2 Choix du formalisme

En λ-calcul typé, programmes et démonstrations se confondent (voir sec. 2.1, p. 3). En sus, un bon aperçu de ces formalismes nous a été donné tout au long du cursus universitaire. Le système Coq est donc un bon candidat. Il est développé et maintenu à l'Inria par l'équipe *TypiCal* (anciennement *LogiCal*).

Coq s'appuie sur le calcul des constructions inductives, qui est un λ-calcul typé d'ordre supérieur suffisamment expressif pour formaliser les théorèmes choisis, ainsi qu'une grande partie des mathématiques connues. Les concepts de treillis sont intimement liés à la théorie des ensembles, tandis que Coq l'est à la théorie des types, un vrai travail de formalisation est donc nécessaire.

Coq est un assistant à la preuve. C'est-à-dire qu'il assiste l'humain dans sa démarche de recherche de preuves, il ne s'agit pas d'un démonstrateur automatique. Le développement commence du but, qu'il faut réduire en utilisant des tactiques, qui génèrent de nouveaux buts intermédiaires, jusqu'à l'obtention de trivialités. En cela, il diffère des démonstrations usuelles des mathématiciens : le contenu d'une démonstration est une succession de noms de tactiques, et le raisonnement est « ascendant » (du but aux hypothèses).

C'est sur ce dernier aspect qu'il a fallu nuancer le respect du style mathématique. En fait, on se contentera de permettre un suivi des preuves aussi agréable et compréhensible que possible — tactiques et lemmes seront imbriqués intelligemment.

Chapitre 2 Calcul des constructions et Coq

Ce chapitre est une rapide présentation des prérequis, suffisants pour une compréhension globale et générale. On va tout d'abord faire un rappel sur le principe de l'isomorphisme de Curry-Howard. Puis voir comment il peut être étendu pour augmenter son pouvoir expressif. Enfin on présentera un aperçu de la façon dont ces idées sont implémentées dans Coq.

2.1 Principes de l'isomorphisme de Curry-Howard

En s'appuyant sur la logique minimale basée sur l'implication, nous rappelons en quoi consiste cet isomorphisme entre types et formules. Par la même occasion, nous introduisons le λ-calcul simplement typé, ce qui fournit aussi un aperçu de la théorie des types.

2.1.1 Déduction naturelle minimale sur l'implication

Pour rappel, les règles d'inférences de la déduction naturelle avec le seul connecteur de l'implication $\rightarrow$ sont les suivantes :

$$\frac{A \in \Gamma}{\Gamma \vdash A}\, hyp \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow_i \qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow_e$$

Ces règles sont des schémas syntaxiques qui permettent d'établir des preuves en se basant sur les intuitions du raisonnement mathématique. Informellement et pour exemple, il faut comprendre la règle $\rightarrow_e$ comme : si d'un ensemble d'hypothèses Γ je peux déduire $A \rightarrow B$ (si A alors B), et si du même ensemble d'hypothèses Γ je peux déduire A, alors sous ces hypothèses je peux déduire B.

Un séquent[1] $\Gamma \vdash F$ est un couple, constitué d'un ensemble de formules Γ, les hypothèses, et d'une formule conclusion F. Les règles se présentent en deux parties situées de part et d'autre du trait. Les séquents prémisses de la règle se situent au-dessus, le séquent conclusion au-dessous.

Démontrer une formule F sous un ensemble d'hypothèses Γ, c'est utiliser les règles syntaxiques précédentes pour construire le séquent $\Gamma \vdash F$. Ces démonstrations peuvent être représentées de façon arborescente (cf. ex. 2.1.2, p. 4).

[1] Ici l'utilisation de séquents a une justification pratique puisqu'elle permet d'exprimer simplement le « déchargement » d'hypothèses (règle $\rightarrow_i$). Mais à l'inverse du calcul des séquents, Γ n'est pas une liste de propositions (ou multi-ensemble), mais bien un ensemble de propositions. De plus les règles sont bien celles de la déduction naturelle. Il ne s'agit donc pas du calcul des séquents.

2.1.2 λ-calcul simplement typé minimal

Habituellement, on présente les règles de typage des termes du λ-calcul typé par des règles d'inférences similaires à celles de la déduction naturelle. Il s'agit de règles de construction des termes typés du calcul : à chaque terme est associé un unique type.

Pour cela, introduisons d'abord les règles de construction des types utilisés. Nous avons deux constructions de types possibles :

– les types atomiques, habituellement notés $\alpha, \beta, \gamma, \ldots$;

– les types « flèche », de la forme $(A \to B)$ où A et B sont des types.

Pour simplifier, on dit que $\to$ est associative à droite, c'est-à-dire qu'on écrit $A \to B \to C$ pour représenter le type $(A \to (B \to C))$.

À présent, nous pouvons donner les règles de construction des termes du calcul. Dans ces règles, l'environnement Γ est en fait une succession de correspondances terme–type, qu'on note $t : T$ ou t^T avec t un terme et T un type.

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\, var \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \to B}\, \lambda \qquad \frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash x : A}{\Gamma \vdash (f\ x) : B}\, app$$

De la même façon, on simplifie l'écriture des termes en établissant une règle d'associativité : l'application (app) est associative à gauche, c'est-à-dire que le terme $f\ x\ y$ représente en fait le terme $((f\ x)\ y)$.

On le voit, les règles de typage des termes sont semblables à celles de la déduction naturelle. En fait, construire une preuve d'une formule F sous l'ensemble d'hypothèses Γ dans la déduction naturelle revient à trouver un terme bien typé de type F utilisant l'environnement Γ. Il y a équivalence entre ces deux notions, ce qui se caractérise formellement par un isomorphisme entre types et formules et entre termes et preuves. C'est d'ailleurs ce qu'explicite l'exemple suivant.

Exemple: On va démontrer le théorème $(P \to Q) \to (P \to R) \to (Q \to R \to T) \to P \to T$, dans la déduction naturelle, puis construire un terme de ce type. L'environnement est vide dans les deux cas.

Posons $\Delta = P \to Q, P \to R, Q \to R \to T, P$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta \vdash Q \to R \to T}\, hyp \quad \dfrac{\dfrac{}{\Delta \vdash P \to Q}\, hyp \quad \dfrac{}{\Delta \vdash P}\, hyp}{\Delta \vdash Q} \to_e}{\Delta \vdash R \to T} \to_e \quad \dfrac{\dfrac{}{\Delta \vdash P \to R}\, hyp \quad \dfrac{}{\Delta \vdash P}\, hyp}{\Delta \vdash R} \to_e}{\Delta \vdash T} \to_e}{\vdots \to_i}$$

$$\vdash (P \to Q) \to (P \to R) \to (Q \to R \to T) \to P \to T$$

Le terme du λ-calcul correspondant à cette preuve est :

$$\lambda f^{P \to Q} \lambda g^{P \to R} \lambda h^{Q \to R \to T} \lambda x^P.h\ (f\ x)\ (g\ x)$$

On vérifie facilement que toutes les règles de typage sont bien respectées, par une succession d'étapes identiques à la démonstration précédente, avec la construction progressive du terme.

Les termes du λ-calcul permettent de conserver toute la preuve dans un formalisme concis, mais ont l'inconvénient d'être relativement difficiles à construire. C'est pourquoi Coq, en remplissant son rôle d'assistant à la preuve, permet de construire ces termes de façon itérative à l'aide de tactiques.

2.2 Exemple de développement avec Coq

Avec Coq, la recherche de preuves est interactive (Coq attend les ordres de l'utilisateur) et s'effectue depuis le(s) but(s). L'état est affiché à chaque application d'une commande. On peut classer ces commandes en plusieurs catégories : les commandes de modification de l'environnement (p.ex. déclaration d'identificateurs), les ordres de modifications du comportement du logiciel (p.ex. ajout de notations), les requêtes au système (p.ex. demande du type d'une expression), et les tactiques. Ces dernières ne sont utilisables que dans le mode de démonstration. Le résultat de l'application des tactiques de Coq est similaire à l'utilisation des règles d'inférence dans la déduction naturelle (p.ex. $\rightarrow_i$ ou *hyp*). C'est d'ailleurs de ce point de vue qu'elles sont abordées dans l'exemple qui suit.

En reprenant l'exemple précédent, nous montrons comment Coq aide à la construction du terme typé.

Les commandes fournies sont placées à gauche, et les réponses de Coq à droite. Un court texte explicatif est associé à chaque ordre donné au système.

`Parameters P Q R T : Prop.`
 	
On définit P, Q, R et T comme symboles de propositions. Ils sont ajoutés à l'environnement.
	`P is assumed`
	`Q is assumed`
	`R is assumed`
	`T is assumed`

`Theorem diamond :`
`(P->Q)->(P->R)->(Q->R->T)->P->T.`
 	
On déclare qu'on veut définir et démontrer un théorème qu'on nomme `diamond`. Coq passe en mode de démonstration interactive. Il y a un seul but à résoudre qui est affiché en dessous de la ligne. Il n'y a pas d'hypothèse ; l'environnement est vide.
	`1 subgoal`
	
	`============================`
	`(P->Q)->(P->R)->(Q->R->T)->P->T`

`intros.`
 	
Cette tactique correspond à plusieurs applications de la règle $\rightarrow_i$. Attention, il faut se souvenir qu'on évolue depuis le but, et donc on « remonte » la déduction.
 	
Remarque: Coq choisit lui-même le nom des hypothèses qu'il introduit. En fait, il est possible de fournir les noms des hypothèses à la tactique : `intros H H0 H1 H2.`
	`1 subgoal`
	
	`H : P -> Q`
	`H0 : P -> R`
	`H1 : Q -> R -> T`
	`H2 : P`
	`============================`
	`T`

`apply H1.`
 	
Correspond à des applications successives de $\rightarrow_e$. On peut aussi le voir comme l'application d'une règle $\rightarrow_e$ généralisée
 	
$$\Gamma \vdash A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n$$
$$\frac{\Gamma \vdash A_1 \quad \Gamma \vdash A_2 \quad \ldots \quad \Gamma \vdash A_{n-1}}{\Gamma \vdash A_n} \rightarrow_{e^*}$$
avec $n = 3$, $A_3 = T$, $A_1 = Q$ et $A_2 = R$. Deux buts sont générés, seul le premier est affiché avec son environnement associé.
	`2 subgoals`
	
	`H : P -> Q`
	`H0 : P -> R`
	`H1 : Q -> R -> T`
	`H2 : P`
	`============================`
	`Q`
	
	`subgoal 2 is:`
	`R`

`apply H.`
 	
Correspond à une application de la règle $\rightarrow_e$. On avance dans la résolution du premier but, le second n'est pas modifié, ni son environnement associé.
	`2 subgoals`
	
	`H : P -> Q`
	`H0 : P -> R`
	`H1 : Q -> R -> T`
	`H2 : P`
	`============================`
	`P`
	
	`subgoal 2 is:`
	`R`

`assumption.`
 	
Correspond à l'application de la règle *hyp*. Le premier but est résolu, il ne reste plus que le second à résoudre.
	`1 subgoal`
	
	`H : P -> Q`
	`H0 : P -> R`
	`H1 : Q -> R -> T`
	`H2 : P`
	`============================`
	`R`

`apply H0; assumption.` C'est une tactique composée : applique `apply H0` au but courant, puis `assumption` à tous les sous-buts générés. Ici il n'y a qu'un seul but généré par `apply H0`.	`Proof completed.`

`Qed.`
On explicite la fin de la démonstration, comme en mathématiques. Coq va alors vérifier que le λ-terme créé est bien typé du type cherché. En plus il affiche l'historique de la preuve.
	`intros.`
	`apply H1.`
	`apply H.`
	`assumption.`
	`apply H0; assumption.`
	`diamond is defined`

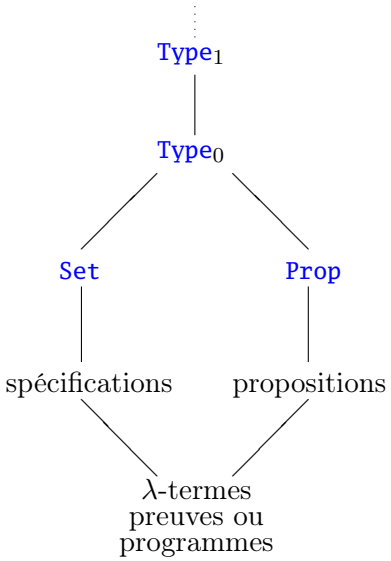
`Print diamond.`
On peut afficher le terme construit. Avec les notations usuelles du λ-calcul typé, le terme Coq correspond au terme :
$\lambda h^{P\to Q} h_0^{P\to R} h_1^{Q\to R\to T} h_2^P . h_1\ (h\ h_2)\ (h_0\ h_2)$
	`diamond = fun`
	`(H:P->Q) (H0:P->R) (H1:Q->R->T) (H2:P)`
	`=> H1 (H H2) (H0 H2)`
	`: (P->Q)->(P->R)->(Q->R->T)->P->T`

Finalement, Coq nous a assisté à la construction du même terme (à renommage de variables liées près) que précédemment. Ce terme a été créé en appliquant des tactiques dont le comportement est identique aux règles d'inférences de la déduction naturelle. Ceci souligne l'opérationnalité de l'isomorphisme de Curry-Howard, et donc de Coq.

2.3 Plus loin dans le calcul des constructions

Le calcul des constructions est bien plus expressif que le λ-calcul simplement typé. Dans ce formalisme, sur lequel s'appuie Coq, les types aussi sont des termes et ont un type. Naturellement, la construction de ces types est alors régie par des règles d'inférences.

Par exemple, les types `nat`, `Z` et `bool`, respectivement pour les entiers naturels, les entiers relatifs et les booléens sont prédéfinis en Coq. Leur type, puisque ce sont des termes, est la *sorte* `Set` : ce sont des spécifications. Cette sorte étant un terme, elle est de type $\texttt{Type}_0$, qui lui-même est de type $\texttt{Type}_1$, et ainsi de suite. Les λ-termes dont le type est un élément de type `Set` sont des programmes.
Coq permet de manipuler programmes et preuves sous le même formalisme, des λ-termes. Mais les preuves sont distinguées par leur type. Le type des propositions est `Prop`. Les éléments de type `Prop` sont donc des propositions, et les éléments dont le type est une proposition sont des preuves (voir exemple précédent). De la même façon, `Prop` est une *sorte* de type $\texttt{Type}_0$.
Grossièrement, cette hiérarchie de types peut être représentée par le schéma ci-contre.



Remarque: Il est nécessaire d'avoir une hiérarchie de types $\texttt{Type}_i$ pour éviter le *paradoxe de Girard* (cf. [11]). Cependant, Coq n'affiche pas l'indice associé à `Type`, et le système de typage s'arrange pour conserver la cohérence et inférer les bons indices. En plus des règles de convertibilité des types (cf. sec. 4.1, p.49), tout ce qui est de type `Set` ou `Prop` est de type $\texttt{Type}_i$, et $\forall i < j$ $\texttt{Type}_i$ est de type $\texttt{Type}_j$.

Afin de bénéficier d'un pouvoir expressif suffisant, Coq dispose aussi du produit dépendant. Il s'agit d'autoriser la construction de types dépendants de paramètres ; des familles de types. Ces types sont représentés à l'aide du quantificateur universel, et sont une généralisation de la construction des types flèches.

La règle de construction de ces types est :

$$\frac{\Gamma \vdash A : s \qquad \Gamma, a : A \vdash B : s'}{\Gamma \vdash \forall a : A, B : s''} \, Prod(s, s', s'')$$

où les types s, s' et s'' sont restreints à certaines combinaisons (voir [3, ch.4]).

Exemple: Présentons l'intérêt de quelques types qu'il est possible de construire, et donnons-leur une interprétation.

- `Z → nat` est un cas particulier de produit dépendant, où le type `nat` ne dépend pas du premier argument de type `Z`. Ce pourrait être le type de la fonction valeur absolue.
- `nat → nat → Prop` typiquement le type d'un prédicat, qui pourrait être le type de la fonction qui à deux éléments a et b de type `nat`, associe la proposition « a divise b ».
- `∀ P : Prop, P → P` est une proposition d'ordre supérieure. Un élément de ce type en est une preuve. Cela permet des définitions imprédicatives (cf. [19, ch.5]), et donc de tous les connecteurs logiques usuels (voir des exemples dans [14]) ;
- `nat → Set` qui pourrait être le type des tableaux paramétrés par leur longueur. Un constructeur de tableaux, nommons-le `tab`, pourrait avoir ce type, tandis qu'un élément de type `tab n` serait un tableau de longueur `n`.
- `∀ A : Set, (A → A) → nat → A → A` permet le polymorphisme des programmes. Par exemple, la fonctionnelle d'itération (`iterate`) d'une fonction (`f`) un nombre (`n`) donné de fois avec cas de base (`b`) aurait ce type (`iterate A f n b`). Tandis que la récursion sur les entiers aurait comme type une instance avec `A=nat` (`iterate nat`). La récursion sur les fonctions d'entiers est aussi une instance de ce type (`iterate (nat → nat)`). Ce qui permet de programmer facilement la fonction d'*Ackermann* (cf. [3, p.95]).

La construction de termes d'un type dépendant n'est en fait qu'une généralisation de l'abstraction.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : \forall x : A, B} \, \lambda \qquad \frac{\Gamma \vdash t : \forall v : A, B \qquad \Gamma \vdash x : A}{\Gamma \vdash t \; x : B[v \leftarrow x]} \, app$$

Exemple: Un exemple de terme dont le type est un produit dépendant :

$$\lambda P^{\mathtt{Prop}} \lambda x^P.x : \forall P, P \to P$$

Si nous l'appliquons à la proposition $Q \to Q$, nous obtenons :

$$\lambda x^{Q \to Q}.x : (Q \to Q) \to (Q \to Q)$$

Dans la version actuelle, Coq se base sur le calcul des constructions *inductives* et les opérateurs logiques sont exprimés de façon inductive — par des axiomes. Nous n'utiliserons pas directement cet aspect dans le développement, et donc nous ne le présentons pas.

2.4 Mécanisme des sections de Coq

Lorsque nous avons donné les règles d'inférences précédentes, nous avons parlé d'environnement. En fait, à chaque instant, il faut en distinguer deux parties : un environnement global et un environnement local. L'environnement global met à disposition des définitions pour tout le développement. Par exemple les définitions des connecteurs logiques et les axiomes et théorèmes associés, les types de données prédéfinis, etc.

Mais il est aussi possible d'ajouter temporairement des définitions dans le but de construire un objet ou de démontrer un théorème, comme cela se fait en programmation et en mathématiques.

À la fin d'une section, toutes les hypothèses sont abstraites et déchargées (règle λ) dans les définitions qui étaient locales. Ainsi il est possible d'utiliser ces définitions de façon globale. Les définitions ne sont donc pas restreintes aux sections, elles y ont seulement une forme simplifiée.

Exemple:

```
Section Diamond.

  Variables P Q R T : Prop.

  Theorem diamond : (P → Q) → (P → R) → (Q → R → T) → P → T.
  Proof.
  ...
  Qed.

End Diamond.
```

À l'intérieur de la section, le type de `diamond` est `(P→Q)→(P→R)→(Q→R→T)→P→T`. Mais une fois la section close, tout ce qu'utilise `diamond` pour sa définition est déchargé, et le type de `diamond` devient `∀ P Q R T : Prop, (P→Q)→(P→R)→(Q→R→T)→P→T`.

Nous emploierons régulièrement ce mécanisme pour structurer le code Coq. D'autant plus que son utilisation est simple et intuitive.

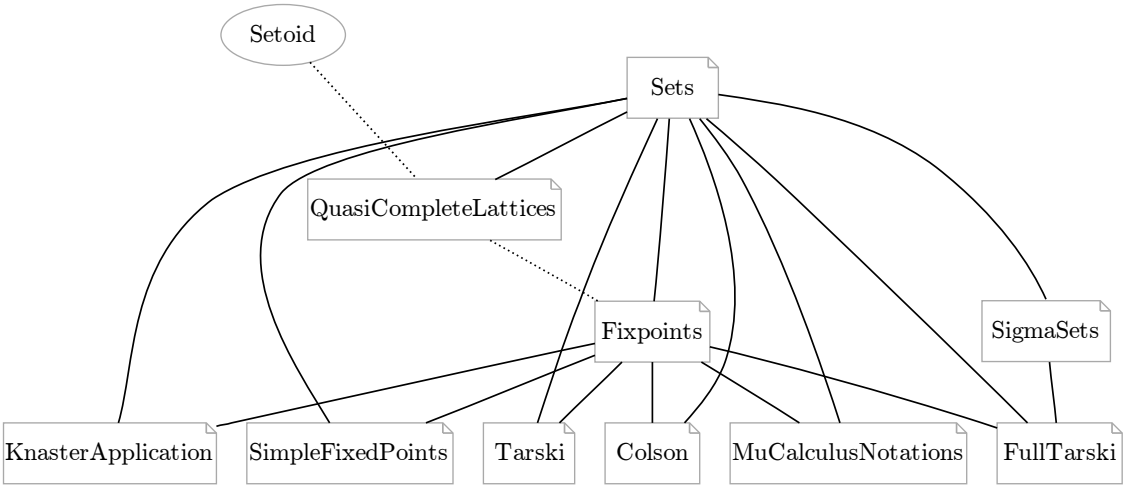
Chapitre 3 Formalisation

La formalisation consiste à traduire des notions mathématiques de treillis et de points fixes dans le calcul des constructions, et plus spécifiquement dans le langage de Coq. Cette formalisation est organisée en plusieurs fichiers de scripts, des sources Coq, qui dépendent les uns des autres. Cela s'apparente, en mathématiques, à la création d'une théorie et à son utilisation dans plusieurs contextes. Cette conception est en fait très développée et utilisée par différents langages de programmation.

Organisation

L'organisation générale de cet ensemble de fichiers est résumée par le graphe des dépendances suivant. Chaque fichier est représenté par un nœud. Un arc orienté de haut en bas d'un nœud a vers un nœud b signifie que le fichier b utilise des définitions contenues dans le fichier a. Un arc en pointillés entre un nœud a et un nœud b indique que tous les fichiers qui importent les éléments du nœud b vont automatiquement récupérer les définitions de a, on parlera d'importations implicites.

Le choix de la découpe est lié aux thèmes, aux articles dans lesquels on trouve les théorèmes énoncés, ainsi qu'au niveau de difficulté. Plutôt que d'utiliser des importations implicites dans le but de réduire les liens du schéma, il a été préféré de les utiliser en fonction du contexte, pour marquer des dépendances de bon sens. Par exemple, les théorèmes de point fixes (*fixpoints*) développés n'ont un sens que dans un treillis (*lattice*). Par contre, la définition d'un ensemble (*set*) n'est pas nécessaire lors de la manipulation de points fixes.



Tous ces éléments utilisent le fichier `utf8.v` fourni avec l'environnement graphique de développement CoqIde. Il permet l'utilisation de certaines notations mathématiques prédéfinies. Ces notations font usage des symboles unicode[1], et facilitent la lecture des théorèmes. Elles sont évidemment indispensables pour se rapprocher des notations mathématiques.

CoqIde permet de suivre et de développer les preuves de façon interactive. Pour ce développement, les versions 8.1pl3 de Coq et de CoqIde de décembre 2007 ont été utilisées.

Conventions

Il s'agit d'un travail de formalisation, avec pour objectif principal de respecter le style mathématique. Plutôt que de présenter les définitions et théorèmes à formaliser puis le code Coq correspondant, nous ne proposons que les sources Coq. Nous verrons que les constructions sont suffisamment proches des habitudes mathématiques, et qu'il n'y aura aucune difficulté à les comprendre. Les démonstrations seront données telles que l'on peut les suivre avec CoqIde.

Le code source de Coq est présenté avec une police à chasse fixe, une barre verticale dans la marge pour le différencier du texte, et avec les mêmes couleurs que celles qu'utilise CoqIde.

Tous les symboles mathématiques utilisés dans les sources sont des symboles unicode, et sont réellement présents dans les scripts Coq.

3.1 Les notations

Avec Coq, il est possible de définir des raccourcis syntaxiques pour désigner des termes ou des constructions particulières. Par exemple, si l'on définit l'addition `add` de deux entiers, nous devons écrire `add x y`. Habituellement, en mathématiques, on utilise une notation pour cela, et on écrit plutôt $x + y$.

Hormis la possibilité d'écrire de façon infixée, des constructions plus complexes sont possibles. Par exemple, un couple s'écrit mieux (a, b) que `pair a b`.

Durant la formalisation, nous allons rencontrer des définitions de notations. Sans entrer dans les détails (voir [21, ch.11]), nous allons exposer certains des aspects qui justifient et qui régissent ces notations.

Prenons les exemples des notations de l'addition et de la multiplication sur les entiers naturels, respectivement `plus` et `mult` en Coq. Lorsque l'on définit les notations $+$ et $\times$ pour ces deux connecteurs respectivement, nous devons prendre en compte différents paramètres :

- définir une règle d'associativité sur $+$ pour lever l'ambiguïté tout en conservant une syntaxe légère lors d'expressions de la forme $x + y + z$;
- définir des règles de précédences entre opérateurs pour pouvoir évaluer des expressions de la forme $x + y \times z$;
- définir un contexte d'utilisation de la notation, si nous décidons d'utiliser les mêmes symboles pour l'addition et la multiplication sur les entiers relatifs par exemple.

[1] http://unicode.org/

C'est exactement ce que permet Coq. Par exemple, ces notations sont définies ainsi :

```
Notation "x + y" := (plus x y) (at level 50, left associativity) : nat_scope.
Notation "x * y" := (mult x y) (at level 40, left associativity) : nat_scope.
```

La notation `*` possède un niveau (*level*) de priorité moins élevé que `+` ; l'opérateur `*` est donc prioritaire sur `+`. Ainsi `x + y * z` est interprété en `x + (y * z)`, c'est-à-dire représente le terme `plus x (mult y z)`.

Les deux opérateurs sont associatifs à gauche, donc `x + y + z` est évalué et traduit en `(x + y) + z`, ce qui représente le terme `plus (plus x y) z`, de même pour `*`.

Ces deux notations sont liées à une « portée », `nat_scope`, celle des entiers naturels. Cela signifie que lorsque l'on travaille sur les entiers naturels, `nat`, alors le symbole `+` désignera l'addition sur les éléments de type `nat`. Mais il reste possible d'associer une même notation à un terme différent si elle est liée à une autre portée ; on peut surcharger les notations.

Les ambiguïtés entre types peuvent également être traitées par un mécanisme de conversions implicites (cf. [21, ch.16]) de Coq, et permet des notations encore plus naturelles. Ce mécanisme n'est pas utilisé et n'a pas été nécessaire dans notre développement. En effet, nous ne travaillons que dans les treillis, et bien que la notation de l'opérateur d'ordre surcharge celle de l'ordre sur les naturels `nat`, les deux éléments ne coexistent pas dans une même expression.

Voici par exemple quelques notations prédéfinies de Coq (ce tableau est non exhaustif).

symbole	niveau	associativité	portée(s)
`F` $\leftrightarrow$ `G`	95	no	`type_scope`
`F` $\vee$ `G`	85	right	`type_scope`
`F` $\wedge$ `G`	80	right	`type_scope`
$\neg$ `F`	75	right	`type_scope`
`a = b`	70	no	`type_scope`
`m < n`	70	no	`nat_scope`
`m > n`	70	no	`nat_scope`
`m` $\leq$ `n`	70	no	`nat_scope`
`m` $\geq$ `n`	70	no	`nat_scope`
`m + n`	50	left	`nat_scope`
`m - n`	50	left	`nat_scope`
`m * n`	40	left	`nat_scope`
`m / n`	40	left	`nat_scope`
`m ^ n`	30	right	`nat_scope`

Il faut bien définir les niveaux de précédences des nouvelles notations, afin de permettre des écritures aussi naturelles que possible. Il a donc fallu être très vigilant lors de la définition des notations, et choisir avec minutie les différents paramètres, notamment le niveau de priorité.

3.2 Les ensembles

```
Implicit Type Ω : Type.
```

Dans la suite, Ω désignera un élément de type `Type` (donc un type). Ce comportement est usuel dans les textes scientifiques. Cette possibilité rapproche du *style* mathématique.

En mathématiques, on peut définir un ensemble en extension en fournissant la liste explicite de ses éléments, ou en intention par une formule que doivent satisfaire les éléments de l'ensemble. La définition en intention est en fait un prédicat. C'est ainsi qu'on a défini un ensemble avec Coq.

```
Definition ℘ Ω := Ω → Prop.
```

Par conséquent, un ensemble d'entiers sera de type `℘(nat)`, donc de type `nat → Prop`. Pour utiliser la notation habituelle des ensembles — par exemple $\{x; x = f(x)\}$ — il suffit alors de définir une variante syntaxique à la définition d'une fonction. Mais pour ne pas couvrir toutes les déclarations de fonction, nous obligeons cette variante à être du type des ensembles qu'on vient de définir.

```
Notation "{ x : T ; P }" := ( (fun x : T => P) : ℘(T) )
        (at level 0, x at level 99) : type_scope.
Notation "{ x ; P }" := ( (fun x => P) : ℘(_) )
        (at level 0, x at level 99) : type_scope.
```

La variable `x` apparaît en général dans la formule `P` qui est de type `Prop`. Elle est alors liée par le terme `fun x => P` construit lorsque la notation est interprétée.

Remarque: Les ensembles ainsi formalisés sont typés. C'est-à-dire qu'ils sont composés d'éléments d'un même type. Par exemple, il n'est pas directement possible d'avoir des éléments de type `nat` et de type `Z` dans un même ensemble. On pourra parler dans la suite du type d'un ensemble en faisant référence au type de ses éléments.

```
Set Implicit Arguments.
```

Dorénavant, Coq va décider de lui-même si les premiers arguments des fonctions sont implicites ou non. Des arguments sont déclarés implicites lorsque leur valeur peut-être déduite du type des autres arguments de la fonction. Et donc, lors de l'appel d'une fonction, il n'est plus nécessaire de fournir d'élément pour les arguments implicites.

Si $E = \{x; P(x)\}$ est un ensemble d'éléments du même type que z, prouver que $z \in E$ revient à prouver $P(z)$. Or notre formalisation des ensembles est telle que `E : ℘(T)` est le prédicat P. Donc $z \in E$, c'est en fait `E z`.

```
Definition member Ω x (Y : ℘(Ω)) := Y x.
Notation "x ∈ Y" := (member x Y)
        (at level 70, no associativity) : type_scope.
```

Remarque: Ici, le paramètre Ω est automatiquement déclaré implicite puisqu'il peut être déduit du type de `Y` lors de l'application (totale) de la fonction `member`. De plus le type de `x` n'est pas explicité lors de la définition ; Coq peut le déduire du type de `Y` et du fait qu'on applique `Y` à `x`.

On définit l'inclusion d'un ensemble dans un autre.

```
Definition subset Ω (X Y : ℘(Ω)) := ∀ x, x ∈ X → x ∈ Y.
Notation "X ⊆ Y" := (subset X Y)
        (at level 70, no associativity) : type_scope.
```

Utiliser cette notation et exprimer l'inclusion dans notre formalisme implique que les deux ensembles considérés soient de même type. En particulier pour l'ensemble vide. Il n'y a donc pas un ensemble vide, mais des ensembles vides, un pour chaque type d'ensemble. De façon similaire, la notation d'« ensemble plein » est nécessaire. Or si une notation répandue de l'ensemble vide existe[2], il en est autrement de l'ensemble plein. Nous avons alors choisi[3] de le noter $\mathcal{E}$.

```
Notation "∅" := (fun _ => False) : type_scope.
Notation "'ℰ'" := (fun _ => True) : type_scope.
```

L'utilisation de ces ensembles nécessite un contexte qui rende leur typage possible. Par exemple l'inclusion dans un autre ensemble dont on connaît le type, ou l'appartenance d'un élément typé à l'un de ces ensembles. Ces notations vont cacher le type implicite de ces ensembles particuliers.

Remarque: Coq « normalise » l'affichage des termes qu'il va présenter pour ensuite leur associer des notations prédéfinies. Et donc il cherche une notation à `fun _ => False` bien qu'on lui ai donné le terme `{x; False}` ou `fun x => False` ; la variable `x` ne jouant aucun rôle, son affichage n'est pas nécessaire et donc elle est oubliée. C'est pourquoi les deux précédents ensembles sont définis sans utiliser notre notation ensembliste.

Il est alors facile de définir la paire de deux éléments d'un même type.

```
Definition pair Ω (a b : Ω) := {x : Ω ; x = a ∨ x = b }.
Notation "{ a , b }" := (pair a b)
        (at level 0, a at level 99, b at level 99) : type_scope.
```

Puis l'union et l'intersection de deux ensembles.

```
Definition union Ω (A B : ℘(Ω)) := {x : Ω; x ∈ A ∨ x ∈ B}.
Definition inter Ω (A B : ℘(Ω)) := {x : Ω; x ∈ A ∧ x ∈ B}.
Notation "A ∪ B" := (union A B)
        (at level 65, right associativity) : type_scope.
Notation "A ∩ B" := (inter A B)
        (at level 65, right associativity) : type_scope.
```

Et plus généralement l'union et l'intersection d'une famille d'ensembles.

```
Definition cap Ω (S : ℘(℘ Ω)) := {x : Ω; ∀ X, X ∈ S → x ∈ X }.
Definition cup Ω (S : ℘(℘ Ω)) := {x : Ω; ∃ X, X ∈ S ∧ x ∈ X }.
Notation "⋂ S" := (cap S)
        (at level 60, right associativity) : type_scope.
Notation "⋃ S" := (cup S)
        (at level 60, right associativity) : type_scope.
```

[2] La notation ∅ est attribuée à André Weil du groupe Nicolas Bourbaki. Elle est utilisée dans le traité de mathématiques *Éléments de mathématique, livre 1 : Théorie des ensembles*, publié en 1939 par le groupe. C'est la notation pour représenter le terme $\forall x, x \notin X$ qui est une relation fonctionnelle en X.

[3] En référence à un cours de topologie accessible à l'adresse http://mathsplp.creteil.iufm.fr/ht_works/exposes/topo0198/topo0198.htm, et à défaut de pouvoir utiliser Ω comme en probabilités — ici il désigne notre type de travail, notre type « univers ».

Puisque la définition sera nécessaire pour la suite, on se donne la composition de deux fonctions, ainsi que la notation associée. Et un théorème prouvant l'associativité de la composition, la preuve étant triviale, elle n'est pas donnée.

```
Definition comp (E F G : Type) (f : F → G) (g : E → F) :=
        fun x => f(g(x)).
Notation "f '∘' g" := (comp f g)
        (at level 5, right associativity) : type_scope.
```

```
Theorem comp_assoc E F G K (f : G → K) (g : F → G) (h : E → F) :
        f∘(g∘h) = (f∘g)∘h.
```

```
Unset Implicit Arguments.
```

Les définitions de ce fichier étant terminées, on signale à Coq de ne plus décider des arguments qui seront implicites.

Et on termine avec quelques raccourcis syntaxiques usuels.

```
Notation "∀ x ∈ Y , F" := (∀ x , x ∈ Y → F)
        (at level 200, x ident) : type_scope.
Notation "∃ x ∈ Y , F" := (∃ x , x ∈ Y ∧ F)
        (at level 200, x ident) : type_scope.
Notation "∀ X ⊆ A , F" := (∀ X, X ⊆ A → F)
        (at level 200, X ident) : type_scope.
Notation "∃ X ⊆ A , F" := (∃ X, X ⊆ A ∧ F)
        (at level 200, X ident) : type_scope.
```

Et des variantes de ces notations, avec quantification sur plusieurs variables (jusqu'à trois), que nous ne donnons pas.

3.3 Les treillis

```
Require Import Sets.
Require Export Setoid.
```

Les définitions du précédent fichier `Sets.v` sont nécessaires à l'implémentation des treillis, ainsi que le module `Setoid` (cf. sec. 4.5, p.52) prédéfini dans les bibliothèques standards de Coq (cf. [21, ch.21]). Ce dernier va notamment nous permettre l'utilisation des tactiques générales comme `reflexivity`, `transitivity` et `symmetry` pour des relations d'équivalence. Il apporte une extension au traitement de l'égalité de Coq, tout en autorisant une certaine homogénéité de traitement.

Un treillis complet est la donnée de trois opérateurs et d'un type. Une relation d'ordre, un opérateur de borne inférieure, et un opérateur de borne supérieure. Une relation d'ordre doit respecter trois propriétés : être réflexive, transitive et antisymétrique. L'opérateur de borne inférieure doit, étant donné un ensemble d'éléments d'un type donné, construire le plus grand des minorants de l'ensemble. L'opérateur de borne supérieure doit, lui, construire le plus petit des majorants de l'ensemble. Un treillis complet est alors un n-uplet contenant le type de travail, les opérateurs ainsi que les propriétés que doivent satisfaire ces opérateurs.

```
Structure QuasiCompleteLattice : Type :=
Build_QuasiCompleteLattice {
  Ω :> Type ;
  le : Ω → Ω → Prop ;
  inf : ℘(Ω) → Ω;
  sup : ℘(Ω) → Ω;

  le_refl : ∀ x, le x x ;
  le_trans : ∀ x y z, le x y → le y z → le x z ;

  inf_lower : ∀ A, ∀ x ∈ A, le (inf A) x ;
  inf_greatest : ∀ A, ∀ z, (∀ x ∈ A, le z x) → le z (inf A) ;

  sup_upper : ∀ A, ∀ x ∈ A, le x (sup A) ;
  sup_least : ∀ A, ∀ z, (∀ x ∈ A, le x z) → le (sup A) z
}.
```

Ainsi, créer un élément de type `QuasiCompleteLattice`, c'est aussi fournir les preuves que les opérateurs respectent bien les propriétés nécessaires. On est donc assuré de manipuler un treillis en manipulant un objet de ce type. En cela, les structures de Coq diffèrent de celles des langages de programmation usuels (C, Pascal, etc.).

Notre relation `le` ne possède pas la propriété d'antisymétrie, c'est un préordre. C'est pourquoi nous avons nommé cette structure quasi treillis complet (*quasi complete lattice*). La raison en est simple ; nous voulons éviter de forcer la déclaration de l'axiome d'extensionalité (cf. sec. 4.2, p.50).

En effet, prenons le cas historique du théorème de Knaster [15]. Si nous voulons le démontrer comme un cas particulier du théorème de Tarski [20], alors il faudra instancier un treillis complet avec la relation d'inclusion sur les ensembles — qui est une relation d'ordre bien connue. C'est-à-dire qu'il faudra montrer que l'inclusion sur les ensembles

est antisymétrique : $\forall XY, X \subseteq Y \rightarrow Y \subseteq X \rightarrow X = Y$. Or montrer l'égalité de deux ensembles dans notre formalisme nécessite l'extensionalité.

De plus, s'il s'avère que la relation utilisée est effectivement antisymétrique sur l'égalité de Coq, nous avons prévu la définition (voir plus loin) d'une relation d'équivalence ($\simeq$) qui coïncidera dans ce cas avec l'égalité de Coq.

Cependant, par la suite nous parlerons de treillis complet pour désigner notre quasi treillis complet.

Remarque: La syntaxe `Ω :> Type` utilisée pour déclarer le type des éléments du treillis permet les mêmes abus d'écriture que les mathématiciens. Par exemple :

> Si L est un treillis complet, alors $\forall x \in L, x \leq x$ est vérifiée.

Rigoureusement, il aurait fallu écrire :

> Si $(L, \leq, \bigwedge, \bigvee)$ est un treillis complet, alors $\forall x \in L, x \leq x$ est vérifiée.

Concrètement en Coq, cela ce traduirait par :

> `∀ L : QuasiCompleteLattice, ∀ x : (Ω L), x ≤ x.`

Ce n'est pas très agréable. Le fait d'indiquer `Ω :> Type` à Coq lui permet de générer un « convertisseur » d'éléments de type `QuasiCompleteLattice` vers le type des éléments de la structure. Cette conversion implicite est effectuée en coulisses (voir [21, ch.16]). Nous sommes alors autorisés à écrire :

> `∀ L : QuasiCompleteLattice, ∀ x : L, x ≤ x.`

Habituellement, dans les treillis complets, on note la relation d'ordre de façon infixe par $\leq$ et les bornes inférieures et supérieures respectivement par $\bigwedge$ et $\bigvee$. Mais la notation $\leq$ existe déjà pour les entiers de Coq, le type `nat`. Il faut donc surcharger cette notation, et pour cela définir une nouvelle portée qu'on associe au type `QuasiCompleteLattice`.

```
Bind Scope qclattice_scope with QuasiCompleteLattice.
Delimit Scope qclattice_scope with qclattice.
```

Ces deux constructions ont respectivement les effets suivants :

1. Une nouvelle portée nommée `qclattice_scope` est liée avec le type de notre treillis complet `QuasiCompleteLattice`. Toute fonction attendant un élément de ce type comme argument va ouvrir automatiquement et temporairement la portée associée `qclattice_scope` pour interpréter les notations utilisées par cet argument.
2. Pour ouvrir temporairement une portée, il faut encadrer un terme de parenthèses et accoler le nom de la portée à ouvrir — `(-1 + (2 + 3)%nat)%Z`. Comme les noms de portées peuvent être longs — `nat_scope`, `Z_scope` — il est préférable de fournir un raccourci de nom de portée pour cette construction particulière. Ici le raccourci pour `qclattice_scope` est `qclattice`.

```
Open Local Scope qclattice_scope.
```

On ouvre la portée que l'on vient de définir. Le mot clé `Local` signifie que la portée sera fermée à la fin du fichier. Toutes les notations (pour l'instant aucune) de cette portée nous sont accessibles en priorité. Cela correspond à la pratique courante dans les textes mathématiques :

> Nous allons travailler dans les treillis complets et utiliser les notations usuelles.

```
Implicit Type L : QuasiCompleteLattice.
```

Dans la suite, `L` désignera un quasi treillis complet.

Il est temps de définir les notations habituelles[4], ainsi que l'opérateur `ge` symétrique de `le` que l'on va prochainement utiliser. Par la même occasion, définissons les éléments particuliers $\top$ (*top*) et $\bot$ (*bottom*) d'un treillis complet, qui sont respectivement le plus grand et le plus petit élément du treillis.

```
Definition ge L x y := le L y x.

Notation "x ≤ y" := (le _ x y)
        (at level 70, no associativity) : qclattice_scope.
Notation "x ≥ y" := (ge _ x y)
        (at level 70, no associativity) : qclattice_scope.
Notation "⋀ X" := (inf _ X)
        (at level 60, no associativity) : qclattice_scope.
Notation "⋁ X" := (sup _ X)
        (at level 60, no associativity) : qclattice_scope.

Notation "x ≤ y ≤ z" := (x ≤ y ∧ y ≤ z )
        (at level 70, y at next level, no associativity) : qclattice_scope.
Notation "x ≥ y ≥ z" := (x ≥ y ∧ y ≥ z )
        (at level 70, y at next level, no associativity) : qclattice_scope.

Definition top L := ⋁{x : L; True}.
Definition bot L := ⋀{x : L; True}.

Notation "⊤" := (top _)
        (at level 65) : type_scope.
Notation "⊥" := (bot _)
        (at level 65) : type_scope.
```

Remarque: Les définitions de `top` et `bot` n'utilisent pas la notation de l'ensemble plein (cf. sec. 3.2). En effet, Coq n'a pas suffisamment d'indices pour déduire à la fois le type des éléments de l'ensemble plein utilisé, et celui de l'argument implicite[5] des opérateurs de bornes inférieures ou supérieures.

Quelques théorèmes triviaux sur $\bot$ et $\top$.

```
Section top_and_bottom.

  Variable L : QuasiCompleteLattice.

  Theorem bot_le_all : ∀ x : L, ⊥ ≤ x.
  Theorem all_le_top : ∀ x : L, x ≤ ⊤.

End top_and_bottom.

Implicit Arguments bot_le_all [L].
Implicit Arguments all_le_top [L].
```

[4] En fait, il existe des constructions syntaxiques permettant de définir les notations en même temps que la définition des objets (cf. [21, ch.11]). Sauf que, pour une raison inconnue, il n'y a pas de telle notation pour les structures de Coq, ou alors non documentée et difficile d'accès.

[5] L'argument est implicite car les notations $\bigwedge$ et $\bigvee$ utilisent le symbole spécial _. C'est une autre façon de rendre des arguments implicites, dont la valeur sera déduite par le système de typage.

Les deux dernières commandes ordonnent à Coq de déclarer le paramètre L, qui est déchargé lors de la fermeture de la section, comme implicite pour les deux théorèmes. En effet, un théorème s'applique tout comme une fonction — en fait c'en est une. Cette possibilité est donc tout à fait justifiée.

Un treillis complet est un cas particulier de treillis. Les opérateurs de bornes inférieure et supérieure y sont binaires. C'est-à-dire qu'ils agissent uniquement sur deux éléments. Il suffit alors de construire ces opérateurs binaires à partir des opérateurs du treillis complet, et de montrer qu'ils vérifient bien les conditions de borne inférieure et supérieure binaire. Les notations usuelles de ces opérateurs binaires sont explicitées dans le source qui suit.

```
Definition inf_bin L (x y : L) := ⋀ {x , y}.
Definition sup_bin L (x y : L) := ⋁ {x , y}.
Notation "x ⊓ y" := (inf_bin _ x y)
        (at level 65, right associativity) : qclattice_scope.
Notation "x ⊔ y" := (sup_bin _ x y)
        (at level 65, right associativity) : qclattice_scope.
```

Les propriétés des opérateurs binaires, évidentes à démontrer, sont les suivantes.

```
Section Quasi_lattice.

  Variable L : QuasiCompleteLattice.
  Variables x y z : L.

  Theorem inf_bin_lower_fst : x ⊓ y ≤ x.
  Theorem inf_bin_lower_snd : x ⊓ y ≤ y.
  Theorem inf_bin_greatest : z ≤ x → z ≤ y → z ≤ x ⊓ y.

  Theorem sup_bin_upper_fst : x ≤ x ⊔ y.
  Theorem sup_bin_upper_snd : y ≤ x ⊔ y.
  Theorem sup_bin_least : x ≤ z → y ≤ z → x ⊔ y ≤ z.

End Quasi_lattice.
```

Le principe de dualité est une notion importante des treillis complets[6]. Celui-ci dit en substance que pour toute propriété $\mathcal{P}$ vraie pour tous les treillis complets, la propriété $\mathcal{P}^\partial$ l'est aussi, où $\mathcal{P}^\partial$ est obtenue depuis $\mathcal{P}$ en remplaçant les occurrences de $\leq$ par $\geq$, de $\bigwedge$ par $\bigvee$ et de $\bigvee$ par $\bigwedge$ et réciproquement. En fait, c'est une conséquence immédiate de la transformation d'un treillis complet $(L, \leq, \bigwedge, \bigvee)$ en son treillis complet dual $(L, \geq, \bigvee, \bigwedge)$. Alors toute propriété vraie pour tous les treillis est vraie pour tous les treillis duaux. Ceci n'est qu'une paraphrase du principe de dualité.

```
Section Duality.

  Variable L : QuasiCompleteLattice.
  Implicit Type x y z : L.
  Implicit Type A : ℘(L).

  Lemma ge_refl : ∀ x : L, x ≥ x.
```

[6] Cette notion étant déjà valable dans le cas général des ensembles partiellement ordonnés (*posets*) (cf. [8, ch.1]).

```
  Lemma ge_trans : ∀ x y z : L, x ≥ y → y ≥ z → x ≥ z.

  Definition inf_ge := sup L.
  Lemma inf_ge_lower : ∀ A, ∀ x ∈ A, inf_ge A ≥ x.
  Lemma inf_ge_greatest : ∀ A, ∀ z, (∀ x ∈ A, z ≥ x) → z ≥ inf_ge A.

  Definition sup_ge := inf L.
  Lemma sup_ge_upper : ∀ A, ∀ x ∈ A, x ≥ sup_ge A.
  Lemma sup_ge_least : ∀ A, ∀ z, (∀ x ∈ A, x ≥ z) → sup_ge A ≥ z.

  Definition dual := Build_QuasiCompleteLattice L
    (ge L) inf_ge sup_ge ge_refl ge_trans
    inf_ge_lower inf_ge_greatest sup_ge_upper sup_ge_least.

End Duality.
```

Notons que les preuves des propriétés des opérateurs d'un treillis doivent être effectuées *a priori*.

Après le déchargement de la variable `L` à la fin de la section `Duality`, `dual` devient une fonction qui à un treillis complet donné lui associe son dual. C'est un constructeur de treillis. De la même façon, il est possible de construire le produit de deux treillis (cf. [8, ch.1]).

```
Section binary_product.

  Variables L₁ L₂ : QuasiCompleteLattice.

  Definition le_prod (x y :  L₁ * L₂) :=
    let (x₁, x₂) := x in
    let (y₁, y₂) := y in
      x₁ ≤ y₁ ∧ x₂ ≤ y₂.
  Definition inf_prod (X : ℘(L₁ * L₂)) :=
    ( ⋀{x₁ ; ∃z, (x₁, z) ∈ X}, ⋀{x₂ ; ∃z, (z, x₂) ∈ X}).
  Definition sup_prod (X : ℘(L₁ * L₂)) :=
    ( ⋁{x₁ ; ∃z, (x₁, z) ∈ X}, ⋁{x₂ ; ∃z, (z, x₂) ∈ X}).

  Lemma le_prod_refl : ∀ x, le_prod x x.
  Lemma le_prod_trans : ∀ x y z, le_prod x y → le_prod y z → le_prod x z.
  Lemma inf_prod_lower : ∀ A, ∀ x ∈ A, le_prod (inf_prod A) x.
  Lemma inf_prod_greatest :
    ∀A, ∀z, (∀x ∈ A, le_prod z x) → le_prod z (inf_prod A).
  Lemma sup_prod_upper : ∀ A, ∀ x ∈ A, le_prod x (sup_prod A).
  Lemma sup_prod_least :
    ∀A, ∀z, (∀x ∈ A, le_prod x z) → le_prod (sup_prod A) z.

  Definition product := Build_QuasiCompleteLattice (L₁ * L₂)
    le_prod inf_prod sup_prod le_prod_refl le_prod_trans
    inf_prod_lower inf_prod_greatest sup_prod_upper sup_prod_least.

End binary_product.
```

Revenons sur la dualité. Nous avons vu que les ensembles que nous manipulons sont typés. Il en va de même avec les opérateurs des treillis, même si leur type est masqué par les notations ou les arguments implicites. Cela peut poser problème, et nous pouvons être surpris de ne pouvoir montrer un but de la forme `x` $\leq$ `y`, avec une hypothèse `x` $\leq$ `y`, ou être en mesure de le démontrer en utilisant l'hypothèse `y` $\leq$ `x`. C'est pourquoi nous définissons de nouvelles notations pour faire apparaître le lien des opérateurs manipulés avec le dual.

```
Notation "x '≤ᵈ' y" := (le (dual _) x y)
        (at level 70, no associativity) : qclattice_scope.
Notation "x '≥ᵈ' y" := (ge (dual _) x y)
        (at level 70, no associativity) : qclattice_scope.
Notation "'⋀ᵈ' X" := (inf (dual _) X)
        (at level 60, no associativity) : qclattice_scope.
Notation "'⋁ᵈ' X" := (sup (dual _) X)
        (at level 60, no associativity) : qclattice_scope.
```

Il est maintenant plus simple de voir que l'on peut démontrer `x` $\leq$ `y` avec l'hypothèse `y` $\leq^{\mathrm{d}}$ `x`. Bien entendu, on a le même problème lorsque l'on manipule des éléments du dual et du dual du dual d'un treillis dans un même contexte.

Définissons le principe de dualité de façon explicite, et un exemple de son application. Une fois le premier théorème démontré, la preuve du second se déduit par dualité de façon triviale.

```
Theorem duality_principle: ∀ P: QuasiCompleteLattice → Prop,
  (∀ L, P L) → ∀ L, P (dual L).

Implicit Arguments duality_principle [P].

Section Inclusion.

  Theorem incl_inf L : ∀ X Y : ℘(L), X ⊆ Y → ⋀Y ≤ ⋀X.

  Theorem incl_sup L : ∀ X Y : ℘(L), X ⊆ Y → ⋁X ≤ ⋁Y.
  Proof.
  apply ( duality_principle incl_inf ).
  Qed.

End Inclusion.
```

Arrêtons-nous un instant sur cette preuve par dualité. Le type de l'expression qu'on utilise pour la preuve de `incl_sup`, i.e. `(duality_principle incl_inf)`, est

$$\forall\ \mathtt{L} : \mathtt{QuasiCompleteLattice},\ \forall\ \mathtt{X\ Y} : \wp(\mathtt{dual\ L}),\ \mathtt{X} \subseteq \mathtt{Y} \rightarrow \bigwedge^{\mathrm{d}}\mathtt{Y} \leq^{\mathrm{d}} \bigwedge^{\mathrm{d}}\mathtt{X}$$

C'est-à-dire, par définition des opérateurs du treillis dual, l'expression est de type :

$$\forall\ \mathtt{L} : \mathtt{QuasiCompleteLattice},\ \forall\ \mathtt{X\ Y} : \wp(\mathtt{L}),\ \mathtt{X} \subseteq \mathtt{Y} \rightarrow \bigvee\mathtt{Y} \geq \bigvee\mathtt{X}$$

D'où par définition de $\geq$:

$$\forall\ \mathtt{L} : \mathtt{QuasiCompleteLattice},\ \forall\ \mathtt{X\ Y} : \wp(\mathtt{L}),\ \mathtt{X} \subseteq \mathtt{Y} \rightarrow \bigvee\mathtt{X} \leq \bigvee\mathtt{Y}$$

D'où la nécessité d'utiliser des règles de réduction (cf. sec. 4.1, p.49), donc de calcul, lors de la preuve de théorèmes. Il n'y a donc pas indépendance entre calcul et raisonnement — s'il était nécessaire de s'en convaincre.

D'un préordre, on peut donner naissance à une relation d'équivalence. Nous utiliserons cette relation d'équivalence $\simeq$ pour exprimer l'égalité entre deux éléments de notre structure de quasi treillis complet, même si elle est en réalité moins forte que l'égalité de Coq (cf. 4.3, p.51).

```
Definition simeq L (x y : L) := x ≤ y ∧ y ≤ x.

Notation "x ≃ y" := (simeq _ x y)
        (at level 70, no associativity) : qclattice_scope.
Notation "x '≃ᵈ' y" := (simeq (dual _) x y)
        (at level 70, no associativity) : qclattice_scope.

Section simeq_equivalence.

  Variable L : QuasiCompleteLattice.

  Theorem simeq_refl : ∀ x : L, x ≃ x.
  Theorem simeq_trans : ∀ x y z : L, x ≃ y → y ≃ z → x ≃ z.
  Theorem simeq_sym : ∀ x y : L, x ≃ y → y ≃ x.

End simeq_equivalence.
```

Pour terminer, quelques lemmes techniques évidents, et qui seront utilisés lors des preuves. Soit pour faciliter leur suivi, ou pour préparer un but à l'application du principe de dualité.

```
Section simeq_and_duality.

  Variable L : QuasiCompleteLattice.
  Variables x y : L.

  Lemma simeq_dual_to_primal : x ≃ᵈ y → x ≃ y.
  Lemma simeq_primal_to_dual : x ≃ y → x ≃ᵈ y.

End simeq_and_duality.
```

```
Section simeq_and_le.

  Variable L : QuasiCompleteLattice.
  Variables x y z : L.

  Lemma le_simeq_right : x ≤ y → y ≃ z → x ≤ z.
  Lemma le_simeq_left : x ≤ y → z ≃ x → z ≤ y.

End simeq_and_le.
```

3.4 Théorèmes de point fixe de Tarski

Les résultats développés ici sont la traduction de l'article de Tarski [20]. On va montrer que dans tout treillis complet, toute fonction croissante possède au moins un point fixe. La réciproque est démontrée dans [9].

```
Require Import Sets.
Require Export QuasiCompleteLattices.

Open Local Scope qclattice_scope.
Implicit Type L : QuasiCompleteLattice.
```

Pour toute fonction, on peut définir l'ensemble de ses points fixes.

```
Definition fixpts L (δ : L → L) := {x ; δ(x) ≃ x }.
Implicit Arguments fixpts [L].
```

On dira que x est un pré-point fixe d'une fonction δ si $\delta(x) \leq x$, et que c'est un post-point fixe si $x \leq \delta(x)$. Le plus petit pré-point fixe d'une fonction croissante est exactement le plus petit point fixe de cette fonction. Et de façon duale, le plus grand post-point fixe est son plus grand point fixe. Commençons par définir ces éléments, et associons-leur une notation, celle utilisée dans [6].

```
Definition inf_pre_fixpts L (δ : L → L) := ⋀{ x ; δ(x) ≤ x }.
Definition sup_post_fixpts L (δ : L → L) := ⋁{ x ; δ(x) ≥ x }.
Implicit Arguments inf_pre_fixpts [L].
Implicit Arguments sup_post_fixpts [L].

Notation "'fixp' _, t" := ( inf_pre_fixpts (fun _ => t) )
        (at level 65, right associativity) : qclattice_scope.
Notation "'fixp' x : L , t" := ( inf_pre_fixpts (fun x : L => t) )
        (at level 65, x ident, right associativity) : qclattice_scope.
Notation "'fixp' x , t" := ( inf_pre_fixpts (fun x => t) )
        (at level 65, x ident, right associativity) : qclattice_scope.

Notation "'Fixp' _, t" := ( sup_post_fixpts (fun _ => t) )
        (at level 65, right associativity) : qclattice_scope.
Notation "'Fixp' x : L , t" := ( sup_post_fixpts (fun x : L => t) )
        (at level 65, x ident, right associativity) : qclattice_scope.
Notation "'Fixp' x , t" := ( sup_post_fixpts (fun x => t) )
        (at level 65, x ident, right associativity) : qclattice_scope.
```

Remarque: Une objection légitime peut être faite à propos de ces notations : pourquoi n'avoir pas simplement défini `fixp` comme synonyme de `inf_pre_fixpts` ?

Tout simplement parce que nous aurons besoin d'écrire `fixp x, fixp y, θ(x,y)` et qu'il faut donc lier les occurrences libres de `x` et de `y` dans `θ(x,y)`. C'est en fait le même principe que pour $\forall$.

Nous allons prouver que le plus petit pré-point fixe d'une fonction croissante est un point fixe. Et nous serons alors capable d'affirmer qu'il est égal au plus petit des points fixes de la fonction. La preuve de la première étape est originale puisqu'elle se divise en deux parties, la première étant utilisée dans la démonstration de la seconde. On va donc commencer par montrer que le plus petit pré-point fixe est un pré-point fixe, et utiliser ce fait pour montrer que c'est aussi un post-point fixe. Fixons notre cadre de travail.

```
Section least_fixed_point.

  Variable L : QuasiCompleteLattice.
  Variable δ : L → L.

  Hypothesis incr : ∀ x y : L, x ≤ y → δ(x) ≤ δ(y).
```

Les démonstrations seront effectuées à la manière de Coq, c'est-à-dire en partant du but, mais en langage naturel.

```
  Lemma fixp_pre_fixpt : δ(fixp x, δ(x)) ≤ fixp x, δ(x).
```

Démonstration.
Cela revient à montrer que δ`(fixp x, `δ`(x)) ≤ ⋀{x; `δ`(x) ≤ x}` par définition de `fixp`. Or `⋀{x; `δ`(x) ≤ x}` est le plus grand des minorants de `{x; `δ`(x) ≤ x}`. Il suffit donc de montrer que δ `(fixp x, `δ`(x))` est un minorant de cet ensemble. Par définition d'un minorant, on doit montrer `∀ x, `δ`(x) ≤ x →` δ`(fixp x, `δ`(x)) ≤ x`. Pour cela, prenons un élément quelconque `x : L`, supposons qu'il vérifie δ`(x) ≤ x` et montrons que δ`(fixp x, `δ`(x)) ≤ x`. Par transitivité de $\leq$, il suffit que δ `(fixp x, `δ`(x)) ≤ `δ`(x)` puisque δ`(x) ≤ x`. Par croissance de δ, il suffit que `fixp x, `δ`(x) ≤ x`, ce qui, par définition de `fixp` est équivalent à `⋀{x; `δ`(x) ≤ x} ≤ x`. Et comme `⋀{x; `δ`(x) ≤ x}` est un minorant de `{x; `δ`(x) ≤ x}`, il suffit que `x ∈ {x; `δ`(x) ≤ x}`, c'est-à-dire δ`(x) ≤ x`, ce qui est vérifié par hypothèse. □

On le voit, cette démonstration est plus détaillée qu'à l'habitude. Et sa présentation parfaitement bien adaptée à la preuve de ce lemme, puisqu'à chaque étape on utilise les seules propriétés applicables de la borne inférieure, ce qui nous conduit à une trivialité. Le prochain lemme est démontré de la même manière, mais sans la rhétorique mathématique : chaque ligne correspond à une étape de la preuve.

```
  Lemma fixp_post_fixpt : fixp x, δ(x) ≤ δ(fixp x, δ(x)).
```

Démonstration.

`⋀{x; `δ`(x) ≤ x} ≤ `δ`(fixp x, `δ`(x))`	(déf. `fixp`)
δ`(fixp x, `δ`(x)) ∈ {x; `δ`(x) ≤ x}`	(par `inf_lower`)
δ`(`δ`(fixp x, `δ`(x))) ≤ `δ`(fixp x, `δ`(x))`	(déf. `member`)
δ`(fixp x, `δ`(x)) ≤ fixp x, `δ`(x)`	(car δ croissante)
	(exactement `fixp_post_fixpt`)

□

D'où l'on déduit sans peine, d'après la définition de $\simeq$, le théorème suivant.

```
  Theorem fixp_fixpt : δ(fixp x, δ(x)) ≃ fixp x, δ(x).
```

À partir duquel on peut montrer qu'il s'agit bien du plus petit point fixe.

```
Notation P := (fixpts δ).

Theorem fixp_inf_fixpts : ⋀ P ≃ fixp x, δ(x).
```

Démonstration.

$\bigwedge$P $\leq$ fixp x, δ(x) et fixp x, δ(x) $\leq \bigwedge$P	(déf. $\simeq$)
fixp x, δ(x) $\in$ P et fixp x, δ(x) $\leq \bigwedge$P	(par inf_lower)
δ(fixp x, δ(x)) $\simeq$ fixp x, δ(x) et fixp x, δ(x) $\leq \bigwedge$P	(déf. de P)
fixp x, δ(x) $\leq \bigwedge$P	(exactement fixp_fixpt)
$\bigwedge${x; δ(x) $\leq$ x} $\leq \bigwedge$P	(déf. fixp)
P $\subseteq$ {x; δ(x) $\leq$ x}	(par incl_inf, cf. 3.3)
$\forall$ x, δ(x) $\simeq$ x $\rightarrow$ δ(x) $\leq$ x	
	(trivial par déf. $\simeq$)

□

```
End least_fixed_point.
```

La preuve pour le plus grand point fixe est duale de celle pour le plus petit point fixe. Et donc le déroulement de la preuve est très similaire à la preuve que l'on vient d'effectuer. D'ailleurs, il est possible d'utiliser le principe de dualité, ou plus directement d'appliquer les théorèmes précédents au treillis dual. Nous n'allons qu'énoncer les théorèmes.

```
Section greatest_fixed_point.

  Variable L : QuasiCompleteLattice.
  Variable γ : L → L.
  Implicit Type x y z : L.

  Hypothesis incr : ∀ x y, x ≤ y → γ(x) ≤ γ(y).
  Notation P := (fixpts γ).

  Theorem Fixp_fixpt : γ(Fixp x, γ(x)) ≃ Fixp x, γ(x).
  Theorem Fixp_sup_fixpts : ⋁ P ≃ Fixp x, γ(x).

End greatest_fixed_point.
```

Pour finir quelques lemmes techniques. Pour une fonction croissante, si un élément du treillis est égal au plus petit point fixe, c'est un point fixe. De même pour le plus grand point fixe. Si deux fonctions sont comparables, alors leurs plus grand et plus petit points fixes le sont également.

```
Section useful_lemmas.

  Variable L : QuasiCompleteLattice.
  Variable δ : L → L.
  Hypothesis incr : ∀ x y : L, x ≤ y → δ(x) ≤ δ(y).

  Lemma simeq_fixp_fixpt : ∀ x, x ≃ fixp x, δ(x) → δ(x) ≃ x.
  Lemma simeq_Fixp_fixpt : ∀ x, x ≃ Fixp x, δ(x) → δ(x) ≃ x.

End useful_lemmas.
```

```
Section useful_lemmas_2.

  Variable L : QuasiCompleteLattice.
  Variables δ γ : L → L.

  Hypothesis δ_le_γ : ∀x : L, δ(x) ≤ γ(x).

  Lemma fixp_le_fixp : fixp x, δ(x) ≤ fixp x, γ(x).
  Lemma Fixp_le_Fixp : Fixp x, δ(x) ≤ Fixp x, γ(x).

End useful_lemmas_2.

Section useful_lemmas_3.

  Variable L : QuasiCompleteLattice.
  Variables δ γ : L → L.

  Hypothesis δ_simeq_γ : ∀x : L, δ(x) ≃ γ(x).

  Lemma fixp_simeq_fixp : fixp x, δ(x) ≃ fixp x, γ(x).
  Lemma Fixp_simeq_Fixp : Fixp x, δ(x) ≃ Fixp x, γ(x).

End useful_lemmas_3.
```

3.5 Quelques applications

Faisons un petit intermède avant d'attaquer des démonstrations plus difficiles. Nous allons présenter quelques applications utilisant le cadre général que nous avons développé jusqu'à présent. Tout d'abord, nous allons montrer que la preuve de Knaster (cf. [15]) n'est qu'un cas particulier du théorème de Tarski (cf. [20]). Puis nous montrerons quelques théorèmes de points fixes, conséquences d'un théorème de Niwiński (cf. [17]). D'ailleurs dans ce cadre, nous utiliserons d'autres notations pour les plus petits et plus grands points fixes, respectivement μ et ν, issues du μ-calcul (cf. [1]).

3.5.1 Théorème de Knaster

Historiquement, le théorème de Knaster a été démontré avant celui de Tarski. Ce dernier en est une généralisation comme nous allons le voir. Le théorème de Knaster dit que toute fonction définie et à valeurs dans une famille de sous-ensembles d'un ensemble et qui est croissante pour l'inclusion, possède au moins un point fixe (cf. [15]). Une application bien connue de ce résultat est la démonstration du théorème de Cantor-Bernstein[7] dans le même article.

Il nous suffit simplement de montrer que l'ensemble des parties d'un ensemble forme un treillis. L'opérateur d'ordre est évidemment l'inclusion sur les ensembles, l'opérateur de borne inférieure est l'intersection d'une famille d'ensembles, et l'opérateur de borne supérieure l'union d'une famille d'ensembles. Les démonstrations sont triviales et ne nécessitent pas d'être explicitées.

```
Require Import Sets.
Require Import Fixpoints.

Section Knaster_fixed_points.

  Variable Ω : Type.
  Implicit Type X Y Z : ℘(Ω).

  Section preordered_set.
    Lemma subset_refl X : X ⊆ X.
    Lemma subset_trans X Y Z : X ⊆ Y → Y ⊆ Z → X ⊆ Z.
  End preordered_set.

  Section lattice.
    Variable S : ℘(℘(Ω)).

    Lemma cap_lower : ∀ X ∈ S, ⋂S ⊆ X.
    Lemma cap_greatest : ∀ X, (∀ Z ∈ S, X ⊆ Z) → X ⊆ ⋂S.
    Lemma cup_upper : ∀ X ∈ S, X ⊆ ⋃S.
    Lemma cup_least : ∀ X, (∀ Z ∈ S, Z ⊆ X) → ⋃S ⊆ X.
  End lattice.

  Definition subsets_lattice := Build_QuasiCompleteLattice (℘(Ω))
    (subset (Ω:=Ω)) (cap (Ω:=Ω)) (cup (Ω:=Ω)) subset_refl subset_trans
    cap_lower cap_greatest cup_upper cup_least.
```

[7] Une démonstration dans une version ancienne du Calcul des Constructions est proposée dans [14].

Une fois le treillis des sous-ensembles construit, nous pouvons obtenir les théorèmes de point fixe par simple application des théorèmes précédemment démontrés — les démonstrations sont données. Nous nous limitons à la démonstration d'inclusion mutuelle car l'extensionnalité n'est pas déclarée. Si c'était le cas, l'égalité des ensembles concernés se déduirait trivialement de ces inclusions.

Remarque: Quelques petites précisions à propos du code Coq précédent :

- il est possible d'imbriquer plusieurs sections ;
- la quantification universelle étant gérée comme l'est l'abstraction, les déclarations `subset_refl` et `subset_trans` sont des fonctions qui prennent respectivement un et trois arguments ;
- l'inclusion que nous avons appelé `subset` comporte un argument implicite qui peut être explicité si besoin par la construction `subset (Ω:=Ω)`.

```
  Section fixed_points.

  Variable δ : ℘(Ω) → ℘(Ω).
  Hypothesis incr : ∀ X Y, X ⊆ Y → δ(X) ⊆ δ(Y).

  Notation L := subsets_lattice.
  Open Local Scope qclattice_scope.

  Theorem least_pre : δ(fixp X : L, δ(X)) ⊆ fixp X : L, δ(X).
  Proof (fixp_pre_fixpt L δ incr).

  Theorem least_post : fixp X : L, δ(X) ⊆ δ(fixp X : L, δ(X)).
  Proof (fixp_post_fixpt L δ incr).

  Theorem greatest_post : Fixp X : L, δ(X) ⊆ δ(Fixp X : L, δ(X)).
  Proof (proj2 ( Fixp_fixpt L δ incr )).

  Theorem greatest_pre : δ(Fixp X : L, δ(X)) ⊆ Fixp X : L, δ(X).
  Proof ( proj1 ( Fixp_fixpt L δ incr )).

  End fixed_points.
End Knaster_fixed_points.
```

Remarque: La construction particulière `Proof (...)` consiste à fabriquer le λ-terme de preuve « à la main ». Il s'agit d'une simple application d'un théorème à des valeurs particulières. C'est tout à fait la pratique usuelle en langage naturel :

> La preuve s'effectue en appliquant le théorème `fixp_pre_fixpt` au treillis `L`, à la fonction δ croissante par `incr`.

Avec un bémol sur l'utilisation du théorème `Fixp_fixpt`, puisqu'il faut en plus en « projeter » la composante de la conjonction qui nous convient[8].

[8] On projette habituellement une composante d'un couple. C'est en fait ce qui est fait ici : en logique intuitionniste, la démonstration d'une conjonction $A \wedge B$ peut être vue comme la donnée d'une preuve de A et d'une preuve de B, c'est-à-dire un couple de preuves (cf. [13]).

3.5.2 Un théorème de Niwiński et ses conséquences

Les notations du μ-calcul sont plus condensées mais aussi moins explicites *a priori*. Elles sont exprimées et agissent comme un masquage des notations précédentes. Leur définition suit le même schéma et le même principe. On écrit $\mu x.f(x)$ pour désigner le plus petit point fixe de la fonction f, et $\nu x.f(x)$ pour son plus grand point fixe — à condition que f soit croissante dans un treillis donné.

```
Require Import Sets.
Require Import Fixpoints.

Notation "'μ' _, t" := ( inf_pre_fixpts (fun _ => t) )
        (at level 65, right associativity) : qclattice_scope.
Notation "'μ' x : L , t" := ( inf_pre_fixpts (fun x : L => t) )
        (at level 65, x ident, right associativity) : qclattice_scope.
Notation "'μ' x , t" := ( inf_pre_fixpts (fun x => t) )
        (at level 65, x ident, right associativity) : qclattice_scope.

Notation "'ν' _, t" := ( sup_post_fixpts (fun _ => t) )
        (at level 65, right associativity) : qclattice_scope.
Notation "'ν' x : L , t" := ( sup_post_fixpts (fun x : L => t) )
        (at level 65, x ident, right associativity) : qclattice_scope.
Notation "'ν' x , t" := ( sup_post_fixpts (fun x => t) )
        (at level 65, x ident, right associativity) : qclattice_scope.
```

Le théorème en question lie les plus petits points fixes des compositions de deux fonctions croissantes. Il est divisé en deux lemmes, chacun dans une section différente.

```
Require Import Sets.
Require Import Fixpoints.
Require Import MuCalculusNotations.

Open Local Scope qclattice_scope.

Section composition_increasing_functions.

  Variables L₁ L₂ : QuasiCompleteLattice.
  Variables (α : L₁ → L₂) (β : L₂ → L₁).

  Hypothesis incr_α : ∀x y, x ≤ y → α(x) ≤ α(y).
  Hypothesis incr_β : ∀x y, x ≤ y → β(x) ≤ β(y).

  Lemma incr_αβ : ∀x y : L₂, x ≤ y → α∘β(x) ≤ α∘β(y).
  Lemma comp_post_fixpt : μx, β∘α(x) ≤ β( μx, α∘β(x) ).
```

Démonstration.

$\bigwedge${x; β∘α(x) ≤ x} ≤ β(μx, α∘β(x))	(déf. de `inf_pre_fixpts`)
β ∘ α (β (μx, α∘β(x))) ≤ β(μx, α∘β(x))	(par `inf_lower`)
α (β (μx, α(β(x)))) ≤ μx, α(β(x))	(par `incr_β`)
∀ x y, x ≤ y → α∘β(x) ≤ α∘β(y)	(par `fixp_pre_fixpt` sur $\alpha \circ \beta$)
	(exactement `incr_αβ`)

□

Une autre section est ouverte avec des déclarations introductives identiques, pour pouvoir profiter de la décharge des variables et des hypothèses en fin de section. En effet, on va utiliser le lemme que l'on vient de démontrer deux fois : une fois sur α puis β, et une autre fois sur β puis α. Nous ne serons alors pas obligé de démontrer en plus et dans la même section μx, $\alpha\circ\beta$(x) $\leq$ α(μx, $\beta\circ\alpha$(x)). C'est un inconvénient de l'utilisation du mécanisme des sections. Il s'agit en fait d'une obligation d'expliciter le cadre de travail à chaque étape, ce qui n'est en général pas le cas en langage naturel, où l'on compte davantage sur la compréhension du lecteur.

```
End composition_increasing_functions.

Section composition_increasing_functions_bis.

  Variables L1 L2 : QuasiCompleteLattice.
  Variables (α : L1 → L2) (β : L2 → L1).

  Hypothesis incr_α : ∀x y, x ≤ y → α(x) ≤ α(y).
  Hypothesis incr_β : ∀x y, x ≤ y → β(x) ≤ β(y).

  Let incr_αβ := incr_αβ _ _ α β incr_α incr_β.

  Theorem comp_fixpt : μx, α∘β(x) ≃ α( μx, β∘α(x) ).
```

Démonstration.

μx, $\alpha\circ\beta$(x) $\leq$ α(μx, $\beta\circ\alpha$(x))	et	(déf. `simeq`)
α (μx, $\beta\circ\alpha$(x)) $\leq$ μx, $\alpha\circ\beta$(x)		
α (μx, $\beta\circ\alpha$(x)) $\leq$ μx, $\alpha\circ\beta$(x)		(par `comp_post_fixpt` sur β et α)
α (μx, $\beta\circ\alpha$(x)) $\leq$ $\alpha\circ\beta$(μx, $\alpha\circ\beta$(x))		(par `fixp_pre_fixpt`)
μx, β(α(x)) $\leq$ β(μx, α(β(x)))		(par `incr_α`)
		(par `comp_post_fixpt` sur α et β)

□

```
End composition_increasing_functions_bis.
```

Nous allons maintenant montrer que $\forall n > 0, \mu x, \delta^n(x) \simeq \mu x, \delta(x)$, si δ est croissante. Pour cela, nous allons passer par quelques lemmes intermédiaires, et la définition de la puissance d'une fonction. Cette définition est très proche de la version CaML[9].

```
Fixpoint pow (E : Type) (f : E → E) (m : nat) {struct m} : E → E :=
  match m with
  | 0 => (fun x => x)
  | S p => f ∘ (f ^ p)
  end
where "f ^ p" := (pow _ f p) : type_scope.

Implicit Arguments pow [E].
```

[9] Dans [18], l'auteur présente en introduction la syntaxe de Coq en comparaison avec celle de CaML.

Remarque:

- lors de la définition de la fonction pow, il faut annoncer à Coq sur quelle variable s'effectue la récurrence ({struct m}) ;
- on peut définir une notation qu'on utilise pendant une définition, ce qui est souvent le cas en mathématiques.

```
Section commutative_functions.

  Variables L : QuasiCompleteLattice.
  Variables (α : L → L) (β : L → L).

  Hypothesis incr_α : ∀x y, x ≤ y → α(x) ≤ α(y).
  Hypothesis incr_β : ∀x y, x ≤ y → β(x) ≤ β(y).
  Hypothesis comm : ∀ x, α∘β(x) ≃ β∘α(x).

  Lemma fixp_comm : μx, α∘β(x) ≃ μx, β∘α(x).
  Lemma fixp_pow : ∀ n : nat, (α^n)( μx, α∘β(x) ) ≃ μx, α∘β(x).
```

Démonstration. Par récurrence sur n. Le cas de base étant trivial par définition de pow, intéressons-nous à l'étape de récurrence.

```
(α^(S n)) (μx, α∘β(x)) ≃ μx, α∘β(x)
α ((α^n) (μx, α∘β(x))) ≃ μx, α∘β(x)      (déf. pow et comp)
α (μx, α∘β(x)) ≃ μx, α∘β(x)              (par H.R.)
α (μx, β∘α(x)) ≃ μx, α∘β(x)              (par fixp_comm)
                                         (par comp_fixpt)
```

□

```
End commutative_functions.
```

Nous pouvons alors attaquer la démonstration du théorème

$$\forall n > 0, \mu x.\delta^n(x) \simeq \mu x.\delta(x)$$

Mais nous allons reformuler cet énoncé pour pouvoir le démontrer plus facilement par la récurrence sur les entiers de Coq. Il s'agit alors de montrer

$$\forall n, \mu x.\delta^{n+1}(x) \simeq \mu x.\delta(x)$$

En effet, sans cela il faudrait utiliser un principe de récurrence différent. Par exemple, pour démontrer une propriété P pour tous les entiers naturels à partir d'un certain rang m, on procède habituellement de la façon suivante :

$$\frac{P(m) \qquad \forall n \geq m, P(n) \to P(n+1)}{\forall n \geq m, P(n)}$$

De plus il est possible de construire une preuve π' de la première formulation à partir d'une preuve π de la seconde, informellement

$$\pi' := \lambda n \lambda H^{n>0}.\pi(n-1)$$

```
Section pow_increasing_function.

  Variable L : QuasiCompleteLattice.
  Variable δ : L → L.

  Hypothesis incr : ∀ x y, x ≤ y → δ(x) ≤ δ(y).

  Lemma incr_pow : ∀ n : nat, ∀ x y, x ≤ y → (δ^n)(x) ≤ (δ^n)(y).
  Lemma pow_comm : ∀ n : nat, ∀ x, δ∘(δ^n)(x) ≃ (δ^n)∘δ(x).

  Theorem pow_fixp : ∀ n : nat, μx, (δ^(n+1))(x) ≃ μx, δ(x).
```

Démonstration. Par récurrence sur `n`. Le cas de base étant trivial par définition de `pow`, intéressons-nous à l'étape de récurrence : `μx, (δ^(S(n)+1))(x) ≃ μx, δ(x)`. Par définition de ≃, le but se décompose en deux parties indépendantes que nous allons montrer à la suite. Commençons par la première.

`μx, (δ^(S(n)+1))(x) ≤ μx, δ(x)`	(par déf. ≃)
`(δ^(S(n)+1)) (μx, δ(x)) ≤ μx, δ(x)`	(par `inf_lower`)
`(δ^(S(n)+1)) (μx, δ(x)) ≤ δ( μx, δ(x) )`	(par `fixp_pre_fixpt`)
`(δ^(n+1)) (μx, δ(x)) ≤ μx, δ(x)`	(par `incr`)
`(δ^(n+1)) (μx, δ(x)) ≃ μx, δ(x)`	(par `le_simeq_right`)
`μx, δ(x) ≃ μx, (δ^(n+1))(x)`	(par `simeq_fixp_fixpt`)
	(par H.R.)

La deuxième partie quand à elle se résout par utilisation du lemme précédent `fixp_pow`.

`μx, δ(x) ≤ μx, (δ^(S(n)+1))(x)`	(par déf. ≃)
`δ (μx, (δ^(S(n)+1))(x)) ≤ μx, (δ^(S(n)+1))(x)`	(par `inf_lower`)
`δ (μx, (δ^(S(n)+1))(x)) ≃ μx, (δ^(S(n)+1))(x)`	(par `le_simeq_right`)
`δ (μx, δ(δ^(n+1))(x)) ≃ μx, δ(δ^(n+1))(x)`	(déf. `comp`)
	(par `fixp_pow` sur δ et `δ^(n+1)`)

□

```
End pow_increasing_function.
```

3.6 Points fixes diagonaux de Niwiński et Colson

Jusqu'à présent, nous avons manipulé des points fixes de fonctions à un argument. Dans cette section, nous nous intéressons aux points fixes diagonaux d'une fonction binaire (i.e. à deux arguments). La notation développée précédemment pour les points fixes nous permet des écritures de la forme `fixp x, fixp y, θ(x,y)`, ou encore `fixp x, Fixp y, θ(x,y)`. Par définition, `fixp x, Fixp y, θ(x,y)` est le raccourci pour $\bigwedge$`{x; Fixp y, θ(x,y) ≤ x}`, c'est-à-dire $\bigwedge$`{x; ⋁{y; y ≤ θ(x,y)} ≤ x}`. Ces définitions sont bien formées, bien typées et ont un sens comme on va le voir.

Dans cette section, nous formalisons l'article de Colson [6]. L'article introduit tout d'abord des théorèmes de Niwiński [17], qu'il généralise en apportant des conditions suffisantes à la démonstration de l'inégalité symétrique du théorème `minimax`.

Définissons la croissance d'une fonction binaire, ainsi que quelques lemmes triviaux mais utiles.

```
Require Import Sets.
Require Import Fixpoints.

Open Local Scope qclattice_scope.

Section diagonal_fixpts_preliminary.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.

  Lemma incr_dual :
    ∀ x y x' y' : dual L, x ≤^d x' → y ≤^d y' → θ x y ≤^d θ x' y'.
  Lemma incr_fst : ∀ y x x', x ≤ x' → θ x y ≤ θ x' y.
  Lemma incr_snd : ∀ x y y', y ≤ y' → θ x y ≤ θ x y'.
  Lemma incr_fixp_fst : ∀ x x', x ≤ x' → fixp y, θ x y ≤ fixp y, θ x' y.
  Lemma incr_fixp_snd : ∀ y y', y ≤ y' → fixp x, θ x y ≤ fixp x, θ x y'.
  Lemma incr_simeq : ∀ x y x' y', x ≃ x' → y ≃ y' → θ x y ≃ θ x' y'.

End diagonal_fixpts_preliminary.

Section diagonal_fixpts_preliminary_bis.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.

  Lemma incr_Fixp_fst : ∀ x x', x ≤ x' → Fixp y, θ x y ≤ Fixp y, θ x' y.
  Lemma incr_Fixp_snd : ∀ y y', y ≤ y' → Fixp x, θ x y ≤ Fixp x, θ x y'.

End diagonal_fixpts_preliminary_bis.
```

Nous pouvons attaquer la démonstration du théorème simple `minimin` qui affirme que `fixp x, fixp y, θ x y` est le plus petit point fixe diagonal de la fonction θ. Par la même occasion nous définissons précisément ce qu'est un point fixe diagonal d'une fonction binaire en définissant l'ensemble de ces points.

```
Section minimin.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.

  Definition diag_fixpts := { x ; θ x x ≃ x }.

  Theorem minimin : fixp z, θ z z ≃ fixp x, fixp y, θ x y.
```

Démonstration. Comme à l'habitude lors d'une preuve impliquant $\simeq$, la démonstration se divise en deux parties. On notera par x_0 le terme `fixp x, fixp y, θ x y`.

`fixp z, θ z z` $\leq x_0$	(déf. de $\simeq$)
$\theta\ x_0\ x_0 \leq x_0$	(par `inf_lower`)
$\theta\ x_0\ x_0 \simeq x_0$	(par `le_simeq_right`)
$x_0 \simeq$ `fixp y, θ` x_0 `y`	(par `simeq_fixp_fixpt`)
$x_0 \simeq$ `fixp x, fixp y, θ x y`	(par `simeq_fixp_fixpt`)
	(trivial)

Puis, en utilisant la définition du plus grand minorant :

$x_0 \leq$ `fixp z, θ z z`	(déf. de $\simeq$)
$\forall$ `x,` θ `x x` $\leq$ `x` $\rightarrow x_0 \leq$ `x`	(par `inf_greatest`)
$x_0 \leq$ `x`	(ajout d'hypothèses)
`fixp y,` θ `x y` $\leq$ `x`	(par `inf_lower`)
θ `x x` $\leq$ `x`	(par `inf_lower`)
	(par hypothèse)

□

Et on peut alors déduire le théorème `maximax` par dualité sur `minimin`, et un corollaire utilisant deux fois l'application de `minimin`. Les scripts des preuves par dualité, très simples, sont donnés.

```
End minimin.

Section maximax.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.

  Theorem maximax : Fixp z, θ z z ≃ Fixp x, Fixp y, θ x y.
  Proof.
  apply simeq_dual_to_primal.
  apply ( minimin (dual L) θ (incr_dual L θ incr) ).
  Qed.

End maximax.
```

D'où l'on déduit le théorème suivant, qui nous autorise à permuter les opérations de plus petit point fixe. Il suffit d'appliquer `minimin` à la fonction **fun** x y => θ x y et à la fonction **fun** x y => θ y x. On en déduit le théorème dual nous permettant de permuter les opérations de plus grand point fixe.

```
Section minimin_maximax_corollary.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.

  Theorem fixp_comm : fixp x, fixp y, θ x y ≃ fixp y, fixp x, θ x y.

End minimin_maximax_corollary.

Section minimin_maximax_corollary_bis.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.

  Theorem Fixp_comm : Fixp x, Fixp y, θ x y ≃ Fixp y, Fixp x, θ x y.
  Proof.
  apply simeq_dual_to_primal.
  apply ( fixp_comm (dual L) θ ).
  apply incr_dual; trivial.
  Qed.

End minimin_maximax_corollary_bis.
```

Définissons et montrons que `minimax` et `maximin` sont des points fixes diagonaux. Une application de `simeq_Fixp_fixpt` et une application de `simeq_fixp_fixpt` suffisent pour `minimax_fixpt`, et `maximin_fixpt` se déduit par dualité.

```
Section minimax_fixpt.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.

  Notation x0 := ( fixp x, Fixp y, θx y ).

  Theorem minimax_fixpt : θ x0 x0 ≃ x0.

End minimax_fixpt.

Section maximin_fixpt.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.
```

```
  Notation X0 := ( Fixp x, fixp y, θx y ).

  Theorem maximin_fixpt : θ X0 X0 ≃ X0.
  Proof.
  apply simeq_dual_to_primal.
  apply ( minimax_fixpt (dual L) θ ).
  apply incr_dual; trivial.
  Qed.

End maximin_fixpt.
```

Un théorème important et non trivial lie les termes `minimax` et `maximin`. Nous appellerons aussi ce théorème `minimax`, le contexte nous aidera à savoir s'il s'agit de la désignation du terme ou du théorème.

```
Section minimax_theorem.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.

  Theorem minimax : fixp x, Fixp y, θ x y ≤ Fixp y, fixp x, θ x y.
```

Démonstration.
Nous noterons par x_0 le terme `fixp x, Fixp y, θ x y`.

$x_0 \leq$ `fixp x, θ x` x_0	(par `sup_upper`)
$x_0 \simeq$ `fixp x, θ x` x_0	(par `le_simeq_left`)

Nous avons commencé par montrer `Fixp y, θ` x_0 `y` $\simeq x_0$, que nous réutiliserons sous le nom `eq1`. En effet, il suffit d'appliquer le lemme technique `simeq_fixp_fixpt` avec la croissance de la fonction donnée par le lemme `incr_Fixp_fst`.

Par transitivité de $\simeq$ sur `fixp x, θ x (Fixp y, θ` x_0 `y)`, nous obtenons deux sous-buts que nous allons résoudre à la suite.

`fixp x, θ x` $x_0 \simeq$ `fixp x, θ x (Fixp y, θ` x_0 `y)`	(par transitivité)
`∀ x, θ x` $x_0 \simeq$ `θ x (Fixp y, θ` x_0 `y)`	(par `fixp_simeq_fixp`)
	(par `incr_simeq` et `eq1`)

Et l'autre partie de la transitivité, où la démonstration de la croissance de la fonction **`fun`** `z => fixpx, θ x (Fixpy, θ z y)` s'effectue par application des lemmes `fixp_le_fixp` et `incr_Fixp_fst` :

`fixp x, θ x (Fixp y, θ` x_0 `y)` $\simeq x_0$	(par transitivité)
$x_0 \simeq$ `fixp x', fixp x, θ x (Fixp y, θ x' y)`	(par `simeq_fixp_fixpt`)

Puis, une application de la transitivité sur `fixpx, θ x (Fixpy, θ x y)`, dont le second but est résolu par application du théorème `minimin` sur la fonction croissante suivante **`fun`** `x' x => θ x (Fixp y, θ x' y)`, nous donne :

`fixp x, θ x (Fixp y, θ x y)` $\simeq x_0$	(par transitivité)
`∀ x, θ x (Fixp y, θ x y)` $\simeq$ `Fixp y, θ x y`	(par `fixp_simeq_fixp`)
	(par `Fixp_fixpt` et `incr_snd`)

□

Remarque: La démonstration n'est certes pas triviale, mais peut être exprimée en quelques lignes « à la main » tout en restant compréhensible (cf. [6]). Ce n'est pas le cas avec Coq, et ce pour plusieurs raisons :

- La démonstration du théorème utilise une succession d'égalités, avec Coq cela se traduit par des utilisations explicites de la transitivité.
- Des égalités sont démontrées et utilisées plusieurs fois dans la preuve. Avec Coq cela nécessite soit de démontrer plusieurs fois la même chose, soit d'extraire un lemme (c'est possible à l'intérieur d'une preuve).
- Toutes les croissances des fonctions, passées sous silence lorsque la démonstration est manuscrite, sont nécessaires pour terminer la preuve avec Coq.

Cela remet en question la recherche de preuve avec un tel outil : si la preuve n'était pas déjà connue, la trouver aurait certainement été bien plus pénible, et le script de la démonstration certainement bien plus long et plus complexe à suivre.

```
End minimax_theorem.
```

Pour montrer la réciproque de `minimax`, il nous faut quelques lemmes préliminaires.

```
Section maximin_preliminary.

  Variable L : QuasiCompleteLattice.
  Variables δ γ : L →L.

  Hypothesis incr_δ : ∀x x', x ≤ x' → δ(x) ≤ δ(x').
  Hypothesis incr_γ : ∀x x', x ≤ x' → γ(x) ≤ γ(x').
  Hypothesis pre_comm : ∀ x, δ∘γ(x) ≤ γ∘δ(x).

  Lemma fixp_le_Fixp : fixp x, δ(x) ≤ Fixp y, γ(y).
```

Démonstration.

`fixp x, δ(x) ≤ γ(fixp x, δ(x))`	(par `sup_upper`)
`δ(γ(fixp x, δ(x))) ≤ γ(fixp x, δ(x))`	(par `inf_lower`)
`γ∘δ(fixp x, δ(x)) ≤ γ(fixp x, δ(x))`	(par `pre_comm`)
`δ(fixp x, δ(x)) ≤fixp x, δ(x)`	(par `incr_γ`)
	(par `fixp_pre_fixpt`)

□

Un plus petit point fixe d'une constante est égal à cette constante — de même pour le plus grand point fixe par dualité. La preuve s'effectue par définition de `inf_pre_fixpts`, c'est-à-dire en appliquant `inf_lower` et `inf_greatest`.

```
  Lemma fixp_const : ∀ a : L, fixp x, a ≃ a.
  Lemma Fixp_const : ∀ a : L, Fixp x, a ≃ a.

End maximin_preliminary.
```

`maximin` peut être obtenu lorsque la fonction binaire considérée est croissante et associative à droite.

```
Section maximin.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.
  Hypothesis right_assoc : ∀ x y z, θ (θ x y) z ≤ θ x (θ y z).

  Theorem maximin : Fixp y, fixp x, θ x y ≤ fixp x, Fixp y, θ x y.
```

Démonstration.

`fixp _,Fixp y,fixp x,θ x y ≤ fixp x,Fixp y,θ x y`	(`le_simeq_left`, `fixp_const`)
`∀ x, Fixp y, fixp x, θ x y ≤ Fixp y, θ x y`	(`fixp_le_fixp`)
`Fixp y, fixp x, θ x y ≤ Fixp _, Fixp y, θ x y`	(`le_simeq_right`, `Fixp_const`)
`∀ y, fixp x, θ x y ≤ Fixp y, θ x y`	(par `Fixp_le_Fixp`)
`∀ y, θ (θ x y) z ≤ θ x (θ y z)`	(par `fixp_le_Fixp`, `incr_fst`, `incr_snd`)
	(par `right_assoc`)

□

Et donc dans ces conditions, on peut déduire l'égalité suivante :

```
  Theorem minimaximin : fixp x, Fixp y, θ x y ≃ Fixp y, fixp x, θ x y.

End maximin.
```

En fait, il est possible d'avoir cette précédente égalité avec une condition moins forte que l'associativité droite de θ. Il s'agit de la quasi associativité droite.

```
Section maximin_improvement.

  Variable L : QuasiCompleteLattice.
  Variable θ : L → L → L.
  Hypothesis incr : ∀ x y x' y', x ≤ x' → y ≤ y' → θ x y ≤ θ x' y'.

  Notation w₀ := (fixp x, Fixp y, θx y).
  Notation w₀' := (Fixp y, fixp x, θx y).
  Hypothesis quasi_right_assoc : ∀ z, θ (θ w₀ z) w₀' ≤ θ w₀ (θ z w₀').

  Theorem weak_minimaximin : w₀ ≃ w₀'.
```

Démonstration. La preuve utilise aussi `fixp_le_Fixp`. Elle est plus courte que la précédente, mais similaire (voir [6]). □

3.7 Théorème de Tarski généralisé

Dans son article, Tarski a aussi démontré une généralisation de son théorème de point fixe d'une fonction croissante : les plus petits et plus grands points fixes communs d'une famille commutante de fonctions croissantes. Les démonstrations sont similaires et ne seront pas données, à l'exception d'une seule. Nous allons donc nous contenter d'énoncer les définitions et les théorèmes (voir [20] pour les démonstrations en langage naturel).

```
Require Import Sets.
Require Import Fixpoints.

Open Local Scope qclattice_scope.

Implicit Type L : QuasiCompleteLattice.

Definition cfixpts L (F : ℘(L → L)) := {x ; ∀ δ ∈ F, δ(x) ≃ x }.
Implicit Arguments cfixpts [L].

Definition inf_pre_cfixpts L (F : ℘(L → L)) :=
        ⋀ { x ; ∀ δ ∈ F, δ(x) ≤ x }.
Definition sup_post_cfixpts L (F : ℘(L → L)) :=
        ⋁ { x ; ∀ δ ∈ F, δ(x) ≥ x }.
Implicit Arguments inf_pre_cfixpts [L].
Implicit Arguments sup_post_cfixpts [L].

Notation "'cfixp' F" := ( inf_pre_cfixpts F )
        (at level 65, x ident, right associativity) : qclattice_scope.
Notation "'cFixp' F" := ( sup_post_cfixpts F )
        (at level 65, x ident, right associativity) : qclattice_scope.


Section least_common_fixed_point.

  Variable L : QuasiCompleteLattice.
  Variable F : ℘(L → L).
  Hypothesis comm : ∀ δ γ ∈ F, ∀x, δ(γ(x)) ≃ γ(δ(x)).
  Hypothesis incr : ∀ δ ∈ F, ∀x y, x ≤ y → δ(x) ≤ δ(y).

  Notation P := (cfixpts F).

  Lemma cfixp_pre_cfixpt : ∀ δ ∈ F, δ(cfixp F) ≤ cfixp F.
  Lemma cfixp_post_cfixpt : ∀ δ ∈ F, cfixp F ≤ δ(cfixp F).
```

Démonstration.

`cfixp F` $\leq \delta$`(cfixp F)`	(ajout d'hypothèses)
$\forall\ \gamma \in$ `F,` $\gamma(\delta$`(cfixp F))` $\leq \delta$`(cfixp F)`	(par `inf_lower`)
$\gamma(\delta$`(cfixp F))` $\leq \delta$`(cfixp F)`	(ajout d'hypothèses)
$\delta(\gamma$`(cfixp F))` $\leq \delta$`(cfixp F)`	(par `comm` et `le_simeq_left` *)
γ`(cfixp F)` $\leq$ `cfixp F`	(par `incr`)
	(par `cfixp_pre_cfixpt`)

L'originalité de la preuve par rapport à la version spécialisée apparaît en * avec l'utilisation de l'hypothèse que les fonctions commutent, `comm`. Le reste du raisonnement est identique. □

```
  Theorem cfixp_cfixpt : ∀ δ ∈ F, δ(cfixp F) ≃ cfixp F.
  Theorem cfixp_inf_cfixpts : ⋀ P ≃ cfixp F.

End least_common_fixed_point.


Section greatest_common_fixed_point.

  Variable L : QuasiCompleteLattice.
  Variable F : ℘(L → L).
  Hypothesis comm : ∀ δ γ ∈ F, ∀x, δ(γ(x)) ≃ γ(δ(x)).
  Hypothesis incr : ∀ δ ∈ F, ∀x y, x ≤ y → δ(x) ≤ δ(y).

  Notation P := (cfixpts F).

  Theorem cFixp_cfixpt : ∀ δ ∈ F, δ(cFixp F) ≃ cFixp F.
  Theorem cFixp_sup_cfixpts : ⋁ P ≃ cFixp F.

End greatest_common_fixed_point.
```

On le voit, les théorèmes de la section précédente sont des cas particuliers pour une famille ne contenant qu'une unique fonction croissante, qui évidemment commute avec elle-même.

3.8 Nouvelle formalisation des ensembles

Dans son article, Tarski va encore plus loin, et il montre que l'ensemble des points fixes d'une fonction croissante dans un treillis complet forme lui-même un treillis complet. C'est-à-dire qu'étant donné un treillis complet L et une fonction croissante $\delta : L \to L$, le sous-ensemble $\{x; \delta(x) = x\}$ de L et des opérateurs construits à partir de ceux du treillis initial forment un treillis complet. Pour démontrer ce théorème en Coq, il nous est nécessaire de développer une notion proche de celle de sous-treillis, ou plus exactement, dont l'ensemble des éléments est un sous-ensemble du treillis initial.

Pour cela, nous avons deux possibilités : (i) redéfinir la structure `QuasiCompleteLattice` afin d'ajouter en plus du type de travail, l'ensemble de travail (ii) ou bien utiliser une nouvelle formalisation des ensembles en tant que type de données.

Supposons que nous ayons choisi la première solution. Nous aurions donc travaillé avec des structures de treillis complet avec cinq composantes $(\Omega, L, \leq, \bigwedge, \bigvee)$, où Ω est le type de travail et L un ensemble d'éléments de type Ω. Cela aurait alors nécessité une réécriture de tous les théorèmes en ajoutant l'hypothèse que δ est une application de L dans L, c'est-à-dire $\forall x \in L, \delta(x) \in L$. Et à chaque utilisation d'un théorème, ce fait aurait dû être démontré. C'est pourquoi nous avons préféré opter pour la seconde possibilité, qui ne rend pas obsolète le développement déjà effectué.

Jusqu'à présent, nous avons défini un ensemble comme un prédicat, par une fonction caractéristique de l'ensemble à formaliser. Pour savoir si un élément appartient à un ensemble, il suffit de lui appliquer ce prédicat et de vérifier le théorème résultant. Chaque fois que nous avons besoin de savoir si cet élément appartient à l'ensemble, il faut démontrer le même théorème. C'est pourquoi il est utile de conserver cette preuve associée à l'élément concerné. Nous obtenons alors des couples constitués d'un élément et de la preuve qu'il appartient bien à l'ensemble. Un tel couple est un terme d'un nouveau type, le type d'un ensemble[10].

Par exemple, en instanciant un élément de type `QuasiCompleteLattice` sur le treillis complet $(\Omega = \{x : T; P(x)\}, \leq, \bigwedge, \bigvee)$, nous manipulons des couples $(x, P(x))$ — donc des éléments de l'ensemble défini par le prédicat P — et toute fonction $\delta : L \to L$ associe un couple $(x, P(x))$ à un couple $(y, P(y))$ — donc associe un élément de l'ensemble décrit par P à un autre élément de cet ensemble.

Coq fournit une notation pour le type d'un ensemble décrit par le prédicat `P`, il s'agit de `{x : T | P }`, où `x` apparaît libre dans `P`. Il faudra alors faire attention à ne pas confondre les notations `{x : T ; P }` et `{x : T | P }`. La notation `{x : T | P }` est en fait un raccourci syntaxique pour le type dépendant `sig P`, type composé uniquement d'éléments de la forme `exist P x (π : P x)`.

Pour pouvoir manipuler ces couples et faire le lien avec notre précédente formalisation des ensembles, nous avons besoin de définir des notations pour les projections prédéfinies sur ces couples, et certains lemmes techniques. Dans la suite, nous parlerons du *type d'un ensemble* pour désigner un ensemble comme un type de la forme `{x : T | P }`, et d'*un ensemble* pour désigner un élément de type `℘(Ω)`.

[10] Il s'agit bien du type d'*un* seul ensemble, alors que dans le premier formalisme développé `℘(Ω)` désignait le type *des* ensembles d'éléments de type Ω.

Commençons pas définir les notations de ces couples particuliers, ainsi que la notation des projections des composantes de ces couples.

```
Require Import Sets.

Notation "⟨ x , y ⟩" := (exist _ x y)
        (at level 0, no associativity) : type_scope.
Notation "'π1' c" := (proj1_sig c)
        (at level 60, right associativity) : type_scope.
Notation "'π2' c" := (proj2_sig c)
        (at level 60, right associativity) : type_scope.
```

Afin de faire le lien avec notre formalisation précédente, nous étendons la projection de la première composante à un ensemble (selon le premier formalisme) de couples.

```
Definition Proj1_sig (A:Type) (P:A→Prop) (E : ℘(sig P)) :=
        { x : A ; ∃ c ∈ E, π1 c = x }.
Implicit Arguments Proj1_sig [A P].
Notation "'Π1' C" := (Proj1_sig C)
        (at level 55, right associativity) : type_scope.
```

Quelques lemmes assez techniques émergent alors. Rappelons que `A : ℘(sig P)` signifie que `A` est un ensemble d'éléments du type d'un ensemble, donc un ensemble de couples. Il est donc naturel de récupérer toutes les premières composantes des couples de `A`, ce qui s'obtient par la construction Π_1 `A`.

```
Section technical.

  Variable Ω : Type.
  Variable P : Ω → Prop.
  Variable A : ℘(sig P).

  Lemma member_proj : ∀ x, x ∈ A → π1 x ∈ Π1A.
  Lemma proj_Proj : ∀ Q : Ω → Prop, (∀ x ∈ A, Q (π1 x)) → ∀x ∈ Π1 A, Q x.
  Lemma pred_proj : ∀ x ∈ Π1 A, P x.

End technical.

Implicit Arguments member_proj [Ω P].
Implicit Arguments proj_Proj [Ω].
Implicit Arguments pred_proj [Ω P].
```

Intuitivement, les significations de ces lemmes sont :

- `member_proj` affirme que l'ensemble Π_1 `A` contient bien toutes les premières composantes des couples de `A` ;
- `proj_Proj` dit que tout prédicat vérifié par toutes les premières composantes des éléments de `A` est vérifié par tous les éléments de Π_1 `A` ;
- `pred_proj` n'est qu'un cas particulier de `proj_Proj` pour lequel le prédicat `Q` est instancié avec le prédicat `P` qui définit le type ensemble des éléments de `A`.

La réciproque de `member_proj` nécessite l'égalité sur les preuves (*proof irrelevance*, voir [21, 3]). En effet, il est possible d'après `pred_proj` de construire un terme `c` tel que π_1 `c = `π_1`x` et `c : sig P`, mais on a `c = x` seulement s'il est possible de montrer l'égalité des preuves contenues dans ces deux couples.

3.9 Treillis complet des points fixes

Pour montrer que l'ensemble des points fixes d'une fonction croissante dans un treillis complet forme un treillis complet (cf. [20]), il faut construire les opérateurs associés. On conserve l'opérateur d'ordre du treillis initial, mais les opérateurs de borne inférieure et supérieure ne peuvent pas être conservés. En effet, si nous prenons un ensemble de points fixes quelconque, sa borne inférieure dans le treillis initial n'est pas nécessairement un point fixe. Pour construire ces opérateurs, nous allons passer par des treillis d'intervalles.

Commençons par définir un intervalle dans un treillis complet donné.

```
Require Import Sets.
Require Import SigmaSets.
Require Import Fixpoints.

Open Local Scope qclattice_scope.

Implicit Type L : QuasiCompleteLattice.

Definition inter L (a b : L) := { x | a ≤ x ≤ b }.
Definition inter_pred L (a b : L) := fun x => a ≤ x ≤ b.
Implicit Arguments inter [L].
Implicit Arguments inter_pred [L].

Notation "[ a , b ]" := (inter a b)
        (at level 0, no associativity) : type_scope.
```

Nous avons défini le type intervalle ainsi que le prédicat qui le caractérise.

Nous allons alors montrer que si $(L, \leq, \bigwedge, \bigvee)$ est un treillis complet, alors pour $a, b \in L$ tels que $a \leq b$, $([a, b], \leq, \bigwedge, \bigvee)$ est un treillis complet. Intéressons-nous tout d'abord à la démonstration en langage naturel. Nous devons montrer que les opérateurs de bornes inférieure et supérieure ne sortent pas de l'intervalle, par exemple pour $\bigwedge$ on doit avoir

$$\forall A \subset [a, b], a \leq \bigwedge A \leq b$$

Démonstration.
D'après la définition de $[a, b]$, on a $\forall x \in [a, b], a \leq x \leq b$, et en particulier $\forall x \in A, a \leq x$. C'est-à-dire que a est un minorant de A, donc $a \leq \bigwedge A$.
Nous savons que $\forall x \in A, x \leq b$ par définition de $[a, b]$, et que $\forall x \in A, \bigwedge A \leq x$. Nous avons alors deux cas :

$A \neq \emptyset$: Soit $x \in A$, par transitivité, nous obtenons $\bigwedge A \leq x \leq b$ et donc $\bigwedge A \leq b$.

$A = \emptyset$: $\bigwedge \emptyset = \top$ sur le treillis complet initial[11], et en général, $\top \leq b$ n'est pas vérifié.

Cette preuve est donc un échec. La cause en est l'utilisation des opérateurs du treillis initial. En fait il ne faut pas les prendre tels quels, ou alors ne pas s'autoriser les opérateurs de borne inférieure et supérieure sur des ensembles vides. □

En conséquence, nous définissons l'opérateur de borne inférieure sur le treillis intervalle par $\bigwedge A \sqcap b$ et l'opérateur de borne supérieure par $\bigvee A \sqcup a$. Et nous allons montrer que les propriétés des bornes sont bien satisfaites.

[11] En effet, tous les éléments du treillis sont des minorants de l'ensemble vide. Et donc le plus grand des minorants, dénoté par $\bigwedge \emptyset$, est $\top$.

```
Section interval_lattice.

  Variable L : QuasiCompleteLattice.
  Variables a b : L.
  Hypothesis a_le_b : a ≤ b.

  Let I := [a , b].

  Implicit Types x y z : I.

  Lemma inf_betw : ∀ A : ℘(I), a ≤ ⋀(Π₁ A) ⊓ b ≤ b.
```

Démonstration. La démonstration du lemme se divise en deux parties.

$a \leq \bigwedge \Pi_1 A \sqcap b$	
$a \leq \bigwedge \Pi_1 A$	(par `inf_bin_greatest` et `a_le_b`)
$\forall x \in \Pi_1 A,\ a \leq x$	(par `inf_greatest`)
$a \leq x$	(ajout d'hypothèses)
$a \leq x \leq b$ et $a \leq x$	(sous-lemme)
$a \leq x$	(par `pred_proj`)
	(par sous-lemme)

Et par simple application de `inf_bin_lower_snd`, on démontre $\bigwedge \Pi_1 A \sqcap b \leq b$.

□

De même on démontre que l'opérateur de borne supérieure reste bien dans l'intervalle.

```
  Lemma sup_betw : ∀ A : ℘(I), a ≤ ⋁(Π₁ A) ⊔ a ≤ b.
```

On peut alors définir nos opérateurs, et des notations associées pour plus de facilité.

```
  Definition lei x y := π₁ x ≤ π₁ y.
  Definition infi A : I := ⟨ ⋀ (Π₁ A) ⊓ b , inf_betw A ⟩.
  Definition supi A : I := ⟨ ⋁ (Π₁ A) ⊔ a , sup_betw A ⟩.

  Infix "≤ᵢ" := lei (at level 70, no associativity).
  Notation "⋀ᵢ A" := (infi A) (at level 60, no associativity).
  Notation "⋁ᵢ A" := (supi A) (at level 60, no associativity).
```

On montre facilement que `lei` est réflexive et transitive.

```
  Lemma lei_refl : ∀ x, x ≤ᵢ x.
  Lemma lei_trans : ∀ x y z, x ≤ᵢ y → y ≤ᵢ z → x ≤ᵢ z.
```

Et on montre que `infi` est bien un opérateur de borne inférieure sur `[a, b]`.

```
  Lemma infi_lower : ∀ A, ∀ x ∈ A, (⋀ᵢ A) ≤ᵢ x.
```

Démonstration.

$\bigwedge_i A \leq_i x$	(ajout d'hypothèses)
$\bigwedge \Pi_1 A \sqcap b \leq \pi_1 x$	(par définitions)
$\bigwedge \Pi_1 A \leq \pi_1 x$	(par transitivité et `inf_bin_lower_fst`)
$\pi_1 x \in \Pi_1 A$	(par `inf_lower`)
	(par `member_proj`)

□

```
Lemma infi_greatest : ∀ A, ∀ z, (∀ x ∈ A, z ≤i x) → z ≤i ⋀iA.
```

Démonstration.

$z \leq_i \bigwedge_i A$	(ajout d'hypothèses)
$\pi_1 z \leq \bigwedge \Pi_1 A \sqcap b$	(par définitions)
$\pi_1 z \leq \bigwedge \Pi_1 A$	(par `inf_bin_greatest` et π_2 z)
$\forall x \in \Pi_1 A, \pi_1 z \leq x$	(par `inf_greatest`)
$\forall x \in A, \pi_1 z \leq \pi_1 x$	(par `proj_Proj`)
	(par hypothèse)

□

De même, on montre pour l'opérateur `supi`, et ainsi nous pouvons construire notre treillis d'intervalle.

```
  Lemma supi_upper : ∀ A, ∀ x ∈ A, x ≤i ⋁iA.
  Lemma supi_least : ∀ A, ∀ z, (∀ x ∈ A, x ≤i z) → ⋁iA ≤i z.

  Definition interval := Build_QuasiCompleteLattice I
        lei infi supi lei_refl lei_trans
        infi_lower infi_greatest supi_upper supi_least.

End interval_lattice.

Notation "( [ a , b ] ,  '≤' )" := (interval _ a b _)
        (at level 0, no associativity) : type_scope.
```

Cette dernière notation consiste à alléger l'environnement affiché par Coq au cours des preuves. Elle ne contient pas suffisamment d'informations pour pouvoir construire un treillis complet.

Maintenant que nous savons qu'un intervalle forme un treillis complet, il est aisé d'en déduire une construction des opérateurs de bornes sur l'ensemble des points fixes d'une fonction croissante δ donnée. Prenons un ensemble $Y \subset \{x; \delta(x) = x\}$. Pour construire sa borne supérieure telle qu'il s'agisse d'un point fixe de δ, il suffit de prendre le plus petit point fixe directement supérieur à $\bigvee Y$. Pour cela, on se place dans le treillis d'intervalle $[\bigvee Y, \top]$, et on y récupère le plus petit point fixe de la restriction de δ à cet intervalle. En effet, puisque $[\bigvee Y, \top]$ est un treillis complet, que δ y est croissante, alors elle possède au moins un point fixe, et l'on sait caractériser le plus petit de ces points fixes. Néanmoins, il ne faut pas négliger une condition importante ; l'ensemble des images de la restriction de δ doit appartenir à $[\bigvee Y, \top]$. C'est en substance le principe de la démonstration. Lors de la démonstration formalisée avec Coq, il faudra à chaque étape se poser la question de l'ensemble et du type de travail.

Nous allons définir le treillis des points fixes. Pour cela, il est nécessaire de passer par le type ensemble des points fixes d'une fonction croissante δ donnée. L'opérateur d'ordre est le plus simple à définir, puisqu'il s'agit d'appliquer l'ordre du treillis initial aux premières composantes des couples.

```
Section fixpoints_lattice.

  Variable L : QuasiCompleteLattice.
  Variable δ : L → L.
```

```
Hypothesis incr : ∀ x y, x ≤ y → δ(x) ≤ δ(y).

Let P := {x ; δ(x) ≃ x}.
Let P' := {x | δ(x) ≃ x}.

Definition lep (x y : P') := π1 x ≤ π1 y.
```

Pour définir les opérateurs de bornes, nous nous donnons un sous-ensemble de points fixes de δ, `A`, ainsi que sa projection dans les éléments du treillis originel, `Y`. Un lemme technique et nécessaire affirme que les éléments de `Y` sont bien des points fixes de δ. Il se démontre directement par utilisation du lemme technique `pred_proj`.

```
Section sup_and_inf.

  Variable A : ℘(P').

  Let Y := Π1 A.

  Lemma A_incl_P : Π1 A ⊆ P.
```

Nous commençons par définir la borne supérieure. Pour cela, nous allons devoir définir la restriction de δ à l'intervalle `[⋁Y, ⊤]`, c'est-à-dire déclarer une fonction `δ'` de type `[⋁Y, ⊤] →[⋁Y, ⊤]`, égale à δ sur cet intervalle. Mais alors, il faut montrer que δ est bien à valeurs dans cet intervalle, lorsque ses antécédents y sont. C'est d'ailleurs le coeur de la démonstration.

```
  Section sup.

    Let I := interval L (⋁ Y) (⊤) (all_le_top (⋁ Y)).

    Lemma restr_betw_sup : ∀ x : L, ⋁Y ≤ x ≤ ⊤→ ⋁Y ≤ δ(x) ≤ ⊤.
```

Démonstration.

$\bigvee Y \leq \delta(x) \leq \top$	(ajout d'hypothèses)
$\bigvee Y \leq \delta(x)$	(par `all_le_top`)
$\bigvee Y \leq \delta(\bigvee Y)$ et $\delta(\bigvee Y) \leq \delta(x)$	(par transitivité)
$\bigvee Y \leq \delta(\bigvee Y)$ et $\bigvee Y \leq x$	(par `incr`)
$\bigvee Y \leq \delta(\bigvee Y)$	(par hypothèse)
$\forall z \in Y, z \leq \delta(\bigvee Y)$	(par `sup_least`)
$z \leq \delta(\bigvee Y)$	(ajout d'hypothèses)
$\delta(z) \leq \delta(\bigvee Y)$	(par $z \in Y$ et `A_incl_P`)
$z \leq \bigvee Y$	(par `incr`)
	(par $z \in Y$ et `sup_upper`)

□

Et nous pouvons définir la restriction de δ au treillis complet intervalle `I`. Pour pouvoir y appliquer le théorème de plus petit point fixe, `δ'` doit être croissante sur `I`, ce qui est trivialement vérifié.

```
    Definition δ' (x : I) : I := ⟨δ (π1 x), restr_betw_sup (π1 x) (π2 x) ⟩.

    Lemma incr' : ∀ x y : I, x ≤ y → δ'(x) ≤ δ'(y).
```

Rappelons que la borne supérieure doit être définie sur `P'`, et donc retourner un élément de type `P'`, soit un couple composé d'un élément de type `L` et de la preuve qu'il s'agit d'un point fixe de δ. Or le plus petit point fixe de δ`'` est un élément de type `I`. Il faut donc extraire la première composante de ce point fixe (de type `L`), et lui associer la preuve que c'est un point fixe de δ. Commençons par établir la preuve.

```
Lemma fixp'_fixpt : δ( π1 fixp x, δ'(x) ) ≃ π1 fixp x, δ'(x).
```

Démonstration.
La difficulté de la preuve tient au type de l'opérateur $\simeq$. En effet, celui du lemme est une relation sur les éléments de type `L`, tandis que dans δ`'(fixp x, `δ`'(x)) ≃ fixp x, `δ`'(x)` (obtenu par `fixp_fixpt` sur `I` et δ`'`), son type est `I`.

Comme `a` $\leq$ `b` sur `I` implique π_1 `a` $\leq$ π_1`b` sur `L`, alors par extension `a` $\simeq$ `b` sur `I` implique π_1 `a` $\simeq$ π_1`b` sur `L`, quelque soient `a` et `b` de type `I`.

Donc de δ`'(fixp x, `δ`'(x)) ≃ fixp x, `δ`'(x)` nous pouvons en déduire que l'on a π_1 δ`'(fixp x, `δ`'(x)) ≃ `π_1 `fixp x, `δ`'(x)`, ce qui par définition de δ`'` se ramène à ce qu'on voulait, c'est-à-dire δ`(`π_1 `fixp x, `δ`'(x)) ≃ `π_1 `fixp x, `δ`'(x)`. □

Et nous pouvons enfin définir l'opérateur de borne supérieure sur `P'`.

```
Definition supp : P' := exist P (π1 fixp x, δ'(x)) fixp'_fixpt.
```

Le prédicat n'étant pas déduit par Coq, nous devons l'expliciter, et nous ne pouvons alors pas utiliser la notation des couples ⟨`a, b`⟩.

Et avant de clore la section, pour profiter la définition de l'intervalle `I`, nous prouvons un lemme technique qui sera utilisé par la suite.

```
  Lemma fixp_least_fixpt : ∀ x : I, δ(π1 x) ≃ π1 x → fixp x, δ'(x) ≤ x.

 End sup.
```

La borne inférieure est définie de la même façon.

```
 Section inf.

   Let J := interval L (⊥) (⋀Y) (bot_le_all (⋀Y)).

   Lemma restr_betw_inf : ∀ x : L, ⊥ ≤ x ≤ ⋀Y → ⊥ ≤ δ(x) ≤ ⋀Y.

   Definition δ'' (x : J) : J := ⟨ δ(π1 x) , restr_betw_inf (π1 x) (π2 x) ⟩.

   Lemma incr'' : ∀ x y : J, x ≤ y → δ''(x) ≤ δ''(y).
   Lemma Fixp''_fixpt : δ( π1 Fixp x, δ''(x) ) ≃ π1 Fixp x, δ''(x).

   Definition infp : P' := exist P (π1 Fixp x, δ''(x)) Fixp''_fixpt.

   Lemma Fixp_greatest_fixpt : ∀ x, δ(π1 x) ≃ π1 x → x ≤ Fixp x, δ''(x).

 End inf.

End sup_and_inf.
```

Il ne nous reste plus qu'à prouver les propriétés sur ces opérateurs de `P'` pour pouvoir construire le treillis complet des points fixes.

```
Infix "≤ᵖ" := lep (at level 70, no associativity).
Notation "⋀ᵖ A" := (infp A) (at level 60, no associativity).
Notation "⋁ᵖ A" := (supp A) (at level 60, no associativity).

Implicit Type A : ℘(P').

Lemma lep_refl : ∀ x, x ≤ᵖ x.
Lemma lep_trans : ∀ x y z, x ≤ᵖ y → y ≤ᵖ z → x ≤ᵖ z.
Lemma infp_lower : ∀ A, ∀ x ∈ A, (⋀ᵖ A) ≤ᵖ x.
```

Démonstration.

$\bigwedge^p$A $\leq^p$ x	(ajout d'hypothèses)
$\pi_1\bigwedge^p$A $\leq$ π_1x	(déf. `lep`)
$\pi_1\bigwedge^p$A $\leq$ $\bigwedge\Pi_1$A et $\bigwedge\Pi_1$A $\leq$ π_1x	(par transitivité)
$\pi_1\bigwedge^p$A $\leq$ $\bigwedge\Pi_1$A et π_1x $\in$ Π_1A	(par `inf_lower`)
$\pi_1\bigwedge^p$A $\leq$ $\bigwedge\Pi_1$A	(par `member_proj` et hypothèse x $\in$ A)
π_1(Fixp x, δ'' A x) $\leq$ $\bigwedge\Pi_1$A	(déf. `infp`)
	(deuxième composante de Fixp x, δ''(x))

□

Des considérations techniques similaires ont lieu lors de la preuve du lemme suivant. Sa démonstration n'est pas donnée, ni celle des lemmes duaux.

```
Lemma infp_greatest : ∀ A, ∀ z, (∀ x ∈ A, z ≤ᵖ x) → z ≤ᵖ ⋀ᵖA.

Lemma supp_upper : ∀ A, ∀ x ∈ A, x ≤ᵖ ⋁ᵖA.
Lemma supp_least : ∀ A, ∀ z, (∀ x ∈ A, x ≤ᵖ z) → ⋁ᵖA ≤ᵖ z.
```

Finalement nous pouvons construire le treillis complet des points fixes.

```
Definition fixpts_lattice := Build_QuasiCompleteLattice P'
  lep infp supp lep_refl lep_trans
  infp_lower infp_greatest supp_upper supp_least.

End fixpoints_lattice.
```

Ceci termine ce chapitre de formalisation des théorèmes de points fixes dans un treillis complet.

Chapitre 4 Remarques d'ordre général

Pendant l'étape de formalisation, tout ne s'est pas toujours déroulé sans difficulté. Ces contretemps obligent à prendre conscience et à considérer la théorie sur laquelle repose Coq, de façon plus précise que le simple aspect système formel sur lequel s'est concentrée l'introduction p.3.

4.1 Règles de réduction

À tout moment, Coq effectue des calculs en réécrivant les termes qu'il manipule. Il y a plusieurs règles de calcul, qui toutes sont nécessaires. Prenons par exemple le théorème `fixp_pow` de la section 3.5 :

```
fixp_pow
     : ∀ L : QuasiCompleteLattice,
       (∀ α β : L → L,
        (∀ x y : L, x ≤ y → α(x) ≤ α(y))
        → (∀ x y : L, x ≤ y → β(x) ≤ β(y))
          → (∀ x : L, α∘β(x) ≃ β∘α(x))
            → (∀ n : nat, (α^n) (μx, α∘β(x)) ≃ μx, α∘β(x)))
```

Supposons que l'on souhaite démontrer $\delta(\delta(\mu x, \delta(x)) \simeq \mu x, \delta(x)$ sachant que δ est croissante dans un treillis donné L. Il est facile de montrer que la fonction identité id est croissante dans le treillis L et qu'elle commute avec δ. Alors nous disons ; « il suffit d'appliquer `fixp_pow` à δ et id et pour $n = 2$ », et nous obtenons le théorème. Mais avant de parvenir à cette conclusion, des étapes et des procédés de calcul sont nécessaires. Ces procédés de calcul sont :

- la δ-réduction qui permet de remplacer un identificateur par sa définition ;
- la β-réduction qui permet de remplacer une expression de la forme $(\lambda x.u)\ t$ par $u[x \leftarrow t]$;
- la ι-réduction qui permet par exemple d'évaluer l'application de la fonction puissance `pow`, dont le deuxième argument est un entier, en effectuant du *pattern-mathing* sur cet entier ;
- la ζ-réduction qui remplace des définitions locales, mais que nous n'utiliserons pas pour nos exemples.

Voici une séquence de réductions qui nous amène à cette conclusion :

$$\begin{array}{lll} \texttt{fixp\_pow}\ L\ \delta\ id\ \ldots 2 & \rhd_{\delta} & (\forall L, \ldots)\ L\ \delta\ id\ \ldots 2 \\ & \rhd^{*}_{\beta} & \delta^2(\mu x, \delta(x)) \simeq \mu x, \delta(x) \\ & \rhd_{\iota} & \delta \circ \delta^1(\mu x, \delta(x)) \simeq \mu x, \delta(x) \\ & \rhd^{*}_{\iota\beta} & \delta \circ \delta(\mu x, \delta(x)) \simeq \mu x, \delta(x) \\ & \rhd^{*}_{\delta\beta} & \delta(\delta(\mu x, \delta(x))) \simeq \mu x, \delta(x) \end{array}$$

Dès lors qu'il n'est plus possible de réduire un terme par l'une de ses règles, il est en forme normale. Dans le calcul des constructions, tous les termes sont normalisables, et ceci quel que soit l'ordre des étapes de calcul (normalisation forte, cf. [3, 21, 10]). La forme normale est unique à renommage de variables liées près (α-conversion).

De ces notions découle une relation entre les termes ; la relation de convertibilité. Deux termes t et t' sont convertibles s'ils ont même forme normale, on note $t \equiv t'$. Ceci vaut également pour les types, puisque ce sont eux-mêmes des termes.

Un terme u de type T est aussi de type T' si T et T' sont convertibles. Ainsi, lors de la démonstration d'un théorème, s'il est possible de construire un terme dont le type est convertible avec le but actuel, alors en appliquant ce terme au but courant, on le résout. Si les types ne sont pas convertibles, Coq signale un échec.

4.2 Preuve par dualité

C'est ce qui se passe par exemple lorsque l'on essaye d'appliquer le principe de dualité à `fixp_inf_fixpts` pour démontrer le théorème `Fixp_sup_fixpts`. Coq signale qu'il ne peut pas réussir à convertir les deux expressions.

Pour résumer la situation, nous voulons montrer $Fixp_x\gamma(x) \simeq \bigvee P$ sachant qu'on a déjà montré $fixp_x\gamma(x) \simeq \bigwedge P$, avec $P = \{x; \gamma(x) \simeq x\}$.

Prenons la formule duale de $fixp_x\gamma(x) \simeq \bigwedge P$.

$$\begin{array}{ll} (fixp_x\gamma(x) \simeq \bigwedge P)^d & \rhd \bigwedge^d\{x; \gamma(x) \leq^d x\} \simeq^d \bigwedge^d\{x; \gamma(x) \simeq^d x\} \\ & \rhd \bigwedge^d\{x; \gamma(x) \simeq^d x\} \simeq \bigwedge^d\{x; \gamma(x) \leq^d x\} \\ & \rhd \bigvee\{x; x \simeq \gamma(x)\} \simeq \bigvee\{x; x \leq \gamma(x)\} \\ & \equiv \bigvee\{x; x \simeq \gamma(x)\} \simeq Fixp_x\gamma(x) \end{array}$$

Ce n'est en effet pas convertible avec $Fixp_x\gamma(x) \simeq \bigvee P$, et ce n'est donc pas une preuve de cette formule. Pour pouvoir tout de même utiliser le principe de dualité, il faut alors « préparer » le but à cet effet. Déjà, nous pouvons ramener la démonstration de $Fixp_x\gamma(x) \simeq \bigvee P$ à la démonstration de $Fixp_x\gamma(x) \simeq^d \bigvee P$ par utilisation du lemme `simeq_dual_to_primal` : $\forall L, \forall xy : L, x \simeq^d y \to x \simeq y$. Puis, si l'on arrive à montrer que $\bigvee\{x; x \simeq \gamma(x)\} \simeq \bigvee P$, alors par transitivité de $\simeq$, nous pourrons réduire le but à $Fixp_x\gamma(x) \simeq^d \bigvee\{x; x \simeq \gamma(x)\}$ et appliquer le principe de dualité.

C'est le cas, voyons ce que donne le code Coq :

```
Section greatest_fixed_point.

  Variable L : QuasiCompleteLattice.
  Variable γ : L → L.

  Implicit Type x y z : L.

  Hypothesis incr : ∀ x y, x ≤ y → γ(x) ≤ γ(y).

  Notation P := (fixpts γ).
  Notation P' := { x ; x ≃ γ(x) }.

  Lemma technical : ⋁P ≃ ⋁P'.
  Lemma incr_dual : ∀ x y, x ≤^d y → γ(x) ≤^d γ(y).
```

```
Theorem Fixp_sup_fixpts : ⋁ P ≃ Fixp x, γ(x).
  Proof.
  rewrite technical.
  apply simeq_dual_to_primal.
  apply ( fixp_inf_fixpts (dual L) γ incr_dual ).
  Qed.

End greatest_fixed_point.
```

Lorsqu'en langage naturel on se permet de dire qu'il suffit d'appliquer le principe de dualité, en fait nous sommes obligés d'effectuer de nombreuses opérations avant d'en être parfaitement convaincus. Évidemment, un mathématicien expert d'un domaine *voit*, ou *sent* ces choses. Et alors ce genre de désagréments devient rapidement ennuyeux. Mais pour la défense de Coq, il est possible, sur la longueur, qu'une faute ou un oubli technique crucial se glisse dans les démonstrations, et alors les conséquences peuvent être désastreuses.

4.3 Égalité de Leibniz et convertibilité

L'égalité en Coq est définie par l'axiome $\forall x, x = x$, ceci quel que soit le type de x. Donc pour montrer que deux éléments sont égaux, i.e. $x = y$, il faut que x et y soient convertibles.

Si l'on veut montrer l'involutivité de l'opérateur de dualité, i.e. $\forall L, dual(dual(L)) = L$, il faut que les deux éléments soient convertibles. Puisqu'il s'agit de structures, c'est-à-dire de n-uplets, il suffit que leurs composantes soient deux à deux égales. En particulier, leurs opérateurs d'ordres doivent être convertibles. Essayons de réduire la formule $(\leq^d)^d = \leq$.

$$\begin{aligned}(\leq^d)^d = \leq \quad &\rhd (\geq)^d = \leq \\ &\rhd (x\ y \mapsto y \leq x)^d = \leq \\ &\rhd (x\ y \mapsto y \leq^d x) = \leq \\ &\rhd (x\ y \mapsto x \leq y) = \leq\end{aligned}$$

Et nous ne pouvons pas aller plus loin. Pourtant les deux fonctions produisent les mêmes résultats sur leurs arguments, et nous souhaiterions pouvoir les déclarer égales. Il nous manque l'axiome d'extensionnalité en Coq. Il existe plusieurs variantes de cet axiome, mais nous pouvons le résumer par :

```
∀ E F : Type, ∀ f g : E → F, (∀ x : E, f x = g x) → f = g.
```

Ainsi, cet axiome nous permet de montrer l'involutivité de l'opérateur de dualité. Mais il existe aussi une autre possibilité ; l'introduction d'une règle de calcul supplémentaire, l'η-réduction :

$$\lambda x.f\ x \rhd_\eta f$$

Dans ces conditions, puisque l'opérateur $\leq$ n'est qu'une notation pour la relation `le`, alors par deux applications de cette règle, nous pouvons déduire l'égalité précédente.

Attention à ne pas confondre, si l'extensionnalité et l'η-réduction nous permettent de terminer la démonstration, et malgré leur lien apparent, il faut bien remarquer qu'ils n'agissent pas au même niveau. L'extensionnalité est un axiome, tandis que l'η-réduction est une règle de calcul.

L'extensionnalité peut être ajoutée en tant qu'axiome et ne rend pas incohérent le système formel. Cependant, nous avons préféré ne rajouter aucun axiome lors du développement, et donc nous en priver. Pourtant à plusieurs occasions ce principe nous aurait

été utile. Notamment pour pouvoir utiliser l'égalité de Coq et ses tactiques efficaces de réécriture.

4.4 La réécriture

L'égalité de Coq est en fait l'égalité de Leibniz ; deux objets sont égaux si et seulement si toute propriété vérifiée pour l'un l'est pour l'autre. Cela peut se résumer par :

$$\forall a\ b, (a = b \Leftrightarrow (\forall P, P(a) \Leftrightarrow P(b)))$$

La définition de l'égalité en Coq est faite de façon inductive (cf. [3, ch.8]). Ainsi un principe d'induction est généré, qui se résume (quel que soit le type de x) en :

$$\forall x, \forall P, P(x) \rightarrow \forall y, x = y \rightarrow P(y)$$

Lorsqu'il faut prouver une expression E où a apparaît libre, et si l'on sait que $a = b$ dans le contexte actuel, on peut réécrire a en b. C'est-à-dire qu'on remplace les occurrences libres de a par b. C'est en fait une simple application du principe d'induction et de la convertibilité : l'expression E est remplacée par $(x \mapsto E[a \leftarrow x])\ a$ et est convertible avec E par β-réduction, puis le principe d'induction est appliqué au prédicat $(x \mapsto E[a \leftarrow x])$ et à $b = a$ (qui est déduit de $a = b$). Ainsi, on est ramené à montrer $E[a \leftarrow b]$.

On peut ne remplacer que certaines occurrences de la variable libre visée par utilisation de la tactique `pattern` à laquelle on donne les numéros des occurrences à remplacer. Cette tactique provoque une β-expansion (l'inverse de la β-réduction) sur les occurrences ciblées.

Mais puisque cette égalité de Leibniz n'a pas été utilisée dans notre développement, la réécriture qui facilite beaucoup la manipulation d'équations (p.ex. dans `minimax` de la section 3.6) n'était pas possible sur notre pseudo-égalité.

4.5 Setoïd

Heureusement, un ensemble de méthodes du fichier prédéfini `Setoid.v` de Coq permettent d'utiliser la réécriture sur des relations moins fortes que l'égalité de Leibniz (voir [21, ch.21]). Par exemple, lorsqu'on a une relation d'équivalence $\simeq$ entre éléments d'un certain type, si nous savons que $a \simeq b$ et nous voulons montrer que $a \simeq c$, alors il est possible de réécrire a en b dans $a \simeq c$ puisque la relation est transitive et réflexive. On peut même étendre le principe à des relations non symétriques (p.ex. $\leq$).

Mais pour pouvoir réécrire un terme a en b sachant que $a \simeq b$, si a est un sous-terme de l'expression courante, alors il faut définir des morphismes entre relations. Par exemple, notre but courant est $f(a) \simeq f(c)$ et nous savons que $a \simeq b$. Pour réécrire a en b dans $f(a) \simeq f(c)$, il faut que f soit un morphisme sur $\simeq$, c'est-à-dire que $\forall x\ y, x \simeq y \rightarrow f(x) \simeq f(y)$. Dans ces conditions, par application de la symétrie et de la transitivité, nous obtenons $f(b) \simeq f(c)$. Ces morphismes et ces relations doivent être spécifié *a priori* par une syntaxe particulière.

Par exemple, dans le théorème `minimax`, nous devons montrer $x_0 \simeq fixp_x\theta(x, x_0)$ sachant que $x_0 \simeq Fixp_y\theta(x_0, y)$. Pour cela, nous désirons passer par l'étape intermédiaire $x_0 \simeq fixp_x\theta(x, Fixp_y\theta(x_0, y))$, c'est-à-dire réécrire la seconde occurrence de x_0. Bien sûr, $y \mapsto fixp_x\theta(x, y)$ est un morphisme sur $\simeq$ (car θ est croissante). Si nous utilisons naïvement la réécriture sur l'hypothèse, nous substituons les deux occurrences de x_0 dans la formule. Mais si nous utilisons la tactique `pattern`, alors le but se transforme en

$(z \mapsto x_0 \simeq fixp_x\theta(x,z))\ x_0$. Or $(z \mapsto x_0 \simeq fixp_x\theta(x,z))$ n'est pas un morphisme sur $\simeq$, et il est donc impossible d'appliquer la réécriture de `Setoid` sur cette expression.

En fait, il existe une tactique, `change`, qui remplace un but par un autre s'ils sont convertibles. Il est donc possible en appliquant cette tactique d'obtenir le but intermédiaire $x_0 \simeq (z \mapsto fixp_x\theta(x,z))\ x_0$, sur lequel il est possible d'utiliser la réécriture de `Setoid` pour obtenir ce qu'on voulait.

Il serait agréable que la tactique `pattern` soit étendue pour permettre ces β-expansions dans des sous-termes. Par exemple, puisqu'un terme peut être représenté par un arbre, on pourrait lui spécifier un chemin dans cet arbre puis spécifier le numéro d'occurrence du terme désiré, pour que la β-expansion s'effectue à partir de ce niveau. Par exemple, on aurait pu écrire `pattern` x_0 `at 1 into r` où `r` désignerait une descente dans la branche droite (*right*) de l'arbre.

4.6 Automatisation

À la longue, les théorèmes sur les treillis sont souvent développés plusieurs fois, les démonstrations de croissance sont toujours demandées et sont très simples, et ce travail répétitif détourne de l'objectif et fait perdre un temps considérable. Coq propose des procédures de décision ou de semi-décision pour certains problèmes ou contextes particuliers. Par exemple, pour la logique propositionnelle intuitionniste, il existe une procédure de décision qui permet de résoudre un but automatiquement, la tactique `tauto`.

Mais plus généralement, la tactique `auto` permet la résolution de buts par recherche exhaustive en tentant d'appliquer des théorèmes d'une base de connaissances. Le principe de la recherche de `auto` est simple. Tout d'abord `auto` essaye de résoudre le but en appliquant la tactique `assumption`. Puis elle réduit le but en utilisant la tactique `intros`, ajoutant les hypothèses ainsi introduites à sa base de données. Et enfin elle essaye d'appliquer les tactiques de sa base de connaissances au but courant.

En fait, pour savoir quel théorème appliquer, le symbole qui se trouve à la racine du terme du but courant est associé à un ensemble de tactiques qui lui sont applicables. Puis ces tactiques sont essayées les unes après les autres, en appliquant la moins coûteuse (en terme de sous-buts générés) en premier. L'arbre de recherche ainsi déployé est limité par défaut à une hauteur de 5. Il est possible d'augmenter la hauteur de l'arbre de recherche à l'invocation de la tactique, p.ex. `auto 10`.

Cette méthode est très générale, et nécessite que l'utilisateur donne des indices pertinents à cette tactique en remplissant sa base de données. Les théorèmes de points fixes de Knaster ainsi que de Tarski peuvent être résolus par ce moyen. Des mécanismes plus puissants ont été utilisés pour cela (cf. [21, ch.8]).

Cependant, si la tactique réussit, le terme généré peut être difficile à interpréter. Et la trace des étapes de la preuve n'est pas donnée. On perd donc la possibilité de comprendre et de suivre la démonstration avec une utilisation intensive de ces mécanismes.

Bonus

Les notations de Coq permettent des définitions récursives. Par exemple, un terme représentant une liste d'éléments s'écrit `cons a (cons b (cons c nil))` avec un λ-terme. Mais il est plus habituel d'avoir $[a; b; c]$. Pour cela, on peut proposer pour chaque longueur de liste, une nouvelle notation. C'est irréalisable. Coq permet donc des notations récursives. Les règles de construction sont strictes. On peut les utiliser pour définir une notation des ensembles en extension.

```
Definition oreq (E:Type) (x a:E) (B:Prop) := x=a ∨ B.
Implicit Arguments oreq [E].
Notation "{ a , b , .. , c }" := ({ x ; oreq x b .. (oreq x c (eq x a)) .. })
        (at level 0, a at level 99, b at level 99, c at level 99) : type_scope.
```

Et pour pouvoir résoudre des buts de la forme `0 ∈{0,1,2}`, il suffit de définir une tactique grâce au langage de tactiques de Coq (cf. [3, ch.7]).

```
Ltac trivial_set :=
  unfold member, oreq; repeat ((left; reflexivity) || (right; try reflexivity)).

Goal 0 ∈ {0,1,2}.
trivial_set.
Qed.

Goal 11 ∈ {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14}.
trivial_set.
Qed.
```

Chapitre 5 Conclusion

À la lumière de notre développement, nous pouvons affirmer que l'utilisation d'outils d'aide à la démonstration pour formaliser des raisonnements mathématiques « en vraie grandeur » n'est pas utopique. Il est possible de formaliser et de démontrer des théorèmes non triviaux dans ces systèmes.

Coq, avec toutes ses notations, ses portées, ses arguments et conversions implicites, ses modules, etc., nous permet d'approcher le style mathématique. Cela nécessite cependant une forte interaction de la part de l'utilisateur. À cet argument, on peut objecter qu'il suffit que les développements soient suffisamment nombreux, organisés et structurés. C'est vrai, mais cela suffira-t'il à pousser les mathématiciens actuels à l'utilisation de tels systèmes ?

Il faut plutôt orienter ses espoirs vers les nouvelles générations d'informaticiens et de mathématiciens, desquelles émergeront peut-être le nouveau métier de « formalisateur ». Il s'agirait d'un travail à mi-chemin entre les mathématiques et l'informatique. Les mathématiques, parce qu'il est essentiel de comprendre un théorème formulé en langage naturel, et de pouvoir « combler » les atteintes à la rigueur des systèmes formels utilisés. L'informatique, puisque les méthodes de développement, les formalismes et les programmes utilisés doivent être maîtrisés, et parce que la formalisation s'effectue évidemment sur ordinateur.

Mais plus largement, que faut-il attendre de tels systèmes ? Pour apporter des éléments de réponse à cette question, reprenons quelques arguments pertinents apportés par Buchberger dans [5] et son projet de « journal en tant qu'acteur mathématique actif ». Le but du projet est de formaliser tous les articles d'un journal scientifique de mathématiques[1]. L'objectif étant de « faire vivre » les textes mathématiques que l'on peut y trouver. En conservant tous les théorèmes dans une base de données, et en y associant des outils de raisonnement et de réorganisation des connaissances, il serait possible de répondre à de nombreuses questions de manière efficace et rapide. Par exemple, il serait possible d'interroger le système pour connaître l'originalité d'un texte, la correction d'un théorème, l'importance d'un résultat par analyse ou comptage de ses corollaires, les hypothèses minimales nécessaires à l'utilisation d'un résultat, etc. Ainsi, le travail de divers acteurs de la publication d'un article serait facilité et modifié, en passant par l'auteur, les correcteurs et les lecteurs.

Si le projet réussit, à terme, le processus de publication serait modifié. Par exemple, chaque article pourrait être accompagné de sa formalisation dans un système particulier. Et la confiance accordée aux théorèmes serait fortement augmentée.

Finalement, pour réconcilier définitivement les mathématiques (classiques) et l'informatique (théorique), il faut conserver cette humanité qui nous uni ; le langage. Les travaux de Coscoy (cf. [7]) sur la traduction de sources formalisées (en Coq) vers le langage naturel est une étape cruciale à cette réussite, et ne doit pas être négligée.

[1] Il s'agit du *Journal of Symbolic Computation* d'Elsevier, fondé par Buchberger en 1985.

Références

[1] A. Arnold et D. Niwiński : *Rudiments of μ-Calculus*, volume 146 de *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2001.

[2] H. Barendregt et F. Wiedijk : The challenge of computer mathematics. *Philosophical transactions of the Royal Society A*, 363(1835):2351–2375, 2005.

[3] Y. Bertot et P. Castéran : *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Springer, 2004.

[4] G. Birkhoff : Lattice theory, 3rd ed. *American Mathematical Society Colloquium Publications*, 25, 1967.

[5] B. Buchberger : Journal as active math-agents : Outline of a project with a mathematics publisher. synasc, 0:11–12, 2007.

[6] L. Colson : On diagonal fixed points of increasing functions. *Theoretical Computer Science*, 222:181–186, 1999.

[7] Y. Coscoy : *Explication textuelle de preuves pour le calcul des constructions inductives*. Thèse de doctorat, Université de Nice-Sophia-Antipolis, 2000.

[8] B. A. Davey et H. A. Priestley : *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[9] A.C. Davis : A characterization of complete lattices. *Pacific Journal of Mathematics*, 5(2): 311–319, 1955.

[10] G. Dowek : *Théorie des types, notes de cours de master 2*, 2004.

[11] J.Y. Girard : Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, 1972. Thèse de Doctorat d'État.

[12] J.Y. Girard : Le lambda-calcul du second ordre. *Séminaire N. Bourbaki*, 1986-87(678):173–185, Feb. 1987.

[13] J.Y. Girard, Y. Lafont et P. Taylor : *Proofs and types*. Cambridge University Press New York, 1989.

[14] H. Herbelin : Le théorème de schroeder-berstein dans le calcul des constructions. Mémoire de D.E.A., INRIA – Rocquencourt, Sep. 1988.

[15] B. Knaster : Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématiques*, 6:133–134, 1928.

[16] P. Martin-Löf et Z.A. Lozinski : Constructive mathematics and computer programming [and discussion]. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences (1934-1990)*, 312(1522):501–518, 1984.

[17] D. Niwiński : Equational μ-calculus. *Lecture Notes in Computer Science*, 208:169–176, 1984.

[18] D. Pichardie : *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. Thèse de doctorat, Université de Rennes 1, Dec. 2005.

[19] H. Poincaré : *Dernières pensées*. Flammarion, 1913.

[20] A. Tarski : A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[21] The Coq Development Team : *The Coq Proof Assistant : Reference Manual*, juillet 2007. v8.1.