Rapport de stage M2
Partial orders and fixpoint theorems, in Coq

Gabrielle Pauvert
stage encadré par Damien Pous et Yannick Zakowski

février-juin 2022

Introduction générale

Parmi toutes les révolutions qu'a permis le développement de l'informatique théorique et expérimentale dans le milieu de la recherche, la preuve formelle occupe une place de choix. Parce qu'une preuve ne sera jamais aussi fiable et rigoureuse que lorsque qu'elle est validée mécaniquement par ordinateur au sein d'une théorie cohérente, ce domaine de recherche offre la promesse d'une garantie sans précédent, en plus de permettre l'élaboration de preuves jusque là inaccessibles à cause de leur longueur et de leur complexité (par exemple, le théorème des quatre couleurs, dont la démonstration de 1991 exigeait de traiter 1478 cas critiques [**?**]). L'un des fers de lance de la preuve formelle est **Coq**, un assistant de preuve développé depuis les années 1980 par les chercheurs de l'INRIA, en collaboration avec l'École Polytechnique, l'université de Paris-Sud, l'université de Paris Diderot, le CNRS et depuis les années 1990, l'ENS de Lyon.

Si la bibliothèque standard de Coq contient déjà de nombreux modules, dont un petit module sur les Ordres Partiels Complets (CPO), il manque encore à Coq une bibliothèque générale regroupant les principaux résultats de la théorie des ordres partiels. En particulier les théorèmes de points fixes. Ces résultats sont utiles à de nombreux domaines de l'informatique, comme la sémantique, la logique, l'interprétation abstraite, l'optimisation ou l'algorithmie. Des versions simplifiées et spécialisées de ces notions ont d'ailleurs parfois été déjà formalisés en Coq pour les besoins de projets concrets, comme , qui pourraient bénéficier d'une bibliothèque générale, polyvalente et modulaire, dans laquelle les principaux résultats auraient déjà été formalisés sous leur forme la plus générale et réutilisable, adaptée à plusieurs niveaux de structures.

Au cours de mon stage, j'ai développé une telle bibliothèque autonome et indépendante, articulée notamment autour du chapitre 8 de La bibliothèque a été construite progressivement en testant différentes formalisations et paramétrisations, dans le but d'englober le plus possible de structures et d'utilisations différentes en un même outil très général. Le code de la bibliothèque, que j'ai intégralement produit, se trouve au lien suivant : https://github.com/Gabzcr/Coq-CPOs/tree/master.

Table des matières

1 Présentation du domaine de recherche

1.1 Notions de théorie des ordres partiels

Commençons par rappeler et définir quelques notions générales du domaine, en particulier les ordres et les structures ordonnées.

Définition 1.1. Ordre (partiel) : Soit P un ensemble. Un ordre $\leq$ sur P est une relation binaire qui est (i) réflexive, (ii) transitive et (iii) antisymétrique. C'est à dire que pour tout x, y et $z \in P$, on a :
(i) $x \leq x$
(ii) si $x \leq y$ et $y \leq z$ alors $x \leq z$.
(iii) si $x \leq y$ et $y \leq x$ alors $x = y$.

Notez que l'égalité entre deux éléments, $x = y$, est loin d'être un problème facile. Les subtilités liées à l'égalité sont particulièrement visibles dans la manipulation d'un assistant de preuve formelle comme Coq. Nous travaillerons donc principalement dans ce projet avec des **pré-ordres** plutôt que des ordres, c'est à dire des relations qui sont seulement réflexives et transitives ((i) et (ii)), et une notion d'égalité ad hoc $\equiv$ définie par $x \equiv y \iff (x \leq y \wedge y \leq x)$.

Définition 1.2. bottom et top : Soit P un ensemble ordonné, i.e. muni d'un ordre $\leq$. On dit que P possède un élément bottom ($\bot$) si : $\exists \bot \in P, \forall x \in P, \bot \leq x$.
Dualement, P possède un top si : $\exists \top \in P, \forall x \in P, x \leq \top$.
Notez que s'ils existent, ces éléments sont uniques par antisymétrie.

Rappelons également rapidement les notions de borne supérieure (abrégée en sup) et de borne inférieure (abrégée en inf) :

Définition 1.3. sup et inf : Soit P un ensemble ordonné et $S \subseteq P$. La borne supérieure de S, si elle existe, est le plus petit des majorants de S. La borne inférieure est le plus grand des minorants (si elle existe).
De manière équivalente, S a un sup si et seulement si : $\forall y \in P, [(\forall s \in S, s \leq y) \iff x \leq y]$.
S'ils existent, on note $\bigvee S$ le sup de S et $\bigwedge S$ l'inf de S.

Maintenant, on peut définir des structures ordonnées plus complexes sur les ensembles ordonnés. Une structure très communément rencontrée et très polyvalente est celle des CPOs (Ordres Partiels Complets). Pour cela, définissons d'abord les ensembles dirigés.

Définition 1.4. Sous-ensemble dirigé : Soit P un ensemble ordonné et $S \subseteq P$. S est dirigé si pour toute paire x, y d'éléments de S, il existe un majorant de $\{x, y\}$ dans S :
$\forall x, y \in S, \exists z \in S, x \leq z \wedge y \leq z$.

En général, on appellera $D \subseteq P$ un sous-ensemble dirigé de P, et on notera $\bigsqcup D = \bigvee D$ le sup d'un ensemble dirigé, quand il existe.

Définition 1.5. CPO : On dit qu'un ensemble ordonné P est un CPO si :
(i) P possède un élément bottom $\bot$.
(ii) $\bigsqcup D$ existe pour tout sous-ensemble dirigé D de P.

Dans la littérature, on trouve de nombreuses variantes des définitions ci-dessus. Notez que contrairement à la définition classique d'un ensemble dirigé, comme celle donnée dans le livre qui me sert de référence, ici j'autorise l'ensemble vide comme étant un ensemble dirigé. Ainsi je peux restreindre la définition de CPO au seul point (ii), car le sup de l'ensemble vide donne bottom par la définition de la borne supérieure : $\bigsqcup \emptyset = \bot$.

Par ailleurs, certains auteurs n'imposent pas du tout l'existence d'un élément bottom dans P et parlent plutôt de **CPO pointé** ("dcppo") lorsque bottom existe. À l'inverse, on peut parler de **pré-CPO** (dcpo) lorsqu'on veut laisser les considérations d'existence de l'élément bottom de côté, ou retirer $\bot$ de la structure de CPO.

Dans tout ce projet, on ne travaillera jamais avec des pré-CPO, toujours avec des CPO contenant $\bot$ car l'existence d'un élément bottom (et en particulier d'un élément tout court) dans P sera une condition nécessaire à l'existence et au calcul de points fixes.

Une structure dans le prolongement de celle de CPO, plus restrictive et plus forte, est celle du treillis complet.

Définition 1.6. Treillis complet Soit P un ensemble ordonné non vide. P est un treillis complet si $\bigvee S$ et $\bigwedge S$ existent pour tout sous-ensemble $S \subseteq P$ quelconque.

On dit que P est un treillis si : $\forall x, y, x \vee y \ \ (= \bigvee\{x, y\})$ et $x \wedge y \ \ (= \bigwedge\{x, y\})$ existent.

Notez qu'un treillis complet est en particulier un CPO.

1.2 Les théorèmes de point fixe sur les ordres partiels

Maintenant que nous avons redéfini les notions et les structures qui constitueront la base de la bibliothèque, voyons les principaux résultats de points fixes construits sur ces structures.

Les différents théorèmes suivants statuent de l'existence d'un point fixe d'une fonction $F : P- > P$ sous différentes conditions plus ou moins fortes, où P est un ensemble ordonné. Commençons par rappeler quelques propriétés sur les fonctions, qui formeront des hypothèses aux théorèmes ci-dessous.

Définition 1.7. Point fixe d'une fonction
Soient $(P, \leq_P)$ et $(Q, \leq_Q)$ deux ensembles ordonnés, $F : P \to Q$ une fonction et $x \in P$.
x est un point fixe de F si $F(x) = x$.
x est un pré-point fixe si $F(x) \leq x$.
x est un post-point fixe si $x \leq F(x)$.

Définition 1.8. Fonction monotone Soient $(P, \leq_P)$ et $(Q, \leq_Q)$ deux ensembles ordonnés. Une fonction $\varphi : P \to Q$ est dite monotone si elle préserve l'ordre, i.e. : $\forall x, y \in P, x \leq_P y \Longrightarrow \varphi(x) \leq_Q \varphi(y)$.

Notez qu'une fonction décroissante au sens usuel n'est pas une fonction monotone, selon cette définition.

Théorème 1.1. ***Knaster-Tarski*** *Soit L un treillis complet et $F : L \to L$ une fonction monotone. Alors : $\alpha := \bigvee\{x \in L \mid x \leq F(x)\}$ est un point fixe de F, et c'est le plus grand point fixe de F.*
Dualement, $\mu = \bigwedge\{x \in L \mid F(x) \leq x\}$ est le plus petit point fixe de F.

Le théorème de Knaster-Tarski est le plus simple et le plus direct à démontrer, mais c'est aussi celui qui demande les hypothèses les plus fortes, notamment de travailler dans un treillis complet. À cause des restrictions imposées sur la structure, ce théorème est moins général que les suivants, mais il offre l'avantage de fournir une formule du point fixe. Les trois théorèmes suivants s'appliquent à n'importe quel CPO.

Définition 1.9. Fonction continue Soient $(P, \leq_P)$ et $(Q, \leq_Q)$ deux CPOs. Une fonction $\varphi : P \to Q$ est dite continue si elle préserve les limites, c'est à dire dans ce contexte les borne supérieures. Plus formellement, φ est continue si : $\forall D \subseteq P$ dirigé , le sous-ensemble $\varphi(D) \subseteq Q$ est dirigé, et

$$\varphi(\bigsqcup D) = \bigsqcup \varphi(D)$$

Avec notre définition d'ensembles dirigés, une fonction continue préserve les éléments bottom. Notez qu'une fonction continue est monotone.

Théorème 1.2. ***Théorème de point fixe I***
Soient P un CPO et $F : P \to P$ une fonction monotone sur P. Posons $\alpha := \bigsqcup_{n\geq 0} F^n(\bot)$.
(i) Si α est un point fixe de F, alors α est le plus petit point fixe de F.
(ii) Si F est continue, alors F a un plus petit point fixe et c'est α.

Théorème 1.3. ***Théorème de point fixe II : Pataraia*** *Soient P un CPO et $F : P \to P$ une fonction monotone. Alors F a un plus petit point fixe.*

Comme nous le développerons plus tard dans ce rapport, le théorème de Pataraia a la particularité très intéressante d'avoir une preuve entièrement calculable, qui fournit une formule et une méthode pour déterminer le point fixe en question. Ceci est notamment utile pour Coq, car il fournit une théorème qui permet de simplifier et calculer concrètement un point fixe, en particulier sur un CPO fini.

Notez que le théorème II implique le théorème I, puisqu'une fonction continue est monotone.

Définition 1.10. Fonction progressive Soient P un CPO. Une fonction $F : P \to P$ est dite progressive si tous les éléments de P sont des post-points fixes de F, i.e : $\forall x \in P, x \leq F(x)$.

Théorème 1.4. ***Théorème de point fixe III : Bourbaki-Witt*** *Soient P un CPO et $F : P \to P$ une fonction progressive. Alors F a un point fixe.*

Contrairement aux deux précédents, le théorème de Bourbaki-Witt n'est pas prouvable en logique intuitionniste. Il faut donc faire appel au tiers exclu ou à d'autres axiomes un peu plus faibles.

Mon stage m'a amené à remarquer une erreur dans le livre Contrairement à ce qui y est écrit, F n'a pas forcément de point fixe *minimal*, et en particulier le top de P_0 n'est pas un point fixe minimal en général, même si c'est bien un point fixe (cf page TODO). En voici un contre-exemple. Un autre contre-exemple a été passé et vérifié dans Coq (fichier Application.v, TODO lignes), et sera mentionné en

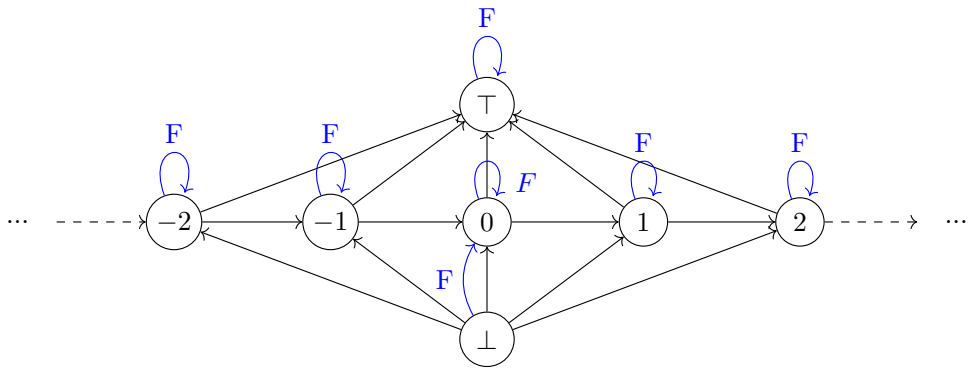

FIGURE 1 – Fonction F progressive sur un CPO, sans point fixe minimal

2 Bibliothèque finale proposée

La bibliothèque Coq dans son état le plus abouti, que j'ai entièrement codée pendant mon stage, est le contenu du dossier `CPO_projet` que l'on peut trouver au lien suivant :
https://github.com/Gabzcr/Coq-CPOs/tree/master/CPO_project.

2.1 Aperçu global de la bibliothèque

Avant de rentrer dans le vif du sujet, donnons rapidement un plan du projet et les caractéristiques principales de chaque fichier. Le dépôt git contient, en plus de la bibliothèque finale, trois fichiers indépendants qui correspondent à des versions antérieures ou alternatives du projet.

- `Basic_CPO.v` est la première version écrite de la bibliothèque, et la plus proche du livre de référence. Elle suit les preuves mathématiques presque pas à pas, quitte à faire quelques détours inutiles. C'est la moins complexe de tous, et donc la plus simple à lire et à comprendre. Je me servirai parfois du travail fait dans ce fichier pour illustrer mes explications sans rajouter la difficulté de la paramétrisation ou de définitions plus alambiquées et plus générales.
- `Propositional_CPO.v` contient une version alternative de la bibliothèque dans lequel le sup est défini comme une proposition liant un ensemble dirigé et un élément, plutôt qu'une fonction associant un élément à un ensemble dirigé. Nous y reviendrons en
- `Parametrized_CPO.v` contient une première version maladroite de la bibliothèque des CPOs paramétrisée par un ensemble de valeurs de vérité, appelé "B" tout au long du projet. Il sera détaillé en

Quand au projet final, `CPO_project`, il est divisé en trois fichiers principaux.

- `CPO.v` est le fichier central de la bibliothèque. Il contient les définitions de structures ordonnées et les théorèmes de point fixe dans leur état final, ainsi qu'un nombre conséquent de résultats intermédiaires et autres propriétés utiles sur les ordres, les fonctions et les bornes sup/inf.
- `FiniteSet.v` contient une définition d'ensembles finis utilisée pour ce projet et des propriétés sur les ensembles finis. Il sera détaillé en
- `Applications.v` contient la définition des CPOs concrets, comme les Propositions, les Booléens et quelques CPOs finis. Le fichier utilise également le travail effectué dans le reste de la bibliothèque pour prouver qu'un ordre partiel fini muni d'un élément bottom est un CPO, ce qui en constitue le résultat principale. Nous en parlerons plus en détail en
  Il contient également un contre-exemple à l'erreur du livre prouvé en Coq, via le lemme `top_of_P0_is_not_minimal`, et un calcul concret de point fixe utilisant les méthodes du théorème II.

2.2 Les enjeux et problématiques de la formalisation

Parmi les différentes façon possibles de définir les structures ordonnées, on recherche une formalisation qui respecte certains critères.

Universalité :
Le premier critère, et le plus important de tous, est que nos structures englobent bien tous les objets mathématiques qui correspondent à notre définition. Par exemple, dans le cas des treillis complets (ou des CPOs), on veut pouvoir définir l'ensemble des Propositions dans Coq comme étant un treillis complet, dont la relation $P \leq Q$ est donnée par $P \to Q$ et le sup d'un ensemble $\mathcal{P}$ est donné par $\bigvee \mathcal{P} = \exists P \in \mathcal{P}, P$.

Mais on veut également pouvoir définir de petites structures, comme par exemple le treillis complet des booléens {`true`, `false`} avec `false` $\leq$ `true`, ou n'importe quel treillis fini. Ici, la formalisation usuelle d'un sous-ensemble $S \subseteq X$ dans Coq ne le permet pas. Dans ce cas, S est vu comme une fonction `S : X -> Prop`.

Or, les Propositions de Coq ne sont pas décidables, et nous ne pouvons donc pas définir le sup $\bigvee S$ d'un sous-ensemble quelconque $\bigvee S, S \subseteq$ {`true`, `false`}. Nous aimerions en effet le définir comme : `if true` $\in S$

`then true else false`. Mais pour cela, nous avons besoin de décider si `true` appartient à l'ensemble S ou non, ce qui correspondrait à décider la Proposition Coq (S `true`).

Notez que décider une proposition est encore plus fort que de supposer le tiers exclu, qui statue simplement que la proposition suivante est vraie :

```
forall (P : Prop), P  ∨   (P → False)
```

Mais ne sait pas déterminer en dehors d'un environnement de preuve lequel des deux côté du "" est vérifié. Nous aurions plutôt besoin du résultat suivant, bien plus contraignant, et qui permet de matcher vers l'un ou l'autre des résultats :

```
forall (P : Prop), { P } + {P → False}
```

Or, ce résultat est faux en général dans les Propositions. Il suffit de penser par exemple au problème de l'arrêt sur les machines de Turing, que l'on peut définir comme une proposition dans Coq. Ici, le problème vient donc des valeurs de vérité que nous avons choisies pour définir nos sous-ensembles, et donc pour définir notre treillis et son sup. Nous voudrions travailler dans `Bool` au lieu de `Prop`, où tout est décidable, et définir les sous-ensembles comme des fonctions `S : X -> Bool`. Mais une telle définition ne fonctionnerait plus pour définir le treillis des Propositions, par exemple.

Pour contourner ce problème, nous avons choisi d'inclure dans la définition des structures ordonnées une paramétrisation par un ensemble de valeurs de vérité, nommé `B`, construit pour englober à la fois `Prop` et `Bool`. Les sous-ensembles (dirigés ou non), sont alors définis à valeurs dans `X -> B`. L'implémentation et la réalisation concrètes seront donnés plus bas, en

Généralité :

Le deuxième critère, qui a déjà été rapidement évoqué, est de faire appel au moins d'axiomes possibles afin de rester le plus général possible. Dans notre cas, nous restons en logique intuitionniste tout du long et évitons au maximum l'utilisation de l'axiome d'extensionnalité fonctionnelle :

```
Axiom functional_extensionality_dep : forall {A} {B : A → Type},
  forall (f g : forall x : A, B x),
  (forall x, f x = g x) → f = g.
```

Cet axiome n'est utilisé que dans le fichier `FiniteSet.v`, pour prouver que les propriétés de finitude que nous imposons sont conservées depuis des ensembles finis `X, Y` vers l'ensemble `X -> Y`. Nous en parlerons plus en détail dans la section en question

Calculabilité :

Un troisième critère toujours aussi fondamental est celui de la calculabilité. Les étapes de preuves d'existence de point fixe doivent le plus possible être algorithmique et constructives, pour permettre de calculer concrètement un point fixe, voire un point fixe minimal, dans un CPO donné.

Malheureusement, les théorèmes I et III ne fournissent pas de telles preuves. En effet, le théorème III utilise un axiome dérivé du tiers exclu. Et le théorème I fournit pour point fixe l'élément $\alpha = \bigvee_{n\geq 0} F^n(\bot)$ qui n'est pas calculable non plus, car il s'agit d'un sup d'ensemble infini indexé par $\mathbb{N}$. En réalité, on pourrait en dériver un algorithme sur un CPO fini. On sait que la suite $(F^n(\bot))$ stagne après un nombre fini d'étapes, au plus égal au cardinal de l'ensemble X support du CPO. Malheureusement, on ne sait pas combien d'étapes sont nécessaires, et en toute généralité (sur un CPO infini) il est difficile de déterminer le point fixe minimal par cette méthode, au lieu de juste montrer son existence.

Mais la preuve du théorème II (Pataraia) fournit une preuve astucieuse et un peu détournée qui a l'avantage d'être entièrement constructive. Avec un peu de travail, nous avons réussi à l'adapter en Coq de sorte à garder cette constructivité. En particulier, dans le cas des CPOs finis qui prennent leurs valeurs de vérité dans les booléens, le plus petit point fixe peut être entièrement calculé et fournir l'élément concret du CPO correspondant. Il est donné dans le fichier `CPO.v` par `lfp_II`.

Le calcul a été testé avec succès dans le fichier `Applications.v` sur le CPO à trois éléments $\bot <= x1 <= x2$ et la fonction F monotone définie par $F(\bot) = F(x1) = x1$ et $F(x2) = x2$, par le code de test suivant :

```
set (x := @lfp_II Bool_B CPO_valid_type B_PO_ex B_CPO_ex Fmon).
vm_compute in x.
```

Ce qui donne le résultat suivant : `x := x1 : CPO_set`

Par ailleurs, la première version de la formule du point fixe minimal donné par le théorème II prenait environ 14s à calculer, un temps désagréablement long pour un si petit CPO. Une accélération conséquente a été obtenu par une méthode de mémoïsation sur l'ensemble le plus long à calculer, P_F défini plus tard en qui consiste à pré-calculer cet ensemble (c'est à dire son image, i.e. sa valeur de vérité, sur chaque élément du CPO). En voici le code, où `(el (projT2 X))` est une liste contenant les éléments du CPO X, `is_member` est une fonction définie plus haut dans le fichier `Applications.v` qui teste l'appartenance d'un élément dans une liste, et `P` sera l'ensemble P_F (de type `X -> Bool`).

```
memo X P := let l := List.filter P (el (projT2 X)) in
            fun x ⇒ is_member x l;
```

Simplicité :

Enfin, il est préférable que la formalisation des structures ordonnées soit la plus simple possible, notamment à manipuler. À cette fin, notre toute première tentative de formalisation de CPO proposait une fonction sup totalisée, définie sur tous les sous-ensembles mais spécifiée uniquement sur les sous-ensembles dirigés, à la manière de la division dans Coq qui est définie partout mais non spécifiée sur 0. La représentation était la suivante (où `Directed D` est la proposition indiquant que D est dirigé, et `D y` indique que $y \in D$ avec `D : X -> Prop` sous-ensemble de X) :

```
sup : (X → Prop) → Prop;
sup_spec: forall D, Directed D → forall z,
          ((sup D) <= z  ↔  forall (y:X), D y → y <= z);
```

Malheureusement, cette représentation par le type `sup : (X -> Prop) -> Prop` ne permettait pas de définir une notion aussi basique que le CPO des fonctions monotones d'un CPO dans un autre. En effet, dans le CPO $\langle P -> Q \rangle$ des fonctions monotones de P dans Q CPOs, on veut définir le sup d'un ensemble dirigé $\mathcal{F}$ par $(\bigvee_{\langle P \to Q \rangle} \mathcal{F})(x) = \bigvee_X \{y \mid \exists f \in \mathcal{F}, y = f(x)\}$, c'est à dire en Coq :

```
(sup F) : mon := fun x ⇒ sup (fun y ⇒ exists f, F f  ∧  y = f x)
```

Or, dans le cas où $\mathcal{F}$ n'est pas dirigé, l'ensemble des $\{y \mid \exists f \in \mathcal{F}, y = f(x)\}$ ne l'est pas non plus donc son sup n'est pas spécifié, et nous ne pouvons pas montrer que le `sup F` défini ici est bien une fonction monotone (donc bien typé) dans le cas où il n'est pas spécifié.

Pour cette raison, j'ai rapidement abandonné cette tentative de formalisation sans la garder dans le dépot git au profit d'un sup défini uniquement sur les ensembles dirigés, malgré la nécessité d'utiliser des types dépendants, un peu complexes à manipuler, pour définir le type des ensembles dirigés.

2.3 Structure de valeurs de vérités

Rentrons maintenant dans les détails de l'implémentation en Coq qui satisfait le plus possible tous ces enjeux. Comme discuté plus haut, nous avons d'abord besoin d'un ensemble de valeurs de vérité B qui puisse être instancié à la fois en `Prop` et `bool`. Nous l'utiliserons pour définir nos sous-ensembles comme des fonctions de type `X -> B` où X est la structure ordonnée que nous voulons définir.

Notre ensemble B contient une fonction d'évaluation `is_true : B -> Prop` qui plonge nos propres valeurs de vérité dans `Prop`. On s'en sert notamment pour pouvoir formuler des propositions à partir de nos objets, et statuer dans Coq que quelque chose est vrai. Par exemple, avec ce qui a déjà été dit ci-dessus, on voudrait pouvoir prouver dans Coq qu'un élément $x \in X$ appartient à ou sous-ensemble $S \subseteq X$, ce qu'on formulerait comme suit :

```
Lemma belongs_to S x : is_true (S x).
```

Ensuite, on souhaite doter B d'un élément `Faux` noté `BFalse`, et des opérations classiques "ou", "et" et l'implication : `\/`, `/\` et `->`, de manière à ce que qu'ils se comportent comme attendu avec l'évaluation `is_true`. Par exemple pour `BFalse` et `/\`, ça donne :

```
BFalse : B;
BFalse_spec : ~ (is_true BFalse);
BAnd : B → B → B;
BAnd_spec : forall b1 b2,
          is_true b1  ∧  is_true b2  ↔  is_true (BAnd b1 b2);
```

La principale difficulté rencontrée pour définir cet ensemble est la définition des opérations `forall` $\forall$ et `exists` $\exists$. Comme ces opérations doivent être décidables dans le cas où `B = bool`, on ne peut pas se permettre de définir ces opérations sur des ensembles quelconques comme dans `Prop`. Mais nous voulons au moins définir ces opérations sur X, sur l'ensemble des sous-ensembles (dirigés ou non) de X ($\forall Y \subseteq X$ dirigé, [...]) et sur l'ensemble des fonctions monotones $X \to X$. Nous en aurons besoin dans les preuves de théorèmes de point fixe.

Pour éviter de définir quatre opérations `Forall` et `Exists` différentes, comme nous l'avions envisagé initialement, nous avons rajouté à la définition de B une opérateur K qui indique sur quels types nous disposons de ces opérations. On appelle **valides** les types sélectionnés par K. K doit vérifier un certain nombres de propriétés. Pour commencer, il faut que l'ensemble X support de notre structure ordonnée soit valide, mais aussi que tous les ensembles mentionnés plus haut le soit. De manière générale, on veut que K soit clôt par passage à l'ensemble des fonctions sur deux types valides $V_1 \to V_2$, et dans le cas des sous-ensembles $X \to B$. On veut aussi qu'un sous-type d'un type valide reste valide, i.e. que K soit clôt par sélection d'éléments d'un ensemble valide par une propriété :

$$\forall V \in \texttt{Type}, \forall P \in (V \to \texttt{Prop}), V \in K \implies \{v : V \mid \texttt{is\_true}(P\ v)\} \in K$$

.

K avait d'abord le type `Type -> Prop`, pour indiquer quels types sont valides, mais il a fallu plutôt lui donner le type légèrement plus troublant `Type -> Type` pour gérer la sélection des types finis, définis au début du fichier `FiniteSet.v`, par le `Record fin`. Ça se manipule de la même façon.

Dans le cas où `B = Prop`, tous les types sont valides, car on peut toujours définir ces opérations, d'où `K = fun (A :Type) => True`. Dans le cas où `B = bool`, on définit les opérations `Forall` et `Exists` sur les types finis avec égalité décidable, et il a fallu prouver ces propriétés de clôture.

```
K : Type → Type;
subtype_closure (A : Type) : K A → forall (P : A → B),
                              K {a : A | is_true (P a)};
function_closure (A B : Type) : K A → K B → K (A → B);
set_closure (A : Type) : K A → K (A → B);

valid_type := { TBody : Type & K TBody};
```

Maintenant que nous nous avons défini notre opérateur K qui sélectionne les types valides, nous pouvons définir les opérations `Forall` et `Exists` sur les types valides :

```
BForall (V : valid_type) : (((TBody V) → B) → B);
BForall_spec (V : valid_type) : forall (P : (TBody V) → B),
  (forall x, is_true (P x))  ↔  is_true (BForall V P);
```

Enfin, comme indiqué plus haut, on intègre à notre ensemble B une fonction de mémoïsation pour pouvoir pré-calculer les sous-ensembles les plus coûteux dans le cas où `B = bool`, et ainsi optimiser les temps d'exécution de calcul concrets de point fixe dans le cas fini.

```
memo (X : valid_type) : ((projT1 X) → B) → ((projT1 X) → B);
memo_spec (X : valid_type) : forall P x, is_true (memo X P x)  ↔  is_true (P x);
```

Le code complet de la définition de la structure des valeurs de vérité se trouve au début du fichier `CPO.v` et est redonné en annexe

Un des inconvénients de travailler avec nos propres valeurs de vérité est que ça alourdit grandement l'écriture des propriétés à prouver. On doit se traîner des `BAnd`, `BOr`, etc. un peu partout avec leur écriture préfixe, au lieu des habituels $/\backslash$, $\backslash/$ infixes. J'aurais dû, au cours du développement de la bibliothèque, rajouter des notations par dessus ces définitions pour les rendre plus lisibles et se ramener à la manipulation connu des Propositions, mais ça n'a pas encore été fait. Pour la suite de ce rapport, j'écrirais autant que possible le code de `Basic_CPO.v` à la place de celui de la bibliothèque, ou alors je modifierai les notations pour revenir à celle des Propositions normales afin de ne pas complexifier inutilement la lecture du code, mais nous resterons bien dans B.

La tactique `unfold_spec` a précisément été créée dans la bibliothèque (`CPO.v`, ligne 63) pour pousser l'évaluation aux feuilles et la logique habituelle dans `Prop`.

Une question qui s'est naturellement posée durant le stage est la suivante : est-il possible de définir une autre structure de vérité, différente de `bool` et `Prop`, qui soit pertinente ou ouvre de nouvelles possibilités. Nous avons exploré les logiques à trois valeurs ou à un nombre fini de valeurs, et considéré la logique de Łukasiewicz sur $[0, 1]$. Mais nous avons rencontré des difficultés à les transcrire dans notre modèle.

Après avoir rencontré quelques difficultés en manipulant les spécifications notamment de l'implication et du "ou" vis-à-vis de l'évaluation, nous en sommes venus à la conclusion que l'évaluation permet, dans le cas fini du moins, de séparer les éléments de notre ensemble B en deux catégories. D'abord, les éléments évalués à true : $\texttt{is\_true}^{-1}(True)$ qui se comportent tous comme le booléen `true`, et les autres éléments qui se comportent tous comme le booléen `false`. Aussi il semble impossible d'ajouter une troisième valeur pertinente dans le cas fini, qui soit réellement distincte de True et False. En revanche, il reste une possibilité de trouver un B pertinent différent de Prop dans le cas infini.

Des tentatives de définition d'un ensemble B fini distinct de `bool` ont été écrites dans le fichier `Applications.v`, section `CPO_based_Truth_values`, ligne 792, notamment pour transcrire une logique à trois éléments $\{\bot, U, \top\}$. Elles vont dans le sens constaté plus haut, le troisième élément U se comporte soit de la même manière que $\top$, soit de la même manière que $\bot$, sans qu'il soit possible de définir autrement l'implication en respectant les spécifications.

2.4 Structure d'ordre, de CPO et de treillis

Maintenant que nous avons vu nos valeurs de vérité dans B, nous pouvons définir les ensembles dirigés comme des sous-ensembles de X de type `X -> B` vérifiant une certaine propriété, et donc les treillis complets et les CPOs. Mais avant cela, nous avons besoin d'une structure plus générale d'ensemble (partiellement) ordonné, appelé PO. Nous posons sur notre ensemble un pré-ordre `leq`, et ajoutons par dessus une notion d'égalité ad hoc spécifique appelée `weq` et notée par le symbole infixe $\equiv$, de sorte que `leq` soit ordre, i.e. qui garantit l'antisymétrie : $x \equiv y \Leftrightarrow (x \leq y / \ y \leq x)$. Comme on définit `leq` et `weq` à valeurs dans B, il faut encore rajouter l'évaluation `is_true` un peu partout pour que ça ait un sens dans les Propositions, et pour en faire de véritables relations d'ordre. Attention, les évaluations `is_true` ne seront plus toujours précisées dans la suite du rapport, pour alléger les notations. On écrira simplement `x <= y` et `x == y` pour les évaluations de `leq` et `weq`.

```
Class B_PO := {
    weq: X → X → B;
    leq: X → X → B;
    Preorder_leq :> PreOrder (fun x y ⇒ is_true (leq x y));
    weq_spec: forall x y, is_true (weq x y)
          ↔ (is_true (leq x y)  ∧  is_true (leq y x));
  }.
```

Maintenant, on peut définir nos ensembles dirigés en traduisant la définition , puis les structures de CPO et de treillis complet par dessus une structure d'ensemble ordonné. (Note : quelques subtilités Coq de coercions de Types ont été laissées de côté ci-dessous.)

```
Definition Directed {X} `(leq : rel X) (D : X → B) : Prop := forall x y,
   is_true (D x) → is_true (D y) → exists z, D z  ∧  x <= z  ∧  y <= z.
Definition directed_set `(leq : X → X → B) :=
   {Dbody : set | is_true (Directed leq Dbody)}.

Class B_CPO `(P' : B_PO) := {
    sup: directed_set leq → X;
    sup_spec: forall D z, (sup D <= z  ↔
      forall (y:X), is_true (D y) → y <= z);
  }.

Class B_CL `(L' : B_PO) := {
    Sup: (X → B) → X;
    Sup_spec: forall Y z, (Sup Y <= z  ↔
      forall y, is_true (Y y) → y <= z);
  }.
```

La seule différence notable est l'ensemble de définition du sup/Sup de la structure. Dans le premier cas, il n'est défini que sur les ensembles dirigés, alors que dans l'autre cas il est défini sur tous les sous-ensembles.

Il a fallu faire un choix entre définir la structure de treillis complet (CL) par-dessus celle de CPO, car un treillis complet est en particulier un CPO, ou séparer les structures comme il a été fait ici. Séparer les structures est à mon sens plus clair, et permet de définir seulement une fonction sup par structure, distinctes. En revanche, ça dédouble certaines preuves basiques qu'on aimerait avoir à la fois dans les deux structures. Pour les preuves plus complexes, on utilise simplement la propriété qu'un CL est un CPO pour obtenir un CPO et appliquer la preuve sur les CPO.

2.5 Détails du fichier CPO.v

Pour donner une rapide idée de tout ce qui a été fait dans la bibliothèque, y compris les résultats sur lesquels je ne vais pas m'attarder, voici un bref résumé du fichier principal, `CPO.v`, section par section.

B (l.6) : La définition de la structure des valeurs de vérité et quelques propriétés sur B.

CPO_CL (l.70) : Les définitions des structures d'Ordre Partiel (PO), de CPO et de treillis complet, ainsi que des ensembles dirigés.

Forall_sets (l.132) : Juste la définition des types $X \to X$, $X \to B$ et $\{D \subseteq X \mid Ddirig\}$ en tant que type valides.

Partial_order (l.152) : $\equiv$ est une relation d'équivalence, définition de fonctions monotones et fonctions particulières.

Sup (l.199) : Propriétés sur la fonction sup, définitions et propriétés de $\bot$ et $\top$, et de la fonction Inf.

ForLattices (l.250) : Propriétés sur les treillis complets uniquement : définitions de *join* et *meet* binaires, i.e. Sup et Inf sur un ensemble à deux éléments, et propriétés.

Knaster_Tarski (l.333) : Les constructions du théorème de Knaster-Tarki, pour les treillis complets.

Function (l.375) : Définitions et propriétés sur les fonctions $X \to X$: images, continuité, points fixes, chaines. Utilisées auparavant pour le théorème I.

Sets (l.458) : Inclusion et Égalités d'ensembles.

Particular_CPOs (l.476) : Définition et preuves du treillis/CPO des fonctions monotones sur X et des fonctions sur X, ainsi que du treillis/CPO des parties de X. Définition de sous-CPO et propriétés.

Invariant_subCPOs (l.702) : Définition de P_F, appelé `PO` dans le fichier, le plus petit sous-CPO invariant de X pour une fonction F. Propriétés essentielles. Cet ensemble sera central dans les théorèmes de point fixe.

Increasing_fixpoint (l.751) : Fonctions progressives, définitions et propriétés. Notamment, existence d'un point fixe commun à toutes les fonctions monotones progressives sur un même CPO. Ce résultat est utilisé dans le théorème II (Pataraia), mais nous avons dû contourner l'utilisation du CPO des fonctions monotones pour des problèmes de types dépendants dans B, aussi cette section n'est finalement pas utilisée telle quelle mais les résultats sont reformulée plus bas sous d'autres formes.

Fixpoint_II (l.805) : Construction et preuve du théorème II, s'appuyant sur les éléments précédents.

Bourbaki_Witt (l.978) : Construction et preuve du théorème III (Bourbaki-Witt).

3 Les preuves des théorèmes de point fixe

Parler ici des subtilités des preuves de théorèmes de point fixe (je ne m'en souviens plus immédiatement, à revoir), et les façons de les contourner. Garder pour plus tard le problème des sous-CPOs qui ne sont pas définissables dans la paramétrisation actuelle, et l'astuce qui a permis de contourner le pb.

Evoquer Knaster-Tarski ?

3.1 Théorème I

Expliquer rapidement le principe, et en quoi c'est impossible à formaliser dans la version finale, mais que c'est une conséquence du théorème II. Evoquer rapidement les pistes explorées côtés ordinaux (notamment pour le théorème III aussi), et chaines.

3.2 Théorème II : Pataraya

Donner le principe qui passe par le CPO des fonctions monotones (?). Plus haut ? Expliquer les difficultés à formaliser un sous-CPO dans Coq. Donner l'astuce (mathématique) qui permet de contourner ce problème et aboutir.

S'arrêter un peu sur P0 et sa définition qui sont centraux pour les deux prochains théorèmes.

3.3 Théorème III : Bourbaki-Witt

Donner l'idée (?). Evoquer le fait que le théorème le Bourbaki-Witt (III) n'est pas prouvable en logique intuitionniste et les axiomes minimaux nécessaires.

4 Représentions alternatives et version précédentes

4.1 Fonction sup totalisée

Exemple : fichier Lattice utilisé pour la coinduction. Faire le parallèle avec la division par zéro (pour expliquer l'intérêt de la totalisation). Donner les limites et les points de blocage : définition du CPO des fonctions, et expliquer pourquoi la méthode a été abandonnée.

4.2 Fonction sup propositionnelle

Parler de la version "work_prop" (à renommer d'ailleurs) dans laquelle le sup était une proposition sur des paires (ensemble, élément) visant à indiquer qu'un élément est le sup d'un ensemble. Indiquer les avantages (simple à la base, permet de définir bool et Prop comme des treillis sans problème) et les problèmes soulevés (non calculable, très fastidieux et laborieux pour faire la moindre preuve, notamment à cause des fonctions représentées par leur graphe).

4.3 Première paramétrisation : Forall dépendants de l'ensemble support de l'ordre partiel

Parler des premières versions tentées pour la paramétrisation et des problèmes rencontrés (la démultiplication des Forall, certains difficiles à définir, notamment pour les futurs utilisateurs).

5 Application : les CPOs finis

5.1 Exemple : Un CPO à trois éléments

Evoquer rapidement l'implémentation du CPO à trois éléments, (re)donner l'exemple concret de calcul d'un sup par le théorème II, et évoquer le contre-exemple à l'erreur du bouquin.

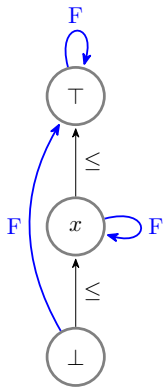

FIGURE 2 – Top de P_0 non minimal

5.2 Représentation d'un ensemble fini

Expliquer plus longuement la représentation d'un ensemble fini par un type dont tous les éléments sont contenus dans une liste.

5.3 Propriétés de clôture

(Passer le code des preuves en annexe ?) Evoquer les difficultés liées aux types dépendants.

5.4 Tout ordre partiel fini est un CPO

5.4.1 Algorithme utilisé pour obtenir le sup

Exposer l'algorithme de parcours avec mise à jour des candidats éléments maximaux

5.4.2 Preuve de correction

(Ne pas expliquer toute la preuve, seulement les principales difficultés !)

6 Limitations et approfondissements possibles

Une section pour dire ce qui reste améliorable, et ce qu'on aurait aimé faire avec plus de temps (nouveaux changements possibles sur B, application en théorie des catégories, etc ?)

Conclusion

A Du code trop gros / long / peu pertinent pour le mettre dans le rapport ?

Définition de l'ensemble des valeurs de vérité.

```
Class B_param := { B : Type;
  K : Type → Type;

  (* Basic operations on B *)
  is_true : B → Prop;

  BFalse : B;
  BTrue : B;
  BFalse_spec : ~ (is_true BFalse);
  BTrue_spec : is_true BTrue;
  BAnd : B → B → B;
  BOr : B → B → B;
  BAnd_spec : forall b1 b2, is_true b1  ∧  is_true b2  ↔  is_true (BAnd b1 b2);
  BOr_spec : forall b1 b2, is_true b1  ∨  is_true b2  ↔  is_true (BOr b1 b2);
  BImpl : B → B → B;
  BImpl_spec : forall b1 b2, (is_true b1 → is_true b2)  ↔  (is_true (BImpl b1 b2));

  (* Closure properties on K *)
  subtype_closure (A : Type) : K A → forall (P : A → B), K {a : A | is_true (P a)}; (* fo
  function_closure (A B : Type) : K A → K B → K (A → B);
  set_closure (A : Type) : K A → K (A → B);

  (* Forall and Exists :*)
  valid_type := { TBody : Type & K TBody};
  TBody (V : valid_type) := projT1 V;

  BForall (V : valid_type) : (((TBody V) → B) → B);
  BForall_spec (V : valid_type) : forall (P : (TBody V) → B),
    (forall x, is_true (P x))  ↔  is_true (BForall V P);
  BExists (V : valid_type) : (((TBody V) → B) → B);
  BExists_spec (V : valid_type) : forall (P : (TBody V) → B),
    (exists x, is_true (P x))  ↔  is_true (BExists V P);


  (* Memoisation for computation speed-up*)
  memo (X : valid_type) : ((projT1 X) → B) → ((projT1 X) → B);
  memo_spec (X : valid_type) : forall P x, is_true (memo X P x)  ↔  is_true (P x);
  }.
```

TODO !

Test de code :

```
CoInductive stream = cons { hd: nat; tl: stream }.
Infix "::" := cons.
```