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 est difficilement 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 [1]). L'un des fers de lance de la preuve formelle est **Coq**, un assistant de preuve développé depuis les années 1980.

Si la bibliothèque standard de Coq contient déjà de nombreux modules, dont un petit module sur les Ordres Partiels Complets (CPO) [2], il manque encore à Coq une bibliothèque générale regroupant les principaux résultats de la théorie des ordres partiels [3]. 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 travaux sur le sujet existent en théorie des domaines [4]. Des versions simplifiées et spécialisées de ces notions ont d'ailleurs parfois été déjà formalisés en Coq pour les besoins de projets concrets, comme les projets de Damien Pous sur la coinduction [5] et les algèbre relationnelles [6]. Ces projets 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.

En particulier, on dispose sur les CPOs de théorèmes de points fixes, c'est à dire d'éléments qui stabilisent une fonction (définition 1.7). Ils ont été étudiés avec des hypothèses variées, et démontrés par de nombreux moyens et à de maintes reprises [3] [7] [8], par exemple avec ou sans méthode déductive, imprédicative, etc. Ces résultats sont centraux dans de nombreux domaines. En sémantique par exemple, on peut donner du sens aux boucles for et while en déterminant le point fixe de leurs programmes. Ils interviennent aussi dans de nombreux algorithmes, comme certains algorithmes qui étudient les relations entre les états d'un graphe

Au cours de mon stage, j'ai développé une telle bibliothèque autonome et indépendante, articulée notamment autour de [3, chapitre 8]. Ce livre servira de référence tout au long de ce rapport.

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 [9] : 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 $\simeq$ définie par $x \simeq y \iff (x \leq y \wedge y \leq x)$.

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* ($\bot$) (terme emprunté à l'Anglais) 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 à égalité $\simeq$ près 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 s'il existe un élément x (le sup) tel que :
$\forall y \in P, [(\forall s \in S, s \leq y) \iff x \leq y]$.

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.

Définissons maintenant les CPOs et les treillis complets, qui sont des ordres partiels sur lesquels on sait qu'une partie plus ou moins vaste des sous-ensembles admet une borne sup.

Définition 1.5. CPO : On dit qu'un ensemble ordonné P est un CPO si :
$\bigsqcup D$ existe pour tout sous-ensemble dirigé D de P.

Dans la littérature, on trouve de nombreuses variantes des définitions ci-dessus. La définition classique d'un ensemble dirigé impose que l'ensemble soit non vide, contrairement à celle qui est utilisée ici. En général, la définition de CPO inclut donc aussi l'existence d'un élément bottom $\bot$ dans P. C'est déjà le cas ici, 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 [3, page 175].

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, qui en est une sous-classe plus restrictive , est celle du treillis complet.

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

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 défini les notions et les structures qui constitueront la base de la bibliothèque, voyons les principaux résultats de points fixes existant sur ces structures.

Les différents théorèmes suivants statuent de l'existence d'un point fixe d'une fonction $F : P \to P$ 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 $(P, \leq_P)$ un ensemble ordonné, $F : P \to P$ 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. : $\forall x, y \in P, x \leq_P y \implies \varphi(x) \leq_Q \varphi(y)$.
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 dirigées. 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 (2.3), le théorème de Pataraia a la particularité très intéressante d'avoir une preuve qui fournit une méthode de calcul concret de point fixe, contrairement au théorème I dont la formule n'est pas exploitable en pratique. Ceci s'avère particulièrement utile pour les opérations concrètes sur les CPO finis.
Notez que le théorème II implique l'existence de point fixe du théorème I, puisqu'une fonction continue est monotone.

Définition 1.10. Fonction progressive (*Increasing* en Anglais) 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 [10]. 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 [3, page 188]. Contrairement à ce qui y est écrit, sous les hypothèses du théorème III, F n'a pas forcément de point fixe *minimal*. En particulier, le point fixe donné n'est pas minimal en général, même si c'est bien un point fixe.
En voici un contre-exemple en Figure 1. Un autre contre-exemple plus spécifique à la formulation du livre a été passé et vérifié dans Coq (fichier Application.v), et sera mentionné en 5.1.
[YZ: Peut-être un tableau avec les quatres théorèmes en ligne, et hypothèse sur P, hypothèses sur F, classique/intuitioniste, formule effective ou non comme colonnes ?]

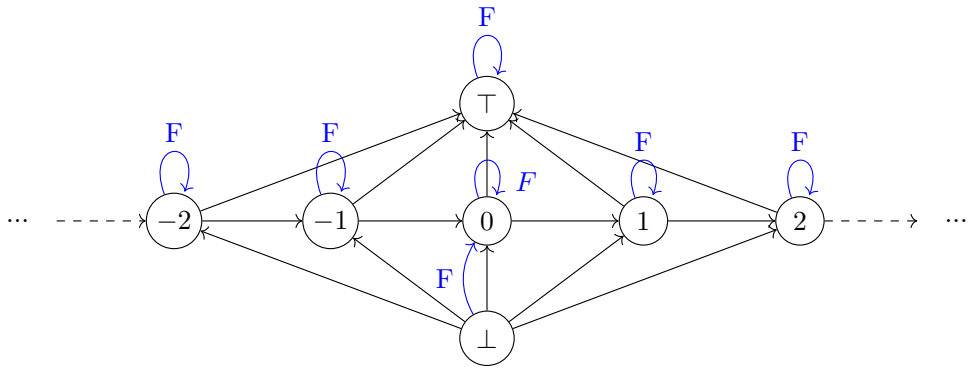

FIGURE 1 – Fonction F progressive sur un CPO, sans point fixe minimal

2 Les enjeux et problématiques de la formalisation

Parmi les différentes façon possibles de modéliser les structures ordonnées, on recherche une formalisation qui respecte certains critères. Nous allons viser dans ce projet l'universalité de la bibliothèque, la minimalité en terme d'axiomes et la calculabilité.

2.1 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. Pour comprendre l'étendue de cet enjeux, explicitons une des spécificités de Coq : la différence entre les booléens `bool` et les propositions `Prop`.

Les booléens forment un ensemble à deux éléments {`true`, `false`} muni des opérations booléennes usuelles ("ou", "et", "non", "implique", etc.). Ainsi, une évaluation booléenne d'une propriété est soit vraie soit fausse, et tout est décidable. En particulier, un "il existe" ou "pour tout" booléen n'existe pas toujours, on ne peut le définir que lorsque ça reste décidable (par exemple, sur une liste finie).

Au contraire, les Propositions de Coq constituent un ensemble bien plus vaste (infini, en fait) qui encode entre autres toute la logique du premier ordre. On y trouve les prédicats de base `True` et `False` mais aussi tous les opérateurs logiques sans restriction $\backslash/, /\backslash, \implies, \neg, \exists, \forall$, une notion d'égalité $=$ etc. `Prop` capture en particulier les propriétés indécidables. Par exemple, on peut exprimer dans `Prop` le fait qu'une machine de Turing s'arrête et encoder le problème de l'arrêt, qu'on peut prouver indécidable.

Une question se pose alors, quel type choisir pour représenter les sous-ensembles ? Sur un ensemble X, une partie $S \subseteq X$ n'est autre qu'une sélection d'éléments dans X, donc la formalisation naturelle est le choix du type `X -> Prop` en Coq. Cette formalisation est en effet la seule qui permettrait de définir l'ensemble `Prop` comme un treillis complet dont l'ordre $P \leq Q$ est donné par $P \to Q$ et le sup d'un ensemble $\mathcal{P}$ est donné par $\bigvee \mathcal{P} = \exists P \in \mathcal{P}, P$. Car on doit par exemple pouvoir y définir le sous-ensemble des propositions décidables, qui est lui même indécidable (problème de la décision).

Or ce simple choix ne permet plus de formaliser le treillis complet à deux éléments constitué des booléens `bool` muni de l'ordre codefalse $\leq$ `true`. Nous aimerions en effet définir sur cette structure le sup d'un sous-ensemble $S \subseteq X$ comme suit : `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`). Or on ne sait pas décider une proposition en général.

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 $\vee$ 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}
```

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. Sur le cas précis des booléens en tant que treillis complet, nous voudrions travailler dans `bool` au lieu de `Prop` 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 de sorte à pouvoir 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 3.2. On obtient ainsi des objets des types suivants :

```
leq : X → X → B
Sup: (X → B) → X
```

2.2 Minimalité

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 par passage de deux ensembles finis `X, Y` vers l'ensemble `X -> Y`. Nous en parlerons plus en détail dans la section 6.2.

2.3 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 $F(x2) = x2$. On obtient bien x_1 en point fixe minimal par calcul de Coq.

3 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.

3.1 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 5.1.
- `Parametrized_CPO.v` contient une première version de la bibliothèque des CPOs paramétrisée par un ensemble de valeurs de vérité, appelé "B" tout au long du projet. Il sera détaillé en 5.2.

Quant 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 6.1 et 6.2.
- `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 6.3.
  Il contient également un contre-exemple à l'erreur du livre prouvé en Coq, via le lemme `top_of_PO_is_not_minimal`, et un calcul concret de point fixe utilisant les méthodes du théorème II.

3.2 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 : $\backslash/$, $/\backslash$ et `->`, de manière à ce que qu'ils se comportent comme attendu avec l'évaluation `is_true`. Par exemple pour `BFalse` et $/\backslash$, 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 (cf 5.2), 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 6.2.

```
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};
```

On considère maintenant la projection TBody comme une coercion de types et on se permet d'écrire simplement `V` pour accéder aux ensembles d'un type valide, plutôt que `TBody V`. 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) : ((V → B) → B);
BForall_spec (V : valid_type) : forall (P : V → B),
  (forall x, is_true (P x))  ↔  is_true (BForall V P);
```

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 A.

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 [11], 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.

3.3 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 commençons par munir notre ensemble d'un pré-ordre `leq`.Comme on le définit à valeurs dans B, il faut encore rajouter l'évaluation `is_true` un peu partout pour pouvoir passer dans les Propositions et énoncer des propriétés à prouver, par exemple pour en faire des relations d'ordre.

```
Class B_PO := {
    leq: X → X → B;
    Preorder_leq :> PreOrder (fun x y ⇒ is_true (leq x y));
   [...] (* quelques manipulations d'egalite ad hoc *)
  }.
```

Pour la suite de ce rapport, on fait de l'évaluation `is_true` une coercion implicite de `B` vers `Prop` pour s'épargner de l'écrire et alléger les notations. On écrira alors simplement `x <= y` pour l'évaluation de `leq`, par exemple.

Maintenant, on peut définir nos ensembles dirigés en traduisant la définition 1.4, puis les structures de CPO et de treillis complet par dessus une structure d'ensemble ordonné.

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

```
  }.

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

Première version totalisée :

Pour la formalisation du sup sur les CPOs, une première tentative visait à en donner une définition plus simple à formaliser et à manipuler. Nous proposions une fonction sup totalisée, définie sur tous les sous-ensembles au lieu de se restreindre aux sous-ensembles dirigés mais spécifiée uniquement sur ces derniers. Ceci permettait notamment d'éviter l'utilisation de types dépendants nécessaire à la définition de sous-ensembles dirigés. Cette méthode n'est pas nouvelle, c'est celle qui est utilisée par exemple par la division dans Coq, définie partout mais non spécifiée sur 0.

La toute première représentation du sup d'un CPO avait donc simplement pour type `sup : (X -> Prop) -> Prop`. Malheureusement, cette représentation ne permettait pas de définir certaines notions pourtant basiques, comme le CPO des fonctions monotones reliant deux CPOs. En effet, dans le CPO $\langle P -> Q \rangle$ des fonctions monotones de P dans Q avec P et Q deux 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 dans le cas où il n'est pas spécifié. Donc, nous ne pouvons pas établir que la définition toute entière est bien typée.

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

3.4 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, chaînes. 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 `P0` 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).

4 Les preuves des théorèmes de point fixe

Détaillons maintenant un peu les mécanismes derrière chacun des trois théorèmes de point fixe, en Coq. Dans toute cette section, P est un CPO et $F : P \to P$ est une endofonction sur P.

4.1 Théorème I

Dans le cas où F est continue, le point fixe minimal est donnée par la formule : $\alpha := \bigsqcup_{n\geq 0} F^n(\bot)$. Il est relativement simple de prouver ce résultat dans `Prop`, en suivant la preuve intuitionniste donnée [3, page 183].

Voici ci-dessous les grandes lignes de formalisation de ce théorème dans le fichier `Basic_CPO.v`, dépourvues de leurs preuves par souci de clarté. Le code complet se trouve aux lignes 268 à 327 (pour la partie travail sur l'ensemble $\{F^n(\bot) \mid n \in \mathbb{N}\}$), puis 663 à 696 pour le théorème I à proprement parler.

```
Fixpoint itere F n x0 : X :=
  match n with
  | 0 ⇒ x0
  | S m ⇒ F (itere F m x0)
  end.

Variant iteres F : X → Prop := fun x ⇒ exists n, x = itere F n bot

Program Definition a := (sup (exist _ (iteres F) _)). (* L'enrobage de
(iteres F) est àl pour en faire un ensemble édirig, le second
underscore remplace une preuve de son ècaractre édirig (omise). *)
cache une preuve du fait que (iteres F) est bien un ensemble édirig *)
Theorem Fixpoint_I_ii : Continuous F → is_least (Fix F) a.
```

En revanche, dans un ensemble de valeurs de vérité quelconque B (typiquement `bool`), on ne peut plus définir l'ensemble `iteres` $= \{F^n(\bot) \mid n \in \mathbb{N}\}$ avec le type `X -> B` en l'énonçant comme la Proposition `exists n, x = itere F n bot` (ou encore `forall n, iteres F (itere F n bot)` comme dans le fichier `Basic_CPO.v`). En particulier, dans `bool`, on aurait besoin d'un opérateur `Forall` sur l'ensemble infini des entiers, qui n'est pas un type valide avec la façon dont nous avons défini nos valeurs de vérité booléennes.

En effet, on ne peut pas avoir de `Forall` calculable sur un ensemble infini a priori, d'où l'impossibilité de définir notre ensemble `iteres : X -> B` dans le cas où `B` est `bool`.

On pourrait tout de même remarquer que l'ensemble `iteres` est inclus dans X donc son cardinal est fini. On sait alors qu'un algorithme qui itère F sur $\bot$ dans le cas où X est fini construit une suite stationnaire, qui stagne en un nombre fini d'étapes. Le problème est que ce nombre d'étapes n'est pas connu à l'avance et $\mathbb{N}$ est trop grand. Il faudrait indexer l'ensemble `iteres` sur X au lieu de $\mathbb{N}$, mais je n'ai pas trouvé de moyen de le définir ainsi en Coq, avec les seuls opérateurs dont nous disposons, et surtout dans le cas `B` général où aucune hypothèse n'est avancée. Et de toute façon, le théorème II est impliqué par le théorème I, qui fournit une preuve intuitionniste **et** calculable effectivement. Cette tentative de formalisation a donc été abandonnée.

4.2 Théorème II : Pataraia

Dans cette sous-section, on considère une fonction F monotone.

Pour les théorèmes suivants, les preuves exploitent un sous-ensemble de X particulier noté P_F. Il s'agit du plus petit sous-CPO de X qui soit F-**invariant**, i.e. tel que $F(P_F) \subseteq P_F$. Cet ensemble est donné par la formule :

$$P_F = \bigcap_{\substack{Y \subseteq X \text{ sous-CPO} \\ Y \text{ F-invariant}}} Y$$

L'idée de la preuve intuitionniste du théorème de Pataraia est d'exploiter le fait que toutes les endofonctions monotones et progressives sur un même CPO ont un point fixe commun. Plus précisément, l'ensemble des fonctions monotones sur un CPO est un CPO et le sous-ensemble des fonctions monotones et progressives est un ensemble dirigé dans ce CPO ; on peut donc considérer la fonction $H_Q = \bigsqcup\{F : Q \to Q \mid F \text{ monotone et progressive}\}$. Alors $\forall x \in Q, H(x)$ est un point fixe de toute fonction monotone et progressive sur Q.

Avec un peu de travail, on peut alors prouver que F est non seulement monotone mais aussi progressive sur P_F, et ainsi que $\mu = H_{P_F}(\bot)$ est un point fixe de F. De plus, on peut montrer que ce point fixe est à la fois le Top $\top$ de P_F et le point fixe minimal de F. Remarque : on s'éloigne un peu ici de la preuve du livre de référence, voir le code Coq pour plus de détails.

La formule obtenue pour le point fixe μ est entièrement calculable dans les cas finis, car on peut déterminer concrètement P_F, l'ensemble des fonctions de $P_F \to P_F$ monotones et progressives et donc son sup, par énumérations. P_F est de loin l'étape de calcul la plus longue à effectuer, d'où l'ajout d'une mémoïsation que nous avons déjà abordée plus haut.

La première version de la bibliothèque suivait fidèlement le schéma de preuve ci-dessus en définissant le CPO des fonctions monotones, puis l'ensemble dirigé des fonctions progressives pour en prendre le sup (lignes 494 à 533, section `Increasing_fixpoint` dans le fichier `Basic_CPO`). On définit ensuite le sous-CPO P_F, qu'on utilise comme CPO de départ de nos endofonctions monotones et progressives pour obtenir le point fixe.

Cette méthode n'était plus possible dans les versions ultérieures de la bibliothèques, avec la paramétrisation par les valeurs de vérité `B`, car la définition d'un sous-CPO en tant que CPO (et en particulier en tant que type ordonné) nécessite l'utilisation de types dépendants. Or nous voulions éviter d'avoir à re-définir les types dépendants dans `B`.

Illustrons le problème en détaillant un peu. Soit X un CPO, et $Y \subseteq X$ un sous-CPO de X, i.e. Y contient $\bot$ et le sup de tout ensemble dirigé de X inclus dans Y est contenu dans Y. Soit $D \subseteq Y \subseteq X$ un ensemble dirigé (dans Y). Pour définir Y en tant que CPO, il faut déjà définir Y en tant que type. Un élément de Y est défini comme un élément de X muni de la propriété `Y x` (i.e. $x \in Y$ dans `B`). D'où un premier type dépendant, qui ne pose pas encore problème à transposer dans `B` en prenant son évaluation dans Prop `is_true (Y x)`.

```
  Definition set_type (Y : X → B) : Type := { x : X | Y x}.
(* deux lignes pour s'epargner d'extraire l'element de sa preuve ensuite :*)
  Definition element Y (y :set_type Y) := proj1_sig y.
```

```
#[global] Coercion element : set_type >-> X.
```

Ensuite, il faut pouvoir définir le sup de D. Comme D est dirigé dans Y, il est dirigé dans X et on voudrait prendre son sup en tant que sous-ensemble dirigé de X. Mais D est défini comme un sous-ensemble de Y, donc est de type `D : Y -> B`. Pour en prendre le sup dans X, il faut compléter cette fonction en un objet de type `D' : X -> B` donné par `complete_body Y D`. Alors on aimerait définir `(D' x)` comme vrai à la double condition que `x` est dans Y et que x **(en tant qu'élément de Y)** est dans D'. En voici le code sans paramétrisation, dans `Basic_CPO` :

```
Definition complete_body {Y : X → B} (D : (set_type Y) → B) : X → B :=
  (fun x ⇒ {is_in_Y : Y x & D (exist _ x is_in_Y)}).
(* [...] *)
Program Instance subCPO (Y: X → B) (H : is_subCPO Y) : (CPO (subPO Y)) :=
  {| sup D := sup (exist (Directed leq) (complete_body D) _) ; |}.
```

Pour adapter ces définition, il nous faudrait définir `D'` à image dans `B` au lieu de `Prop`, et donc munir `B` d'un opérateur "et" dépendant, dont la deuxième condition dépende de la première, comme `&` dans `Prop`.

Pour contourner ce problème, nous avons modifié la preuve du théorème de Pataraia. Une version sans la paramétrisation peut-être trouvée section `thm_no_subCPO`, lignes 917 à 983 du fichier `Basic_CPO`. Version finale lignes 871 à 971 du fichier `CPO.v`. L'idée est de travailler directement dans X sur les ensembles :

$$E_{Y,z} = \{h(z) \in X \mid h : Y \to Y \text{ monotone progressive}\}$$

où les fonctions $h : Y \to Y$ sont vues comme des fonctions de $X \to X$ qui vérifient $\forall x \in Y, h(x) \in Y$ (i.e. Y est h-invariant). On s'intéresse alors à $E_{Y,\perp}$ et on montre que c'est un sous-CPO (i.e. il contient $\perp$ et les sup de ses ensembles dirigés). Pour cela on s'appuie sur la fonction monotone progressive $h : x \mapsto \bigvee E_{Y,x}$, cf lemme `set_of_fun_is_subCPO`, d'où la nécessité de considérer un z quelconque dans ces ensembles et de ne pas prendre $\perp$ directement. Le sup de $E_{P_F,\perp}$ est le point fixe minimal recherché (et $\top$ de P_F).

```
Definition E_Y_z (Y : X → B) (z : X) (* z sera bot *) (x0 : X) :=
  exists (h : X → X), x0 = h z
   ∧  (forall x y, Y x → Y y → leq x y → leq (h x) (h y)) (* h monotone *)
   ∧  (forall x, Y x → leq x (h x)) (* h progressive *)
   ∧  (forall x, Y x → Y (h x))  (* Y h-invariant (h bien édfini sur Y) *)
   ∧  Y z. (* érduit tout à l'ensemble vide si z n'est pas dans Y *)
```

La dernière condition `Y z` est une astuce qui élimine les problèmes dans les preuves par la suite. Si on considère un élément z qui n'est pas dans Y, on met l'ensemble à $\emptyset$ (il n'est pertinent de travailler que sur des éléments dans Y), ainsi on s'assure de toujours travailler dans Y.

4.3 Théorème III : Bourbaki-Witt

Dans cette sous-section, on considère F une fonction progressive sur X (mais pas monotone en général).

Dans ce cas, on ne peut plus utiliser la preuve de Pataraia. On prouve directement que le top $\top$ de P_F est un point fixe, en montrant que P_F est une chaîne, c'est à dire que l'ordre $\leq$ est total sur P_F : $\forall x, y \in P_F, x \leq y$ ou $y \leq x$. Une preuve un peu technique de ce résultat peut être trouvée dans une précédente version du livre de référence , mais aussi de manière très claire dans l'oeuvre de Lang [12]. Je ne vais pas la re-détailler ici, car la formalisation en Coq suit assez fidèlement la preuve sans ajout significatif de ma part, malgré la technicité notamment dans la version paramétrisée.

Comme l'a montré Andrej Bauer dans un de ses papiers [10], le théorème de Bourbaki-Witt n'est pas prouvable en logique intuitionniste, seulement en logique classique. En réalité, on peut affaiblir un tout petit peu l'axiome du tiers exclu et se contenter de vérifier deux propriétés qui en découlent. Premièrement, on veut que l'égalité soit décidable sur X, i.e. pour tout $x, y \in X, x \equiv y \setminus/ y \equiv x$. Ensuite, on utilise le fait que si pour tout $x \in X$ et propriétés `P` et `Q`, on a `(P \/ Q)(x)` vrai, alors soit on sait que pour tout $x \in X$ `P`(x) est vérifié, soit il existe un élément x_0 tel que `Q`(x_0) est vérifié.

```
Definition decidable_weq := forall (x y : X), (x ≃ y) ∨ not (x≃y).

Definition weak_classic_axiom := forall (R P Q: X → Prop),
   (forall x, R x → (P x ∨ Q x))
    → (forall x, R x → P x) ∨ (exists x, R x ∧ Q x).
```

Dans mes premières tentatives de formalisation du théorème III, en essayant de garder la preuve constructive, je me suis intéressée à une autre formulation possible de P_F qui rend plus apparent le fait qu'il s'agit d'une chaîne, comme étant le plus petit ensemble contenant $\bot$ stable par F et par passage à la borne supérieure (cf section `S_chain` lignes 869 à 912, fichier `basic_CPO.v`.) Ou, de manière équivalente :

$$P_F = \{F^\alpha(\bot) \mid \alpha \text{ ordinal}\}$$

où on définit $F^{\alpha+1}(x) = F(F^\alpha(x))$ si $\alpha+1$ est un ordinal successeur et $F^\omega(x) = \bigvee\{F^\alpha(x) | \alpha \leqslant \omega\}$ si ω est un ordinal limite.

```
Inductive PF : X → Prop :=
 | S_bot : PF bot
 | S_succ : forall x, PF x → PF (F x)
 | S_sup : forall (D : directed_set leq), included D PF → PF (sup D).
```

Ainsi, j'espérais à tort pouvoir m'inspirer de la preuve du théorème I pour obtenir une preuve constructive dans `Prop`. Malheureusement, non seulement la manipulation des ordinaux peut être non constructive, mais je n'ai même pas réussi à aboutir à une formulation de l'ensemble P_F indexée par les ordinaux. J'ai tenté de m'inspirer de [13] et [14], mais il s'est avéré difficile de construire P_F progressivement comme une chaîne, en Coq. De plus, la trichotomie sur les ordinaux implique l'axiome du choix dont nous voulons nous passer [15].

5 Représentions alternatives et version précédentes

La bibliothèque proposée a fait l'objet de plusieurs modifications de plus ou moins grande ampleur pendant sa conception. Je me propose d'en dépeindre maintenant les grandes étapes rapidement.

Comme il a déjà été expliqué en 3.3, la première approche a été de définir une fonction `sup` totalisée définie sur tous les sous-ensembles d'un CPO et spécifiée uniquement sur les sous-ensembles dirigés. Après l'abandon de cette tentative, d'autres approches ont été explorées.

5.1 Fonction sup propositionnelle

Un premier moyen de résoudre le problème d'universalité évoqué en 2.1, i.e. d'englober à la fois des structures de CPO infinies (comme `Prop`) et d'autres structures de CPO finies (comme `bool`), a été de définir le `sup` dans Coq comme une proposition mettant en relation un ensemble dirigé D et sa borne supérieure $\bigvee D$, au lieu d'une fonction associant $\bigvee D$ à D. Ainsi `sup D x` est vrai si et seulement si $x = \bigvee D$. C'est d'ailleurs l'idée exploitée actuellement pas la maigre bibliothèque standard sur les CPO, qui donne les principales définitions de la structure : Library Coq.Sets.Cpo.

Cette tentative a abouti et la bibliothèque alternative obtenue se trouve dans le fichier `Propositional_CPO.v`, dont voici la définition d'un CPO :

```
Class CPO (X: Type) `(P' : PO X) := {
    sup: directed_set leq → X → Prop;
    sup_spec: forall D d, sup D d ↔
              (forall z, (leq d z ↔ forall (y:X), D y → leq y z));
    sup_exists : forall D, exists d, sup D d
  }.
```

Cette approche particulièrement simple à la base permet bel et bien d'englober les CPOs finis, contrairement au sup défini comme la fonction `(X -> Prop) -> X`. En effet, prenons l'exemple des booléens en tant que CPO. Pour peu que l'on dispose du tiers exclu (pour pouvoir dire si un élément appartient ou non à un sous-ensemble), il suffit de définir `sup D x` comme la proposition suivante :

```
sup D x := (D true  ∧  x ≃ true)
        ∨  (not (D true)  ∧  x ≃ false)
```

Pour cette raison, j'ai longtemps utilisé cette bibliothèque pour prouver des contre-exemples et d'autres petits résultats sur des CPO finis (à trois éléments par exemple), avant de disposer de la bibliothèque paramétrée. Cette version m'a permis de prouver la validité de mon contre-exemple à trois éléments au fait que le top de P_F n'est pas nécessairement un point fixe minimal, dans le cas d'une fonction seulement progressive (lemme `top_of_P0_is_not_minimal`, section `CounterExample`, lignes 1125 à 1266, réécrit plus tard dans `Applications.v`). Voici le CPO et la fonction en question :

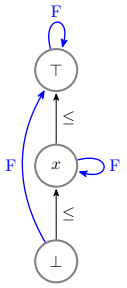

FIGURE 2 – Top de P_F non minimal pour F progressive

J'ai aussi prouvé que le fait que P_F soit une chaîne n'est pas suffisant en général pour conclure à l'existence d'un point fixe (lignes 1272 à 1295), avec le contre-exemple suivant sur le CPO à trois éléments $\{\bot, x1, x2\}$ avec $\bot \leq x1 \leq x2$:

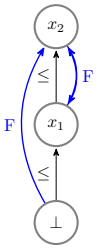

FIGURE 3 – P_F chaîne sans point fixe

Malheureusement, cette approche n'était pas satisfaisante pour plusieurs raisons. Premièrement, avec cette méthode, on perd tout espoir de calculabilité puisque tout vit dans `Prop`, donc rien n'est décidable ou calculable, même le sup d'un ensemble dirigé donné concrètement. Sur un CPO fini, ça signifie d'être incapable de calculer un point fixe concrètement, même avec la méthode fournie par le théorème de Pataraia.

Ensuite, on a besoin pour définir des CPOs très simples (comme `bool` ou le CPO à trois éléments utilisé ci-dessus) d'axiomes dont on aimerait grandement se passer, en l'occurrence le tiers exclu.

Enfin, la manipulation des CPO définis aussi abstraitement est très fastidieuse. La moindre manipulation est laborieuse, et rien que la bibliothèque a été longue et technique à développer. Mais le problème est surtout que les CPOs sont aussi longs et fastidieux à manipuler par les utilisateurs de la bibliothèque, comme j'ai pu m'en rendre compte en travaillant sur mes petits contre-exemples. Pour ne donner qu'un exemple, j'ai dû représenter les fonctions monotones comme des relations (on manipule leurs graphes) au lieu de fonctions

qui fournissent concrètement une image, ce qui rajoute des couches de technicités à toutes les preuves et notions qui les manipulent.

Pour toutes ces raisons, cette approche a été par la suite laissée de côté pour s'intéresser à une première version de CPOs paramétrisés par les valeurs de vérité de la bibliothèque, qui a fini par céder la place à la version finale du projet.

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

Dans notre première tentative de définition d'un ensemble `B` de valeurs de vérité, qui se trouve dans le fichier `Parametrized_CPO.v`, nous avons séparé la structure indépendante des valeurs de vérité, qui contient `BTrue` et `BFalse`, l'évaluation, le "ou", le "et" et l'implication ; d'une structure que l'on vient poser par-dessus qui munit l'ensemble des `Forall` et `Exists` dont nous avons besoin. Le but était de garder un ensemble de valeurs de vérité le plus indépendant possible de l'ensemble de base de notre CPO, or ces deux dernières opérations doivent se faire sur un ensemble à spécifier dans la définition.

Le principal problème est qu'au fur et à mesure du développement de la bibliothèque, de plus en plus d'opérations `Forall` et `Exists` devenaient nécessaires à définir sur des ensembles variés : X, $X \to X$ et les fonctions monotones, les sous-ensembles de X et sous-ensembles dirigés ($X \to B$), et même les entiers si on veut suivre la méthode du théorème I. Pour éviter cette démultiplication d'opérateurs, nous avons d'abord essayé de définir des `Forall` et `Exists` généraux, sur des sous-ensembles de Types en sélectionnant les éléments selon une propriétés. Mais il en restait plusieurs à définir (fonctions et sous-ensembles).

Cette version n'était pas pratique pour les utilisateurs. Elle les force à définir eux-mêmes tous ces opérateurs au moment de créer leur CPO et leur valeur de vérité. Et les opérations sur les valeurs de vérité, omniprésentes, sont également laborieuses à utiliser dans les preuves avec de nombreux soucis de typage et de paramétrisation implicite qui apparaissent à la manipulation conjointe de `B` et X.

Comme nous l'avons déjà évoqué, ce problème a été résolu en rajoutant à notre structure de valeurs de vérité la définition d'un opérateur `K` qui sélectionne les types "valides" (avec un certains nombre de propriétés de clôture) et une définition de `Forall` et `Exists` généraux sur ces types. De plus, ainsi `B` ne dépend plus vraiment de l'ensemble `X` sur lequel on veut définir notre structure ordonnée de CPO ou de treillis, ce qui retire une couche de définition et de complexité notamment dans la manipulation par l'utilisateur et la paramétrisation implicite de Coq.

C'est aussi à partir de cette étape qu'est apparue l'impossibilité de définir un sous-CPO en tant que CPO sans rajouter de types dépendants dans `B`, comme vu plus haut.

6 Application : les CPOs finis

L'un des principaux intérêts de la bibliothèque créée est de pouvoir travailler sur les CPOs finis de manière calculable. Un travail conséquent a été fourni pour définir les ensembles finis et exploiter nos structures d'ordre afin de construire des ensembles finis ordonnés et en démontrer certaines propriétés.

Nous sommes bien conscients que les ensembles finis ont déjà été formalisés en Coq, notamment par le biais de l'extension SSReflect, et de manière plus efficace et minimale que ce qui est proposé dans ce projet (en particulier, sans utiliser l'axiome `functional extensionality`). Cependant, les ensembles finis représentaient une excellente occasion de manipuler la bibliothèque concrètement et la voir à l'œuvre. De plus, ils permettent de faire du projet un bloc autonome et indépendant pour la bibliothèque standard de Coq. Enfin, il s'agit d'un bon exercice de stage de master dans l'apprentissage approfondi de Coq, avec des subtilités techniques de types dépendants entre autres.

6.1 Représentation d'un ensemble fini

Intéressons nous donc au cas des CPOs finis, utilisés conjointement avec les valeurs de vérité `B = bool` pour garder la calculabilité. Ces objets reposent sur une notion d'ensembles finis, dont nous avons donné une

définition et des propriétés dans le fichier `FiniteSet.v`, indépendant du reste du projet et notamment des structures ordonnées.

Nous avons choisi de définir les ensembles finis de sorte que l'égalité soit décidable et que tous ses éléments soient contenus dans une liste (d'où la finitude).

```
Record fin X := {
  eq_dec : forall (a b : X), {a = b} + {a <> b};
  el : list X;
  all_el : forall a, List.In a el
  }.
```

Ces types sont les types valides de notre structure de vérité booléenne, que nous pourrons enfin définir comme ci-dessous. `is_member x l` est une fonction booléenne qui évalue si x est contenu dans l, définie plus tôt dans le fichier, et `el (projT2 X)` est la liste finie énumérant les éléments de X.

```
Program Instance Bool_B : B_param :=
  {|
    B := bool;
    K := fin;
    BTrue := true;
    BAnd := andb;
    BForall V := fun P ⇒ List.forallb P (el (projT2 V));
    memo X P := let l := List.filter P (el (projT2 X)) in
                        fun x ⇒ is_member x l;
    [...] (* code abrege pour la lisibilite *)
  |}.
```

Alors, il devient possible et même facile de définir par exemple un ensemble à deux éléments, comme `bool`, en tant que CPO. Comme expliqué en 2.1, C'est une définition qu'il n'était pas possible de faire dans `Prop`.

```
Program Instance B_CPO_bool : B_CPO B_PO_bool :=    {| sup D := D true; |}.
```

Ou encore un CPO à trois éléments $\bot \leq x_1 \leq x_2$ en ajustant simplement la définition du sup :

```
Definition sup_ex (D :@directed_set Bool_B CPO_valid_type leq3) :=
   if (D x2) then x2 else (if (D x1) then x1 else bottom).
```

Revenons un peu plus en détail sur la définition de l'opération de mémoïsation `memo` sur X. Cette opération est utilisée pour améliorer le temps de calcul concret d'un point fixe minimal donné par le théorème II. On pré-calcule et stocke le résultat du prédicat `P` sur X à l'avance, pour le réutiliser plus tard sans le recalculer à chaque étape. Ici comme `P` est de type `X -> bool`, il s'agit d'un sous-ensemble dans notre formalisation. On stocke donc en fait le sous-ensemble `P`, ou plus précisément sa valeur de vérité sur chaque élément du CPO, i.e. les résultats du booléen `P x` pour tout $x \in X$ dans la liste finie des éléments de X. Dans ce projet, on utilise la fonction memo pour pré-calculer le sous-ensemble P_F, qui est de loin le plus long de tous les calculs à effectuer.

La première version de la formule du point fixe (sans optimisation de ce genre) prenait environ 14s à calculer sur un CPO à trois éléments, un temps beaucoup trop long pour un si petit CPO. Par cette méthode, une accélération conséquente a été obtenue, réduisant le temps de calcul à quelque chose de quasiment immédiat (clairement moins d'une seconde).

6.2 Propriétés de clôture

Afin de faire des ensembles finis une famille de types valides pour notre structure de valeurs de vérité `B = bool`, il faut tout de même que ces propriétés vérifient les spécificités de `K`, à savoir que si V_1 et V_2 sont deux

ensembles finis par la définition ci-dessus, et P est une proposition sur V_1 (`P : V1 -> B`), alors $V_1 \to B$, $V_1 \to V_2$ et { v : V_1 | `is_true(P v)` } le sont aussi.

C'est là (et uniquement là) que nous avons besoin de l'axiome d'extensionnalité fonctionnelle donné en 2.2. En effet, pour prouver que l'égalité est décidable sur les fonctions $V_1 \to V_2$, il faut pouvoir manipuler l'égalité entre deux fonctions de ce type et se ramener à des égalités qu'on sait décidables sur V_1 et V_2. Il en va de même dans les preuves de finitudes : pour montrer que toutes les fonctions peuvent être mises dans une liste, on a besoin de manipuler des égalités de fonction.

Le fichier `FiniteSet.v` est relativement technique. Pour prouver les finitudes, il a fallu d'abord définir des algorithmes qui énumèrent toutes les fonctions entre deux types finis $V_1 \to V_2$ (ou de $V_1 \to B$ pour commencer plus simplement) et les stockent dans une liste. Le cas des ensembles vides était délicat à traiter, car il existe une unique fonction $\emptyset \to V_2$ pour tout ensemble V_2, mais aucune fonction de $V_1 \to \emptyset$ si $V_1 \neq \emptyset$. L'algorithme d'énumération est constitué des fonctions `add_last_image`, `build_fun_opt` et `build_fun` (lignes 116-151). Sa preuve de correction est une obligation de la définition de l'ensemble fini des fonctions, lignes 169 à 230, et s'appuie sur plusieurs lemmes précédents.

Un point qui rend ces preuves très techniques est la manipulation de types dépendants, puisque les objets que nous construisons dépendent de propriétés sur nos arguments, donc de preuves, qui se retrouvent dans le typage des *matching*. Par exemple, Le cas des ensembles finis nous oblige à examiner l'égalité entre notre liste `(el Cfin)` et la liste vide, pour construire l'unique fonction qui part de l'ensemble vide si besoin, et sinon énumérer les possibles images des éléments de la liste. Voici un échantillon de manipulation de types dépendants dans un *matching* (lignes 146-149) :

```
(match (el Cfin) as l return (el Cfin = l) → list (C → D) with
            | nil ⇒ (fun Heq ⇒  cons (@empty_fun C D Cfin Heq) nil)
            | c :: qc ⇒ fun _ ⇒ nil
            end) eq_refl
```

Le problème est que l'évaluation de ce genre de *match* est rendue difficile par le fait que Coq ne peut pas automatiquement remplacer une expression par son évaluation si le type de l'objet évalué est utilisé ailleurs. Il faut donc d'abord prouver que les deux types sont bien égaux avant de les remplacer l'un par l'autre pour pouvoir évaluer, et le tout est très fastidieux. Un exemple illustrant ce problème plus en détail est donné en annexe B.

Les preuves font par ailleurs intervenir des résultats intermédiaires techniques de décidabilité et de *proof irrelevance* (le fait que deux preuves d'une même propriété sont nécessairement égales, autrement dit peu importe la preuve qu'on choisit), comme ceux ci-dessous :

```
Lemma In_is_decidable : forall (l : list A) (a : A), {In a l} + {not (In a l)}.

Lemma Is_true_proof_irrelevance : forall b (p1 p2 : Bool.Is_true b), p1 = p2.
```

6.3 Ordre partiel fini

Au cours de mon stage, j'ai eu l'occasion d'utiliser ma bibliothèque pour prouver un résultat intéressant, et le certifier dans Coq.

Théorème 6.1. *Tout ensemble fini muni d'un ordre partiel et d'un élément bottom est un CPO.*

Cette propriété est prouvée par la définition du CPO `FinitePO_to_CPO` dans la section `FiniteOrder` du fichier `Applications.v`, lignes 128 à 494. C'était bien plus délicat à démontrer que l'on peut croire de prime abord, notamment pour trouver le sup d'un ensemble dirigé et en prouver les propriété.

6.3.1 Algorithme utilisé pour obtenir le sup

Pour trouver le sup d'un sous-ensemble $Y \subseteq X$ dirigé, on parcourt chacun des éléments de Y (i.e. les éléments x de X fini, énumérés dans sa liste, qui satisfont la propriété `Y x`) et on maintient à jour une liste

`candidats` des éléments maximaux rencontrés. Pour se faire, à chaque nouvel élément rencontré, s'il n'y a pas d'éléments plus grand que lui dans la liste de `candidats`, on l'ajoute ; et on enlève de la liste tous les éléments qui lui sont inférieurs.

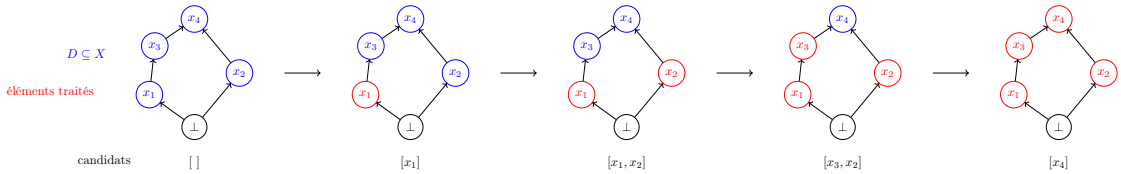

FIGURE 4 – Algorithme de recherche d'éléments maximaux

Cette construction est effectuée par la fonction `build_sup_candidate` et ses deux fonctions auxiliaires précédentes, lignes 134 à 148. Si l'algorithme est simple à comprendre et à formuler, il est difficile de montrer que la liste des candidats qu'on obtient à la fin de son exécution contient bien l'unique sup d'un sous-ensemble, lorsque celui-ci est dirigé et non vide. S'il est vide, son sup est l'élément bottom.

```
sup D := hd bottom (build_sup_candidate D)
```

6.3.2 Preuve de correction

La longue preuve de correction de cet algorithme se base sur plusieurs axes. On montre par étapes que :

1. la liste qu'on construit au fur et à mesure contient uniquement des éléments deux à deux incomparables (lignes 169 à 212)
2. cette liste ne contient que des éléments contenus dans le sous-ensemble $D \subseteq X$ (lignes 215 à 232)
3. la liste ne contient pas de dupliqués, chaque élément y apparaît au plus une fois (lignes 235 à 262)
4. les éléments de la liste `candidate` finale dominent tous les éléments de D, i.e. que tout élément de D est inférieur à un élément de la liste (lignes 266 à 441)
5. la liste finale contient au plus un unique élément si l'ensemble D est dirigé, i.e. sa longueur est au plus égale à 1 (lignes 444 à 472).

Détaillons un peu le point 4., le plus complexe. L'invariant principal que l'on cherche à obtenir est que chaque élément de D déjà traité par l'algorithme est dominé par un élément de la liste de candidats en cours de construction. C'est à dire que si on a fini de traiter les éléments contenus dans une sous-liste `lst`, alors : $\forall x \in D$, soit x n'est pas dans `lst` (il n'a pas encore été vu), soit $\exists y \in$ `candidats`, $x \leq y$.

```
Lemma main_invariant D : forall lst x, Is_true (D x) → not (In x lst)
    ∨  exists y, In y (build_sup_candidate_aux D lst nil)  ∧  x <= y.
```

Ce résultat fait appel à des lemmes assez précis et techniques, comme `In_update_is_commutative` qui établit que cette propriété reste vérifiée peu importe l'ordre dans lequel on examine deux éléments pour mettre à jour la liste des `candidats`, ou encore `build_sup_domination_update_elim` qui établit que si un élément x est dominé par les `candidats` avant de traiter un autre élément, il reste dominé après la mise à jour due à ce nouvel élément, et de nombreux autres.

Avec tous ces résultats intermédiaires, on peut enfin construire un CPO par dessus un ordre partiel fini dont la seule hypothèse est l'existence d'un élément bottom donnée par :

```
Variable bottom : X.
Hypothesis bottom_is_bot : forall (x : X), bottom <= x).
```

Conclusion

Pour résumer, le stage a abouti à une bibliothèque modulaire et quasi minimale en terme d'axiomes permettant de formaliser les ordres et deux structures ordonnées essentielles : les CPOs et les treillis complets.

Par le biais de la définition de structures de vérité définies par la bibliothèque (ou par l'utilisateur), le projet englobe des ensembles et des visées très larges, allant des Propositions infinies avec leurs éléments indécidables aux ensembles finis et calculables. De plus, quelques structures exemples ont été proposés, et certains résultats ont été démontrés en Coq à l'aide de cette bibliothèque.

Pour aller plus loin, des pistes supplémentaires à explorer ont été envisagées. D'abord, il reste le problème de la nécessité d'utiliser l'axiome `functional_extensionality` pour la manipulation des ensembles finis. Pour améliorer la bibliothèque à ce sujet, nous aurions pu tenter de passer à SSReflect, un sous-langage de Coq particulièrement adapté à la manipulation de telles structures. Une autre solution envisagée que nous aurions mise en place si nous avions eu plus de temps est une restructuration des valeurs de vérité `B` pour y intégrer par exemple une notion d'égalité définie par l'utilisateur ou la bibliothèque, avec ses propres propriétés.

On pourrait aussi chercher du côté des ensembles de valeurs de vérité qui seraient pertinents à ajouter, ou modifier la définition de `B` de sorte à pouvoir travailler dans des logiques à trois éléments ou à un nombre fini d'éléments distincts de `True` et `False`. Nous avons tenté des approches permettant de transformer un CPO fini quelconque en ensemble de valeurs de vérité, sur lequel nous pourrions définir de nouveaux CPOs à leur tour, mais ces tentatives n'ont pas abouti. En l'état, il reste à explorer la possibilités d'ensembles de valeurs de vérités infinies distinctes de `Prop`.

Enfin, ce sujet appelle à un sujet de thèse bien plus vaste dans lequel les ordres et notamment les CPOs pourraient être utilisés conjointement avec la théorie des catégories pour améliorer le noyau de Coq, notamment en ce qui concerne la coinduction et ses failles.

Références

[1] Neil Robertson, Daniel P. Sanders, Paul Seymour, and Robin Thomas. A new proof of the four-colour theorem. *Electronic Research Announcements of the American Mathematical Society, Volume 2, Number 1*, pages 17–25, August 1996.

[2] https ://coq.inria.fr/distrib/current/stdlib/coq.sets.cpo.html.

[3] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2 edition, 2002.

[4] Robert Dockins. Formalized, effective domain theory in coq. 07 2014.

[5] Damien Pous. Coinduction All the Way Up. In *Thirty-First Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2016)*, New York, United States, July 2016. ACM.

[6] Damien Pous. https ://github.com/damien-pous/relation-algebra.

[7] Hervé Grall. Proving Fixed Points. working paper or preprint, July 2010.

[8] Mengqiao Huang and Yuxi Fu. A note on the knaster–tarski fixpoint theorem. *Algebra universalis*, 81, 11 2020.

[9] https ://github.com/gabzcr/coq-cpos/tree/master.

[10] Andrej Bauer and Peter Lumsdaine. On the bourbaki-witt principle in toposes. *Mathematical Proceedings of the Cambridge Philosophical Society*, 155, 01 2012.

[11] Katalin Varga, Gábor Alagi, and Magda Várterész. Many-valued logics – implications and semantic consequences. *Acta Universitatis Sapientiae. Informatica*, 5, 12 2013.

[12] Serge Lang. *Algebra*. Springer, New York, NY, 2002.

[13] Pierre Castéran, Jérémy Damour, Karl Palmskog, Clément Pit-Claudel, and Théo Zimmermann. Hydras & Co. : Formalized mathematics in Coq for inspiration and entertainment. working paper or preprint, October 2021.

[14] José Grimm. Implementation of three types of ordinals in coq. 01 2013.

[15] Erwin Engeler. *Axiom of Choice and Continuum Hypothesis*, pages 37–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

A Code : définition complète des valeurs de vérité B

```
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);
  }.
```

Pour les explications sur la fonction de mémoïsation `memo`, voir 6.1.

B Matching de types dépendants

Regardons l'example suivant, issu d'une preuve Coq du fichier `FiniteSet.v` :

```
1 subgoal
A : Type
B : Type
Afin : fin A
Bfin : fin B
a : A -> B
EQB : el Bfin = nil
EQA : el Afin = nil
______________________________________(1/1)
In a
  (match el Afin as l return (el Afin = l -> list (A -> B)) with
   | nil => fun Heq : el Afin = nil => empty_fun Heq :: nil
   | c :: qc => fun _ : el Afin = c :: qc => nil
   end eq_refl)
```

Dans cet exemple, on sait que `(el Afin) = nil` par l'hypothèse `EQA`. On aimerait donc simplement évaluer le goal, car on sait être dans le cas `nil` qui donne `fun Heq : el Afin = nil => empty_fun Heq : : nil`. Or les tactiques comme `simpl` ou `cbn` échoue. Une tentative de remplacer `(el Afin)` dans le code par la tactique `rewrite EQA` (ou `induction EQA`), pour forcer le match à s'évaluer, donne l'erreur suivante :

```
Abstracting over the term "l" leads to a term [...] which is ill-typed.
Reason is: Illegal application:
The term "@empty_fun" [...] cannot be applied to the terms [...].
The 4th term has type "el Afin = l0" which should be coercible to "empty Afin".
```

Or ici on sait que dans le cas où cette égalité est utilisée, `l0 = nil` et même directement que Afin est vide car par l'hypothèse EQA, `el Afin = nil`. Ici, pour contourner le problème, j'ai réécrit le goal de sorte d'y faire disparaitre Heq, en le remplaçant par une hypothèse `"Em : empty Afin"` facile à prouver. Ainsi, on peut faire le *rewrite* sur EQA sans plus de problème. Le morceau de preuve analysé ici se trouve aux lignes 175 à 182 du fichier `FiniteSet.v`.

Parfois, le conflit n'est pas aussi simple à résoudre que dans ce premier cas...