Je pense avoir réussi à implémenter la plupart des fonctionnalités importantes.

Sur la première partie du projet, j'ai implémenté par nécessité jusqu'à la définition de l'inférence de type. Pour l'inférence de type, il a fallu faire plusieurs choix d'implémentation. Dans un premier temps, j'ai voulu faire fonctionner le type de l'application d'une abstraction à un terme. Il a fallu rendre possible la vérification de l'égalité entre deux types, et j'ai décidé de la faire de manière simple, sans notion de béta réduction, ou d'alpha-équivalence.

Commenté [EMX1]: pak

L'égalité entre deux types se définit simplement de manière récursive, avec comme cas de base l'égalité entre les variables, si elles portent le même nom.

La fonction ***infer_type*** permet d'implémenter certaines règles d'éliminations, App(Abs(« x », y, Var(« x »)) sera de même type que y.

Il aurait été possible implémenter les beta-réductions, et de typer les objets en les réduisant, ce que j'ai fait dans la deuxième partie du projet, mais ici les règles sont simplement implémentées dans ***infer_type***.

Conjonction

J'ai décidé de créer un type ty Conj of ty * ty. Cela permettait d'implémenter facilement les règles d'introduction et d'élimination.

Dans le premier projet, l'objectif étant toujours de bien implémenter les règles d'introduction et d'élimination, j'ai toujours essayé de choisir une façon qui rendait cette implémentation naturelle.
Dans l'idée d'un prover, nous avons besoin de définir la conjonction telle que prouver A/\B revient à prouver A et à prouver B.

Ainsi, un objet qui prouve A/\B doit être une preuve de A et également une preuve de B, et on doit pouvoir en extraire chacune de ces preuves.

J'ai choisi d'utiliser une implémentation sous la forme d'une paire, un objet du type A/\B est une paire x = <π_l(x) , π_r(x)> avec π_l(x) :A et π_r(x) : B. Cela permet également, lors de l'utilisation d'une conjonction comme hypothèse, de pouvoir « récupérer » une preuve de A et une preuve de B.

Une preuve de A/\B => A donne donc :
P = fun(x : A/\B)=> π_r(x)
Une preuve de A/\B => B/\A
P = (fun : (x : (A /\ B)) -> <π_r(x) , π_l(x)>) (and_comm)

J'ai au début eu un peu de mal à comprendre l'abstraction type / proposition et objet/preuve. Cela m'a un peu freiné dans l'élaboration du projet car les instructions étant brèves, il était nécessaire de bien comprendre les objectifs et la théorie derrière.

Commenté [EMX2]: je pense que j'écrirais pas ça

Le type / terme Truth

Pour la vérité, j'ai rajouté un type Truthty ainsi qu'un terme Truth. Il est nécessaire que Truth soit toujours prouvable, et qu'il soit ainsi toujours possible de construire un objet de type Truth, peu importe le contexte. En particulier infer_ty(Truth) = Truthty de manière invariante.

```
test = Abs("x", Impl(Truthty, Vari("A")), App(Var("x"),Truth));;
```

est de type : ((⊤ ⇒ A) ⇒ A)

On ne peut pas prouver ((B ⇒ A) ⇒A), car on ne pas construire d'objet de type B. Cela montre la spécificité du terme truth, qui est toujours de type Truthty. Truthty est donc toujours prouvable tant que l'on peut exhiber le terme truth.

Truth est toujours un élément du contexte.

Cela sera implémenté plus tard en utilisant les règles d'introduction dans le prover interactif.

Falsity -

La règle à implémenter est le principe d'explosion, ⊥ => A pour tout A.

Pour ceci, j'ai décidé d'ajouter un type « PCase » constructible à partir d'un terme et d'un type (a,b), si le terme a est Faux, alors PCase est de type b.

Pour montrer :
(A∧(A⇒⊥))⇒B

```
let test = Abs("x", Conj(Vari("A"), Impl(Vari("A"), Falsety)), PCase(Vari("B"),
App(Snd(Var("x")) ,Fst(Var("x")))));;
```

Disjonction -

La disjonction a été la plus dure à implémenter, j'ai procédé de la manière suivante :
Ajout d'un type **Disj(a,b)** qui est donc A\/B, un élément de type A\/B est forcément de type A ou de type B.
Il ne s'agit cependant pas de seulement regrouper les éléments de type A et ceux de type B sous une même étiquette, mais de créer un type « superposé ».

Les fonctions Abs(« x », A\/B, f(x)) regroupent plus d'éléments que les fonctions Abs(« x », A, f(x)) et Abs(« x », B, f(x)) réunies.

Du point de vue de la preuve, nous souhaitons pouvoir créer des preuves du type :

A\/B => C

On veut pouvoir « emprunter » différents chemins ***ie*** : Supposons A alors C vrai, Supposons B alors C vrai.
Pour ceci, nous avons besoin de pouvoir créer une preuve (ie un lambda terme) qui corresponde à cette disjonction de cas.

Pour cela, j'ai ajouté un lambda terme « Case » qui permet de faire cette preuve.

On le construit avec Case(x, P(D)|X :A, P(D)|X :B)
Avec P(D) | X :A une preuve de D avec X de type A dans le contexte
Avec P(D) | X : B une preuve de D avec X de type B dans le contexte

Il faut également implémenter la règle d'introduction x : A => x :A\/B
Ainsi, pour prouver A\/B, prouver A est suffisant. Une preuve de A est une preuve de A\/B et de même pour B.

Pour ceci, j'ai rajouté deux types, right et left, si x est de type A right(x,Type) donne un terme de type B \/ A
Et left (x,Type) donne un terme de type A\/B

À partir d'ici, nous pouvons construire un bon nombre de preuves, mais cela est encore très fastidieux, et nous voulons pouvoir implémenter des « tactiques » qui correspondent à certaines règles d'éliminations et qui permettent d'utiliser des conditions suffisantes à nos preuves.

Le parsing des éléments se fait grâce au code fournit, auquel j'ai rajouté quelques spécificités.

Implications :

Le plus simple à programmer était la tactique d'élimination des flèches des implications. Pour prouver A => B, il suffit d'ajouter A au contexte sous une variable, puis de prouver B à partir du contexte.
Le terme associé sera simplement, Abs(« x », Vari(« A »), Preuve(B|X :A)) sera de type A=>B

Cette règle peut s'appliquer successivement, pour éliminer plusieurs flèches.

Un ajout intéressant dans le programme aurait été la gestion automatique des noms de variables.

Cut elimination :

La tactique de cut elimination a été plutôt simple à implémenter, nous voulons montrer C, nous montrons B => C puis B.
Le terme associé sera App(Preuve(B=>C), Preuve(B))

J'ai ajouté de manière similaire des stratégies pour faire des preuves sur des conjonctions et des disjonctions. L'élément Truth peut être ajouté au contexte en tapant intro () on peut tout prouver en faisant elim x lorsque x est de type False.

Les entiers naturels :

Les entiers naturels sont définis à partir du zéro et des successeurs. Un entier naturel est un Zéro ou un successeur d'un autre entier naturel.

On peut également définir des relations entre des entiers naturels, notamment des fonctions.
Nous souhaitons pouvoir définir des fonctions de manière récursives.

Rec(n, u, xy, v), est de même type que u. Nous pouvons à partir de cet élément construire les fonctions récursives primitives. Nous travaillerons plus en détails ces fonctions dans la deuxième partie du projet.

Sans réduction, je n'ai pas explicité la nature de Rec(n, u,xy,v) dans mon code, mais j'ai simplement ajouter une manière de construire ces éléments. Par exemple, nous pouvons montrer que l'élément Rec(n, Zero, xy, x) est un entier naturel pour tout N, en vertu du principe de récurrence grâce au mot « elim ». Nous nous restreignons à la constructibilité des termes, à partir des sucesseurs, des zéros et de Red.

On le construit dans le fichier pred.proof

DEUXIEME PARTIE

La deuxième partie reprend la première sans la partie de stratégie de preuve mais en rajoutant les réductions, les équivalences et les types dépendants.

Le plus important était l'implémentation de la substitution dont tous les autres mécanismes ont découlé. Ici, on peut prouver plus de choses, on peut prouver tout ce qu'on pouvait prouver avant mais également, add 3 5 = 8 car on peut réduire add 3 5. Les preuves peuvent également se calculer à partir des réductions.
On peut également montrer des choses sur des types dépendants, c'est-à-dire sur des types qui dépendent d'un paramètre.
Cela permet de démontrer des résultats plus généraux, notamment des propriétés P(x). Le type P(x) existant Pi(x :A) -> P(x)

Il a fallu également implémenter les opérateurs de récurrence Ind et celui d'égalité J. Ind permet de construire des preuves par récurrence à partir d'une preuve de P(0) et d'une preuve de P(n) => P(n+1)

J quant à lui, permet de montrer x=y => P(x,y) à partir d'une preuve de P(x,x) et d'une preuve d'égalité entre x et y.

Ces principes ne peuvent être implémentés que grâce à la réduction et contrairement à la première partie, nous pouvons désormais faire des preuves par récurrence, et montrer des égalités complexes.

L'égalité entre deux termes n'est plus seulement définitionnelle comme c'était le cas avant, mais également d'un point de vue d'une équivalence.

Je me suis arrêté ici car je pense que cette partie était celle qui apportait le plus de compréhension à la théorie, et que refaire le travail sur les techniques d'élimination relevait d'un travail technique compliqué.

Ce que je n'ai pas réussi à prouver :

- `impdm.proof: ((not A) \/ B) => A => B`
- `russel.proof: (A => not A) => (not A => A) => _`

Pour ces preuves, j'ai l'impression d'avoir besoin de la disjonction A\/(A=>_). Je n'ai pas implémenté celle-ci, que j'aurais pu je pense ajouter. Elle m'aurait permis d'effectuer ces deux preuves, mais je ne savais pas si cela était nécessaire.

Je n'ai pas eu le temps de trouver les termes qui permettaient de prouver les dernières questions, je pense que le prover devrait fonctionner en trouvant les bons termes.

```
? check zadd = Pi (n : Nat) -> Eq (add Z n) n
Ok.
? check addz = Pi (n : Nat) -> Eq (add n Z) n
```

- Show that addition is associative and commutative.
- Define multiplication and show similar properties.