Relecture PComp

fanxiang Zeng 28600693

- language de relecture : Java
- language utilisé pour l'implémentation : Python

Introduction

Dans le cadre de l'ue programmation comparé, nous devons effectué une relecture d'interpreter Prolog avec un language de programmation différent de celle que nous avons uilisé pour implémenter notre projet. L'objectif de cette relecture est de lire un programme dans des langages différents afin de comprendre et critiquer de façon constructive l'implémentation d'un programme, puis de comparer l'implémentation d'un même projet dans différents langages en terme des structures des données et de l'implémentation des algorithmes.

I. Test et proposition des améliorations du code sur les 4 jalons du projet de relecture

jalon 1

Ce jalon consiste à, dans un premier temps, proposer deux structures de donnés adapté pour représenter les `Equation` et les `Environnement` du prolog, qui permettront dans la suite à l'implémentation une fonction unification qui consiste à lever une exception, lorsque une equation n'est pas unifiable, ou de renvoyer l'environnment résultant d'une liste d'équation. L'unification est impossible si on ne peut appliquer aucune de ces 4 règles suivantes : effacer, remplacer , orienter et décomposer. Quand on essaye d'ajouter une variable dans l'environnemen et le terme à droite contient la variable en question , l'unification échoue également, cette condition est vérifié par une fonction `OccurCheck` effectuée avant tout remplacement et avant d'ajouter une variable à l'environnement.

1. proposition des amélioration du code

- lisibilité : code bien structuré lisible avec des commentaire explicative sous forme de javadoc , et une class intermédiaire `Système` qui contient la liste d'équation pour faire l'unification, l'environnement de retour et la fonction d'unification. En passant par `System` il suffit donc juste de créer l'équations avec 2 `Predicat` et de l'ajouter au système, et d'appeler la fonction `unify()` de la classe `System` pour obtenir l'environnment final
- élégance :
  - (+) code élégant factorisant le code, avec dans `Equation` les fonctions `remplacer(), orienter(), decomposer(), effeacer()` qui vérifie unification à pour chaque équation en renvoyant un boolean et `subst()` éffectue les substitutions décrite dans l'environnement.Ensuite dans `System`, l'appel de ces fonctions revient à faire une parcours de la liste des Equation, et pour chaque element de la liste on applique les fonctions correspondant. Les exception sont levées lorsque une décomposition est impossible où lors d'un occurcheck. La fonction unification revient simplement à appeler ces 4 fonctions) chaque tour de boucle et si la condition en fin de boucle reste toujours faux, alors aucune regle n'a ete appliquee sur le systeme pendant ce tour.

    - (-) La fonction `subst` étaler un peu vaguement dans les 4 classes `Equation, System ,Predicate ,TermPredicate`, ce qui rend la relecture de cette méthode un peu compliquer.
    - (-) trop de copy fait pour effectuer l'unification
- l'efficacité :
    - choix de faire une class pour `Equation` avec comme attribut `gauche` et `droit` qui sont des `Term`, bonne choix de structure qui permet d'accèder aux attributs via des getteurs en O(1) et gérer les Term gauches et droits via des méthodes dans la classe. C'est un très bon choix de structure de données
    - choix d'utiliser un `HashMap` dans la class `Environnement` pour représenter l'environnement, avec comme clé la variable et comme valeur le terme correspondant. Du coup accès et ajout en O(1) très efficace .

2. test avec des programmes Prolog

```
p(Z, h(Z, W), f(W)) = p(f(X), h(Y, f(a)), Y)
```

en executant ce systeme unifiable simple, nous n'avons pas rencontrer d'exception, ca voudrait dire que le programme fonctionne sur ce test simple. Mais parcontre, on obtient un environnement `X = f(a), Y = f(f(a)), W = f(a) et Z = Y`,ici normalement comme `Y = f(f(a))` le resultat attendue pour Z est normalement `Z = f(f(a)`, il manquait une substitution pour ce cas.

```
q(f(a), g(X)) = q(Y, Y)
```

en executant ce système non unifiable simple nous obtenons une exception tel que`pcomp.prolog.ast.excep.NoSolutionException: D�composition impossible sur l'Equation g(X) = f(a)` ce qui montre le bon fonctionnement de l'unification

```

```

en exécutant ce systeme l'unification échoue, dans le terme à droite on retrouve la variable elle même se retrouve dans le terme à droite, et une exception expliquant erreur est bien levée `pcomp.prolog.ast.excep.NoSolutionException: OccurCheck true : X = f(f(f(X)))`

```

```

un exemple unifiable compliquer, mais la fonction unification fonctionne tourjour bien pas d'exception lever, avec un environnment `U -> Z , X -> r(b, Z), Y -> b, Z -> a` qui manque potentiellement des substitution également car `Z -> a` du coup `X ->r(b,a)` devrait etre le resultat

conclusion sur jalon 1: la fonction `unify()` dans la class `System` fonctionne très bien en levant des exception quand non unifiable et renvoie l'environnement, mais pour l'environnement renvoyer dans le

jalon 1, il manque probablement un dernier remplacement a faire dans l'environnement. Y'a moyen qu'ils ont pas remarqué ce problème dans jalon 1 car ils ont pas affiché l'environnment.

jalon 2

Ce jalon consiste à coder 3 interprete:

- interprete0 : qui prend en argument l'AST d'un programme constitué d'un fait et d'un (seul) but et qui renvoie l'environnement permettant de résoudre le programme en unifiant les deux termes.
- interprete1 : qui prend en argument l'AST d'un programme constitué de plusieurs faits (mais un seul par symbole de prédicat) et d'un but, qui choisit le fait correspondant au symbole de prédicat du but, et qui renvoie l'environnement permettant de résoudre le programme en unifiant les deux termes.
- interprete2 : qui prend en argument l'AST d'un programme constitué de plusieurs faits (un seul par symbole de prédicat) et de plusieurs buts et qui renvoie l'environnement permettant de résoudre tous les buts.

1. proposition des amélioration du code

- lisibilité :
    - (-) jalon2 inclus dans jalon1 qui diminue légèrement la lisibilité
    - (+) Code bien commenté avec un javadoc
- élégance :
    - (+) code élégant en utilisant les travaux du jalon1 dans une nouvelle classe `Interprete`, qui contient le code des interprete 0,1 et 2 sous forme de fonction static. Ce qui faciliteront les appels avec un simple `Interpret.interpret1` par exemple
- l'efficacité :
    - utilisation des d'une classe contenant les interprete qui appel les différent class et méthode faites dans jalon 1. Accès des interprete par des méthodes static en O(1)

2. test avec des programmes Prolog

- interprete 0

```
r(V, W), Y)
q(Z).
p(Z, h(Z, W), f(W)).
?- p(f(X), h(Y, f(a)), Y).
```

en testant avec le programme ci-dessus une `IllegalArgumentException` est levée en indiquant que le nombre de faits est incorrect, il n'en faut qu'un. Cela répond bien au critère de `interpret0`

```
p(Z, h(Z, W), f(W)).
?- p(f(X), h(Y, f(a)), Y), q(f(W)), r(Y, Y, h(Y)).
```

en testant avec le programme ci-dessus une `IllegalArgumentException` est levée en indiquant que le nombre de buts est incorrect, il n'en faut qu'un. Cela répond bien au critère de `interpret0`

```
p(Z, h(Z, W), f(W)).
 ?- p(f(X), h(Y, f(a)), Y).
```

en testant avec le programme ci-dessus un environnement `W -> f(a), X -> f(a), Y -> f(f(a)) ,Z -> Y` sans exception.Interprete0 fonctionne bien avec le même problème dans jalon1 que Y n'a pas été remplacer.

```
q(Z, h(Z, W), f(W)).
 ?- p(f(X), h(Y, f(a)), Y).
```

en testant avec le programme ci-dessus `NoSolutionException` est levée. On peut du coup conclure que `interprete0` fonctionne parfaitement pour tester si un programme est unifiable, mais manque probablement un dernier remplacement pour l'environnement qui n'est pas si gênant pour les utilisation futures

- interprete 1

```
r(X, Y, h(Z)).
q(Z).
p(Z, h(Z, W), f(W)).
 ?- p(f(X), h(Y, f(a)), Y).
```

en testant avec le programme ci-dessus un environnement `W -> f(a), X -> f(a), Y -> f(f(a)) ,Z -> Y` sans exception. Cela répond bien au critère d'interprete1 qui peuvent prendre un programme constitué de plusieur fait.

```
r(X, Y, h(Z)).
q(Z).
p(Z, h(Z, W), f(W)).
 ?- p(f(X), h(Y, f(a)), Y), r(X, Y, h(Z)).
```

en testant avec le programme ci-dessus une `IllegalArgumentException` est levée en indiquant que le nombre de buts est incorrect, il n'en faut qu'un. Cela répond bien au critère de `interpret1`

```
insecte(fourmi).
felin(chat).
animal(humain).
felin(lion).
 ?- animal(X).
```

en testant avec le programme ci-dessus une `IllegalArgumentException` est levée en indiquant que Il faut un fait par symbole de prédicat. Cela répond bien au critère de `interpret1`. On peut du coup conclure que Interpret1 répond au ensemble des critères.

- interprete 2

```
r(X, Y, h(Z)).
q(Z).
p(Z, h(Z, W), f(W)).
 ?- p(f(X), h(Y, f(a)), Y), q(f(W)).
```

en testant avec le programme ci-dessus un environnement `W -> f(a), X -> f(a), Y -> f(f(a)) ,Z -> Y` sans exception. Cela répond bien au critère d'interprete2 qui peuvent prendre un programme constitué de plusieur fait et de plusieur but.

```
insecte(fourmi).
felin(chat).
animal(humain).
?- animal(X), felin(Y).
```

en testant avec le programme ci-dessus un environnement `X -> humain, Y -> chat` sans exception. Cela répond bien au critère d'interprete2 qui peuvent prendre un programme constitué de plusieur fait et de plusieur but.

```
insecte(fourmi).
felin(chat).
animal(humain).
animal(chien)
 ?- animal(X), felin(Y).
```

en testant avec le programme ci-dessus une `IllegalArgumentException` est levée en indiquant que Il faut un fait par symbole de prédicat. Cela répond bien au critère de `interpret1`. On peut du coup conclure que Interpret2 répond au ensemble des critères.

conclusion sur jalon2: l'ensemble des interprete fait dans jalon2 sont correcte, lève une exception lorsque le programme ne réponds pas au critère où non unifiable, et dans l'exception la raison que ça lève une exception est bien expliqué, et quand on a un programme unifiable l'environnement est renvoyé.

jalon 3

Ce jalon consiste à faire la résolution des règle, en implementant les algorithmes `rename, choose , solve` et au final d'implémenter `interprete3` qui prend en argument un programme Prolog qui contient un ou plusieurs buts et une seule règle par symbole de prédicat, et qui renvoie une solution ou lève une exception.

1. proposition des amélioration du code

- lisibilité :
  - (-) code de `choose()` et de `solve()` tous dans interprete avec beaucoup de surchare difficile à repérer quel est le méthode appellé
  - (+) Code bien commenté avec un javadoc
- élégance :
  - (+) code élégant en avec `interprete3` comme un fonction static utilisant les fonction `choose et solve` qui sont dans la class `Interprete`, et `rename` utilisant le design pattern visitor pour produire une nouvelle règle renommer
  - (-) mettre les fonction `choose et solve` en private car ce n'est utilisé seulement dans les interprètes et pas d'autre part
- l'efficacité :
  - Accès d'interprete3 par des méthodes static en O(1), utilisation de visitor qui permet à accédé aux `DeclAssertion et Predicate`. Du coup une implémentation plutot efficace.

2. test avec des programmes Prolog l'ensemble des fonctions sont appellé par l'interprete3 du coup en testant seulement interprete on peut voir si le ce jalon est bien réussi. Du coup ci-dessous sont les tests fait par l'interprete3

```
p(Z, h(Z, W), f(W)) :- r(X, XX), q(XX).
q(a).
r(b, a).
 ?- p(f(X), h(Y, a), Y), q(a).
```

Environnement obtenu : {X=a, Y=f(a)} sans exception, un tel programme avec des règles corps non vide unifiable l'interprete3 a résolu sans problème en renvoyant l'environnement filtré qui ne contient que les variables du buts, cette filtrage a été fait par le méthode `nettoieEnv()` de la classe `Environnment`

```
vote(X, Y) :- majeur(X), nationalite(X, Y).
majeur(X) :- personne(X).
pays(Chine).
nationalite(X, Y) :- personne(X), pays(Y).
personne(Fanxiang).
 ?- vote(X, Y).
```

Environnement obtenu {X=Fanxiang, Y=Chine} en testant avec le programme ci-dessus, `interprete3` fonctionne toujours de manière performante

```
p(Z, h(Z, W), f(W)) :- r(X, XX), q(XX).
 ?- p(f(X), h(Y, a), Y).
```

en testant avec le programme ci-dessus une `NoSolutionException` est levée en indiquant pas d'environnment correspondant pour le but r(X1, XX1), ce qui totalement correcte car lorsque le but s'unifie

avec la tete, le corpsici y'a pas de fait pour résoudre r(X,XX), du coup non unifiable.Cela répond bien au critère de `interpret3`.

```
animal(X) :- mammifere(X).
animal(x) :- insecte(X).
mammifere :- felin(X).
felin(chat).
insecte(fourmi).
 ?- felin(X), insecte(Y).
```

en testant avec le programme ci-dessus une `IllegalArgumentException` est levée en indiquant que Il faut un fait par symbole de prédicat. Cela répond bien au critère de `interpret3`. On peut du coup conclure que Interpret3 répond au ensemble des critères.

conclusion jalon3 : un jalon donnnant des résultats correcte sur l'ensemble des exemples, et implémenté de manière efficace

jalon 4

Ce jalon consiste à implémenter l'algorithme de Backtraking, en créant une structure de données contexte qui contient la liste des buts à résoudre, la suite des règles à explorer et l'environnement. Cette Structure a pour but de mémoriser le contexte courant au moment du choix afin de pouvoir revenir au point de choix précedent. Ce algorithme de backtracking sera utilisé dans le nouveau `interprete4` de ce jalon, qui prend l'AST d'un programme Prolog et qui renvoie une solution sans aucune contrainte.

1. proposition des amélioration du code

- lisibilité :
    - (-) code de `choose()` et de `solve()` tous dans interprete avec beaucoup de surchare difficile à repérer quel est le méthode appellé
    - (+) Code bien commenté avec un javadoc
- élégance :
    - (+) code élégant en avec `interprete4` comme un fonction static utilisant les fonction `choose et solve` qui sont dans la class `Interprete`, et `rename` utilisant le design pattern visitor pour produire une nouvelle règle renommer
    - (-) mettre les fonction `choose et solve` en private car ce n'est utilisé seulement dans les interprètes et pas d'autre part
    - (-) définition d'un nouveau `choose`, redondant au choose du jalon 3, le nouveau `solve()` implementé revient à faire un simple `try catch` sur le nouveau `choose`
    -
- l'efficacité :
    - Accès d'interprete3 par des méthodes static en O(1), utilisation de visitor qui permet à accédé aux `DeclAssertion et Predicate` en O(1). Du coup une implémentation plutot efficace.
    - (+) algorithme de backtracking dans le nouveau `choose` définie de manière arborescent avec des appels récursif, qui pour le premier but, essaie d'unifier avec chaque règle, si ça s'unife avec la tete d'une règle, la méthode ajoute le corps de la règle dans la liste des nouveaux buts du nouveau contexte et enlève le but courrent et fait un appele récursif avec le nouveau

contexte. L'algorithme s'arrete lorsque la liste des buts est vide où bien aucun règle permet faire l'unification à la première appel.**Ce algorithme est optimisé grâce au if qui vérifie si le contexte courrent est déjà dans la liste des Contextes, si il est déja dedans on passe directement au règle prochain. Le if vérifie également si le but et le règle de ce tour de boucle n'a pas le même symbole on passe directement au prochain règle qui permet d'éviter les récursions inutiles. Donc un algorithmes de Backtracking très efficace**

2. test avec des programmes Prolog l'ensemble des fonctions sont appellé par l'`interprete4` du coup en testant seulement interprete on peut voir si le ce jalon est bien réussi. Du coup ci-dessous sont les tests fait par l'`interprete4`

```
p(X) :- q(X).
p(X) :- r(X).
q(a).
g(b)
?- p(b), r(a).
```

en testant avec le programme ci-dessus une `NoSolutionException` est levée en indiquant problème non satisfiable. En effet, lorsque `p(b)` s'unifie avec `p(X)`, `q(X)` est ajouté au liste des buts et dans l'environnement nous avons seulement `X->a`,résultant de l'unification `q(X) = q(a)`, et l'autre fait c'est `g(b)` du coup `p(b)` n'est pas unifiable. L'interprete4 répond au critère lorsqu'il faut levé des exceptions.

```
animal(X) :- insecte(X).
animal(X) :- mammifere(X).
mammifere(X) :- felin(X).
insecte(fourmi).
felin(chat).
 ?- animal(X), mammifere(Y).
```

En testant avec le programme ci-dessus, avec plusieurs plusieurs règles , faits et buts ,on obtient l'Environnement : {X=fourmi, Y=chat} qui donne un environnement correcte.

conclusion jalon4 : algorithme de backtracking bien implémenté et optimisé avec ensemble de test qui fonctionne bien. Suite à ce jalon, `interpret4` peut résoudre l'ensemble des programmes Prolog, qui ne reste plus à réutiliser dans le jalon 5.

conclusion partie 1

une bonne lisibilité avec commentaire sur l'ensemble des jalons malgrés quelque étalements des fonctions et des beaucoup de surcharge dans l'interprete. Code élégant et correcte sur l'ensemble des jalons avec des résultats qui passe sur l'ensemble des tests, qui permet la réutilisation des codes dans le jalon5. Une bonne efficacité avec des temps d'accès en O(1) et des conditions `if` pour évité les itérations inutiles dans des algorithmes compliqué comme le Backtracking. Pour conclure, c'est un très bon projet qui répond au consigne et avec un interpreteur final qui peut interpreter l'ensemble des programmes Prolog.

II. Comparaison des deux projets dans 2 languages différents (python et Java)

###s comaparaison des structures de données

1. Environnment

- En java dans le projet de relecture, l'environnement a été représenté par une classe `Environnement`, avec un `HashMap` comme attribut de classe qui prend comme clé un variable et comme valeur un terme. Ensuite dans la classe on retrouve le constructeur avec ensemble des getteur,toString() et les méthodes pour gérer l'environnement.
- En python pour notre projet, l'environnement a été représenté par une classe `Environnement`, avec un dictionnaire comme attribut de classe qui prend comme clé un variable et comme valeur un terme. Ensuite des méthodes pour gérer les ajouts des éléments dans le dictionnaire. **comparaison :** l'implémentation de l'environnement dans les 2 languages ont été similaire avec à peu près la même complexité, car un `HashMap` et `dict` utilise tous les 2 un table d'hachage en interne et le temps d'accès au valeur par clé sont tous en O(1), mais sauf que en terme de language de programmation java est plus rapide que python. Java a aussi une avantage par rapport à python c'est que c'est meilleur en terme de lisibilité. Le fait de sécuriser la class avec des attributs ou des méthodes privates en java est très lisible avec le mot clé `private` , alors qu'en python faut utiliser `__` pour mettre un attribut en private, mais de manière général quand on fait de la programmation objet en python c'est pas vraiment utilisé, du coup dans notre projet on a pas utilisé dses attributs private. Donc dans les 2 languagesl'implémentation d'environnment est très similaire avec java qui est plus lisible et plus rapide en terme de language de programmation.

2. Contexte

- En java dans le projet de relecture, le contexte a été représenté par une classe `CurrContexte`, avec comme attribut private la liste des buts à résoudre, la liste des règles à explorer et l'environnement comme indiqué dans l'énoncé, et les autres attribut ,`choice , nextChoices, pere`pour aidé à faire le backtracking. Ensuite on retrouve en java toujours le constructeur ,les getteurs et le toString().
- En python pour notre projet, le contexte a été représenté par une classe `Contexte`, avec les attributs basiques comme en java. Et bien sur un constructeur `__init__` une fonction pour gérer les unifications **comparaison :** l'implémentation des 2 structures toujours similaire car nous avons choisit d'implémenter sous forme des objets. Ce qui diffère des deux languages est surtout java est strictement typé, pour avoir les attributs les getteurs sont nécessaire alors qu'en python c'est faisable également, mais ce n'est pas la manière général dont on code en python.

conclusion sur les structures de donées

La structure de données dans les 2 languages sont très similaire si en python on procède de manière objet, les différences sont surtout au niveau syntaxique

comaparaison des algorithmes.

1. l'unification.

- En java dans le projet de relecture, l'unification a été étalé sur plusieur classe. Pour accèder au `Term`, le design pattern Visitor pourrait être utilisé. Ici dans le projet, c'est utilisé dans `occurCheck` pour trouver l'ensemble des variable dans un prédicat. Puis les 4 algorithmes, `orienter , effacer , décomposer et subst`, pour faire l'unification sont implémentés dans un premier temps dans la

classe `Equation` et renvoie tous un boolean, pour vérifié si pour cette équation 1 de ces 4 algorithmes sont appliqués. Ensuite dans la class `System` qui contient la liste des équation à résoudre, la redéfinition de ces algorithmes revient à faire une simple boucle, qui pour chaque équation appel l'algorithmes dans `Equation` correspondant, et renvoie un boolean. Du coup la fonction `unify()` dans `System` revient à simplement à faire une boucle `while`, avec une condition tant que la liste des équation n'est pas vide et que au moins l'un des 4 règle a ét appliqué, on continue à boucler. Les exceptions sont levées lorsque dans `décomposer` et `occurCheck` de la classe `Equation`, du coup si pour une équation dans la liste des équations, la décomposition est impossible ou bien quand le terme gauche est une variable et `occurCheck` renvoie `True` (le code de `formatROK`), une exception `NoSolutionException` est levée.

```
public void unify() {
    // regleapp sert de condition d'arret de notre boucle
d'unification
    boolean regleapp = true;

    while (regleapp && !eqs.isEmpty()) {
        regleapp = false; //aucune regle n'a ete appliquee sur le
systeme pendant ce tour
        // Application des regles
        subst();
        regleapp = regleapp || effacer();
        regleapp = regleapp || orienter();
        regleapp = regleapp || decomposer();
        regleapp = regleapp || remplacer();
    }
  }
```

- En python pour notre projet, l'accès de terme dans les prédicats doivent se faire avec un parcours de liste de l'attribut args de `Predicate` et pour distinguer si c'est une variable on a encore de passer par une condition `if`. Pour unification nous avons fait une une simple fonction `unify`qui vérife les 4 algorithmes à la fois avec des`if elif` dans une boucle `for` qui parcours la liste des équation, et quand y'a décomposition, on crée une nouvelle liste d'équation et on fait une appel récursif avec cette nouvelle liste. Les exception sont levé quand une décompostion est impossible, ou bien avant de retourner l'environnment, on fait un parcours de environnement, si l'occurCheck revoie true pour un élement de l'environnment on lève une exception.

```
def unif(env: Environment, equ: List[Equation], first=True) -> Environment:
    """Fonction d'unification

    Args:
        env: l'environnement de base
        equ: une liste d'equations a resoudre
        first: booléen premettant de différencier un premier appel d'un
appel récursif

    Returns:
        un environnement permettant de resoudre le systeme d'equations
```

```
    Raises:
        NotUnifiable: si il est impossible de résoudre le systeme
d'equations

    """

    for i in range(len(equ)):

        t1 = subst(equ[i].term1, env)  # effectuer les substitution
possible,
        t2 = subst(equ[i].term2, env)
        if isinstance(t1, TermVariable) and isinstance(
            t2, TermVariable
        ):  # si on a des variables a gauche et a droite
            if t1 == t2:  # si egal on efface
                pass
                # del equ[i]
            else:
                env.add(t1, t2)  # sinon on ajoute dans l'environnement
        elif isinstance(t1, TermVariable) and isinstance(
            t2, TermPredicate
        ):  # ajout dans l'environnement
            env.add(t1, t2)
        elif isinstance(t1, TermPredicate) and isinstance(
            t2, TermVariable
        ):  # ajout dans l'environnment et orientant
            env.add(t2, t1)
        elif isinstance(t1, TermPredicate) and isinstance(
            t2, TermPredicate
        ):  # si 2 cote des predicats
            if t1.pred.symbol == t2.pred.symbol:
                equations = []  # liste d'equation a resoudre avec ajout
par boucle
                for j in range(min(len(t1.pred.args), len(t2.pred.args))):
                    equations.append(Equation(t1.pred.args[j],
t2.pred.args[j]))
                env.update(unif(env, equations, False))  # appel recursif
            else:
                raise NotUnifiable

    if first:
        # first est utilisé ici pour différencier le premier appel de la
fonction, d'un appel recursif
        # si c'est le premier appel, toutes les recursions possibles ont
deja ete effectuées, on a donc le resultat final
        # on peut donc verifier si l'environnement permet de resoudre le
systeme d'equation
        subst(equ[i].term1, env)
        subst(equ[i].term2, env)

        keys = env.keys()

        for k, v in env.env.items():
```

```
            # pas de variable a droite
            if isinstance(v, TermVariable):
                # passer si variable à droite
                pass
            else:
                # pas de clé dans les predicats a droite
                for k2 in keys:
                    if OccurCheck(v, k2):
                        raise NotUnifiable()

    return env
```

Comparaison : en terme de lisibilité le code java de relecture est factorisé par les fonctions intermédiaires, qui sont étalé dans plusieurs classe, cela permet de factoriser le code de l'unifcation, mais en terme de lisibilité faut naviguer dans plusieurs classe alors qu'en python malgré d'un code plus compliqué, le code concentré dans un même endroit. Du coup l'avantage en java c'est que le code est factorisé par plusieur fonction intermédiaire qui rend le code de chaque bloque plus simple, mais étalé vaguement dans les classes d'où l'inconvénient, et pour python l'avantage c'est que on retrouve tous qu'on a besoin dans un même bloque de code, mais un bloque assez compliqué. Ensuite pour java en terme d'exécution, tant que la liste des équations n'est pas vide, les 4 algorithmes sont appelés et chacun fait un parcours de la liste des équations, en plus `decomposer` fait une boucle sur chaque équation, pour trouver la nouvelle liste d'équation, du coup la complexité O(n^3). En python, c'est un parcours sur équation, et un récursion est faites pour décomposition, et pour créer la nouvelle liste des équations un parcours sur l'équation courrent est effectué, la complexité est de O(n^3) aussi. Pour conclure, la différence a lieu surtout en terme d'implémentation , ils ont à peu près la même complexité. En java des fonctions intermédiaires et les délegations permet de factoriser le code de `unify()` mais le code est très étalé, notament la fonction `subst()`. Et en python , le code n'est pas factorisé, et a l'air compliqué mais tous le code besoin se trouve dans le même bloque. En python si on procède vraiment de manière objet, le travaux fait en java pourrait d'être rait en python

2. résolution sans backtraking.

- Java

```
public static Environnement choose(int n, Environnement v, Predicate but,
List<DeclAssertion> rules, List<Predicate> nouvGoals) {
        //choose fait aussi l'unification pour les faits
        for (DeclAssertion d : rules) {
            if (d.getHead().getSymbol().equals(but.getSymbol())) {
                // on match, donc on renomme
                DeclAssertion renamed = d.rename(n);
                Systeme s = new Systeme();
                s.setEnv(v.copy()); // copie de l'environnement pour ne pas
modifier celui pass? en param?tre
                s.addEquation(new Equation(
                        new
TermPredicate(renamed.getHead(),renamed.getPosition()),
                        new TermPredicate(but,but.getPosition())));
                s.unify();
```

```
                nouvGoals.addAll(renamed.getPredicates());
                return s.getEnv();
            }
        }
        throw new NoSolutionException("pas d'environnement correspondant
pour le but "+but);
    }

    /**
     * R�sout le probl�me d�crit par les param�tres
     * @param goals : liste de buts � r�soudre
     * @param rules : liste de r�gles
     * @return l'environnement solution, lance NoSolutionException si le
probl�me n'est pas satisfiable
     */
    public static Environnement solve(List<Predicate> goals,
List<DeclAssertion> rules) {
        Environnement res = new Environnement();
        int cpt = 1;
        List<TermVariable> vars = vars(goals);
        while (!goals.isEmpty()) {
            Tools.addText("Buts � v�rifier : "+goals);
            res = choose(cpt,res,goals.get(0),rules,goals);
            goals.remove(0);
            cpt++;
        }
        res.nettoieEnv(vars);
        return res;
    }
```

La résolution sans backtracking se fait par les algorithmes `choose() et solve()`.`choose()` prennant en argument un entier n pour le renommage, un environnment , un but, une liste des règles et liste des prédicats qui représente les nouveaux buts `nouvGoals` (comme c'est définie comme une fonction statique). Cette méthode va parcourir la liste des règles, si une règle ou un fait a le même symbole que le but passer en argument, renome cette règle avec `rename()` et fait une copy de l'environnement pour ne pas modifier l'environnement de départ, puis unifie ce but avec le règle, si l'unification se passe bien ajoute le corps de la règle dans la liste des nouveau règles et renvoie l'environnent. `solve` revient à faire, tant que la liste des buts n'est pas vide, appeller `choose()` sur le premier but, avec la liste des buts `goals`comme `nouvGoals`, puis enlève le premier but après `choose`. Et `solve` va renvoyer un environnement filtré qui va être l'environnment final. La complexite de la résolution est de O(n²). *Python :

```
def choose(n: int, env: Environment, goal: Predicate, rules) ->
Tuple[Environment, list]:
    """Fonction permettant de trouver des regles et un environnement
permettant d'unifier un goal

    Args:
        n: la valeur du compteur (pour le renommage)
        env: l'environnement a utiliser
        goal: le goal a unifier
```

```
        rules: une liste de regles

    Returns:
        env: l'environnement
        body: liste de regles
    """
    body = None

    regles = copy.copy(rules)
    cpt = 0
    for i in regles:
        try:
            if isinstance(i, DeclAssertion):
                r = rename(n, i)
                env = unif(env, [Equation(TermPredicate(r.head),
TermPredicate(goal))])

                body = r.preds
        except NotUnifiable:
            cpt += 1
            continue

    if cpt == len(regles):
        raise NotUnifiable()

    return env, body


def solve(buts: DeclGoal, rules: list) -> Environment:
    """Fonction permettant de prouver successivement chaque but de la liste
goals à l'aide d'une suite de regles rules

    Args:
        goals: une liste de buts a prouver
        rules: une liste de regles

    Returns:
        env: un environnement permettant de verifier tous les buts de goals

    Raises:
        NotUnifiable: si aucun environnement ne permet de verifier tous les
buts

    """
    env = Environment()
    goals = buts.preds
    i = 0
    while goals != []:

        but = goals[0]
        env, body = choose(i, env, but, rules)
        i += 1
        goals.extend(body)
        del goals[0]
```

```
    return env
```

la résolution sans backtracking en python de notre projet est très similaire à celui de java, juste dans `choose` l'unification est gérée par exception, si pour un but passer en paramètre, une règle lève une exception, on regarde le prochain règle, si toutes les règles lève une exception alors le système n'est pas unifiable. Si pas d'exception `choose` renvoie un tuple contenant l'environnement et le corps de la règle qui l'a unifié avec. Dans `solve` c'est pareil que java, juste l'ajout des nouveaux buts est fait ici et l'environnment est filtré dans les interpretes. **comaparaison :** la résolution en java est sans doute meilleur, car dans `choose` son vérification est faites sur les symboles, si le symbole est différent ont passe directement au prochain règle. Alors que dans notre projet en python, pour chaque règle on applique `unify()` et si exception on passe à la prochaine. Mais en terme de language, ce qui est fait en java peut être refait en python. La complexite de la résolution est de $O(n^2)$.

conclusion parti 2

L'implémentation du projet dans les 2 languages est très similaire, la version faites en Java peut être refait de manière similaire en python, les différences des algorithmes et des structures de données viennent des choix des implémentations chosit, et des différences syntaxique

conclusion finale

En faisant la relecture de ce projet, j'ai découvert de nombreux chose que on aurait pu utilisé dans notre projet. Ce projet passe parfaitement sur un ensemble de test que j'ai effectué, en les prennant de mes projets. C'est aussi un projet qui m'a pas posé de grand problème pour la relecture, malgré avec un peu d'étalement de code. En comparant les 2 projets, j'ai remarqué que ce qui a été fait java peuvent être refait de manière similaire en python, et j'ai découvert des méthodes que j'aurais pu utilsés pour optimiser mes algorithmes. Pour conclure, à travers la relecture de ce projet j'ai pu comprendre comment un même projet pouvant être fait dans 2 languages différent, avec des suptilités syntaxique, et j'ai appris comment réadapter un projet java en python orienté objet et inversement.