

IMT – FIL A3 – Capitrain

Rémi BARDON, Lucas ROURET

25 novembre 2022

- Organisation du projet
 - Package main
 - Package test
 - Comment lancer les tests
 - Stratégie de test
- Ce que nous avons fait
- Points forts de notre projet
- Code
 - Bien structuré
 - Lisible
 - Très orienté objet
 - Variables et noms de fonctions clairs
- Tests
 - Vraie librairie de tests (pas simplement un `main`)
- Points faibles de notre projet
- Un choix discuté
 - Option 1 : Faire référence aux types par des `String`
 - Option 2 : Faire référence aux types par des objets
 - Notre choix
- Ce que nous aurions fait différemment si c'était à refaire
 - Traduire le code Haskell en Java
 - Compréhension du document
- Autres informations pouvant être utiles
- Log du projet

Organisation du projet

Le projet possède 2 packages mère :

- `main` ayant la structure de FJ.
- `test` avant nos tests.

En effet, pour tester le projet, nous avons décidé d'utiliser Junit5.

Package main

Voici la structure du package main :

- model
 - error
 - java
 - expression

- misc
- type
- misc
- util.haskell

Description des packages :

util.haskell: Possède une classe "Haskell" qui permet de faire des opérations sur les listes que Java ne possède pas.

error: Possède les classes d'erreurs tel que "VariableNotFound" ou "TypeError".

java.expression: Possède les classes relatives aux expressions tel que "FJCast" ou "FJLambda".

java.type: Possède les classes relatives aux types tel que "FJClass" ou "FJType".

java.misc: Possède les classes relatives aux autres éléments tel que "FJSignature" ou "FJConstructor".

Dans l'UML, vous pouvez voir dans quelle package se trouve chaque classe.

Package test

Dans le package test, nous avons la class **TypeCheckerTest** qui permet de tester le type checker des approches 1 et 2.

Nous avons aussi **EvaluationTests** qui permet de tester la réduction que n'avons pas eu le temps de tester.

Puis, nous avons **FJTest** qui permet de tester l'exemple donné dans le document de recherche.

Comment lancer les tests

Comme mentionner plus haut, nous avons utilisé Junit5 pour effectuer nos tests unitaires.




Pour lancer les tests, il suffit de lancer la classe **FJTest** et **TypeCheckerTest** qui se trouve dans le package **test**. Pour ça, nous avons préconfiguré une configuration pour Eclipse.








Pour la retrouver il faut aller dans Run > Run Configurations > JUnit vous devriez voir une configuration nommée "TypeCheckTests". Il suffit simplement de lancer la configuration pour lancer les tests de tous nos tests.

Stratégie de test

Nous avons décidé de tester chaque séquent pour vérifier le bon fonctionnement de l'implémentation. À voir dans la partie "Points forts de notre projet"

Ce que nous avons fait

-  Création du projet
-  Création de la structure de FJ en Java
 - Nous avons utilisé la structure de FJ-Lam
-  Type Checker approche 1
 - Nous avons implémenté le type checker de l'approche 1, en se basant sur la définition donnée dans le document, mais aussi le code haskell.

-  Type Checker approche 2
-  Evaluation approche 1
-  Evaluation approche 2
-  Tests Type Checker approche 1
 - Nos tests permettent de vérifier : T-Field, T-Invk, T-Var, T-New T-Lam, T-UCast, T-Dcast et T-SCast
-  Tests Type Checker approche 2
 - Nos tests permettent de vérifier : T-Field, T-Invk, T-Var, T-New T-Lam, T-UCast, T-Dcast et T-SCast
-  Tests Evaluation approche 1
-  Tests Evaluation approche 2

Points forts de notre projet

Code

Bien structuré

Comme mentionné dans la partie organisation, nous avons fait un effort sur la structuration du projet, qu'on peut voir dans l'UML.

Lisible

Le code est lisible et commenté. Nous avons fait un effort sur la lisibilité du code. Utiliser des interfaces par exemple pour simplifier la compréhension du code et de son développement

Très orienté objet

Il est simple de vérifier et de comprendre les liens entre les classes. Le type checker et l'évaluation est adapté à l'objet.

Variables et noms de fonctions clairs

Les variables et les noms de fonctions sont clairs et explicites.

Tests

Vraie librairie de tests (pas simplement un `main`)

Pour **simplifier la création des objets**, nous avons utilisé les builders. Cela nous a permis de créer des objets complexes sans avoir à passer par un constructeur. Cela nous a permis de simplifier la création des objets complexes.

Voici un exemple:

```
/*
Eq:
class C extends Objects extends B {
    public C() {super();}
}
```

```
*/
FJClass classC = new FJClassBuilder()
    .name("C")
    .constructor()
    .extendsName("B")
    .build();
```

Points faibles de notre projet

- Utilisation d'une *type table*
- Faible séparation entre l'approche 1 et l'approche 2
- Pas de séparation du *type checker* et de l'évaluateur
- Code en soi très similaire au code Haskell, pas vraiment innovant dans la vision

Un choix discuté

Option 1 : Faire référence aux types par des *String*

Comme dans le code Haskell rédigé par les auteurs du papier étudié, nous avons utilisé le concept de "class table". Selon le document,

a class table CT is a mapping from class or interface names, to class or interface declarations

Dans notre code Java, cela correspond à `Map<String, FJType>`. Nous avons pris la liberté de renommer ce concept en *type table* car on ne stocke pas que des classes, mais aussi des interfaces et potentiellement des types "fonction" (ou autre si évolutions du langage).

Le *parser* FJ (non implémenté dans notre cas) lit du code Java, donc cela a du sens de stocker des *String*. Cependant, cela rend plus complexe le code (entendre complexité algorithmique) car il faut toujours aller chercher dans la *type table* pour récupérer les instances des objets.

Nous avons donc besoin d'écrire

```
final String expectedReturnTypeName = typedBody.typeName();
final boolean returnTypeIsCorrect =
context.typeTable.isSubtype(expectedReturnTypeName,
this.signature.returnTypeName());
if (!returnTypeIsCorrect) {
    // ...
}
```

là où

```
if (!typedBody.type().isSubtype(this.signature.returnType)) {
    // ...
}
```

pourrait suffir si on stackait des `FJType` directement.

Option 2 : Faire référence aux types par des objets

Nous avons comme tâche d'écrire le code "le plus objet possible", alors nous avons rapidement pensé à faire référence aux types par des objets. Cela nous aurait permis de faire disparaître la *type table*, qui est un point central source de beaucoup d'erreurs dans l'autre cas.

La plupart des `Exceptions` que l'on utilise (notamment `ClassNotFoundException` quand une classe n'est pas trouvée dans la *type table*) pourraient disparaître et ainsi simplifier grandement le code. Cela nous permettrait par exemple d'utiliser des `Stream` que nous avons du remplacer par des "for each" car `Stream::map` ne permet pas de passer une méthode/lambda qui lève des `Exceptions`. Ces erreurs nous empêchent d'écrire

```
return typeTable.isSubtype(this.extendsName, otherTypeName)
    || this.implementsNames.stream().anyMatch(t -> typeTable.isSubtype(t,
otherTypeName));
```

et nous forcent à écrire

```
if(typeTable.isSubtype(this.extendsName, otherTypeName)){
    return true;
}
boolean isSubType = true;
for (String implementsName : this.implementsNames) {
    isSubType &= !typeTable.isSubtype(implementsName, otherTypeName);
}
return isSubType;
```

Ce n'est pas très important, mais cela impacte la lisibilité du code.

Le principal inconvénient de cette option est qu'elle complexifie grandement la création des objets. Pour résoudre ce problème, nous avons imaginé une solution expliquée ci-dessous.

Notre choix

explication de pourquoi l'option 1 a été choisie plutôt que l'option 2

Nous avons choisi d'utiliser une *type table* et de stocker des `String` car c'est comme ça que fonctionne le code original en Haskell, mais nous avons prévu (si on avait eu plus de temps) d'implémenter l'option 2 à terme.

Lors de ce changement d'implémentation interne à la librairie, il était important pour nous de ne pas "casser" les tests. Garder les tests fonctionnels permet de vérifier que l'on n'intère pas de régression, ce qui est crucial pendant un *refactoring*. Pour faciliter l'implémentation, nous avons donc créé des *builders* qui font l'interface entre les tests rédigés avec des `String`, et la librairie qui ensuite fonctionnerait avec des objets.

Ce que nous aurions fait différemment si c'était à refaire

Traduire le code Haskell en Java

Au début du projet, nous avons décidé de traduire le code Haskell pour avoir une base solide. Puis à partir de la traduction, adapté le code actuels pour le rendre fonctionnel en objet. Cette erreur, nous a fait perdre beaucoup de temps.

Compréhension du document

Nous aurions dû directement partie sur les séquents. En effet, prendre le temps d'avoir une compréhension complète du document, pour en déduire la structure et les implémentations. En effet, cette traduction nous a orienté vers une compréhension vague sur la différence entre l'approche 1 et 2 que, nous avons compris par la suite.

La compréhension du document totale du document aurait dû être notre premier objectif lors du commencement du projet.




Autres informations pouvant être utiles










Nous avons joint quelques fichiers utiles :

- **coverage.png** : Le résultat du *coverage report* fait sur notre librairie
- **LEXIQUE.md** : Un lexique des termes utilisés dans les séquents du papier de recherche
- **SEQUENTS.md** : Une traduction en langage naturel de quelques séquents
 - Pas complète par manque de temps, mais une base de compréhension pour le binôme
- L'implémentation Swift n'est pas fournie mais peut l'être à la demande

Log du projet

Légende du tableau :

-  = Créneau prévu dans l'emploi du temps
-  = Tâche prévue et faite
-  = Tâche pas prévue mais faite

Date	Tâches réalisées
2022-11-25 PM	 Tester chaque séquent  Écrire chaque partie du rapport  Générer le rapport complet  Déposer le rapport
2022-11-24 PM	 Diagramme UML  Plus de notes dans le papier de recherche  Début d'automatisation de la génération du rapport
2022-11-24 AM 	∅ (IELTS)
2022-11-23 PM	 Tester chaque séquent

Date	Tâches réalisées
2022-11-20 PM	<ul style="list-style-type: none"> ✓ Traduire FJTypeChecker.hs approche 1 en Java ✓ Tester chaque séquent
2022-11-19 PM	<ul style="list-style-type: none"> ✓ Améliorer/finir le lexique ✓ Traduire des séquents en langage naturel en vue du rapport
2022-11-18 PM  17	<ul style="list-style-type: none"> ✓ Introduire JUnit ✨ Configurer la collecte de <i>code coverage</i> ✨ Rendre plus ergonomique la création d'objets dans les tests ✓ Préparer l'implémentation de l'approche 1
2022-11-16 PM	<ul style="list-style-type: none"> ✓ Améliorer la gestion des erreurs et le <i>logging</i> pour faciliter la résolution de <i>bugs</i>
2022-11-16 AM  17	<ul style="list-style-type: none"> ✓ Créer TypeCheckingContext pour remplacer (TypeTable, Map<String, String>) ✓ Implémenter des <i>builders</i> pour faciliter la création d'objets dans les tests ✓ Implémenter les premiers tests
2022-11-15 PM	<ul style="list-style-type: none"> ✓ <i>Refactoring</i> fonctionnel vers objet ✓ Simplifier le code ✨ Créer TypeTable pour remplacer Map<String, FJType> ✓ Supprimer FJUtils ✓ Mettre en place un <i>linter</i> et fixer tous les <i>warnings</i>
2022-11-14 PM	<ul style="list-style-type: none"> ✓ <i>Refactoring</i> fonctionnel vers objet ✓ Traduire FJUtils.hs en Java ✨ Quelques <i>bug fixes</i>
2022-11-13 PM	<ul style="list-style-type: none"> ✓ Traduire FJUtils.hs en Java
2022-11-12 PM	<ul style="list-style-type: none"> ✓ Traduire FJUtils.hs en Java
2022-11-07 AM	<ul style="list-style-type: none"> ✓ Traduire FJTypeChecker.hs approche 2 en Java ✨ Quelques <i>bug fixes</i>
2022-11-04 PM  17	<ul style="list-style-type: none"> ✓ Traduire FJTypeChecker.hs et FJInterpreter.hs approche 2 en Swift ✓ Écrire un test en Swift
2022-11-03 PM	<ul style="list-style-type: none"> ✓ Traduire FJUtils.hs en Swift ✓ Représenter un programme FJ avec les types définis en Swift
2022-11-02 PM	<ul style="list-style-type: none"> ✓ Traduire FJParser.hs en Swift ✓ Avancer notre exploration en Java
2022-10-28 PM  17	<ul style="list-style-type: none"> ∅ (Travail de lecture fait en amont)
2022-10-27 PM	<ul style="list-style-type: none"> ✓ Créer le dépôt GitHub ✓ Ajouter des annotations sur le papier de recherche ✓ Début d'exploration en Java