Research Project
Transformation sécurisée d'un langage fonctionel

Enseignant : JABER Guilhem
FILAUDEAU Éloi, BOURSIER Louis, LASHERME Loïc

1 Introduction

En informatique, il est fréquent de faire appel à des fonctions proposées par des bibliothèques afin de déléguer une partie du travail de programmation. Ce faisant, le programme devient un programme ouvert dans le sens où il est maintenant lié à un environnement qui lui fournit les fonctions dont il a besoin. À ce moment là, des problèmes de sécurité peuvent survenir, dus notamment au fait que le langage utilisé dans la bibliothèque peut ne pas préserver les abstractions du langage source, ou bien fournir des abstractions différentes. On parle alors d'environnement malicieux.

Afin de préserver les abstractions de nos programmes contre l'environnement malicieux, nous pouvons mettre en place des sécurités. Ces sécurités permettent de préserver les abstractions du langage comme cité précédemment. Une abstraction peut être la préservation de la visibilité d'une variable de classe en Java. C'est-à-dire empêcher l'environnement de pouvoir lire une variable déclarée comme privé.

Pour comprendre la suite, il faut s'intéresser à la définition de l'équivalence contextuelle. Le fait que deux programmes soient contextuellement équivalent signifie qu'ils implémentent la même signature, et qu'aucun programme tiers ne peut les distinguer.

Pour répondre à ce problème, on peut utiliser un compilateur sécurisé. Un critère définissant si un compilateur est sécurisé ou non est la compilation *fully abstract*. Un compilateur *fully abstract* assure que deux programmes contextuellement équivalent dans un langage source le sont aussi dans un langage destination, et que donc les propriétés d'abstractions sont préservées. Cela a pour conséquence qu'aucun contexte ne peut distinguer un programme de l'autre en essayant de leur faire produire des comportements *input/output* différents. On dit aussi qu'on observe le même *control flow* pour les deux programmes.

Formally, a fully-abstract compiler preserves and reflects observational equivalence (usually contextual equivalence) between source and target programs. Reflection of observational equivalence means that the compiler outputs target-level components that behave as their source-level counterparts, this is generally a consequence of the compiler being correct. Preservation of observational equivalence implies that the source-level abstractions in the generated target-level output are not violated by a target-level client[3].

Nous définissons une traduction de notre langage source vers le langage enrichi avec les exceptions. Cette traduction préserve l'équivalence contextuelle du langage source. Nous aurons alors la garantie de préserver les abstractions du langage source. Ainsi, la traduction n'introduit pas de nouvelles vulnérabilités. Cependant, elle ne pourra pas non plus enlever celles qui pourraient être présentes dans le langage source. Nous utiliserons des modèles dénotationnels *fully abstract* utilisant la sémantique des traces pour montrer l'équivalence dans le modèle $[\![t_1]\!] = [\![t_2]\!]$ et donc l'équivalence dans les programmes $t_1 \simeq_{ctx} t_2$.

2 Méthodologie

2.1 Syntaxe de notre langage

Dans un premier temps, nous devons créer notre petit langage de programmation ressemblant fortement à Mini-ML, et inspiré de celui de Pierce[1], afin de raisonner sur notre problème. De plus, nous voulons pouvoir avoir accès aux exceptions pour pouvoir appliquer notre transformation. La syntaxe de notre langage est donc la suivante :

$v ::=$		valeurs :
	$true$	constante true
	$false$	constante false
	$()$	constante unit
	n	avec $n \in \mathbb{Z}$
	x	avec $x \in$ variables
	fun $x \mapsto t$	
	fix $f(x) \mapsto t$	avec $f \in$ variables

$t ::=$		termes :
	v	
	$t \circ t'$	$\circ \in \{+, -, \times, \div\}$
	$t = t'$	égalité
	$t\ t'$	application
	if t then t' else t''	conditionnelle
	try t with t'	exception
	$error$	erreur
	let $t_1 = t_2$ in t_3	sucre syntaxique

$T ::=$		types :
	$Unit$	type unit
	$Bool$	type booléan
	Int	type entier
	$T \to T'$	

$E ::=$	contexte d'évaluation :
	$\bullet\ t$
	$v\ \bullet$
	$\bullet + t$
	$v + \bullet$
	if $\bullet$ then t else t'
	try $\bullet$ with t

2.2 Sémantique de notre langage

Une fois notre syntaxe déclarée, il faut pouvoir y rajouter du sens. Pour cela, nous définissons notre sémantique de la manière suivante :

$\dfrac{t_1 \xrightarrow{*} true}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \to t_2}$	Si la condition t_1 se réduit à $true$ alors on évalue t_2.
$\dfrac{t_1 \xrightarrow{*} false}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \to t_3}$	Si la condition t_1 se réduit à $false$ alors on évalue t_3.
$\dfrac{}{\text{try } v \text{ with } t \to v}$	Si aucune $error$ n'est remontée dans le try, alors on évalue seulement la valeur v.
$\dfrac{t_1 \xrightarrow{*} error}{\text{try } t_1 \text{ with } t_2 \to t_2}$	Si la condition t_1 renvoie une $error$ alors on évalue t_2.
$\dfrac{}{(\text{fun } x \mapsto t)v \to t\{v/x\}}$	On évalue d'abord l'argument passé à la fonction avant de l'appliquer : c'est un appel par valeur différent de l'appel par nom.
$\dfrac{u = \text{fix } f(x) \mapsto M \qquad u\ v \to M\{v/x\}\{u/f\}}{E[u\ v] \to E[M\{v/x\}\{u/f\}]}$	La règle d'évaluation du point fixe.
$\dfrac{t \to t'}{E[t] \to E[t']}$	Si un terme t se réduit vers un autre terme t', alors ce même terme se réduit de la même façon dans le contexte.

À ceci, nous rajoutons les règles de typages de notre langage :

$\dfrac{}{\Gamma \vdash true : Bool}$	On évalue $true$ au type $Bool$.
$\dfrac{}{\Gamma \vdash false : Bool}$	On évalue $false$ au type $Bool$.
$\dfrac{}{\Gamma \vdash () : Unit}$	On évalue () au type $Unit$.
$\dfrac{}{\Gamma \vdash n : Int}$	On évalue $n \in \mathbb{Z}$ au type Int.
$\dfrac{}{\Gamma \vdash error : T}$	On évalue $error$ en un type T.
$\dfrac{x : T \in \Gamma}{\Gamma \vdash x : T}$	Si la variable x de type T appartient au tas, alors son évaluation renvoie T.
$\dfrac{\Gamma \vdash t : Bool \qquad \Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash \text{if } t \text{ then } t_1 \text{ else } t_2 : T}$	Dans une conditionnelle, le premier terme doit s'évaluer sur le type $Bool$ et les deux autres sur le même type T.
$\dfrac{\Gamma \vdash t : T \qquad \Gamma \vdash t' : T}{\Gamma \vdash \text{try } t \text{ with } t' : T}$	Les deux termes d'une exception doivent s'évaluer sur le même type T.
$\dfrac{\Gamma, x : T \vdash t : U}{\Gamma \vdash \text{fun } x \mapsto t : T \to U}$ $\dfrac{\Gamma, x : T, f : T \to U \vdash t : U}{\Gamma \vdash \text{fix } f(x) \mapsto t : T \to U}$ $\dfrac{\Gamma \vdash t : T \to U \qquad \Gamma \vdash t' : T}{\Gamma \vdash t\ t' : U}$	Si un terme t se réduit de T en U et que l'on lui applique un terme t' de type T, alors le résultat de cette application est de type U.

2.3 Les énoncés

2.3.1 Les interprétations d'un terme t

Soit t un terme tel qu'il existe Γ et T avec $\Gamma \vdash t : T$ et toutes les variables de Γ sont de type $B \to B$. Alors :
(Avec $B ::= Unit \mid Bool \mid Int$)

* Soit t diverge
* Soit il existe une valeur v tel que : $t \xrightarrow{*} v$ et $\Gamma \vdash v : T$
* Soit il existe E, f, v tel que : $t \xrightarrow{*} E[fv]$ avec f une variable libre

2.3.2 Définition équivalence contextuelle/observationelle

$\Gamma \vdash t_1, t_2 : T$
$\Gamma \vdash t_1 \simeq_{ctx} t_2$ ssi pour tout contexte E tel que :

(1) $x : T \vdash C[x] : Bool, \qquad C[t_1] \xrightarrow{*} true$ ssi $C[t_2] \xrightarrow{*} true$

(2) $x : T \vdash C[x] : Unit, \qquad C[t_1] \xrightarrow{*} ()$ ssi $C[t_2] \xrightarrow{*} ()$

(1) $\Rightarrow$ (2) $x : T \vdash C[x] : Bool \rightsquigarrow C' =$ if C then () else Ω
(2) $\Rightarrow$ (1) $x : T \vdash C[x] : Unit \rightsquigarrow C' = C; true \qquad C'_n =$ if $C = n$ then () else Ω
Avec Ω symbole pour la divergence.

2.3.3 Définition du théorème de transformation

La transformation de programme :
Soit $(.)^*$ la transformation de programme. Avec $(.)^* : L \to L$
$L \subseteq L_{ex}$ nos deux langages de programmation
Pour tous programmes P_1, P_2 de L, si $P_1 \simeq_L P_2$ alors $(P_1)^* \simeq_{L_{ex}} (P_2)^*$
Soit t_1, t_2 deux programmes de notre langage L tel que :
$\Gamma \vdash t_1, t_2 : T$ et $t_1 \simeq_{ctx}^{L} t_2 \qquad [\![t_1]\!]_L = [\![t_2]\!]_L$
Alors $(t_1)^* \simeq_{ctx}^{L_{ex}} (t_2)^* \qquad$ Alors $[\![(t_1)^*]\!]_{L_{ex}} = [\![(t_2)^*]\!]_{L_{ex}}$

2.4 Quelques exemples sur des fonctions

2.4.1 estPair

$$\text{estPair} = \text{fix } f(x) \mapsto \text{if } x = 0 \text{ then } true \text{ else (if } x - 1 = 0 \text{ then } false \text{ else } f(x-2))$$

On souhaite évaluer : estPair 3 par rapport à un contexte, ce qui donne l'évaluation : E[estPair 3]
Avec nos règles décrites plus haut, cela nous donne cette évalution :

$\to$ if $3 = 0$ then $true$ else (if $2 = 0$ then $false$ else estPair $(3-2)$)

$\to$ if $2 = 0$ then $false$ else estPair $(3-2)$

$\to$ estPair $(3-2)$

$\to$ if $1 = 0$ then $true$ else (if $0 = 0$ then $false$ else estPair $(1-2)$)

$\to$ $false$

Ce qui revient à évaluer $E[false]$ dans le contexte.

2.4.2 estDiviseur

$$\text{estDiviseur } = \text{fix } f(x,y) \mapsto \text{if } y \leq 0 \text{ then (if } y = 0 \text{ then } true \text{ else } false) \text{ else } f(x, y-x)$$

Même façon de procéder que pour estPair, on souhaite évaluer : estDiviseur 3 7 dans le contexte ce qui donne : E[estDiviseur 3 7]. En réduisant on obtient :

$\to$ if $7 \leq 0$ then (if $7 = 0$ then $true$ else $false$) else estDiviseur 3 $(7-3)$

$\to$ estDiviseur 3 $(7-3)$

$\to$ if $4 \leq 0$ then (if $4 = 0$ then $true$ else $false$) else estDiviseur 3 $(4-3)$

$\to$ estDiviseur 3 $(4-3)$

$\to$ if $1 \leq 0$ then (if $1 = 0$ then $true$ else $false$) else estDiviseur 3 $(1-3)$

$\to$ estDiviseur 3 $(1-3)$

$\to$ if $-2 \leq 0$ then (if $-2 = 0$ then $true$ else $false$) else estDiviseur 3 $(-2-3)$

$\to$ $false$

Ce qui revient de nouveau à évaluer $E[false]$ dans le contexte.

3 Résultats

3.1 Exemple de transformation de programme en Ocaml

Dans notre exemple, nous voulons préserver l'équivalence contextuelle des fonctions *fonction_ test_ bis* et *fonction_ test*. Les deux fonctions renvoient l'entier 1. Avant de renvoyer l'entier, les deux fonctions ont deux appels réentrants. *fonction_ test_ bis* exécute seulement ses deux appels, puis renvoie 1. *fonction_ test* renvoie la valeur de la référence à x à la fin de la fonction, et entre chaque appel réentrant, modifie la valeur de la référence. Pour préserver l'équivalence, il faut que les deux fonctions puissent renvoyer la même sortie en toute circonstance. Malheureusement, ce n'est pas le cas. En effet, *fonction_ environnement* qui représente une fonction de l'environnement, prend en paramètre une autre fonction et l'exécute. En passant nos deux fonctions en paramètre, ces deux mêmes fonctions ne renvoient pas la même sortie. *fonction_ test_ bis* renvoie bien 1 comme souhaité, mais *fonction_ test* renvoie 0. Ce problème apparaît à cause de *fonction_ environnement*, qui compte le nombre d'appel réentrant et renvoie une erreur lorsque qu'il y en a déjà eu un. De plus, *fonction_ test* renvoie une référence, qui est modifiée entre chaque appel. Ainsi, l'erreur provoquée par *fonction_ environnement* ne prend pas en compte la modification de la référence et la laisse à 0. Donc, en l'état, nos deux fonctions ne sont pas contextuellement équivalentes.

Pour permettre de préserver l'équivalence, nous avons mis au point une transformation. Celle-ci consiste à créer une pile d'identifiants inaccessible par l'environnement. Pour protéger notre fonction, nous établissons un identifiant à l'intérieur de celle-ci que nous empilons sur notre pile. À la fin de chaque appel réentrant, nous vérifions que l'identifiant en tête de pile est bien le même que celui de notre fonction. Si c'est le cas, nous continuons, sinon, nous provoquons une erreur, car l'environnement essaye de différencier notre fonction. À la fin de notre fonction, nous supprimons l'identifiant en tête de notre pile. Grâce à cette transformation, il nous est possible de garder l'équivalence de *fonction_ test* et de *fonction_ test_ bis*.

```
exception Error;;

let pile = ref [];;
let id = ref 0;;

(* Permet de créer un id différent à chaque appel *)
let generate_id () =
   let x = !id in
      id := !id + 1;
      x
;;

(* Permet de push l'id dans la pile *)
let push idCourant =
   pile := idCourant :: !pile
;;

(* Permet de supprimer l'id en tête de pile *)
let pop () =
   match !pile with
      | [] -> failwith "Erreur, pile vide"
      | hd::tl -> pile := tl
;;

(* Permet de comparer id_courant avec l'id en tête de pile *)
let check_id id_courant =
   match !pile with
      | [] -> failwith "Erreur, id different"
      | hd::_ ->  if hd != id_courant
                  then failwith "Erreur, id différent"
;;
```

```
(* Fonction sans les sécurités *)
let fonction_test =
   let x = ref 0 in
      fun f ->
         x := 0;
         f ();
         x := 1;
         f ();
         !x
;;

(* Est censée être équivalente à fonction_test *)
(* Mais ne l'est pas car l'environnement peut les distinguer *)
let fonction_test_bis =
   fun f ->
            f ();
            f ();
            1
;;

(* Fonction avec les sécurités *)
let fonction_test_secu =
      let x = ref 0 in
         fun f ->
            let id_courant = generate_id () in
               push id_courant;
               x := 0;
               f ();
               check_id id_courant;
               x := 1;
               f ();
               check_id id_courant;
               let n = !x in
                  pop ();
                  n
;;

(* Fonction malicieuse fournie pas l'environnement *)
(* N'est pas censée être présente dans le programme source *)
(* Nous pouvons uniquement l'appeler car nous avons théoriquement load la bibliothèque auparavant *)
let fonction_environnement f =
   let c = ref 0 in
   f (fun _ -> if !c = 0
               then c := !c + 1
               else  try   begin
                              f(fun _ -> raise (Error));
                              ()
                           end
                     with Error -> ())
;;

(* Appel de la fonction fournie par l'environnement avec nos fonctions *)
(* Ne renvoie pas le bon résultat à cause de la fonction malicieuse de l'environnement *)
fonction_environnement fonction_test;;

(* Renvoit une erreur car l'environnement interrompt notre fonction en plein milieu*)
fonction_environnement fonction_test_secu;;
```

---

4 Automatisation de la transformation

4.1 Implémentation en Java

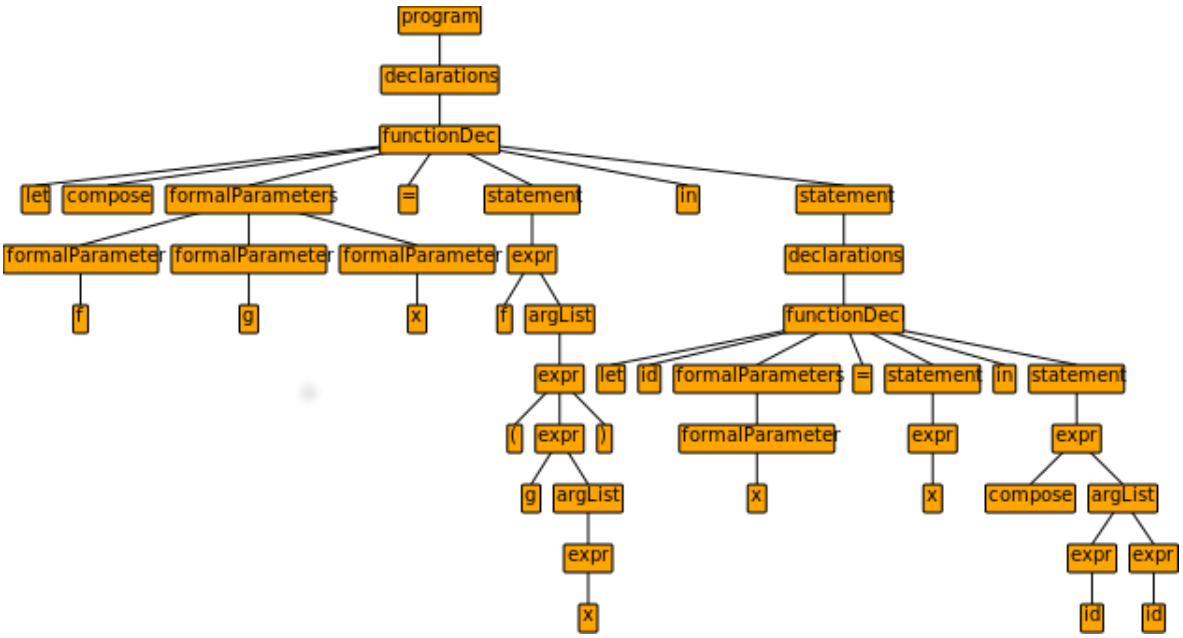
`https://github.com/loutouk/Safe-Translation?fbclid=IwAR3VX2RZ0ay8Dt8g9gZg-7xj-hWe_dimD8fQYJjXJWNgpl0JE8uOleZW95E`

Une implémentation du traducteur a été faite en Java. Ce choix est motivé par deux raisons principales. La première est que la bibliothèque **ANTLR** est disponible sur Java. La seconde est que ce langage de programmation nous est bien connu, et nous sommes donc productifs avec.

ANTLR — **AN**other **T**ool for **L**anguage **R**ecognition — est un compilateur de compilateur écrit par Terrence Parr. Il fonctionne avec une analyse syntaxique descendante LL(*) et prend en entrée

une grammaire context-free de type **EBNF** pour produire les lexeurs et parseurs associés. Un point fort de **ANTLR** est la construction automatique des arbres syntaxiques et les outils disponibles pour explorer ces arbres. Notamment avec l'utilisation des patterns Visitor et Listener[4]. C'est d'ailleurs le pattern Listener que nous avons utilisé pour effectuer la traduction. Son fonctionnement est similaire à celui d'un listener sur un composant d'une interface graphique : un appel est déclenché chaque fois que le composant est activé. Ici, un appel est déclenché chaque fois qu'une règle est reconnue lors de l'analyse d'un programme.

La première étape consiste à écrire la grammaire de notre langage. Celle-ci est largement inspirée de OCaml puisque notre langage doit pouvoir être compilé par un compilateur OCaml. Nous avons donc choisi de décrire un sous-ensemble de ce langage. Globalement, un programme est un ensemble de déclarations de variables.




```
program:
      (declarations)+
;

expr:
      expr TIMES expr          # Mult
    | expr PLUS expr           # Add
    | expr MINUS expr          # Sub
;
```

On peut définir un listener pour chaque règle de la grammaire. Et on peut, si nécessaire, affiner la précision sur le type de noeud en générant un appel différent pour chaque variante de la règle à l'aide du #. Une fois la grammaire compilée, **ANTLR** génère des interfaces Java contenant les listeners sur ces règles. Il suffit d'implémenter l'interface dans une classe pour pouvoir en bénéficier :

```
void enterMult(RefMLParser.MultContext ctx);

void exitMult(RefMLParser.MultContext ctx);
```

Le but de la traduction est de sécuriser les appels aux fonctions de bibliothèques externes, c'est-à-dire les variables libres. Pour ce faire, il faut tout d'abord identifier ces appels, et ensuite identifier où placer les instructions pour sécuriser chaque appel.

Pour identifier les appels aux fonctions externes, il faut déjà savoir identifier les appels de fonctions. Comme expliqué au dessus, cela est réalisé facilement avec **ANTLR**.

Il faut ensuite savoir différencier les appels aux fonctions externes des appels aux fonctions déclarées localement. Même si l'on pourrait simplement sécuriser les appels aux fonctions locales également, cela est inutile et rend le programme moins lisible et moins performant. Une solution classique consiste à créer une table des symboles qu'on remplit durant la première exploration de l'arbre syntaxique. Pour chaque portée (scope), on renseigne les fonctions et les variables qui y sont déclarées.

Ainsi, lors d'une deuxième passe, nous serons capables d'identifier quels symboles sont des variables libres.

En OCaml et dans notre langage, la déclaration de variables est raccrochée à une portée :

```
let average a b =
   let sum = a +. b in
      sum /. 2.0;;
```

* *average* a une portée globale
* *a* et *b* ont une portée locale limitée à la fonction
* *sum* a une portée locale limitée à l'expression qui suit le mot clé in

Comme notre grammaire est construite pour représenter les règles du langages, les appels déclenchés par le listener nous informe dans quel type d'expression nous sommes (déclaration de fonction, partie droite d'un in...). Ainsi, il est facile de créer une nouvelle portée pour la table des symboles quand c'est nécessaire. On y déclare les fonctions et variables qu'on rencontre dans cette portée, en maintenant la connaissance de la portée dans laquelle nous nous trouvons lors du parcours de l'arbre.

```
variableDecl:
     LET ID EQUAL REF? expr          # VarDecl
;

public void exitVarDecl(RefMLParser.VarDeclContext ctx) {
   defineVar(ctx.ID().getSymbol(), Type.SymbolType.VAR);
}

void defineVar(Token nameToken, Type.SymbolType type) {
   VariableSymbol var = new VariableSymbol(nameToken.getText(), type);
   currentScope.define(var); // Define symbol in current scope
}

public void define(Symbol sym) {
   orderedArgs.put(sym.name, sym);
   sym.scope = this; // track the scope in each symbol
}
```

Lors de la deuxième passe, lorsque l'on cherche à savoir si une variable est libre, il suffit de remonter toutes les portées, de la portée locale actuelle jusqu'à la portée globale initiale, pour trouver la variable rencontrée. Si on ne la trouve pas, c'est qu'elle n'a pas été déclarée et que c'est donc une variable libre.

```
expr:
     ID argList          # Call
;

public void exitCall(RefMLParser.CallContext ctx) {
   Symbol symbol = currentScope.resolve(ctx.ID().getText());
   if(symbol == null) {
      // protect
   }
}

public Symbol resolve(String name) {
   Symbol s = orderedArgs.get(name);
   if (s != null) return s;
   // if not here, check any enclosing scope
   if (getEnclosingScope() != null) {
      return getEnclosingScope().resolve(name);
   }
   return null; // not found
}
```

Maintenant que l'on sait quelles sont les variables libres, il faut savoir où placer les fonctions qui sécurisent l'appel à la fonction externe.

La traduction repose sur le fait de pouvoir identifier les comportements indésirables d'un programme. L'idée de base est de constater qu'un appel externe s'est bien comporté, c'est à dire qu'il a respecté les propriétés d'abstractions garanties par notre langage, en installant un mécanisme de vérification. Ce mécanisme de vérification repose sur l'analyse du flot de contrôle du programme en vérifiant

que l'instruction qui devait se produire s'est bien exécutée entièrement. Cette garantie peut être mise à mal par les exceptions qui ont pour définition de rediriger le flot de contrôle vers un handler. En définissant une suite d'identifiants pour chaque portée où sont utilisées des fonctions externes, on peut vérifier que le flot de contrôle suit son bon déroulement en s'assurant que l'identifiant local correspond bien à l'identifiant global de la pile. Cet ordre doit correspondre à celui de la pile d'exécution lorsque les instructions sont exécutées entièrement. Une pile est utilisée pour conserver les identifiants globaux. Le choix d'une pile est naturel étant donné le comportement **LIFO** des appels de fonctions. Un exemple est fourni dans le rapport pour bien comprendre le mécanisme et le lien avec les exceptions.

Lorsque que des appels à des fonctions externes sont identifiés dans une fonction, c'est à dire dans une portée, on va protéger ces appels dans la portée qui contient ces fonctions. Pour cela, on commence par générer un identifiant local qui est simplement une incrémentation du compteur global. On ajoute ensuite cet identifiant local à la pile globale. A partir de ce moment, l'identifiant local doit être le même que celui de la pile. Sinon, c'est que le flot de contrôle a été dérouté. Etant donné que le flot ne peut être dérouté que par les appels aux fonctions externes, on vérifie cela après chaque appel de ce genre. A la fin de la portée, on fait sauter le premier identifiant, l'identifiant courant, de la pile. Cela représente le fait que l'on a changé de portée, et que les appels aux fonctions externes doivent maintenant déboucher sur un état mémoire de la pile qui correspond à cette nouvelle portée.

```
let test f =
   let x = ref 0 in
      x := 0;
      f ();
      x := 1;
      f ();
      !x

let test f =
   let x = ref 0 in
      let id = generateId () in
         pushId id;
         x := 0;
         f ();
         checkId id;
         x := 1;
         f ();
         checkId id;
         !x;
         popId ()
```

Cette première traduction peut générer une erreur lors de la compilation. En effet, popId() retourne un type unit, ce qui pose problème lorsque que la fonction courante retourne un autre type. Pour cela, on utilise une construction avec le mot clé in pour stocker le résultat de la fonction courante et le retourner après avoir utilisé popId(). Ainsi, on préserve le type de retour de la fonction courante :

```
let test f =
   let x = ref 0 in
      let return_stat =
         let id = generateId () in
            pushId id ;
            x := 0;
            f ();
            checkId id ;
            x := 1;
            f ();
            checkId id;
            !x in
               popId ();
               return_stat
```

Il faut également prendre en compte que plusieurs portées peuvent contenir des appels à des fonctions externes. Il faut donc générer des noms de variables différents et non ambigus :

```
let test f =
   let x = ref 0 in
      let return_stat1 = let id1 = generateId () in
         pushId id1;
         x := 0;
         f ();
         checkId id1;
         x := 1;
         f ();
         checkId id1;
         !x in
            popId ();
            return_stat1
```

La déclaration des fonctions popId(), pushId, checkId et generateId() sont insérées au début du programme. Le programme est ensuite traduit, et on retourne le résultat.

L'écriture de la traduction se fait avec l'objet TokenStreamRewriter, qui permet de modifier directement le flux de token lu par le parseur :

```
CharStream input = new ANTLRFileStream("input");
TLexer lex = new TLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lex);
T parser = new T(tokens);
TokenStreamRewriter rewriter = new TokenStreamRewriter(tokens);
parser.startRule();
```

On peut ensuite modifier le flux de token dans les appels de listener :

```
private static final String INIT_PILE = "let␣stack␣=␣ref␣[]␣;\n";

public void enterProgram(RefMLParser.ProgramContext ctx) {
   rewriter.insertAfter(ctx.start.getStartIndex()-1, INIT_PILE);
}
```

4.2 Implémentation en OCaml

https://gitlab.com/lo.lasherme/compilateur-cameuuuuule/-/tree/master

L'objectif de cette implémentation est le même que celle en Java. Malheureusement, par manque de temps, elle n'a pas pu être achevée. Elle peut néanmoins repérer les fonctions à transformer et ajouter la fonction pushId et popId. Pour continuer, il faudrait ajouter le repérage des appels de fonction externe à l'intérieur des fonctions à transformer et faire l'ajout du checkId ; ajouter tous les en-têtes de notre tranformation — la définition des fonctions popId, pushId, etc..., notre référence de pile et notre référence d'identifiant.

Pour cette implémentation nous avons utilisé ocamllex et ocamlyacc afin de pouvoir définir notre syntaxe et notre sémantique. Une fois cela réalisé, nous obtenons un arbre de syntaxe sur lequel nous allons baser tout notre raisonnement. Nous le parcourons une première fois à l'aide de la fonction $check_a st_b is$. Une fois ce parcours réalisé, nous stockons les sous-fonctions de notre programme à transformer dans une liste. Ensuite à l'aide de $transformation$ nous reconstruisons l'arbre avec l'ajout de noeud qui permettent la transformation. Enfin, nous l'affichons à l'aide de la fonction "affiche".

Voici un exemple de ce que le programme peut réaliser comme pseudo-transformation :

```
let test_1 = 0 in
        let test_2 a = a + 2 in
                let test_3 = fun f -> f() in
                        extern test_2 test_3 test_1

let test_1 = 0 in
   let test_2 a =
      push id_0;
      a + 2;
      pop () in
         let test_3 =
            push id_1;
            fun f -> f ();
            pop () in
               extern test_2 test_3 test_1
```

5 Conclusion

Pour réaliser au mieux ce projet d'initiation à la recherche, nous avons dû dans un premier temps passer par une phase de découverte et d'apprentissage afin de pouvoir s'approprier le sujet. En premier lieu, nous avons étudier le livre de Gilles Dowek[2] pour avoir une bonne idée de la théorie des langages de programmation. Par la suite, pour ancrer ces notions, nous avons suivit l'introduction du cours de Xavier Leroy[5] et le livre de Benjamin C. Pierce[1]. Ce dernier nous a orienté vers le sujet de nos recherches. En effet, il nous a permis de comprendre comment introduire les exceptions au sein de notre langage. En plus de ces lectures, nous avons dû apprendre OCaml pour pouvoir raisonner sur un exemple concret de langage fonctionnel.

Ce projet nous a permis d'avoir une idée globale du travail de chercheur. En effet, nous avons pu observer plusieurs phases au cours de ces six mois. La première a été la recherche lié à l'apprentissage. La seconde a été la réflexion afin de trouver l'idée pour notre transformation. Ensuite, il y a eu une phase plus concrète où nous avons essayé de mettre en pratique ce que nous avons imaginé durant la phase de réflexion. Enfin, la phase de rédaction de ce rapport, permettant de structurer et partager notre travail.

Pour conclure, il est certain que notre sujet, la compilation *fully abstract*, est un domaine assez complexe. Le simple fait de proposer une transformation de programme pour un petit langage comme le nôtre a été assez difficile à réaliser. Il va sans dire qu'effectuer une transformation de programme pour préserver les abstractions offertes par celui-ci sur un vrai langage de programmation est bien plus compliqué que dans notre cas.

Références

[1]Benjamin C. Pierce, *"Types and Programming Languages"*, Chapitres 2,3,5,8,9,11 et 14, 2002

[2]Gilles Dowek, Jean-Jacques Lévy, *"Introduction à la théorie des langages de programmation"*, Les éditions de l'école polytechnique, 2006

[3]Marco Patrignani, Amal Ahmed, Dave Clarke, *"Formal Approaches to Secure Compilation : A Survey of Fully Abstract Compilation and Related Work"*, ACM Computing Surveys, Article No. : 125, 2019

[4]Terence Parr, *"Language implementation patterns"*, 2009

[5]Xavier Leroy, *"Des expressions et des commandes : la sémantique d'un langage impératif"*, `https://www.college-de-france.fr/site/xavier-leroy/course-2019-11-28-09h30.htm`, (consulté 02.2020)