Rapport Projet Traduction Des Langages

Projet de Programmation Fonctionnelle et de Traduction des Langages

GRETHEN Clementine
WEISBECKER Lisa
SHRIVASTAVA Anushree

Table des matières

Introduction

Ce projet, dans l'UE de Programmation Fonctionnelle et de Traduction Des Langages, consiste à étendre le compilateur du langage RAT réalisé en TP pour traiter de nouvelles constructions : les pointeurs, le bloc else optionnel dans la conditionnelle, la conditionnelle sous la forme d'un opérateur ternaire, les boucles 'loop' à la RUST. Le compilateur sera écrit en OCaml. Durant les TP, nous avons implanté le compilateur RAT basique. Nous aborderons les choix de conception et les ajouts que nous avons faits au langage et à l'AST pour ces extensions. Voici les fichiers que nous avons :

ast.ml contient deux interfaces. Une donnant la structure générale des arbres créés par les 4 passes différentes : TDS, Type, Placement, et Code, l'autre donnant l'affichage de ces arbres. Il contient aussi des modules comportant les definitions du type des AST. lexer.mll réalise l'analyse lexicale. type.ml contient les types utilisés durant les TPs et ceux ajoutés, notamment pour la partie pointeur et enumération. parser.mly réalise l'analyse syntaxique et la construction de l'arbre abstrait Les fichiers "passe*.ml" contiennent les codes des différentes passes. Les fichiers "test*.ml" contiennent les tests des passes ainsi que ceux des extensions que nous avons ajoutées

1 Extension du langage RAT

1.1 Ajout des pointeurs

Nous nous sommes en grande partie appuyé sur le td que nous avons réalisé sur le sujet. Un pointeur est une variable qui contient l'adresse d'un autre objet informatique. Les opérateurs à ajouter son et *. Nous avons donc commencé par modifier l'AST en créant dans expression : Null, New, Adresse et Affectable. Cela implique la création du type Affectable dans l'AST et l'ajout des points précédents dans le type Expression. Ensuite, nous avons également modifié le lexer.mll pour y ajouter les différents token :

"new", NEW ;

"null", NULL ;

"break", BREAK ;

"" ADRESSE "

Et bien sûr, nous avons ajouté la lexicographie des pointeurs dans le parser. Le type affectable concerne un accèes sur un affectable, qui est un type ajouté et qui comprend Ident et Valeur.

Pour traiter les affectables, dans la passe des identifiants, doit montrer la différence entre une lecture et une écriture, nous avons donc rajouté le paramètre modif. On notera que selon que l'affectable est en partie droite ou gauche d'une affectation, le traitement n'est pas même : par exemple, une constante est autorisée en partie droite, mais pas en partie gauche. Il y a plusieurs possibilités pour résoudre ce problème, ici nous en proposons une avec un booléen (modif) qui indique si l'affectable est modifié (partie gauche d'une affectation) ou pas (les autres cas).

Il a fallu également modifier le fichier de type pour traiter le nouveau type : Pointeur of typ (notamment gérer getTaille en disant qu'un pointeur a une taille 1 en mémoire). Lors de la passe de gestion des identifiants, l'AST est modifié pour remplacer les identifiants par un pointeur sur leur information.

Lors de la passe de typage, les types sont supprimés de l'AST car ajoutés aux informations (pas de changement particulier pour les pointeurs) et l'AST est modifié pour la résolution de surcharge (là

encore pas de traitement particulier).

Cet ajout étai assez long, car il y'a plusieurs fichiers à modifier et c'est un ajout complet (on ne pouvait pas se calquer sur ce qui avait été fait précédemment.). TEST REALISES : les tests sont visibles avec un nom explicite dans les fichiers de test, tout fonctionne sans problème.

Vous pouvez retrouver les différents tests dans le fichier "tests", : "testPointeuri.rat", ils sont dans les fichiers "avec fonction" pour la passe de gestion des identifiants. Pour la passe de placement, nous avons rajouté des pointeurs dans "test1.rat" pour vérifier le placement mémoire d'un pointeur. Pour la passe de typage, on retrouve les tests dans "type"/avecfonction/"testPointeuri.rat". Les tests passent sans problème.

1.2 Conditionnelle sans else

Nous souhaitons ne pas être obligé de fournir un bloc else à la conditionnelle. Une règle de production est ajoutée à la grammaire :

— I → if E BLOC Cet ajout était assez simple, car il suffisait de se calquer sur la réalisation de la conditionnelle et d'enlever l'argument du deuxième bloc. Pour cela, nous avons ajouté dans l'ast, dans le type instruction :

Conditionnellesanselse of expression * bloc

Pour chaque passe, nous avons copié la fonction d'analyse d'une conditionnelle classique en enlevant l'analyse du deuxième bloc : on analyse l'expression puis on analyse le bloc. Un ajout dans le parser est également nécessaire pour reconnaître la syntaxe de ce genre de conditionnelle lors de la compilation.

Les passes de typages, d'identifiants et de placement sont donc évidentes : analyse de l'expression, puis du bloc (on ne détail pas plus cette partie, car elle a été vue en td et tp). Pour la génération de code, on utilise des fonctions telles que "analysecodeexpression", "getEtiquette", "analysecodebloc" pour générer des étiquettes, analyse les expressions et les blocs de codes.

On prend en entrée les variables "c" et "b" qui sont des conditions et un bloc de code respectivement, puis on utilise "analysecodeexpression" sur c pour analyser c, suivi de l'utilisation de "getEtiquette" pour obtenir un label pour le début de la condition (iffc) et un autre label pour la fin de la condition (fic). On va ensuite utiliser l'instruction jumpif 0 iffc qui va sauter vers l'étiquette iffc si la condition est vraie (0) , après cela, on va analyser le bloc de code b via "analysecodebloc" puis ajouter les étiquettes iffc et fic.

TESTS REALISES : Pour cette partie, nous n'avons pas besoin de beaucoup de tests, car il n'y a pas de cas de blocage, et de retour d'erreur (pas de surcharge, pas d'identifiants,...). Nous avons créé 'testConditionnelleSansElse.rat" dans les tests des gestionid, qui réalise une boucle if classique sans else. Les tests passent sans problème.

1.3 La conditionnelle sous la forme d'un opérateur ternaire

Nous souhaitons, comme en C, permettre d'écrire la conditionnelle sous la forme d'un opérateur ternaire : On ajoute donc une règle de production à la grammaire : —E→(E ?E :E).

Cela se traduit par un ajout dans l'ast dans le type expression :

— ConditionnelleTernaire of expression * expression * expression

On voit également l'apparition de deux nouveaux terminaux : " ?" et " :". Il faut donc les ajouter dans le lexer.mll avec ces deux lignes :

— "" ADRESSE — " ?" QUESTION

De plus, il faut signaler la lexicographie de la conditionnelle ternaire dans le parser. Concernant les différentes passes, il faut rajouter un case au match dans les analyses des expressions. Pour la passe des identifiants, il s'agit simplement d'analyser les expressions des trois expressions contenues dans la conditionnelle ternaire : on n'a pas d'identifiant associé à cet objet, donc il est inutile d'ajouter une info dans la tds associé à la conditionnelle ternaire.

Dans la passe de typage, il est nécessaire d'assurer que les types sont cohérents : la première expression (de ConditionnelleTernaire of expression * expression * expression) doit être un booleen et les deux autres expressions doivent avoir le même type. Ainsi, on peut raise des exceptions de type "typeInattendu".

Dans la passe de placement mémoire,il n'y a rien à traiter en plus. Dans la passe de génération de code,on utilise alors deux étiquettes, "sinon" et "finSi", ainsi que des fonctions supplémentaires, "getEtiquette()" et "analysecodeexpression()".

Le code analyse ensuite la première expression e1 et génère une instruction jumpif 0 vers l'étiquette "sinon" si l'expression évalue à faux. Il analyse ensuite e2 et génère une instruction jump vers l'étiquette "finSi". Ensuite, le code génère l'étiquette "sinon" et analyse e3. Enfin, le code génère l'étiquette "finSi" Ce bloc de code génère un code machine qui effectue une instruction de saut conditionnel ou de branche basé sur l'évaluation de l'expression e1, exécute un bloc de code spécifique si e1 évalue à vrai, et exécute un bloc de code différent si e1 évalue à faux.

TEST Realisés : Pour les tests, nous avons créé des fichiers avce un nom explicite.

1.4 Loop à la Rust

Notre de choix de conception, un peu lourd, se base sur la réalisation de deux types de boucle (donc deux types de Break et deux types de continue). Modification dans l'AST :

— Loopetiquette of string * bloc

(* Break associé à un Loop Rust avec une etiquette *) — Breaketiquette of string

(* Continue associé à un Loop Rust avec une etiquette *) — Continueetiquette of string

(* Boucle loop à la Rust sans étiquette*) — Loop of bloc

(* Break associé à un Loop Rust sans etiquette *) — Break

(* Continue associé à un Loop Rust sans etiquette *) — Continue

(String devient infoast pour la suite) Nous pensons qu'il aurait été plus simple de faire un seul type de loop avec éventuellement une etiquette à undefined. Pour le lexer,il a fallu ajouter les tokens en conséquence : "break", BREAK ;

"continue",CONTINUE ;

"loop", LOOP ;

Puis il faut rajouter dans le parser la reconnaissance syntaxique. On a aussi créé des nouvelles info dans la tds : infoBoucle(string) qui permet de stocker l'information associée à une Loop, infoBreak(String) pour stcoker l'information associée à un break et InfoContinu (String) pour stocker l'information associée à un continu. Passe des identifiants :

On s'assure que les break sont bien contenus dans une boucle loop, qu'ils correspondent bien à l'identifiant de la loop, et on va créer une infoBreak(n)pour gérer les différents break dans une même loop. Pour la boucle, on va créer une InfoBoucle(string) pour ajouter la boucle à la TDS. On réalise le même traitement pour Continue. Quand il n'y a pas d'étiquette, on réalise le même traitement sans prendre en compte les conflits d'identifiant : un break dans une loop fonctionne peu importe (respectivement avec un continue).

Passe de Typage :

Il s'agit simplement d'analyser le bloc de la loop.

Passe de placement : Rien de plus à faire : l'analyse du bloc s'en charge.

Passe de génération de code

* Pour les loop avec étiquette :

Pour analyseloop ,on "match" sur la fonction "infoasttoinfo id" pour obtenir la valeur "InfoBoucle(étiquette)". Si cette correspondance est réussie, le code définit deux nouvelles variables "labelDebut" et "labelFin" en appelant la fonction "label" sur des chaînes de caractères concaténées de "étiquette" et "start" ou "end". Ensuite, on concatène le résultat de "labelDebut" avec "bloc" (le résultat de l'appel à "analysecodebloc b") et "labelFin" et renvoie la chaîne concaténée finale. Si cela échoue, on lèvera une erreur "failwith" avec le message "Internal error"

La deuxième fonction "Breaketiquette" et la troisième fonction "Continueetiquette" sont assez similaires, elles utilisent également une expression "match" sur "infoasttoinfo id" pour obtenir la valeur "InfoBoucle(étiquette)" Si la correspondance est réussie, la deuxième fonction "Breaketiquette" définit une nouvelle variable "labelFin" en appelant la fonction "label" sur la chaîne concaténée de "étiquette" et "end", puis renvoie une instruction de saut vers "labelFin" Si la correspondance est réussie, la troisième fonction "Continueetiquette" définit une nouvelle variable "labelDebut" en appelant la fonction "label" sur la chaîne concaténée de "étiquette" et "start", puis renvoie une instruction de saut vers "labelDebut" Si cela échoue, on lèvera une erreur "failwith" avec le message "Internal error" La fonction Loopetiquette génère une boucle, tandis que les fonctions Breaketiquette et Continueetiquette permettent de sortir ou de continuer une boucle, respectivement, et elles utilisent la variable "étiquette" comme étiquette pour identifier la boucle spécifique.

TESTS : les tests sont visibles avec un nom explicite dans les fichiers de test, tout fonctionne sans problème.

1.5 Réalisation de la gestion de la surcharge des fonctions

Ayant réalisé une reflexion prémiliminaire sur la surcharge pendant le partiel , nous avons décidé de l'implenter pour ce projet.

Ainsi, on rappelle que la surcharge permet d'avoir deux noms similaires pour deux fonctions, à conditions que les paramètres soient différents. L'idée générale que j'ai eu partiel, est lors de l'appel de fonction dans la gestion des identifiant, on en ressort une liste contenant l'ensemble des fonctions du nom recherché et on trouve la fonction correspondante lors de la passe de typage (types des paramètres correspondants).

MODIFICATION DE L'AST :

Dans l'astType on change le type de AstTds.AppelFonction :

— AppelFonction of (Tds.infoast list) * expression list

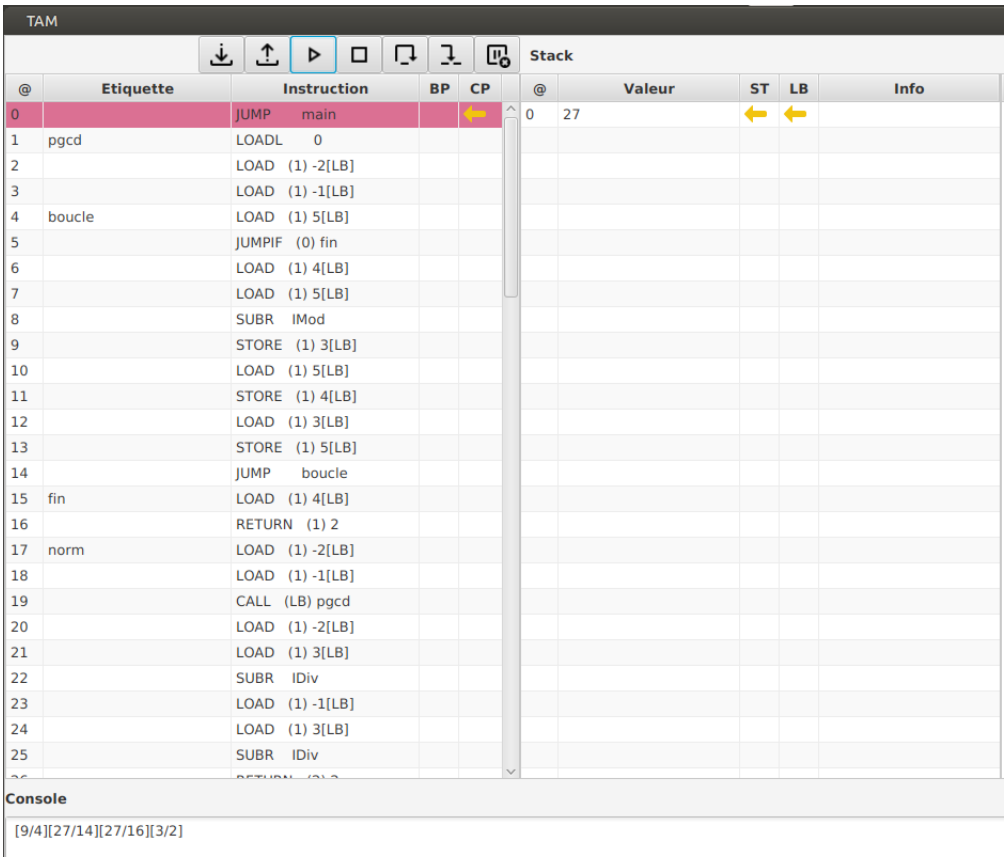
Ici, le premier paramètre correspond la liste des infoFun (cf l'idée exprimée) et la liste d'expression de la fonction.

Dans la passe des identifiants, dans l'analyse fonction, on ajoute la infoFun dans le mainTDS, on ajoute les paramètres dans la tdsf (tds fille) de la fonction. Pour l'appel de fonction, on recherche globalement les fonctions (fonction auxiliaire codée pour retourner la liste des fonctions de nom name , afin de gérer la surcharge). On s'assure que la liste ne contient que des infoFun.

Dans la passe de typage, dans l'appel de fonction, on part d'une liste d'infoFun, il faut donc trouver la correspondance avec la fonction souhaitée (vérifier la correspondance de typage des paramètres). Il faut donc analyser chaque élément de la liste et vérifier la comptabilité des types avec une des listes des paramètres stockés (en analysant l'expression). Pour l'analyse de fonction, on modifie dans la tds le type de retour et les types des paramètres. Nous regardons si la fonction en entréee ne contient pas les mêmes parametres que celles du mˆeme nom d eja stockéees. Dans le cas de deux fonctions déclaréees avec le même nom et les mêmes paramètres, nous levons l'exception DoubleDeclaration, sinon, on ajoute la liste des paramètres à la liste de liste d'InfoFun.

2 Problèmes avec la passe de génération de code et tests avec Itam

Nous avons rencontré certaines difficultés lors des tests sur la passe de génération de code, notamment des erreurs en java dont nous avons eu du mal à trouver la source. Nous avons donc commencé par extraire les résultats des tests afin de les tester avec itam et confirmer qu'on obtenait bien un code conforme à TAM.
Par exemple, pour le test "complique.rat", on a obtenu le résultat suivant :



Le résultat en fin d'exécution est cohérent avec le résultat attendu par le test. Après quelques modifications, notamment sur "analyse_code_affectable" (ajout d'un booléen permettant de faire une disjonction de cas entre affectable et affectation), tout les tests sans fonction passent.
Après corrections supplémentaires, seul le test "testfonction5.rat" ne passe pas, mais nous ne sommes malheureusement pas parvenu à cerner le problème.
Nous avons ensuite ajouté des tests pour les loop, la conditionelle ternaire et les pointeurs. Seul le test Loop ne passe pas. Impossible cependant d'obtenir une sortie textuelle à ce test (erreur), nous n'avons donc pas pu tester avec itam à ce niveau là. Nous ne sommes pas parvenues à régler ce problème sur la génération de code pour les boucles également.
Sur notre environnement personnel, les tests pour tam ne fonctionnent pas (problème de version Java), si besoin, compilerVersFichier "tests/tam/sansfonction/fichiersRat/testif1.rat" "test.txt" ; ; qui affiche les résultats, si une erreur java apparaît.

3 Conclusion

En conclusion, ce projet nous a permis de comprendre le fonctionnement d'un compilateur et d'une traduction de langage. Nous avons pu nous améliorer au langage fonctionnel Ocaml en utilisant notamment beaucoup d'itérateur. Ce projet a été un peu difficile pour nous, car nous n'avions pas bien compris l'objectif des TP. Nous avons donc beaucoup travaillé sur ce projet, notamment pendant les vacances (ou notre environnement informatique n'était pas idéal). Mais au final, nous avons bien compris les différentes notions, malgrés le fait que cela nous a pris peut-être trop de temps.

Les premiers ajouts ne nous ont pas posé de problème, cependant les pointeurs et la loop ont été plus compliqué à gérer. La passe de génération de code ne fonctionne pas totalement, nous avons passé

beaucoup de temps à essayer de gérer cela, mais nous avons choisi de continuer et de compléter le projet autrement.

(A noté que nous sommes 3, certes, mais étant donné les difficultés et les problèmes de santé d'Anushree, ce projet représente une charge de travail équivalente à deux personnes).

Ce qu'il manque : Nous nous sommes rendu compte à la fin du projet (trop tardivement et n'ayant plus accès à l'environnement) que pour les Loop, nous avons fait une erreur de syntaxe : nous avons cru que les loop étaient de la forme : " a loop " au lieu de (a :loop). Cette erreur est très facilement corrigeable en modifiant le paser : n=ID DEUXPOINTS LOOP b=bloc. De même nos tests ne sont pas très complets pour certains points (manque de temps à cause d'un long blocage sur la génération de code).