Rapport Traduction des Langages

Hajem Yassine - Murugesapillai Vithursan

Département Sciences du Numérique - Deuxième année
2023-2024

Table des matières

Table des figures

1 Introduction

Le compilateur RAT est un compilateur développé pour le langage RAT, un langage de programmation conçu pour comprendre les concepts fondamentaux de la compilation. En illustrant les différentes étapes du processus de compilation, du code source jusqu'à l'exécution. Le compilateur RAT met en oeuvre les différentes étapes du processus de compilation. Cela inclut l'analyse lexicale, l'analyse syntaxique, le placement en mémoire et la génération de code. Le code source est donc analysé et mis sous une arborescence dans laquelle nous allons gérer les identifiants dans le passTdsRat avec la table des symboles, puis gérer le typage dans le passeTypeRat, gérer le placement en mémoire avec les variables et les données qui sont organisées et stockées en mémoire lors de l'exécution d'un programme avec le passePlacementRat et enfin la génération de code TAM en assembleur à partir du code source avec le passeCodeRatToTam.

Le but du projet de programmation fonctionnelle et de traduction des langages est d'étendrele compilateur du langage RAT réalisé en TP de traduction des langages pour traiter de nouvellesconstructions, qui sont : les pointeurs, les tableaux, les boucles "for" et les Goto. Nous allons implé-menter le compilateur en OCaml. Pour chacune des nouvelles constructions nous allons introduireles différentes règles à implémenter ainsi qu'une explication pertinente sur leur traitement. En ex-pliquant l'évolution de l'AST. Nous donnerons également les jugements de typage afin de montrerla cohérence de la gestion des types dans notre projet. Nous donnerons également le traitementassocié à chacune des constructions demandées et enfin nous vous pourrez trouver les tests unitaireset d'intégrations dans les livrables qui justifient le bon fonctionnement de notre compilateur RAT.

Pour compléter le compilateur, nous suivrons les étapes de traitement de l'arbre abstrait jusqu'à la réalisation des tests unitaires et d'intégration comme énoncé dans le sujet. Finalement ce projet offre une opportunité d'appliquer les connaissances acquises en programmation fonctionnelle et en traduction des langages à des concepts avancés tels que les pointeurs, les tableaux et les structures de contrôle étendues.

2 Extensions du langage RAT

2.1 Les pointeurs

RAT étendu permet de manipuler les pointeurs à l'aide d'une notation proche de celle de C :

- $A \rightarrow (*A)$: déréférencement : accès en lecture ou écriture à la valeur pointée par A ;
- $TYPE \rightarrow TYPE*$: type des pointeurs sur un type TYPE ;
- $E \rightarrow null$: pointeur null ;
- $E \rightarrow (new\, TYPE)$: initialisation d'un pointeur de type TYPE ;
- $E \rightarrow \& id$: accès à l'adresse d'une variable.

2.1.1 Evolution de l'AST/Lexer/Parser

Pour le traitement des pointeurs, nous avons rajouté d'abord un nouveau TYPE : Pointeur of TYPE, pour pouvoir utiliser les pointeurs.

Ensuite, nous avons modifié l'AST en ajoutant le type Affectable, pour l'identificateur ($A \rightarrow id*$) et le déférencement ($A \rightarrow *(A)$). Pour les autres opérations, nous avons ajouté des nouvelles expressions : Null pour le pointeur null, New pour initialiser un pointeur et Adresse pour accéder à l'adresse d'une variable.

Suite à cela, nous avons fait les modifications nécessaires pour le lexer.ml afin que le code soit transformé en token (ajout de tokens NULL, NEW et ADRESSE) et le parser.ml avec l'ajout des règles de grammaire.

2.1.2 Jugement de typage

$$\frac{\Gamma \vdash A : Pointeur(\sigma) \equiv \sigma *}{\Gamma \vdash (*A) : \sigma}$$

$$\frac{}{\Gamma \vdash null : Undefined}$$

$$\frac{}{\Gamma \vdash (new\, TYPE) : Pointeur(TYPE)}$$

$$\frac{\Gamma(id) = \sigma}{\Gamma \vdash \&\, id : Pointeur(\sigma)}$$

2.1.3 Traitements

Pour l'extension du passeTdsRat, nous avons cherché à traiter les Affectables de la manière suivante. Nous avons introduit une fonction nommée : `analyse_td_affectable`, qui prend en paramètre une table des symboles (tds), un affectable (a), et un booléen (modif) indiquant si l'affectable est situé à gauche d'une affectation. La fonction vise à analyser l'affectable et à retourner une expression de type AstTds.affectable. Elle effectue également des vérifications pour assurer la bonne utilisation des identifiants et génère des erreurs en cas de mauvaise utilisation. En effet, la fonction récursive vérifie si l'identifiant existe dans la Tds, si elle n'est pas trouvée une exception est levée car l'identifiant n'a pas été déclarée, sinon on récupère l'information et selon celle-ci on applique un traitement différent. Si l'identifiant est une variable (InfoVar), elle renvoie AstTds.Ident(info), si c'est une InfoConst elle vérifie si l'affectable est à gauche d'une affectation. Si c'est le cas, elle lève une exception car une constante ne peut pas être à gauche cela reviendrait à modifier la valeur d'une constante qui doit être immuable tout le long d'un l'algorithme par principe, d'où la présence du booléen modif qui permet de vérifier si l'affectable est à gauche de l'affectation. Également on ne veut pas avoir de fonction ou étiquette à gauche d'un affectation d'où la présence d'exception. Enfin la récursivité intervient si l'affectable est un déréférencement (AstSyntax.Dref(a)). On applique récursivement la fonction `analyse_td_affectable` à l'affectable (a) contenu dans le déréférencement et renvoie un nouvel affectable de type AstTds.affectable avec le déréférencement appliqué. Pour le traitement du typage. On fait apparaitre le type lié à la variable et un entier si c'est une constante. En conséquence on allouera en espace mémoire pour la gestion de passe de mémoire, la taille correspondant au type de la variable analysée.

2.2 Les tableaux

RAT étendu permet de manipuler les tableaux à l'aide d'une notation proche de celle de C :

- $A \rightarrow (A\,[\,E\,])$: accès en lecture ou écriture à la case d'indice E du tableau A ;
- $TYPE \rightarrow TYPE\ []$: type des tableaux de TYPE (la syntaxe est différente de celle de C pour simplifier l'écriture de la grammaire) ;
- $E \rightarrow (new\ TYPE\ [\,E\,])$: création d'un tableau de TYPE et de taille E.
- $E \rightarrow \{\,CP\,\}$: initialisation d'un tableau avec un ensemble de valeurs

2.2.1 Evolution de l'AST/Lexer/Parser

Pour traiter les tableaux, on rajoute dans un premier temps un nouveau type Tableau de type qui nous permettra de les distinguer avec les autres types.

On crée ensuite, dans l'Ast, un nouveau affectable qui permet d'accéder à une case d'un tableau *Acces* qui prend un affectable et une expression. On aura aussi deux nouvelles expressions, une qui crée le tableau $(new\ TYPE\ [\,E\,])$ et une qui initialise le tableau avec des valeurs CP.

2.2.2 Jugement de typage

$$\frac{\Gamma \vdash A : \sigma[] \quad \Gamma \vdash e : \text{Int}}{\Gamma \vdash (A[e]) : \sigma}$$

$$\frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash (\text{new TYPE[e]}) : \text{TYPE[]}}$$

$$\frac{\Gamma \vdash e_1, e_2, ..., e_n : \sigma}{\Gamma \vdash \{\, CP \,\} \equiv \{e_1, e_2, ..., e_n\} : \sigma[]}$$

2.2.3 Traitements

Dans le passeTdsRat, pour traiter l'affectable Acces on renvoie un AstTds.Acces qui analyse l'affectable et l'expression associée. Concernant les expressions, pour le Newtab on analyse l'expression spécifiant la taille du tableau dans la table des symboles. Le résultat de cette analyse est stocké pour qu'ensuite, une nouvelle instruction AstTds.NewTab soit créée avec le type du tableau et l'expression de taille analysée. Pour l'initialisation du tableau, on vérifie la bonne utilisation des identifiants dans la liste des valeurs. Chaque élément de la liste est analysé dans la table des symboles. Pour le passeTypeRat, plus spécifiquement pour le Newtab, il faut vérifier que le type associé à la taille du tableau est bien un int, sinon on lève une exception. Pour InitTab, on vérifie que tous les éléments de la liste ont le même type et on renvoie un tableau de ce type. Le passePlacement garantit que les tableaux sont toujours de taille 1 dans la pile principale, et on stocke ces éléments dans le tas "ST". Dans notre compilateur, on stocke bien les éléments des tableaux et on prépare le bon nombre de cases dans le tas pour chaque tableau mais l'accès pour affecter des valeurs ou accéder à des cases d'un tableau rencontre des problèmes, sans cela nous aurions pu rendre un compilateur parfaitement fonctionnel.

2.3 Les boucles for

RAT étendu permet d'écrire des boucles "for" à l'aide d'une notation proche de celle de C :

- $I \rightarrow for\ (\ int\ id = E;\ E; id = E)\ BLOC$

Le premier paramètre correspond à la définition et l'initialisation de l'indice de boucle. Le second paramètre correspond à la condition d'arrêt de la boucle. Le dernier paramètre correspond à l'évolution de l'indice de boucle .

2.3.1 Evolution de l'AST/Lexer/Parser

Pour la boucle For, on rajoute dans l'Instruction de l'Ast une nouvelle instruction. Dans l'AstSyntax le For prend le nom de l'indice, une expression pour initialiser l'indice, une expression de la condition d'arrêt, une expression de l'affectation pour l'incrémentation et un bloc à exécuté. Le AstSyntax ne prend pas en compte le type de l'indice étant donné que c'est toujours un entier et que le parser oblige l'utilisateur à saisir un 'int'. Cette instruction évolue avec l'AstTDS qui rend le For composé d'une instruction de type Déclaration, une expression pour l'arrêt de boucle, une instruction de type Affectation pour l'incrémentation de l'indice et un bloc.

2.3.2 Jugement de typage

$$\frac{\Gamma \vdash \text{e1} : \text{Int} \qquad \Gamma \vdash \text{e2} : \text{bool} \qquad \Gamma \vdash \text{e3} : \text{Int} \qquad \Gamma\text{,(id,Int)} \vdash \text{bloc} : \sigma}{\Gamma \vdash \text{for (int id} = \text{e1 ; e2 ; id} = \text{e3) bloc} : \sigma}$$

2.3.3 Traitements

Dans le premier PassTDS, il est nécessaire de respecter la transformation choisie pour l'Ast. On crée alors une nouvelle instruction pour la déclaration de l'indice de la boucle et une autre pour l'affectation de l'incrémentation.

Le choix de transformer l'Ast a pour but de simplifier le traitement de la boucle. On utilisera les fonctions déjà codées pour le traitement des deux instructions et de l'expression. En effet, Le traitement de l'instruction de boucle for s'effectue en vérifiant d'abord si l'indice spécifié dans la boucle est le même que celui utilisé pour vérifier la cohérence d'utilisation d'indice, puis en assurant que cet indice est déclaré dans la table des symboles globale. Si l'indice n'est pas déclarée, une déclaration est effectuée avec une initialisation, suivie de l'analyse de l'expression conditionnelle d'arrêt de la boucle, d'une nouvelle instruction d'affectation pour l'incrémentation et le bloc de la boucle est également analysé. Si l'indice est déjà déclaré, il est traité comme une déclaration si c'est une étiquette car il est possible de déclarer des variables qui ont le même nom qu'une étiquette, sinon, une exception est levée pour éviter une double déclaration. Pour le typage, on analyse la condition d'arrêt en vérifiant que c'est bien un booléen, si cela est conforme on procède successivement à l'analyse de l'incrémentation de l'indice de boucle et du bloc de la boucle. S'il y a non conformité on lève une exception.

2.4 Goto

RAT étendu permet d'écrire des sauts à l'aide de l'instruction goto. Cette instruction permet de sauter à un point précis du programme déterminé à l'avance. Pour ce faire, des instructions peuvent être marquées à l'aide d'étiquettes. Une étiquette est un nom suivi du caractère ' :'.

- $I \rightarrow goto\ id;$: à l'exécution, l'instruction suivante sera celle marquée par l'identifiant ;
- $I \rightarrow id:$: marque l'instruction qui suit comme destination possible d'un goto.

2.4.1 Evolution de l'AST/Lexer/Parser

Pour le traitement des Goto, nous avons rajouté d'abord un nouveau type d'information INFO : InfoEtiq of string * bool, pour pouvoir utiliser identifier les Goto.

Le string correspond au nom de l'étiquette et le booléen indique si l'étiquette est définie ou non, pour pouvoir utiliser des identifiants d'étiquettes avant de les définir.

Ensuite, nous avons modifié l'AST en ajoutant le type Instruction : un Goto pour l'appel de l'étiquette ($I \rightarrow goto\ id\,;$) et IdentGoto pour la définition de l'étiquette ($I \rightarrow id\ :$).

2.4.2 Traitements

Pour les GOTO le passe le plus important à traiter est le PassTds.

Dans l'instruction 'goto id', on vérifie si l'étiquette est définie, si c'est le cas, on le renvoie avec l'étiquette correspondante. Sinon, on ajoute l'étiquette à la table des symboles avec le booléen à faux.

Dans la déclaration d'une étiquette, on vérifie si l'étiquette n'est pas déjà définie, si c'est le cas, on le définit et on met le booléen à vrai. Sinon, on renvoie une erreur dans le cas où l'étiquette est définie avec true si non on change le booléen à vrai.

Pour garantir que les étiquettes d'une part et les variables et les constantes d'autre part peuvent avoir le même nom, on regarde à chaque fois qu'on cherche une info dans la table des symboles si c'est une étiquette ou une variable/constante. Dans le cas où c'est une étiquette, on le manipule comme si on n'a pas trouvé l'info dans la table des symboles.

Il faut aussi vérifier que toutes les étiquettes sont définies à la fin du fichier. Pour cela, et après avoir analysé les fonctions et le programme principal, on parcourt la table des symboles et on regarde si toutes les étiquettes soient définies, si ce n'est pas le cas, on renvoie une erreur.

3 Conclusion

Le compilateur RAT étendu, développé dans le cadre du projet de programmation fonctionnelle et de traduction des langages, a permis d'intégrer des fonctionnalités avancées telles que les pointeurs, les tableaux, les boucles "for" et les instructions "goto". L'introduction des pointeurs a demandé la création d'un nouveau type, l'évolution de l'AST avec de nouveaux affectables, et l'ajout de règles de grammaire dans le lexer et le parser. La gestion du typage et du placement en mémoire a été adaptée pour prendre en compte ces nouvelles constructions. De surcroît, les instructions "goto" ont été introduites, nécessitant la création d'un nouveau type d'information dans la table des symboles, des ajustements dans l'AST, et une vérification spécifique lors de la définition et de l'utilisation des étiquettes. L'introduction de la boucle for ou l'étiquette goto nous ont pas posé de problèmes. Contrairement aux pointeurs ou le traitement était plus conséqent et complet et l'accès pour affecter des valeurs ou accéder à des cases d'un tableau rencontre également des problèmes. En effet, pour les tableaux nous avons localisé l'erreur mais n'avons pas réussis à la résoudre sachant qu'elle est minime et ne permet pas un fonctionnement parfait du compilateur comme nous le voulons. Cependant le projet nous a permis de voir les fondements même d'un compilateur, l'expérience acquise lors de ce projet à été très instructive et le projet a été très stimulant tant dans la conception que dans la rigueur et la méthodologie demandées. Nous vous remercions pour la conception d'un tel sujet qui a été un excellent challenge.