Rapport Projet Traduction Des Langages

Rokia SBAI

Félix PARAIN

Table des matières

2021-2022

I. Introduction :

Ce projet a eu pour but d'ajouter quelques constructions au compilateur qui traduit le langage formel RAT en un langage cible, exécuté sur une machine à pile (TAM), développé lors des séances de TP. Ce compilateur fonctionne par passe. L'Ast est construit lors de l'analyse syntaxique et est parcouru et modifié lors des autres passes. Il s'agit d'un langage simple avec trois types de bases (Bool, Int, Rat). Le typage est fort est statique (avec contrôle de type). Nous avons rajouté les pointeurs, l'assignation de l'addition, les types nommés et les enregistrements. Chaque fonctionnalité a nécessité une modification globale ou locale au niveau des passes et de l'Ast.

II. Pointeurs :

Dans le fichier lexer.mll, nous avons rajouté 2 nouveaux mots-clés (new et null) ainsi que les caractères spéciaux &. Dans le fichier parser.mly, nous avons aussi rajouté les tokens correspondants (NEW, ADRESSE, NULL), mis à jour les terminaux selon la grammaire, rajouté le type de l'attribut synthétisé des non-terminaux à savoir A pour affectable. Nous avons rajouté la taille d'un pointeur qui est 1, modifié le fichier type.ml pour prendre en compte le type Pointeur of typ et rajouté une possibilité dans la fonction est_compatible. Et pour finir, nous avons rajouté les nouvelles règles et modifié celle du programme tout en complétant la structure de l'arbre dans le fichier ast. Par la suite et en vérifiant à chaque fin d'étape la bonne exécution du compilateur, nous avons modifié toutes les passes :

PasseTdsRat :

Nous avons ajouté l'analyse de l'affectable et modifié l'instruction affectation comme suit

(* analyse_tds_affectable : AstSyntax.affectable -> AstTds.affectable *)

(* Paramètre tds : la table des symboles courante *)

(* Paramètre e : l'affectable à analyser *)

(* Vérifie la bonne utilisation des identifiants et tranforme l'affectable en une affectable de type AstTds.affectable *)

(* Erreur si mauvaise utilisation des identifiants *)

PasseTypeRat :

Nous avons ajouté l'analyse de l'affectable et modifié l'instruction affectation comme suit

(* analyse_type_exaffectablepression : AstTds.affectable -> (AstType.affectable*typ) *)

(* Paramètre e : l'affectable à analyser *)

(* Vérifie la cohérence des types, supprime la surcharge et tranforme l'affectable en une affectable de type AstType.affectable *)

(* Erreur si types incohérents et inattendus *)

PassePlacementRat :

Pas de modification

PasseCodeRatToTam :

Nous avons ajouté une fonction qui retourne la taille d'une affectable comme suit

(* getTailleAffectable : AstType.affectable -> int *)

(* Paramètre a : le pointeur *)

(* renvoie la taille du type pointé "final"*)

Nous avons également ajouté l'analyse de l'affectable et modifié l'instruction affectation comme suit

(* analyse_code_affectable : AstType.affectable -> string *)

(* Paramètre a : l'affectable à encoder *)

(* renvoie le code tam de l'affectable a *)

Jugements de typage :

$$\frac{}{\sigma \vdash \textbf{null} : \textbf{Pointeur}(\textbf{Undefined})}$$

$$\frac{\sigma \vdash t : \tau}{\sigma \vdash \textbf{new } t : \textbf{Pointeur}(\tau)}$$

$$\frac{\sigma \vdash a : \textbf{Pointeur}(\tau)}{\sigma \vdash * a : \tau}$$

$$\frac{\sigma \vdash id:}{\sigma \vdash \&id : \textbf{Pointeur}(\tau)}$$

III. Assignation de l'addition :

Dans le fichier lexer.mll, nous avons rajouté le caractère spécial += sans espace. Dans parser.mly, nous avons défini le token ADD et rajouté le squelette de la nouvelle instruction et compléter la structure de l'arbre dans le fichier ast notamment la résolution de surcharge à partir de l'AstType où l'on distingue AdditionInt et AdditionRat. Par la suite et en vérifiant à chaque fin d'étape la bonne exécution du compilateur, nous avons modifié toutes les passes :

PasseTdsRat :

AstTds.Addition (a,e) où l'on analyse l'affectable en premier, l'expression ensuite et on retourne la nouvelle instruction.

PasseTypeRat :

AstType.Addition (a,e) où l'on analyse l'affectable en premier, l'expression ensuite. Si le type retourné par les deux analyses est compatible et est compatible avec Int alors on résolve la surcharge en faisant appel à AdditionInt(na,ne), pareil pour les rationnels.

PassePlacementRat :

Renvoyer AstPlacement.AdditionInt (a,e) (respectivement AdditionRat) et le déplacement correspondant.

PasseCodeRatToTam :

Que ce soit pour l'assignation de l'addition des entiers ou des rationnels, le but est d'empiler la taille de l'affectable lus à l'adresse précédemment empilée et l'écrire à l'adresse empilée.

Jugements de typage :

$$\frac{\sigma \vdash A : Int \qquad \sigma \vdash E : Int}{\sigma \vdash A += E : Int}$$

$$\frac{\sigma \vdash A : Rat \qquad \sigma \vdash E : Rat}{\sigma \vdash A += E : Rat}$$

IV. Types nommés :

Dans le fichier lexer.mll, nous avons rajouté 1 nouveaux mots-clés (typedef) ainsi que la structure de tid qui est la chaine de caractères du type nommé commencant par une majuscule. Dans le fichier parser.mly, nous avons aussi rajouté les tokens correspondants (TID, TYPEDEF), mis à jour les terminaux selon la grammaire, rajouté le type de l'attribut synthétisé des non-terminaux à savoir TD pour la liste des typedef dans un programme. Nous avons rajouté la taille d'un pointeur qui est liée à la taille du

vrai type, modifié le fichier type.ml pour prendre en compte le type typenomme of typ*string et rajouté des possibilités dans la fonction est_compatible. Nous avons aussi rajouté, dans le fichier tds.ml (respectivement tds.mli), un nouveau type d'informations associées aux identifiants notamment InfoTypeDef. Et pour finir, nous avons rajouté les nouvelles règles et modifié celle du programme tout en complétant la structure de l'arbre dans le fichier ast. Par la suite et en vérifiant à chaque fin d'étape la bonne exécution du compilateur, nous avons modifié toutes les passes :

PasseTdsRat :

Nous avons ajouté une nouvelle fonction qui indique si c'est un type nommé ou pas, comme suit

(* retrouver_type : typ -> tds -> typ *)

(* Paramètre typ : le type à "analyser" *)

(* Paramètre tds : la tds *)

(* renvoie le même type si ce n'est pas un type nommé *)

(* Si c'est un type nommé, renvoie un type nommé avec son type réel qu'on recherche dans la tds au lieu de Undefined *)

Nous avons ajouté l'analyse du typedef pour le programme comme suit

(* analyse_tds_typedef : AstSyntax.typedef -> AstTds.typedef *)

(* Paramètre tds : la table des symboles courante *)

(* Paramètre : le typedef à analyser *)

(* Vérifie la bonne utilisation des identifiants et tranforme le typedef

en un typedef de type AstTds.typedef *)

(* Erreur si mauvaise utilisation des identifiants *)

PasseTypeRat :

Nous avons ajouté une nouvelle fonction pour relever le type du type nommé comme suit

(* type_reel : typ -> typ *)

(* Paramètre typ : le type dont on veut obtenir le type réel*)

(* renvoie le type "réel", c'est à dire le même type sans les types nommés qui sont remplacés par le type qu'ils nomment *)

Nous avons refait l'analyse du typedef pour le programme comme suit

(* analyse_type_typedef : AstTds.typedef -> (AstType.typedef *)

(* modifie le type dans le pointeur et *)

(* Erreur si types incohérents et inattendus *)

PassePlacementRat :

Nous avons analysé le typedef comme suit :

(*analyse_placement_typedef : AstType.typedef) -> (AstPlacement.typedef*)

(*Paramètre : le typedef à analyser*)

(*retourne simplement le nouveau typedef sans changement*)

PasseCodeRatToTam :

Nous avons analysé le typedef comme suit :

(* analyse_code_typedef : AstType.typedef -> string *)

(* Paramètre a : le typedef à encoder *)

(* renvoie le code tam de du typedef *)

Jugements de typage :

$$\frac{\sigma \vdash TD : void,\ \sigma' \qquad \sigma'@\sigma \vdash FUN : void, \sigma'' \quad \sigma''@\sigma' \vdash Prog : void, \sigma'''}{\sigma \vdash FUN\ PROG : void, \sigma'''@\sigma''@\sigma}$$

$$\frac{\sigma \vdash tid : \tau \qquad \sigma \vdash TYPE : \tau \quad \sigma'@\sigma \vdash TD : void, \sigma''}{\sigma \vdash typedef\ tid = TYPE\ ; TD : void, \sigma''@\sigma}$$

$$\frac{\sigma \vdash tid : \tau \qquad \sigma \vdash TYPE : \tau}{\sigma \vdash typedef\ tid = TYPE\ : void, []}$$

$$\frac{\sigma \vdash tid : \tau}{\sigma \vdash TYPE\ tid : \tau}$$

V. Enregistrements :

Dans le fichier lexer.mll, nous avons rajouté 1 nouveaux mots-clés (struct) ainsi que le caractère spécial (point). Dans le fichier parser.mly, nous avons aussi rajouté les

tokens correspondants (POINT, STRUCT), mis à jour les terminaux selon la grammaire, ajouté la taille d'un enregistrement, modifié le fichier type.ml pour prendre en compte le type enregistrement of typ*string list et rajouté des possibilités dans la fonction est_compatible. Nous avons aussi rajouté, dans le fichier tds.ml (respectivement tds.mli), un nouveau type d'informations associées aux identifiants notamment InfoChamps. Et pour finir, nous avons rajouté les nouvelles règles tout en complétant la structure de l'arbre dans le fichier ast. Par la suite et en vérifiant à chaque fin d'étape la bonne exécution du compilateur, nous avons modifié toutes les passes :

PasseTdsRat :

Nous avons rajouté l'analyse du type affectable Acces, analysé la nouvelle expression qui est une liste d'expression et rajouté 2 nouvelles fonctions comme suit

La première : (* analyse_placement_type : typ ->unit*)

(* Si le type en paramètre est un enregistrement, *)

(*affecte à chacun de ses champs son déplacement par rapport au début de l'enregistrement*)

(* ne fait rien sinon *)

La deuxième : (* analyse_tds_type : typ -> tds -> unit *)

(* Paramètre typ : le type à "analyser" *)

(* Paramètre tds : la tds *)

(* si c'est un enregistrement, ajoute les champs à la tds*)

(* ne fait rien sinon *)

PasseTypeRat :

Nous avons rajouté une possibilité dans analyse_type_affectable (Acces) et rajouté une nouvelle expression qui fait un List.map.

PassePlacementRat :

Pas de changement particulier puisque pas d'expressions et pas d'affetcable.

PasseCodeRatToTam :

Nous avons ajouté Acces à analyse_placement affectable et rajouté une nouvelle expression qui fait un List.map.

Jugements de typage :

$$\frac{\boldsymbol{\sigma} \neg \boldsymbol{A} : \quad \boldsymbol{\sigma} \neg \boldsymbol{id} :}{\boldsymbol{\sigma} \neg \boldsymbol{A}.\boldsymbol{id} \ : \ \boldsymbol{void}, []}$$

$$\frac{\boldsymbol{\sigma}, \boldsymbol{\tau} \neg \boldsymbol{CP} : \boldsymbol{void}, \boldsymbol{\sigma}'}{\boldsymbol{\sigma}, \boldsymbol{\tau} \neg \{\boldsymbol{CP}\} : \boldsymbol{void}}$$

VI. Tests :

En ce qui concerne les tests, nous avons rajouté les tests dans le fichier type.ml pour appuyer et confirmer le bon fonctionnement des différentes fonctions liés aux types rajoutés (pointeur, type nommé et enregistrement).

Pour les autres tests, nous avons créé dans le fichier fichiersRat un dossier pour chaque nouvelle construction qui contient les tests qui nous semblent pertinents.

Nous les avons bien évidemment testés dans chacun des fichiers test du compilateur.

VII. Conclusion :

Nous avons tous les deux travaillé ensemble tout au long de ce projet et nous avons beaucoup appris. Nous avons non seulement compris comment marchait un compilateur mais nous avons aussi appris à en faire un. Nous nous sommes aussi plus familiarisés avec le langage OCaml que nous avons vu en programmation fonctionnel.

Les difficultés que nous avons rencontrés concerne principalement la surcharge de type pour la gestion des types nommés mais que nous avons su surpasser, à cela s'ajoute les types récursifs avec les enregistrements que nous n'avons malheureusement pas pu parfaire.