Projet de programmation fonctionnelle et de traduction des langages

MDAA Saad — El Bouzekraoui Younes

Département Sciences du Numérique - Deuxième année
2020-2021

Contents

1 Introduction

Le but de ce projet est de ajouter les extensions du language Rat réalisé en TP de traduction des langages en Ocaml afin de traiter les nouvelles constructions

- pointeurs
- surcharge des fonctions
- types énumérés
- switch/case

Dans la suite du rapport on va aborder les changements sur la structure du compilateur fin de tp : les changements sur l'ast,tds et les différentes passe.

2 Pointeurs

2.1 Modifications de la grammaire

On a ajoute les règles suivantes :

- TYPE → TYPE *
- E → A
- A → (* A)
- A → id
- E → null
- E → (new TYPE)
- E → & id
- I → A = E

2.2 Modifications de l'Ast

- Afin de traiter les pointeurs et conformément au dernier tp, on a ajouté un nouveau type **affectable** à l'ast qui peut être un identifiant qui permet l'accès a une variable ou une valeur qui permet l'accès a la valeur pointé par un pointeur.
- On a ajouté un nouveau type dans **type.ml** : **Pointeur of typ**: un pointeur sur null a un type de Pointeur(Undefined), et deux variable de type $Pointeur(t_1)$ et $Pointeur(t_2)$ sont compatible si t_1 et t_2 sont compatible.
- Lors de la **passeCodeRatToTam** et pour l'analyse d'un affectable on a choisi de factoriser l'operation LOAD et STORE dans une unique fonction en ajoutant un deuxième paramètre de type boolean qui va indiquer si on veut effectuer un LOAD ou un STORE.

2.3 Jugement de typage

$$\frac{\sigma \vdash a : Pointeur(\tau)}{\sigma \vdash *a : \tau} \quad (1)$$

$$\frac{\sigma \vdash a : \tau}{\sigma \vdash \&a : Pointeur(\tau)} \quad (2)$$

$$\sigma \vdash null : Pointeur(Undefined) \quad (3)$$

$$\frac{\sigma \vdash a : \tau}{\sigma \vdash newa : Pointeur(\tau)} \quad (4)$$

3 Surcharge des fonctions

3.1 Modifications de la TDS

- Afin de traiter la surcharge des fonction (types des arguments) On a modifié la liste des types d'un **InfoFun** par une liste des listes des types pour prendre en compte les différentes signature qu'une fonction peut prendre, et on a défini une fonction *ajouter_signature* qui permet d'ajouter une signature a une fonction qui existe dans la tds.

3.2 Modifications de l'Ast

- Lors de la **passeTdsRat** pour analyser une fonction on cherche la signature dans la tds et si elle existe déjà on lève l'exception **DoubleDeclaration** ,sinon on ajoute la liste des types à la liste des listes des types.
- Lors de la **passeTypeRat** pour analyser une expression de type **AppelFonction(info, le)** on cherche s'il existe une signature dans la tds compatible avec la liste des paramètre sinon on lève l'exception **TypesParametresInattendus**.
- Lors de la **passeCodeRatToTam** afin de distinguer les différentes signature des fonctions lors du call on a choisi de concaténer le nom de la fonction avec ses paramètres comme label
exemple :

```
int permute (int* p1 int* p2){...}
int permute (Mois* p1 Mois* p2) {...}

permutePointeursurIntPointeursurInt
permutePointeursurEnumerationdeMoisPointeursurEnumerationdeMois
```

4 Types énumérés

4.1 Modifications de la grammaire

On ajoute les règles suivantes :

- Main → ENUMS PROG
- ENUMS → Λ
- ENUMS → ENUM ENUMS
- ENUM → enum tid IDS
- IDS → tid
- IDS → tid, IDS
- TYPE → tid
- E → tid

4.2 Modifications de l'Ast

- Afin de traiter les types énumérés, on a ajouté un nouveau type **enumeration** à l'ast qui est un couple de type de l'énumération et une liste des différentes valeurs.
- le type programme devient un triplet (liste des énumérations, liste des fonctions, bloc).
- On a ajouté un nouveau type dans **type.ml** : **Enum of string**: deux variable de type $Enum(e_1)$ et $Enum(e_2)$ sont compatible si $e_1 = e_2$.
- Lors de la **passeTdsRat** l'analyse d'un énumération est similaire à l'analyse d'une instruction de type Déclaration: On vérifie que le type de l'énumération n'est pas déjà déclaré et que toutes valeurs sont déclarés une seule fois en les ajoutant dans la tds.
- Lors de la **passePlacemmentRat** on place toutes les valeurs de type Enum _ dans les premières addresses du stack et pour les variables du bloc on commence a l'address qui correspond au nombre des valeurs de type Enum _ .
- Lors de la **passeCodeRatToTam** chaque valeur de type Enum _ sera traité comme sa valeur d'address dans le stack puisque on l'unicité des addresses et lors de la passe de typage on a éliminer les cas ou on fait des operations sur deux valeurs de type Enum _ différentes. Donc l'opérateur binaire **EquEnum** qu'on a défini pour traiter la surcharge du + sera analogue à **EquInt**

4.3 Jugement de typage

$$\frac{\sigma \vdash E_1 : Enum(n), \sigma \vdash E_2 : Enum(n)}{\sigma \vdash (E_1 = E_2) : bool} \quad (5)$$

5 Switch/case

5.1 Modifications de la grammaire

On ajoute les règles suivantes :

- I → switch (E) LCase
- LCase → Case LCase
- LCase → Λ
- Case → case tid : IS B
- Case → case entier : IS B
- Case → case true : IS B
- Case → case false : IS B
- Case → default : IS B
- B → Λ
- B → break;

5.2 Modifications de l'Ast

- Afin de traiter le switch case on a ajouté l'instruction

$$Swicth(e : expression, (e : expression, b : bloc, i : instruction)list)$$

l'instruction i correspond a **Break** si on a un *break* à la fin du bloc et a **Notbreak** sinon, de plus on ajouté l'expression **Default** de type Undefined qui correspond au dernier cas d'un switch.
- Lors de la **passeTypeRat** on vérifie que chaque expression dans la liste a un type compatible avec le type de l'expression sur laquelle on fait le switch.
- Exemple simple de génération de code :

```
test{
    int z = 8;
    switch (z) {
        case 7 :
            print z;
        case 8 :
            print (z + 1);
        default :
            print (z + 2);
            break;
    }
}

PUSH 1
LOADL 8
STORE (1) 0[SB]

LOAD (1) 0[SB]
LOADL 7
SUBR IEq
JUMPIF (0)      label152 (label cas suivant case: 8)
```

```
label161        (label cas courant case: 7)
LOAD (1) 0[SB]
SUBR IOut       (print z)
POP (0) 0
                (absence du break on passe directement au bloc suivant)
label162
LOAD (1) 0[SB]
LOADL 1
SUBR IAdd
SUBR IOut       (print z + 1)
POP (0) 0
                (absence du break on passe directement au bloc suivant)
label163
LOAD (1) 0[SB]
LOADL 2
SUBR IAdd
SUBR IOut       (print z + 2)
POP (0) 0
JUMP label150  (on a break on sort du switch)
...
...             (pareil pour les autres cas)
label150
POP (0) 1
HALT
```

5.3 Jugement de typage

$$\frac{\sigma \vdash E : \tau, \sigma \vdash E_1 : \tau, \sigma \vdash Bloc_1 : void, \sigma \vdash default : Undefined, \sigma \vdash Bloc_3 : void}{\sigma \vdash switch(E) case E_1 Bloc_1 ... default Bloc_3 : void, []} \quad (6)$$

6 Tests

6.1 Remarque

- A cause de la difference du retour a ligne entre windows et Unix il se peut que le lexer renvoie l'erreur suivante :

```
Rat.Lexer.Error("Unexpected char: \r at 109-110")
```

il suffit d'exécuter la commande suivant si c'est le cas

```
dos2unix *.rat
```

6.2 Test d'integration

- On a défini des tests dans chaque dossier **src-rat-*-test** testant des différentes cas après l'implémentation de chaque extension du language Rat à l'instar des tests déjà fourni, et à la fin on a testé les quartes extensions sur l'exemple donnée dans le sujet **src-rat-tam-test/testUltime.rat**

7 Conclusion

- Les quartes extensions sont de difficulté variables : l'ajout du pointeur a été très détaillé lors du dernier tp donc ca avance vite, par contre la difficulté principale dans le switch/case est dans la phase de génération code. L'ajout de la partie énumération nous a permit de bien comprendre la gestion du stack. Enfin l'ajout du surcharge des fonctions nous a permit de nous bien familiariser avec la structure car il nécessite une bonne compréhension de l'architecture globale du projet.
- Le code Tam généré passe les tests qu'on défini mais il est loin d'être optimale a cause du code mort surtout pour le switch/Case ,mais la passe d'optimisation n'est pas inclue dans l'UE
- En guise de conclusion ce projet est vachement intéressant : il nous a permit de nous familiariser avec les

mécanisme d'un compilateur en pratiquant les notions de la programmation fonctionnel en Ocaml toute en jouant avec structures de donnée complexe.