Compilateur du langage RAT

Meriem BOULTAM - Salahdine OUHMMIALI

Département Sciences du Numérique - Deuxième année
2021-2022

Table des matières

1 Introduction

L'objectif de ce projet est de compléter l'implémentation du compilateur du langage RAT qu'on a entamé en séances de Tp en langage OCaml. Le compilateur est basé sur quatre grands passes importantes qui modifient au fur et à mesure l'arbre syntaxique abstrait (ast) jusqu'à la génération du code en langage de machine Tam. Les nouvelles constructions qu'on a ajoutées au compilateur sont : **les pointeurs, l'opérateur d'assignation d'addition, les enregistrements et les types nommés**.

2 Pointeur

2.1 Modification du type et de l'AST

On a commencé par ajouter les nouveaux terminaux dans le lexer à savoir **null, new, &** . Ensuite, après avoir supprimé l'élément **Ident**, on a modifié le parser pour prendre en compte les nouvelles règles de la production suivantes :

- **affectable = Variable of string | Deref of affectable**
- **expression = Null**
- **expression = Initilisation of typ**
- **expression = Val of affectable**
- **expression = Adresse of string**
- **Type = Pointeur of type**
- **instruction = Affectation of affectable * expression**

2.2 Modification des passes

2.2.1 Passe de la gestion des identifiants

On commence par modifier les éléments dans l'ast en remplaçant les symboles (string) par des pointeurs vers l'information de chaque variable (Tds.infoast). Ensuite on a modifié le fichier d'analyse (PasseTdsRat.ml) En ajoutant une fonction qui fait l'analyse de notre nouveau élément (affectable), on sépare le cas où l'affectable est en affectation et le cas de l'utilisation de sa valeur à droite d'une équation dans deux fonctions : **analyse_tds_affectable_affectation** et **analyse_tds_affectable_value** qu'on utilise successivement dans l'analyse de l'affectation et dans l'expression de l'évaluation d'un pointeur. On ajoute dans les fonctions de **analyse_tds_expression** l'analyse des nouvelles expressions ajoutées.

2.2.2 Passe du typage

On commence par ajouter le nouveau type **Pointeur of typ** dans le module **type.ml** en modifiant les fonctions de taille et de compatibilité des types dont on aura besoin dans les passes suivants.

On a ajouté une fonction **analyse_type_affectable** pour l'analyse de l'élément affectable qu'on utilise dans l'analyse de l'affectation. On ajoute également dans **analyse_type_expression** l'analyse des nouvelles expressions.

2.2.3 Passe du placement mémoire

Dans cette passe, on ne change pas le module de passe, on ajoute juste la taille du pointeur dans le module **type.ml** qu'on a choisi égal à 1.

2.2.4 Passe de la génération du code

On a ajouté une fonction **analyse_affectable** pour traiter l'élément affectable et dans laquelle on distingue le cas en affectation et le cas en évaluation, et on l'utilise dans l'analyse de l'affectation et de l'expression d'évaluation. On ajoute également l'analyse des nouvelles expressions.

3 Opérateur d'assignation d'addition

3.1 Modification du type et de l'AST

On a commencé par ajouter le nouveau terminal d'assignation += dans le lexer. Ensuite on a modifié le parser pour prendre en compte les nouvelles règles de la production suivante :

— **instruction = Assignation of affectable * expression**

3.2 Modification des passes

3.2.1 Passe de la gestion des identifiants

On commence par modifier les éléments dans l'ast en remplaçant les symboles (string) par des pointeurs vers l'information de chaque variable (Tds.infoast). Ensuite on a modifié le fichier d'analyse (PasseTdsRat.ml) En ajoutant l'analyse de la nouvelle instruction **Assignation**. Comme dans l'analyse d'affectation on fait appel à la fonction **analyse_tds_affectable_affectation**.

3.2.2 Passe du typage

Comme le passe précédent, on ajoute l'analyse de l'instruction **Assignation** qui utilise la fonction **analyse_type_affectable**, qui est similaire à l'analyse d'une affectation.

3.2.3 Passe du placement mémoire

Vu que les opérations d'affectation ne modifient pas le décalage mémoire, on ne changera rien dans la passe du placement mémoire.

3.2.4 Passe de la génération du code

On a ajouté l'analyse de l'instruction **Assignation** qui utilise la fonction **analyse_affectable**, et qui est identique à l'analyse d'une affectation normale.

4 Types nommés

4.1 Modification du type et de l'AST

On a commencé par ajouter le nouveau terminal **typedef**dans le lexer et l'identifiant TID. Ensuite on a modifié le parser en modifiant la structure du programme dans l'analyseur pour permettre de reconnaitre les types nommés déclarés globalement (avant le bloc principal) et pour prendre en compte les nouvelles règles de la production suivantes :

— Pour type nommé global **typeNommeGlobal = typenomme of string * typ**
On modifie la structure du programme dans l'AST comme suit :
programme = typeNommeGlobal list * fonction list * bloc
— Pour type nommé local : **instruction = DeclTypeNomme of string * typ**
Type = TypeNomme of string

4.2 Modification des passes

4.2.1 Passe de la gestion des identifiants

On commence par modifier les éléments dans l'ast en remplaçant les symboles (string) par des pointeurs vers l'information de chaque variable (Tds.infoast), et l'ajout d'un nouveau type d'info **InfoTypeDef** dans le module tds.ml avec ces fonctions associées (modification du type info etc.) qui correspond à l'info associée à un type nommé. Ensuite on a modifié le module d'analyse (PasseTdsRat.ml) En ajoutant l'analyse de la nouvelle instruction **DeclTypeNomme** pour analyser les déclarations des types nommés locaux. Pour les déclarations globales, on a ajouté une fonction **analyse_tds_typenomme** qui ajoute les noms des types nommés globaux à la tds. On a ajouté aussi la fonction **get_type** qui permet d'obtenir le type réel d'un type nommé, et on utilise cette fonction pour modifier les types dans la déclaration etc.

On modifie également la fonction principale **analyser** car la structure de l'élément programme a été modifiée.

4.2.2 Passe du typage

On a ajouté le type **typeNomme of string** dans le module **type.ml** avec ses fonctions associées (compatibilité et taille).

On a ajouté l'analyse de la nouvelle instruction **DeclTypeNomme** pour les types nommés locaux dans laquelle on ajoute le type du type nommé dans son pointeur info_ast et on retourne **Empty** car on aura pas besoin de cette instruction dans les passes suivants.

Pour les types nommés déclarés globalement on a ajouté la fonction **analyse_type_typenomme** qui ajoute le type dans le pointeur info_ast de la variable. Et on appelle cette fonction dans la fonction principale **analyser**.

4.2.3 Passe du placement mémoire

Dans cette passe, on ne change pas le module. Il faut juste ajouter la taille du type nommé dans le module **type.ml** qu'on a choisi égale à 1. Cette taille n'a pas vraiment d'intérêt car l'interpréteur utilise le type réel du type nommé.

4.2.4 Passe de la génération du code

On a rien à changer dans ce module. Car les fonctions qui utilisent les variables ayant un type défini comme type nommé, sont déjà implémentées, et utilisent les caractéristiques du type réel du **typeNomme**.

5 Enregistrements

5.1 Modification du type et de l'AST

On a commencé par ajouter les nouveaux terminaux dans le lexer à savoir **struct**. Ensuite, on a modifié le parser pour prendre en compte les nouvelles règles de la production suivantes :

- **affectable = Variable of string | Deref of affectable | Acces of affectable * string**
- **expression = Listexp of expression**
- **Type = Struct of (typ * string) list**

Vu la difficulté qu'on a trouvé dans la passe du placement mémoire, on était amenés à modifier la structure de l'expression déclaration, de manière à y stocker les info_ast des variables de l'enregistrement.

5.2 Modification des passes

5.2.1 Passe de la gestion des identifiants

On commence par modifier les éléments dans l'ast en remplaçant les symboles (string) par des pointeurs vers l'information de chaque variable (Tds.infoast). Ensuite on a modifié le fichier d'analyse (PasseTdsRat.ml) En ajoutant une fonction qui fait l'analyse de notre nouveau élément (affectable de type accès à un enregistrement), on sépare le cas où l'affectable est en affectation et le cas de l'utilisation de sa valeur à droite d'une équation dans deux fonctions : **analyse_tds_affectable_affectation** et **analyse_tds_affectable_value** qu'on utilise successivement dans l'analyse de l'affectation et dans l'expression de l'évaluation d'une variable d'un enregistrement. On ajoute dans les fonctions de **analyse_tds_expression** l'analyse des nouvelles expressions ajoutées. On a modifié également l'analyse de la déclaration dans laquelle, dans le cas d'une déclaration d'un struct on ajoute les variables de cestruct à la tds avec les noms **(nom_struct . nom_varible)**. De manière à ce que les accès en affectation ou en évaluation se font naturellement comme en manipulant des affectables de type varibles.

5.2.2 Passe du typage

On commence par ajouter le nouveau type **Struct of (typ * string) list** dans le module **type.ml** en modifiant les fonctions de taille et de compatibilité.
Ensuite on a modifié les fonction d'analyse des affectables de type accès à un struct en associant à l'affectable son type et vérifiant qu'il est conforme au type de struct.
On a ajouté le traitement de l'expression Listexp dans l'analyse des expression.

5.2.3 Passe du placement mémoire

Dans cette passe, vu le changement sur la structure de la déclaration en y ajoutant la liste des info_ast des variables lorsque le type déclaré est un struct. Alors on a modifié l'analyse de la déclaration, de manière à décaler la base par la somme des taille des variables de struct. On a donné comme taille au type struct la valeur 0, car dans la stratégie qu'on a adoptée on considère un struct comme une suite de déclarations de variables.

5.2.4 Passe de la génération du code

On a ajouté une fonction **analyse_affectable** pour traiter l'élément affectable de type accès à une variable d'un struct et dans laquelle on distingue le cas en affectation et le cas en évaluation, et on l'utilise dans l'analyse de l'affectation et de l'expression d'évaluation. On ajoute également l'analyse des nouvelles expressions. On a modifié l'analyse de la déclaration en séparant les cas d'un type struct et les autres cas.Dans le cas d'un struct on génère le code de chaque variable comme un code de déclaration avec son expression ssociée dans la listexp.

6 Tests

On a réalisé plusieurs tests pour les différents éléments ajoutés, dans les fichiers : **src-rat-assign, src -rat-struct, src-rat-pointeurs, src-rat-typeNomme** . Et qu'on teste dans le fichier **testTam.ml**.

7 Conclusion

Dans ce projet on a fait l'implémentation d'un compilateur du langage RAT. Les nouvelles constructions qu'on a ajoutées au compilateur sont : **les pointeurs, l'opérateur d'assignation d'addition, les enregistrements et les types nommés**.
Au long du projet on a rencontré quelques difficultés à savoir par exemple :

pour les types nommés, il fallait faire un traitement différent pour la déclaration globale et la déclaration locale ce qui était compliqué au début.

Pour les enregistrements, il faut savoir le lien entre le struct et ses variables pour pouvoir faire l'accès à ces variables, la méthode qu'on a faite consiste à ajouter ces variables à la Tds et transporter leurs infos dans l'expression déclaration, nous sommes conscients que cette approche est à améliorer car elle n'est pas très optimale. Une solution alternative pour ne pas modifier la structure de la déclaration est de modifier l'analyse du bloc, à chaque fois qu'on trouve une déclaration d'un enregistrement, on crée une liste de déclarations des variables du struct avec leurs expressions associées, et on ajoute cette liste à la liste des instructions analysée dans le bloc, de cette manière le compilateur fait le décalage mémoire nécessaire pour chaque strcut déclaré.