Rapport de Projet
Programmation Fonctionnelle et Traduction des langages
Compilateur RAT

Léo Meissner et Alexis Gosselin

Département Sciences du Numérique - Deuxième année
2022-2023

Table des matières

1 Introduction

Dans la suite de ce rapport, nous présentons les étapes que nous avons suivies pour étendre le compilateur du langage RAT, à savoir : l'ajout de pointeurs, l'utilisation d'un bloc else optionnel dans les instructions conditionnelles, la possibilité d'utiliser une notation ternaire pour les instructions conditionnelles et la prise en charge de boucles "loop" similaires à celles utilisées dans le langage Rust. Nous aborderons les choix de conceptions réalisés et les ajouts que nous avons fait au langage et à l'AST.
Enfin, nous allons revenir sur la validation de notre implémentation à l'aide de différents tests unitaires.

2 Extension du langage RAT

2.1 Les pointeurs

L'intégration des pointeurs a été abordée en séance de TD. Un pointeur est une variable qui contient l'adresse d'une autre variable ou d'un emplacement mémoire. Il permet de stocker une référence à une zone de mémoire au lieu de stocker directement la valeur elle-même.

Nous avons donc, comme suggéré, ajoutés dans type.ml un nouveau type Pointeur of typ et de nouvelles expressions dans l'AST et dans parser.mly :

- Null
- New of typ
- Affectable of $affectable$
- Adresse of $string$

Nous avons également dû ajouter dans lexer.mll un caractère spécial et les tokens suivants afin que le compilateur puisse reconnaître la boucle loop :

- " & " $\{ADRESSE\}$
- "null", NULL
- "new", NEW

Dans la fonction getTaille de type.ml, nous avons rajouter le type Pointeur de taille 1 et Undefined de taille 0.

Un affectable est un type ajouté qui comprend Ident et Valeur. Le traitement d'un affecteur est différent pour une lecture ou une écriture quand il s'agit d'un pointeur sur une constante, c'est pourquoi il est nécessaire de fournir en paramètre un booléen "$modif$" à la fonction $analyse\_tds\_affectable$.
Nous ne détaillerons pas plus ici l'implémentation des pointeurs puisque la majeur partie a été abordée en TD.

Pour l'affichage des pointeurs, nous utilisons simplement le même Code que pour AffichageInt. En piste d'amélioration, on pourrait rajouter un "@" devant l'entier si c'est une adresse.

Jugement de typage

1. Expression de l'adresse $null$: $null : Pointeur(Undefined)$
2. Expression de l'accès à l'adresse d'un Ident : $\frac{\sigma \vdash id : t}{\sigma \vdash id : Pointeur(t)}$
3. Expression d'allocation de la mémoire dans le tas New : $\frac{t \vdash id : t}{\sigma \vdash new\ t : Pointeur(t)}$
4. Affectable accédé en valeur $*a$: $\frac{\sigma \vdash a : Pointeur(t)}{\sigma \vdash *a : t}$

2.2 Le bloc else optionnel dans la conditionnelle

Afin de traiter le bloc else optionnel, nous avons décidé d'ajouter une nouvelle instruction ConditionnelleOptionnelle of $expression * bloc$ dans l'AST et dans parser.mly, où expression correspond au booléen de la conditionnelle et bloc correspond aux instructions du if qui sont traitées uniquement si le booléen est à true. Nous avons également dû ajouter cette nouvelle instruction dans parser.mly.

Ensuite, nous ajoutons simplement cette instruction dans les 4 passes et nous la traitons.
Pour la passe TDS, la passe de Type ainsi que la passe de Placement, la conditionnelle optionnelle est traitée de la même manière que la conditionnelle classique, à la différence qu'il n'y a qu'un bloc à analyser. De la même manière, dans la passe de Code, on supprime simplement le label qui ramène en haut du bloc correspondant au else et on jump directement au label de fin de la conditionnelle dans le cas où le booléen est à false.

2.3 La conditionnelle sous la forme d'un opérateur ternaire

En ce qui concerne le cas de la conditionnelle via un opérateur ternaire, nous avons fait le choix d'ajouter une expression Ternaire of $expression * expression * expression$ dans l'AST et dans parser.mly. Chacune des expressions correspond respectivement à la condition qui est de type booléenne, la valeur de l'expression si la condition est vraie et celle-ci si la condition est fausse.
Nous avons aussi dû ajouter 2 caractères spéciaux à RAT dans lexer.mll : " ? " et " : ".

Dans la passe de TDS, il suffit d'analyser les 3 expressions initiales avec $analyse\_tds\_expression$ puis de renvoyer l'AstTds. Ternaire avec en paramètre les 3 nouvelles expressions.

Nous procédons de la même manière dans la passe de Type en s'assurant que la première expression est de type booléen et que les expressions 2 et 3 sont de même type. Dans le cas contraire, on lève une exception "TypeInattendu".

La passe de Placement ne change rien aux expressions donc nous n'avons rien dû modifier pour l'implantation de cette conditionnelle.

Finalement, le traitement dans la passe de Code est similaire à celui de la conditionnelle if, à la différence que l'on a 3 expressions et non pas 1 expression puis 2 blocs. On doit analyser les 3 expressions et créer 2 labels : Un localisé après l'analyse de l'expression correspondant à $condition = true$ pour sauter la 1re expression et un localisé après l'analyse de l'expression correspondant à $condition = false$ pour sauter la 2e expression.
Enfin, on ajoute un Jumpif qui renvoie à la l'expression correspondant à $condition = false$ dans le cas où l'analyse de la condition renvoie false et un jump vers la fin de la conditionnelle après l'analyse de l'expression correspondant à $condition = true$.

2.4 Les boucles ”loop” à la Rust

Une boucle loop à la Rust demande suite au mot-clé "loop" d'exécuter un bloc de code à l'infini ou jusqu'à ce que le programme soit arrêté manuellement, par un ctrl-c par exemple. Il faut donc fournir un autre moyen de sortir d'une boucle en utilisant du code.
Le mot-clé "break" à l'intérieur de la boucle demande au programme d'arrêter la boucle.
L'instruction "continue" est utilisée pour ignorer le reste de l'itération en cours et en débuter une nouvelle.

Il est également possible d'associer une étiquette de boucle à une boucle qu'il sera ensuite possible d'utiliser en association avec break ou un continue pour préciser que ce mot-clé s'applique sur la boucle correspondant à l'étiquette plutôt qu'à la boucle la plus proche possible.
Nous avons ainsi décidé d'ajouter 3 instructions dans l'AST et dans parser.mly :

- Loop of *bloc*
- Break of *string*
- Continue of *string*

Nous avons également dû ajouter dans lexer.mll les tokens suivants afin que le compilateur puisse reconnaître la boucle loop :

- "loop", LOOP
- "break", BREAK
- "continue", CONTINUE

Nous avons également fait le choix d'ajouter une nouvelle table "tdl" (table de loop) qui va contenir les infos sur toutes les boucles loop et un nouveau type d'info dans tds.ml afin de pouvoir référer à une boucle loop grâce à son identifiant quand celle-ci est nommée : InfoLoop of $string * string * string$

Ainsi, dans la passe de TDS, pour Loop on récupére l'InfoLoop puis on crée une Tdl fille et on ajoute la boucle Loop dans cette tdl. Enfin, il ne reste qu'à analyser le bloc de la boucle. Pour un break ou un continue, il suffit de rechercher l'identifiant de la boucle loop associée dans la tdl puis on renvoie l'info_ast associée.

Dans la passe de Type, on analyse le bloc de la boucle loop et on renvoie simplement le break et le continue tel quel. Il en est de même pour la passe de Placement où chacune des instructions prend 0 de place dans la tds.

Enfin, dans la passe de Code, pour le loop on crée d'abord une étiquette pour le début et la fin du loop, on rentre ces étiquettes dans l'InfoLoop et ensuite le traitement se fait de façon similaire à une boucle "TantQue" à la différence qu'on doit a pas besoin d'analyser une condition ou de faire un JumpIf puisque les "break" et "continue" sont inclus dans le bloc.
Pour le break, on doit simplement récupérer dans l'InfoLoop associée l'étiquette du "endLoop". De même pour le continue avec l'étiquette du "loop".

3 Tests

Pour chacune de ces extensions du langage RAT, nous avons ajouté des tests dans le dossier tests pour chaque passe dans la partie sans_fonction. Ces tests passent tous sans erreurs sur nos machines.

4 Conclusion

En conclusion, ce projet nous a permis de mieux comprendre le fonctionnement d'un compilateur. Il nous a aussi permis de mettre en pratique les connaissances acquises en cours de traduction des langages en implémentant les fonctionnalités essentielles d'un compilateur ainsi que des éléments plus avancés tels que la reconnaissance des pointeurs, les conditionnels avec option "else" facultative, les conditionnelle sous forme d'opérateur ternaire et une boucle loop de type "Rust".

Nous n'avons pas eu beaucoup de difficultés à ajouter les conditionnelles "else optionnel" et "ternaire" puisque l'implémentation ressemble beaucoup à la conditionnelle if/else que nous avions déjà implémentés. Nous avons eu un peu plus de difficultés à implémenter les pointeurs , bien que le début de l'implantation ait été abordé en TD, il nous a quand même fallu du temps pour tout traiter. Mais ce qui nous a pris le plus de temps et d'effort reste la boucle loop à la Rust puisqu'il fallait trouver comment créer un lien entre une boucle loop et le break ou continue qui sont situés dans le bloc. Il nous a fallu plusieurs essais avant de trouver la solution finalement retenue, à savoir l'ajout d'un type d'info InfoLoop et d'une table des loop (tdl).

Les résultats obtenus sont satisfaisant puisque les fonctionnalités ont été validées par des tests, bien que des améliorations peuvent encore être apportées. Notamment, on pourrait encore ajouter un certain nombre de fonctionnalités tel que la surchage des fonctions, les boucles while, les structure switch/case, les types énumérés, les tableaux...