Programmation fonctionnelle & Traduction des langages

Rapport de projet compilateur RAT/TAM

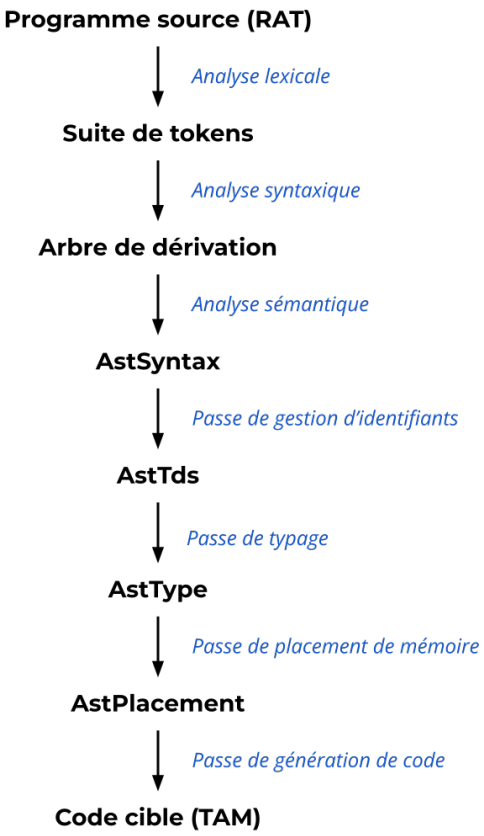
Martin Caissial Axel Metzinger

Deuxième Année - Département SN - Filière ASR
Année 2022-2023

Table des matières

1 Introduction

Dans le cadre de ce projet de *Programmation Fonctionnelle* et de *Traduction des Langages* nous avons modifié le compilateur créé en TP en y ajoutant de nouvelles règles de grammaire et en adaptant le code à celles-ci. Les règles ajoutées permettent l'implémentation des pointeurs, d'une conditionnelle avec bloc *else* optionnel, d'un opérateur ternaire et de boucles *loop* fonctionnant sur le modèle du langage Rust. Dans la suite de ce rapport, nous allons vous présenter les jugements de typage associés aux règles nouvellement ajoutées ainsi qu'une description de l'implémentation que nous avons réalisée.



2 Règles ajoutées

2.1 Les Pointeurs

2.1.1 Les jugements de typage

Les règles de grammaire ajoutées pour cette étape sont les suivantes :

- $I \rightarrow A \ = \ E$ *: une affectation, remplace* $I \rightarrow id = \ E$
- $A \rightarrow id$ *: un identifiant, remplace* $E \rightarrow id$
- $A \rightarrow (*A)$ *: un dé-référencement, accès en lecture ou en écriture à la valeur pointée par* A
- $E \rightarrow A$ *: un affectable*
- $E \rightarrow null$ *: un pointeur null*
- $E \rightarrow (new\ TYPE)$ *: une initialisation d'un espace mémoire pour un type donné*
- $E \rightarrow \&id$ *: un pointeur donnant accès à une variable*
- $TYPE \rightarrow TYPE *$ *: un pointeur sur un type donné*
- $TYPE \rightarrow (TYPE)$ *: un type parenthésé*

Voici dans l'ordre les jugements de typage associés à ces règles :

$$\frac{\sigma \vdash A : \tau \qquad \sigma \vdash E : \tau}{\sigma, \tau_r \vdash A \ = \ E : void, []} \quad (1)$$

$$\frac{}{\sigma \vdash id : \tau} \quad (2)$$

$$\frac{\sigma \vdash A : Pointeur(\tau)}{\sigma \vdash *A : \tau} \quad (3)$$

$$\frac{}{\sigma \vdash null : Pointeur(Undefined)} \quad (4)$$

$$\frac{\sigma \vdash TYPE : \tau}{\sigma, \tau_r \vdash (new\ TYPE) : Pointeur(\tau)} \quad (5)$$

$$\frac{\sigma \vdash id : \tau}{\sigma, \tau_r \vdash \&id : Pointeur(\tau)} \quad (6)$$

$$\frac{\sigma \vdash TYPE : \tau}{\sigma \vdash TYPE * : Pointeur(\tau)} \quad (7)$$

$$\frac{\sigma \vdash TYPE : \tau}{\sigma \vdash (TYPE) : \tau} \quad (8)$$

2.1.2 Description de l'implémentation

Pour l'implémentation des pointeurs, nous avons d'abord commencé par ajouter les affectables, qui remplacent les identifiants grâce à deux nouvelles règles de grammaire. Il fallait alors vérifier le bon fonctionnement du code avec l'ajout de ce nouveau type, et qu'aucun bug n'apparaisse sur cette étape pour que nous puissions implémenter la totalité des règles de grammaire des pointeurs.

Une fois cela fait et tous les tests passés sans aucune régression, nous avons pu continuer avec les règles restantes. Si jusqu'ici le code se comportait de la même manière, il a fallu améliorer notre travail précédent afin de réussir à intégrer chaque nouvelle règle au lexer, au parser et à l'AST. Ceci consistait en l'ajout de tokens au lexer ainsi qu'en l'ajout de toutes les règles de grammaire au parser. Pour l'AST, nous avons dû rajouter les différentes instructions ainsi que le déréférencement des pointeurs aux affectables. Cependant, nous avons dû changer la gestion des constantes que nous ne pouvions plus traiter comme précédemment. Nous avons alors décidé d'ajouter les constantes aux affectables, même si cette solution peut paraître illogique puisque les constantes ne peuvent pas être modifiées.

Avec l'ajout des pointeurs, la vérification de la compatibilité des types dans la passe de typage ne pouvait plus uniquement se faire au travers d'une simple égalité. Il a donc fallu utiliser la méthode *est_compatible* et la modifier pour qu'elle fonctionne bien avec des pointeurs. Pour cela, il fallait qu'une comparaison entre un pointeur de type *undefined* et un type quelconque soit acceptée, et qu'on puisse accéder au type d'une variable pointée. Il ne restait ainsi plus qu'à faire une attribution correcte des types pour chaque instruction relative aux pointeurs en respectant les jugements de typage.

Au sein de la passe de placement, nous n'avons fait aucune modification particulière. Les affectables, à l'instar des expressions, n'ont pas besoin d'un placement mémoire spécifique.

Pour finir, nous avons réalisé la passe de code qui nous a demandé plus de réflexion afin de faire fonctionner en concomitance les variables classiques avec les pointeurs. Il nous a ainsi fallu ajouter une distinction entre l'écriture et la lecture d'un affectable, qui modifie alors le code généré. Cela se traduit en un booléen, argument de la fonction d'analyse du code des affectables. Nous avons donc dû déplacer le traitement des affectations auparavant réalisé dans les instructions au niveau du traitement des affectables. Ainsi, nous avons pu traiter les pointeurs en exploitant le fonctionnement des adresses du langage TAM.

2.2 Le bloc "else" optionnel dans la conditionnelle

2.2.1 Les jugements de typage

Les règles de grammaire ajoutées pour cette étape sont les suivantes :

- $I \rightarrow if\ E\ BLOC$ *: une conditionnelle sans bloc else*

Voici dans l'ordre les jugements de typage associés à ces règles :

$$\frac{\sigma \vdash E_c : bool \qquad \sigma, \tau_r \vdash BLOC : void}{\sigma, \tau_r \vdash if\ E_c\ BLOC : void, []} \tag{9}$$

2.2.2 Description de l'implémentation

Pour cette étape, nous avons implémenté une reprise de l'instruction *if BLOC else BLOC* avec le bloc associé à *else* n'étant plus qu'une liste d'instruction vide dans le parser. Par conséquent, la suite du traitement reste exactement la même. Nous avons cependant apporté une optimisation dans la passe de génération du code, en y rendant optionnel la génération des étiquettes du second bloc si celui-ci est vide.

2.3 L'opérateur ternaire

2.3.1 Les jugements de typage

Les règles de grammaire ajoutées pour cette étape sont les suivantes :

- $E \rightarrow (E_c\ ?\ E_1 : E_2)$ *: un opérateur ternaire, si* E_c *est vraie, on retourne* E_1*, sinon on retourne* E_2

Voici dans l'ordre les jugements de typage associés à ces règles :

$$\frac{\sigma \vdash E_c : bool \qquad \sigma \vdash E_1 : \tau \qquad \sigma \vdash E_2 : \tau}{\sigma \vdash (E_c\ ?\ E_1 : E_2) : \tau} \tag{10}$$

2.3.2 Description de l'implémentation

Pour la mise en place de l'opérateur ternaire $(E_c\ ?\ E_1 : E_2)$, nous avons repris la logique utilisée pour la conditionnelle. La seule vraie différence réside dans le fait que la conditionnelle est une instruction, tandis que l'opérateur ternaire est une expression utilisée dans une instruction et doit donc avoir une valeur de retour. Ainsi, le code utilisé dans les deux cas est le même, cependant, nous analysons maintenant une expression au lieu d'un bloc. D'autre part, les types de retour des expressions E_1 et E_2 doivent être les mêmes pour le bon fonctionnement de l'opérateur.

2.4 Les boucles "loop" à la Rust

2.4.1 Les jugements de typage

Les règles de grammaire ajoutées pour cette étape sont les suivantes :

- $I \rightarrow loop\ BLOC$: *début d'une boucle*
- $I \rightarrow id\ :\ loop\ BLOC$: *début d'une boucle nommée*
- $I \rightarrow break$: *on sort de la boucle courante*
- $I \rightarrow break\ id$: *on sort de la boucle indiquée*
- $I \rightarrow continue$: *on recommence la boucle courante*
- $I \rightarrow continue\ id$: *on recommence la boucle indiquée*

Voici dans l'ordre les jugements de typage associés à ces règles :

$$\frac{\sigma, \tau_r \vdash BLOC : void}{\sigma, \tau_r \vdash loop\ BLOC : void, []} \tag{11}$$

$$\frac{(id, void) :: \sigma, \tau_r \vdash BLOC : void}{\sigma, \tau_r \vdash id : loop\ BLOC : void, []} \tag{12}$$

$$\frac{}{\sigma \vdash continue : void} \tag{13}$$

$$\frac{}{\sigma \vdash continue\ id : void} \tag{14}$$

$$\frac{}{\sigma \vdash break : void} \tag{15}$$

$$\frac{}{\sigma \vdash break\ id : void} \tag{16}$$

2.4.2 Description de l'implémentation

Pour l'implémentation des boucles **loop** à la manière du langage Rust, nous avons commencé par créer les nouveaux tokens *(loop, continue et break)* dans le lexer et le parser. Il a ensuite fallu travailler sur les différentes passes.

Pour la passe de gestion des identifiants, il nous a d'abord fallu rajouter un argument à la fonction d'analyse des blocs et à la fonction d'analyse des instructions contenant le nom des boucles ouvertes au niveau d'une instruction donnée. Cela représente en réalité une liste d'identifiants à laquelle on ajoute le nom d'une boucle lorsque l'on entre à l'intérieur de celle-ci. Les boucles se voient attribuer un identifiant s'il est précisé, sinon on leur donne un nom générique afin de les différencier du reste. Cela nous permet de détecter la création de boucles imbriquées de mêmes noms et d'envoyer un avertissement dans ce cas. Nous avons aussi traité le cas de deux boucles sans identifiant, qui techniquement auront le même nom dans notre code mais qui n'entraîneront donc pas d'avertissement.

Pour la passe de gestion des types et la passe de placement mémoire, nous n'avons pas rajouté de traitement spécifique aux boucles.

Finalement, lors de la passe de code, nous avons à nouveau utilisé une liste, afin de traiter les étiquettes d'une manière particulière en raison des *break id* et *continue id*. On génère une étiquette de fin unique qu'on ajoute à la liste dans un couple avec l'identifiant de la boucle pour la durée de son bloc. L'étiquette de début est alors l'identifiant de la fonction concaténé à l'étiquette de fin. Cela permet d'avoir une étiquette de début unique, qui ne créé pas de confusion lors d'un appel à *break id* ou à *continue id*. Ainsi, ces instructions fonctionnent comme attendu avec les boucles, qu'elles soient nommées ou non.

3 Conclusion

En conclusion, nous avons réussi à implémenter dans le compilateur les fonctionnalités supplémentaires demandées. Par ailleurs, nous avons ajouté au compilateur la vérification de l'existence d'une instruction *return* au sein de chaque fonction au niveau de la passe de gestion des identifiants. Nous émettons un *warning* à l'utilisateur lorsque du code mort est détecté, cela correspondant à du code inatteignable à cause d'un *return*.

Nous avons cependant rencontré quelques difficultés lors des différentes étapes, à commencer par la mise en place des pointeurs. En effet, lors de la passe de génération de code, nous avons eu une période de débuggage assez longue constituée de plusieurs itérations de tests et de changement au niveau du code avant d'obtenir quelque chose de fonctionnel. Nous nous sommes majoritairement heurtés à des problèmes avec les pointeurs sur des types de taille supérieur à deux en mémoire, ainsi qu'avec les pointeurs de pointeurs qui ne fonctionnaient pas comme attendu. Ces problèmes ont pu être réglés en modifiant notre implémentation au niveau de la passe de code, qui ne prenait pas en compte certains cas, causant des erreurs lorsque nous avons rajouté des tests.

Nous avons aussi rencontré des problèmes lors de la gestion des identifiants pour les boucles. Pour obtenir un fonctionnement similaire au langage Rust, il fallait trouver une manière d'enregistrer les boucles de façon à pouvoir facilement y accéder au besoin, et de les supprimer tout aussi aisément à la sortie du bloc. Nous avons d'abord pensé à recréer des tables des symboles pendant la passe de gestion d'identifiants, mais finalement nous nous sommes tournés vers l'ajout de listes. Cela nous a semblé plus facile à implémenter et plus dynamique notamment vis à vis de l'ajout et la suppression d'identifiants au fur et à mesure.

En résumé, on peut dire que malgré nos quelques difficultés, ce projet aura été pour nous une réussite aussi bien d'un point de vue humain que de la mise en oeuvre des compétences apprises en cours.