Projet Traduction des langages et Programmation fonctionelle

WU Christophe - JEANDON Emile

Département Sciences du Numérique - Parcours Architecture Système Réseaux
2022-2023

Table des matières

1 Introduction **3**

2 Fonctionnalités ajoutées **4**
2.1 Les Pointeurs 4
2.2 Le bloc else optionnel 5
2.3 La conditionnelle sous la forme d'un opérateur ternaire 5
2.4 Les boucles "loop" à la Rust 5

3 Conclusion **7**

Table des figures

1 Grammaire du langage RAT étendu 3

1 Introduction

Le but de ce projet de programmation fonctionnelle et de traduction des langages est d'étendre le compilateur du langage RAT réalisé en TP de traduction des langages pour traiter de nouvelles fonctionnalités : les pointeurs, le bloc else optionnel dans la conditionnelle, la conditionnelle sous la forme d'un opérateur ternaire, les boucles "loop" à la Rust. Les détails d'implémentation des différentes fonctionnalités et les modifications de l'Ast réalisées pour permettre leur implémentation sont explicités dans la section suivante. Des tests additionnels ont également été ajoutés dans les sources pour tester les différentes fonctionnalités implémentées. La grammaire attendue pour le langage RAT étendu est décrite ci-dessous :

1. $MAIN \rightarrow PROG$
2. $PROG \rightarrow FUN\ PROG$
3. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ BLOC$
4. $PROG \rightarrow id\ \ BLOC$
5. $BLOC \rightarrow \{\ IS\ \}$
6. $IS \rightarrow I\ \ IS$
7. $IS \rightarrow$
8. $I \rightarrow TYPE\ \ id\ =\ E\ ;$
9. $I \rightarrow A\ =\ E\ ;$
10. $I \rightarrow const\ \ id\ =\ entier\ ;$
11. $I \rightarrow print\ \ E;$
12. $I \rightarrow if\ \ E\ \ BLOC\ \ else\ \ BLOC$
13. $I \rightarrow if\ \ E\ \ BLOC$
14. $I \rightarrow while\ \ E\ \ BLOC$
15. $I \rightarrow return\ E;$
16. $I \rightarrow loop\ BLOC$
17. $I \rightarrow id\ :\ loop\ BLOC$
18. $I \rightarrow break;$
19. $I \rightarrow break\ id;$
20. $A \rightarrow id$
21. $A \rightarrow (*\ A)$
22. $DP \rightarrow$
23. $DP \rightarrow TYPE\ id\ DP$
24. $TYPE \rightarrow\ bool$
25. $TYPE \rightarrow\ int$
26. $TYPE \rightarrow\ rat$
27. $TYPE \rightarrow\ TYPE\ *$
28. $E \rightarrow call\ id\ (\ CP\ )$
29. $CP \rightarrow$
30. $CP \rightarrow E\ CP$
31. $E \rightarrow\ [\ E\ /\ E\ ]$
32. $E \rightarrow\ num\ \ E$
33. $E \rightarrow\ denom\ \ E$
34. ~~$E \rightarrow id$~~
35. $E \rightarrow\ true$
36. $E \rightarrow\ false$
37. $E \rightarrow\ entier$
38. $E \rightarrow\ (\ E\ +\ \ E\ )$
39. $E \rightarrow\ (\ E\ *\ \ E\ )$
40. $E \rightarrow\ (\ E\ =\ \ E\ )$
41. $E \rightarrow\ (\ E\ <\ \ E\ )$
42. $E \rightarrow\ (\ E\ ?\ E\ :\ E\ )$
43. $E \rightarrow A$
44. $E \rightarrow null$
45. $E \rightarrow (new\ TYPE)$
46. $E \rightarrow \&\ id$

FIGURE 1 – Grammaire du langage RAT étendu

2 Fonctionnalités ajoutées

2.1 Les Pointeurs

Jugement de Typage

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

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

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

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

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

Evolution de l'Ast

Conformément à la grammaire présentée plus tôt 1, un nouveau type **affectable** a été défini, il représente les affectables du langage et est défini par **Ident of String** (avant passage par la passe Tds, **Ident of Tds.info_ast** par la suite) qui reprend le comportement des identifiants déjà établi auparavant et **Deref of affectable** pour les déréférencements.

De nouvelles expressions ont été definies, on y retrouve **Adresse of String** (avant passage par la passe Tds, **Adresse of Tds.info_ast** par la suite) qui représente l'adresse d'une variable, **Null** qui représente le pointeur null et **New of typ** qui permet d'initialiser un nouveau pointeur de type **typ**.

Enfin, un nouveau type a été ajouté à la liste des type ainsi le type **typ** comprend désormais aussi **Pointeur of typ** qui représente les pointeurs. L'avantage de le définir de manière récursive de cette manière permet de gérer les enchaînements de pointeurs.

Implémentation

Le token **&**, les terminaux **new** et **null** ont été ajouté au lexer, par la suite les tokens associés ont été déclarés dans le parser.

Une fonction qui permet d'analyser de manière récursive les affectables a été ajoutée à chaque passe. En ce qui concerne la passe de typage, il est possible d'effectuer une affectation ou une déclaration entre un pointeur null (de type **Pointeur of Undefined**) et tout autre type de pointeur. Les fonctions **est_compatible** et **getTaille** ont été adaptées pour répondre à la nouvelle demande.

Une attention a été portée sur le fait qu'on ne puisse pas réaliser d'opérations réservées aux pointeurs sur des non pointeurs (l'exceptions **PasUnPointeur of string** a été ajoutée) et sur le fait qu'on ne puisse utiliser le token "&" uniquement devant une variable.

En ce qui concerne la génération de code la gestion du déréférencement a été réalisée en remontant de manière récursive la chaîne de déréférencement en effectuant des **LOADI** jusqu'au point souhaité et effectuer un **STOREI** ou un **LOADI** selon les circonstances.

2.2 Le bloc else optionnel

Jugement de Typage

$$\frac{\sigma \vdash E1 : bool \quad \sigma \vdash E2 : \tau}{\sigma \ \ (if\ E1\ E2) : \tau}$$

Evolution de l'Ast

Le traitement du bloc else optionnel n'a pas engendré de modification de l'Ast, sa gestion a directement été faite dans le type **Conditionelle of expression * bloc * bloc** en y fournissant une liste vide en tant que bloc else.

Implémentation

L'implémentation du bloc else optionnel a été réalisé en effectuant une reconnaissance sur le contenu du bloc else, si ce dernier est vide on reconnaît le cas où on n'a pas besoin de bloc else, l'analyse se poursuit comme si il n'existait pas. Dans le cas contraire, on reprend le comportement déjà implémenté.

2.3 La conditionnelle sous la forme d'un opérateur ternaire

Jugement de Typage

$$\frac{\sigma \vdash E1 : bool \quad \sigma \vdash E2 : \tau \quad \sigma \vdash E3 : \tau}{\sigma \vdash (E1\ ?\ E2 : E3) : \tau}$$

Evolution de l'Ast

Le traitement de la conditionnelle sous la forme d'un opérateur ternaire a nécessité l'ajout d'un nouveau type **Ternaire of expression * expression * expression** qui est construit à partir de l'expression formant la condition, l'expression de la valeur si la condition est vérifiée et l'expression de la valeur dans le cas contraire.

Implémentation

Les token ? et : ont été déclarés dans le lexer et le parser.

L'ajout de l'opérateur ternaire n'a pas nécessité de points particuliers, il est néanmoins important de vérifier au niveau de la passe de typage que la première expression est bien de type **Bool** et que les deux autres expressions sont de même type. (les exceptions **TypeCondTernaireInattendu of typ * typ** et **TypeValTernaireInattendu of typ * typ** ont été ajoutées)

La génération de code s'apparente à celle d'une conditionnelle avec un bloc else.

2.4 Les boucles "loop" à la Rust

Jugement de Typage

$$\frac{\sigma,\tau_(r) \vdash BLOC : void}{\sigma,\tau_(r) \vdash id : loop\ BLOC : void,[]}$$

Evolution de l'Ast

La gestion des boucles "loop" à la Rust au niveau de l'Ast a nécessité la création de 3 nouveaux types :

— **Loop of string option * bloc** (avant passage par la passe Tds, **Loop of Tds.info_ast * bloc** par la suite) construit à partir d'un string si la boucle est nommée, de **None** sinon (l'info de la boucle après passage par la passe Tds) et du bloc contenant la liste des instructions de la boucle.
— **Break of string option** (avant passage par la passe Tds, **Break of Tds.info_ast** par la suite) construit à partir d'un string si le break a été appelé avec un identifiant, **None** sinon (l'info de la boucle la plus imbriquée dans le cas d'un break non nommé et l'info de la boucle à interrompre dans le cas d'un break avec un identifiant, après passage par la passe Tds).
— **Continue of string option** (avant passage par la passe Tds, **Continue of Tds.info_ast** par la suite) construit à partir d'un string si le continue a été appelé avec un identifiant, **None** sinon (l'info de la boucle la plus imbriquée dans le cas d'un continue non nommé et l'info de la boucle à remonter dans le cas d'un continue avec un identifiant, après passage par la passe Tds).

Implémentation

Les terminaux **loop**, **break** et **continue** ont été déclarés dans le lexer, de même pour les tokens associés dans le parser.

Le choix d'utiliser des constructeurs unique pour les éléments nommés et non nommés implique de réaliser une séparation de ces cas dans chaque passe au moment d'analyser l'instruction **Loop**, **Break** ou **Continue**. L'implémentation a principalement nécessité des actions dans la passe de génération de code et dans la passe Tds.

Pour ajouter les boucles à la tds, le type **InfoLoop of string * string * string * string list** a été rajouté, ce dernier est construit à partir de l'identifiant de la boucle ("" dans le cas d'une boucle non nommée), son étiquette de début, son étiquette de fin et la liste des identifiants des boucles de niveau supérieur (l'identifiant de la boucle courante compris). Dans les deux cas, on ajoute l'**InfoLoop** à la tds courante même si une boucle de même identifiant a déjà été déclarée dans la tds courante pour ainsi permettre les boucles de même identifiant au même niveau.

Afin de permettre à une variable/constante d'avoir le même identifiant qu'une boucle et à une fonction d'avoir le même identifiant qu'une boucle (on rappelle que les deux cas ne peuvent pas avoir lieu simultanément), les fonctions **chercherLocalement** et **chercherGlobalement** ont été modifiées pour prendre en paramètre un booléen qui permet d'indiquer si on chercher une boucle ou non dans la tds. En fonction de la valeur de ce booléen, on filtre la sortie en fonction de la demande.

Le traitement des opérations sur des boucles imbriquées a été fait de la manière suivante :
— dans le cas d'un break non nommé, on lui associe l'**InfoLoop** de la boucle la plus imbriquée
— dans le cas d'un break nommé, on vérifie si l'identifiant associé est dans la liste des identifiants de boucles de niveau supérieur (la boucle à partir de laquelle a été appelée le break comprise) si c'est le cas, on lui associe l'**InfoLoop** de la boucle en question.

Au niveau de la génération de code, lors de l'analyse d'une boucle, la fonction **ajouter_etiquette** permet d'ajouter les étiquettes de début et de fin à l'**InfoLoop** de la boucle concernée. Pour que l'instruction **Break** puisse la récupérer.

Les implémentations de l'instruction **Break** et de l'instruction **Continue** ont été traitées de manière similaire (l'unique différence étant qu'on renvoie à l'étiquette de fin dans le cas d'un **break** et à l'étiquette de début dans le cas d'un **continue** au niveau de la génération de code).

Les exceptions **BreakMalPlace of string**, **ContinueMalPlace of string**, **BreakNonNommeMalPlace**, **ContinueNonNommeMalPlace** et **BoucleInconnue of string** ont été ajoutées pour répondre aux différents besoins. Le choix de différencier les cas nommés et non nommés a été fait pour

avoir une meilleur traçabilité dans le cas des **break**/**continue** nommés.

3 Conclusion

Au cours de ce projet, de nombreux cas ont été testés pour compléter le code du compilateur. Trouver des tests qui révèlent des failles dans le compilateur a été une tâche essentielle au bon déroulement de ce projet. Les principales difficultés rencontrées au cours de ce projet ont été la gestion de la surchage variable/boucle, la gestion complète du pointeur null et la gestion des warning de Menhir qui ne donne pas de traçabilité.

Le compilateur peut être améliorer de différente sorte, en gérant l'affichage d'un warning lorsque deux boucle de même nom ont été déclaré au même niveau, en offrant une gestion plus complète du pointeur null, en gérant la surchage au niveau des fonctions (génération de code à modifié car plusieurs labels identiques ne sont pas permis dans le code généré) ou en permettant à une boucle de s'appeler "loop" (impossible dans l'état actuel car refusé au moment de l'exécution) par exemple.