Projet de programmation fonctonnelle et de traduction des langages

BECKER Alaric - DEGAIL Dylan

Département Sciences du Numérique - Deuxième année
Année 2022/2023

Table des matières

1 Introduction **3**

2 Types **3**

3 Jugement de typage **3**
3.1 Pointeurs 3
3.2 Bloc else optionnel 4
3.3 Conditionnelle sous la forme d'un opérateur ternaire 4
3.4 Boucle "loop" à la Rust 4
3.4.1 Boucle 4
3.4.2 Break 4
3.4.3 Continue 4

4 Pointeurs **5**
4.1 Grammaire 5
4.1.1 Règles ajoutées 5
4.1.2 Règles modifiées 5
4.1.3 Règles supprimées 5
4.2 Structure de l'AST 5
4.3 Construction 6

5 Bloc else optionnel **6**

6 Conditionnelle ternaire **6**

7 Loop à la Rust **7**
7.1 Problème 8

8 Conclusion **9**

1 Introduction

Ce projet a pour but de mettre en pratique l'ensemble des connaissances acquissent dans le cours de programmation fonctionnelle et traduction des langages. De ce faite, le but du projet était d'étendre un compilateur du **langage RAT** en utilisant **OCaml** et le début du travail a été effectué en TP en traduction des langages.

Nous allons donc aborder l'ensemble des nouvelles fonctionnalités ajoutées durant ce projet :

— Les **pointeurs**
— Le **bloc else optionnel**
— La **conditionnelle sous la forme d'un opérateur ternaire**
— Les **boucles "loop" à la Rust**

Pour chacune des fonctionnalités, nous allons expliquer leur mise en place au sein du compilateur à chaque niveau d'analyse. De plus, on precisera, pour chaque types, l'évolution de la structure de l'AST et les jugements de typage liés aux nouvelles constructions du langage.

2 Types

Dans cette partie, nous allons donner des explications sur l'évolution de l'AST au cours des passes de l'**analyse sémantique**.

Tout d'abord, lors de la **passe de gestion des identifiants**, on stocke les informations des identifiants dans une TDS qui sera aussi dans l'AST. On aura donc plus de noms dans l'AST et cela permet aussi de ne plus avoir besoin d'une instruction $Constante\ of\ string * int$.

Pour la **passe de typage**, on enlève les types dans l'AST pour les ajouter, après vérifications, dans la TDS de l'identifiant concerné. La surchage des types intervient à ce moment là.

Pour la passe de **placement mémoire**, on rajoute encore une fois une nouvelle information à notre TDS qui est la taille dans la mémoire et le registre pour chaque identifiant. Au niveau de l'AST, on propage la taille que prennent les blocs mais aussi la taille du retour et des paramètres d'une fonction.

Après ceci, l'AST n'évolue plus.

3 Jugement de typage

3.1 Pointeurs

$$-\frac{\sigma \vdash A : \tau \quad \sigma \vdash E : \tau}{\sigma, \tau_r \vdash A = E : void, []}$$

$$-\frac{\sigma \vdash A : Pointeur(\tau)}{\sigma, \tau_r \vdash (*A) : \tau, []}$$

$$-\frac{}{\sigma \vdash null : Pointeur(Undefined), []}$$

$$-\frac{\sigma \vdash TYPE : \tau}{\sigma \vdash (new\ TYPE) : Pointeur(\tau), []}$$

$$-\frac{\sigma \vdash id : \tau}{\sigma \vdash (\&\ id) : Pointeur(\tau), []}$$

3.2 Bloc else optionnel

$$-\frac{\sigma \vdash E : bool \quad \sigma, \tau_r \vdash BLOC : void}{\sigma, \tau_r \vdash if\ E\ BLOC : void, []}$$

3.3 Conditionnelle sous la forme d'un opérateur ternaire

$$-\frac{\sigma \vdash E_1 : bool \quad \sigma \vdash E_2 : \tau}{\sigma \vdash E_1\ ?\ E_2 : E_2 : void, []}$$

3.4 Boucle "loop" à la Rust

3.4.1 Boucle

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

$$-\frac{\sigma \vdash id : void \quad \sigma, \tau_r \vdash BLOC : void}{\sigma, \tau_r \vdash id : loop\ BLOC : void, [id]}$$

3.4.2 Break

$$-\frac{}{\sigma, \tau_r \vdash break : void, []}$$

$$-\frac{\sigma \vdash id : void}{\sigma, \tau_r \vdash break\ id : void, [id]}$$

3.4.3 Continue

$$-\frac{}{\sigma, \tau_r \vdash continue : void, []}$$

$$-\frac{\sigma \vdash id : void}{\sigma, \tau_r \vdash continue\ id : void, [id]}$$

4 Pointeurs

Cette fonctionnalité a été assez longue à traiter. Beaucoup de nouveautés dans l'ensemble des analyses (lexical, syntaxique et sémantique).

Concernant le *lexer.ml*, trois tokens **null**, **new** et **&** à ajouter pour son bon fonctionnement.

Au niveau de la **grammaire**, l'ajout d'une nouvelle structure $Affectable$ et de cinq règles pour prendre en compte cette syntaxe dans notre compilateur (*parser.ml*). On a aussi du modifier certaines règles :

4.1 Grammaire

4.1.1 Règles ajoutées

$A \rightarrow (* A)$
$TYPE \rightarrow TYPE *$
$TYPE \rightarrow (TYPE)$: Permet de gérer les pointeurs de pointeurs
$E \rightarrow null$
$E \rightarrow (new\ TYPE)$
$E \rightarrow \&\ id$
$I \rightarrow continue\ id;$

4.1.2 Règles modifiées

$I \rightarrow A\ =\ E\ ;$

4.1.3 Règles supprimées

$E \rightarrow id$

4.2 Structure de l'AST

Pour ce qui est de la structure de l'AST, on a rajouté :
Nouveau type affectable contenant :
$Ident\ of\ string$: Identifiant dans l'AST.
$Dref\ of\ affectable$: Déréférencement d'un Pointeur de type t.

$Affectable\ of\ affectable$: Le nouveau type affectable.
$Null$: Pointeur null (Undefined).
$New\ of\ typ$: Allouer un nouveau Pointeur de type t.
$Adress\ of\ string$: Adresse d'un Pointeur.

$Affectation\ of\ affectable\ *\ expression$: On passe maintenant par un affectable pour l'affectation.

4.3 Construction

Sur la construction de cette fonctionnalité lors de l'**analyse sémantique**, plusieurs choses à traiter :

— La première chose à prendre en compte était de différencier si l'affectable était à gauche de l'affectation (en mode écriture) ou à droite (en mode lecture). Pour respecter ça, on utilise simplement un *bool* pour préciser si on est en mode **écriture** ou pas.

— La deuxième chose est au niveau du type **Pointeur**. On a rajoute de nouveaux tests avec ce type et gérer la compatibilité entre les types.

5 Bloc else optionnel

Cette fonctionnalité ressemble sur plusieurs aspects à la structure du conditionnelle déjà implanté dans le compilateur. Elle est donc plûtot facile à traiter.

Au niveau de la **grammaire**, une petite modification pour prendre en compte cette syntaxe dans notre compilateur pour être accepté par le *parser.ml* :
$I \rightarrow if\ E\ BLOC$

Concernant le *lexer.ml*, aucune modifications à effectuer.

Pour ce qui est de la structure de l'AST, on a rajouté un $ElseOptionnel of expression * bloc$ pour différencier avec le *Conditionnel*

Sur la construction de cette fonctionnalité lors de **l'analyse sémantique**, très peu de changements par rapport au conditionnelle. La seule différence concerne la suppression de l'analyse du second $BLOC$.

Une autre conception possible serait de ne pas modifier la structure l'AST mais d'utiliser le *Conditionnel* en donnant un $BLOC$ else vide. Un point négatif de cette conception est la génération de bouts de codes morts ($JUMP$ et labels inutiles).

6 Conditionnelle ternaire

Cette fonctionnalité est, au final, une *Conditionnelle* sous une syntaxe et forme différente. Elle est donc, comme la fonctionnalité précédente, facile à traiter.

Concernant le *lexer.ml*, deux tokens **?** et **:** à ajouter pour son bon fonctionnement.

Au niveau de la **grammaire**, une petite modification pour prendre en compte cette syntaxe dans notre compilateur pour être accepté par le *parser.ml* :
$E \rightarrow (\ E\ ?\ E : E\ )$

Pour ce qui est de la structure de l'AST, on a rajouté un $Ternaire\ of\ bool * expression * expression$.

Sur la construction de cette fonctionnalité lors de **l'analyse sémantique**, plusieurs choses à traiter. Premierement, nous n'avons pas de $BLOC$ comme dans une *Conditionnelle*. Cette opération ternaire est en effet une *expression* et non une *instruction*. On a donc seulement à analyser les trois *expression* de l'opération. On doit vérifier, lors de la **passe de typage**, que la première *expression* qui est la condition doit être de type *bool*. Au niveau des autres passes, très peu de

différences avec une *Conditionnelle*.

Une autre conception possible serait de, au vu des différentes formes que peut pendre une *Conditionnelle*, généraliser son traitement au niveau de **l'analyse sémantique**. On aurait donc à chaque fois **3 attributs** : une **condition** (pour traiter l'un ou l'autre) et **deux attributs généralisés** (un *bloc*, une *expression*, etc.).

7 Loop à la Rust

Cette fonctionnalité a été la plus longue et complexe de notre point de vue. En effet, beaucoup de contraintes devaient être respecter comme par exemple avoir deux boucles de même nom au même niveau ou avoir une boucle et une variable peut avoir le même nom. On n'a pas réussi à finir son traitement.

Concernant le *lexer.ml*, trois tokens **loop**, **break** et **continue** à ajouter pour son bon fonctionnement.

Au niveau de la **grammaire**, l'ajout de six règles pour prendre en compte cette syntaxe dans notre compilateur pour être accepté par le *parser.ml* :
$I \rightarrow loop\ BLOC$
$I \rightarrow id\ :\ loop\ BLOC$
$I \rightarrow break;$
$I \rightarrow break\ id;$
$I \rightarrow continue;$
$I \rightarrow continue\ id;$

Pour ce qui est de la structure de l'AST, on a rajouté :
$BoucleInfinie\ of\ bloc$
$BoucleInfinieNommee\ of\ string * bloc$
Break : On le supprime après la passe de gestion des identifiants.
$BreakNommee\ of\ string$
Continue : On le supprime après la passe de gestion des identifiants.
$ContinueNommee\ of\ string$

Sur la construction de cette fonctionnalité lors de l'**analyse sémantique**, plusieurs choses à traiter :

— La première difficulté concerne la contrainte d'avoir des boucles de même nom que des variables ou fonctions. Pour cela, lors de la **passe de gestion des identifiants**, nous avons mis en place un $InfoBoucle\ of\ string$ mais surtout, on n'a pas ajouté les $InfoBoucle$ dans la TDS pour respecter cette contrainte.

— La deuxième difficulté est de lever un warning quand deux boucles imbriquées ont le même nom. Pour gérer ceci, on utilise une liste de $iast\ InfoBoucle$ qu'on propage dans l'analyse du bloc de la boucle et réciproquement dans l'analyse de l'instruction pour savoir si on a des boucles imbriquées de même nom.

— La troisième difficulté (celle dont on n'a pas totalement réussi) a été de gérer les *break* et *continue* lors de la **passe de génération de code**. Pour cela, on utilise la même idée qui est de propager, dans ce cas là, une liste de labels de débuts et une liste de labels de fin dans l'analyse du bloc. Comme ça, ils peuvent récupérer soit le label de fin pour le *break*, soit le label de début pour le *continue*.

7.1 Problème

Avec notre conception, on a donc un problème. En effet, on gère bien le fait d'avoir des boucles de même nom (imbriquées ou pas) lors de la **passe de gestion des identifiants**. Cela engendre un problème lors de la **passe de génération de code** avec *break* et *continue*. Comme on génère une étiquette pour les différencier, on tombe sur quelque chose d'aléatoire et donc impossible à récupérer correctement. On avait une idée avec la méthode $String.ends_with$ mais cette méthode n'est disponible qu'à partir de la version 4.13 de OCaml. On aurait pu trouver le bon label en cherchant dans la liste des labels, un label finissant avec le nom du break.

On a donc réfléchit à une autre solution. La solution serait d'utiliser une autre structure de données pour stocker spécifiquement les identifiants des bloucles (par exemple une pile). Les avantages sont donc multiples : pas de problemes de noms entre les variables / fonctions et les boucles, une gestion (surement) plus simple de boucles imbriquées grâce au concept de FIFO de la pile.

8 Conclusion

Ce projet a été un bon moyen pour comprendre de manière générale le fonctionnement d'un compilateur. Même si on a été limité à l'analyse lexical, syntaxique et sémantique, on a appris à les comprendre, à les modifier et les utiliser pour traduire notre code RAT en TAM.

Ça a été compliqué pour traiter les fonctionnalités comme les pointeurs ou les blocs loop à la Rust. On a perdu pas mal de temps avant de s'aider de la pile d'exécution. C'était clairement un gain de temps à ne pas négliger lors des sessions de débogages.

Au niveau des améliorations, la factorisation du code serait la bienvenue quand plusieurs fonctionnalités se ressemblent.