Projet de programmation fonctionnelle et de traduction des langages

Thierry Xu
Tom Bonetto

Département Sciences du Numérique - Deuxième année
2022-2023

Contents

1 Introduction

Le but du 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 constructions : les pointeurs, le bloc else optionnel dans la conditionnelle, la conditionnelle sous la forme d'un opérateur ternaire, les boucles "loop" à la Rust. Ce rapport résume les modifications apportées au compilateur (lexer, parser, AST, différentes passes) pour traiter ses nouvelles constructions.

2 Pointeurs

RAT étendu permet de manipuler les pointeurs à l'aide d'une notation proche de celle de C.

2.1 Lexer et Parser

Pour cela, nous avons rajouté dans le lexer les terminaux (**&**, **new** et **null**).
Dans le parser nous avons rajouté le token associé aux nouveaux terminaux et les règles de production est ajoutée à la grammaire :

- A → (* A) : déréférencement : accès en lecture ou écriture) la valeur pointée par A;
- TYPE → TYPE * : type des pointeurs sur un type TYPE;
- E → null;
- E → (new TYPE) : initialisation d'un pointeur de type TYPE;
- & id : accès à l'adresse d'une variable

2.2 AST

Au niveau de l'AST, nous avons ajouté le type affectable qui peut être soit un **Ident of string** soit un **Deref of affectable**, selon s'il s'agit d'un déréférencement. De plus, l'instruction d'affectation est maintenant de type **affectable * expression**. Pour respecter les nouvelles grammaires ajoutées, il faut également considérer les expressions **Null**, **New of typ** et **Adresse of string**.

2.3 Passes

- Passe Tds : Au niveau de la passe de gestion des identifiants, il faut prendre en compte les changements effectués au niveau de l'AST et notamment changer les string en Tds.infoast. Au niveau du code, il faudra prendre en compte les modifications apportées à l'AST. De plus, une fonction *analyse_ tds_ affectable* a été défini pour analyser les affectables
- Passe Type :

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

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

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

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

- Passe Placement : Pour la passe de placement, on considère qu'un pointeur occupe une place de taille 1 en mémoire, indépendamment du type pointé.
- Passe Code : Pour la passe de génération de code, une fonction *analyse_tds_affectable* a été ajouté. De plus, le code a été adapté afin de considérer les nouvelles expressions ajoutées.

3 Bloc else optionnel

Nous souhaitons ne pas être obligé de fournir un bloc else à la conditionnelle.

3.1 Lexer et Parser

Une règle de production est ajoutée à la grammaire :

- I → if E BLOC

3.2 AST

Nous avons rajouté à l'AST un nouveau type d'expression semblable à celui de la conditionnelle: Conditionnelle_opt of expression * bloc, qui restera inchangé après les différentes phases.

3.3 Passes

- Passe Tds : A l'image de la conditionnelle traditionnelle, on analyse simplement l'expression et le bloc.
- Passe Type : Il faut vérifier que le type de l'expression est bien un booléen.

$$\frac{\sigma \vdash E_1 : bool \quad \sigma \vdash E_2 : \tau}{\sigma \vdash (if E_1\ E_2) : \tau}$$

- Passe Placement : Contrairement à la conditionnelle standard, on analyse simplement le bloc.
- Passe Code : Pratiquement identique à la conditionnelle standard, sans le traitement du deuxième bloc.

4 Conditionnelle ternaire

Nous souhaitons, comme en C, permettre d'écrire la conditionnelle sous la forme d'un opérateur ternaire : condition ? valeur si vrai : valeur si faux ;

4.1 Lexer et Parser

Nous avons rajouté dans le lexer les terminaux (**?** et **:**).
Dans le parser nous avons rajouté les tokens associés aux nouveaux terminaux et une règle de production est ajoutée à la grammaire :

- E → (E ? E : E)

4.2 AST

Nous avons rajouté à l'AST un nouveau type d'expression : Ternaire of expression * expression * expression, qui restera inchangé après les différentes phases.

4.3 Passes

- Passe Tds : rien de particulier, on analyse simplement les trois expressions.
- Passe Type : Il faut vérifier que le type de la 1ère expression est un booléen et que le type de la 2ème et 3ème expressions sont les mêmes.

$$\frac{\sigma \vdash E_1 : bool \quad \sigma \vdash E_2 : \tau \quad \sigma \vdash E_3 : \tau}{\sigma \vdash (E_1 \ ? \ E_2 : E_3) : \tau}$$

- Passe Placement : Inchangée
- Passe Code : Similaire, voire quasi identique à la Conditionnelle de base (fait en TP)

5 Boucle "loop" à la Rust

Nous souhaitons rajouter au langage RAT les boucles "loop" à la Rust

5.1 Lexer et Parser

Nous avons rajouté dans le lexer les terminaux (**loop**, **break** et **continue**).
Dans le parser nous avons rajouté les tokens associés aux nouveaux terminaux et 6 règles de production ont sont ajoutées à la grammaire :

- I → loop BLOC
- I → id : loop BLOC
- I → break;
- I → break id;
- I → continue;
- I → continue id;

5.2 AST

Dans l'AST, 6 types d'instructions sont rajoutés : Boucle of bloc, BoucleID of string * bloc, Break, BreakID of string, Continue et ContinueID of string. Après la passe de gestion des identifiants.

5.3 Passes

- Passe Tds : Le nom des boucles est remplacé par leur info_ast, les noms des break et des continue sont remplacés par les info_ des boucles auxquelles ils sont associés. De même on rajoute une info_ast pour les boucles, break et continue qui n'ont pas de noms et n'avaient donc pas d'argument sur l'Ast syntax.
  Pour ce qui est de l'analyse des instructions, les traitements des continue sont identique à ceux des break.

  1. Boucle : Une InfoBoucle est créée, elle prend 4 arguments : une string qui correspond au nom de la boucle ("" si c'est une boucle sans identifiant), deux string qui accueilleront plus tard les étiquettes pour la génération de code et enfin une info option. L'info option correspond à l'information de la fonction dans laquelle la boucle se trouve, cet argument permet avec une fonction auxiliaire de remonter dans les boucles imbriquées pour associés les break/continue qui ont un identifiant.

2. : Break/Continue : Une InfoBreak et une InfoContinue sont créées, avec comme un argument un string pour stocker le nom. Cela permet de lever des exceptions quand des break et des continue de même nom sont déclarés localement.

- Passe type : on fait simplement une analyse type bloc pour les blocs d'instructions des boucles (avec identifiant et sans).

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

- Passe placement : on fait simplement une analyse placement bloc pour les blocs d'instructions des boucles (avec identifiant et sans).
- Passe Code : les boucles avec id ou sans sont traitées de la même manière, de même pour les break et les continue.

1. Boucle : Plutôt similaire aux boucles while. La particularité est qu'une fois avoir créer les étiquettes, on les ajoute dans l'InfoBoucle avec une fonction modifier_eti.
2. Break/Continue : Grâce à l'info_ast en paramètre, on récupère les étiquettes qui nous intéressent dans l'InfoBoucle pour effectuer le jump au bon endroit (étiquette de fin de boucle pour le break et étiquette de début de boucle pour le continue).

6 Surcharge des fonctions

La surcharge de fonction est une fonctionnalité non demandée dans le sujet, que nous avons décidé de traiter entièrement (code + test). La surcharge de fonction offre la possibilité de définir plusieurs fonctions de même nom, mais qui diffèrent par le nombre ou le type des paramètres effectifs.

6.1 Lexer et Parser

Pas de changement pour le lexer et le parser.

6.2 AST

Avec la surcharge, on peut désormais avoir plusieurs fonctions portant le même nom, mais qui se distinguent soit par leur nombre de paramètres soit par les types de ces paramètres, n'ayant pas accès à ses informations lors de la passe de gestion des identifiants, nous ne sommes donc pas en mesure de choisir la bonne fonction à appeler lors d'une expression de type : appel de fonction. On va donc laisser la passe de typage s'occuper de choisir la bonne fonction à appeler, il faut pour cela, lui fournir une liste qui contient l'ensemble des fonctions qui portent le même nom. Ce qui veut dire que nous devons modifier l'AstTds pour ne plus avoir une seule fonction : AppelFonction of info_ast*expression list mais une liste de fonction : AppelFonction of (info_ast list)*expression list.

6.3 Passes

- Passe Tds : Deux choses à modifier :

1. La fonction d'analyse de fonction : Avant on levait directement une exception de double déclaration si on trouvait une fonction déjà existante portant le même nom, maintenant, on vérifie en plus si le nombre de paramètres est égal ou si les types des paramètres sont égaux, avant de lever l'exception, si ce n'est pas le cas alors on ajoute à la tds cette nouvelle fonction.

2. L'expression d'appel de fonction : Avant on cherchait dans la tds l'unique fonction qui était appelée et on la renvoyait, maintenant, on renvoie une liste avec toutes les fonctions grâce à la fonction chercherGlobalementFonction que nous avons défini dans tds.ml et on renvoie cette liste.

- Passe Type : Seule l'expression du type : Appel de fonction est modifiée. Comme dit précédemment, c'est à la passe de typage de renvoyer la bonne fonction dans la liste qu'on lui a fourni en sortie de passe Tds. On a donc crée une fonction interne qui permet de parcourir cette liste en regardant pour chaque fonction si la liste des types est compatible en taille et en type respectif, une fois trouvée, on la renvoie.
- Les 2 autres passes restent inchangées.

7 Conclusion

Au final, nous avons rajouté différents tests pour ces nouvelles structures et nous avons essayé de corriger les différents warnings qui pouvaient être affichés à la compilation du projet.

Pour ce qui est de la surcharge de fonction, l'implémentation fonctionne pour les 3 premières passes. Pour la passe de génération de code en revanche, lorsque deux fonctions de même nom sont déclarées, la fonction analyse_code_fonction leur donne un label correspond à leur nom. On se retrouve donc avec deux labels identiques ce qui génère une erreur de syntaxe du code assembleur et nous ne pouvons donc pas exécuter les tests. Nous n'avons pas réussi à trouver une solution qui ne restreint pas l'utilisateur sur le choix des noms de fonctions pour palier ce problème. Il faudrait donner une étiquette arbitraire aux fonctions pour qu'il n'y est pas de conflit.

Pour les boucles "loop" à la Rust, l'implémentation fonctionne dans son ensemble. Nous n'avons pas géré la génération de warnings dans le cas où deux boucles porteraient le même nom et nous n'avons pas non plus géré le fait qu'un warning "unreachable expression" est appelé lorsque du code est présent après un continue ou un break. Nous avons cependant ajouté un aspect qui n'existe pas de base dans le langage Rust, qui est de lever une exception de doubles déclarations lorsque des break et des continue de même nom sont déclarés localement, ce n'est pas forcément utile, mais cela permet d'éviter en partie d'avoir du code mort.

Nous avons rencontré une difficulté lors de l'implémentation des boucles. En effet, lors de la passe de gestion des identifiants, l'AstTds est modifié de telle sorte que les break/continue aient en argument, l'information de la boucle qui leur est associée. Cela permet au break et au continue d'accéder aux étiquettes quand elles sont ajoutées dans l'InfoBoucle. Or le choix d'implémentation que nous avons utilisé, fait que l'InfoBoucle que l'on donnait aux break/continue n'était qu'une copie de la vraie. Les deux informations ne pointant pas vers la même adresse mémoire, les modifications n'étaient donc jamais perçu par les break/continue. Nous avons réglé ce problème avec la fonction trouverDansTds qui permet de retourner la véritable information de la boucle.