Rapport projet de programmation fonctionnelle et de traduction des langages

Corentin Cousty
Wilkens Marc Johnley Joseph

Décembre 2024

Table des matières

1 Introduction **2**
1.1 Structure du compilateur et organisation des passes 2
1.2 Les fonctionnalités ajoutées 2

2 Extensions du langage RAT **2**
2.1 Les pointeurs 2
2.2 Les variables globales 3
2.3 Les variables statiques locales 3
2.4 Les paramètres par défaut 3

3 Méthodologie **3**
3.1 Pointeurs 3
3.2 Variables globales 5
3.3 Variables statiques locales 5
3.4 Paramètres par défaut 6

4 Résultats et analyse **7**
4.1 Présentation des cas de test et validation 7
4.2 Analyse des résultats obtenus 7

5 Discussion **7**
5.1 Alternatives 7
5.2 Difficultés rencontrées et choix de conception 8
5.3 Améliorations potentielles 8

6 Conclusion **8**

1 Introduction

Dans ce projet, nous avons à étudier et à améliorer le langage RAT, un langage simplifié conçu à des fins pédagogiques. L'objectif était d'ajouter de nouvelles fonctionnalités au compilateur RAT pour enrichir son expressivité et sa puissance tout en conservant sa simplicité d'utilisation.

1.1 Structure du compilateur et organisation des passes

Le compilateur RAT est organisé en plusieurs passes successives. Chaque passe applique des transformations spécifiques sur le code :

- **Analyse syntaxique :** conversion du code source en un arbre syntaxique abstrait (AST).
- **Analyse des identifiants :** validation et résolution des noms dans la table des symboles (TDS).
- **Typage :** vérification des types des expressions et instructions.
- **Placement mémoire :** calcul des adresses mémoires des variables.
- **Génération de code TAM :** traduction de l'AST en instructions TAM exécutables.

1.2 Les fonctionnalités ajoutées

- Les pointeurs, permettant de manipuler des adresses mémoires directement.
- Les variables globales, accessible à tout endroit du programme y compris les fonctions.
- Les variables statiques locales, qui conservent leur valeur entre plusieurs appels de la même fonction.
- Les paramètres par défaut, permettant de simplifier les appels de fonctions en réduisant le nombre d'arguments obligatoires grâce à des valeurs par défaut.

2 Extensions du langage RAT

2.1 Les pointeurs

Les pointeurs permettent d'accéder directement à des adresses mémoires. Ils sont utilisés pour manipuler des structures complexes ou partager des

données entre différentes parties d'un programme. Dans RAT, nous avons ajouté la possibilité de déclarer, d'affecter et de déréférencer des pointeurs.

- **Grammaire :** ajout de la déclaration et de l'utilisation de pointeurs.
- **TDS :** gestion des identifiants pour les pointeurs.
- **Code TAM :** génération des instructions TAM pour l'allocation et l'accés mémoire.

2.2 Les variables globales

Les variables globales permettent de partager des données entre les différentes fonctions d'un programme. Nous avons ajouté leur gestion dans le langage RAT.

- **Grammaire :** ajout des déclarations globales.
- **Placement mémoire :** allocation en segment de base (SB).
- **Code TAM :** accès à ces variables via des adresses fixes.

2.3 Les variables statiques locales

Ces variables conservent leur valeur entre plusieurs appels à une fonction, comme en C.

- **Grammaire :** ajout des déclarations statiques.
- **TDS :** marquage des variables comme statiques.
- **Code TAM :** conservation des valeurs sur le segment de base (SB).

2.4 Les paramètres par défaut

Les paramètres par défaut simplifient l'utilisation des fonctions en fournissant des valeurs par défaut.

- **Grammaire :** support des valeurs par défaut dans les déclarations de fonctions.
- **Analyse syntaxique :** gestion des paramètres optionnels.
- **Code TAM :** passage automatique des valeurs par défaut lors des appels.

3 Méthodologie

3.1 Pointeurs

Modifications de la grammaire et analyse syntaxique La grammaire a été étendue pour permettre la déclaration et l'utilisation des pointeurs, notamment leur déréférencement et leur affectation.

- Un type "Pointeur t" et un type "affectable" ont donc été ajoutés pour traiter les pointeurs et tous les identifiants de la même manière.
- Cela entraine le remplacement de l'expression Ident par Affectable qui inclue Ident et Deref.
- Il y a aussi l'ajout des expressions New of typ et Adresse of string pour déclarer un Pointeur et accéder à l'adresse du pointeur.
- Il y a aussi l'expression et type Null pour les pointeurs pour définir le pointeur Null. Nous n'avons pas utilisé Undefined car nous le considérons réservé aux cas d'erreurs, lorsqu'il n'y a pas de type et faire un Pointeur of Undefined signifie qu'il faut rendre Undefined compatible avec tous les autres types pour éviter des erreurs ce qui semble peu cohérent.

Gestion des identifiants Les pointeurs ont été ajoutés dans une nouvelle expression dans les AST : Affectable.

- On ajoute le type :
  type affouexpTds =
  | Affectable of AstTds.affectable
  | Expression of AstTds.expression
- Ce type permet de stocker des Affectable ou des Expressions de AstTds. Il est utile pour que analyse_tds_affectable puisse renvoyer des entiers donc une expression au lieu d'ajouter les Entier aux Affectable et de les garder à chaque passe alors que ce n'est pas pertinent.
- La vérification des types et des déréférencements a été incluse.
- Une fonction analyse_tds_affectable a été ajouté dans cette passe pour les traiter.
- Le type est aussi placé dans l'InfoVar directement lors de la déclaration dans cette passe.

Gestion des types On vérifie qu'on déréférence bien des Pointeurs en vérifiant le type.

Placement mémoire Il a fallu définir la taille du type Pointeur dans la fonction getTaille dans le module type. La taille d'un pointeur est 1.

Génération de code Pour générer le code il a d'abord fallu ajouter des fonctions telles que get_type_affectable pour obtenir le type d'une expression Affectable.

- La fonction analyse_code_affectable analyse les expressions Affectable en vérifiant si l'accès est en écriture ou non.
- Pour l'expression New, on utilise l'instruction tam MAlloc pour allouer l'espace nécessaire au pointeur.
- Pour l'expression Adresse, on utilise loada pour l'obtenir.

3.2 Variables globales

Modifications de la grammaire et analyse syntaxique Les variables globales sont séparées du reste du code grâce au parser. Il y a un nouveau type : $variable_globale = DeclarationGlobaleoftyp * string * expression$ Et à présent un programme est constitué tel que $programme = Programmeofvariable_globalelist * fonctionlist * bloc$

Gestion des identifiants Les variables globales sont enregistrées dans la TDS originelle avec une portée globale.

- Ajout de analyse_tds_variable_globale pour les analyser en les plaçant directement dans la tds principale.
- Ils deviennent une liste de AstTds.Declaration après cette passe - c'est donc un bloc d'instruction.

Type, Placement mémoire et Génération de code Les variables globales ont des adresses fixes dans le registre SB et sont placées au tout début, on les analyse donc comme des blocs dans toutes les autres passes.

Précision pour la passe de placement : on récupère la place occupée par leurs déclarations puis on donne toujours en argument le décalage actuel dans SB à analyse_placement_fonctions et analyse_placement_bloc pour l'analyse du bloc principal du programme pour ne pas réécrire par dessus ces emplacements utilisés.

3.3 Variables statiques locales

Modifications de la grammaire et analyse syntaxique La grammaire a été enrichie pour inclure les déclarations statiques locales, limitées à une fonction avec DeclarationStatic et le mot clé static.

Gestion des identifiants Les variables sont simplement ajoutées dans la tds avec un InfoVar.

Gestion des types On les traite comme les instructions Declaration mais on renvoie une instruction DeclarationStatic.

Placement mémoire On a créé une fonction analyse_placement_instruction_fonction pour analyser les instructions des fonctions à part, notamment pour les variables statiques, cela permet d'être sûr qu'elles sont utilisées dans les fonctions.

- On place donc les variables statiques dans SB, selon le décalage.
- La fonction qui analyse les instructions d'une fonction renvoie la taille occupée dans LB et SB pour suivre l'occupation des registres constamment. Les autres fonctions touchant à l'analyse des fonctions sont donc adaptées en conséquence.
- On récupère aussi la liste des déclarations statiques pour les sortir du traitement des fonctions et les placer directement dans le programme grâce à separer_declaration_static qui récupère séparément les instructions de la fonction - utilisant le registre LB - et les DeclarationStatic.
- Après cette passe on a donc $programme = Programmeofbloc*fonctionlist*$ $bloc*bloc$ pour contenir le bloc des variables globales, le bloc des instructions des fonctions, le bloc des variables statique et le bloc principal.

Génération de code Il suffit de récupérer le bloc de variables statiques dans le programme et d'appeler analyse_code_bloc dessus.

3.4 Paramètres par défaut

Modifications de la grammaire et analyse syntaxique Les paramètres par défaut ont été ajoutés à la grammaire pour permettre leur définition dans les fonctions.

- On a ajouté le type $defaut = Defautofexpression$
- Ainsi la liste des paramètres d'une fonction contient aussi un champ de type $defautoption$ pour prendre une valeur par défaut optionnelle.

Gestion des identifiants Dans cette passe on ajoute les paramètres par défaut à l'appel s'il en manque.

- On modifie InfoFun pour ajouter un champ pour contenir la liste des valeurs par défaut des paramètres sous la forme de $defautoptionlist$. None quand le paramètre n'a pas de valeur par défaut et Some d s'il en a une.
- On ajoute une fonction verifier_param dont le but est de vérifier que toutes les paramètres obligatoires soient présentes avant ceux par défauts. On récupère aussi les valeurs par défaut des paramètres sous la forme de defaut option pour les placer dans l'InfoFun de la fonction.
- Ensuite la fonction completer_arguments complète les arguments donnés lors de l'appel avec les paramètres par défaut disponibles pendant l'analyse de l'appel de fonction. Ensuite on renvoie simplement un AstTds.AppelFonction.

Type, Placement mémoire et Génération de code Il n'y avait rien à changer car les paramètres par défaut sont devenus partis intégrante de l'appel.

4 Résultats et analyse

4.1 Présentation des cas de test et validation

Chaque fonctionnalité a été testée avec des cas d'utilisation simples et complexes pour valider son comportement y compris les fonctions internes aux passes à l'aide de tests unitaires.

4.2 Analyse des résultats obtenus

Les résultats montrent que toutes les nouvelles fonctionnalités fonctionnent correctement dans les scénarios prévus.

5 Discussion

5.1 Alternatives

- Pour les variables globales nous avons hésité nous pensions au début ajouter une instruction DeclarationGlobale pour les gérer dans tout le

programme comme des expressions normales. Puis nous avons trouvé une alternative beaucoup plus simple à implanter.

- A l'origine nous pensions utiliser des InfoVarStatic pour gérer les variables statiques mais ce n'était pas nécessaire.

5.2 Difficultés rencontrées et choix de conception

Certaines difficultés ont été rencontrées lors de l'implémentation, notamment :

- Gestion des TDS pour les variables statiques et la gestion des Entier parmis les Affectable qui a mené à la création de affouexpTds.
- Compatibilité avec les anciens tests lorsque j'ai voulu ajouter des Exceptions plus précises.

5.3 Améliorations potentielles

Pour aller plus loin, voici quelques pistes d'amélioration :

- Permettre de déclarer des variables globales n'importe où : grâce à la fonction permettant de récupérer la tds originelle, il est possible d'utiliser la fonction analyse_tds_variable_globale sur toutes les expressions DeclarationGlobale, en les ajoutant aux autres expressions normales, il ne serait donc plus nécessaire de séparer les variables globales au début du programme.
- Ajouter le support des tableaux.
- Inclure des structures de données comme les structs vu en examen.
- Optimiser le code TAM généré pour réduire son empreinte mémoire.

6 Conclusion

En conclusion, ce projet a permis d'ajouter des fonctionnalités puissantes au langage RAT tout en renforçant la structure de son compilateur. Les pistes d'amélioration offrent de nombreuses opportunités d'enrichissement pour ce langage. On remarque aussi que notre manière d'analyser et de faire les passes permet d'implanter de nouvelles fonctionnalités assez facilement, peu de modifications extérieures aux passes et au parser sont nécessaires.