Traduction des langages

SARHANE ABDELMOUHAIMEN
OURKIA ABDELHAKIM

Département Sciences du Numérique - Deuxième année
2023-2024

Contents

1 Introduction

Le projet présenté s'inscrit dans la continuité du développement du compilateur pour le langage RAT, qui, dans sa forme initiale, traitait les aspects fondamentaux de la compilation : la gestion des identifiants, le typage, le placement en mémoire et la génération de code. La version de base, bien que fonctionnelle, ne couvrait pas l'éventail complet des fonctionnalités fréquemment requises par les programmeurs modernes. En particulier, des constructions telles que les pointeurs, les tableaux, les boucles "for" et les instructions de contrôle de flux 'goto' n'étaient pas prises en charge. Ce projet vise donc à pallier ces manques en enrichissant le compilateur du langage RAT avec ces constructions additionnelles, en assurant leur intégration cohérente dans le cadre des quatre passes de compilation existantes.

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

Figure 1: Grammaire (EBNF) du langage RAT étendu

2 Modification de l'AST et Tds

2.1 Pointeurs

- L'ajout d'un nouveau cas de type : `Pointeur of type`.
- L'ajout du nouveau type : `affectable = Ident of string | Deref of affectable`.
- Modifier un cas d'instruction : `Affectation of affectable * expression`.
- L'ajout des cas aux expressions : `Affectable of affectable | Null | New of typ | Adresse of string`.

2.2 Tableaux

- Introduction du type `Tableau of typ` pour représenter des tableaux de n'importe quel type.
- Ajout de `Newtableau of typ * expression` pour la création de tableaux.
- Ajout de `Initialisation of expression list` pour l'initialisation des tableaux.
- Extension de `affectable` pour inclure `Acces of affectable * expression` pour l'accès aux éléments du tableau.

2.3 Boucles "For"

- Ajout de `For of typ * string * expression * expression * string * expression * bloc`, représentant la structure d'une boucle for.

2.4 Goto

- Introduction de `Goto of string` pour le saut à une étiquette et `Etiquette of string` pour la définition d'une étiquette.

3 Passe de Gestion d'Identifiants (AstTds)

Cette section se concentre sur la gestion des identifiants et leur portée dans la table des symboles.

3.1 pointeurs

Pour la gestion des identifiants, la première étape consistait naturellement à incorporer les modifications déjà présentées dans la section ASTSyntax à l'AstTds, en remplaçant les chaînes de caractères par des références Tds.infoast. Par la suite, nous avons introduit une méthode dédiée à l'analyse des affectables, prenant en compte les deux situations possibles : lorsque l'affectable se trouve à gauche ou à droite de l'affectation. Il était ensuite nécessaire de traiter les occurrences du type Affectable ajouté à l'AST au sein des méthodes d'analyse des expressions et des instructions.

3.2 tableaux

Pour les tableaux, ASTTDS a été étendu pour traiter les nouvelles formes d'expressions et d'affectations. Cela inclut la gestion des expressions de création de nouveaux tableaux (Newtableau) et l'initialisation des tableaux (Initialisation). La complexité résidait dans la vérification des dimensions et types des tableaux lors des accès et affectations.

3.3 Boucles For

L'introduction des boucles "for" a entraîné la nécessité d'une gestion spécifique des variables d'itération au sein de la Table des Symboles (TDS). Chaque boucle "for" possède sa propre portée, et les variables d'itération sont traitées comme des variables locales à cette boucle. En d'autres termes, lorsqu'une boucle "for" est rencontrée, une vérification préalable est effectuée pour s'assurer que la variable d'itération n'est pas déjà définie dans le bloc courant. Ensuite, une TDS fille est créée, dans laquelle la variable d'itération est ajoutée, et cette TDS est utilisée pour analyser à la fois l'expression définissant la condition d'arrêt et l'instruction de mise à jour de la variable d'itération. Cette TDS spécifique à la boucle "for" est ensuite transmise à la fonction **analyse_tds_bloc** pour l'analyse du bloc associé au "for". La transformation de l'AST implique ainsi la prise en charge des informations relatives aux variables d'itération et à la structure même de la boucle.

3.4 Goto

Dans la phase de gestion des identifiants, nous avons incorporé la prise en charge des instructions **Goto(id)** et **Etiquette(id)**. Afin de permettre à une étiquette d'avoir le même nom qu'une variable, une constante ou une fonction, nous avons choisi d'adopter une approche qui implique l'utilisation d'une table de symboles distincte réservée à la gestion des identifiants des étiquettes. Ceci garantit qu'aucune étiquette ne partage le même nom, même au sein de blocs différents. Concrètement, une seule table des symboles (TDS) est créée pour le bloc principal, et elle est responsable de la gestion de toutes les étiquettes définies à l'intérieur de ce bloc.

Pour chaque définition de fonction, une TDS distincte est créée, dédiée à la gestion de toutes les étiquettes définies et utilisées dans cette fonction spécifique.

Afin de permettre l'utilisation de labels non encore définis, mais existants ailleurs dans le programme, nous avons introduit des liste en tant que des variables globales. De plus, un autre paramètre a été ajouté aux fonctions **(analyse_tds_instruction, analyse_tds_bloc)**. Dans la fonction principale d'analyse, la variable globale **(ma_liste_globale_main)** est instanciée et transmise à (analyse_tds_bloc).

Lorsqu'une instruction **Goto(id)** est rencontrée, une vérification préalable est effectuée pour déterminer si l'identifiant est présent dans la TDS réservée aux étiquettes. Si c'est le cas, un **InfoGoto(n)** est créé, remplaçant l'identifiant par un pointeur pointant vers **InfoGoto(n)**. Si l'identifiant n'est pas trouvé dans la TDS, une autre vérification est effectuée dans la liste des étiquettes non encore définies. Si l'identifiant n'est pas présent, il est ajouté à la liste avec son info_ast.

Lorsqu'une instruction **Etiquette(id)** est rencontrée, nous vérifions que l'identifiant n'est pas déjà présent dans la TDS réservée aux étiquettes. S'il est absent, il est ajouté à la TDS et retiré de la liste des étiquettes non encore définies. Cette procédure est également appliquée aux fonctions, où une variable globale est créée pour être utilisée dans le corps de la fonction, accompagnée d'une TDS spécifique pour les étiquettes de cette fonction. La fonction **analyse_tds_fonction** retourne également la liste globale mise à jour.

En conclusion, dans la fonction principale d'analyse, les listes du programme principal et de chaque fonction sont récupérées. Si la taille de ces listes est nulle, cela indique qu'aucune étiquette n'est indéfinie. Cependant, si une liste n'est pas de taille nulle, cela signifie qu'une étiquette dans le code est utilisée sans être définie.

3.5 Défis et Solutions

Un défi majeur était de maintenir la cohérence de la TDS tout en introduisant ces nouvelles constructions. Par exemple, dans le cas des pointeurs, il fallait s'assurer que la déréférenciation et l'adressage ne violaient pas les règles de portée. Pour les tableaux, le défi était de gérer

correctement les expressions complexes d'initialisation et d'accès. Les boucles "for" et les instructions goto nécessitaient une gestion précise des portées pour éviter les conflits et erreurs d'interprétation.

4 Passe de Typage (AstType)

Discussion sur la passe de typage, y compris l'inférence de type et la vérification du type des constructions étendues.

4.1 pointeurs

Durant la phase de typage, nous avons étendu notre système en introduisant un nouveau type, le "Pointeur de type", et mis à jour les différentes fonctions d'analyse pour tenir compte des instructions et des expressions liées aux pointeurs. De plus, nous avons incorporé les jugements de typage suivants dans le système de types du langage RAT, assurant ainsi la cohérence de l'utilisation des pointeurs.

$$\sigma \vdash \text{ null } : \text{ Pointeur(Undefined)}$$

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

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

$$\frac{\sigma \vdash a : \text{ Pointeur (T)}}{\sigma \vdash *a : \tau}$$

4.2 tableaux

Pour les tableaux,ous avons ajouté un nouveau type, à savoir **Tableau of type**, et nous avons modifié le code pour que la passe de typage vérifie la cohérence des types lors de la création (Newtableau) et de l'initialisation (Initialisation). Nous avons particulièrement mis l'accent sur la conformité des types des éléments du tableau et la validité des indices utilisés dans les expressions d'accès.

$$\frac{\sigma \vdash CP : \tau * ...\tau \quad \sigma \vdash E : \text{ int}}{\sigma \vdash \{CP\} : \text{tab}(\tau)}$$

$$\frac{\sigma \vdash TYPE : \tau \quad \sigma \vdash E : \text{ int}}{\sigma \vdash \text{ new } TYPE[E] : \text{tab}(\tau)}$$

$$\frac{\sigma \vdash A : \text{tab}(\tau) \quad \sigma \vdash E : \text{ int}}{\sigma \vdash (A[E]) : \tau}$$

4.3 Boucles For

Dans les boucles "for", nous avons vérifié que la variable d'itération était correctement déclarée et utilisée. Les expressions de contrôle de la boucle, telles que la condition d'arrêt et l'expression

de mise à jour, ont été analysées pour garantir leur conformité avec le type attendu de la variable d'itération.

$$\frac{\sigma \vdash E_1 : int \quad \sigma \vdash id : int \quad (id, int) :: \sigma \vdash BLOC : \ void, [] \quad (id, int) :: \sigma \vdash E_2 : bool \quad (id, int) :: \sigma \vdash E_3 : int}{\sigma \vdash \text{ for } (int \quad id = E_1 \quad ; \quad E_2 \quad ; \quad id = E_3) : void, []}$$

4.4 Goto

L'introduction de goto et Etiquette n'a pas posé de défis majeurs en termes de typage, car ces constructions n'impliquent pas de vérification de type complexe. Toutefois, nous avons veillé à ce que les étiquettes soient correctement identifiées et utilisées dans le contexte approprié.

4.5 Défis et Solutions

Un des défis majeurs rencontrés lors de la passe de typage était d'assurer la cohérence des types à travers les différentes constructions, en particulier pour les tableaux et les pointeurs. Pour résoudre cela, nous avons renforcé les vérifications et les inférences de type dans les cas complexes, comme les tableaux multidimensionnels et les opérations sur les pointeurs. La résolution de la surcharge pour les opérateurs binaires et unaires a également été un élément clé pour maintenir la cohérence du typage dans le langage.

5 Passe de Placement Mémoire (AstPlacement)

Analyse de la manière dont le compilateur détermine l'emplacement mémoire des différentes variables et structures.

5.1 pointeurs

Dans la phase de placement en mémoire, la taille attribuée à un pointeur est fixée à 1. Cette décision reflète le fait que les pointeurs occupent généralement un unique octet en mémoire, lequel contient l'adresse de référence vers le tas.

5.2 tableaux

Pour les tableaux, notre choix a été d'opter pour le stockage dans le tas, étant donné que la taille du tableau ne peut pas être connue à la compilation. Ainsi, une variable de type tableau contient un pointeur vers cette zone en mémoire. C'est la raison pour laquelle nous avons décidé de fixer la taille du type tableau à 1, symbolisant simplement la présence du pointeur vers la zone mémoire allouée dynamiquement.

5.3 Boucles For

La gestion des boucles "for" a nécessité l'allocation de mémoire pour les variables d'itération. Cependant, le déplacement dans la pile introduit par cette instruction est égal à 0. Cela s'explique par le fait qu'à la fin de cette instruction, les variables d'itération doivent être dépilées, annulant ainsi tout déplacement dans la pile effectué au début de la boucle.

5.4 Goto

L'introduction de goto et Etiquette n'a pas présenté de défis majeurs du point de vue du placement mémoire, car ces constructions ne nécessitent pas d'allocation mémoire spécifique. Néanmoins, nous avons veillé à ce que les étiquettes soient correctement résolues et accessibles dans le bon contexte.

6 Passe de Génération de Code (Code Cible TAM)

Cette section couvre la traduction du code source en un code intermédiaire pour la machine cible.

`analyse_code_affectable`

- Traite les différentes formes d'affectables (variables, déréférencements, accès tableau).
- Pour un `Ident`, génère le code pour charger ou stocker une valeur en fonction de `modif`.
- Pour un `Deref`, génère du code pour accéder à une adresse mémoire, en utilisant `loadi` ou `storei`.
- Pour un `Acces` (accès tableau), calcule l'adresse de l'élément dans le tableau et charge ou stocke la valeur.

`analyse_code_expression`

- Gère les différentes formes d'expressions (booléens, entiers, pointeurs, créations de tableau, etc.).
- Pour un `Newtableau`, calcule la taille nécessaire et alloue de l'espace mémoire.
- Pour une `Initialisation`, alloue l'espace mémoire pour le tableau et initialise ses éléments.
  - Calcule la taille totale nécessaire pour le tableau.
  - Alloue la mémoire pour le tableau en utilisant `Malloc`.
  - Évalue chaque expression d'initialisation du tableau et génère le code correspondant.
  - Initialise le tableau avec les valeurs calculées.
  - Ajuste la pile pour s'assurer que le tableau est correctement configuré en mémoire Tas.
- Inclut la génération de code pour des expressions unaires et binaires, ainsi que pour les appels de fonction.

`analyse_tds_instruction`

- Gère les différentes instructions du langage, y compris les affectations, déclarations, et structures de contrôle.
- Pour une `Affectation`, utilise `analyse_code_expression` et `analyse_code_affectable`.
- Pour une `Declaration`, génère le code pour initialiser une variable.
- Gère les structures de contrôle telles que les boucles `TantQue`, `Conditionnelle`, et `For`. Le code pour l'analyse des boucles "for" est expliqué ci-dessous :

  - Initialise la variable de boucle et génère le code pour la première affectation.
  - Évalue la condition de la boucle et génère le code pour le contrôle de flux.
  - Exécute le bloc d'instructions de la boucle.
  - Met à jour la variable de boucle à chaque itération.
  - Utilise des étiquettes pour contrôler le début et la fin de la boucle.
- Prend en charge les instructions `Goto` et `Etiquette`.

`analyse_code_bloc`

- Traite un bloc d'instructions, en générant le code pour chaque instruction dans le bloc.
- Utilise `analyse_tds_instruction` pour chaque instruction du bloc.
- Gère la taille de la pile nécessaire pour les variables locales dans le bloc.

`analyse_code_fonction`

- Génère le code pour une fonction, y compris l'étiquette de la fonction et le code de son corps.
- Utilise `analyse_code_bloc` pour générer le code du corps de la fonction.

`analyser`

- Point d'entrée principal pour la génération de code.
- Génère le code pour toutes les fonctions et pour le programme principal.
- Utilise `analyse_code_fonction` pour les fonctions et `analyse_code_bloc` pour le programme principal.

6.1 Défis et Solutions

Le principal défi dans la génération de code pour notre compilateur RAT concernait la gestion des pointeurs et des tableaux, notamment l'initialisation correcte des tableaux. En revanche, l'intégration de l'instruction Goto et des étiquettes n'a pas présenté de difficultés majeures.

7 Conclusion

En abordant l'extension du compilateur pour le langage RAT, ce projet nous a offert une expérience d'apprentissage enrichissante, jalonnée de défis significatifs et d'opportunités d'amélioration.

Améliorations et Perspectives

Ce projet a mis en lumière plusieurs axes d'amélioration. Une optimisation du code généré, en particulier pour les tableaux et structures de données complexes, pourrait être envisagée. Le développement d'un système de gestion des erreurs plus avancé, offrant des diagnostics plus détaillés lors de la compilation, constitue également un domaine d'amélioration prometteur. Par ailleurs, l'introduction d'optimisations au niveau du compilateur, comme l'élimination du code mort et l'inlining, pourrait significativement améliorer les performances du code compilé.

En somme, bien que le projet ait été exigeant, il a été extrêmement enrichissant et instructif. Les défis rencontrés ont stimulé notre créativité et notre persévérance, et les perspectives

d'amélioration ouvrent la voie à de futures avancées passionnantes dans le domaine de la compilation de langages de programmation.