Projet de Programmation Fonctionnelle et de Traduction des Langages

Rapport

Hamza Atif Marieme Chakhs

16 janvier 2025
ENSEEIHT – 2SN-A

Table des matières

1 Introduction

Ce projet a pour objectif d'enrichir les fonctionnalités du compilateur RAT → TAM, développé dans le cadre des séances de TP et TD de Traduction des Langages. Les améliorations apportées visent à étendre les capacités initiales du compilateur en y intégrant plusieurs nouvelles fonctionnalités :

— Gestion des pointeurs
— Prise en charge des variables globales
— Prise en charge des variables locales statiques
— Ajout de paramètres par défaut
— Combinaisons des différentes constructions

Ces extensions, décrites en détail dans la section suivante, ont été développées conformément aux attentes du projet. Toutes les fonctionnalités demandées ont été implémentées et testées avec succès.

Un ensemble de tests a été élaboré pour valider le bon fonctionnement des extensions, et ces derniers ont donné des résultats concluants.

2 Extensions du langage

2.1 Pointeurs

Jugements de typage

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

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

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

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

Evolution des AST

L'intégration des pointeurs dans le langage RAT a conduit à une évolution significative des AST. Le type `affectable`, a été introduit pour représenter les affectables, et a été étendu pour inclure des fonctionnalités liées aux pointeurs. Ainsi, il comprend désormais deux cas : `Ident of string` pour représenter un identifiant classique (nom de variable) et `Deref of affectable` pour représenter le déréférencement d'un affectable, permettant d'accéder à la valeur pointée par un pointeur. Par ailleurs, le type `expression` a été enrichi avec les cas `Adresse of string` pour accéder à l'adresse d'une variable, `Null` pour représenter un pointeur nul, et `New of typ` pour l'initialisation de nouveaux pointeurs. Enfin, le type `typ` a été modifié pour inclure `Pointeur of typ`, permettant de représenter des pointeurs sur des types et facilitant la gestion des pointeurs chaînés. Ces ajouts permettent au langage de manipuler efficacement les pointeurs tout en maintenant une structure claire pour les opérations sur la mémoire.

Implémentation

Dans la passe de gestion des identifiants, un paramètre `modif` a été introduit pour distinguer les contextes de modification et de lecture des identifiants, garantissant que seules les variables modifiables peuvent être utilisées dans des affectations. La gestion des pointeurs a été étendue avec l'ajout du cas `Deref` pour traiter les expressions de type `*aff`, et les expressions `New`, `Null` et l'opération `Adresse` ont été adaptées pour produire des pointeurs. Les affectations et déclarations, ainsi que les blocs et fonctions, ont été modifiés pour gérer les pointeurs correctement. Dans la passe de typage, le type `Pointeur` a été ajouté, et une gestion spécifique a été mise en place pour les affectables via le constructeur `Deref`, vérifiant que le type est bien un `Pointeur(t)`. Les affectations ont été modifiées pour inclure des vérifications de compatibilité des types, et les variables globales et locales, y compris les statiques, peuvent maintenant être des pointeurs, avec un typage vérifié. Les fonctions ont également été ajustées pour gérer des paramètres et des retours de type pointeur, assurant ainsi une prise en charge complète et cohérente des pointeurs dans le système.

2.2 Les variables globales

Jugements de typage

$$\frac{\sigma \vdash E : T}{\sigma \vdash static\, T\, id = E : void}$$

Évolution des AST

Pour intégrer la gestion des variables globales, le lexer et le parser ont été modifiés afin de respecter la nouvelle grammaire, notamment avec l'ajout du mot-clé `STATIC` et d'un nouveau type pour l'attribut synthétisé `<var> var`. Dans le fichier `ast.ml`, un nouveau type nommé `var` a été ajouté pour représenter explicitement les variables globales. Dans la structure du programme, une liste de type `var list` dédiée aux variables globales a été ajoutée pour les distinguer des autres éléments.

Implémentation

Lors de la passe de gestion des identifiants, les variables globales sont insérées dans la `mainTDS` du programme, ce qui les rend accessibles depuis n'importe quelle partie du code. Pendant la phase de typage, le type de ces variables est vérifié afin de garantir leur cohérence.

Durant la phase de placement mémoire, les variables globales sont toutes placées dans le registre `SB` (Static Base). Le déplacement mémoire de ces variables est calculé en premier, et ce déplacement sert de point de départ pour le placement mémoire du reste du programme.

Une fonction spécifique, `analyse_code_vars`, a été implémentée pour gérer la génération de code des variables globales. Cette fonction s'inspire des mécanismes déjà existants et utilise les déplacements et registres calculés pendant la phase de placement mémoire.

Grâce à cette organisation, le code correspondant aux variables globales est généré sans problème.

2.3 Les variables statiques locales

Jugements de typage

$$\frac{\sigma \vdash E : T}{\sigma \vdash static\, T\, id = E : void}$$

Évolution des AST

Un nouveau type d'instruction a été introduit avec le constructeur `Staticlocal` dans les modules `AstSyntax` et `AstTds`, permettant de représenter une variable locale statique, incluant son type, son identifiant et son expression d'initialisation. Ce constructeur a été ajouté pour gérer spécifiquement les variables locales statiques dans les blocs d'instructions. Cependant, dans le module `AstType`, ces variables n'apparaissent plus dans le bloc d'instructions puisqu'elles sont traitées comme des variables globales.

Implémentation

L'implémentation des variables locales statiques a nécessité plusieurs ajustements :

- **Gestion des Identifiants :**Lors de la phase de gestion des identifiants, on ajoute un nouveau type d'instruction pour représenter une déclaration de variable locale statique, qu'on traite de façon similaire à une déclaration normale.
- **Phase de Typage :** En phase de typage, nous avons créé une série de fonctions qui transforment les variables locales statiques en variables globales. Ces variables sont ensuite intégrées à la liste des variables globales du programme et, par conséquent, placées dans le registre `SB` sans difficulté.

2.4 Les paramètres par défaut

Jugements de typage

$$\frac{\sigma \vdash E_1 : T_1 \quad \sigma \vdash E_2 : T_2 \quad \dots \quad \sigma \vdash E_n : T_n}{\sigma \vdash TYPE\ id_1 = E_1 \quad TYPE\ id_2 = E_2 \quad \dots \quad TYPE\ id_n = E_n : T_1 \times T_2 \times \dots \times T_n}$$

Evolution des AST

L'intégration des paramètres par défaut dans l'AST a entraîné l'ajout d'un nouveau type `defaut` pour représenter ces paramètres. Ce type est défini par `type defaut = Defaut of expression`, permettant ainsi d'associer une valeur par défaut à chaque paramètre de fonction. Cette évolution a modifié la structure des paramètres dans les fonctions, en incluant une option pour spécifier une expression représentant la valeur par défaut. Cette option permet de gérer les paramètres avec ou sans valeur par défaut, et a été prise en compte lors de la phase d'analyse syntaxique et des identifiants. Ainsi, l'AST a été ajustée pour gérer de manière flexible les paramètres facultatifs, tout en maintenant la possibilité de déclarer des paramètres obligatoires.

Implémentation

Les seuls ajustements apportés pour implémenter les paramètres par défaut ont été effectués au niveau de la phase de gestion des identifiants. Lors de l'analyse d'une fonction avec `analyse_tds_fonction`, une liste des paramètres par défaut est générée à l'aide de la fonction `get_liste_param_defaut`, qui parcourt les paramètres de la fonction et les ajoute à une table de hachage `param_defaut`. Cette table associe chaque fonction à ses paramètres par défaut. Lors de l'analyse d'un appel de fonction, la fonction `param_complet` compare les arguments fournis avec les paramètres par défaut et les complète si nécessaire. De plus, `get_liste_param_defaut` vérifie que les paramètres par défaut ne suivent pas un paramètre sans valeur par défaut, car si un paramètre avec une valeur par défaut était placé avant un paramètre sans valeur par défaut, il deviendrait impossible de déterminer quels arguments sont associés à quels paramètres lors de l'appel de la fonction, car il n'y aurait pas de correspondance explicite entre les arguments fournis et les paramètres sans valeur par défaut. Cela entraînerait une ambiguïté dans l'appel de la fonction et provoquerait une erreur de syntaxe. Enfin, si un paramètre par défaut est défini, son expression associée est analysée et transformée en une expression de type `AstTds.expression`.

2.5 Combinaisons des différentes constructions

L'intégration des pointeurs, des variables globales, des variables statiques locales et des paramètres par défaut dans le compilateur du langage RAT a été conçue pour garantir leur fonctionnement non seulement de manière individuelle, mais également en combinaison. Les tests effectués ont confirmé que ces différentes constructions interagissent de manière cohérente et fiable, même dans des scénarios complexes.

Le succès des tests de combinaisons démontre la robustesse des choix d'implémentation et la capacité du compilateur à gérer des cas complexes tout en respectant les principes de la programmation fonctionnelle. Cela garantit une prise en charge cohérente et performante des fonctionnalités avancées dans le langage RAT.

3 Conclusion

Le projet a présenté plusieurs défis, principalement liés à la détermination des structures de données appropriées et au choix des traitements à effectuer lors des différentes passes pour

les extensions. La programmation en OCaml n'a pas constitué un obstacle majeur, grâce à l'utilisation d'itérateurs et de techniques de filtrage qui ont facilité l'implémentation. Cependant, un problème important est survenu lors de la phase de placement mémoire, en particulier pour les variables statiques locales. Le calcul du déplacement correct de ces variables dans les fonctions s'est avéré complexe, car la fonction `analyse_placement_fonction` appelait `analyse_placement_bloc` avec le registre LB, tandis que nous souhaitions utiliser le registre SB pour garantir la persistance des valeurs après la sortie de la fonction. Les calculs nécessaires ont été difficiles à réaliser, ce qui nous a conduits à adopter une solution alternative plus simple : traiter les variables locales statiques comme des variables globales. Cette approche a simplifié leur placement dans le registre SB et a permis de contourner les difficultés initiales. Enfin, ce projet a été une expérience enrichissante et plaisante, nous permettant de relever des défis intéressants tout en approfondissant nos compétences.