Projet de Programmation Fonctionnelle et de Traduction des Langages

Rapport

Enzo PETIT Nam VU

13 janvier 2022
ENSEEIHT – 2SN-A

Table des matières

1 Introduction **2**

2 Extensions du langage **2**
2.1 Pointeurs 2
2.2 Opérateur d'assignation d'addition 3
2.3 Types nommés 4
2.4 Enregistrements 4

3 Conclusion **5**

1 Introduction

Ce projet cherche à étendre les capacités du compilateur RAT → TAM développé lors des séances de TP/TD de Traduction des Langages. Les extensions réalisées rajoutent au compilateur les fonctionnalités suivantes :

— Pointeurs
— Opérateur d'assignation d'addition
— Types nommés
— Enregistrements

Les détails d'implémentation des différentes extensions sont énumérés dans la section suivante.

Dans notre rendu du projet, tous les points demandés ont été réalisés et sont à priori fonctionnels, excepté le traitement bonus des structures récursives.

Des tests ont par ailleurs été rédigés et passent avec avec succès.

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

Un nouveau type `affectable` représentant les affectables du langage est défini par `Ident of string/TDS.info_ast` et `Deref of affectable` pour les déréférencements.

Le type `expression` contient de plus `Adresse of string/TDS.info_ast` (adresse d'une variable), `Null` (pointeur null) et `New of typ` (nouveau pointeur de type `typ`).

Le type `typ` comprend un `Pointeur of typ` représentant les pointeurs. Etant récursif il permet d'enchaîner les pointeurs.

Implémentation

Dans les différentes passes une analyse récursive des affectables a été rajoutée.

Pour permettre l'affectation du pointeur `null`, on utilise le type `Pointeur Undefined` et on a autorisé les affectations/déclarations entre ce type et n'importe quel autre pointeur.

La principale difficulté dans les pointeurs fût au niveau de la génération de code : il faut récursivement déréférencer avec des `LOADI` jusqu'à arriver au *niveau de déréférencement* voulu et là faire un `STOREI/LOADI` de la taille de ce qu'on pointe.

2.2 Opérateur d'assignation d'addition

Jugements de typage

$$\frac{\sigma \vdash a : int \quad \sigma \vdash e : int}{(a, int) :: \sigma \vdash a \mathrel{+}= e : void}$$

$$\frac{\sigma \vdash a : rat \quad \sigma \vdash e : rat}{(a, rat) :: \sigma \vdash a \mathrel{+}= e : void}$$

Evolution des AST

De manière similaire au traitement de l'affectation, on a tout d'abord ajouté une nouvelle instruction `AddAff`, permettant de représenter l'addition-affectation, dans les AST `Syntaxe` et `TDS`.

Puis, de manière équivalent au traitement de l'addition, on gère la surcharge dans la passe de typage grâce à deux nouvelles instructions qui viennent remplacer la première en fonction du type de l'opération (sur des rationnels ou sur des entiers).

Implémentation

Pour des variables normales, l'implémentation de l'addition-affectation ne rajoute pas de point particuliers comparé aux implémentations respectives de l'addition et de l'affectation.

L'addition-affectation de pointeurs est court-circuitée (comparé à effectuer l'addition dans un premier temps puis l'affectation), pour descendre plus rapidement les chaines de pointeurs.

Au lieu de charger uniquement la variable pointée, on garde également le dernier pointeur (qui pointe sur la variable) en mémoire, on effectue le calcul, et on réutilise le pointeur sauvegardé pour réaliser l'affectation.

2.3 Types nommés

Jugements de typage

$$\frac{\sigma \vdash Tid : \tau}{\sigma \vdash typedef\ Tid = \tau : void, [Tid, \tau]}$$

Evolution des AST

Seul l'`AstSyntax` a été modifié :

- Un type `typedef` a été déclaré avec pour seul constructeur `TypedefGlobal of string * typ` et représente les typedefs globaux.
- Un constructeur `TypedefLocal of string * typ` a été rajouté à `instruction` pour les typedefs déclarés dans un bloc (locaux).
- Le constructeur du type `programme` évolue en `Programme of typedef list * fonction list * bloc` pour inclure les typedefs globaux.

Le type `typ` s'est vu rajouter un constructeur `NamedTyp of string` qui représente les types nommés.

`InfoTyp of string * typ` a été rajouté pour enregistrer les types nommés dans la TDS.

Implémentation

Le lexer et le parser ont été adaptés pour respecter la nouvelle grammaire (identifiants de type, mot clé `typedef`).

Pour la suite, la stratégie adoptée a été de *mettre à plat* tous les types utilisés dans la passe TDS afin de n'avoir plus que des types primitifs dans les passes suivantes.

Pour cela, une `analyse_tds_type` a été rajoutée dans `PasseTdsRat` et a pour but de donner un équivalent du type passé en entrée en type primitif. Il ne vérifie aucun typage, juste si un type nommé utilisé a bien été déclaré en amont.

Suite à ça, les passes suivante n'ont pas eu à être modifiées, tout est comme si il n'y avait pas de types nommés dans le langage.

2.4 Enregistrements

Jugements de typage

$$\frac{\sigma \vdash A : Struct([..., x_k : \tau, ...])}{\sigma \vdash (A.x_k) : \tau}$$

$$\frac{\sigma \vdash E_1 : \tau_1 \quad ... \quad \sigma \vdash E_n : \tau_n}{\sigma \vdash \{E_1 ... E_n\} : Struct([x_1 : \tau_1, ..., x_n : \tau_n])}$$

Evolution des AST

Le type `affectable` est amendé d'un nouveau constructeur `Acces of affectable * string/TDS.info_ast` représentant les accès à un champ d'un enregistrement.

Dans le type `expression`, `StructExpr of expression list` représente les expressions de structures.

Le type `typ` comprend un nouveau constructeur `Struct of (typ * string) list` représente le type enregistrement (liste des champs).

Dans le type `info`, deux nouveaux constructeurs `InfoStruct of string * typ * int * string * info_ast list` et `InfoAttr of string * typ * int` ont été rajoutés pour désigner respectivement une structure et un champ.

Implémentation

Sans rentrer dans les détails du code, dans la passe TDS on a rajouté une nouvelle fonction `analyse_tds_struct`, utilisée dans `analyse_tds_type` définie précédemment, qui se charge de vérifier si l'utilisation de la structure est correcte (pas de double déclaration, etc.). On y a aussi rajouté une fonction `creer_info_ast` qui renvoie une `info_ast` d'`InfoVar` ou d'`InfoStruct` le cas échéant. `analyse_tds_affectable` a aussi été modifié pour vérifier que les accès sur les variables sont corrects (fonction auxiliaire `analyse_acces`).

Pour la passe type, on a dû adapter `est_compatible` pour les structures.

Dans la passe de placement, `analyse_placement_struct` se charge de placer les attributs des structures, relativement à la base de la structure.

Enfin, dans la passe de génération de code, `generer_code_affectable` a été modifié afin de pouvoir y forcer un *offset* et un *type* lors des accès à l'attribut d'une structure.

3 Conclusion

La difficulté du projet résidait surtout dans la détermination des bonnes structures de données et le choix des traitements à faire lors des différentes passes pour les différentes extensions. La programmation OCaml n'était pas réellement un obstacle sur ce projet, les itérateurs et le filtrage aidant beaucoup au final.

Le déboggage n'était néanmoins pas très aisé : pas de réelle possibilité de lancer un test particuler avec des breakpoints ou de logger des valeurs sur la sortie standard facilement.

Concernant les enregistrements, les contraintes sur les bonnes définitions des structures, censées *simplifier les analyses*, nous ont parues assez compliqué à vérifier en pratique, du moins dans notre version du compilateur…

Il n'en reste pas moins qu'un projet qu'on a trouvé globalement enrichissant et assez agréable à réaliser !

PS : Nous avons trouvé que l'extension OCaml Platform pour Visual Studio Code s'averrait bien plus fonctionnelle que celle conseillée en TP…