Rapport : Projet de programmation fonctionnelle et de traduction des langages

Faisin Laurent
Guillotin Damien
2SN-M2

Introduction

Lors de nos TP, nous avons écris un compilateur. Ce compilateur part du langage source RAT et le transcrit en TAM. Le langage RAT est un langage assez simple prenant compte différentes fonctionnalités classiques telles que les boucles `TantQue` ou les blocs conditionnels `SiAlorsSinon`. Le langage TAM quand a lui est un langage proche de l'assembleur.

Le but de ce projet est d'ajouter différentes fonctionnalités au langage RAT. La première fonctionnalité ajouté est le type `pointeur`. La deuxième implantation est celle de l'opérateur `+=` qui permet d'additionner et d'affecter en une même instruction. En suite, nous avons ajouté les `typedef`. Et enfin, nous avons implanté les type `struct`. Ces trois fonctionnalités permettent au langage RAT d'être plus complet et ainsi plus efficace pour une personne souhaitant l'utiliser.

Pour chacune de ses implantations, il a fallut modiffier les codes correspondants aux différentes étapes de la compilation. La modification du `lexer` (pour reconnaitre les nouveaux mots), le parser (pour reconnaitre la nouvelle grammaire) et les différentes passes pour vérifier la cohérence sémantique du code RAT à compiler.

Modifications générales de l'AST

Lors de l'ajout des différentes extensions, nous avons évidement du modifier l'AST pour l'adapter aux différents ajouts. Le plus gros ajout, qui est commun a plusieurs extensions, est l'affectable. L'affect, comme son nom l'indique, va désigner un objet pouvant être affecté. Avant l'ajout des différentes expression, il n'y avait que les identificateurs qui pouvaient faire office d'affectbale mais maintenant, il y a les `dref`, les `attribut` d'un objet de type `struct` ou encore les identifiants.

De plus, pour chaque nouveau type, il a fallu ajouter dans l'AST de la passe typage, dans les instructions, un type print tel que `printStruct` pour les structs.

Pointeurs

L'implenation des pointeurs a commencé par l'ajout du type `Pointeur of typ`. Ainsi, le type pointeur reste assez abstrait pour pouvoir ajouter des types sans avoir a modifier du code au niveau des pointeurs. De plus, cette implentation permet de construire des "pointeurs de pointeurs".

En ce qui concerne les modifications de l'AST, nous avons ajouté la `dref` pour l'affectable. Les nouvelles expressions apportés sont `Null`, `NewType` et `Adresse`.

Les modifications apportées à la passe TDS est l'utilisation que de `dref` et d'`Adresse` doit être utilisé sur des identifiants déjà existant. Dans le cas contraire, une exception d'utilisation d'identifiant est levé.

Le plus gros du travail pour les pointeurs a été réalisé sur la passe de typage. En effet, cette passe doit vérifier que les objets sont typement compatibles et ce de manière récusirve.

Pour la passe de code, l'utilisation des instruction machine permet l'implentation de manière assez intuitive.

Incrémentation

Cette fonctionnalité est très simple à implémenter, on remplace simplement toutes les instances de `a += b` par `a = a + b`. On a ainsi dans `parler.mly`

```
| aff=a PLUS EQUAL exp=e PV {Affectation (aff, Binaire (Plus, Affectable(aff), exp))}
```

Nous avons fait se choix d'impletation car il n'apport aucune modification à l'AST. En effet, ajouter de l'incrémentation comme une instruction impliquerait de modifier toutes les passes.

Type nommé

Cette fonctionnalité permet de créer des alias de types. Pour l'implémenter dans notre compilateur nous rajoutons: - `TIdent of string` dans `typ` de `type.ml` - `type typedef = TypeDef of string * typ` dans AstSyntax de `ast.ml` - `LocalTypeDef of typedef` dans AstSyntax.instruction de `ast.ml`

Il n'est pas nécéssaire d'ajouter des fonctionnalités dans les autres ASTs puisque lors de passe de l'AstSyntax à l'AstTds, nous remplaçons dans les informations les alias par leurs vrais types. En effet, lors de la definition d'un type nommé, nous l'ajoutons a la TDS comme étant un type nommé. Puis lors de la lecture d'un type, nous cherchons le type réel de ce type (avec la fonction `recherche_type`) : si c'est un "vrai" type, alors on le revoie le type en question, sinon nous cherchons dans la TDS le type correpondant au nom du type demandé.

Structure

Pour implémenter cette fonctionnalité nous avons rajouté: - `Struct of (typ * string) list` dans `typ` de `type.ml` - `Tuple of expression list` dans l'ast.

Nous nous servons de `Tuple` lors de l'assignation ou de l'initialisation de variables structs. Peu de choses changent réellement dans le code, on effectue simplement les opérations habituelles de chaques passes avec un itérateur.

La plus grande différence provient de la passe CodeRatToTam et à la passe placement. En effet, un tuple/struct a comme taille la somme des tailles de tous ses éléments. De même pour la génération de code RAT, nous empilons tous les éléments de la structures les uns à la suite des autres (de manière contigu).

Pour récupérer un élément d'un tuple, on récupère le placement de ce tuple puis on ajoute la taille des éléments précédent pour savoir ou se situe l'élement en mémoire. Nous pouvons récupérer la taille de chaque élément grace au type de la variable. Nous pouvons donc tiré la taille des élément avant et même la taille d'un élément précis.

Tests

L'ensembles des tests passent pour notre compilateur. De même l'ensemble des contrats de toutes les fonctions des passes ont été rédigés, et des commentaires ont été introduits dans les sections les plus difficiles à comprendre.

Jugements de typages

Pointeurs

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

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

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

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

Incrémentation

Puisque nous remplaçons le `+=` par une assignation combinée à l'opérateur binaire `+`, le jugement de typage est le même que ceux dans le cours page 101. Dans notre cas on obtient:

$$\frac{\sigma \vdash a : \text{int} \quad \sigma \vdash e : \text{int}}{\sigma \vdash a \mathrel{+}= e : \text{int}}$$

Type nommé

Puisque nous remplaçons lors de l'étape de la passe AstTds les alias par leurs vrais type, les jugements de typage vus en cours s'appliquent directement.

Enregistrements

Lors de l'appel d'un attribut d'un affectable, il faut que cet affectable soit un type struct et que affectable ai bien un attribut du nom demandé. Ainsi, sont type est le type de l'objet du nom demandé dans le type de l'affectable.

$$\frac{\sigma \vdash a : \text{struct}\{\dots,\ n : \tau,\ \dots\}}{\sigma \vdash a.n : \tau}$$

Ensuite il faut vérifier lors d'une initialisation ou d'une affectation que les types de chaques tuples sont deux à deux compatibles. De même cela implique que la cardinalité des deux tuples est égale.

$$\frac{\sigma \vdash a : \text{struct}\{\dots,\ s_n : \tau_1,\ \dots\} \quad \sigma \vdash e : \text{struct}\{\dots,\ s_n : \tau_2,\ \dots\} \quad \tau_1 = \tau_2\ \forall n}{\sigma \vdash a = e}$$

Conclusion

Dfficultés rencontrées

L'implentation des type nommés et de l'instruction assignation-addition a été assez intuitive et rapide à faire. En revanche, pour l'ajout des pointeurs, bien que nous l'ayons vu en TD, il a été assez difficile de comprendre et mettre en place l'utisation d'affectables. En effet, l'utilsation de `dref` étant récursive, elle fut subtile à implenter.

Bien que nous étions déjà assez familier avec les affectables, l'apparition des enregistrements nous a insité à créer beaucoup de fonctions intermédiaires en vu de la complexité de la structure du type `struct`.

Améliorations éventuelles

Comme nous l'avons fait avec les `print`, afin d'améliorer le nombre d'opérateurs disponibles à l'utilisateur sans qu'il y ait à créer des fonctions supplémentaire, nous pourrions ajouter l'opérateur `=` pour les rationels ou les enregistrement, ainsi que généraliser l'opérateur `+` pour les rationels.