Rapport de projet Sémantique et Traduction des langages

Damien Hostettler, Simon Maurel, Qi Chen et Vicky Dincher

17 Juin 2016

Table des matières

Introduction

Le but de ce projet a été de réaliser un compilateur pour les langages μC et $\mu C\#$. Ce compilateur doit vérifier les erreurs détectables lors de la compilation (erreurs de types, variable non définies ...) et doit générer la traduction du programme compilé en langage **TAM**.
La réalisation de ce compilateur passe passe par la gestion de la table des symboles, des erreurs de type, ainsi que la génération de code

1 Construction de la table des symboles

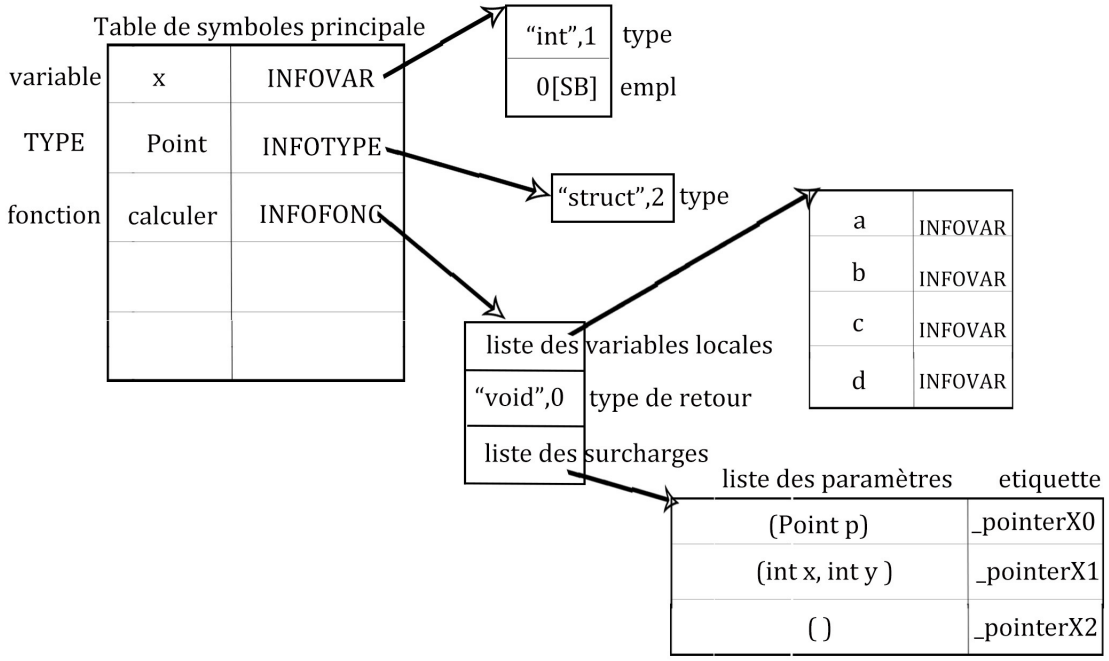
La table des symboles doit contenir toutes les informations sur ce qui est déclaré dans le programme (variables, types, fonction) sauf leur valeur en temps réel.
Une table des symboles est une liste d'élément de type INFO que l'on peut repérer par leur nom (nom de variable par exemple).

1.1 Contenu et hiérarchie

Nous avons donc modélisé notre table comme un $HashMap < String$,INFOVAR$>$. Ce sont les différents couples (Nom des variables (fonctions ...), informations liées).
On trouve ainsi plusieurs type d'informations (toutes héritées de la classe INFO) :

- Les INFOVAR liées au variables. Elles contiennent simplement le type de la variable, et son emplacement dans la pile.
- Les INFOTYPE liées aux types créés avec *typedef*. Elles contiennent un type (celui créé).
- Les INFOFONC, liées aux fonctions. Elles contiennent le type de retour de la fonction, la liste des différentes possibilités de paramètres pouvant être utilisés avec cette fonction (surcharges), ainsi qu'une TDS fille de la TDS courante, contenant les informations sur les variables (ou types) locales à la fonction.

On crée donc une TDS fille à chaque nouvelle fonction, mais également lorsque l'on rentre dans un nouveau bloc. On obtient ainsi la hiérarchie suivante :



1.2 Gestion des variables globales

Le compilateur implémenté permet l'utilisation de variables globales (déclarées tout au début du programme). Pour ce faire, l'emplacement des variables globales au programme (qui ne sont pas déclarées à l'intérieur d'un bloc) se situera dans le registre **SB**, et les variables locales seront déclarées dans le registre **LB**.

1.3 Ajouts pour le $\mu C\#$

Les notions de *namespace* et de *classe* ont été ajoutées dans le langage $\mu C\#$. Deux nouveaux types d'informations seront donc présents dans la table :

— les INFONAMESPACE, liées aux namespace. Elles contiennent la TDS des classes de ce namespace.
— les INFOCLASS liées aux classes. Elles contiennent un STRUCT de ses attributs, une liste des méthodes de la classe (sous forme d'INFOFONC).

2 Préconditions et gestions des erreurs de type

Pour la gestion des types et des erreurs de types, nous avons imposé certaines conditions, notamment pour la compatibilité des opérateurs. Dans la gestion des opérateurs, on prend en compte le type **bool**, pris en compte dans le langage $\mu C\#$.

2.1 Opérateurs et compatibilité de types

— Les opérateurs de comparaison $\leq$, $\geq$,<,>,+ (unaire), − (unaire), − (binaire), $*$, /, % ne sont compatible qu'avec les types **int**.
— Les opérateurs = et $\neq$ sont compatible avec les types **int** et **char**.
— L'opérateur + est compatible avec les types **int**, **string** et **char** .
— Les opérateurs $\vee, \wedge$, *not* sont compatibles avec les opérateurs **int** et **bool** .

Ainsi, afin de pouvoir gérer les types lors des opérations sur des variables, nous avons créé le type Opérateur, qui affecte à chaque numéro d'opérateurs (numérotés de 1 à 15) une liste des noms des types admis.
On peut ainsi tester la compatibilité d'un type pour un opérateur dans la classe DTYPE. On peut donc effectuer une opération à condition que les deux opérandes aient le même type, et que ce type soit compatible avec celui de l'opérateur. Sinon, on renvoie une erreur.

2.2 Types particuliers

Les types particuliers comme les POINTEURS et les STRUCTS ont été géré de la même manière qu'en TP.

3 Fonctions et leurs surcharges

Les fonctions ont été implémentées de manière à pouvoir les surcharger *i.e* : on peut appeler une fonction avec plusieurs listes de paramètres différentes, mais **le type de retour doit toujours rester le même**. Pour ce faire, nous avons implémenté les fonctions de la manière suivante :
Une INFOFONC contient une liste de SURCHARGES.
Une SURCHARGE est une liste d'INFOVAR ainsi que l'étiquette associée à la surcharge (ce sera dans notre cas *nom_fonctionXnum_surcharge*

Ainsi, lors de la déclaration d'une fonction, on réalise les actions suivantes :

```
INFO i = TDS.RechercherGlobalement(nom_fonction)
  Si i = null  alors
     Créer une nouvelle INFOFONC, avec la surcharge correspondante aux
     paramètres déclarés associée au numéro X0
```

Sinon

```
Vérifier que i est bien une INFOFONC
Ajouter la surcharge des paramètres déclarés aux autres surcharges,
associée à un nouveau numéro
```

Fin si

Ainsi, lors de l'appel, on génère une liste ordonnée de types (correspondante aux paramètres avec lesquels la fonction est appelée), et on la compare avec chacune des surcharges de la fonction (dont l'info est obtenue grâce au nom).
Si l'une des surcharges correspond, on appelle la fonction avec cette surcharge.

4 Génération de code

La génération de code a été gérée de la même façon qu'en TP, mais quelques ajouts ont été faits.

4.1 Opérateurs

Afin de réaliser les différentes opérations, nous avons créé la fonction *GenSubr* dans l'interface machine. Cette fonction prend un opérateur, et le type sur lequel il est appliqué, et envoie le code généré.
Par exemple, pour l'addition de deux entiers, le résultat sera "tSUBR IAdd ".

4.2 Appel de fonctions

Lorsqu'une fonction est appelée, on obtient l'étiquette correspondante à la fonction à appeler en comparant les paramètres d'appels, et les paramètres déclarés, comme expliqué précédemment. L'étiquette est donc unique pour chaque surcharge de la fonction, et on peut ainsi génére le code de l'appel à la fonction avec la méthode *genCall*.

Conclusion