Projet TDL

Année 2024/2025

Florian Poli, Baptiste Desnouk

Sommaire

16 janvier 2025

1 Introduction

Le but du projet était d'ajouter des fonctionnalités au langage Rat, codé pendant les TP de TLD, ainsi que les nouvelles fonctionnalités demandées. Nous avons, pas à pas, ajouté les fonctionnalités suivantes : pointeurs, variables globales, variables statiques locales et paramètres par défaut.

À chaque passe, nous avons vérifié que les fonctionnalités précédentes restaient utilisables et nous avons produit des tests pour valider les cas unitaires, les exceptions levées, ainsi que des programmes complets pour tester l'interfonctionnalité des différentes fonctionnalités.

Dans la suite de ce rapport, nous décrivons, pour chaque fonctionnalité et chaque passe, les changements effectués dans le code et dans l'AST. Pour la passe de typage, nous notons également les jugements de typage.

2 Les pointeurs

L'objectif est d'ajouter dans notre code la notion de pointeurs, qui sont des variables stockant une adresse mémoire.

Dans l'`AstSyntax`, plusieurs modifications sont nécessaires pour intégrer cette fonctionnalité :

- Ajouter le type `pointeur`.
- Introduire le type `affectable`, défini comme soit une chaîne de caractères (`string` identifiant) et un déférencement d'un affectable.
- Ajouter un mécanisme pour référencer une adresse mémoire dans les expressions (`adresse`).
- Gérer l'utilisation d'un `affectable` :
  - En tant qu'expression (utilisation en tant que variable à doite d'un =).
  - En tant qu'instruction, pour leurs déclarations.

```
type typ = Bool | Int | Rat | Undefined | Pointeur of typ

type affectable =
    | Ident of string
    | Deref of affectable

type instruction =
    | Declaration of typ * string * exp
```

```
    | Affectation of affectable * exp

type expression =
    | Affectable of affectable
    | Adresse of string
    | New of typ
    | Null
```

2.1 Passe TDS

Nous voulons produire cette AstTds :

```
type affectable =
  | Ident of info_ast
  | Deref of affectable

type instruction =
  | Declaration of typ * info_ast * exp
  | Affectation of affectable * exp

type expression =
  | Affectable of affectable
  | Adresse of info_ast
  | New of typ
  | Null
```

Les enjeux de cette passe sont de savoir si on souhaite modifier une constante ou non, car maintenant, on ne sait pas directement si c'est une modification ou un accès. Ainsi, on utilise un bouleen dans l'analyse d'un affectable qui se nomme `modif` qui est vrai si on souhaite modifier la variable/constante, faux sinon. Ainsi, on peut lancer une erreur si on tente de modifier une variable.

2.2 Passe type

L'Ast Type que l'on modifie est :

```
type affectable =
  | Ident of Tds.info_ast
  | Deref of affectable * typ

type instruction =
  | Declaration of Tds.info_ast * expression
  | Affectation of affectable * expression
```

Les jugements de typage pour les pointeurs (vu en TD) ont été définis dans la passe de typage. Ils permettent de vérifier qu'un pointeur est correctement déclaré avec le type `pointeur` et que celui-ci pointe bien vers le type attendu.

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

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

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

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

Ainsi l'analyse de l'affectable renvoie le type comme pour une déclaration lorsque qu'il atteint l'identifiant afin de comparer les type dans l'analyse d'une instruction.

2.3 Passe placement

Dans cette passe, il faut juste ajouter que la taille d'un pointeur est de 1.

2.4 Passe code

Lorsqu'on analyse un affectable, on prend en compte si l'on modifie la variable/constante ou non, de la même manière que pour la déclaration ou l'accès à une variable auparavant.

Pour la gestion des pointeurs, nous utilisons une opération de `malloc`, qui permet de réserver un bloc de mémoire dans le tas. Une fois cette mémoire allouée, l'adresse de cette zone mémoire est stockée dans le pointeur.

Pour gérer les déréférencements, on récupère d'abord l'adresse associée au pointeur, puis on réalise soit un `loadi`, soit un `storei` en fonction de si l'on veut modifier la valeur à cette adresse ou seulement y accéder.

Pour gérer les accès aux adresses de variables, on utlise la fonction pour charger l'adresse : `loada`

3 Les variables globales

Les variables globales permettent de créer des variables en début de programme utilisable partout que ce soit dans une fonction ou dans le main.

```
type var = Var of typ * string  * expression
type programme = Programme of var list * fonction list * bloc
```

3.1 Passe TDS

L'AstTds ressemble à ceci :

```
type var = Var of tds.info ast  * expresssion
type programme = Programme of var list * fonction list * bloc
```

Les variables globales doivent être stockées dans une TDS, et nous avons fait le choix de les mettre dans la TDS mère créée au niveau de l'analyse du programme. Ainsi, l'analyse des variables statiques se fait comme une déclaration de variable.

3.2 Passe type

La structure de l'Ast reste la même qu'à la passe d'avant pour les variables globales

```
type var = Var of Tds.info_ast * expression
type programme = Programme of var list * fonction list * bloc
```

Pour cette passe, on s'assure que le type déclaré est le bon pour la déclaration de la variable comme on le fait lors d'une déclaration d'une variable normale.

3.3 Passe placement

La structure de l'AST placement change légèrement pour prendre en compte le déplacement qu'induiront les variables globales :

```
type var = Var of tds.info ast  * expression
type programme = Programme of (var list*int) * fonction list * bloc
```

Afin de placer les variables globales, nous avons décidé de les placer en tout début du registre SB ainsi ce sont les variables déclarées en première puisqu'elles peuvent être utilisées par l'ensemble du programme et des fonctions.

Nous avons donc ajouté une fonction d'analyse de ces variables globales qui calcule leurs déplacements. Ainsi, on donne ensuite le déplacement total induit par les variables globales à l'analyse du bloc main en tant que point de départ.

3.4 Passe code

Enfin, avec les déplacements stockés dans les infos des variables globales, on peut stocker leurs valeurs dans SB avant toute autre analyse. Ensuite, l'analyse pour générer le code se déroule comme d'habitude.

4 Les variables statiques locales

Les variables statiques locales sont des variables déclarées dans une fonction et qui gardent la valeur même après la fin d'un appel.

Elle est gérée comme une instruction d'un bloc, plus particulièrement d'une fonction et à la même forme qu'une déclaration dans l'ASTsyntax :

```
type affectable =
    | Static of typ * string * expression
```

4.1 Passe tds

La sortie de cette passe doit avoir la même forme qu'une déclaration également afin de stocker les informations sur la variable statique :

```
type affectable =
    | Static of typ * Tds.infoast * expression
```

Tout d'abord, on vérifie que cette instruction est bien effectuée dans un bloc associé à une fonction (comme on le faisait déjà pour le retour). Puis, on la gère comme une variable locale de la fonction, c'est-à-dire que l'on vérifie la double déclaration et, sinon, on l'ajoute à la TDS locale.

L'expression associée à une variable statique ne peut utiliser que des littéraux ou des variables globales puisqu'on l'analyse dans la tds racine. (Cela est à peu près comme en C juste, on accepte les variables globales en plus, mais pas les constantes)

4.2 Passe type

La sortie de cette passe ajoute l'instruction Statique :

```
type affectable =
    | Static of Tds.infoast * expression
```

Pour cette passe, on gère la variable statique locale comme une déclaration c'est-à-dire que l'on vérifie que le type de la déclaration est compatible avec le type de l'expression.

$$\frac{\sigma\vdash\text{Type}:\tau \quad \sigma\vdash E:\tau}{\sigma,\tau\vdash\text{static Type id} = \text{E:void},[id,\tau]}$$

4.3 Passe Placement

Dans cette passe, on ajoute également la gestion de la variable statique et l'ASTplacement du programme à qui on ajoute la liste des instructions des variables statiques :

```
type affectable =
    | Static of Tds.infoast * expression
type programme =  Programme of (var list * int ) * (fonction list) * bloc * instruction
```

Cependant, dans cette passe, nous avions pensé au départ à une solution de changer la forme de la tds en ajoutant des infos dans l'info d'une fonction pour prendre en compte les variables statiques.

Mais nous avons finalement choisi de s'occuper des instructions d'un bloc, d'une fonction séparément afin de pouvoir effectuer une autre gestion. Si l'instruction n'est pas une variable statique, on a juste à appeler l'analyse d'instruction déjà codée.

On place alors, comme les variables globales, les variables statiques tout en haut de la pile principale, juste après les variables globales (Puisqu'on connait leur valeur avant le 1er appel de fonction). Ainsi, on doit calculer le déplacement total généré par les variables statiques locales et le faire remonter pour faire commencer le placement du bloc main au déplacement généré par les variables statiques locales et globales.

On fait également remonter la liste des infos des variables statiques dans le programme.

4.4 Passe code

Pour cette passe, nous traitons les variables statics comme les variables globales.

4.5 Amélioration

Comme nous initialisons les variables statiques au début du programme, lorsque l'on initialise une variable statique, l'expression associée ne peut prendre comme valeur qu'un litéral ou une variable globale.

Cependant, nous pensons qu'il peut y avoir une amélioration en autorisant seulement les constantes et les littéraux dans l'initialisation des variables statiques locales comme cela est fait en langage C.

5 Les paramètres par défaut

Pour implémenter cette fonctionnalité, il a fallu modifier l'AST syntaxe, le parseur et la passe de gestion des identifiants.

5.1 Le parseur

On modifie la liste des paramètres d'une fonction pour pouvoir y mettre une valeur par défaut :

```
fonc : t=typ n=ID PO lp=separated_list(VIRG,param) PF li=bloc {Fonction(t,n,lp,li)}

param : t=typ n=ID  d1=option(d)  {(t,n,d1)}

d : EQUAL e1=e  {Defaut e1}
```

L'option sur `d` permet de savoir s'il y a un paramètre par défaut ou non par la suite.

5.2 L'AST syntaxe

On modifie la structure des listes de paramètres pour coller avec le parseur :

```
type defaut = Defaut of expression
(* Structure des fonctions de Rat *)
(* type de retour - nom - liste des paramètres (association type et nom) - corps de la f
type fonction = Fonction of typ * string * (typ * string * defaut option) list * bloc
```

On a donc maintenant, en plus du type et du nom des paramètres, leurs valeurs par défaut si elles existent (`None` si pas de variable par défaut et `Some _` sinon).

5.3 Passe TDS (Gestion des identifiants)

Le principe pour traiter les paramètres par défaut est simple : on analyse les paramètres par défaut lorsqu'on parcourt les fonctions pour connaître la valeur des paramètres par défaut, afin de remonter ces valeurs dans les appels de fonctions et compléter, si nécessaire, les arguments de la fonction avec les paramètres par défaut (La fonction `completer_arguments` se charge de compléter ces arguments).

Ce "remontage" est réalisé à l'aide d'une table de hachage qui associe, à chaque nom de fonction, sa liste de paramètres par défaut. Cette table est complétée au cours de l'analyse d'une

fonction et est utilisée lorsqu'on rencontre un appel de fonction pour compléter les arguments manquants avec leurs valeurs par défaut.

Ainsi, on se balade dans les analyses avec la table de hachage et la TDS racine pour remonter les valeurs par défaut au niveau de l'appel de fonction.

Pourquoi la TDS racine ? Les paramètres par défaut sont définis au niveau de la fonction et seules les variables globales devraient y être visibles. D'où l'utilisation de la TDS racine dans l'analyse des expressions des paramètres par défaut.

5.4 Et les autres passes ?

Aucune modification n'est nécessaire puisqu'une fois l'appel de fonction complété. Cela revient au cas en l'absence de paramètres par défaut. Par exemple, si un appel n'a pas assez d'arguments, même en complétant avec les paramètres par défaut, il manquera toujours au moins un argument.

6 Conclusion

Les difficultés que l'on pouvait avoir sur ce projet sont le fait qu'il n'existe pas qu'une seule solution pour satisfaire les exigences et parfois, nous n'avions pas la même idée. Les choix que nous faisions pouvaient aussi nous mener à des impossibilités de prendre en compte certaines exigences et donc parfois devoir changer complètement la façon de mettre en place la fonctionnalité.

Aussi, les fonctionnalités avaient souvent une part d'ambiguïté qui demandait une réflexion sur la mise en pratique, comme pour les variables statiques par exemple.

De plus, nous ne prenions pas en compte certains cas limites dans nos tests, ce qui menait à une reprise de la gestion de la fonctionnalité.

Cependant, finalement, nous avons réussi à implémenter ces fonctionnalités dans le langage Rat.