TOULOUSE INP N7

Rapport de projet: RAT

Téo Pisenti et Ewen Le Bihan

Traduction des langages et programmation fonctionnelle

Janvier 2024

Table des matières

1 Introduction

Le but des TPs était la réalisation d'un compilateur pour le langage RAT, compilant vers du code TAM (Triangle Abstract Machine), une machine à pile.

Le langage RAT décrit la définition de variables, de fonctions, de conditions (if/else) et de boucles tant-que (`white`) et les types entier, booléen et rationnels.

Le but du projet a été d'étendre le travail réalisé en TP par la spécification de nouvelles fonctionnalités :

- Les pointeurs
- Les tableaux
- Les boucles `for`
- Les goto (saut à un endroit arbitraire du code d'une fonction ou du bloc principal)

2 Implémentation des pointeurs

2.1 Syntaxe

On commence par modifier la grammaire du langage (cf Figure 1) : l'affectation à et la déclaration de variables ne se fait plus directement sur des identifiants mais peut être effectué sur des pointeurs en les déréférençant avec `*`.

On introduit donc un nouveau type de nœud dans l'AST, des *cibles d'affectation*, ou *assignment target* en anglais.

Ce type a deux constructeurs, un pour désigner un identifiant direct, (donc une variable), et un autre pour désigner un déréférencement d'une cible d'affectation (la nature récursive permet de déréférencer profondément un pointeur de pointeur de... à un ordre arbitraire).

On ajoute aussi un nouveau constructeur au type `typ`, définissant les types du langage : il faut pouvoir déclarer que des variables, des paramètres ou le retour d'une fonction sont des pointeurs sur un certain type de donnée. Là encore, le caractère récursif du constructeur permettra de définir de pointeurs d'ordre supérieur.

Enfin, il faut rajouter quelques expressions :

- Instanciation de nouveaux pointeurs (avec `new`)
- Opérateur pour prendre l'adresse d'une variable (on n'autorise pas de prendre l'adresse d'une variable déréférencée, par exemple `&*a`) (avec `&`)
- Utilisation de variables déréférencées (avec `*`)
- Une valeur spéciale `null` représentant un pointeur vide (avec `null`)

Ceci demande donc l'ajout de trois nouveaux tokens, `new`, `null` (des mots-clés) et `&`.

```
  I ->  TYPE id = E;
-       | id = E;
+       | A = E;
...
// definition des assignment targets
+ A -> id | (* A)
...
// definition de la syntaxe pour les types pointeurs
  TYPE -> bool
        | int
        | rat
+       | TYPE *
...
//definition de new, & et *
  E ->  id ( CP )
        | [ E / E ]
        | num E
        | denom E
-       | id
+       | A
...
+       | null
+       | (new TYPE)
+       | & id
...
```

FIGURE 1 – Modification des règles EBNF pour définir la grammaire des relative aux pointeurs

2.2 Jugement de types

On étend les mécanismes déjà existants de vérification de la cohérence des types entre :

- les paramètres formels et effectifs,
- les `return` et le type de retour déclaré de la fonction), et
- dans les déclarations de variables.

Pour cela, on rend la fonction `estCompatible t1 t2` récursive, et on rajoute le cas des Pointeurs, en comparant simplement si les types intérieurs des pointeurs sont compatibles.

On a le cas particulier du pointeur null, qui est un autre constructeur de type `typ`, et qui a la particularité d'être compatible avec n'importe quelle type pointeur, peut importe le type intérieur de celui-ci.

$$\frac{\sigma \vdash v : \text{Pointeur}(\tau)}{\sigma \vdash \&\, v : \tau} \qquad \frac{\sigma \vdash e : \tau}{\sigma \vdash *e : \text{Pointeur}(\tau)}$$

2.3 Placement mémoire

On étend la fonction calculant la taille des types avec le cas des pointeurs. Un pointeur prend toujours une place de 1 (on stocke une adresse, qui est un entier).

2.4 Génération de code

On utilise l'instruction `LOADA` de TAM pour charger une valeur dans la pile se trouvant à une adresse déjà chargée dans la pile.

3 Tableaux

3.1 Syntaxe

Pour la définition des tableaux, il y a plusieurs éléments de langage à définir :

- Les types tableau (par ex. `int[]`)
- Les littéraux de tableaux (pour décrire un tableau en fournissant explicitement ses éléments, par ex. `{1, 2, 3, 4}`)
- L'indexation de tableaux (permettant de récupérer ou de modifier un élément en particulier, par ex. `a[0]`)
- L'initialisation de tableaux (pour initialiser un tableau vide en précisant sa capacité en nombre d'éléments, par ex. `new int[10]`)

On modifie donc la grammaire, cf Figure 2

```
  A -> id
    | (* A)
// indexation
+   | ( A[E] )
...
  TYPE -> bool
        | int
        | rat
        | TYPE*
// types
+       | TYPE[]
...
  E -> id ( CP )
    | [ E / E ]
...
// initialisation
+   | (new TYPE[E])
// litteraux
+   | { CP }
```

FIGURE 2 – Ajout des règles EBNF relative aux tableaux

3.2 Jugement de types

Types Extension de la fonction regardant la compatibilité de deux types, en regardant la compatibilité des types des éléments

Littéraux Vérification de l'homogénéité des types des éléments déclarés

Indexation On vérifie que l'expression servant d'index est bien de type Entier. On pourrait considérer que le nombre d'éléments fait partie du type du tableau pour mettre en place une vérification et éviter les indexations en dehors du tableau (si `index` $\geq$ card(`tableau`) ou `index` < 0)

Initialisation On vérifie que l'expression donnant la taille du tableau est bien un entier.

Soit τ un type.

$$\frac{\sigma \vdash e_1 : \tau \qquad (e_1, \tau) :: \sigma \vdash e_2 : \tau \qquad \cdots \qquad (e_1, \tau) :: \sigma \vdash e_n : \tau}{\sigma \vdash \{e_1, e_2, \ldots, e_n\} : \text{Tableau}(\tau)}$$

$$\frac{\sigma \vdash v : \text{Tableau}(\tau) \qquad \sigma \vdash i : \text{Int}}{\sigma \vdash (v[i]) : \tau}$$

$$\frac{\sigma \vdash e : \text{Entier}}{\sigma \vdash \text{new}\, \tau[e] : \text{Tableau}(\tau)}$$

3.3 Placement mémoire

On étend la fonction calculant la taille en même d'un type, en ajoutant le cas des tableaux :

$$|T[n]| = n|T|$$

3.4 Génération de code

Pour les littéraux, on génère le code en chargeant dans l'ordre chacune des expressions.

Cependant, il faut mettre dans la pile l'adresse vers la base du tableau à la fin, pour que l'on puisse utiliser ce tableau dans des affectations et autres. (En effet, un tableau est un pointeur sur son premier élément)

On utilise `malloc 0` pour charger l'adresse actuelle dans la pile.

4 Boucle `for`

4.1 Syntaxe

On ajoute ensuite les boucles `for`. Une boucle `for` comporte 4 parties :

Initialisation On initialialise une variable

Test On fourni une expression a vérifier pour savoir si l'on continue à effectuer la boucle.

Incrémentation On change la valeur de la variable par celle d'une expression calculée à chaque fin de tour de boucle. C'est souvent une incrémentation de la variable (qui agit alors comme un compteur).

Corps Les instructions à effectuer à chaque tour de boucle.

La syntaxe ressemble à ceci :

```
for (INITIALISATION; TEST; INCREMENTATION) {
    CORPS
}
```

On rajoute donc une nouvelle variante dans l'union de la règle de production des instructions :

```
  I -> TYPE id = E;
    | A = E;
...
+   | for (int id = E; E; id = E) BLOC
...
```

On préfère ceci à utiliser `I` pour `INITIALISATION` (resp. `INCRÉMENTATION`) pour limiter le travail à effectuer dans la passe suivante (il faudrait vérifier que l'instructions est bien une déclaration (resp. affectation)).

4.2 Gestion des identifiants

Cependant, les passes suivantes bénéficieront d'un nœud plus simple, contenant simplement quatres nœuds. On transforme donc, dès la première phase, `int id = E` et `id = E`. On en profite pour vérifier que ces deux identifiants ont bien le même nom.

On crée aussi une TDS fille intermédiaire pour stocker la variable initialisée par la boucle for. L'injecter dans la TDS qui sera créée par le corps de la boucle est possible mais complexe car cela demande de faire attention à ne pas ré-initialiser la variable à chaque tour de boucle. De plus, c'est complexe architecturellement pour le code.

4.3 Jugement de types

Initialisation On vérifie que la variable initialisée est bien de type Entier.

Test On vérifie que l'expression est bien de type Booléen.

Incrémentation Rien de spécial à implémenter (vu que le jugement sera simplement effectué par l'analyse de l'affectation)

Corps Rien de spécial non plus.

$$\frac{\sigma \vdash e_1 : \text{Int} \qquad (i, \text{Int}) :: \sigma \vdash e_2 : \text{Bool} \qquad (i, \text{Int}) :: \sigma \vdash e_3 : \text{Int}}{\sigma \vdash \text{for}(\text{int } i = e_1; e_2; i = e_3)\ \{\cdots\}}$$

4.4 Génération de code

On effectue l'initialisation, on marque le début et la fin de la boucle avec des étiquettes, on effectue l'incrémentation après les instructions, on évalue l'expression du test et on saute au début de la boucle si le test s'évalue à vrai.

5 Étiquettes et `goto`

5.1 Syntaxe

L'instruction `goto` permet de sauter à un autre endroit du code. Il faut donc non seulement définir la syntaxe de cette instruction, mais également définir comment marquer cet autre endroit du code, en lui donnant un nom.

On rajoute donc deux productions : l'instruction `goto`, et une instruction spéciale "étiquette" qui comporte le nom de l'étiquette à définir et l'instruction à marquer avec celle-ci. Les modifications sont décrites figure 3.

```
  I -> TYPE id = E;
    | A = E;
...
+   | goto id;
+   | id : I
```

FIGURE 3 – Modifications des règles EBNF relatives à `goto`

5.2 Gestion des identifiants

On stocke les étiquettes dans une TDS séparée pour éviter les collisions.

Il y a une difficultée dans le fait qu'une étiquette peut être analysée dans un nœud Goto avant d'avoir analysé sa déclaration. Il faut donc stocker deux types d'étiquettes dans la TDS : les utilisations d'étiquettes, et les déclarations.

Quand on rencontre une déclaration d'étiquette, on vérifie qu'elle n'existe pas déjà pour éviter les doubles déclarations. Si il existe une utilisation d'étiquette portant le même nom, on remplace celle-ci par une déclaration d'étiquette, afin de marquer que l'étiquette est bien définie.

Enfin, si l'on essaie d'utiliser `goto` avec une étiquette non définie, la passe de génération de code ne trouvera pas l'étiquette car elle ne cherche que dans les `info_ast` de constructeur `InfoLabelDef` (définitions des étiquettes), et non les `InfoLabelUse` (utilisation des étiquettes). Une solution plus propre serait de vérifier que chaque `InfoLabelUse` possède un `InfoLabelDef` correspondant, après analyse des identifiants de chaque fonction (et du programme principal).

Difficultées rencontrées Il est impossible d'utiliser les opérateurs standards d'OCaml `ref` et `!` en dehors du fichier `tds.ml`, car celui-ci définit les types `info` et `info_ast` qui opacifie le fait que `info_ast` soit une `ref`. Il faut donc définir une fonction *dans tds.ml* qui modifiera le constructeur de l'étiquette pour passer celle-ci d'une utilisation à une définition, quand on rencontre une instruction avec étiquette *après* avoir recontré son utilisation par un goto.

5.3 Génération de code

Il suffit d'utiliser les instructions TAM `JUMP` et les étiquettes, fonctionnalité native à TAM.

Il faut cependant penser à namespacer les noms des étiquettes avec un préfixe pour éviter tout conflit avec des étiquettes internes générées par, par exemple, les boucles.

5.4 Jugement de typages

Les étiquettes n'étant pas des valeurs, elles n'ont pas de types. Il n'y a donc rien à juger.

6 Annexes

6.1 Grammaire complète

Le non-terminal servant de point d'entrée au parser est ⟨début⟩.

6.1.1 Terminaux spéciaux

EOF fin du fichier

LT caractère <

Λ chaîne vide

'*' caractère * (le * seul servant à dénoter la répétition)

⟨*début*⟩ : := ⟨*programme*⟩ EOF

⟨*programme*⟩ : := ⟨*fonction*⟩* id ⟨*bloc*⟩

⟨*fonction*⟩ : := ⟨*type*⟩ id (⟨*paramètres*⟩) ⟨*bloc*⟩

⟨*bloc*⟩ : := { ⟨*instruction*⟩* }

⟨*instruction*⟩ : := ⟨*type*⟩ id = ⟨*expression*⟩ ;
| ⟨*cible*⟩ = ⟨*expression*⟩ ;
| const id = ⟨*expression*⟩ ;
| print ⟨*expression*⟩ ;
| if ⟨*expression*⟩ ⟨*bloc*⟩ else ⟨*bloc*⟩
| while ⟨*expression*⟩ ⟨*bloc*⟩
| return ⟨*expression*⟩ ;
| for (int id = ⟨*expression*⟩ ; ⟨*expression*⟩ ; id = ⟨*expression*⟩) ⟨*bloc*⟩
| goto id ;
| id : ⟨*instruction*⟩

⟨*cible*⟩ : := id
| ('⋆' ⟨*cible*⟩)
| (⟨*cible*⟩ [⟨*expression*⟩])

⟨*paramètres*⟩ : := ⟨*type*⟩ id
| ⟨*type*⟩ id, ⟨*paramètres*⟩
| Λ

⟨*type*⟩ : := bool
| int
| rat
| ⟨*type*⟩ '⋆'
| ⟨*type*⟩ [ ]

⟨*expression*⟩ : := id (⟨*liste d'expressions*⟩)
| [⟨*expression*⟩ / ⟨*expression*⟩]
| num ⟨*expression*⟩
| denom ⟨*expression*⟩
| ⟨*cible*⟩
| & id
| true
| false
| null
| entier
| (⟨*expression*⟩ + ⟨*expression*⟩)
| (⟨*expression*⟩ * ⟨*expression*⟩)
| (⟨*expression*⟩ = ⟨*expression*⟩)
| (⟨*expression*⟩ LT ⟨*expression*⟩)
| (⟨*expression*⟩)
| (new ⟨*type*⟩)

| (new ⟨*type*⟩ [⟨*expression*⟩])
| { ⟨*liste d'expressions*⟩ }

⟨*liste d'expressions*⟩ ::= ⟨*expression*⟩
| ⟨*expression*⟩, ⟨*liste d'expressions*⟩
| Λ