Traduction des Langages

Quentin Fraty
Nathan Maillet

1 Introduction

Dans la continuité des cours / TDs / TPs effectués en rapport avec le concept de Traduction des Langages, un projet de développement de compilateur a été réalisé. Le langage d'origine choisi fut le Rat, car étant largement limité en termes d'opérations réalisables dans un programme.
Le langage cible, quand à lui était un pseudo langage assembleur pouvant être exécuté directement à travers un programme spécialisé.

Comme nous allons le voir tout au long de ce rapport, une grande variété de fonctionnalités ont été ajoutées au langage Rat, avec comme objectif de le rendre plus complet, et plus proche des langages bas niveau manipulés aujourd'hui (Rust, C).
Notamment, voici les différentes fonctionnalités implémentées et opérationnelles, qui seront abordées :

- **Ajout des pointeurs, et accès par référence à des variables.**
- **Ajout du côté optionel au bloc else d'une conditionnelle.**
- **Ajout des expressions ternaires.**
- **Ajout des boucles 'à la Rust'.**

Voici ensuite les fonctionnalités non demandées implémentées :

- **Ajout de la surcharge de fonctions (même nom, différents types de paramètres).**
- **Ajout d'instructions pour l'incrémentation rapide (v++, ++v, v+=42).**
- **Ajout d'une backtrace en cas d'erreur.**

2 Mutation de *tds.ml* en *mtds*

La première étape avant de se lancer sur le projet était de modifier *tds.ml* afin de ne plus le modifier, même si par la suite nous voulions modifier notre raisonnement, comme par exemple ce qui est arrivé sur les pointeurs (cf 3.Pointeurs).

***mtds.ml* a donc pour but de généraliser *tds.ml*. Nous sommes donc passés d'Identifiants *string* en un type paramétré *'a*, ce qui explicite l'aspect monadique de la tds sans pour contraindre le module, ce qui aurait été un apport négligeable en OCaml 4. Par ailleurs, dans le cas où nous voulions plus tard modifier notre langage rat il aurait été intéressant d'avoir parétrer *tds.ml* pour ajouter, par exemple, des casts de type ou si nous voulions donner la possibilité au codeur d'expliciter les kinds de ses types s'ils étaient implémentés.**

3 Pointeurs

Le choix lié au traitement des pointeurs revient a se demander comment nous considérons les mots *int * a*. Alors que nous pouvons considérer une variable de type *Pointeur(int)* et d'indentifiant *a*, il nous semblait plus famillier de voir ça comme une variable de type *int* et d'identifiant *Pointeur(a)*.

Cela nous a mené vers une première tentative qui était devenue trop rigide pour les manipulations que nous voulions faire avec les pointeurs. C'est alors que nous avons changé pour une troisième façon de voir les choses : c'est tout simplement une variable avec un type *int*, une *marque Pointeur(Neant)* et un symbole *a*, la *marque* et le symbole formant l'identifiant.

Une marque représente alors le niveau de pointeurs et n'est ni lié au type ni au symbole. Avec cette représentation, nous avons alors pu nous lancer sereinement dans l'implémentation des pointeurs, avec une modification de tous nos jugements de typages suivant celui ci-dessous :

$\frac{\sigma \vdash id : \tau}{\sigma \vdash *id : \tau, \pi}$ où π est la marque.

Par suite, τ_r est l'environnement de type et de marque.
Les jugements de typages sont alors inchangés, simplement on peut remplacer pour les variables id par *id en suivant le typage ci-dessus.

Pour les ASTs, seul les ajouts lexicaux ont été inclus sans nouvelle modification.

On peut noter que dans l'implémentation des pointeurs que nous avons fait, il est possible d'affecter un pointeur à la déclaration. Par exemple, est possible :

```
int a = 5;
int *b = &a;
```

4 Bloc else optionnel

Grâce à la façon choisie initialement pour traiter les blocs conditionnels, il fut assez simple d'implémenter cette fonctionnalité. En effet, lors du parsing, il suffit de créer un *AstSyntax.Conditionnelle* avec un bloc else vide, et de le traiter comme un bloc conditionnel classique. Le typage est inchangé, et le code produit est le même que si le bloc else était présent.

Cette solution permet de minimiser la redondance de code, et de ne pas avoir à traiter le bloc else comme un cas particulier, avec un ajout d'un else dans la passe de génération de code dans tous les cas. Cela n'a aucun impact sur l'exécution du programme.

Au niveau du jugement de typage, dans la mesure où le bloc else est vide, celui-ci peut être résumé comme suit :

$$\frac{\sigma \vdash E : \text{bool} \quad \sigma, \tau_r \vdash \text{BLOC} : \text{void}}{\sigma, \tau_r \vdash \text{if } E \text{ BLOC} : \text{void}, []}$$

5 Conditionnelle ternaire

Cette fonctionnalité ajoutée au langage Rat a nécessité de créer un nouveau type d'expression, *AstSyntax.Ternaire*. Tout d'abord, il a fallu modifier la grammaire pour accepter cette nouvelle expression.

Au niveau du lexer, il a fallu ajouter le symbole `?` et `:` dans la liste des symboles à accepter. Au niveau du parser, il a fallu ajouter une règle pour accepter cette nouvelle expression :

```
| PO e1=e QMARK e2=e COLON e3=e PF  {Ternaire (e1,e2,e3)}
```

Ensuite, les différentes passes du compilateur ont successivement vérifié les choses suivantes :

- **La conformité des identifiants utilisés dans les trois expressions de la conditionnelle ternaire.**
- **Le fait que la premiere expression soit bien booléenne, et que les deux autres soient de même type. Cela peut être résumé par le jugement de typage suivant :**

$$\frac{\sigma \vdash E : \text{bool} \quad \sigma \vdash E_1 : \tau \quad \sigma \vdash E_2 : \tau}{\sigma \vdash (E \ ? \ E_1 : E_2) : \tau}$$

Enfin, la génération du code associé à un ternaire se rapproche à celui d'une instruction conditionnelle, dans la mesure où la valeur mise en sommet de pile est celle de l'une des deux expressions : l'utilisation de jump est alors nécessaire.

```
| AstType.Ternaire(e1, e2, e3) ->
      let labElse = getEtiquette ()
      and labEndIF = getEtiquette () in
      ast_to_tam_expression e1
      ^ jumpif 0 labElse
      ^ ast_to_tam_expression e2
      ^ jump labEndIF
      ^ label labElse
      ^ ast_to_tam_expression e3
      ^ label labEndIF
```

6 Loop à la Rust

Les loop à la rust sont une extension du langage Rat qui permet de créer des boucles infinies, avec la possibilité d'en sortir à l'aide du mot clé `break` et de terminer prématurément une itération à l'aide du mot clé `continue`.

TODO : grammaire

Comme nous allons le voir, toute la complexité des boucles à la rust est liée à la gestion des labels, qui est exclusivement gérée par la passe de gestion d'identifiants. Tout d'abord, il a été choisi de créer un nouveau type de données insérable dans la table des symboles.

En effet, il est nécessaire de stocker les informations liées à une boucle pour pouvoir la terminer, ou la recommencer lorsqu'un `break` ou `continue` est utilisé.
Cependant, et ce comme énoncé dans le sujet, il fallait donner la possibilité à l'utilisateur d'imbriquer des fonctions de même label.
Pour ce faire, ce nouvel élément de la TDS associe à un label une InfoBoucle, qui

contient en réalité une liste d'informations (couple de *string*) associées à ce label.
Cela permet de prendre en compte le cas où deux boucles de même label sont imbriquées, et de pouvoir les distinguer.

Un deuxième choix de conception que nous avons fait est sur les identifiants de boucles dans la TDS.
Tout d'abord, à toute boucle est associé un identifiant pour la TDS : soit celui donné par l'utilisateur, soit un identifiant uniqué généré par le compilateur de manière analogue à la génération de labels lors de la passe de génération de code.

```
let giveID =
  let num = ref 0 in
  fun () ->
    num := (!num)+1 ;
    "id"^((string_of_int (!num)))
```

Ensuite, l'information associée à une boucle est le couple (label de début, label de fin), qui sont générés par le compilateur à partir du label précédent.
Ainsi, lorsque deux boucles de même label sont utilisées, l'*infoboucle* associée au label est une liste de couples (label de début, label de fin), dont la tête est le couple associé à la plus imbriquée. En exploitant l'aspect récursif des fonctions, il est possible de récupérer le label de début/fin de la boucle la plus imbriquée facilement, ce qui est la logique choisie pour le `break` et le `continue`.

De plus, il fallait donner la possibilité à l'utilisateur de ne pas utiliser de labels pour les `loops`, `break` et `continue`. Le comportement choisi devait alors être que tout `break` / `continue` aurait un effet sur la boucle courante, et dans le cas d'imbrication de boucles, celle la plus imbriquée.
Pour pouvoir implémenter cela, il a été choisi de *décorer l'arbre* avec l'infoboucle associée à la boucle courante pour pouvoir associer au `break` / `continue` la bonne boucle.

Suite à cette passe, celle de typage est comparativement triviale, dans la mesure où celle-ci revient à une analyse du typage d'un bloc. Voici les jugements de typage associé :

$$\frac{\sigma, \tau_r \vdash \text{BLOC} : \text{void}}{\sigma, \tau_r \vdash \text{loop BLOC} : \text{void}, []} \text{ et } \frac{id :: \sigma, \tau_r \vdash \text{BLOC} : \text{void}}{\sigma, \tau_r \vdash \text{define id : loop BLOC} : \text{void}, []}$$

La passe de placement mémoire, de manière analogue, revient à analyser le placement mémoire des variables du bloc de la boucle.

En revanche, la passe de génération de code s'avérera poser des difficultés que nous n'avions pas anticipé. En effet, chaque itération de boucle doit écrire ses variables locales au même endroit, ce qui pose des problèmes lorsqu'un `break` ou un `continue` sont utilisés : alors que le TantQue implémenté en Rat assurait l'exécution entière de son bloc courant avant de se finir, ce n'est pas le cas de la boucle à la Rust.
Ainsi, en imaginant qu'après un `break` il y a une déclaration de variable (et donc un push de la pile), il s'avérait nécessaire d'avoir le déplacement dans la boucle jusqu'au `break` en tant que paramètre hérité (décoration d'arbre). Cependant même avec une telle chose, le cas d'imbrication de boucle ne semblait pas traitable facilement de manière pure, dans la mesure où un `break` peut aussi faire sortir d'une boucle moins imbriqué, comme dans l'exemple suivant :

```
main {
    int x = 3;
    define l : loop {
        x++;
        loop {
                int y = 5;
                break l;
                bool z = true;
        };
        rat r = [3/2];
    };
}
```

Ici, il faudrait se rappeler au niveau du `break` l que cela revient à pop 1, et pas 2 ou 3 qui serait obtenu si on prenait toutes les déclarations réalisées.
N'ayant pas de solution simple à ce problème et par manque de temps, il a été choisi de ne *pas* pop le contexte d'une boucle en cas de `break` ou `continue`. Ce choix, bien que semblant fonctionner, peut poser des problèmes de pile. En effet, si un *print* survient après la boucle, il sera en haut de ce qui n'a pas été nettoyer dans la pile. Dans le cas où nous sommes en haut de la pile, ce *print* va donc load dans le tas ou échoué, ce qui n'aurait pas eu lieu avec une pile mieux gérée.

7 Surcharge de fonctions

8 Incréments

Il a été choisi d'implémenter dans le langage Rat de nouvelles fonctionnalités qui puissent êtres traitées au niveau du parseur. C'est pourquoi l'incrémentation comme en C (variable++ et ++variable) a été partiellement ajoutée : contrairement à C, ces deux éléments sont considérés comme des instructions à part entière, ce qui limite leur intérét à celui d'un compteur. Voici les lignes associées dans le fichier Parser :

```
| PLUS PLUS n=r PV
{Affectation (n,Binaire (Plus,Identifiant n,Entier 1))}
| n=r PLUS PLUS PV
{Affectation (n,Binaire (Plus,Identifiant n,Entier 1))}
```

De manière similaire, l'opérateur '+=' a été ajouté au langage Rat, et ce grâce à une transformation directement dans le parseur :

```
| n=r PLUS EQUAL e1=e PV
{Affectation (n,Binaire (Plus,Identifiant n,e1))}
```

Ainsi, dans la mesure où ces instructions sont 'converties' en combinaisons d'éléments simples compris par notre compilateur, aucune passe n'a à être adaptée pour prendre en compte ces opérations.

Les jugements de typages liés à ces instructions peuvent alors être obtenu par déduction de ceux sur les affectations d'entiers :

$$\frac{\sigma \vdash id : \text{Int}}{\sigma \vdash \text{id}++ : \text{Int}} \text{ et } \frac{\sigma \vdash id : \text{Int}}{\sigma \vdash ++\text{id} : \text{Int}}$$

9 Affichage des erreurs

La génération de backtrace est la dernière fonctionnalité optionnelle que nous avons ajouté à notre compilateur Rat.
Celle-ci permet à un développeur de savoir précisément à quelle instruction de quel bloc se situe l'erreur empêchant la compilation. Voici par exemple la backtrace générée :

```
int fact (int i int n){
  int res = 0;
  bool pascompiler = true;
  if (i=n){
    res = i;
  } else {
    res = (( i * call fact ((i+1) n)) + pascompiler);
  }
  return res;
}

test{
  int x = call fact (1 5);
  print x;
}
```

```
===== ERROR =====
Contexte :
Instruction 0 : fact
Instruction 4 : bloc else
Erreur instruction 7
Type inattendu pour l'opérateur + : Int et Bool
===== END ERROR =====
Exception:
Rat.Exceptions.TypeBinaireInattendu (Rat.Ast.AstSyntax.Plus,
 (Rat.Type.Int, Rat.Type.Neant), (Rat.Type.Bool, Rat.Type.Neant)).
```

Ce résultat utilise le concept de décoration d'arbre, comme réalisé pour d'autres fonctionnalités : après la passe de gestion des identifants, à chaque instruction est rajouté un contexte. Il s'agit d'une liste de couple (numéro de ligne, nom du bloc) représentant avec précision la localisation de chaque instruction dans le code. Ainsi, toute erreur levée fait désormais appel à notre fonction 'afficher_erreur' prenant en paramètres un contexte,une exception et son numéro de ligne, et l'affichant comme montré précédemment.

10 Conclusion