

Traduction des Langages

Quentin FRATY
Nathan MAILLET

1 Introduction

Dans la continuité des cours / TDs / TP effectués en cours de Traduction des Langages, un projet de développement de compilateur a été réalisé. Le langage d'origine choisi fut le Rat, un langage très limité en nombre d'instructions différentes réalisables.

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 pour 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 autant contraindre le module, ce qui aurait été un apport négligeable en OCaml 4 pour le cadre dans lequel nous avons codé (pas de gestion très poussée des exceptions ou de foncteurs). Par ailleurs, dans le cas où nous voudrions plus tard modifier notre langage Rat il aurait été intéressant d'avoir paramétré *tds.ml* pour ajouter, par exemple, des casts de type s'ils étaient implémentés.

3 Pointeurs

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

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

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 les symboles `?` 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 première 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`.

Face aux difficultés que celle-ci a pu nous poser, il nous a semblé important de commenter nos choix effectués pour obtenir une grammaire cohérente. En effet, en reprenant la syntaxe du sujet pour les boucles nommées (`l : loopbloc`), des problèmes d'ambiguïté empêchaient Menhir de comprendre correctement la grammaire. Comme le montre la capture d'écran ci-après, l'erreur ne nous indiquait pas la source de l'ambiguïté. Mais même en analysant les logs de compilation, nous ne sommes pas parvenus à déterminer l'origine de ces conflits, empêchant un fonctionnement de notre compilateur :

```
/Documents/Annee_2/TDL/projet/rat_compiler)-(172K:29)
$ dune clean; dune build
Warning: 2 states have shift/reduce conflicts.
Warning: 2 shift/reduce conflicts were arbitrarily resolved.
```

Pour éviter cette erreur, nous avons fait le choix de modifier la grammaire d'une telle manière que les règles d'écriture d'une boucle ne pourraient plus être ambiguës pour Menhir. Ainsi, voici comment nous avons choisi qu'une boucle nommée devait s'écrire :

```
define l : loop { bloc };
```

Par l'ajout du mot-clé `define` en début de règle, et d'un point virgule en fin, l'ambiguïté est alors levée, et le compilateur fonctionne correctement.

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. Un choix a été fait de ne pas autoriser un même nom de variable et de boucle en même temps. 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 unique généré par le compilateur de manière analogue à la génération de labels lors de la passe de génération de code. Il est intéressant de noter que cette nomenclature est conflictuelle si le développeur nomme sa boucle `id` alors qu'il y a déjà une boucle sans `id` avant. Cela peut être réglé en vérifiant les identifiants de boucle pour passer à l'id suivant tant qu'il y a un conflit.

```
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 directe, car elle revient à une analyse du typage d'un bloc. Voici les jugements de typage associés :

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

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 dans la pile, 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'avérerait 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 imbriquée, 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 nettoyé dans la pile. Dans le cas où nous sommes en haut de la pile, ce *print* va donc `load` dans le tas ou échouer, ce qui n'aurait pas eu lieu avec une pile mieux gérée.

7 Surcharge de fonctions

Nous avons décidé d'implémenter en plus dans notre projet une notion évoquée lors de l'examen écrit de TDL : la surcharge de fonctions. Ce concept autorise l'existence dans un même programme de fonctions ayant le même nom, mais une signature différente au niveau des paramètres.

Pour pouvoir accepter la surcharge de fonctions, il a tout d'abord fallu modifier le type InfoFun, dans la mesure où l'on devait connaître toutes les différentes signatures qui pouvaient être associées à un nom. Ainsi, une InfoFun contient désormais une liste de signatures, chacune représentée sous la forme d'une liste de paramètres/retour.

Ce changement implique le fait de modifier l'analyse de gestion des identifiants des fonctions dans la passe associée : l'exception DoubleDéclaration ne doit désormais être levée que si l'on essaie de définir une fonction avec le même nom et les mêmes types de paramètres.

La dernière modification à effectuer sur le compilateur consiste en la résolution de surcharge lors de l'appel d'une fonction. Cela est réalisé dans la passe de Typage, où les types des paramètres utilisés sont comparés aux différentes signatures connues de la fonction appelée. Ainsi, la surcharge de fonction est belle et bien fonctionnelle.

8 Incréments

Il a été choisi d'implémenter dans le langage Rat de nouvelles fonctionnalités qui puissent être 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 et ne peut être utilisé dans une expression. 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 :

$$\frac{\sigma \vdash id : \text{Int}}{\sigma \vdash id ++ : \text{Int}}, \frac{\sigma \vdash id : \text{Int}}{\sigma \vdash ++ id : \text{Int}} \text{ et } \frac{\sigma \vdash id : \tau \quad E : \tau}{\sigma \vdash id + = E : \tau}$$

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 identifiants, à chaque instruction est rajouté un contexte. Il s'agit d'une liste de couple (numéro de l'instruction, 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

Différentes améliorations pourraient être apporté à ce compilateur. Au-delà du fait que nous ne traitons pas le cas de l'optimisation du code source, les difficultés mentionnées tout au long de ce rapport mettent en évidence les améliorations possibles suivantes :

- Les initiatives prises au niveau du parsing mettent en évidence un problème qui pourrait être corrigé.
- Les pointeurs ne sont effectifs que sur des variables et retours. Il peut pourtant être très intéressant d'avoir des pointeurs sur fonction qui impliqueraient une gestion du LB par le code assembleur mais permettrait de faire facilement des fonctions d'ordres "supérieurs" (dans l'absolu on gèrerait des pointeurs et non des fonctions).

De plus, et ce malgré les apports demandés par le sujet et nos initiatives personnelles, le langage Rat reste très limité. Voici les différentes fonctionnalités que nous aurions

désiré implémenter mais n'avons pas pu, faute de temps :

- Fonctions d'ordre supérieur
- Les modules
- Déclaration de fonctions auxiliaires

Enfin, et ce au niveau du compilateur en lui-même, il aurait été intéressant d'aborder l'optimisation de compilation, à travers par exemple la suppression de code mort (toute instruction après un retour dans un même bloc).