Traduction des Langages

Quentin Fraty
Nathan Maillet

1 Introduction

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

3 Pointeurs

4 Bloc else optionnel

Grâce à la façon choisie initialement pour traiter les blocs conditionnels, il fut relativement 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 comme seul bémol que la passe de génération de code rajoute dans tous les cas un label pour le else, qu'il soit vide ou non. 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 de les casser 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 insérables 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 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 réside dans le choix des informations stockées dans la TDS pour les boucles :
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 analoque 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. Grâce à la récursivité du langage fonctionnel, il est possible de récupérer le label de début/fin de la boucle la plus imbriquée, ce qui est la logique choisie pour le break et le continue.

7 Surcharge de fonctions

8 Incréments

9 Affichage des erreurs

10 Conclusion