OCalme
Outil de Compilation Avancé pour Langage Machine (mais En fait c'est du python)

Julien Marquet

31 mai 2019

Table des matières

1 Utilisation

Un exemple de programme est contenu dans `example.oklm`.
Essayez :

- `python3 main.py example.oklm` pour lancer le programme écrit dans `example.oklm`.
- `python3 main.py -h` pour obtenir de l'aide.
- `python3 main.py example.oklm --norun --ast` pour avoir une représentation lisible de l'AST généré sans exécuter le programme.
- `python3 main.py example.oklm --norun --ir` pour avoir une représentation lisible de l'IR généré sans exécuter le programme.
- `python3 main.py example.oklm --norun --gen` pour avoir le code généré sans exécuter le programme.

2 Structure d'un programme

Détaillons la structure d'un programme en nous fondant sur quelques exemples.

2.1 La fonction "main"

```
fn main: void -> void {
  print("Hello world !")
}
```

Le "Hello world!" en OCalme!
Ici, rien de bien surprenant. La fonction `main` est celle qui est appelée en première au moment du lancement du programme. Elle ne prend aucun argument, et ne renvoie rien, d'où son type `void -> void`. Le nom "`void`" est directement repris de `C`, et sert à signaler qu'une fonction ne prend pas d'arguments ou n'en renvoie pas.
La syntaxe des appels de fonctions n'est pas surprenante non plus, elle est calquée sur celle de `C` (et de nombreux autres langages).

Notons simplement le type de la fonction `print` : `String -> void`.

2.2 Variables

```
fn exemple_avec_des_variables: void -> void {
  let t = 0;
  t = t + 1;
  let (t, u) = {
    let v = t + 4;
    (2, 3*v)
  }
}
```

La syntaxe `let ... = ...` est explicite. On assigne au membre de gauche (la "lvalue") la valeur du membre de droite (la "rvalue").
Notons tout de même que :

- On peut assigner plusieurs variables en même temps.
  En fait, les "lvalues" peuvent être des variables, ou des n-uplets de lvalues : un n-uplet s'écrit, en Ocalme : `(<élément 1>, ..., <élement n>)` où les parenthèses sont obligatoires (on ne peut pas, comme on peut le faire en Python, écrire : `a, b = b, a`, mais il faut écrire : `(a, b) = (b, a)`).
- Les blocs de code *sont des rvalues*. La valeur d'un bloc est alors la valeur retournée par sa dernière expression, comme dans l'exemple.
  Ce comportement est copié sur celui du langage `Rust`.

2.3 Fonctions de plusieurs variables, typage des fonctions

```
fn plusieurs_variables:
  (a: Integer)
  -> (b: Integer)
  -> (Integer, Integer) {
```

```
  (b, a)
}
```

La syntaxe des fonctions de plusieurs variables est, cette fois-ci, un peu plus... créative.
On peut la voir comme un mélange de la syntaxe des types de `Caml` et de la convention d'écrire, lorsque la variable `v` est de type `T` : `v: T`.
Les différents couples `<variable>: <Type>` doivent être entourés de parenthèses (comme dans l'exemple), on sépare les différentes variables par des flèches `"->"`, et l'élément qui suit la dernière flèche est le type de retour de la fonction.
Ici, la valeur retournée par la fonction est un couple d'entiers, noté `(Integer, Integer)`.

2.4 Structures de contrôle

```
\paragraph{} Voici une fonction qui calcule la somme des n premiers
entiers naturels.

fn structures_de_controle: (n: Integer) -> Integer {
  let t = 0;
  let acc = 0;
  loop {
    if (t = n) then {
      break acc
    };
    t = t + 1;
    acc = acc + t
  }
}
```

Cette fonction utilise les deux structures de contrôle de OCalme.
Leurs principes sont clairs, mais notons que :

- Un `if` s'attend à recevoir un booléen. Ici, le test `(t = n)` a un type booléen. Le symbole `=` teste l'égalité des deux termes qui lui sont donnés.
- Une boucle `loop` peut renvoyer une valeur. Pour cela, il suffit de mettre une rvalue après `break`, comme dans l'exemple.
- Les structures de contrôle étant des rvalues, elles doivent être succédées d'un point-virgule lorsqu'elles n'occupent pas la dernière position dans un bloc.

2.5 La syntaxe "if unwrap" et les "box"

Voici, on l'attendait tous, la fonction `rev` :

```
fn rev: (li: List(GenericBox<T>))
  -> (base: List(GenericBox<T>))
  -> List(GenericBox<T>) {
  let acc = base;
  let top = li;

  loop {
    if unwrap top on (x, t) then {
      top = t;
```

```
    } else {
      break acc
    }
  }
}
```

Cette fonction utilise la syntaxe "`if unwrap`".

```
if unwrap <box> on <lvalue> then rvalue else rvalue
```

où `<box>` est un objet de type "Box". Ce est l'équivalent d'un pointeur. Une `box(T)` peut être vide (éqivalent du pointeur nul), ou remplie, et, lorsqu'elle est remplie, contient un objet du type `T` : les `box` sont des types génériques – même si la généricité n'est supportée que de façon très superficielle en OCalme.

Généricité OCalme permet d'introduire des types génériques, mais cette fonctionnalité n'a pas réellement été développée. On pourra se contenter de noter qu'un objet de type `GenericBox<T>` représente une `box` dont on ne connaît pas le type du contenu. La seule garantie que l'on a est que toutes les occurences de `GenericBox<T>` dans la signature d'une même fonction seront considérées du même type, et que les fonctions génériques "se comportent bien" du côté de la fonction appelante.

3 Données

Les types de donnés représentant l'AST et l'IR sont tous équipés d'une fonction `pretty` qui est faite pour générer des messages à la fois lisibles et reflétant les données contenues par la structure sur laquelle elle a été appelée.
Les options `--ast` et `--ir` de `main.py` utilisent cette fonction pour afficher les structures de données générées.
On peut se référer aux messages produits par `python3 main.py example.oklm` pour comprendre la signification des structures de données.

4 Algorithme

Le compilateur est séparé en plusieurs "étages" : On récupère le programme dans le fichier donné en argument (sous forme de chaîne de caractères), puis on effectue les étapes ci-dessous.

4.1 Parsing

C'est la construction de l'AST ("Abstract Syntax Tree").
Les fichiers contenant les algorithmes sont `parse_util.py`, `parseBlock.py`, `parseFunction.py`, `parseLRValue.py`, `PrecedenceBuilder.py` et `Tokenizer.py`.
Les définitions des types de données de l'AST sont dans `AST.py`.
Le principe général est qu'un fichier OCalme est structuré en :

- fonctions (`parseFunction.py`)
  représentées par leurs paramètres et un bloc de code :
  - blocs (`parseBlock.py`) contiennent des commandes (assignements, ou simplement des calculs).

— commandes : Les commandes sont composées d'éventuellement un symbole spécial (le symbole `=`, pour l'assignement), et de :
  — lvalues : Ce sont les valeurs qui peuvent se trouver à gauche dans une égalité, c'est à dire des variables ou des n-uplets.
  — rvalues : Les valeurs qui peuvent se trouver à droite d'un symbole `=`. Les variables et les n-uplets sont bien sûr des rvalues, mais aussi les structures `if`, `loop` et `if unwrap` ; et les blocs de code.

  Le fichier concerné est `parseLRValue`.
  L'algorithme contenu dans ce fichier est en fait capable de parser des lvalues *et* des rvalues. En fait, les lvalues sont des cas particuliers de rvalues (ce sont les rvalues uniquement composées de variables et de n-uplets de lvalues).

4.2 Flattening

Après que l'AST a été construit, on peut générer une représentation intermédiaire (`IR`, `Intermediate Representation`) qui est plus facile à manipuler par un programme (le but étant que la transpilation vers Python soit triviale).
On profite de cette étape pour vérifier que le programme est bien typé.

Remarque Il semble, d'après la taille du fichier `flattener.py` et la complexité des algorithmes qu'il contient, qu'il aurait été plus sage de réserver la vérification des types à une autre étape de la compilation...

4.3 Transpilation

On transforme l'IR en un programme Python. L'algorithme est dans `transpile.py`. Il est assez simple, et montre comment utiliser les structures de données générées par le Flattener.

5 Que lire ?

Le fichier `transpile.py` contient la source du transpilateur, qui transforme l'IR en du code Python. Il est assez court et montre bien comment interpréter l'IR.
On peut par exemple comparer le résultat de `python3 main.py example.oklm --norun --ir` et celut de `python3 main.py example.oklm --norun --gen`, sachant que la transformation réalisée est effectuée par `transpile.py` (à un détail près : les fonctions `wrapper` et `mkBoxId` sont toutes deux générées dans `OklmRunner.py`).

On peut aussi lire `OklmRunner.py` qui fait l'interface entre le parseur, le Flattener et le transpilateur.

Les fichiers `AST.py` et `FlatIR.py` contiennent les définitions des structures de données qui composent l'AST et l'IR, on peut les survoler pour voir ce que ces structures contiennent. Le plus intéressant est de comparer ces deux fichiers aux résultats de `python3 main.py example.oklm --norun --ast` et de `python3 main.py example.oklm --norun --ir`.