# Rapport projet de TDL

Maxence Ahlouche Martin Carton
Clément Hubin-Andrieu

4 mai 2014

## Table des matières

# 1 Introduction

Nous avons décider de compiler en *tam* et en langage intermédiaire *llvm*. De plus les interfaces devraient être suffisamment génériques pour permettre de compiler dans n'importe quel langage. Pour utiliser la machine *llvm*, il suffit d'appeler le script `mocc` avec l'option `-m llvm` (la machine *tam* est la machine par défaut).

La compilation en langage intermédiaire *llvm* nous permet de générer des exécutables natifs utilisant la *libc* bien plus facilement, et de manière plus portable [1] que ne l'aurait été la génération en directement en *x86*. De plus le langage étant plus expressif (il est notamment typé), il permet de trouver plus facilement les erreurs éventuelles. Enfin il nous permet de bénéficier des différentes passes d'optimisation du compilateur `llc` (il est intéressant de noter qu'il est possible de voir le résultat après chaque passe).

Nous avons donc supprimé l'assembleur en-ligne tel qu'il était fourni (dans `ASM.egg`), et rajouté une instruction `asm` qui prend une chaine de caractères, ce qui permet d'inclure de l'assembleur *llvm* ou *tam* selon la machine cible voulue.

*tam* a cependant plusieurs avantages par rapport à *llvm* : pour commencer *llvm* est beaucoup plus expressif et typé : c'est souvent un avantage, mais rend parfois certaines tâches très complexe (par exemple la gestion de la *vtable*, il est aussi nécessaire de préciser le type des variables et registres à chaque utilisation (et non déclaration), ce qui est très répétitif).

Ensuite le nombre d'instructions très réduit de *tam* rend la machine *tam* relativement simple, alors que la machine *llvm* est bien plus complexe.

Enfin la possibilité de voir l'évolution de la pile et du tas instruction par instruction est un gros avantage de *tam*. Pour pouvoir débugger la machine *llvm* il aurait fallut ajouter les informations de débuggage au fichier construit, rendant la machine encore plus complexe.

# 2 Tests

Nous avons écrit beaucoup de tests. Il y en a 4 types (tous dans le dossier `tests`) :

— ceux des dossiers `success`, `warning`, `failure` testent l'analyse syntaxique et sémantique du compilateur (respectivement que le code est correct, qu'il génère un warning (voir section 6) ou qu'il génère une erreur). Ces exemples ne font rien de particulier ;

— ceux du dossier `runnable` testent le code généré, les exemples sont compilés par `mocc` puis doivent être compilés avec `llc` ou lancés dans `tam` et lancés.
La sortie attendue pour le programme `exemple.moc` se trouve dans le fichier `exemple.moc.output`.
Ces fichiers nécessitent d'être préprocessés avant d'être compilés afin d'y inclure les fonctions d'affichages spécifique à *llvm* ou *tam* (écrites en assembleur).

1. Bien que nous compilions uniquement pour une machine linux 64 bits, le porter en 32 bits serait assez simple. Le porter sur Windows nécessiterait uniquement d'ajouter des déclarations pour le linkage.

Les scripts `tests/run-llvm` et `tests/run-tam` permettent de préprocesser, compiler et lancer ces tests.

# 3 Génération de code

Nous avons utilisé la syntaxe de Maxime, celle-ci étant plus légère (elle permet notamment de chainer les appels de méthode).

Il n'y a pas de code spécifique à *llvm* ou a *tam* en dehors de leur dossier respectif. L'interface `IMachine` et le `egg` ont été prévues pour être le plus génériques possible, on devrait pas exemple pouvoir l'utiliser pour générer du *C*.

En particulier, la classe `Type` n'a pas de taille. En effet, si l'on devait générer par exemple du *C*, celle-ci ne servirait pas (on utiliserait plutôt l'opérateur `sizeof`). Pour récupérer la taille d'un type, mais aussi son nom en *llvm* (par exemple le type `Int` est représenté par un `i64`, la constante `"hello"` par un `[6 x i8]` – le nom n'ayant de sens qu'avec cette machine) on utilise un visiteur.

De même, il y a deux interfaces `IExpr` et `ILocation` utilisées par l'interface `IMachine` et le `egg`. Ces interfaces représentent une expression et un emplacement mémoire pour chaque machine.

En *llvm*, une expression contient soit un emplacement mémoire (qui est un nom de la forme `%nom` pour les variables nommées, ou `%5` pour les temporaires) et le code nécessaire pour la générer, soit une constante (par exemple `42`, ou encore `null`). Une expression contient aussi un booléen indiquant s'il est nécessaire d'utiliser l'instruction `load` pour obtenir la valeur.

En *tam*, une expression contient le code nécessaire pour la générer, et un booléen indiquant s'il est nécessaire d'utiliser l'instruction `loadi` pour obtenir la valeur – ce booléen n'a pas forcément la même valeur que celui de la machine *llvm*, et serait inutile si l'on générait par exemple du *C*, il ne fait donc pas parti de l'interface commune. Les expressions *tam* n'ont pas besoin de contenir d'adresse, celles-ci ne servent que pour obtenir l'expression associée à un identifiant de la table des symboles (variable locale, paramètre de fonction, attribut d'une classe).

# 4 Extensions du langage

## 4.1 Asm

Comme indiqué dans l'introduction, nous avons remplacé l'instruction `asm` fournie par une instruction plus simple prenant une chaine de caractère.

```
// en llvm:
void put_char(Char c) {
    asm("%1 = load i8* %c.0, align 1");
```

```
    asm("%2 = sext i8 %1 to i32");
    asm("%3 = call i32 @putchar(i32 %2)");
}

// en tam:
void put_char(Char c) {
    asm("LOAD (1) -1[LB]");
    asm("SUBR COut");
}
```

Il n'est pas possible d'utiliser les variables autrement que par leur adresse. Vu le peu d'assembleur en-ligne que nous utilisons, nous n'avons pas jugé qu'ajouter la possibilité d'utiliser des variables par leur nom en valait la peine.

## 4.2 Alias de type

Il est possible de définir des alias de type :

```
using NouveauNom = NomExistant;
```

La syntaxe évite volontairement le `typedef` bizarre du C.

## 4.3 Tableaux

On peut créer des tableaux :

```
Char[5] s = "net7";
```

La taille se met après le type, et non le nom comme en C.
Ils sont convertibles en pointeurs vers le type correspondant, mais le font dans moins de cas qu'en C (notamment les tableaux sont copiables, passables comme paramètre de fonction et peuvent être retournés). Le type tableau est un vrai type.

## 4.4 Opérateurs new et delete

Ces opérateurs sont équivalents à `malloc` et `free`, mais sont typés (le langage ne possède pas de type `void*`).

```
Int* taille = new(Int);
*taille = 10;

Char* test = new[*taille](Char);
delete(taille);
delete(test);
```

L'opérateur `new` a aussi l'avantage de tenir compte de la taille et d'initialiser la *vtable* (voir section 5.4.1) dans le cas d'une classe.

### 4.5 Opérateur sizeof

Cet opérateur retourne la taille d'un type :

```
Int a = sizeof(Int); // 1 en tam, 8 en llvm sur une machine 64bits
```

Il tient compte de la *vtable* dans le cas d'une classe.

### 4.6 Boucle

Nous avons ajouté une boucle `while`.

## 5 Design decisions

### 5.1 Typage

Le typage est plus fort qu'en C.

En particulier, il n'y a pas de type `void*`, ce qui empêche notamment d'écrire une fonction comme `malloc`, mais nous avons ajouté un opérateur `new` (voir section 4.4).

Le typage des tableaux (voir section 4.3) est aussi plus fort qu'en C.

Cependant les entiers se comportent comme des booléens : nous avons commencé comme cela, nous aurions dû ajouter le type booléen dès le début.

#### 5.1.1 Conversions

Il a deux types de casts : implicites et explicites.

Les casts implicites ont lieux dans peu de cas. Par exemple la constante `null` est de type `NullType` mais est convertible implicitement vers tout les types de pointeur et un tableau est convertible vers un pointeur du même type. De même, dans un `if` ou un `while` la « condition » peut être un pointeur.

Les casts explicites se font à l'aide de `(Type)valeur`. Tout les casts ne sont pas permis cependant : par exemple un caractère ne peut pas être converti en pointeur, bien qu'en *tam* ils soient tout les deux représentés par un mot et qu'en *llvm* un caractère fait 8 bits et un pointeur 64.

### 5.2 Noms de types

Les noms de type commencent tous par une majuscule. Ceci afin de permettre d'avoir des alias de type (voir section 4.2) et par uniformité avec les nom de classes.

Cette restriction permet de désambiguïser une instruction comme `a*b;` qui pourrait être interprétée comme la multiplication de `a` et `b` ou la déclaration d'un pointeur `b` de type `a*`.

## 5.3 NULL et nil, YES et NO

Il n'y a pas de `nil` (qui serait inutile vu que nous n'avons pas vu de différence avec `NULL`) et `NULL` s'écrit `null` par consistance avec les autres variables.

De même `YES` et `NO` s'écrivent `yes` et `no`.

## 5.4 Classes

Nous avons supprimé le `@` devant `@class`. Les méthodes se mettent entre les accolades, après les attributs ; il n'y a donc plus de `@end`.

Par exemple :

```
class Point {
    Int x;
    Int y;

    +(void) init {
    }

    -(Int) x {
        return 0;
    }

    -(Int) y {
        return 0;
    }
}
```

### 5.4.1 Table virtuelle

Afin de permettre l'appel de méthode par liaison tardive, chaque classe possède un membre implicite : sa table virtuelle. Ce membre est implicitement initialisé par l'opérateur `new`.

Lors d'un appel de méthode sur une instance de type connu, l'existence de la méthode et le type des paramètres est vérifié à la compilation. La méthode à appeler est cherchée dans la *vtable* par son index à l'exécution.

Cependant, afin de permettre d'appeler une méthode sur une instance de type inconnu (`id`), la table virtuelle n'est pas un simple tableau contenant l'adresse des fonctions. À la place, c'est une table associative `nom -> adresse` (où le « nom » est par exemple de la forme `x:y:z:` pour refléter le fait qu'en *Objective-C* une méthode a plusieurs noms).

Pour appeler une méthode sur une instance de type inconnu, on pourrait alors chercher la méthode, par son nom dans cette table. Cependant, nous n'avons pas eu le temps d'implémenter ce type d'appel.

# 6 Warnings

Nous avons ajouté des warnings au compilateur.
Il suffit d'appeler `mocc` avec `-w nom_du_warning`. Il y a aussi un warning `all`.

## 6.1 unreachable

`unreachable` vérifie la présence d'instructions inutiles.
Par exemple :

```
Int test() {
    [...]

    if(a) {
        return 123;
    }
    else {
        return 456;
    }
    f(); // unreachable
}
```

## 6.2 shadow

`shadow` vérifie qu'une déclaration ne masque pas une déclaration précédente, par exemple :

```
Int test() {
    Int a;

    if(a) {
        Char a; // shadow
    }
}
```