Introduction à l'infrastructure `clang-LLVM`

Manuel Selva
manuel.selva@inria.fr

Février 2017

Résumé

Ce TP a pour objectif principal la découverte de l'infrastructure de compilation `clang-LLVM`. À la fin du TP, nous devons être en mesure d'implémenter une, voire plusieurs des passes proposées en dernière section. Afin de se concentrer sur l'essentiel et de ne pas perdre trop de temps, notamment sur les aspects mise en place de l'environnement, le sujet guidera dans un premier temps le lecteur de manière précise pour installer un environnement de développement `clang-LLVM` fonctionnel. La deuxième partie de ce document introduit l'infrastructure `clang-LLVM` de façon générale puis présente les outils fondamentaux de cette infrastructure. La troisième partie du TP concerne le développement de passes `clang-LLVM`, c'est à dire d'analyses et de transformations de code lors de la compilation d'un programme source vers un programme machine. Là encore, le sujet guidera le lecteur pas à pas afin d'écrire, de compiler et d'appliquer une passe relativement simple. Cette section présente les concepts fondamentaux nécessaires pour commencer à travailler avec l'infrastructure `clang-LLVM`. La dernière section du sujet propose une liste de passes à réaliser. Le sujet est délibérément ouvert sur ce dernier point, de manière à laisser le lecteur découvrir de façon pratique et autonome le développement dans l'infrastructure `clang-LLVM`.

Note : ce document s'inspire largement d'un article d'Adrian Sampson disponible sur le web : `https://www.cs.cornell.edu/~asampson/blog/llvm.html`. Merci de me signaler toute erreur en utilisant l'adresse électronique ci-dessus.

Table des matières

1 Mise en place de l'environnement de travail **3**
1.1 Installation `clang-LLVM` depuis les sources 4
1.2 Verification de l'installation 5

2 L'infrastructure `clang-LLVM` **6**
2.1 Vue d'ensemble 6
2.2 Compilation et exécution d'une passe 7
2.3 Liste des outils - Kit de survie 8
2.3.1 `clang` 8
2.3.2 `opt` 10
2.3.3 `llc` 10
2.3.4 `llvm-as` et `llvm-dis` 11
2.3.5 `lli` 11

3 Développement d'une passe `LLVM` **12**
3.1 IR `LLVM` 12
3.2 Création d'une passe 14
3.3 Parcourir l'IR `LLVM` 15
3.4 Modification de l'IR `LLVM` 15
3.5 Enregistrement et exécution d'une passe 16
3.6 Assemblage 17

4 Passes à développer **18**

5 Liens **19**

1 Mise en place de l'environnement de travail

L'environnement de travail que nous utiliserons aura la forme suivante :

```
tp-llvm
 ├── llvm.3.9.1-install-release
 ├── llvm.3.9.1-install-debug (optionnel)
 ├── llvm.3.9.1.src (optionnel)
 ├── pass-function-names
 │    ├── CMakeLists.txt
 │    └── FunctionNamePass.cpp
 ├── pass-mul-to-add
 │    ├── CMakeLists.txt
 │    └── MulToAddPass.cpp
 ├── pass-yyy
 │    ├── ...
 │    └── ...
 ├── pass-zzz
 │    ├── ...
 │    └── ...
 └── test.c
```

Nous commencerons donc récupérer l'archive préparée pour le TP :

```
cp /mnt/media/USB-TP-LLVM/tp-llvm.tgz ~
cd ~
tar xvf tp-llvm.tgz
```

Dans tous les exemples de commandes shell ci-dessous, nous considérons que nous nous trouvons à la racine du dossier `tp-llvm`.

Afin de développer nos propres passes `LLVM` nous devons installer une version développement de `LLVM` et de `clang`. Dans le cadre de notre TP, **nous utiliserons la version 3.9.1**. L'archive du TP contient une version déjà compilée de `clang` et de `LLVM`. Cette version compilée se trouve dans le dossier `llvm.3.9.1-install-release`.

Nous décrivons également ci dessous dans la section 1.1 comment compiler `clang` et `LLVM` depuis les sources car il peut être intéressant d'accéder au code source afin de pouvoir se plonger dans celui-ci. Pour pouvoir utiliser cette version, tout ce que nous avons à faire consiste ajouter les outils `clang-LLVM` à notre `path`.

```
# à éxécuter à la racine du répertoire du TP
export PATH=`pwd`/llvm-3.9.1-install-release/bin:${PATH}
```

1.1 Installation clang-LLVM depuis les sources

Comme cette solution requiert une compilation complète de LLVM et de clang, il faut compter beaucoup de temps (en fonction de la machine, à titre d'exemple, environ 25 minutes sur une machine Intel Nehalem avec 6 cœurs) et beaucoup de mémoire vive (environ 16 gigas en mode Debug).

Récupération des sources LLVM Nous devons dans un premier temps télécharger les sources de LLVM. Nous utiliserons les commandes suivantes pour télécharger et décompresser la versions 3.9.1 :

```
# à exécuter à la racine du répertoire du TP
wget http://releases.llvm.org/3.9.1/llvm-3.9.1.src.tar.xz
tar xvf llvm-3.9.1.src.tar.xz
```

Une fois l'archive téléchargée et décompressée, nous pouvons **compiler** LLVM. Néanmoins, l'archive que nous venons de télécharger et de décompresser ne contient que le code source de LLVM. Comme nous travaillerons sur des programmes écrits en C, nous devons également installer clang.

Récupération des sources clang La version de clang que nous utiliserons doit être la même que celle de LLVM. De plus, l'archive de clang doit être décompressée dans le répertoire tools des sources LLVM décompressées précédemment. Nous utiliserons les commandes suivantes :

```
# à exécuter à la racine du répertoire du TP
cd llvm-3.9.1.src/tools
wget http://releases.llvm.org/3.9.1/cfe-3.9.1.src.tar.xz
mkdir clang
tar xvf cfe-3.9.1.src.tar.xz -C clang --strip-components 1
```

Compilation de LLVM et clang Une fois que nous avons récupéré les sources de LLVM et de clang, nous pouvons procéder à leur compilation en mode Debug à l'aide des commandes suivantes :

```
# à exécuter à la racine du répertoire du TP
cd llvm-3.9.1.src
mkdir build-debug;cd build-debug
cmake \
 -DCMAKE_BUILD_TYPE=Debug -DLLVM_TARGETS_TO_BUILD=X86 \
 -DCMAKE_INSTALL_PREFIX=`pwd`\
   /../../llvm-3.9.1-install-debug ..
make -j6 install
```

La dernière commande `make -j6 install` démarre la compilation en utilisant 6 tâches parallèles. Il est en général recommandé d'utiliser un nombre de tâches légèrement inférieur au nombre de cœurs de la machine.

Une fois la compilation terminée, nous ajouterons tous les outils que nous venons de compiler à nore `PATH` de la manière suivante :

```
# à exécuter à la racine du répertoire du TP
export PATH=`pwd`/llvm-3.9.1-install-debug/bin:${PATH}
```

1.2 Verification de l'installation

Une fois les outils `clang-LLVM` installés, nous vérifierons leur bon fonctionnement à l'aide des commandes suivantes :

```
# à exécuter à la racine du répertoire du TP
clang --version
clang version 3.9.1 (tags/RELEASE_391/final)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /chemin/llvm/install/bin
```

```
# à exécuter à la racine du répertoire du TP
opt --version
LLVM (http://llvm.org/):
  LLVM version 3.9.1
  DEBUG build with assertions
  Built Feb 20 2017 (11:54:49).
  Default target: x86_64-unknown-linux-gnu
  Host CPU: nehalem
```

La sortie de ces commandes est dépendante de la version de `LLVM` que nous utilisons et de l'option `Release` ou `Debug` utilisée lors de la compilation.

2 L'infrastructure `clang-LLVM`

Cette section présente l'infrastructure `clang-LLVM` dans son ensemble. L'objectif principal est de se familiariser avec la terminologie, afin de savoir notamment **"qui fait quoi ?"** Nous parlerons ici de `clang-LLVM`, car à proprement parler, l'infrastructure `LLVM` ne contient pas les outils nécessaires pour compiler du code source C ou C++ mais uniquement des outils d'optimisation et de génération de code machine à partir d'un format intermédiaire.

2.1 Vue d'ensemble

L'infrastructure `clang-LLVM`, de façon macroscopique, est construite de manière similaire à tout compilateur moderne. La figure 1 ci dessous présente l'architecture globale.

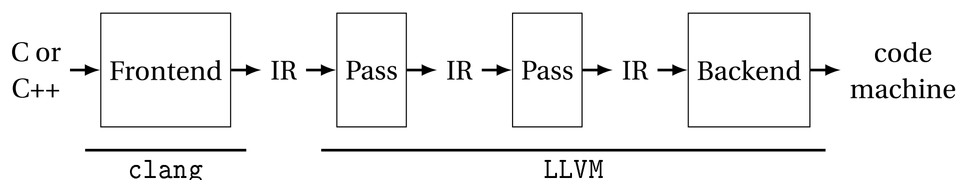

FIGURE 1 – L'infrastructure `clang-LLVM`

— le **frontend** est le premier bloc de tout compilateur. Il a pour objectif de valider que le programme est syntaxiquement et sémantiquement correct puis de le traduire vers une représentation intermédiaire (IR pour Intermediate Representation). L'un des objectifs de cette représentation intermédiaire étant de simplifier le travail des autres blocs qui ne peuvent pas travailler avec la complexité de code source C ou encore pire C++. Comme indiqué sur la figure, le frontend C/C++ pour la représentation intermédiaire `LLVM` est `clang`. Dans ce TP nous utiliserons `clang` pour générer de l'IR `LLVM` à partir de sources C/C++ mais nous ne le modifierons pas.

— les **passes** sont en charge d'analyser et/ou de transformer l'IR en optimisant certaines choses tout en préservant la sémantique du code. L'objectif étant très souvent la maximisation des performances du code mais cela peut aussi être par exemple la taille du code. Durant ce TP, c'est ici que nous travaillerons. Autrement dit, **nous allons développer des passes** `LLVM`.

— enfin, le **backend** est en charge de transformer l'IR vers du code machine pour une architecture donnée. Dans notre cas, nous utiliserons donc le backend X86 de `LLVM`. Nous n'interviendrons pas à ce niveau durant ce TP.

Concernant la représentation intermédiaire LLVM, la fameuse IR LLVM[1], il faut savoir que celle-ci peut être sauvegardée sur notre disque dur dans deux formats différents :

— format binaire, généralement appelé bitcode, dont les fichiers ont l'extension .bc

— format textuel, parfois appelé LLVM assembly, lisible par un être humain, dont les fichiers portent l'extension .ll

Les outils llvm-as et llvm-dis décrits dans la section 2.3.4 permettent de passer de l'une à l'autre de ces représentations.

2.2 Compilation et exécution d'une passe

Avant de décrire en détails les différents outils réalisant la chaîne de compilation décrite ci dessus, nous allons voir comment compiler et exécuter nos propres passes. L'archive du TP inclut la passe function-names, qui affiche sur la sortie standard le nom de toutes les fonctions du programme en cours de compilation. Afin de compiler une passe LLVM, nous utilisons en général des scripts cmake car l'infrastructure clang-LLVM utilise elle même cmake. Un ficher CMakeLists.txt pour la passe function-names est inclus dans l'archive.

Pour compiler le code source de cette passe, nous utiliserons donc les commandes suivantes :

```
# à exécuter à la racine du répertoire du TP
cd pass-function-names
mkdir build; cd build
cmake ..
make
```

Si tout se passe bien, le fichier libLLVMFunctionNamesPass.so est créé dans le répertoire build. Sinon, nous devons lire attentivement le/les message/s d'erreur affichés sur la console afin de comprendre l'origine du problème pour pouvoir le corriger.

Une passe LLVM consiste donc en une librairie dynamique, un fichier ayant l'extension .so sous Linux. Nous utiliserons les commandes suivantes pour invoquer notre passe sur le fichier test.c inclus dans l'archive du TP :

```
# à exécuter à la racine du répertoire du TP
clang -S -emit-llvm -o test.ll test.c
opt -load \
`pwd`/pass-function-names/build/libLLVMFunctionNamesPass.so \
-fnames -o /dev/null \
test.ll
```

1. http://llvm.org/docs/LangRef.html

2.3 Liste des outils - Kit de survie

De nombreux outils sont disponibles afin d'intervenir aux différents niveaux de la chaîne de compilation présentée dans la section précédente. Il est fondamental de connaître ces outils afin de pouvoir travailler efficacement avec l'infrastructure `clang-LLVM`.

Afin de se familiariser tout de suite avec ces outils, nous commençons par donner un aperçu des différents outils via un exemple concret figure 2 opérant sur le programme C suivant :

```c
#include <stdio.h>

static void never_called() {
  printf("I am never called\n");
}

int main() {
  int fortytwo = 42;
  int number = fortytwo * fortytwo;
  if (number < 42) {
    never_called();
  }
  printf("numbers is %d\n", number);
  return 0;
}
```

Le détail de chacun de ces outils est donné dans les prochaines sections. Il est recommandé de tester les outils au fil de la lecture de cette section. De plus, afin de commencer à se familiariser avec l'IR `LLVM` décrite plus en détails par la section 3.1 nous explorerons le contenu de tous les fichiers `.ll` créés.

2.3.1 `clang`

Comme décrit dans la section 2, `clang` est à proprement parler le frontend C/C++ de l'infrastructure `LLVM`. Néanmoins, comme il serait pénible pour l'utilisateur de devoir invoquer plusieurs outils afin de compiler son code source, **l'outil** `clang` est un “driver de compilation” [2] au même titre que l'est l'outil `gcc`. Autrement dit, l'outil `clang` inclut toujours le frontend `clang` mais peut aussi inclure certains des autres outils décrits ci-dessous afin de générer directement du code machine à partir du code source.

C'est d'ailleurs le comportement par défaut de `clang`. La commande suivante permet par exemple de compiler le fichier `test.c` directement vers du code machine :

2. https://clang.llvm.org/docs/DriverInternals.html

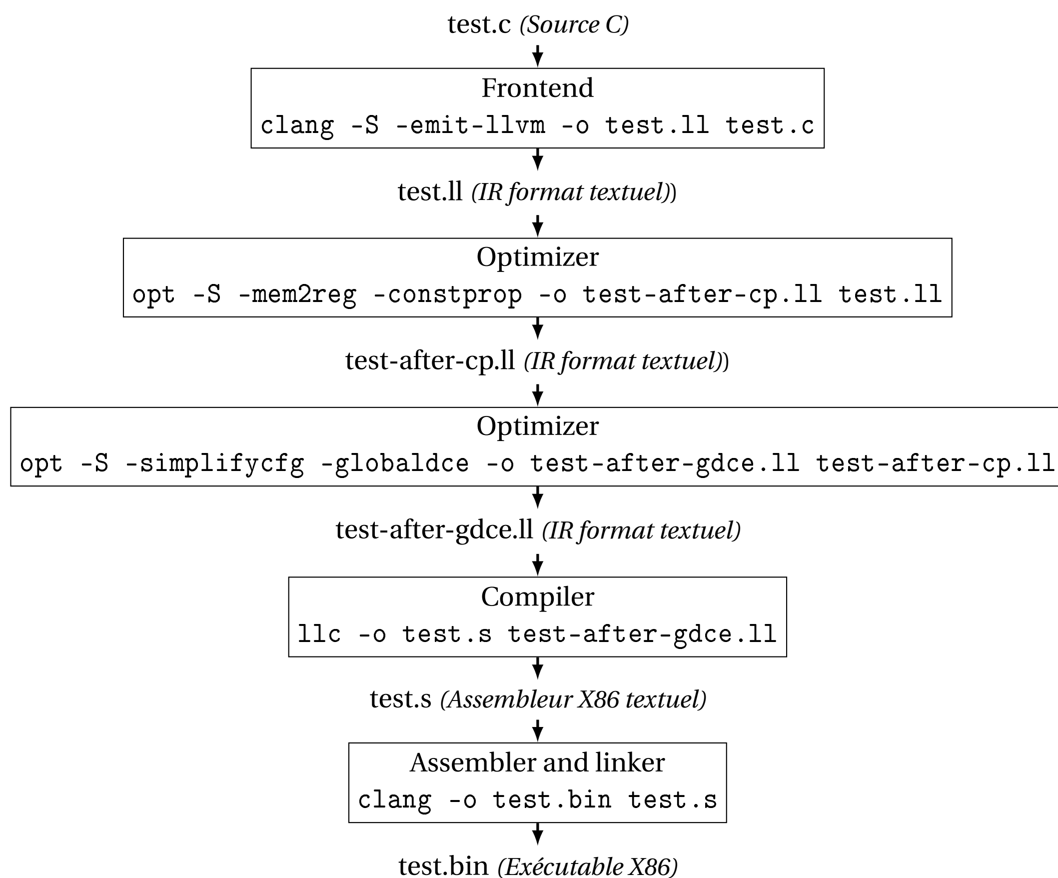

FIGURE 2 – Exemple d'utilisation des outils de l'infrastructure `clang-LLVM`

```
clang -o test.bin test.c
```

En tant que driver de compilation, l'outil `clang` peut être arrêté à différents niveaux de la chaîne de compilation. Comme le montre la première étape de notre exemple sur la figure 2, `clang` peut-être utilisé en tant que frontend uniquement, c'est à dire générant de l'IR au format textuel (`.ll`) à partir du code source. Pour cela il faut lui donner les options suivantes :

```
clang -S -emit-llvm -o test.ll test.c
```

L'option `-S` indique que nous souhaitons uniquement compiler (sans assembler ni linker) notre programme source au même titre que l'option `-S` de `gcc`. L'option `-emit-llvm` indique que nous souhaitons que la sortie soit au format IR textuel `LLVM` et non assembleur de la machine cible (X86 dans notre cas).

L'outil `clang` peut aussi être utilisé comme assembleur et linker comme le

montre la dernière étape de de notre exemple sur la figure 2. L'outil `clang` détecte automatiquement qu'il faut simplement invoqué l'assembleur et le linker lorsque l'extension du fichier d'entrée est `.s` tel que dans notre exemple :

```
clang -o test.bin test.s
```

2.3.2 `opt`

L'outil `opt` permet d'appliquer un ensemble de passes `LLVM`. L'entrée de `opt` est un fichier au format IR (bitcode ou textuel) et la sortie produite est également un fichier au format IR (bitcode ou textuel).

Pour le choix des passes à appliquer, l'outil `opt` peut être utilisé avec les option usuelles `-O1, -O2, -O3`.

Par défaut, si aucune des options `-Ox` n'est spécifiée, `opt` n'applique aucune passe. Dans ce cas, nous pourrons spécifier individuellement les passes que nous souhaitons appliquer. La seconde étape de de notre exemple sur la figure 2 indique à `opt` que nous voulons appliquer les passes "mem2reg" et "constant propagation" :

```
opt -S -mem2reg -constprop -o test-after-cp.ll test.ll
```

La passe "mem2reg" est une passe nécessaire afin de promouvoir les allocations dans la pile générées par le frontend vers des variables ayant la forme Single Static Assignement (SSA) [3]. Sans rentrer dans les détails, une représentation intermédiaire SSA est une representation dans laquelle chaque variable est assignée une seule fois. Cette propriété facilite grandement de nombreuses optimisations efféctuées par le compilateur. Enfin, l'option -S indique ici que nous souhaitons que la sortie soit au format textuel et non au format bitcode (par défaut).

2.3.3 `llc`

L'outil `llc`, pour `LLVM` compiler, permet de compiler du code au format `LLVM` IR (bitcode ou textuel) vers du code assembleur pour une architecture donnée, par défaut l'architecture sur laquelle `llc` est exécuté. Dans notre exemple figure 2, `llc` est invoqué après avoir utilisé `opt` pour optimiser l'IR de notre programme de test :

```
llc -o test.s test-after-gdce.ll
```

Le fichier de sortie, `test.s`, est au format assembleur textuel (X86 dans notre cas). Afin de terminer la compilation de notre programme pour pouvoir l'exécuter il est nécessaire d'assembler puis de linker le fichier `test.s`. Pour cela, nous utiliserons à nouveau l'outil `clang`, driver de compilation :

3. http://llvm.org/docs/tutorial/OCamlLangImpl7.html

```
clang -o test.bin test.s
```

2.3.4 `llvm-as` et `llvm-dis`

`llvm-as` et `llvm-dis` permettent respectivement de passer du format `LLVM` IR textuel au format `LLVM` IR bitcode et inversement. Par exemple :

```
llvm-as -o test.bc test.ll
```

2.3.5 `lli`

Le dernier outil que nous présentons est l'interpréteur d'IR `LLVM`, `lli`. Cet outil permet d'exécuter du code au format `LLVM` IR (bitcode ou textuel) non pas en le compilant vers du code machine mais en l'interprétant directement :

```
lli test.ll
```

3 Développement d'une passe LLVM

Cette section présente pas à pas les concepts nécessaires à la réalisation d'une passe LLVM. Nous réaliserons au fil de cette section une passe qui transforme toutes les multiplications d'un programme en additions. Cette passe n'est pas réaliste car elle change la sémantique du code, mais elle permet d'illustrer les concepts fondamentaux concernant les passes LLVM.

3.1 IR LLVM

Dans la représentation intermédiaire LLVM, un programme est un ensemble de modules. Ces modules sont les conteneurs de base de tous les autres objets de l'IR LLVM comme le montre la figure 3.

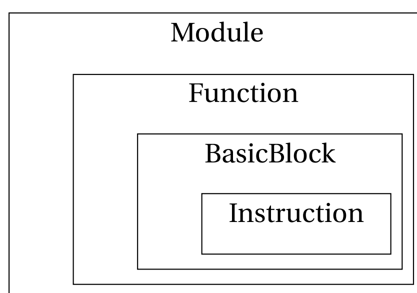

FIGURE 3 – Hiérarchie des objets de l'IR LLVM

Ces quatre objets, `Module`, `Function`, `BasicBlock` et `Instruction` sont les éléments fondamentaux de la représentation intermédiaire LLVM. Grossièrement, un module correspond à un fichier C, avec une liste de variable globales, une liste de dépendances vers d'autres modules ainsi qu'une d'une liste de fonctions. Une fonction LLVM correspond directement à une fonction C. Chaque fonction contient une liste de blocs de base. Un bloc de base est une suite d'instructions avec un seul point d'entrée (la première instruction) et un seul point de sortie (la dernière instruction). Les instructions représentent des opérations basiques telle qu'une addition, un accès mémoire ou une division flottante.

Afin de voir concrètement à quoi correspondent ces concepts, nous allons analyser le fichier `test.ll` généré dans la section précédente et contenant l'IR LLVM de notre programme `test.c` au format textuel. Ce fichier a **été généré avec une version** `Debug` de l'infrastructure `clang-LLVM`, c'est à dire compilée en mode `Debug`. En effet, afin de retrouver le nom des variables du code source dans l'IR comme c'est le cas ici, il faut utiliser une version `Debug` de `clang-LLVM`.

La première ligne du fichier `test.ll` indique le nom du module. Celui-ci correspond au nom du fichier C. Deux variables globales `@.str` et `@.str.1` ont été créées bien que n'existant pas directement dans notre programme source. Ces variables correspondent aux deux chaînes de caractères littérales utilisées dans notre programme source.

```
; ModuleID = 'test.c'
source_filename = "test.c"
@.str = private unnamed_addr constant [14 x i8]
        c"number is %d\0A\00", align 1
@.str.1 = private unnamed_addr constant [19 x i8]
        c"I am never called\0A\00", align 1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %fortytwo = alloca i32, align 4
  %number = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 42, i32* %fortytwo, align 4
  %0 = load i32, i32* %fortytwo, align 4
  %1 = load i32, i32* %fortytwo, align 4
  %mul = mul nsw i32 %0, %1
  store i32 %mul, i32* %number, align 4
  %2 = load i32, i32* %number, align 4
  %cmp = icmp slt i32 %2, 42
  br i1 %cmp, label %if.then, label %if.end

if.then:                              ; preds = %entry
  call void @never_called()
  br label %if.end

if.end:                               ; preds = %if.then, %entry
  %3 = load i32, i32* %number, align 4
  %call = call i32 (i8*, ...) @printf(--)
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

; Function Attrs: nounwind uwtable
define internal void @never_called() #0 {
entry:
  %call = call i32 (i8*, ...) @printf(--)
  ret void
}
```

Concernant les fonctions, nous pouvons constater que notre module contient deux définitions et une déclaration. Les deux définitions correspondent respec-

tivement aux fonctions `main` et `never_called` définies dans notre programme C. La déclaration concerne la fonction `printf` de la librairie standard que nous appelons depuis `main` et `never_called`.

Enfin, concernant les instructions, nous pouvons identifier différents types :

— allocations dans la pile avec l'instruction `alloca`
— accès mémoires avec les instructions `store` et `load`
— opérations arithmétiques avec l'instruction `mul`
— branchements conditionnels avec l'instruction `br`
— appels de fonctions avec l'instruction `call`
— retours de fonction avec l'instruction `ret`

Comme le montre cet exemple, la représentation intermédiaire de `LLVM` est fortement typée. En effet toutes les instructions spécifient le type des données sur lesquelles elles opèrent.

3.2 Création d'une passe

Maintenant que nous savons en quoi consiste l'IR `LLVM`, nous allons créer une passe. Rappelons que l'objectif d'une passe est d'analyser et/ou de transformer de l'IR `LLVM`.

Les passes `LLVM` peuvent opérer à différent niveau de l'IR. Ici nous allons travailler sur une passe opérant au niveau des fonctions. Cette passe remplacera toutes les multiplications rencontrées dans la fonction par des additions. L'objectif de cette passe est **uniquement pédagogique** dans la mesure ou celle-ci change la sémantique du programme. Il est important de noter qu'une passe de type fonction opère bien sur toutes les fonctions de tous les modules en cours de compilation, mais de façon individuelle. Autrement dit, notre passe sera invoquée par l'infrastructure `LLVM` pour chaque fonction.

Pour créer une passe de type fonction, il suffit de créer une sous classe de la classe `FunctionPass`.

```
struct MulToAddPass : public FunctionPass {
  static char ID;
  MulToAddPass() : FunctionPass(ID) {}
  virtual bool runOnFunction(Function &F) {
      // The body of our pass
      ...
  }
};
```

Cette sous classe doit déclarer un attribut statique de type char nommé `ID`. Le corps de la passe sera implémenté dans la fonction `runOnFunction`. C'est cette fonction qui sera invoquée par `LLVM` sur toutes les fonctions du code source en cours de compilation.

3.3 Parcourir l'IR LLVM

Le point d'entrée de notre passe est donc la fonction `runOnFunction`. En paramètre de cette fonction nous recevons un objet de type `Function` représentant la fonction LLVM IR sur laquelle notre passe va opérer. Il est possible d'itérer très facilement sur les différents objets de l'IR LLVM. Comme une fonction contient des `BasicBlock`, nous utiliserons le code suivant pour les parcourir, en parcourant également pour chacun d'eux toutes les instructions :

```
virtual bool runOnFunction(Function &F) {

  // Iterrate over blocks of the function
  for (BasicBlock &B : F) {

    // Iterrate over instructions of the block
    for (Instruction &I : B) {

      // Check that op is a multiplication
      BinaryOperator* op = dyn_cast<BinaryOperator>(&I);
      if (op && op->getOpcode() == Instruction::Mul) {
        // Replace the operator by an addition
        ...
      }
    }
  }
  // True means here that our pass
  // has modified the function F
  return true;
}
```

Comme le montre cette exemple, nous utilisons le template `dyn_cast` de l'API LLVM afin d'identifier les instructions qui nous intéressent, à savoir les multiplications.

3.4 Modification de l'IR LLVM

Maintenant que nous savons comment parcourir l'IR LLVM et identifier les multiplications, nous allons voir comment remplacer ces dernières par des additions. Nous allons donc devoir maintenant utiliser l'API LLVM pour **modifier** la représentation intermédiaire.

La classe LLVM clef pour cette opération est la classe `IRBuilder`. Le code ci dessous indique comment construire et utiliser un `IRBuilder` afin d'effectuer les modifications qui nous intéressent.

Les commentaires inclus dans le code ci dessous indiquent la signification de chacun des blocs de code.

```
virtual bool runOnFunction(Function &F) {
  for (BasicBlock B : F) {
    for (Instruction I : B) {
      if (MultOperator* op = dyn_cast<MultOperator>(&I)) {
        // Insert at the point where the
        // instruction `op` appears.
        IRBuilder<> builder(op);

        // Make an addition with the same operands as `op`.
        Value* lhs = op->getOperand(0);
        Value* rhs = op->getOperand(1);
        Value* add = builder.CreateAdd(lhs, rhs);

        // Everywhere the old instruction was used as an
        // operand, use our new add instruction instead.
        for (auto& U : op->uses()) {
          User* user = U.getUser();
          // A User is anything with operands.
          user->setOperand(U.getOperandNo(), add);
        }
      }
    }
  }
  return true;
}
```

3.5 Enregistrement et exécution d'une passe

Enfin, afin de pouvoir utiliser notre passe, nous devons enregistrer celle-ci au près de l'infrastructure LLVM. Il existe deux façons différentes d'enregistrer une passe LLVM dont le résultat sera différent en terme d'utilisation de la passe.

La première solution consiste à enregistrer notre passe en lui associant le nom d'une option qui sera associée à l'outil opt. Pour ce faire, il faut utiliser le template LLVM RegisterPass de la façon suivante :

```
static RegisterPass<MulToAddPass> MulToAdd(
"multoadd",
"Pass that convert multiplications to additions");
```

Dans cet exemple, le nom de l'option à utiliser pour invoquer notre passe est multoadd.

Afin d'appliquer une passe enregistrée de cette façon, il faut obligatoirement convertir le code source vers de l'IR LLVM puis invoquer opt en utilisant l'option définie via RegisterPass :

```
clang -S -emit-llvm -o test.ll test.c
opt -S -load \
`pwd`/pass-mul-to-add/build/libLLVMMulToAddPass.so \
-multoadd -o test-after-multoadd.ll \
test.ll
```

Cette solution peut rapidement devenir très complexe lorsque l'on souhaite appliquer nos propres passes sur tous les fichiers sources (donc modules LLVM) d'un gros projet. Dans ce cas, nous souhaitons simplement modifier la ligne de commande du compilateur spécifiée dans des fichiers `makefile` pour indiquer que nous souhaitons appliquer notre passe. Dans ce cas, il faut enregistrer notre passe au près du driver de compilation `clang` afin que celui-ci l'applique automatiquement lorsque la librairie dynamique la contenant est chargée. Voici comment faire cet enregistrement :

```
// Function registering our pass in the
// provided pass manager
static void registerMulToAddPass(const PassManagerBuilder &,
legacy::PassManagerBase &PM) {
  PM.add(new MulToAddPass());
}

// Standard function given the function above
// and a parameter indicating when the registered
// pass should be invoked
static RegisterStandardPasses
RegisterMyPass(PassManagerBuilder::EP_EarlyAsPossible,
registerMulToAddPass);
```

Comme nous pouvons le constater sur cet exemple, nous devons spécifier quand est-ce que notre passe doit être appliquée, `EP_EarlyAsPossible` ici, par rapport aux autres passes appliquées par le driver de compilation. Une passe enregistrée de cette manière s'utilise ensuite de la façon suivante :

```
clang -o test.bin \
-Xclang -load \
-Xclang `pwd`/pass-mul-to-add/build/libLLVMMulToAddPass.so \
test.c
```

3.6 Assemblage

L'archive du TP contient le code source complet de notre passe `MulToAdd` ainsi que les scripts nécessaires à sa compilation. Voici les commandes à utiliser pour tester cette passe :

```
# à exécuter à la racine du répertoire du TP
cd pass-mul-to-add
mkdir build
cd build
cmake ..
make
cd ../../
clang -o test.bin \
-Xclang -load \
-Xclang `pwd`/pass-mul-to-add/build/libLLVMMulToAddPass.so \
test.c
./test.bin
```

4 Passes à développer

Voici une liste de passes intéressantes du point de vue pédagogique que nous réaliserons :

- **LiteralPrimeNumberCounter :** cette passe, de type module, compte le nombre de nombres premiers parmi tous les entiers littéraux présents dans le code du module.
- **FunctionParametersPrinter :** cette passe, de type module, modifie l'IR de chaque fonction définie dans le module de telle sorte que celles-ci affichent la valeur effective de tous leur paramètres lorsqu'elles sont invoquées.
- **FunctionParametersPrimeCounter :** cette passe, de type module, modifie l'IR de chaque fonction définie dans le module de telle sorte que celles-ci affichent le nombre de paramètres effectifs entiers et premiers. Le test de primalité sera effectué par un appel à une fonction écrite par ailleurs en C. Nous utiliserons le chapitre "Linking with a runtime library" de la page `https://www.cs.cornell.edu/~asampson/blog/llvm.html` pour voir comment utiliser une telle fonction.
- **MemoryProfiler :** cette passe, de type module, modifie l'IR de chaque fonction définie dans le module de telle sorte que tous les accès mémoires (load et store) soient affichés sur la sortie standard lors de l'exécution du programme. Pour chaque accès, le message indiquera s'il s'agit d'un load ou d'un store ainsi que l'adresse mémoire visée. Dans le cas des stores, le message indiquera également la valeur écrite en mémoire.
- **FunctionParallelizer :** cette passe, de type fonction, modifiera tous les appels de fonctions présents dans la fonction `main` de telle sorte que ces derniers s'effectuent de façon parallèle dans un nouveau thread. Nous commencerons par les appels vers des fonctions sans paramètres ne retournant aucune valeur. La fonction main devra attendre la terminaison de chacun de ces threads avant de se terminer elle même.

5 Liens

— Documentation officielle de l'IR `LLVM` :
  http://llvm.org/docs/LangRef.html
— Écrire une passe `LLVM` :
  http://llvm.org/docs/WritingAnLLVMPass.html
— `LLVM` for grad students :
  https://www.cs.cornell.edu/~asampson/blog/llvm.html
— Appliquer une passe automatiquement avec `clang` :
  https://www.cs.cornell.edu/~asampson/blog/clangpass.html
— Le driver de compilation `clang` :
  https://clang.llvm.org/docs/DriverInternals.html