DF00100 Compiler Labs: Part 1 - LLVM IR

August Ernstsson (based on previous material by Erik Hansson)

1. LLVM

LLVM (https://llvm.org) stands for low level virtual machine and is an unlimited register machine. Register values can only be set once, and has a type. LLVM has three equivalent forms of intermediate representations (IR): assembly form, in-memory representation, and on-disk bitcode.

This part of the labs focus on how to work with the IR of LLVM.

1.1 Installation

We prefer to install LLVM from source code. We will also use the Clang front-end. The following steps are done on Linux, using git and CMake, but LLVM may also work on Mac or Windows, see the documentation.

Clone the LLVM source (Clang is now included in the repository) into **LLVM_DIR** (placeholder for your repository path, put it wherever you like) and configure and build LLVM + Clang.

```
git clone https://github.com/llvm/llvm-project.git
cd LLVM_DIR
git checkout release/10.x
mkdir build
cd build
cmake -G "Unix Makefiles" -DLLVM_ENABLE_PROJECTS="clang" ../llvm
make
```

It is recommended to check out a stable branch, e.g. version 10, but the exact version is not important (note it in your report).

(Optional) Add LLVM_DIR/build/bin/ to your PATH, but be careful of conflicts with existing installations of e.g. clang.

1.2 First step

```
#include <stdio.h>

int main()
{
	printf("Hello World\n");
}
```

Compile it using Clang:

```
LLVM_DIR/build/bin/clang -o test test.c
```

This will produce native code, which you can then run (`./test`).

To generate LLVM IR, in assembly form, run clang as below which will produce test.ll.

```
LLVM_DIR/build/bin/clang test.c -S -emit-llvm
```

Now run the same command but with optimization flag (-O3) and compare the output. Compile other input programs, with and with out optimization, for example:

```
int main()
{
        int x = 0;
        return x++;
}
```

And compare the IR code.

1.3 A simple pass

Write a simple pass that calculates how many calls there are to the `printf()` function. Output can for example be something like:

```
main:
  printf(): 12 calls
foo:
  printf(): 23 calls
```

See http://llvm.org/docs/WritingAnLLVMPass.html or https://llvm.org/docs/WritingAnLLVMNewPMPass.html, for a good introduction about how to get started writing passes. There are different approaches to build and register passes in LLVM, but find one that works for you and describe your approach in the report.

To convert test.ll to bitcode format, run `llvm-as` as below which produces test.bc.

```
LLVM_DIR/build/bin/llvm-as test.ll
```

What will your pass report if `printf()` is called inside a for loop?

1.4 Exercise 1

Rewrite your printf calculation pass so it handles loops. At least if the number of times the loop body will execute can be determined statically, and the loops are perfectly nested. Example:

```
for (int i = 2; i < 4; i++)
{
        printf("Hello World\n");
}
```

Feel free to run/use already existing passes that may be useful, for example:

- mem2reg
- indvars
- loop-rotate

1.5 Exercise 2a

Write a pass (or a set of passes) that recognizes the vector init (initialization of a vector with a constant), example:

```
for (int i = 0; i < 5; i++)
        v [i] = 42;
```

The pass should report: Matched computation, operand, size etc.

Optionally you may replace matched loops by an equivalent function call or perform some other transformation of your choice, e.g. unrolling or parallelization.

1.6 Exercise 2b (optional, gives bonus points in the exam)

Write a pass that recognizes dot product calculation,

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n} a_i b_i$$

where **a** and **b** are vectors with elements a_0, ..., a_n respectively b_0, ..., b_n. The corresponding C code could look like:

```
double dotproduct(double *x, double *y, int size)
{
        double result = 0;

        for (int i = 0; i < size; i++)
        {
                result += x[i] * y[i];
        }

        return result;
}
```

Also consider implementations with double for-loops:

```
for(int i=.., .., ..)
       for(int j=.., .., ..)
              a[i] += b[i][j] * c[j];
```

Example of things that should not be matched:

```
s = s + a[i] * b[j]
```

and

```
s = t + a[i] * b[i]
```

The pass should report: Matched computation, operand, size etc. Alternatively, replace matched loops by an equivalent function call.

1.7 What should be in your report

- Strategy for solving the problems.
- Results of your tests, with comments.
- Well commented source code for your LLVM passes.
- Test programs, in C and LLVM assembly. Do not forget to write which built in passes you used (mem2reg etc.) and if you used any optimizations. Their invocation order is interesting as well.
- See lab intro slides for further information.

Also write which version of LLVM you used and which platform you used. If you downloaded LLVM via git, please write the branch/commit ID/tag you used.

Also include a brief discussion part where you take up any problems you had and what you found most interesting with the lab.