DOCUMENTATION FOR THE LLVM PROJECT

INTRODUCTION AND PROBLEM STATEMENT

LLVM is the name of a compiler that began as a research project in UIUC and is now maintained at Apple. Its main website is located here. LLVM is open-source software, and has been installed in Glue (Linux) for use in this class project.

The goal of this project is to implement *function inlining*, which is when a copy of the callee function's code is placed at the call site of the caller function. The original callee function is not deleted. In this project, inlining should be done only when all of the actual arguments of a function are constants at their call sites and the function has less than ten instructions. For example, consider the following simple C code:

```
main () {
      int diff;
      diff1 = sub(3, 7);
      printf("diff = %d\n", diff);
}
int sub(int x, int y) {
      int z;
      z = x – y;
      return z;
}
```

The above C code needs to be transformed into:

```
main () {
      int diff;

      int x = 3;
      int y = 7;
      int z;
      z = x – y;
      diff = z;

      printf("diff = %d\n", diff);
}
```

Whenever all of the arguments passed to a function are constants at a call site, modify the instructions so that constant arguments are replaced by local variables and then, copy the modified instructions to the call site. Note that the inlined function will have the same functionality as the original function, but with the constant arguments being converted into local variables.

***Note**: In this project, assume that **each function has only one call-site**, meaning that each function is called only once. This is an assumption, which means your code need not check it.*

In the end, the inbuilt LLVM "const prop" (constant propagation) pass should be invoked on the code and the final code will look like the following.

```
main () {
      int diff;

      int x = 3;
      int y = 7;
      int z;
      z = 3 – 7;
      diff = z;

      printf("diff = %d\n", diff);
}
```

All transformations in LLVM are performed on the intermediate representation (IR) of the code. As a result, you need to count the number of IR level instructions to check whether the function has less than ten IR instructions or not. Only if all the arguments are constant and the number of instructions are less than ten, then the function needs to be inlined.

The C code will be translated by LLVM automatically to internally use single static assignment (SSA) form, which will be discussed later in class. For now, suffice it to say that SSA translates code by renaming each original variable into multiple SSA variables, one per write to the original variable, such that every SSA variable is defined (assigned) only once; thus the name.

Your goal is to write a *function inlining* pass in LLVM that automatically performs the above transformation on any program provided to LLVM. Your pass itself will work on LLVM IR provided as input, and produce modified LLVM IR as output, allowing easy integration into the existing LLVM compiler. To do so, you must become familiar with LLVM. Please follow the steps in the rest of this document to achieve this.

COLLABORATION POLICY

This project should be done by students in groups of two each. That arrangement is ideal since one member of a team might get stuck by themselves, and collaboration can help solve this problem, and encourages learning. When doing the project in groups, both students are expected to be knowledgeable with all aspects of the project.

THINGS TO SUBMIT

The project should be uploaded electronically on the class website (the elms/canvas site). The deadline for submission is 11:59pm on Tuesday, May 1, 2018. Later, the TA will email the class mailing list with the exact instructions on how and what to submit. However the submission will include at least the following:

(i) The source code for your pass.

(ii) The output IR after running your pass on each of the examples in the llvm/test_codes directory.

(iii) Any modified Makefile you use to compile your pass.

MACHINE TO USE LLVM

Although most computers on the university glue system use the Solaris OS, a few use the Linux OS. Since LLVM is compatible with Linux but not Solaris, *you must use a Linux glue machine for this project, as otherwise the software will not work*. The following glue Linux machine has been provided to you for this project:

compute.ece.umd.edu

Please log into the above machine using ssh or some other secure remote login method. You can use your existing UMD glue account name and directory ID to login. If you do not have a glue account, please obtain one immediately. If you are having problems accessing your glue account, or you don't have a glue account, the UMD IT help desk recommended that all students call them to get help. Please call them at:

Phone Support: 301.405.1500

Hours: Mon - Fri 8:00 a.m. - 6:00 p.m.

BASICS TO USE LLVM

A brief tutorial to use llvm is described here. LLVM is already installed on the system, so don't try reinstalling it. Some of the important steps and additional steps are listed below.

First remember to add the llvm and llvm-gcc install directories to your environment variable PATH using

```
setenv PATH /afs/glue/class/old/enee/759c/llvm/llvm-3.4-install/opt/bin/:$PATH
```

Next, copy over the example files in /afs/glue/class/old/enee/759c/llvm/test_codes/ to a local directory in your home directory. Then the output bytecode file (e.g., example1.bc) can be obtained from source code written in C (e.g., example1.c) using:

```
clang -c -emit-llvm example1.c -o example1.bc
```

Bytecode is the Intermediate Representation of the LLVM compiler. Please test your programs on the test codes placed in the test_codes folder.

This bytecode can be converted to assembly code (e.g., example1.s) using:

```
llc < input bytecode > -o < output assembly file >
```

This < output assembly file > can be assembled to an executable (e.g., example1) using gcc:

```
gcc  < assembly file > -o < output executable >
```

We can use llvm-dis to disassemble bytecode to a human readable IR file (e.g., example1.ll)

```
llvm-dis < input bytecode > -o < output human readable IR file >
```

This output file is human readable. The instructions in it will be explained in the next section.

This human readable IR can be assembled to bytecode using llvm-as as follows

```
llvm-as < input human readable IR file > -o < output bytecode >
```

INTRODUCTION TO LLVM_IR

The human readable IR that was obtained in the last section can be opened using any editor of your choice.

The first thing that will be helpful in understanding the human readable LLVM would be to understand few of the instructions present in the intermediate representation of LLVM. The language reference manual is present here. A few instructions that would be good to know are:

- Terminator Instructions
- Binary Operations
- Memory Access and Addressing Operations
- Integer Compare Instruction The types present in llvm are described here.

IMPLEMENTATION IN LLVM

You can follow the following procedure to implement the function argument in-lining pass:

1. At every function call in the program, examine its argument list to determine if all of the actual arguments are constants. In addition, check if the function has less than ten instructions.

2. If so, modify the instructions by removing the formal arguments corresponding to constants from the function and add them as local variables.

3. Copy the modified instructions to the call site. The call instruction needs to be removed.

4. Finally, invoke “constprop” pass to eliminate uses of constant local variables and replace them by constant values.

Hints on how to write the above passes are in the “Steps involved” section below.

INFRASTRUCTURE TO WRITE YOUR OWN PASS

A sample project has been created for you to write your own pass. A good initial step before writing your own pass is to compile the pass in the sample folder and check if it works. This will ensure that you do not have any environment bugs when you write your own pass. The example pass present in the lib folder called Hello prints out the name of every function present in the bytecode. Follow the steps below to get it running on your machine.

(1) First copy the sample directory present at /afs/glue/class/old/enee/759c/llvm/sample to your local directory, say ~/xyz/sample. You can use "cp -R" to copy a directory and its contents.

(2) Create two other directories at the same level (i) obj directory, say ~/xyz/obj and (ii) an install directory called opt at ~/xyz/opt

(3) Configure the build system in the obj directory as follows:

~/xyz/obj$ ../sample/configure --with-llvmsrc=/afs/glue/class/old/enee/759c/llvm/llvm-3.4.src --with-llvmobj=/afs/glue/class/old/enee/759c/llvm/llvm-3.4-install/obj --prefix=/homes/your-user-name/xyz/opt

(4) You can build it and test it out as follows:

~/xyz/obj$ make install

llvm[0]: Installing include files
make[1]: Entering directory `/afs/glue.umd.edu/home/glue/n/e/neroam/home/xyz/obj/lib'
make[2]: Entering directory `/afs/glue.umd.edu/home/glue/n/e/neroam/home/xyz/obj/lib/Hello'
llvm[2]: Compiling Hello.cpp for Release+Asserts build (PIC)
llvm[2]: Linking Release+Asserts Shared Library libHello.so
llvm[2]: Building Release+Asserts Archive Library libHello.a
llvm[2]: Installing Release+Asserts Shared Library /homes/neroam/xyz/opt/lib/libHello.so
llvm[2]: Installing Release+Asserts Archive Library /homes/neroam/xyz/opt/lib/libHello.a
make[2]: Leaving directory `/afs/glue.umd.edu/home/glue/n/e/neroam/home/xyz/obj/lib/Hello'
make[1]: Leaving directory `/afs/glue.umd.edu/home/glue/n/e/neroam/home/xyz/obj/lib'

[There would be many more lines ... only few reproduced here]

(5) Set the PATH variable to also have the llvm installation as follows:

~/xyz/obj: setenv PATH /afs/glue/class/old/enee/759c/llvm/llvm-3.4-install/opt/bin/:$PATH

Check that the installation is correct as follows

~/xyz/obj: opt --version
LLVM (http://llvm.org/):
  LLVM version 3.4
  Optimized build.
  Built Mar 17 2014 (12:45:22).
  Default target: x86_64-unknown-linux-gnu
  Host CPU: penryn

(6) You can run your pass as follows:

~/xyz/obj: opt -load ../opt/lib/libHello.so -hello
/afs/glue/class/old/enee/759c/llvm/test_codes/example1.bc -o example.1hello.bc

The above command will optimize the example1.bc bytecode with the "-hello" optimization and produce as output optimized bytecode in example1.hello.bc. This optimization is present in the libHello.so library.

[The output should look like]

Hello: main

It prints out the names of all functions present in this module.

Later when you write your own optimization, write it in the sample/lib directory. The compile and install it using steps (3) and (4) above. This should produce a new library with a .so extension. Then use the above command with the library name that you generated in place of libHello.so, and the name of your optimization in place of "-hello". This will apply your transformation to the bytecode file specified.

WRITING YOUR OWN PASS

In LLVM the transformations and optimizations are written as **Passes**. The documentation about an LLVM Pass and method to write it is here . The main sections that will be of interest to this project are

- Introduction
- Quick Start
- ModulePass
- FunctionPass
- AnalysisRequired

The example code present in the sample folder has an example code and Makefiles. The documentation in this section and the example code should help you in your project. You can add

a new pass to the sample code by creating a new directory in the lib folder and adding the name of this new folder to the Makefile in the lib folder.

The programmer's manual here gives an introduction to some important and useful API's in LLVM. A few sections of the programmer's manual that will be interest to this project are:

- Common Operations
- Core LLVM classes

These sections show methods to add and delete instructions in the input bytecode and also present methods for basic inspection and traversal routines.

The doxygen for the LLVM project is present at here.

STEPS INVOLVED

Below are helpful suggestions on how to implement some of the tasks in the project. You can choose to use them, but full credit will be given for other implementations, provided you write them yourself without assistance or copying code from elsewhere. Of course, you are allowed to look at code already in the LLVM compiler as examples.

1. Iterate over all instructions

First iterate over all instructions in the original code and for each instruction cast it to a CallInst, if this succeeds then you have an instruction that is of interest. The *CallInst* class is defined in http://llvm.org/doxygen/Instructions_8h_source.html. Some of functions of interest in this class are *getCalledFunction()* -- This will tell you which function is being called

2. Access the argument list of a function.

arg_begin() arg_end() iterators will help you to get to the arguments. This also is present in the CallInst class. Another way to access arguments is

unsigned getNumArgOperands () const --- gives number of operands
Value * getArgOperand (unsigned i) const -- will help you get the argument at index i

3. Check whether an actual argument is a constant.

To know if an object of type Value* (say V) is a constant, the following code can be used

```
if (isa<Constant>(V))
   return true;
```

4. Create a constant symbol of type Value

Create a new constant symbol in main memory with the value of the constant argument. To do this, use any suitable constructor in the **ConstantInt** class in LLVM.

For example, in the code above create a constant called *x* with the value of 3 in the main memory. /* Note this constant is not added to IR yet. That is fine. */

Use the get( ) constructor in ConstantInt class for this purpose. Below is the prototype,

static **ConstantInt** *	**get** (**IntegerType** *Ty, uint64_t V, **bool** isSigned=**false**)
	Return a **ConstantInt** with the specified integer value for the specified type.

5. Replace all uses of a formal argument with the constant symbol you created above.

First, the Function class defined in http://llvm.org/doxygen/Function_8h_source.html gives you capabilities to access the arguments of a function.

You can iterate over the arguments using *arg_begin() arg_end()* . You can get to objects of class type *Value* from these. This class is defined in

http://llvm.org/doxygen/Value_8h_source.html

Let the formal argument corresponding to the constant be constArg.

Value * constArg

Then, we can replace all uses of this argument with the constant you created in the previous step (x) as follows. We call *replaceAllUsesWith(Value* V)* function on the object constArg as shown below :

constArg -> replaceAllUsesWith(x)

In the example in the previous step, this will replace all uses of formal argument x with 3. Look at this for replaceAllUsesWith function prototype. It's also shown below:

void	**replaceAllUsesWith** (**Value** *V)
	Change all uses of this to point to a new **Value**.

Note that this step does NOT remove the argument, it just makes it dead by removing all uses to it.

6. Copy the modified instructions to the call-site.

You need to create and insert new instruction to the call-site. The following link gives an example.

http://llvm.org/docs/ProgrammersManual.html#creating-and-inserting-new-instructions

Then you need to clone each instruction separately into the new instruction that you created above. You may need to fix some references after cloning the instructions to the new location. The following is an example:

```
llvm::ValueToValueMapTy vmap;

for (auto *inst: instructions_to_clone) {
        auto *new_inst = inst->clone();
      new_inst->insertBefore(insertion_pos);
      vmap[inst] = new_inst;
      llvm::RemapInstruction(new_inst, vmap,
                          RF_NoModuleLevelChanges |
RF_IgnoreMissingLocals);
}
```

7. Remove the call instruction.

You can use the eraseFromParent() function to remove the call instruction. The following link can be helpful.

http://llvm.org/docs/ProgrammersManual.html#deleting-instructions

8. Constant propagate the constant local variables.

This step propagates the constant values in the local variables that you just created in step 4. The easiest way to do this is to use the pre-existing LLVM pass:

-*constprop:* This pass implements constant propagation and merging. It looks for instructions involving only constant operands and replaces them with a constant value instead of an instruction.

No need to write new LLVM code at all for this step! The constants will be propagated by themselves, for example, x will be replaced by 3 and y will be replaced by 7.

Run the *constprop* pass separately on the command line with the byte code obtained from running your pass as the input bytecode. Below is the command to do it.

opt --constprop < input bytecode > -o < output byte code >

No need to integrate the calling of constprop pass in your function in-lining pass code.

Some Useful Links In General

(a) A list of all classes present in llvm -- http://llvm.org/doxygen/classes.html

(b) A list of all files present in llvm -- http://llvm.org/doxygen/files.html

(c) Link to llvm's programmer manual -
- http://llvm.org/releases/3.4/docs/ProgrammersManual.html

Some subsections of interest here may be *cast templates, iterating over basic blocks in a function, iterating over instructions in a Basicblock and so on. Iterating over def-use chains, Creating and inserting new instructions, deleting instructions, replacing an instruction with another value, Creating types, the Value class, the Constant class, the Function class and so on.*

More Useful Links

- Writing an LLVM pass – Refer to the page titled iterating through Functions, Blocks, Instructions.