CS 380C Compilers — Spring 2014	Programming Assignment #3
Data-Flow Analysis	Due March 19, 2014

Introduction

We have seen in class how data-flow analyses share a common theoretical foundation, so it makes sense to write a generic framework that can be parameterized to solve specific data-flow analyses. In this assignment, you will implement such an iterative data-flow analysis framework in LLVM and use it to implement both a forward data-flow analysis (Reaching Definitions) and a backward data-flow analysis (Liveness). Liveness and Reaching Definitions implementations are already available in LLVM, but they are not of the iterative flavor.

Any clarifications and revisions to the assignment will be posted on Piazza.

1 Iterative Data Flow Analysis Framework

You will implement a general parameterized iterative data-flow analysis framework that allows a developer to easily implement any unidirectional data-flow analysis pass by providing the following information:

1. Domain, including the semi-lattice
2. Direction (forwards or backwards)
3. Transfer functions
4. Meet operator
5. Initial flow values (Top)
6. Boundary condition (flow value for entry node—or exit node for a backwards analysis)

In C++, two techniques are commonly used for implementing a generalized abstraction:

- Inheritance provides runtime subclass polymorphism, and in principle it works just like the inheritance you learned from other languages (e.g. Java, Ruby or Python): if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program. However, if you come from a Java background, you need to be aware that C++ implements the inheritance system slightly different from Java in the following ways:
  - Non-static Java methods are implicitly polymorphic and thus will get dynamically dispatched by default. On the other hand, non-static C++ methods are implicitly non-polymorphic and static dispatch is the default option. If you intend to use a C++ method polymorphically, you need to mark it as `virtual`.
  - C++ supports true multiple inheritance, while Java only has single inheritance with interfaces. Multiple inheritance works in a fairly twisted way and is arguably considered an anti-pattern in C++ programming, but if it really improves code reuses, or if you just want to find an alternative to Java interface, don't be afraid of using it.
  - The C++ counterpart of Java reflection is called "Run-Time Type Information" (RTTI) but its usage is strongly discouraged in developing LLVM-based applications for performance reasons[1]. If you find yourself tempted to write, say, Java `instanceof()`, use compile-time tag-based casting as described here: `http://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html`.
- C++ templates are a powerful technique that implement all kinds of compile-time polymorphisms. Just like Java generics, two common scenarios where templates come in handy are:

[1] In addition, RTTI will not even work properly in our virtual machine because both the skeleton codes and the LLVM library codes are compiled with "no-rtti" flag.

- You have multiple functions that perform exactly the same operation except that the type of their operands are different, and you want to write a generic function that work for those types to improve code reuse.
- You want to write a generic container class that works no matter what type its elements have.

Despite the syntactic similarity to generics, C++ templates are actually strictly more powerful and work in a fundamentally different way[2]. They serve essentially as a code rewriting system and resembles Lisp macros in functionality more than it resembles Java generics. We encourage the usage of template in this assignment if you are an experienced template programmer and know how to use it to write short and efficient codes.

In this assignment, we do not have hard requirement on how the analysis framework should be implemented except that it is abstracted and generic, and it should use a worklist. To give you an example, here is one possible implementation strategy:

- The analysis framework as a template base class from which all the analyses can derive.
- The operations, such as the meet operator, are pure virtual methods that will be implemented in the subclass.
- The type used to represent values from the domain is a type parameter for the template[3]
- Give a careful thought about how the other analysis parameters are represented. For example, *direction* could reasonably be represented as a boolean.

You should do a good job on this assignment because you will be reusing your framework in Assignment 4.

2 Data-Flow Analyses

You will now use your iterative data-flow analysis framework to implement Liveness and Reaching Definitions. If you wish to use a bit vector to represent the sets you should look at `llvm::BitVector`, `llvm::SparseBitVector`, or `std::bitset`; however, you can use a different data type if you think it is more appropriate. A utility Annotator pass that helps print out the analysis result will be posted later on piazza. Modify it to dump whatever information your analysis finds to stderr.

Liveness. On convergence, your Liveness pass should report for each program point all variables that are live at that point. You might debug your code by comparing it against the results of LLVM's Liveness pass. Please call this pass "live".

Reaching Definitions. On convergence, your Reaching Definitions pass should report for each program point, all the definition sites that reach that program point. Please call this pass "reach".

2.1 Implementation Issues[4]

LLVM's representation of Single Static Assignment (SSA) form presents some unique challenges when performing iterative data-flow analysis.

1. Values in LLVM are represented by the Value class. In SSA form, every value has a single definition, so instead of representing values as some distinct variable or pseudo register class, LLVM represents values by their *defining instruction*. That is, Instruction is a subclass of Value. There are other subclasses of Value, such as basic blocks, constants, and function arguments. For this assignment, we will only track the liveness of instruction-defined values and function arguments.

[2] A comprehensive list of the differences between C++ template and Java generics can be found here: `http://en.wikipedia.org/wiki/Comparison_of_Java_and_C%2B%2B#Templates_vs._generics`

[3] *If you do not know how to implement a template or how to use pure virtual methods, please ask for help on Piazza or come to the TA's office hours. The TA can help you, as can your fellow classmates.*

[4] Based on earlier editions of Todd Mowry's class, as conveyed by Seth Goldtein and David Koes.

```
int sum (int a, int b)
{
 int i;
 int res = 1;

 for (i = a; i < b; i++)
 {
    res *= i;
 }
 return res;
}
```

(a)

```
define i32 @sum(i32 %a, i32 %b) nounwind readnone ssp {
entry:
  %0 = icmp slt i32 %a, %b
  br i1 %0, label %bb.nph, label %bb2

bb.nph: ; preds = %entry
  %tmp = sub i32 %b, %a
  br label %bb

bb:     ; preds = %bb, %bb.nph
  %indvar = phi i32 [ 0, %bb.nph ], [ %indvar.next, %bb ]
  %res.05 = phi i32 [ 1, %bb.nph ], [ %1, %bb ]
  %i.04 = add i32 %indvar, %a
  %1 = mul nsw i32 %res.05, %i.04
  %indvar.next = add i32 %indvar, 1
  %exitcond = icmp eq i32 %indvar.next, %tmp
  br i1 %exitcond, label %bb2, label %bb

bb2:    ; preds = %bb, %entry
  %res.0.lcssa = phi i32 [ 1, %entry ], [ %1, %bb ]
  ret i32 %res.0.lcssa
```

(b)

Figure 1: (a) Simple loop code and (b) corresponding optimized (-O) LLVM assembly. Note that assembly you generate from this C code is likely to differ somewhat. Even different configurations of the same version of LLVM/Clang can give different output.

2. ϕ instructions are pseudo instructions that are used in the SSA representation and need to be handled specially by both Liveness and Reaching Definitions. Since SSA form requires that values have a unique definition at any program point (P), it is natural to wonder how you should handle a value that is live at P but has different definitions on the paths leading to it. The SSA solution is to introduce ϕ instructions at the beginning of the basic block containing P that "combine" the different definitions, so that all the uses in the block (including the one at P) see only the definition by the phi instruction. Consider the uses of ϕ instructions in Figure 1(b) as illustrations. You should carefully consider how your analysis passes are affected by ϕ instructions. For example, your passes should not output results for the program point preceding a phi instruction since they are pseudo instructions which will not appear in the executable. To guide you in formatting the output of your passes, the expected output of running Liveness analysis on the assembly from Figure 1(b) is shown in Figure 2.

3. The fact that you will be working on code in SSA form means that computed values are never destroyed. Think carefully about the ramifications of this fact on your implementation.

2.2 Banned parts of the LLVM API

Below is a list of APIs in LLVM that you are not allowed to use on this assignment. This list may not be complete: if you find any API that trivializes the assignment, please let us know (because it should probably be on this list). If you are in doubt as to whether an API is allowed, just ask.

In addition, do not copy any code from LLVM into your own source, and do not use any APIs that are private in the sense that they are not accessible through the LLVM header files.

Do **not** use any include files in these directories in the LLVM include tree:

- Analysis
- CodeGen

	define i32 @sum(i32 %a, i32 %b) nounwind readnone ssp {
	entry:
{**%a,%b**}	
	%0 = icmp slt i32 %a, %b
{**%a,%b,%0**}	
	br i1 %0, label %bb.nph, label %bb2
	bb.nph: ; preds = %entry
{**%a,%b**}	
	%tmp = sub i32 %b, %a
{**%a,%tmp**}	
	br label %bb
	bb: ; preds = %bb, %bb.nph
	%indvar = phi i32 [0, %bb.nph], [%indvar.next, %bb]
	%res.05 = phi i32 [1, %bb.nph], [%1, %bb]
{**%a,%tmp,%indvar,%res.05**}	
	%i.04 = add i32 %indvar, %a
{**%a,%tmp,%indvar,%res.05,%i.04**}	
	%1 = mul nsw i32 %res.05, %i.04
{**%a,%tmp,%indvar,%1**}	
	%indvar.next = add i32 %indvar, 1
{**%a,%tmp,%1,%indvar.next**}	
	%exitcond = icmp eq i32 %indvar.next, %tmp
{**%a,%tmp,%1,%indvar.next,%exitcond**}	
	br i1 %exitcond, label %bb2, label %bb
	bb2: ; preds = %bb, %entry
	%res.0.lcssa = phi i32 [1, %entry], [%1, %bb]
{**%res.0.lcssa**}	
	ret i32 %res.0.lcssa
{}	
	}

Figure 2: Output of Liveness on the assembly in Figure 1(b).

- ExecutionEngine
- Transforms

3 Submission

This assignment is due at 5:00pm on the due date. Submit via Canvas a single `tar.gz` or `tar.bz2` file that contains your source code in a directory with a `Makefile` that will build it. The `Makefile` should build two separate `so`'s: `live.so`, and `reach.so` (for your liveness and reaching analyses, respectively). Please name the directory `assignment3`. Make sure that your code builds correctly on the provided virtual machine and does not depend on any files outside the code you submit.

Acknowledgments. This assignment was originally created by Todd Mowry and then modified by Calvin Lin, Arthur Peters, and Jia Chen.