CS 380C Compilers — Spring 2014 **Programming Assignment #3**
Data-Flow Analysis **Due February 17, 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 an 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)

You should implement your analysis framework as a template base class from which all the analyses can derive. The operations, such as the meet operator, should be pure virtual methods that will be implemented in the subclass. The type used to represent values from the domain should be a type parameter for the template. *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. You should give careful thought about how the other analysis parameters are represented. For example, *direction* could reasonably be represented as a boolean. You should make your engine reasonably efficient, so you will want to use a work queue.

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`; however, you can use a different data type if you think it is more appropriate.

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".

```
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.

2.1 Implementation Issues[1]

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. Thus, when determining what values are used by an instruction, you will use code like this:

   ```
   User::op_iterator OI, OE;
   for (OI = insn->op_begin(), OE = insn->op_end(); OI != OE; ++OI)
   {
    Value *val = *OI;
    if (isa<Instruction>(val) || isa<Argument>(val)) {
      // val is used by insn
    }
   }
   ```

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. We will discuss SSA form in our next class, but a brief description is appropriate here, especially with regards to ϕ instructions. 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

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

```
                                     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).

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.

3 Submission

Use the `turnin` program to submit 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`. The turn-in command will be: `turnin --submit amp cs380c_assgn3 assignment3.tar.gz`. Make sure that your code builds correctly on the provided virtual machine and does not depend on any files outside the code you submit.

This assignment is due before class on the due date.

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