UG3 Compiling Techniques — Autumn 2018	Final Project
Iterative Liveness Analysis and Dead Code Elimination	Due January 14, 2019

Introduction

In this assignment, you will implement an iterative data-flow analysis framework in LLVM for a backward data-flow analysis (Liveness). A liveness implementation is already available in LLVM, but it is not of the iterative flavor. You will then use your framework to implement dead code elimination. Any clarifications and revisions to the assignment will be posted on Piazza.

1 Iterative Liveness Analysis

You will implement iterative Liveness analysis. 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. In this assignment, we do not have a hard requirement on how the analysis framework should be implemented except that it should use a worklist.

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

```
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.

```
                                    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).

1.1 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.

2. ϕ instructions are pseudo instructions that are used in the SSA representation and need to be handled specially by Liveness analysis. 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 pass is affected by ϕ instructions. For example, your pass 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.

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

1.2 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
- ExecutionEngine
- Transforms

2 Dead Code Elimination

Use your liveness analysis pass to determine dead code and eliminate the dead instructions. Replace the isInstruction-TriviallyDead() method from LLVM with your own method to identify dead code.

3 Submission

This assignment is due at 10:00am on the due date. Submit via Gitlab TODO file that contains your source code in a directory with a `Makefile` that will build it. The `Makefile` should build a `live.so` for your liveness analysis. Please name the directory `TODO`. Make sure that your code builds correctly on DICE and does not depend on any files outside the code you submit.

Acknowledgments. This assignment was originally created by Todd Mowry at CMU and then modified by Calvin Lin, Arthur Peters, and Jia Chen at the University of Texas at Austin. The current document was revised by Aaron Smith at the University of Edinburgh.