Dead Code Elimination (DCE)

- Dead code elimination is an optimization that removes DEAD variables
- A variable that is defined and not LIVE OUT is DEAD

```java
do {
    computeLiveness()
    foreach instruction I {
        if (defs.contains(I) && !out.contains(I))
            remove(I)
    }
} while(changed)
```
DCE Example

```c
int foo(int x, int y) {
    int a = x + y;
    a = 1;
    return a;
}
```

```llvm
define i32 @foo(i32 %x, i32 %y) #0 {
  entry:
  %add = add nsw i32 %x, %y
  ret i32 1
}
```

```bash
clang -emit-llvm -S dead.c -Xclang -disable-O0-optnone

opt dead.ll -S -mem2reg
```

```llvm
define i32 @foo(i32 %x, i32 %y) #0 {
  entry:
  %add = add nsw i32 %x, %y
  ret i32 1
}
```

```bash
opt dead.ll -S -mem2reg -dce
```

UG3 Compiling Techniques
DCE and Memory References

• A “dead store” might be clearing out sensitive information (a password for example) or writing to a device
  • store 0xffff1000

• DCE cannot remove the store to the global variable ‘b’, therefore it cannot remove the assignment to ‘a’

```c
int b;

int foo(int x, int y) {
    int a = x + y;
    b = a;
    return x;
}

@b = common global i32 0, align 4
define i32 @foo(i32 %x, i32 %y) #0 {
entry:
    %add = add nsw i32 %x, %y
    store i32 %add, i32* @b, align 4
    ret i32 %x
}
```
DCE and Volatile Variables

• Volatile is a way (by convention) to tell the compiler not to optimize an expression and to keep it in memory (on the stack or in the heap)
  • volatile int addr = 0xffff1000;

• The compiler does not know why the programmer declared the variable volatile and must be conservative

• Common reasons are memory mapped I/O, i.e. devices (keyboard, mouse, LEDs, etc), explicit type conversions, multi-threading, and to work around bugs in the compiler or software
Volatile Example

- Volatile variables will appear as “store volatile” and “load volatile” in LLVM IR
- Be careful not to eliminate volatile variables in your pass!
  - llvm::Instruction::mayHaveSideEffects()

```c
int b;

int foo(int x, int y) {
    volatile int a = x + y;
    b = a;
    return x;
}
```

```ll
@b = common global i32 0, align 4
define i32 @foo(i32 %x, i32 %y) #0 {
  entry:
  %a = alloca i32, align 4
  %add = add nsw i32 %x, %y
  store volatile i32 %add, i32* %a, align 4
  %0 = load volatile i32, i32* %a, align 4
  store i32 %0, i32* @b, align 4
  ret i32 %x
}
```

UG3 Compiling Techniques
DCE and Control Flow

- Programmers (and compiler optimizations!) often introduce useless control flow (branching)
- DCE only removes variables; removing unnecessary control flow is called “flow optimization”
  - The opt ‘-simplifycfg’ option will cleanup the control flow

```c
int foo(int x, int y) {
    int a = x + y;
    if (x > 0)
        a = 1;
    return y;
}
```

```llvm
define i32 @foo(i32 %x, i32 %y) #0 {
  entry:
  %cmp = icmp sgt i32 %x, 0
  br i1 %cmp, label %if.then, label %if.end

  if.then:
  br label %if.end

  if.end:
  ret i32 %y
}
```
Eliminating Dead Code in LLVM

• Look at instruction.def for the possible instructions
  • https://github.com/llvm-mirror/llvm/blob/master/include/llvm/IR/Instruction.def

• Do not remove
  • control flow (ReturnInst, SwitchInst, BranchInst, IndirectBrInst, CallInst)
    • llvm::Instruction::IsTerminator()
  • stores (StoreInst)
    • llvm::Instruction::mayHaveSideEffects()

• Do remove
  • AllocInst, LoadInst, GetElementPtrInst
  • SelectInst, ExtractElementInst, InsertElementInst, ExtractValue, InsertValue
  • binary instructions
  • comparisons
  • casts

• How to eliminate dead code?
  • Compute the OUT set for each instruction (liveness)
  • If the virtual register defined by an instruction is not in the OUT set, remove it!
    • llvm::Instruction::eraseFromParent()
  • Iterate until there are no changes
The Instruction class in LLVM inherits from the Value class.

Iterating over instructions in a function/basic block returns a Value*

How do you know which type of instruction you are looking at?
  
  isa<Type>(Value)

```c
#include "llvm/IR/Instructions.h"
for(inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I)
    if (isa<CallInst>(*I))
        outs() << “Found a call: “ << *I << “\n”;
```
Removing Instructions

• You cannot change an iterator while iterating over it
• To remove instructions, first collect the instructions to remove, then remove them in a separate pass
• What does this example do?

```c++
#include "llvm/ADT/SmallVector.h"

SmallVector<Instruction*,128> WL;
for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I)
    if (isa<CallInst>(*I))
        WL.push_back(&*I);

while (!WL.empty()) {
    Instruction* I = WL.pop_back_val();
    I->eraseFromParent();
}
```

UG3 Compiling Techniques
Computing Liveness Iteratively

• A variable is LIVE at some point in the program if it’s used in the future; otherwise it’s DEAD

• Liveness is a backwards flow problem
  • You need to propagate values from OUT to IN

• Compute the IN/OUT sets for each instruction
  • GEN = source operand(s)
  • KILL = destination operand(s)
  • IN(s) = GEN(s) U (OUT(s) – KILL(s))
  • OUT(s) = U of s’ successors IN(s’)

• You will need to handle PHIs to properly compute these sets
What’s a PHI?

• A PHI is pseudo instruction (it does not exist) used to make reasoning about backwards flow easier, i.e. def-use chains
  • $X = \text{PHI}(X', X'', X''', \ldots)$

• There is a source operand ($X', \ldots$) for each incoming edge in the flow graph that represents the value of $X$ on that path in the flow graph

• PHIs always appear at the beginning of a basic block before other instructions

• The compiler must remove PHIs before the program is executable, which usually means inserting a MOV (copy) into the predecessor BB

• Often copy propagation is run after PHI elimination to clean up any redundant/useless copies
PHIs in LLVM

• A PHI will only exist at a join in the flow graph

```c
int foo(int x, int y) {
    int a = x + y;
    if (x > 0)
        a = 1;
    return a;
}
```

```llvm
define i32 @foo(i32 %x, i32 %y) #0 {
  entry:
      %add = add nsw i32 %x, %y
      %cmp = icmp sgt i32 %x, 0
      br i1 %cmp, label %if.then, label %if.end

  if.then:
      br label %if.end

  if.end:
      %a.0 = phi i32 [ 1, %if.then ], [ %add, %entry ]
      ret i32 %a.0
}
```