Transformation from SSA Form

Proposal
- Restore original variable names (i.e., drop subscripts)
- Delete all $\phi$-functions

Complications
- What if versions get out of order?
  (simultaneously live ranges)

Alternative
- Perform dead code elimination (to prune $\phi$-functions)
- Replace $\phi$-functions with copies in predecessors
- Rely on register allocation coalescing to remove unnecessary copies

$x_0 = x_1 = x_0 = x_1$
Revisiting Data-flow Analyses in SSA Form

How do our various data-flow analyses change, if at all?
- Liveness
- Available expressions
- Common sub-expression elimination
- Reaching definitions

LLVM

Partial SSA form
- Top-level variables are in SSA from
- Address-taken variables are not

Why does LLVM do this?
- Pointers are difficult in SSA form

\[ *p := \ldots \]
\[ p := \ldots \]
\[ ? := f(?) \]
SSA in LLVM

SSA form
- Everything that starts with % or @ is in SSA
- Once initialized, they become immutable
- Every definition has to dominate all of its uses
- “Virtual register” or “top-level variable”
- Use fresh names %1, %2 rather than subscripts to rename variables

```
int f(int a)
{
    a = a * 2;
    a = a + 1;
    return a;
}

define i32 @f(i32 %a) {
    %1 = mul nsw i32 %a, 2
    %2 = add nsw i32 %a, 1
    ret i32 %2
}
```

LLVM (cont)

ϕ-function in LLVM
- Both the incoming value and the incoming block have to be specified

Example

```
; <label>: bb1
...

; <label>: bb2
%1 = ...

; <label>: bb3
%2 = ...

; <label>: bb4
%3 = phi i32 [ %1, %bb2], [%2, %bb3]
...%3...
```
mutable variables
– Partial SSA
  – Values that reside in memory are “address-taken variables”
  – These variables are mutable

memory operations
– Alloca for stack allocation
– Load & store for memory read/write

Example

```c
int f()
{
  int a = 5;
  int b = a - 3;
  b = 42;
  return b;
}
```

```llvm
define i32 @f() {
  %a = alloca i32
  %b = alloca i32
  store i32 5, %i32* %a
  %1 = load i32* %a
  %2 = sub nsw i32 %1, 3
  store i32 %2, %i32* %b
  store i32 42 i32* %b
  %3 = load i32* %b
  ret i32 %3
}
```

In the IR, what do variable a and b in the source language get translated to?
In C/C++, every variable is mutable

- By default, clang will generate one memory allocation for every single C variable
- But this is wasteful!

Mem2reg to the rescue

- LLVM has a highly-tuned optimization pass called “mem2reg”
  - promotes allocas into virtual registers
  - inserts φ-node as appropriate
- This pass is essentially a dominance-frontier finder already written for you

Mem2reg example

Before
```assembly
define i32 @f() {
  %a = alloca i32
  %b = alloca i32
  store i32 5, %i32* %a
  %1 = load i32* %a
  %2 = sub nsw i32 %1, 3
  store i32 %2, %i32* %b
  store i32 42 i32* %b
  %3 = load i32* %b
  ret i32 %3
}
```

After
```assembly
define i32 @f() {
  %1 = sub nsw i32 5, 3
  ret i32 42
}
LLVM (cont)

Summary
- In LLVM, values can be stored in virtual registers or memory
- Virtual register values are required to be in SSA form
- Memory values can be mutable
- Use --mem2reg to eliminate unnecessary allocations

Concepts

Data dependences
- Three kinds of data dependences
  - du-chains
Alternate representations
SSA form
Conversion to SSA form
- $\phi$-function placement
  - Dominance frontiers
- Variable renaming
  - Dominance trees
Conversion from SSA form
LLVM and partial SSA form
Next Time

**Lecture**
- Reuse optimizations

**Reading**
- Wegman and Zadeck paper due Tuesday February 24th at 5:00pm