Lecture 26: IL for Arrays, Local Optimization

[Adapted from notes by R. Bodik and G. Necula]

Generating Intermediate Language (IL) Code

- For this lecture, let's assume a function—called *cgen*—that converts ASTs (denoted by program fragments) into IL code:

```
cgen (E, R):
    """Generate IL code that evaluates E and puts
    the result (if any) into virtual register R."""
```

- We'll use the C notations `&V` to denote the address of entity `V`, and `*T` to denote the contents of memory whose address is `T`.
- We'll use `t0`, `t1`, etc., to denote virtual registers. If undeclared, assume they are freshly generated virtual registers.
- Finally, we'll use the notation "$\Rightarrow C$" where C is IL to mean "output code C".

One-dimensional Arrays

- How do we process retrieval from and assignment to `x[i]`, for an array `x`?
- We assume that all items of the array have fixed size—S bytes—and are arranged sequentially in memory (the usual representation).
- Easy to see that the address of `x[i]` must be

  $$\&x + S \cdot i,$$

  where `&x` is intended to denote the address of the beginning of `x`.
- Generically, we call such formulae for getting an element of a data structure *access algorithms.*
- The IL might look like this:

  cgen(&A[E], t_0):
  - cgen(&A, t_1)
  - cgen(E, t_2)
  - $\Rightarrow t_3$:= t_2 * S
  - $\Rightarrow t_0$:= t_1 + t_3

Multi-dimensional Arrays

- A 2D array is a 1D array of 1D arrays.
- Java uses arrays of pointers to arrays for >1D arrays.
- But if row size constant, for faster access and compactness, may prefer to represent an MxN array as a 1D array of 1D rows (not pointers to rows): *row-major order*. . .
- Or, as in FORTRAN, a 1D array of 1D columns: *column-major order.*
- So apply the formula for 1D arrays repeatedly—first to compute the beginning of a row and then to compute the column within that row:

  $$\&A[i][j] = \&A + i \cdot S \cdot N + j \cdot S$$

  for an `M`-row by `N`-column array, where `S`, again, is the size of an individual element.

IL for $M \times N$ 2D array

```
cgen(&e1[e2,e3], t):
    cgen(e1, t1); cgen(e2,t2); cgen(e3,t3)
    cgen(N, t4) # (N need not be constant)
    ⇒ t5 := t4 * t2
    ⇒ t6 := t5 + t3
    ⇒ t7 := t6 * S
    ⇒ t := t7 + t1
```

Array Descriptors

- Calculation of element address `&e1[e2,e3]` has the form

  $VO + S1 \times e2 + S2 \times e3$

  , where

  - VO (`&e1[0,0]`) is the *virtual origin*.
  - `S1` and `S2` are *strides*.
  - All three of these are constant throughout the lifetime of the array (assuming arrays of constant size).

- Therefore, we can package these up into an *array descriptor,* which can be passed in lieu of the array itself, as a kind of "*fat pointer*" to the array:

`&e1[0][0]`	S×N	S

Array Descriptors (II)

- Assuming that e1 now evaluates to the address of a 2D array descriptor, the IL code becomes:

```
cgen(&e1[e2,e3], t):
    cgen(e1, t1); cgen(e2,t2); cgen(e3,t3)
    ⇒ t4 := *t1;      # The VO
    ⇒ t5 := *(t1+4)   # Stride #1
    ⇒ t6 := *(t1+8)   # Stride #2
    ⇒ t7 := t5 * t2
    ⇒ t8 := t6 * t3
    ⇒ t9 := t4 + t7
    ⇒ t10:= t9 + t8
```

Array Descriptors (III)

- By judicious choice of descriptor values, can make the same formula work for different kinds of array.
- For example, if lower bounds of indices are 1 rather than 0, must compute address

  `&e[1,1] + S1 × (e2-1) + S2 × (e3-1)`

- But some algebra puts this into the form

  `VO' + S1 × e2 + S2 × e3`

  where

  `VO' = &e[1,1] - S1 - S2 = &e[0,0]` *(if it existed).*

- So with the descriptor

VO'	S×N	S

we can use the same code as on the last slide.

Observation

- These examples show profligate use of registers.
- Doesn't matter, because this is Intermediate Code. Rely on later optimization stages to do the right thing...
- ...As we'll start discussing next.

Introduction to Code Optimization

Code optimization is the usual term, but is grossly misnamed, since code produced by "optimizers" is not optimal in any reasonable sense. *Program improvement* would be more appropriate.

Topics:

- Basic blocks
- Control-flow graphs (CFGs)
- Algebraic simplification
- Constant folding
- Static single-assignment form (SSA)
- Common-subexpression elimination (CSE)
- Copy propagation
- Dead-code elimination
- Peephole optimizations

Basic Blocks

- A *basic block* is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except in the last instruction)
- Idea:
  - Cannot jump into a basic block, except at the beginning.
  - Cannot jump within a basic block, except at end.
  - Therefore, each instruction in a basic block is executed after all the preceding instructions have been executed

Basic-Block Example

- Consider the basic block

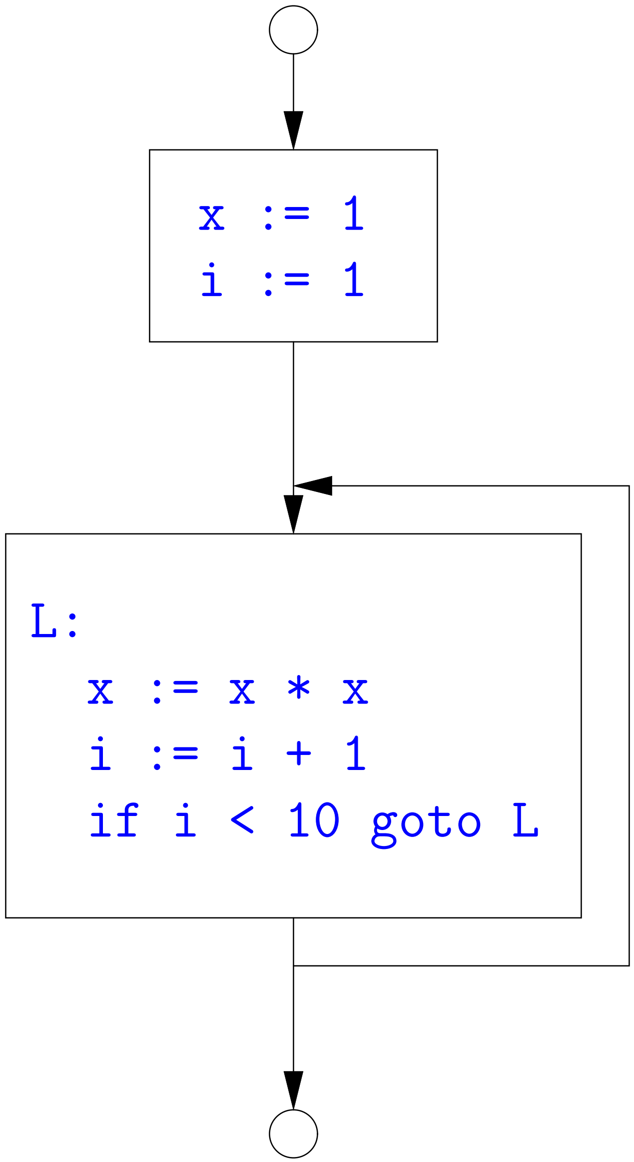
```
1. L1:
2. t := 2 * x
3. w := t + x
4. if w > 0 goto L2
```

- No way for (3) to be executed without (2) having been executed right before
- We can change (3) to `w := 3 * x`
- Can we eliminate (2) as well?

Control-Flow Graphs (CFGs)

- A control-flow graph is a directed graph with basic blocks as nodes
- There is an edge from block A to block B if the execution can flow from the last instruction in A to the first instruction in B:
  - The last instruction in A can be a jump to the label of B.
  - Or execution can fall through from the end of block A to the beginning of block B.

Control-Flow Graphs: Example



- The body of a method (or procedure) can be represented as a CFG
- There is one initial node
- All "return" nodes are terminal

Optimization Overview

- Optimization seeks to improve a program's utilization of some resource:
  - Execution time (most often)
  - Code size
  - Network messages sent
  - Battery power used, etc.
- Optimization should not depart from the programming language's semantics
- So if the semantics of a particular program is deterministic, optimization must not change the answer.
- On the other hand, some program behavior is undefined (e.g., what happens when an unchecked rule in C is violated), and in those cases, optimization may cause differences in results.

A Classification of Optimizations

- For languages like C and Java there are three granularities of optimizations
  1. *Local optimizations:* Apply to a basic block in isolation.
  2. *Global optimizations:* Apply to a control-flow graph (single function body) in isolation.
  3. *Inter-procedural optimizations:* Apply across function boundaries.
- Most compilers do (1), many do (2) and very few do (3)
- Problem is expense: (2) and (3) typically require superlinear time. Can usually handle that when limited to a single function, but gets problematic for larger program.
- In practice, generally *don't* implement fanciest known optimizations: some are hard to implement (esp., hard to get right), some require a lot of compilation time.
- The goal: maximum improvement with minimum cost.

Local Optimizations: Algebraic Simplification

- Some statements can be deleted

```
x := x + 0
x := x * 1
```

- Some statements can be simplified or converted to use faster operations:

Original	Simplified
`x := x * 0`	`x := 0`
`y := y ** 2`	`y := y * y`
`x := x * 8`	`x := x << 3`
`x := x * 15`	`t := x << 4; x := t - x`

(on some machines << is faster than $*$; but not on all!)

Local Optimization: Constant Folding

- Operations on constants can be computed at compile time.
- Example: `x := 2 + 2` becomes x := 4.
- Example: if 2 < 0 jump L becomes a no-op.
- When might constant folding be dangerous?

Global Optimization: Unreachable code elimination

- Basic blocks that are not reachable from the entry point of the CFG may be eliminated.
- Why would such basic blocks occur?
- Removing unreachable code makes the program smaller (sometimes also faster, due to instruction-cache effects, but this is probably not a terribly large effect.)

Single Assignment Form

- Some optimizations are simplified if each assignment is to a temporary that has not appeared already in the basic block.
- Intermediate code can be rewritten to be in *(static) single assignment (SSA) form:*

`x := a + y`	`x := a + y`
`a := x`	`a1 := x`
`x := a * x`	`x1 := a1 * x`
`b := x + a`	`b := x1 + a1`

where `x1` and `a1` are fresh temporaries.

Common SubExpression (CSE) Elimination in Basic Blocks

- A *common subexpression* is an expression that appears multiple times on a right-hand side in contexts where the operands have the same values in each case (so that the expression will yield the same value).
- Assume that the basic block on the left is in single assignment form.

```
x := y + z                    x := y + z
              ...
              ...
w := y + z                    w := x
```

- That is, if two assignments have the same right-hand side, we can replace the second instance of that right-hand side with the variable that was assigned the first instance.
- How did we use the assumption of single assignment here?

Copy Propagation

- If `w := x` appears in a block, can replace all subsequent uses of `w` with uses of `x`.
- Example:

`b:=z+y`	`b:=z+y`
`a := b`	`a := b`
`x:=2*a`	`x:=2*b`

- This does not make the program smaller or faster but might enable other optimizations. For example, if `a` is not used after this statement, we need not assign to it.
- Or consider:

`b:=13`	`b:=13`
`x:=2*a`	`x:=2*13`

  which immediately enables constant folding.

- Again, the optimization, as described, won't work unless the block is in single assignment form.

Another Example of Copy Propagation and Constant Folding

```
a := 5         a := 5         a := 5         a := 5          a := 5
x := 2 * a     x := 2 * 5     x := 10        x := 10         x := 10
y := x + 6     y := x + 6     y := 10 + 6    y := 16         y := 16
t := x * y     t := x * y     t := 10 * y    t := 10 * 16    t := 160
```

Dead Code Elimination

- If that statement `w := rhs` appears in a basic block and `w` does not appear anywhere else in the program, we say that the statement is *dead* and can be eliminated; it does not contribute to the program's result.
- Example: (`a` is not used anywhere else)

```
x := z + y        b := z + y    b := z + y
a := x            a := b
x := 2 * a        x := 2 * b    x := 2 * b
```

- How have I used SSA here?

Applying Local Optimizations

- As the examples show, each local optimization does very little by itself.
- Typically, optimizations interact: performing one optimization enables others.
- So typical optimizing compilers repeatedly perform optimizations until no improvement is possible, or it is no longer cost effective.

An Example: Initial Code

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example II: Algebraic simplification

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example II: Algebraic simplification

```
a := x * x
b := 3
c := x
d := c * c
e := b + b
f := a + d
g := e * f
```

An Example: Copy propagation

```
a := x * x
b := 3
c := x
d := c * c
e := b + b
f := a + d
g := e * f
```

An Example: Copy propagation

```
a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
f := a + d
g := e * f
```

An Example: Constant folding

```
a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
f := a + d
g := e * f
```

An Example: Constant folding

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example: Common Subexpression Elimination

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example: Common Subexpression Elimination

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example: Copy propagation

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example: Copy propagation

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

An Example: Dead code elimination

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

An Example: Dead code elimination

```
a := x * x
```

```
f := a + a
g := 6 * f
```

This is the final form.

Peephole Optimizations on Assembly Code

- The optimizations presented before work on intermediate code.
- *Peephole optimization* is a technique for improving assembly code directly
  - The "*peephole*" is a short subsequence of (usually contiguous) instructions, either continguous, or linked together by the fact that they operate on certain registers that no intervening instructions modify.
  - The optimizer replaces the sequence with another equivalent, but (one hopes) better one.
  - Write peephole optimizations as replacement rules

    i1; . . . ; in ⇒ j1; . . . ; jm

    possibly plus additional constraints. The j's are the improved version of the i's.

Peephole optimization examples:

- We'll use the notation '@A' for pattern variables.
- Example:

  movl %@a %@b; L: movl %@b %@a $\Rightarrow$ movl %@a %@b

  assuming L is not the target of a jump.
- Example:

  addl $@k1, %@a; movl @k2(%@a), %@b
  $\Rightarrow$ movl @k1+@k2(%@a), %@b

  assuming %@a is "dead".
- Example (PDP11):

  mov #@I, @I(@ra) $\Rightarrow$ mov (r7), @I(@ra)

  This is a real hack: we reuse the value I as both the immediate value and the offset from ra. On the PDP11, the program counter is r7.
- As for local optimizations, peephole optimizations need to be applied repeatedly to get maximum effect.

Problems:

- Serious problem: what to do with pointers? Problem is *aliasing:* two names for the same variable:
  - As a result, `*t` may change even if local variable `t` does not and we never assign to `*t`.
  - Affects language design: rules about overlapping parameters in Fortran, and the **restrict** keyword in C.
  - Arrays are a special case (address calculation): is `A[i]` the same as `A[j]`? Sometimes the compiler can tell, depending on what it knows about `i` and `j`.
- What about globals variables and calls?
  - Calls are not exactly jumps, because they (almost) always return.
  - Can modify global variables used by caller