CS 4240: Compilers

Lecture 2: Control Flow Graphs, Reaching Definitions

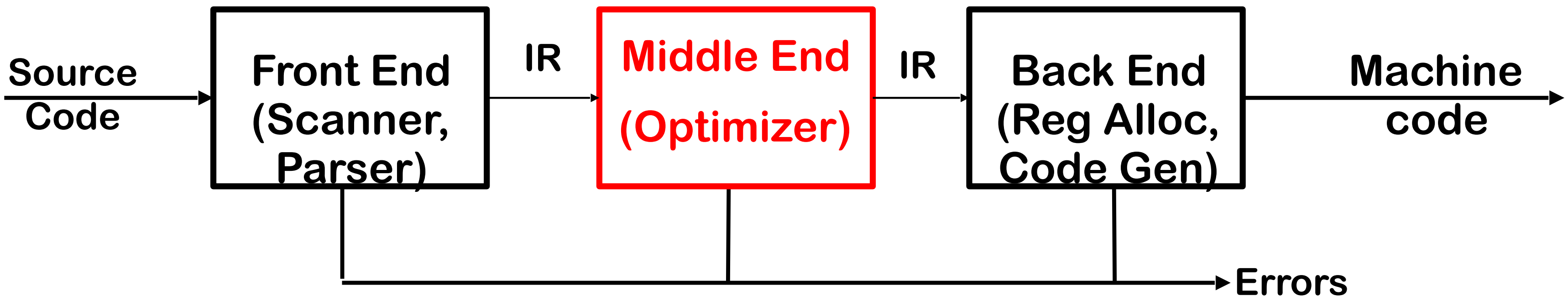
Instructor: Vivek Sarkar (vsarkar@gatech.edu)

January 7, 2019

Course Announcements

- Ensure that you can access the course Piazza site
  - http://piazza.com/gatech/spring2019/cs4240a
- There will be 3 homeworks and 3 projects during the semester
  - Release and due dates to be announced this week on Piazza
- Forming project teams
  - For the 3 projects in this course, you will be working in teams of 2-3 people. We will provide some helper code for people implementing projects in Java, but you are welcome to use a different language if you prefer.
  - We will create a 0-point (pseudo) assignment in Canvas for you to report your team members and implementation language.
  - You can use "Search for Teammates!" in Piazza if needed.

Traditional Three-pass Compiler (Recap)



Middle End

- Analyzes IR and rewrites (or <u>transforms</u>) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - Measured by values of named variable
- Can also be used to produce static code analysis reports that go beyond programming language errors, e.g., API misuse, security vulnerabilities, …

Dead code elimination (Recap)

DEAD

- Conceptually similar to mark-sweep garbage collection
  - Mark useful operations
  - Everything not marked is useless
- Need an efficient way to find and to mark useful operations
  - Start with critical operations
  - Work back up data flow edges to find their antecedents

Define critical operations

- I/O statements, linkage code (entry & exit blocks), return values, calls to other procedures
- Global variables that can be visible on program exit

Simple Dead-code elimination algorithm (Recap)

Mark

```
for each op i
    clear i's mark
    if i is critical then
        mark i
        add i to WorkList

while (Worklist ≠ Ø)
    remove i from  WorkList
        (i has form "x←y op z" )
    for each instruction j that
    writes to y or z
      if j is not marked then
        mark j
        add j to WorkList
```

Sweep

```
for each op i
    if i is not marked then
        delete i
```

NOTES:
1) Not all instructions that write to y or z need to be marked. We can only focus on "reaching definitions" (next lecture).
2) Branch instructions need special handling in general. A simple approach is to mark all branch instructions as critical. See textbook for more sophisticated approaches.

Improved Dead-code elimination algorithm #1

Mark

```
for each op i
    clear i's mark
    if i is critical then
        mark i
        add i to WorkList

while (Worklist ≠ Ø)
    remove i from  WorkList
        (i has form "x←y op z" )
    for each instruction j that
    writes to y (or z), and is not
    followed by a subsequent
    write of y (or z) before i
      if j is not marked then
        mark j
        add j to WorkList
```

Sweep

```
for each op i
    if i is not marked then
        delete i
```

NOTES:

1) This is simple to do if there is a "straight line" control from instruction j to i, with no intervening branch instructions
2) Identifying minimum set of instructions j that contribute to inputs of instruction i is more complicated in the presence of control flow ==> ***need to build control flow graph***

Control Flow Graphs: Motivation

- **Control flow** pertains to transfer of execution across program statements/instructions
  - Default execution is sequential
  - Can be altered by unconditional/conditional branch instructions and call instructions in intermediate code and machine code
- **Control flow graphs** fill the need for an abstract representation of control flow in programs. They also enable the use of several important algorithms from graph theory in compilers.

What is control flow?

A program is a sequence of instructions that are executed in order from top to bottom. However, we can interrupt this sequence of execution through special *control* instructions which move executions somewhere else.

- branches
- loops
- gotos
- etc.

We call the possible pattern of executions of the program its **control flow**.

What is a control flow graph?

We can capture the control flow of a single procedure in a directed graph called a **Control Flow Graph** (CFG).

In these graphs, vertices contain **basic blocks** (sequences of instructions), while edges indicate control flow from one basic block to another.

Why do we care about control flow graphs?

- CFGs abstract away the different control flow mechanisms in the language, leaving only the single control mechanism represented by graph edges.
- CFGs represent programs as graphs, so we can call upon our general knowledge of graphs to manipulate programs.
- CFGs offer a *visual* presentation of a program, which can be a useful tool for understanding.

What is a basic block?

The vertices of CFGs are labelled with basic blocks. A **Basic Block** is a contiguous sequence of program instructions such that if the first instruction in the block is executed, so are the rest.

- It is common for compilers to choose basic blocks that are as large as possible, for efficiency reasons (since doing so leads to smaller CFGs)
- However, all the algorithms we study will also be applicable to instruction-level basic blocks, i.e., when each CFG vertex corresponds to one instruction.
- Instruction-level CFGs may also be more convenient than maximal basic blocks in your project implementations.

Building a CFG

The basic idea is to:

- determine where all the basic blocks are (based on control instructions)
- add a vertex for each basic block
- add the appropriate instructions to each basic block
- draw edges between the vertices (based on control instructions)

Building a CFG

We can translate a sequence of instructions to a CFG with an efficient algorithm.

But first we need to know what the **Leaders** are. Every basic block has a leader, which is the first instruction. An instruction is a leader if it is:

1. the first instruction in the procedure
2. the target of a **goto** or **branch** instruction.
3. the successor of a **branch** instruction.

Leaders: Example

```
1:  SEARCH'(arr, size, n)
2:      i = −1
3:      out = −1
4:      branch (i ≥ size) 11
5:      i = i + 1
6:      v = arr[i]
7:      branch (v ≠ n) 10
8:      out = i
9:      goto 11
10:     goto 4
11:     return out
```

Leaders: Example

```
1:  SEARCH'(arr, size, n)
2:      i = −1                      ▷ {2}
3:      out = −1
4:      branch (i ≥ size) 11
5:      i = i + 1
6:      v = arr[i]
7:      branch (v ≠ n) 10
8:      out = i
9:      goto 11
10:     goto 4
11:     return out
```

Leaders: Example

```
1:  SEARCH'(arr, size, n)
2:      i = −1                          ▷ {2}
3:      out = −1
4:      branch (i ≥ size) 11            ▷ {5, 11}
5:      i = i + 1
6:      v = arr[i]
7:      branch (v ≠ n) 10
8:      out = i
9:      goto 11
10:     goto 4
11:     return out
```

Leaders: Example

```
 1: SEARCH'(arr, size, n)
 2:     i = −1                         ▷ {2}
 3:     out = −1
 4:     branch (i ≥ size) 11           ▷ {5, 11}
 5:     i = i + 1
 6:     v = arr[i]
 7:     branch (v ≠ n) 10              ▷ {8, 10}
 8:     out = i
 9:     goto 11
10:     goto 4
11:     return out
```

Leaders: Example

```
1:  Search'(arr, size, n)
2:      i = −1                          ▷ {2}
3:      out = −1
4:      branch (i ≥ size) 11            ▷ {5, 11}
5:      i = i + 1
6:      v = arr[i]
7:      branch (v ≠ n) 10               ▷ {8, 10}
8:      out = i
9:      goto 11                         ▷ {11}
10:     goto 4
11:     return out
```

Leaders: Example

1: $\textsc{Search'}(arr, size, n)$
2: $\quad i = -1$ $\qquad \triangleright \{2\}$
3: $\quad out = -1$
4: $\quad$ **branch** $(i \geq size)$ 11 $\qquad \triangleright \{5, 11\}$
5: $\quad i = i + 1$
6: $\quad v = arr[i]$
7: $\quad$ **branch** $(v \neq n)$ 10 $\qquad \triangleright \{8, 10\}$
8: $\quad out = i$
9: $\quad$ **goto** 11 $\qquad \triangleright \{11\}$
10: $\quad$ **goto** 4 $\qquad \triangleright \{4\}$
11: $\quad$ **return** *out*

Leaders: Example

1: $\textsc{Search'}(\mathit{arr}, \mathit{size}, n)$
2: $\quad i = -1$ $\qquad \triangleright \{2\}$
3: $\quad \mathit{out} = -1$
4: $\quad$ **branch** $(i \geq \mathit{size})$ 11 $\qquad \triangleright \{5, 11\}$
5: $\quad i = i + 1$
6: $\quad v = \mathit{arr}[i]$
7: $\quad$ **branch** $(v \neq n)$ 10 $\qquad \triangleright \{8, 10\}$
8: $\quad \mathit{out} = i$
9: $\quad$ **goto** 11 $\qquad \triangleright \{11\}$
10: $\quad$ **goto** 4 $\qquad \triangleright \{4\}$
11: $\quad$ **return** *out*

$$\{2, 4, 5, 8, 10, 11\}$$

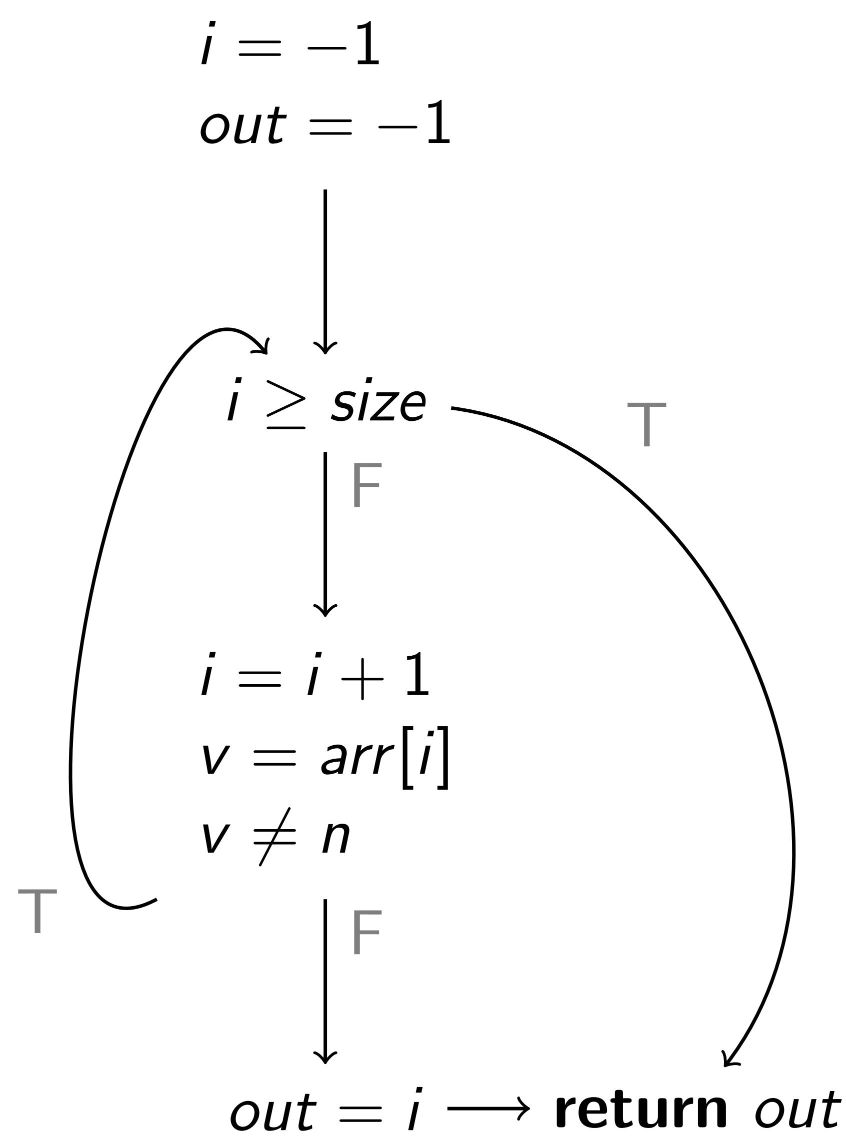
Building a CFG: Algorithm

```
MkCFG(is)
    Add a fresh vertex for each leader to the graph
    curr = null
    for i ∈ is do
        if i is a leader then
            if curr is not null then
                Add an edge from curr to vertex with leader i
            curr = vertex with leader i
        if i is a goto with target t then
            Add an edge from curr to t
        else if i is a branch with condition c and target t then
            Append c to curr
            Add an edge from curr to i + 1
            Add an edge from curr to t
        else
            Append i to curr
```

Example of converting IR region to a CFG

```
1:  SEARCH'(arr, size, n)
2:      i = −1
3:      out = −1
4:      branch (i ≥ size) 11
5:      i = i + 1
6:      v = arr[i]
7:      branch (v ≠ n) 10
8:      out = i
9:      goto 11
10:     goto 4
11:     return out
```

$$\{2, 4, 5, 8, 10, 11\}$$



Reaching Definitions

- Def = Write to a variable in an IR instruction
  - An IR instruction typically has a single def, but there may be exceptions, e.g., a procedure call that updates multiple global variables
- Use = Read of a variable in an IR instruction
  - It is common for an IR instruction to have more than one use
- A definition **d** reaches program point **u** if there is a control-flow path from **d** to **u** that **does not** contain an intervening definition of the same variable as **d**
  - Implies that there may be some program execution in which the value of d may reach u; this is not a requirement for all program executions
  - Definition applies to any program point u, but we will be especially interested in the case when u corresponds to a use of the variable written by d

Applications of Reaching Definitions

- Improve the precision of dead code elimination
  - Only mark statements based on reaching definitions (rather than all definitions in simple algorithm)
- Constant Propagation
  - For a use of variable v in statement n, n: x = ... v ...
  - if the definitions of v that reach n are all of the form
  - d: v = c [c a constant]
    then replace the use of v in n with c
- Many others, as we'll see later in the course . . .

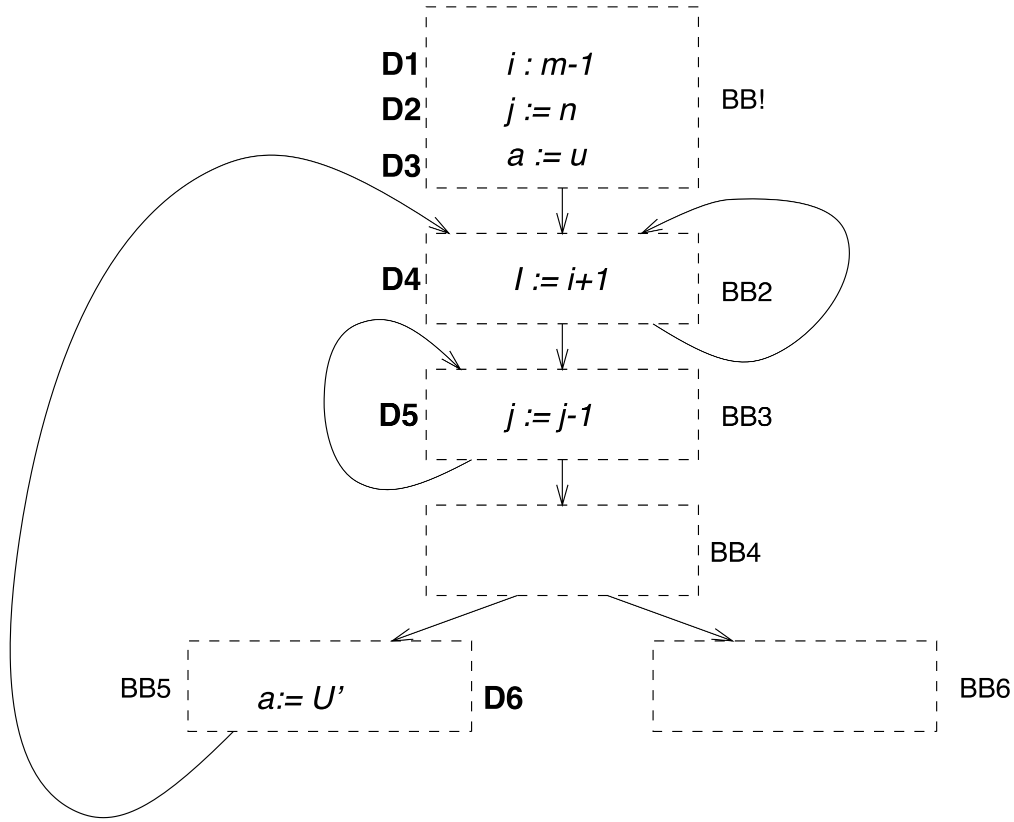
Formalizing a Solution to the Reaching Definitions Problem

- Given a statement/instruction S, define
  - Local sets that can be extracted from S
    - GEN[S] = set of definitions in S ("generated" by S)
    - KILL[S] = set of definitions that may be overwritten by S (e.g., all definitions in program that write to S's lval, whether or not they reach S)
  - Global sets to be computed using CFG
    - IN[S] = set of definitions that reach the entry point of S
    - OUT[S] = set of definitions in S as well as definitions from IN[S] that go beyond S (are not "killed" by S)
- Data flow equations (invariants) for these sets

$$\text{OUT[S]} = \text{GEN[S]} \cup (\text{IN[S]} - \text{KILL[S]})$$

$$\text{IN[S]} = \cup_{p \in predecessors} \text{OUT[p]}$$

Example

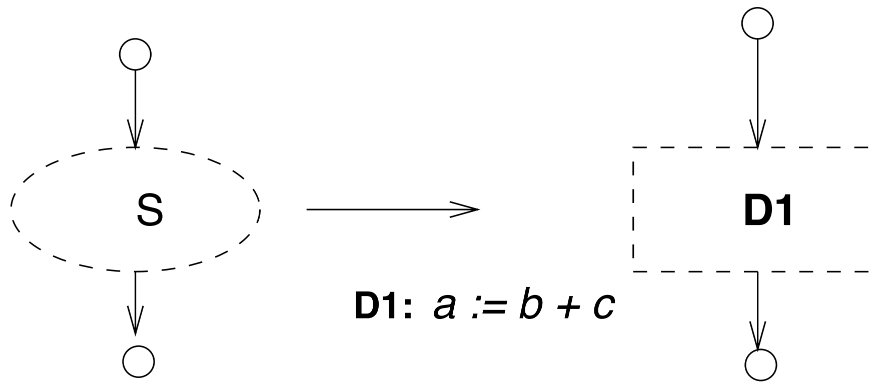


- **D1** reaches **D4** but *not* beyond; why?
  Think of the "kill" sets of **D4**
- **D4** reaches itself due to cyclic dependences in the control-flow
- **D1** reaches **D6** and so on

Basics

- *gen* A set of definitions that the reach the end of statement `S` whether they reach its beginning or not
- *kill* A set of definitions that *never* reach the end of `S` even if they reach the beginning
- *in* A set of definitions that reach the entry to statement `S` in the obvious way
- *out* A set of definitions that go past a statement `S` which include those that reach it and are not killed, and those in gen

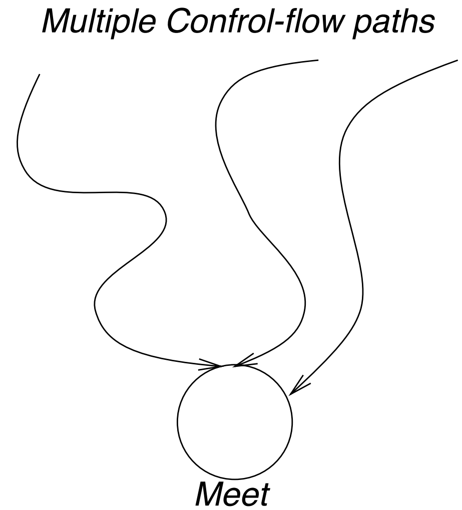
Single Statements



- $gen[S] = \{\mathbf{D1}\}$
- $kill[S] = \{D_a - \{\mathbf{D1}\}\}$ where D_a is the set of *all* definitions of a in the program
- $OUT[S] = GEN[S] \cup (IN[S] - KILL[S])$

Iterative Approaches

Staying with reaching definitions



- The definitions reaching the "join" node are the *union* of all those reaching each of the (three) predecessors (in this example)
- $in[join] = \cup_{p \in predecessors}\, out[p]$

Today’s in-class Worksheet

- Worksheets can be solved collaboratively
  - All other course work must be done individually or in project groups (see syllabus for details)
- Each student should turn in their own solution, based on collaborative discussions
- Worksheets will not be graded or returned, but solutions will be provided
- Worksheets will contribute to class participation grade
- Worksheets will inform teaching staff of concepts that need to be reviewed/reinforced in future lectures