Control Flow Graphs

- **Control Flow Graph (CFG)** = graph representation of computation and control flow in the program
  - framework to statically analyze program control-flow

- In a CFG:
  - Nodes are basic blocks; they represent computation
  - Edges characterize control flow between basic blocks

- Can build the CFG representation either from the high IR or from the low IR

**Build CFG from High IR**

```c
while (c) {
  x = y + 1;
  y = 2 * z;
  if (d) x = y + z;
  z = 1;
}
```

**Build CFG from Low IR**

```assembly
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y + z;
label L3
z = 1;
jump L1
label L2
z = x;
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y + z;
label L3
z = 1;
jump L1
label L2
z = x;
```

**Program Points**

- **Two program points** for each instruction:
  - There is a program point before each instruction
  - There is a program point after each instruction

  Point before
  \[ x = y + 1 \]

  Point after

- In a basic block:
  - Program point after an instruction = program point before the successor instruction

**Using CFGs**

- **Next:** use CFG representation to statically extract information about the program
  - Reason at compile-time
  - About the run-time values of variables and expressions in all program executions

- Extracted information example: live variables

- **Idea:**
  - Define **program points** in the CFG
  - Reason statically about how the information flows between these program points
Program Points: Example

- Multiple successor blocks means that point at the end of a block has multiple successor program points
- Depending on the execution, control flows from a program point to one of its successors
- Also multiple predecessors
- How does information propagate between program points?

Flow of Extracted Information

- Question 1: how does information flow between the program points before and after an instruction?
- Question 2: how does information flow between successor and predecessor basic blocks?
- ... in other words:
  - Q1: what is the effect of instructions?
  - Q2: what is the effect of control flow?

Using CFGs

- To extract information: reason about how it propagates between program points
- Rest of this lecture: how to use CFGs to compute information at each program point for:
  - Live variable analysis, which computes live variables are live at each program point
  - Copy propagation analysis, which computes the variable copies available at each program point

Live Variable Analysis

- Computes live variables at each program point
  - i.e. variables holding values which may be used later (in some execution of the program)
- For an instruction $I$, consider:
  - $\text{in}[I] = \text{live variables at program point before } I$
  - $\text{out}[I] = \text{live variables at program point after } I$
- For a basic block $B$, consider:
  - $\text{in}[B] = \text{live variables at beginning of } B$
  - $\text{out}[B] = \text{live variables at end of } B$
- If $I = \text{first instruction in } B$, then $\text{in}[B] = \text{in}[I]$
- If $I' = \text{last instruction in } B$, then $\text{out}[B] = \text{out}[I']$

How to Compute Liveness?

- Answer question 1: for each instruction $I$, what is the relation between $\text{in}[I]$ and $\text{out}[I]$?
- Answer question 2: for each basic block $B$ with successor blocks $B_1, \ldots, B_n$, what is the relation between $\text{out}[B]$ and $\text{in}[B_1], \ldots, \text{in}[B_n]$?

Part 1: Analyze Instructions

- Question: what is the relation between sets of live variables before and after an instruction?
- Examples:
  - $\text{in}[I] = \{y,z\}$  $\text{in}[I] = \{y,z,t\}$  $\text{in}[I] = \{x,t\}$
  - $x = y+z$  $x = y+z$  $x = x+1$
  - $\text{out}[I] = \{z\}$  $\text{out}[I] = \{x,t\}$  $\text{out}[I] = \{x,t\}$
- ... is there a general rule?
Analyze Instructions

- **Yes:** knowing variables live after I, can compute variables live before I:
  - All variables live after I are also live before I, unless I defines (writes) them
  - All variables that I uses (reads) are also live before instruction I
- **Mathematically:**
  \[ \text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \]

  where:
  - \( \text{def}[I] \): variables defined (written) by instruction I
  - \( \text{use}[I] \): variables used (read) by instruction I

Computing Use/Def

- **Compute** \( \text{use}[I] \) and \( \text{def}[I] \) for each instruction I:
  - if \( I \) is \( x = y \ OR z \) : \( \text{use}[I] = \{y, z\} \) \( \text{def}[I] = \{x\} \)
  - if \( I \) is \( x = OP y \) : \( \text{use}[I] = \{y\} \) \( \text{def}[I] = \{x\} \)
  - if \( I \) is \( x = \text{addr} y \) : \( \text{use}[I] = \{} \) \( \text{def}[I] = \{x\} \)
  - if \( I \) is \( \text{if} (x) \) : \( \text{use}[I] = \{x\} \) \( \text{def}[I] = \{} \)
  - if \( I \) is \( \text{return} x \) : \( \text{use}[I] = \{x\} \) \( \text{def}[I] = \{} \)
  - if \( I \) is \( x = f(y_1, \ldots, y_n) \) : \( \text{use}[I] = \{y_1, \ldots, y_n\} \) \( \text{def}[I] = \{x\} \)

  (For now, ignore load and store instructions)

Example

- **Example:** block B with three instructions I1, I2, I3:
  - \( \text{Live}_1 = \text{in}[B] = \text{in}[I_1] \)
  - \( \text{Live}_2 = \text{out}[I_1] = \text{in}[I_2] \)
  - \( \text{Live}_3 = \text{out}[I_2] = \text{in}[I_3] \)
  - \( \text{Live}_4 = \text{out}[I_3] = \text{out}[B] \)

  - Relation between Live sets:
    - \( \text{Live}_1 = (\text{Live}_2 - \{x\}) \cup \{y\} \)
    - \( \text{Live}_2 = (\text{Live}_3 - \{y\}) \cup \{z\} \)
    - \( \text{Live}_3 = (\text{Live}_4 - \{} \cup \{d\} \)

Backward Flow

- **Relation:**
  \[ \text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \]

- **The information flows backward!**
- **Instructions:** can compute \( \text{in}[I] \) if we know \( \text{out}[I] \)
- **Basic blocks:** information about live variables flows from \( \text{out}[B] \) to \( \text{in}[B] \)

Part 2: Analyze Control Flow

- **Question:** for each basic block B with successor blocks \( B_1, \ldots, B_n \), what is the relation between \( \text{out}[B] \) and \( \text{in}[B_i], \ldots, \text{in}[B_n] \)?

Examples:

- When a variable is live in one successor block, it is also live in the original block.

  \[
  \begin{align*}
  \text{in}[B] & = \{x, y\} \\
  \text{out}[B] & = \{x, y, z\} \\
  \text{in}[B_1] & = \{x\} \\
  \text{in}[B_2] & = \{y\} \\
  \text{in}[B_3] & = \{z\}
  \end{align*}
  \]

- What is the general rule?

   \[ \text{out}[B] = \bigcup_{B' \in \text{succ}(B)} \text{in}[B'] \]

   Again, information flows backward: from successors B’ of B to basic block B

Analyze Control Flow
**Constraint System**

- **Put parts together:** start with CFG and derive a system of constraints between live variable sets:
  
  \[
  \begin{align*}
  \text{in}[I] &= (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] & \text{for each instruction } I \\
  \text{out}[B] &= \bigcup_{B' \in \text{succ}(B')} \text{in}[B'] & \text{for each basic block } B
  \end{align*}
  \]

- **Solve constraints:**
  - Start with empty sets of live variables
  - Iteratively apply constraints
  - Stop when we reach a fixed point

**Constraint Solving Algorithm**

For all instructions \( \text{in}[I] = \text{out}[I] = \emptyset \)

Repeat

  For each instruction \( I \)
  
  \( \text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \)

  For each basic block \( B \)
  
  \( \text{out}[B] = \bigcup_{B' \in \text{succ}(B')} \text{in}[B'] \)

Until no change in live sets

**Example**

\[
\begin{align*}
\text{def} &= \{\}, \ \text{use} = \{c\} \\
\text{def} &= \{x\}, \ \text{use} = \{y\} \\
\text{def} &= \{y\}, \ \text{use} = \{z\} \\
\text{def} &= \{\}, \ \text{use} = \{d\} \\
\text{def} &= \{x\}, \ \text{use} = \{y,z\} \\
\text{def} &= \{\}, \ \text{use} = \{x\} \\
\text{def} &= \{z\}, \ \text{use} = \{x\}
\end{align*}
\]

**Copy Propagation**

- **Goal:** determine copies available at each program point
- **Information:** set of copies \(<x=y>\) at each point

  - For each instruction \( I \):
    - \( \text{in}[I] \) = copies available at program point before \( I \)
    - \( \text{out}[I] \) = copies available at program point after \( I \)

  - For each basic block \( B \):
    - \( \text{in}[B] \) = copies available at beginning of \( B \)
    - \( \text{out}[B] \) = copies available at end of \( B \)

  - If \( I \) = first instruction in \( B \), then \( \text{in}[B] = \text{in}[I] \)
  - If \( I' \) = last instruction in \( B \), then \( \text{out}[B] = \text{out}[I'] \)

**Analyze Instructions**

- Knowing \( \text{in}[I] \), can compute \( \text{out}[I] \):
  - Remove from \( \text{in}[I] \) all copies \(<uv>\) if variable \( u \) or \( v \) is written by \( I \)
  - Keep all other copies from \( \text{in}[I] \)
  - If \( I \) is of the form \( x=y \), add it to \( \text{out}[I] \)

  - Mathematically:
    \[
    \text{out}[I] = (\text{in}[I] - \text{kill}[I]) \cup \text{gen}[I]
    \]

    where:
    - \( \text{kill}[I] \) = copies “killed” by instruction \( I \)
    - \( \text{gen}[I] \) = copies “generated” by instruction \( I \)
Computing Kill/Gen

- Compute kill[I] and gen[I] for each instruction I:
  - if I is \( x = y \op z \): \( \text{gen}[I] = \{ \} \), \( \text{kill}[I] = \{ u=v | u \text{ or } v \text{ is } x \} \)
  - if I is \( x = \op y \): \( \text{gen}[I] = \{ \} \), \( \text{kill}[I] = \{ u=v | u \text{ or } v \text{ is } x \} \)
  - if I is \( x = y \): \( \text{gen}[I] = \{ x=y \} \), \( \text{kill}[I] = \{ u=v | u \text{ or } v \text{ is } x \} \)
  - if I is \( x = \text{addr } y \): \( \text{gen}[I] = \{ \} \), \( \text{kill}[I] = \{ u=v | u \text{ or } v \text{ is } x \} \)
  - if I is \( \text{if (x)} \): \( \text{gen}[I] = \{ \} \), \( \text{kill}[I] = \{ \} \)
  - if I is \( \text{return } x \): \( \text{gen}[I] = \{ \} \), \( \text{kill}[I] = \{ \} \)
  - if I is \( x = f(y_1, \ldots, y_n) \): \( \text{gen}[I] = \{ \} \), \( \text{kill}[I] = \{ u=v | u \text{ or } v \text{ is } x \} \)

(again, ignore load and store instructions)

Forward Flow

- Relation: \( \text{out}[I] = (\text{in}[I] - \text{kill}[I]) \cup \text{gen}[I] \)
- The information flows forward!
- Instructions: can compute \( \text{out}[I] \) if we know \( \text{in}[I] \)
- Basic blocks: information about available copies flows from \( \text{in}[B] \) to \( \text{out}[B] \)

Analyze Control Flow

- Rule: A copy is available at end of block \( B \) if it is live at the beginning of all predecessor blocks
- Characterizes all possible program executions
- Mathematically: \( \text{in}[B] = \bigcap_{B' \in \text{prec}(B)} \text{out}[B'] \)

- Information flows forward: from predecessors \( B' \) of \( B \) to basic block \( B \)

Example

- What are the available copies at the end of the program?
  - \( x=y \)?
  - \( z=t \)?
  - \( x=z \)?

Constraint System

- Build constraints: start with CFG and derive a system of constraints between sets of available copies:
  \[
  \text{out}[I] = (\text{in}[I] - \text{kill}[I]) \cup \text{gen}[I]
  \]
  \[
  \text{in}[B] = \bigcap_{B' \in \text{prec}(B)} \text{out}[B']
  \]
- Solve constraints:
  - Start with empty sets of available copies
  - Iteratively apply constraints
  - Stop when we reach a fixed point

Summary

- Extracting information about live variables and available copies is similar
  - Define the required information
  - Define information before/after instructions
  - Define information at entry/exit of blocks
  - Build constraints for instructions/control flow
  - Solve constraints to get needed information
- ... is there a general framework?
  - Yes: dataflow analysis!