Live variables and copy propagation

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

Control Flow Graphs

Build CFG from High IR

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

Build CFG from Low IR

while (c) {
  x = y + 1;
  y = 2 * z;
  if (d) x = y+z;
  z = 1;
  jump L1
}

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;
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

• **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: $x = y + 1$

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

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 which variables are live at each program point
  - Copy propagation analysis, which computes the variable copies available at each program point

Live variables

- A statement is a *definition* of a variable \( v \) if it may write to \( v \).
- A statement is a *use of* variable \( v \) if it may read from \( v \).
- A variable \( v \) is live at a point \( p \) in a CFG if
  - there is a path from \( p \) to a use of \( v \), and
  - that path does not contain a definition of \( v \)

Computing Use/Def

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

(For now, ignore load and store instructions)

Part 1: Analyze Instructions

- Question: what is the relation between sets of reaching definitions before and after an instruction? in\([I]\) out\([I]\)

- Examples:
  - \( x = y+z; \)
  - assume out\([I] = \{ x, t \} \)

- ... is there a general rule?
Live Variable Analysis

• Computes live variables at each program point
  - i.e., variables holding values that may be used later (in some execution of the program)

• For an instruction I, consider:
  - in[I] = live variables at program point before I
  - out[I] = live variables at program point after I

• For a basic block B, consider:
  - in[B] = live variables at beginning of B
  - out[B] = live variables at end of B

• If I = first instruction in B, then in[B] = in[I]
• If I’ = last instruction in B, then out[B] = out[I’]

How to Compute Liveness?

• Answer question 1: for each instruction I, what is the relation between in[I] and out[I]?
  \[ \text{in}[I] \quad \text{out}[I] \]

• Answer question 2: for each basic block B with successor blocks B₁, …, Bₙ, what is the relation between out[B] and \( \text{in}[Bₐ], \ldots, \text{in}[Bₙ] \)?

Part 1: Analyze Instructions

• Question: what is the relation between sets of live variables before and after an instruction?

• Examples:
  - \( x = y + z; \) \quad in[I] = \{y, z\} \quad out[I] = \{z\}
  - \( x = x + 1; \) \quad in[I] = \{x, t\} \quad out[I] = \{x, t\}

• Yes: knowing variables live after I, can compute variables live before I:
  \[ \text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \]

where:
  - \( \text{def}[I] \) = variables defined (written) by instruction I
  - \( \text{use}[I] \) = variables used (read) by instruction I
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:
  - \( \text{Live}_1 = (\text{Live}_2 - \{x\}) \cup \{y\} \)
  - \( \text{Live}_2 = (\text{Live}_3 - \{y\}) \cup \{z\} \)
  - \( \text{Live}_3 = (\text{Live}_4 - \{} \cup \{d\} \)

\[ y = 2z \]

\[ \text{if (d) \text{Live}_4} \]

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 \( \text{in}[B_1], \ldots, \text{in}[B_n] \)?

- Examples:

  \[ \{x,y\} \quad \{y\} \quad \{z\} \]

  \[ B_1 \quad B_2 \quad B_3 \]

- What is the general rule?

Analyze Control Flow

- Rule: A variable is live at end of block \( B \) if it is live at the beginning of one (or more) successor blocks

- Characterizes all possible program executions

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

- Again, information flows backward: from successors \( B' \) of \( B \) to basic block \( B \)
Constraint System

- **Put parts together:** start with CFG and derive a system of constraints between live variable sets:

  \[ \text{in}[I] = \{ \text{out}[I] - \text{def}[I] \} \cup \text{use}[I] \quad \text{for each instruction } I \]

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

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

### Live Variables

- **L10 = {}**
- **L3 = \{x\} U (L10 - \{z\})**
- **L9 = L2 U L3 U \{c\}**
- **L8 = L9 - \{z\}**
- **L7 = L9 - \{z\}**
- **L6 = \{y,z\} U (L8 - \{x\})**
- **L5 = L6 U L7 U \{d\}**
- **L4 = \{z\} U (L5 - \{y\})**
- **L2 = \{y\} U (L4 - \{x\})**
- **L1 = L2 U L3 U \{c\}**

### Examples

**Example:**

- `def = {}, use = (c)`
- `def = (x), use = (y)`
- `def = (y), use = (z)`
- `def = (x), use = {d}`
- `def = (x), use = \{y,z\}`
- `def = (z), use = {}`
Example

def = (), use = (c)

def = (x), use = {y}
def = (y), use = {z}
def = (), use = (d)
def = (x), use = {y, z}
def = (z), use = ()
def = (z), use = {x}

25

Example

x = y + 1
y = 2 * z
if (d)

z = x
z = 1

26

Example

def = (), use = (c)

def = (x), use = {y}
def = (y), use = {z}
def = (), use = (d)
def = (x), use = {y, z}
def = (z), use = {}
def = (z), use = {x}

27

Example

x = y + 1
y = 2 * z
if (d)

z = x
z = 1

28
General questions

• Do systems of equations of this sort always have solutions?
• If so, do they have unique solutions?
• If there are multiple solutions, which one is the “right” one?
• How do we solve such systems of equations in general?
• If we use the iterative method, does it always terminate and if so, does it always produce a unique answer?

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 \(\text{out}[B] = \text{out}[I']\)
Same Methodology

1. **Express flow of information** (i.e., available copies):
   - For points before and after each instruction \( \text{in}[I], \text{out}[I] \)
   - For points at exit and entry of basic blocks \( \text{in}[B], \text{out}[B] \)

2. **Build constraint system** using the relations between available copies

3. **Solve constraints** to determine available copies at each point in the program

---

Analyze Instructions

• Knowing \( \text{in}[I] \), can compute \( \text{out}[I] \):
  - Remove from \( \text{in}[I] \) all copies \(<u=v>\) 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 \( \text{kill}[I] \) and \( \text{gen}[I] \) for each instruction \( I \):
  
  - if \( I \) is \( x = y \ OP \ z \) : \( \text{gen}[I] = {} \) \( \text{kill}[I] = \{ u=v \mid u \text{ or } v \text{ is } x \} \)
  - if \( I \) is \( x = OP \ y \) : \( \text{gen}[I] = {} \) \( \text{kill}[I] = \{ u=v \mid u \text{ or } v \text{ is } x \} \)
  - if \( I \) is \( x = y \) : \( \text{gen}[I] = \{ x=y \} \) \( \text{kill}[I] = \{ u=v \mid u \text{ or } v \text{ is } x \} \)
  - if \( I \) is \( x = \text{addr } y \) : \( \text{gen}[I] = {} \) \( \text{kill}[I] = \{ u=v \mid u \text{ or } v \text{ is } x \} \)
  - if \( I \) is \( \text{if } (d) \) : \( \text{gen}[I] = {} \) \( \text{kill}[I] = {} \)
  - if \( I \) is \( \text{return } x \) : \( \text{gen}[I] = {} \) \( \text{kill}[I] = {} \)
  - if \( I \) is \( x = f(y_1, ..., y_n) \) : \( \text{gen}[I] = {} \) \( \text{kill}[I] = \{ u=v \mid u \text{ or } v \text{ is } x \} \)

  (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 beginning of block B if it is available at the end of all predecessor blocks.
- **Characterizes all possible program executions**
- Mathematically:
  \[ \text{in}[B] = \bigcap_{B' \in \text{pred}(B)} \text{out}[B'] \]
- Information flows forward: from predecessors \( B' \) of B to basic block B.

### Constraint System

- **Build constraints:** start with CFG and derive a system of constraints between sets of available copies:
  \[
  \begin{align*}
  \text{out}[I] &= (\text{in}[I] - \text{kill}[I]) \cup \text{gen}[I] \\
  \text{in}[B] &= \bigcap_{B' \in \text{pred}(B)} \text{out}[B']
  \end{align*}
  \]
  for each instruction I
  for each basic block B
- **Solve constraints:**
  - Start with empty set of available copies at start and universal set of available copies everywhere else
  - Iteratively apply constraints
  - Stop when we reach a fixed point

### Example

- What are the available copies at the end of the program?

```plaintext
x = y
z = t
if (c)
  x = z
  y = 2*z
  if (d)
    x = y
    z = t
    t = 1
u = z + 1
z = t
```

### Example

- What are the available copies at the end of the program?

```plaintext
x = y
z = t
if (c)
  x = z
  y = 2*z
  if (d)
    x = y
    z = t
    t = 1
u = z + 1
z = t
```
Iteration 1

- What are the available copies at the end of the program?

\[
x = y \\
z = t
\]

if (c)

\[
x = z \\
y = 2z
\]

if (d)

\[
t = 1 \\
u = z + 1 \\
z = t
\]

\[
x = z? \\
z = t?
\]

Fixed Point Reached!

- What are the available copies at the end of the program?

\[
x = y NO \\
z = t YES \\
x = z NO
\]

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!