CS42/413

Introduction to Compilers
Radu Rugina

Lecture 24: Liveness and Copy Propagation
31 Mar 04

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

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

Build CFG from Low IR

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


• 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 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:
  - \( \text{in}[I] = \) live variables at program point before I.
  - \( \text{out}[I] = \) live variables at program point after I.
- For a basic block B, consider:
  - \( \text{in}[B] = \) live variables at beginning of B.
  - \( \text{out}[B] = \) live variables at end of B.
- If \( I \) is first instruction in B, then \( \text{in}[B] = \text{in}[I] \).
- If \( I \) is 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 \( \text{out}[I] \)?
- \( \text{in}[I] \) \( \text{out}[I] \)

- Answer question 2: for each basic block B with successor blocks \( B_1, ..., B_n \), what is the relation between \( \text{out}[B] \) and \( \text{in}[B_1], ..., \text{in}[B_n] \)?

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

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

  (For now, ignore load and store instructions)

---

### Example

- Example: block B with three instructions I1, I2, I3:
  \[ \begin{align*}
  \text{Live1} &= \text{in}[B] = \text{in}[I1] \\
  \text{Live2} &= \text{out}[I1] = \text{in}[I2] \\
  \text{Live3} &= \text{out}[I2] = \text{in}[I3] \\
  \text{Live4} &= \text{out}[I3] = \text{out}[B]
  \end{align*} \]

- Relation between Live sets:
  \[ \begin{align*}
  \text{Live1} &= (\text{Live2}(\cdot x)) \cup \{ y \} \\
  \text{Live2} &= (\text{Live3}(\cdot y)) \cup \{ z \} \\
  \text{Live3} &= (\text{Live4}(\cdot y)) \cup \{ d \}
  \end{align*} \]

---

### Backward Flow

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

- **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 \( B_1, ..., B_m \), what is the relation between \( \text{out}[B] \) and \( \text{in}[B_1], ..., \text{in}[B_m] \)?

- **Examples:**

- **What is the general rule?**

---

### Analyze Control Flow

- **Rule:** A variables is live at end of block B if it is live at the beginning of one successor block

- **Characterizes all possible program executions**

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

- **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:
  \[
  \begin{align*}
  \text{in}[I] &= (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \\
  \text{out}[B] &= \bigcup_{B' \subseteq \text{succ}(B)} \text{in}[B']
  \end{align*}
  \]
  for each instruction I
  for each basic block B

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

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 is first instruction in B, then \(\text{in}[B] = \text{in}[I]\)
- If I is 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 available copies
3. Solve constraints to determine available copies at each point in the program

Analyze Instructions

- Knowing \(\text{in}[I]\), can compute \(\text{out}[I]\):
  \[
  \begin{align*}
  \text{in}[I] &\quad \text{out}[I] \\
  \text{I} &\quad \text{I}
  \end{align*}
  \]
  - 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 kill[I] and gen[I] for each instruction I:
  if I is x = y OP z : gen[I] = {} kill[I] = {u} if u = v or u or v is x
  if I is x = y OP z : gen[I] = {y} kill[I] = {u} if u = v or u or v is x
  if I is x = y : gen[I] = {x,y} kill[I] = {u} if u = v or u or v is x
  if I is x = addr y : gen[I] = {} kill[I] = {u} if u = v or u or v is x
  if I is if (x) : gen[I] = {} kill[I] = {}
  if I is return x : gen[I] = {} kill[I] = {}
  if I is x = f(y1,..,yn) : gen[I] = {} kill[I] = {u} if u = v or u or v is x

(again, ignore load and store instructions)

Forward Flow

- Relation:
  out[I] = (in[I] - kill[I]) \cup gen[I]
  out[I] = in[I]

- The information flows forward!

- Instructions: can compute out[I] if we know in[I]

- Basic blocks: information about available copies flows from in[B] to 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:
  in[B] = \bigcap_{B' \in \text{pred}(B)} 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?

- 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

Constraint System

- Build constraints: start with CFG and derive a system of constraints between sets of available copies:

  - for each instruction I:
    \[ \text{out}[I] = (\text{in}[I] - \text{kill}[I]) \cup \text{gen}[I] \]
  - for each basic block B:
    \[ \text{in}[B] = \bigcap_{B' \in \text{pred}(B)} \text{out}[B'] \]

- Solve constraints:
  - Start with empty sets of available copies
  - Iteratively apply constraints
  - Stop when we reach a fixed point

Summary