CS42/413

Introduction to Compilers
Radu Rugina

Lecture 21: Liveness and Copy Propagation
12 Mar 03

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;
}
```

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

```
z = x;
```

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:
  - \( \text{in}[I] = \text{live variables at program point before I} \)
  - \( \text{out}[I] = \text{live variables at program point after I} \)
- 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 \( I \) = first instruction in \( B \), then \( \text{in}[B] = \text{in}[I] \)
- If \( I' \) = 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?
- \( \text{in}[I] = (y,z) \)  \( \text{in}[I] = (y,z,t) \)  \( \text{in}[I] = (x,t) \)
  - \( \text{out}[I] = (z) \)  \( \text{out}[I] = (x,t) \)  \( \text{out}[I] = (x,t) \)

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

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

(For now, ignore load and store instructions)

Example

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

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

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

- Examples:
  \[
  \begin{align*}
  B_1 &\to B_2 \quad (x, y) \\
  B_3 &\to B_4 \quad (x, y, z)
  \end{align*}
  \]

- 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:
  \[
  \text{out}[B] = \bigcup_{B' \text{ is succ of } B} \text{in}[B']
  \]

- 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:

\[
\begin{align*}
in[I] &= (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \\
\text{out}[B] &= \bigcup_{B' = \text{ succ}(B)} \text{in}[B']
\end{align*}
\]

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

Constraint Solving Algorithm

For all instructions in[I] = out[I] = ∅

Repeat

For each instruction I

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

For each basic block B

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

Until no change in live sets

Example

\[
\begin{align*}
def &= \{\}, \text{ use } = \{c\} & \text{ if (c)} \\
def &= \{x\}, \text{ use } = \{y\} & x = y + 1 \\
def &= \{y\}, \text{ use } = \{z\} & y = 2z \\
def &= \{\}, \text{ use } = \{d\} & \text{ if (d)} \\
def &= \{x\}, \text{ use } = \{y,z\} & x = y + z \\
def &= \{x\}, \text{ use } = \{\} & z = 1 \\
def &= \{z\}, \text{ use } = \{x\} & z = x
\end{align*}
\]

Copy Propagation

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

For each instruction I:

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

For each basic block B:

- \(\text{in}[B] = \text{copies available at beginning of B}\)
- \(\text{out}[B] = \text{copies available at end of B}\)

If I = first instruction in B, then in[B] = in[I]

If I' = last instruction in B, then out[B] = out[I']

Same Methodology

1. Express flow of information (i.e. available copies):
   - For points before and after each instruction (in[I], out[I])
   - For points at exit and entry of basic blocks (in[B], 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 in[I], can compute out[I]:
  - Remove from in[I] all copies <u | v> if variable u or v is written by I
  - Keep all other copies from in[I]
  - If I is of the form x = y, add it to 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 | u or v is x }
  - if I is x = y OP y : gen[I] = { } kill[I] = { u = v | u or v is x }
  - if I is x = y : gen[I] = { x = y } kill[I] = { u = v | u or v is x }
  - if I is x = addr y : gen[I] = { } kill[I] = { u = v | 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(y_1, ..., y_n) : gen[I] = { } kill[I] = { u | v } u or v is x

(again, ignore load and store instructions)

Forward Flow

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

  \[ \text{in}[I] \quad \text{out}[I] \]

- The information flows forward!

  \[ \text{in}[B] \quad \text{out}[I] \]

- 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:
  \[ \text{in}[B] = \bigcap_{B' \in \text{pred}(B)} \text{out}[B'] \]

- 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:

  \[ \text{out}[I] = ( \text{in}[I] - \text{kill}[I] ) \cup \text{gen}[I] \quad \text{for each instruction I} \]

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

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

Example

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

  \[ \frac{x = y \quad z = t}{x = ? \quad z = t} \]

  \[ \frac{if (c)}{x = z \quad y = 2z \quad if (d)} \]

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

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!