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
while (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

\[ x = y + 1 \]

- **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?

```
x = y + 1
y = 2 * z
if (d)
  x = y + z
z = 1
```
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 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:
  – $\text{in}[I] =$ 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 $I =$ 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 $\text{out}[I]$?

• **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:**

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

• … is there a general rule?
Analyze Instructions

• **Yes:** knowing variables live after I, can compute variables live before I:
  
  - Each variable live after I is also live before I, unless I defines (writes) it
  
  - Each variable that I uses (reads) is also live before instruction I

• **Mathematically:**

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

where:

- \text{def}[I] = variables defined (written) by instruction I
- \text{use}[I] = variables used (read) by instruction I
Computing Use/Def

• Compute use[I] and def[I] for each instruction I:

  if I is \( x = y \ OP \ z \) : \( \text{use}[I] = \{y, z\} \quad \text{def}[I] = \{x\} \)
  if I is \( x = \ OP \ y \) : \( \text{use}[I] = \{y\} \quad \text{def}[I] = \{x\} \)
  if I is \( x = y \) : \( \text{use}[I] = \{y\} \quad \text{def}[I] = \{x\} \)
  if I is \( x = \text{addr} \ y \) : \( \text{use}[I] = \{\} \quad \text{def}[I] = \{x\} \)
  if I is \( \text{if} \ (x) \) : \( \text{use}[I] = \{x\} \quad \text{def}[I] = \{\} \)
  if I is \( \text{return} \ x \) : \( \text{use}[I] = \{x\} \quad \text{def}[I] = \{\} \)
  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:

- \( \text{Live}_1 = \text{in}[B] = \text{in}[I1] \)
- \( \text{Live}_2 = \text{out}[I1] = \text{in}[I2] \)
- \( \text{Live}_3 = \text{out}[I2] = \text{in}[I3] \)
- \( \text{Live}_4 = \text{out}[I3] = \text{out}[B] \)

• 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 in[I] if we know out[I]

• Basic blocks: information about live variables flows from out[B] to 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:**

- **What is the general rule?**
Analyze Control Flow

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

• **Characterizes all possible program executions**

• **Mathematically:**

\[
\text{out}[B] = \bigcup \text{in}[B']
\]

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

• 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:

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

• **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 I do in[I] = out[I] = ∅;
repeat
    select an instruction I (or a basic block B) such that
    in[I] ≠ ( out[I] – def[I] ) ∪ use[I]
    or (respectively)
    out[B] ≠ ∪
    B’ ∈ succ(B)
    and update in[I] (or out[B]) accordingly
until no such change is possible
Example

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

```
if (c)
```

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

```
x = y+z
```

```
z = 1
```

```
z = x
```

**strategy:** pick program points in postorder
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 }  

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

{ }_1  
{ }_2  
{ }_3  
{ }_4  
{ }_5  
{ }_6  
{ }_7  
{ }_8  
{ }_9  
{ }_10
Example

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

```
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}
def = {x}, use = {y}
```
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}

if (c)

x = y+1

y = 2*z

if (d)

z = x

x = y+z

z = 1

z = x
Example

def = {}, use = {c} -----> if (c)

def = {x}, use = {y} -----> x = y + 1

def = {y}, use = {z} -----> y = 2 * z

def = {}, use = {d} -----> if (d)

def = {x}, use = {y,z} -----> z = x

def = {z}, use = {} -----> z = 1

def = {z}, use = {x} -----> z = x
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}

def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}

def = {y,z}
Example

def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {x}

def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z}
def = {z}, use = {}
Example

def = {}, use = {c}

\[ x = y + 1 \]

\[ y = 2z \]

if (d)\[ z = x \]
if (c)\[ x = y + z \]
\[ z = 1 \]
\[ z = x \]
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}

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

z = 1
z = x
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}

def = {}, use = {c}
def = {x}, use = {y}
def = {y}, use = {z}
def = {}, 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}
```

```
if (c)
```

```
x = y+1
```

```
y = 2*z
```

```
if (d)
```

```
x = y+z
```

```
z = 1
```

```
z = x
```

Example

def = {}, use = {c} ---------------------

def = {x}, use = {y}
def = {y}, use = {z}
def = {}, use = {d}
def = {x}, use = {y,z} {y,z}_4

def = {z}, use = {}
def = {z}, use = {x}

{x,y,z,d,c}_{10} 
{x,y,z,d}_{9} 
{y,z,d}_6 
{y,z}_5 
{ }_2 
{x,y,z,d,c}_{1} 
{x}_8 
{ }_7

{x,y,z,d,c}_{10} 
{x,y,z,d}_9 
{y,z,d}_6 
{y,z}_5 
{ }_2 
{x,y,z,d,c}_{1} 
{x}_8 
{ }_7
Example

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

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

\[
\begin{align*}
\{x, y, z, d, c\}_{10} & \\
\{x, y, z, d\}_9 & \\
\{y, z, d\}_6 & \\
\{y, z\}_5 & \\
\{x, y, d, c\}_2 & \\
\{x, y, z, d, c\}_1 & \\
\{x\}_8 & \\
\{\}_7
\end{align*}
\]
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}
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}

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}

def = {x,y,z,d,c}_{10}
def = {x,y,z,d}_{9}
def = {y,z,d}_{6}
def = {y,z} _{5}
def = {x,y,d,c}_{4}
def = {x,y,d,c}_{3}
def = {x,y,d,c}_{2}
def = {x,y,z,d,c}_{1}
def = {x} _{8}
def = { } _{7}
Example

\[
x = y + 1 \\
y = 2 \times z \\
\text{if (d)} \\
z = x \\
\text{if (c)} \\
x = y + z \\
z = 1 \\
z = x
\]
Example

def = {}, use = {c} → if (c) → {x,y,z,d,c}_{10}
def = {x}, use = {y} → x = y+1 → {x,y,z,d,c}_{9}
def = {y}, use = {z} → y = 2*z → {y,z,d,c}_{6}
def = {}, use = {d} → if (d) → {x,y,z,d,c}_{5}
def = {x}, use = {y,z} → x = y+z → {x,y,z,d,c}_{4}
def = {z}, use = {} → z = 1 → {x,y,d,c}_{2}
def = {z}, use = {x} → z = x → {x}_{8}
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}

{x,y,z,d,c}₁₀ → if (c)

{x,y,z,d}₉
{y,z,d,c}₆

x = y+1

{y,z,d,c}₄
{y,z,d}₃

y = 2*z

if (d)

{x,y,d,c}₂
{x,y,z,d,c}₁

z = 1

{x,y,z,d,c}₁

{x}₈

z = x

{ }₇
Example

\[ x = y + 1 \]
\[ y = 2 \cdot z \]
\[ \text{if (d)} \]
\[ z = x \]
\[ \text{if (c)} \]
\[ x = y + z \]
\[ z = 1 \]
\[ z = x \]

def = \{\}, use = \{c\}
\[ \{x,y,z,d,c\}_{10} \]

def = \{x\}, use = \{y\}
\[ \{x,y,z,d\}_{9} \]

def = \{y\}, use = \{z\}
\[ \{y,z,d,c\}_{6} \]

def = \{\}, use = \{d\}
\[ \{x,y,z,d,c\}_{5} \]

def = \{x\}, use = \{y,z\}
\[ \{y,z,d,c\}_{4} \]
\[ \{x,y,d,c\}_{3} \]

def = \{z\}, use = \{\}  
\[ \{x,y,d,c\}_{2} \]
\[ \{x,y,z,d,c\}_{1} \]

def = \{z\}, use = \{x\}
\[ \{x\}_{8} \]
\[ \{\}\_{7} \]
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}

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}

if (c)
{x,y,z,d,c}_10
{y,z,d,c}_9
if (d)
{x,y,z,d,c}_5
x = y+1
y = 2*z
z = x
{x,y,z,d,c}_1

{y,z,d,c}_4
{y,z,d,c}_3
{x,y,z,d,c}_2
z = 1
{x,y,d,c}_7
{x}_8
z = x
{ }
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}

{x,y,z,d,c}_{10} → if (c)

{x,y,z,d,c}_{9}
{y,z,d,c}_{6} → y = 2*z → if (d)

{x,y,z,d,c}_{5}

x = y + 1

{x,y,d,c}_{4}

{{x,y,d,c}_{3}} → x = y + z

{y,z,d,c}_{4}

z = 1

{x,y,d,c}_{2}

{x,y,z,d,c}_{1}

{x}_{8}

{ }_{7} → z = x

{ }_{7}
Fixed Point Reached

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}

{x,y,z,d,c}^{10} \rightarrow \ if (c) \rightarrow \ {x,y,z,d,c}^{9} \rightarrow \ x = y + 1 \rightarrow \ {y,z,d,c}^{6} \rightarrow \ y = 2 * z \rightarrow \ {x,y,z,d,c}^{5} \rightarrow \ if (d) \rightarrow \ {x,y,d,c}^{3} \rightarrow \ x = y + z \rightarrow \ {x,y,d,c}^{2} \rightarrow \ z = 1 \rightarrow \ {x,y,d,c}^{1} \rightarrow \ z = x \rightarrow \ \{\}\_7 

\{x,y,z,d,c\}^{10} \rightarrow \ if (c) \rightarrow \ \{x,y,z,d,c\}^{9} \rightarrow \ x = y + 1 \rightarrow \ \{y,z,d,c\}^{6} \rightarrow \ y = 2 * z \rightarrow \ \{x,y,z,d,c\}^{5} \rightarrow \ if (d) \rightarrow \ \{x,y,d,c\}^{3} \rightarrow \ x = y + z \rightarrow \ \{x,y,d,c\}^{2} \rightarrow \ z = 1 \rightarrow \ \{x,y,d,c\}^{1} \rightarrow \ z = x \rightarrow \ \{\}\_7
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 (in[I], out[I])
   - For points at exit and entry of basic blocks (in[B], 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 in[1], can compute out[1]:
  - Remove from in[1] all copies <u=v> if variable u or v is written by I
  - Keep all other copies from in[1]
  - If I is of the form x=y, add it to out[1]

- Mathematically:
  \[ \text{out}[1] = (\text{in}[1] - \text{kill}[1]) \cup \text{gen}[1] \]

where:
- \( \text{kill}[1] \) = copies “killed” by instruction I
- \( \text{gen}[1] \) = copies “generated” by instruction I
Computing Kill/Gen

• Compute $\text{kill}[I]$ and $\text{gen}[I]$ for each instruction $I$:

- if $I$ is $x = y \ \text{OP} \ z$ : $\text{gen}[I] = \{\} \quad \text{kill}[I] = \{u=v | u \text{ or } v \text{ is } x\}$
- if $I$ is $x = \text{OP} \ y$ : $\text{gen}[I] = \{\} \quad \text{kill}[I] = \{u=v | u \text{ or } v \text{ is } x\}$
- if $I$ is $x = y$ : $\text{gen}[I] = \{x=y\} \quad \text{kill}[I] = \{u=v | u \text{ or } v \text{ is } x\}$
- if $I$ is $x = \text{addr} \ y$ : $\text{gen}[I] = \{\} \quad \text{kill}[I] = \{u=v | u \text{ or } v \text{ is } x\}$
- if $I$ is $\text{if} \ (x)$ : $\text{gen}[I] = \{\} \quad \text{kill}[I] = \{\}$
- if $I$ is $\text{return} \ x$ : $\text{gen}[I] = \{\} \quad \text{kill}[I] = \{\}$
- if $I$ is $x = f(y_1, \ldots, y_n)$ : $\text{gen}[I] = \{\} \quad \text{kill}[I] = \{u=v | u \text{ or } v \text{ is } x\}$

(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]

\[
\begin{align*}
\text{in}[B] & \\
  x &= y \\
  y &= 2z \\
  \text{if} \ (d) \\
  \text{out}[B]
\end{align*}
\]
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:

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

- 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:

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

- **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?

  - $x = y$?
  - $z = t$?
  - $x = z$?
Example

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

  - x = y?
  - z = t?
  - x = z?
Iteration 1

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

x = y?

z = t?

x = z?
Iteration 2

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

$\text{x=y?}$

$\text{z=t?}$

$\text{x=z?}$
Fixed Point Reached!

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

  $x = y$? NO
  $z = t$? YES
  $x = z$? NO

```
L1 = {}
L2 = {x=y}
L3 = {x=y, z=t}
L4 = {z=t}
L5 = {z=t}
L6 = {z=t}
L7 = {z=t, x=z}
L8 = {z=t, x=z}
L9 = {z=t, x=z}
L10 = {z=t, x=z}
L11 = {x=z}
L12 = {x=z}
L13 = {x=z}
L14 = {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!