Recap: Control Flow Graphs

- **Control Flow Graph (CFG):** graph representation of computation and control flow in the program
- Framework to statically analyze program control-flow
- Next: use CFG to statically extract information about program
- Reason at compile-time about run-time values of variables in all program executions
- Data-flow analysis: gather information about the possible set of values of variables at various points in a program
Liveness

- Liveness is a data-flow property of variables: “Is the value of this variable needed?”
- Optimization: eliminate assignments to dead variables (i.e. variables that are never used after definitions)
  ```c
  int f(int x, int y) {
      int z = x + y;
      ...
  }
  ```
- Live variable analysis is **undecidable** in general
- We compute a syntactic and conservative approximation of liveness
  - Like many other data-flow analysis techniques
  ```c
  int f(int x, int y) {
      int z = x + y;
      if (tricky-calculation) x = z;
      return x;
  }
  ```
Live Variable Analysis: Backward

- Liveness is naturally computed using **backward** data-flow analysis
- Usage information from future statements must be propagated backward through the program to discover which variables are live

```c
int f(int x, int y) {
    int z = x + y;
    ...
    int t = z - 2;
    println(z);
    if(z != 2) ...;
}
```
Variable liveness flows backward through the program.

Each statement has an effect on liveness information as it flows past.

A statement makes a variable **live** when it reads it.

\[
\begin{align*}
    \text{if } (y > 1) & \text{ println}(y); \quad \text{reads } y \\
    b &= z - x; \quad \text{reads } z, x \\
    a &= x + 1; \quad \text{reads } x \\
\end{align*}
\]
Live Variable Analysis

- Variable liveness flows backward through the program.
- Each statement has an effect on liveness information as it flows past.
- A statement makes a variable **dead** when it defines (assigns to) it.

```
{ }  // defines z
{ x, y, z}

x = 5;  // defines x
{ x}

{ x, y}

y = 10;  // defines y

{ x, y}

z = 1;  // defines z
{ x, y, z}
```
As liveness flows backwards past an statement, we modify liveness information:

- Add any variables which it reads (they become live)
- Remove any variables which it defines (they become dead)

\[
\text{use}(\text{println}(x)) = \{\, x \,\} \\
\text{def}(x = 3) = \{\, x \,\}
\]

Variable \( v \) is live before a statement \( S \) if:

1. There is a statement \( S' \) in CFG that \textit{uses} \( v \)
2. There exists a path from \( S \) to \( S' \) passing through no \textit{def} of \( v \)
Live Variable Analysis

- If a statement both references and defines variables, remove the defined variables before adding the read ones.
- $L_0$ Initial set of live variables

\[
L_0 = L_1 \cup \{x, y\} \\
\text{read}(x,y) \\
L_1 = L_2 \setminus \{x\} \\
\text{write}(x)
\]

\[
L_0 = (L_2 \setminus \{x\}) \cup \{x, y\}
\]

In general:

- $\text{in}(S)$: set of live variables immediately before statement $S$
- $\text{out}(S)$: set of live variables immediately after statement $S$

\[
\text{in}(S) = (\text{out}(S) \setminus \text{def}(S)) \cup \text{use}(S)
\]
Straight-Line Code

- In straight-line code each node has a unique successor
- Variables live at the exit of a node are exactly those variables live at the entry of its successor

```
i_1:

\text{in}(i_1) = \{x, y, z\}
\text{out}(i_1) = \{x, y, z\}

i_2: x = x+y

\text{in}(i_2) = \{x, y, z\}
\text{out}(i_2) = \{x, z\}

i_3: println(x)

\text{in}(i_3) = \{x, z\}
\text{out}(i_3) = \{z\}

i_4:

\text{in}(i_4) = \{z\}
```
Multiple Successors

- In general each node has an arbitrary number of successors
- Variables live at the exit of a node are exactly those variables live at the entry of all its successors

Example:

\[
\begin{align*}
\text{out}(S) &= \{ x, y, z \} \\
in(S_1) &= \{ x, z \} \\
in(S_2) &= \{ x, y \}
\end{align*}
\]

General:

\[
\text{out}(S) = \bigcup_{S_i \in \text{succ}(S)} \text{in}(S_i)
\]
Data-flow Equations

• Start with CFG and derive a system of constraints between live variable sets

\[
in(S') = (out(S') \setminus def(S')) \cup use(S')
\]
\[
out(S') = \bigcup_{S_i \in succ(S)} in(S_i)
\]

Solve constraints:

• Start with empty sets of live variables
• Iteratively apply constraints
• Stop when we reach a fixed point
for all statements $S$ do

\( \text{in}(S) = \text{out}(S) = \emptyset \)

repeat

select a statement $S$ such that

\( \text{in}(S) \neq (\text{out}(S) \setminus \text{def}(S)) \cup \text{use}(S) \)

or (respectively)

\( \text{out}(S) \neq \bigcup_{S_i \in \text{succ}(S)} \text{in}(S_i) \)

update $\text{in}(S)$ (or $\text{out}(S)$) accordingly

until no such change is possible
Exercise

- Compute the set of live variables at each point of the program

```c
x = 5;
y = 10;
z = 0;
while (x > 0) {
    x = x - 1;
    u = y;
    while (u > 0) {
        u = u - 1;
        z = z + 1;
    }
}
```
• Compute the set of live variables at each point of the program

\[
\begin{align*}
x &= 5; \\
y &= 10; \\
z &= 0; \\
\text{while } (x > 0) \{ \\
& \quad x = x - 1; \\
& \quad u = y; \\
& \quad \text{while } (u > 0) \{ \\
& \quad \quad u = u - 1; \\
& \quad \quad z = z + 1; \\
& \quad \} \\
& \} \\
\end{align*}
\]

\[
\begin{align*}
in(S) &= (\text{out}(S) \setminus \text{def}(S)) \cup \text{use}(S) \\
\text{out}(S) &= \bigcup_{S_i \in \text{succ}(S)} \text{in}(S_i)
\end{align*}
\]
• **Intra-procedural** Analysis: analyzing the body of a single procedure
• **Inter-procedural** Analysis: analyzing the whole program with function calls

```c
x = 0;
  // is y live here? (yes iff used in procedure P)
P();
  // is x still equal to 0 here?
  // (yes iff not changed in P)
y = x;
```
A naïve and safe approach to inter-procedural analysis:

- Assume any function will read and write all global variables (worst case scenario)
- Every global variable is live before any function call!

- Leads to over-cautious optimizations
- There are more accurate inter-procedural analyses that consider the call graph of a program
  - (beyond the scope of the course)
• Most languages use variables containing addresses
  • e.g. pointers (C,C++), references (Java), call-by-reference parameters (Pascal, C++, Fortran)
• Pointer aliases: multiple names for the same memory location
  • Dereferencing the aliases returns the same object
• Problem: Don’t know what variables read and written by accesses via pointer aliases
  (e.g. *p=y; x=*p; p->f=y; x=p->f; etc.)
• Need to know accessed variables to compute dataflow information after each instruction
• Worst-case scenarios:

- \( *p = y \) may write any memory location
- \( x = *p \) may read any memory location

• All the variables may be live before \( x = *p \)
• Leads to over-cautious optimizations
• There are more accurate pointer alias analyses
  • (beyond the scope of the course)