More Dataflow Analyses
Steps to building analysis

- Step 1: Choose lattice
- Step 2: Choose direction of dataflow (forward or backward)
- Step 3: Create transfer function
- Step 4: Choose confluence operator (i.e., what to do at merges)
  - Either join or meet in the lattice
- Let’s walk through these steps for a new (old) analysis
Liveness analysis

• Which variables are live at a particular program point?

• Used all over the place in compilers
  • Register allocation
  • Loop optimizations

• We’ve done this for single basic blocks, but what about across basic blocks?
Choose lattice

• What do we want to know?
  • At each program point, want to maintain the set of variables that are live
  • Lattice elements: sets of variables
  • Natural choice for lattice: powerset of variables!
Choose dataflow direction

• A variable is \textit{live} if it is used later in the program without being redefined

• At a given program point, we want to know information about what happens later in the program

• This is information about the future of the program

• This means that liveness is a \textit{backwards} analysis

  • No reason to run the program forward!

  • Recall that we did liveness backwards when we looked at single basic blocks

• Rule of thumb: if symbolic information you are tracking is about what happens in the future, run the analysis backwards
symbolically executing a statement

• What do we do for a statement like:

• $x = y + z$

• If $x$ was live “before” (i.e., live after the statement), it isn’t now (i.e., is not live before the statement)

• If $y$ and $z$ were not live “before,” they are now

• What about:

• $x = x$
symbolically executing a statement

- Let’s generalize
- For any statement $s$, we can look at which live variables are killed, and which new variables are made live (generated)
- Which variables are killed in $s$?
  - The variables that are defined in $s$: $\text{DEF}(s)$
- Which variables are made live in $s$?
  - The variables that are used in $s$: $\text{USE}(s)$
- If the set of variables that are live after $s$ is $X$, what is the set of variables live before $s$?

$$T_s(X) = \text{use}(s) \cup (X - \text{def}(s))$$
Dealing with aliases

• Aliases, as usual, cause problems

• Consider

```c
int x, y, r, s
int *z, *w;
if (...) z = &y else z = &x
if (...) w = &r else w = &s
*z = *w; //which variable is defined? which is used?
```

• What should USE(*z = *w) and DEF(*z = *w) be?
  • Keep in mind: the goal is to get a list of variables that may be live at a program point

• For now, assume there is no aliasing
Dealing with function calls

- Similar problem as aliases:
  ```c
  int foo(int &x, int &y); //pass by reference!
  ```

```c
  void main() {
    int x, y, z;
    z = foo(x, y);
  }
```

- Simple solution: functions can do anything – redefine variables, use variables
  - So DEF(foo()) is { } and USE(foo()) is V

- Real solution: *interprocedural* analysis, which determines what variables are used and defined in foo
What about merges?

• What happens at a merge point?

  • The variables live into a merge point are the variables that are live along *either* branch

  • Confluence operator: Set union (∪) of all live sets of outgoing edges

\[
T_{\text{merge}} = \bigcup_{X \in \text{succ(merge)}} X
\]
How to initialize analysis?

• At the end of the program, we know no variables are live $\rightarrow$ value at exit point is {}.

• What about if we’re analyzing a single function?

• Need to make conservative assumption about what may be live.

• What about elsewhere in the program?

• We should initialize other sets to {}.
liveness example