More Dataflow Analysis
Recall steps to building analysis

- Step 1: Choose lattice
- Step 2: Choose direction of dataflow (forward or backward)
- Step 3: Create monotonic 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 analysis
Liveness analysis

• Which variables are live at a particular program point?
• Used all over the place in compilers
  • Register allocation
  • Loop optimizations
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 *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 means that liveness is a *backwards* analysis

• Recall that we did liveness backwards when we looked at single basic blocks
Create x-fer functions

- 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:
  \[ x = x \]
Create x-fer functions

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

• Aliases, as usual, cause problems

• Consider

```c
int x, y
int *z, *w;
if (...) z = &y else z = &x
if (...) w = &y else w = &x
*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:

    int foo(int &x, int &y); //pass by reference!

    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
Choose confluence operator

- 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 ($\sqcup$) of all live sets of outgoing edges

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

- At the end of the program, we know no variables are live → value at exit point is \{ \}

- What about elsewhere in the program?
  - We should initialize other sets to \{ \}
    - This is consistent with our approach to finding the least fixpoint
\[ X = 1 \]

\[ X < N? \]

\[ X = X + Z \]

\[ \text{PRINT}(X) \]
An alternate approach

- Dataflow analyses like live-variable analysis are *bit-vector* analyses: are even more structured than regular dataflow analysis
  - Consistent lattice: powerset
  - Consistent transfer functions
- Many sources only talk about bitvector dataflow
Bit-vector lattices

- Consider a single element, $V$, of the powerset($S$) lattice
- Each item in $S$ either appears in $V$ or does not: can represent using a single bit
  - Can represent $V$ as a *bit vector*
    - $\{a, b, c\} = <1, 1, 1>$
    - $\emptyset = <0, 0, 0>$
    - $\{b, c\} = <0, 1, 1>$
  - $\cup$ and $\cap$ (which are just $\cup$ and $\cap$) are simply bitwise $\lor$ and $\land$, respectively
Eliminating merge nodes

- Many dataflow presentations do not use explicit merge nodes in CFG
- How do we handle this?
- Problem: now a node may be a statement and a merge point
- Solution: compose confluence operator and transfer functions
- Note: non-merge nodes have just one successor; this equation works for all nodes!

\[ T(s) = \text{use}(s) \cup \left( \bigcup_{X \in \text{succ}(s)} X \right) - \text{def}(s) \]
Simplifying matters

\[ T(s) = \text{use}(s) \cup \left( \bigcup_{X \in \text{succ}(s)} X \right) - \text{def}(s) \]

• Lets split this up into two different sets

  • \( \text{OUT}(s) \): the set of variables that are live \textit{immediately after} a statement is executed

  • \( \text{IN}(s) \): the set of variables that are live \textit{immediately before} a statement is executed

\[
\begin{align*}
\text{IN}(s) &= \text{use}(s) \cup (\text{OUT}(s) - \text{def}(s)) \\
\text{OUT}(s) &= \bigcup_{t \in \text{succ}(s)} \text{IN}(t)
\end{align*}
\]
Generalizing

- USE(s) are the variables that become live due to a statement—they are *generated* by this statement
- DEF(s) are the variables that stop being live due to a statement—they are *killed* by this statement

\[
\begin{align*}
IN(s) & = \text{gen}(s) \cup (OUT(s) - \text{kill}(s)) \\
OUT(s) & = \bigcup_{t \in \text{succ}(s)} IN(t)
\end{align*}
\]
Bit-vector analyses

- A bit-vector analysis is any analysis that
  - Operates over the powerset lattice, ordered by $\subseteq$ and with $\cup$ and $\cap$ as its meet and join
  - Has transfer functions that can be written in the form:
    
    $$
    \begin{align*}
    IN(s) &= \text{gen}(s) \cup (OUT(s) - \text{kill}(s)) \\
    OUT(s) &= \bigcup_{t \in \text{succ}(s)} IN(t)
    \end{align*}
    $$

- Are these transfer functions monotonic? (Hint: if $f$ and $g$ are monotonic, is $f \circ g$ monotonic?)

- gen and kill are dependent on the statement, but not on IN or OUT

- Things are a little different for forward analyses, and some analyses use $\cap$ instead of $\cup$
Reaching definitions

• What definitions of a variable *reach* a particular program point

• A definition of variable \( x \) from statement \( s \) reaches a statement \( t \) if there is a path from \( s \) to \( t \) where \( x \) is not redefined

• Especially important if \( x \) is used in \( t \)

• Used to build *def-use* chains and *use-def* chains, which are key building blocks of other analyses

  • Used to determine dependences: if \( x \) is defined in \( s \) and that definition reaches \( t \) then there is a flow dependence from \( s \) to \( t \)

• We used this to determine if statements were loop invaraint

  • All definitions that reach an expression must originate from outside the loop, or themselves be invariant
Creating a reaching-def analysis

• Can we use a powerset lattice?

• At each program point, we want to know which definitions have reached a particular point
  
  • Can use powerset of set of definitions in the program

  • $V$ is set of variables, $S$ is set of program statements

  • Definition: $d \in V \times S$

  • Use a tuple, $<v, s>$

• How big is this set?

  • At most $|V \times S|$ definitions
Forward or backward?

• What do you think?
Choose confluence operator

- Remember: we want to know if a definition *may* reach a program point
- What happens if we are at a merge point and a definition reaches from one branch but not the other?
  - We don’t know which branch is taken!
  - We should union the two sets – any of those definitions can reach
- We want to avoid getting too many reaching definitions → should start sets at \( \perp \)
Transfer functions

- Forward analysis, so need a slightly different formulation
  - Merged data flowing into a statement
    \[
    \begin{align*}
    IN(s) &= \bigcup_{t \in \text{pred}(s)} OUT(t) \\
    OUT(s) &= gen(s) \cup (IN(s) \setminus \text{kill}(s))
    \end{align*}
    \]

- What are gen and kill?
  - gen(s): the set of definitions that may occur at s
    - e.g., gen(s₁: x = e) is <s₁, x>
  - kill(s): all previous definitions of variables that are definitely redefined by s
    - e.g., kill(s₁: x = e) is <*, x>
Available expressions

• We’ve seen this one before
• What is the lattice? powerset of all expressions appearing in a procedure
• Forward or backward?
• Confluence operator?
Transfer functions for meet

- What do the transfer functions look like if we are doing a meet?

\[
\begin{align*}
IN(S) & = \bigcap_{t \in pred(s)} OUT(t) \\
OUT(S) & = \text{gen}(s) \cup (IN(S) - \text{kill}(s))
\end{align*}
\]

- gen(s): expressions that \textit{must be} computed in this statement
- kill(s): expressions that use variables that \textit{may} be defined in this statement
  - Note difference between these sets and the sets for reaching definitions or liveness
- Insight: gen and kill must never lead to incorrect results
  - Must not decide an expression is available when it isn’t, but OK to be safe and say it isn’t
  - Must not decide a definition \textit{doesn’t} reach, but OK to overestimate and say it does
Analysis initialization

- Remember our formalization
  - If we start with everything initialized to $\bot$, we compute the least fixpoint
  - If we start with everything initialized to $\top$, we compute the greatest fixpoint
- Which do we want? It depends!
  - Reaching definitions: a definition that *may* reach this point
    - We want to have as few reaching definitions as possible $\rightarrow$ use least fixpoint
  - Available expressions: an expression that *was definitely* computed earlier
    - We want to have as many available expressions as possible $\rightarrow$ use greatest fixpoint
- Rule of thumb: if confluence operator is $\sqcup$, start with $\bot$, otherwise start with $\top$
Analysis initialization (II)

- The set at the entry of a program (for forward analyses) or exit of a program (for backward analyses) may be different.
  - One way of looking at this: start statement and end statement have their own transfer functions.
- General rule for bitvector analyses: no information at beginning of analysis, so first set is always \( \{ \} \).
Very busy expressions

- An expression is *very busy* if it is computed on *every path* that leads from a program point

- Why does this matter?

- Can calculate very busy expressions early without wasting computation (since the expression is used at least once on every outgoing path) – this can save space

- Good candidates for loop invariant code motion
Very busy expressions

- Lattice?
- Direction?
- Confluence operator?
- Initialization?
- Transfer functions?
  - Gen? Kill?
Four types of dataflow

- Analysis can either be forward or backward
- Analysis can either be over all paths or over any path
  - All paths: merges consider values from all paths
  - Any path: merges consider values from any path

<table>
<thead>
<tr>
<th></th>
<th>All paths</th>
<th>Any path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>available expressions</td>
<td>reaching definitions</td>
</tr>
<tr>
<td>Backward</td>
<td>very busy expressions</td>
<td>liveness analysis</td>
</tr>
</tbody>
</table>

- What kind of analysis is constant propagation?