Register allocation:

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult
  $\Rightarrow$ NP-complete for $k \geq 1$ registers
Liveness analysis

Problem:

- IR contains an unbounded number of temporaries
- machine has bounded number of registers

Approach:

- temporaries with disjoint live ranges can map to same register
- if not enough registers then spill some temporaries (i.e., keep them in memory)

The compiler must perform liveness analysis for each temporary:

It is live if it holds a value that may be needed in future
Control flow analysis

Before performing liveness analysis, need to understand the control flow by building a *control flow graph* (CFG):

- nodes may be individual program statements or basic blocks
- edges represent potential flow of control

*Out-edges* from node $n$ lead to *successor* nodes, $\text{succ}[n]$

*In-edges* to node $n$ come from *predecessor* nodes, $\text{pred}[n]$

Example:

\[
\begin{align*}
L_1 : & \quad a \leftarrow 0 \\
& \quad b \leftarrow a + 1 \\
& \quad c \leftarrow c + b \\
& \quad a \leftarrow b \times 2 \\
& \quad \text{if } a < N \text{ goto } L_1 \\
& \quad \text{return } c
\end{align*}
\]
Liveness analysis

Gathering liveness information is a form of data flow analysis operating over the CFG:

- liveness of variables “flows” around the edges of the graph
- assignments define a variable, \( v \):
  - \( \text{def}(v) \) = set of graph nodes that define \( v \)
  - \( \text{def}[n] \) = set of variables defined by \( n \)
- occurrences of \( v \) in expressions use it:
  - \( \text{use}(v) \) = set of nodes that use \( v \)
  - \( \text{use}[n] \) = set of variables used in \( n \)

Liveness: \( v \) is live on edge \( e \) if there is a directed path from \( e \) to a use of \( v \) that does not pass through any \( \text{def}(v) \)

\( v \) is live-in at node \( n \) if live on any of \( n \)'s in-edges

\( v \) is live-out at \( n \) if live on any of \( n \)'s out-edges

\( v \in \text{use}[n] \Rightarrow v \) live-in at \( n \)

\( v \) live-in at \( n \Rightarrow v \) live-out at all \( m \in \text{pred}[n] \)

\( v \) live-out at \( n, v \not\in \text{def}[n] \Rightarrow v \) live-in at \( n \)
Liveness analysis

Define:

\( in[n] \): variables live-in at \( n \)
\( in[n] \): variables live-out at \( n \)

Then:

\[
out[n] = \bigcup_{s \in succ(n)} in[s]
\]

\( succ[n] = \emptyset \Rightarrow out[n] = \emptyset \)

Note:

\( in[n] \supseteq use[n] \)

\( in[n] \supseteq out[n] - def[n] \)

\( use[n] \) and \( def[n] \) are constant (independent of control flow)

Now, \( v \in in[n] \) iff. \( v \in use[n] \) or \( v \in out[n] - def[n] \)

Thus, \( in[n] = use[n] \cup (out[n] - def[n]) \)
Iterative solution for liveness

```
foreach n
    in[n] ← φ
    out[n] ← φ

repeat
    foreach n
        in'[n] ← in[n];
        out'[n] ← out[n];
        in[n] ← use[n] ∪ (out[n] − def[n])
        out[n] ← ∪_{s ∈ succ[n]} in[s]
until in'[n] = in[n] ∧ out'[n] = out[n], ∀n
```

Notes:

- should order computation of inner loop to follow the “flow”
- liveness flows backward along control-flow arcs, from `out` to `in`
- nodes can just as easily be basic blocks to reduce CFG size
- could do one variable at a time, from `uses` back to `defs`, noting liveness along the way
Iterative solution for liveness

Complexity: for input program of size $N$

- $\leq N$ nodes in CFG
  - $\Rightarrow \leq N$ variables
  - $\Rightarrow N$ elements per in/out
  - $\Rightarrow O(N)$ time per set-union

- **for** loop performs constant number of set operations per node
  - $\Rightarrow O(N^2)$ time for **for** loop

- each iteration of **repeat** loop can only add to each set
  - sets can contain at most every variable
  - $\Rightarrow$ sizes of all in and out sets sum to $2N^2$,
    bounding the number of iterations of the **repeat** loop
  - $\Rightarrow$ worst-case complexity of $O(N^4)$

- ordering can cut **repeat** loop down to 2-3 iterations
  - $\Rightarrow O(N)$ or $O(N^2)$ in practice
Iterative solution for liveness

Least fixed points

There is often more than one solution for a given dataflow problem (see example).

Any solution to dataflow equations is a conservative approximation:

- \( v \) has some later use downstream from \( n \)
  \[ \Rightarrow v \in \text{out}(n) \]

- but not the converse

Conservatively assuming a variable is live does not break the program; just means more registers may be needed.

Assuming a variable is dead when it is really live will break things.

May be many possible solutions but want the “smallest”: the least fixpoint.

The iterative liveness computation computes this least fixpoint.
Another DF analysis example

• Problem: given a program, identify all possible null pointer dereference errors.

```
1. p=&A;
2. i=0;
3. While (i<N) {
4.   sum=sum+*p;
5.   if (i>3)
6.     p=0;
   else
7.     p++;
8.   i++
}
```

There is a null pointer dereference error in the code snippet on the left, when the program takes the path 6-8-3-4.
• A naïve solution: identify all pointer dereference points, for each deref point, enumerate all backward paths from the point to see if a null assignment (def) can be encountered without encountering another def.
  — Problem: path explosion and loops

• Data flow equation:
  \[
  \text{IN}[n] = \bigcup_{p \in \text{pred}(n)} \text{OUT}[p] \\
  \text{OUT}[n] = (\text{IN}[n] - \text{all null defs}) \cup (\text{if } n \text{ is a null def then } \{n\} \text{ else } \{\})
  \]

• Full algorithm:

  Initialize IN[] and OUT[] to {}
  changed = 1;
  While (changed) {
    changed = 0
    for (each node n in topological order)
      update IN[n] and OUT[n] according to the above equations.
      changed = new IN[n]/OUT[n] is observed
  }
• Proof of Termination:

\[ \text{IN}[n] = \bigcup_{p \in \text{pred}(n)} \text{OUT}[p] \]
\[ \text{OUT}[n] = (\text{IN}[n] - \text{all null defs } ) \cup (\text{if n is a null def then \{n\} else \{\}}) \]

The set of all null defs and the set (if n is a null def then \{n\} else\{\}) are constants regarding a specific n, let's represent them as Kill and Gen, the equation becomes:

\[ \text{IN}[n] = \bigcup_{p \in \text{pred}(n)} \text{OUT}[p] \]
\[ \text{OUT}[n] = (\text{IN}[n] - \text{Kill}) \cup \text{Gen} \]

Since they are constant, the two equations are monotonic, meaning \text{IN}[n] increases if \text{OUT}[p] increases, and vice versa.
And, the maximal value of \text{IN}[n] and \text{OUT}[n] is bounded.