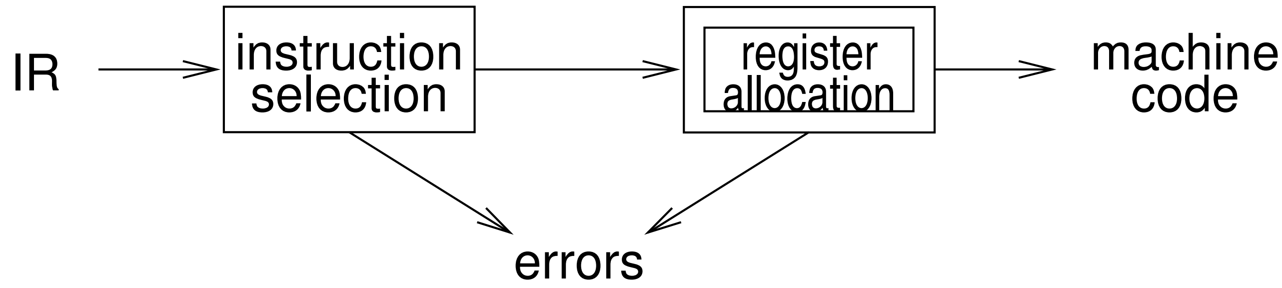
Register allocation



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 which are not needed at the same time can map to same register
- if not enough registers then *spill* some temporaries (i.e., keep them in memory)

The compiler performs a *liveness analysis* for each temporary:

- a temporary is *live* if it holds a value that may be needed in future
- temporaries with disjoint *live ranges* can map to same register

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
- an edge from n to n' represents a potential control transfer from (the end of) n to (the beginning of) n'

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

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

Example:

$$
\begin{array}{rl}
 & a \leftarrow 0 \\
L_1: & b \leftarrow a+1 \\
 & c \leftarrow c+b \\
 & a \leftarrow b \times 2 \\
 & \text{if } a < N \text{ goto } L_1 \\
 & \text{return } c
\end{array}
$$

Liveness analysis

Liveness analysis is a *data flow analysis* operating on the CFG:

- liveness of variables “flows” along the edges of the graph
- an assignment *defines* a variable, v:
  - *def*$(v) =$ set of graph nodes that define v
  - *def*$[n] =$ set of variables defined by n
- an occurrence of v in an expression *uses* it:
  - *use*$(v) =$ set of nodes that use v
  - *use*$[n] =$ set of variables used in n

Definition (*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 *def*(v)

v is *live-in* at node n if v is live on any of n’s in-edges

v is *live-out* at n if v is live on any of n’s out-edges

$v \in \mathit{use}[n] \Rightarrow v$ live-in at n

v live-in at $n \Rightarrow v$ live-out at all $m \in \mathit{pred}[n]$

v live-out at $n, v \notin \mathit{def}[n] \Rightarrow v$ live-in at n

Liveness analysis

Define:

in$[n]$: set of variables live-in at n

out$[n]$: set of variables live-out at n

Then:

$$\mathit{out}[n] = \bigcup_{s \in \mathit{succ}[n]} \mathit{in}[s]$$

$$\mathit{succ}[n] = \emptyset \Rightarrow \mathit{out}[n] = \emptyset$$

Note:

$$\mathit{in}[n] \supseteq \mathit{use}[n]$$

$$\mathit{in}[n] \supseteq \mathit{out}[n] - \mathit{def}[n]$$

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

Now, $v \in \mathit{in}[n]$ iff. $v \in \mathit{use}[n]$ or $v \in \mathit{out}[n] - \mathit{def}[n]$

Thus, $\mathit{in}[n] = \mathit{use}[n] \cup (\mathit{out}[n] - \mathit{def}[n])$

Iterative computation for liveness information

foreach n

  in$[n] \leftarrow \emptyset$

  out$[n] \leftarrow \emptyset$

repeat

    foreach node n

      in$'[n] \leftarrow$ in$[n]$;

      out$'[n] \leftarrow$ out$[n]$;

      in$[n] \leftarrow$ use$[n] \cup ($out$[n] -$ def$[n])$

      out$[n] \leftarrow \bigcup_{s \in \text{succ}[n]}$ in$[s]$

until in$'[n] =$ in$[n] \wedge$ out$'[n] =$ out$[n], \forall n$

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*
  $\Rightarrow \mathrm{O}(N)$ time per set-union
- **for** loop performs constant number of set operations per node
  $\Rightarrow \mathrm{O}(N^2)$ 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 $\mathrm{O}(N^4)$

- ordering can cut **repeat** loop down to 2-3 iterations
  $\Rightarrow \mathrm{O}(N)$ or $\mathrm{O}(N^2)$ in practice

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
  $\Rightarrow v \in \mathit{out}(n)$
- 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.