- we have instruction sequence – almost there…
  - just need to figure out the registers
- instruction sequence may use hundreds or temporaries – but we only have n registers
- need register allocation algorithm
- before – need to make some analysis

Liveness analysis

- key to register allocation determine which temporaries can be bound to same register
  - variables which are not “in use” at the same time
- variable a is “live” if the value it holds is/will be used
- register which holds a “live” variable cannot be allocated to another var
  - if all “live” vars don’t fit in registers, the excess can be kept in memory

---

- In order to perform liveness analysis we need control-flow graph
  - CFG elements – statements, instructions, basic blocks…
- Live range – segment of the graph where variable is live

```
a <- 0
L1:
b <- a+1
c <- c+b
a <- b*2
if a < N goto L1
return c
```

- var v is live at stmt/inst/block I in graph G if:
  - there is a path from I to a use of v which does not include assignments of v
    - live range of v starts half-way through assignment
    - live range of v ends half-way into the assignment
    - an assignment *defines* a variable/temporary

- `def(v)` – set of graph nodes where v is defined
- `def(I)` – set of variables defined at I
- `use(I)` – set of variables used at I (rhs, other exp).
- `use(v)` – set of nodes at which v is used

- *liveness*: a variable is live at edge e in G if there is a directed path from a `use` of the variable that does not go through any `defs`

Calculating liveness:

- since live ranges start/end “half-way” need to look at edges in and out of stmts/insts/blocks in the flow graph.
- `live-in(I)` variables live at the top of I
- `live-out(I)` variables live on *some* edges going out of I
  - var may not be used after block I
  - instruction following I may rely on sideeffects

```
in[n] = use[n] U (out[n]-def[n])
out[n] = U in[s], where s is in succ[n]

foreach I do
  in[n] <- out[n] <- {}
repeat
  foreach n
     in’[n]<- in[n], out’[n]<- out[n]
     in[n]<- use[n] U (out[n]- def[n])
     out[n]<- U in[s], where s in succ[n]
until in’[n]==in[n], out’[n]==out[n]
```

```
foreach I do
  in[n] <- out[n] <- {}
  WL <- WL U n

while WL <> 0
     remove n from WL
     in[n]<- use[n] U (out[n]- def[n])
     out[n]<- U in[s], where s in succ[n]
     if in[n] changed, then WL <- WL U pred[n]
```

- could have algorithms that start pessimistically, and then improve estimate by eliminating registers which are not going to be necessary

---

- computation of algorithm converges more quickly if in opposite direction of control flow
  - do loops "backwards"
- consider basic blocks at a time, not individual instructions
  - block size?
  - need to know what really needs to be live at end of block
- many variables short-lived – algorithm will converge quickly for those, so perform it one variable at a time

---

- representing the sets in[], out[]…
  - bit arrays: N variables in N/K words; union == OR; good for dense sets
  - linked lists: union == merge – duplicates, good for sparse sets (<< N/K)
- iterations will stop:
  - each iteration may only increase sets (or not change sets and stop)
  - set sizes are limited by N
  - worst-case cost O(N^4), in practice O(n) – O(n^2)

Conservative approximation for liveness:

- cannot be sure that a variable will be used, so it's safe to be conservative and keep it in live sets
- dynamic liveness: for *some execution* we go from I to use(v) without defs
- static liveness: there is *some path* from I to use(v) without defs
- *in general* cannot determine reachability of use(v)
  - or label, or end of program (halting problem)

Halting problem

- there is no program H that takes any program P and input X and H(P,X) returns true if P(X) halts and H(P,X) returns false if P(X) infinite-loops
- Suppose H exists.
- define F(Y) = if H(Y,Y) then (while true do ()) else true
- if F(F) halts, then H(F,F) is true, but by def F(F) loops!
- if F(F) loops, then H(F,F) is false, but by def F(F) halts (and returns true)
- same: no program H can tell for any program X that a label L within the program can be reached

---

Interference graph

- Liveness analysis used for register allocation:
  - if vars a and b are live at the same point I, they cannot be allocated to same register

  ⇒a and b interfere

  - if a used by instruction i which cannot use register r => cannot allocate a to r, and a and r interfere

- Construct undirected graph: nodes are variables, and edges connect variables that interfere.
- each pair of adjacent nodes must be allocated in different registers
- register allocation translates to graph coloring problem

- at any nonmove instruction that defines a, with live-out vars b1, .. bj, add interference edges (a, b1), .. (a, bj)
- at a move instruction a <- c, with live-out vars b1, .. bj, add interference edges (a, bi), for any bi <> c.

- move is reg move x <- y
- add edge later, if assign appears, while other var still live

Data Flow Analysis

- Liveness analysis – DFA needed to perform register allocation
- DFA can be performed for other reasons:
  - analyze/traverse flow graph and keep track of a (set of) property (information)
  - use info gathered to modify the program in some consistent way

  - (Chapter 17)

Pair analysis and transformation

- Liveness & register allocation
- Liveness & dead code elimination
- Reaching definitions & constant propagation
- Available expressions & common subexpression elimination

The objective…

- Perform certain optimizations
- Current representation of program is correct, but we'd like to
  - reduce temporaries, variables
  - eliminate unnecessary operations
  - …

Dataflow Equations

- Liveness

in[n] = gen[n] U (out[n] – kill[n])

out[n] = U (in[s]) for s in succ[n]

gen – set of generated live vars (use)

kill – set of vars for which liveness property no longer holds (newly def)

- starts with empty sets
- builds sets for which liveness holds at n -> cannot terminate early
- the flow is analyzed backwards

Other DFAs

- Similar equations
- gen and kill sets at each nodes which refer to statements/variables for which property is valid
- in/out sets and `transfer' function
- `confluence operator' – how are resulting sets processed together (union or intersection)
  - depends on property which is analyzed
- may be backward or forward
- increasing or decreasing (pessimistic/optimistic)

Reaching definitions

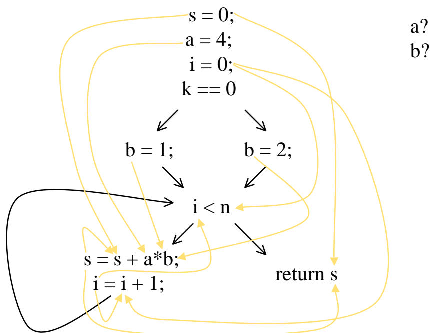
in[n] = U out[p], where p is in pred[n]

out[n] = gen[n] U (in[n] – kill[n])

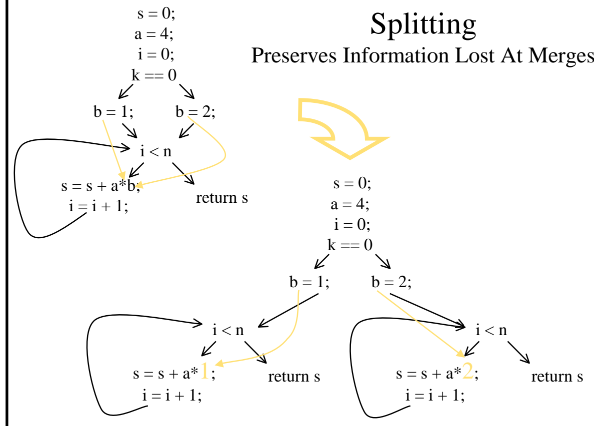
s	gen[n]	kill[s]
d: t ← b op c	{d}	defs(t)-{d}

in[n], out[n]=?

Reaching Definitions



a?
b?



Available expressions

- will want to remove them,
  - so have to be conservative not to remove something that's needed
  - still want to find the largest set

in[n] = ∩ out[p], where p is in pred[n]

out[n] = gen[n] U (in[n] – kill[n])

in[n], out[n]=?

s: t ← b op c

gen: {b op c} – kill [s]

kill: expressions with t

- Liveness for dead code elimination
- Very busy expressions - expression is v.b. at the exit from a label if, no matter what path is taken from the label, the expression is always used before any of the variables occurring in it are redefined.
  - Code hoisting finds expressions that are always evaluated following some point in the program regardless of the execution path — and moves them to the latest point beyond which they would always be executed.