More Examples of Abstract Interpretation

Register Allocation

1) Constant Propagation

Special case of interval analysis:

$D = \{ [a,a] \mid a \in \{-M,-(M-1),\ldots,-2,-1,0,1,2,3,\ldots,M-1\}\}$
$\quad\cup \{\perp, \top\}$

Write [a,a] simply as a. So values are:

- a known constant at this point: a
- "we could not show it is constant": $\top$
- "we did not reach this program point": $\perp$

Convergence fast - lattice has small height

Transfer Function for Plus in pscala

$$\begin{tikzcd} v_1 \arrow[d] \\ v_2 \end{tikzcd}$$

x = y + z

For each variable (x,y,z) we store a constant $\perp$, or T

table for +:

z \ y	$\perp$	C_y	T
$\perp$			
C_z			
T			

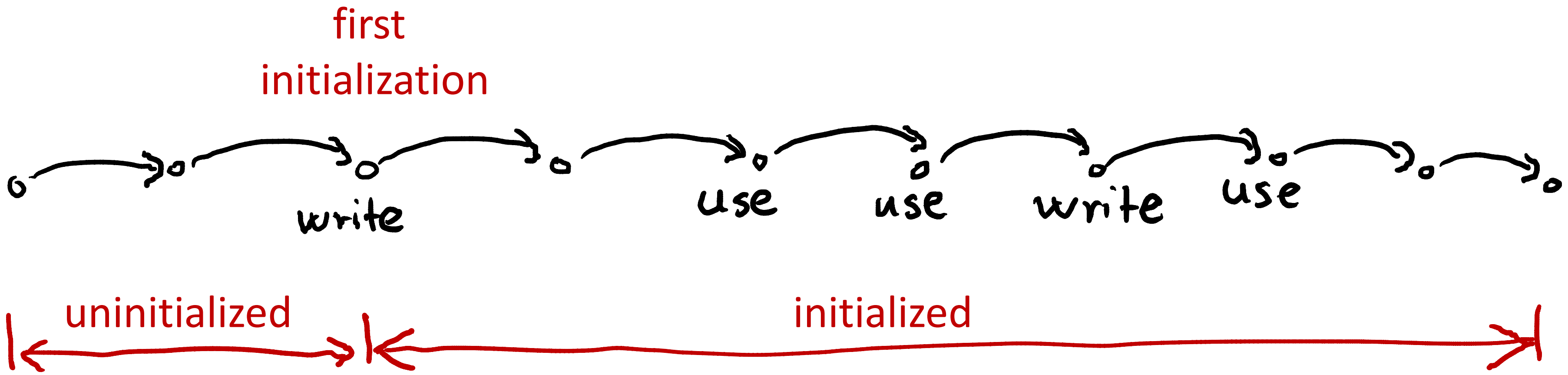
```
abstract class Element
case class Top extends Element
case class Bot extends Element
case class Const(v:Int) extends Element
var facts : Map[Nodes,Map[VarNames,Element]]
```

what executes during analysis:

```
oldY = facts(v1)("y")
oldZ = facts(v1)("z")
newX = tableForPlus(oldY, oldZ)
facts(v2) = facts(v2) join facts(v1).updated("x", newX)
```

```
def tableForPlus(y:Element, z:Element) =
(x,y) match {
case (Const(cy),Const(cz)) => Const(cy+cz)
case (Bot,_) => Bot
case (_,Bot) => Bot
case (Top,Const(cz)) => Top
case (Const(cy),Top) => Top
}
```

2) Initialization Analysis



What does javac say to this:

```
class Test {
    static void test(int p) {
            int n;
            p = p - 1;
            if (p > 0) {
                n = 100;
            }
            while (n != 0) {
                System.out.println(n);
                n = n - p;
            }
    }
}
```

```
Test.java:8: variable n might not have been initialized
            while (n > 0) {
            ^
1 error
```

Program that compiles in java

```
class Test {
   static void test(int p) {
            int n;
            p = p - 1;
            if (p > 0) {
               n = 100;
            }
            else {
               n = -100;
            }
            while (n != 0) {
               System.out.println(n);
               n = n - p;
            }
   }
} // Try using if (p>0) second time.
```

We would like variables to be initialized on all execution paths.

Otherwise, the program execution could be undesirable affected by the value that was in the variable initially.

We can enforce such check using initialization analysis.

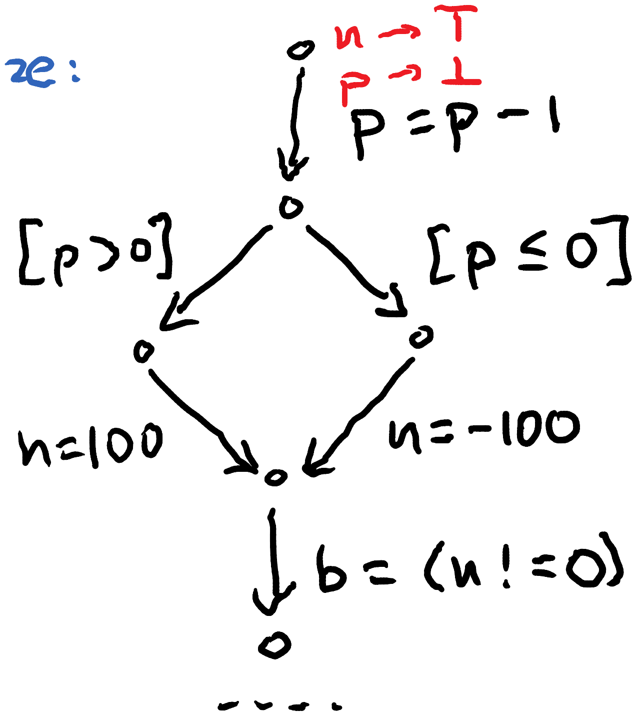
Initialization Analysis

```
class Test {
  static void test(int p) {
        int n;
        p = p - 1;
        if (p > 0) {
          n = 100;
        }
        else {
          n = -100;
        }
        while (n != 0) {
          System.out.println(n);
          n = n - p;
        }
  }
} // Try using if (p>0) second time.
```

T indicates presence of flow from states where variable was not initialized:

- If variable is possibly uninitialized, we use T
- Otherwise (initialized, or unreachable): ⊥

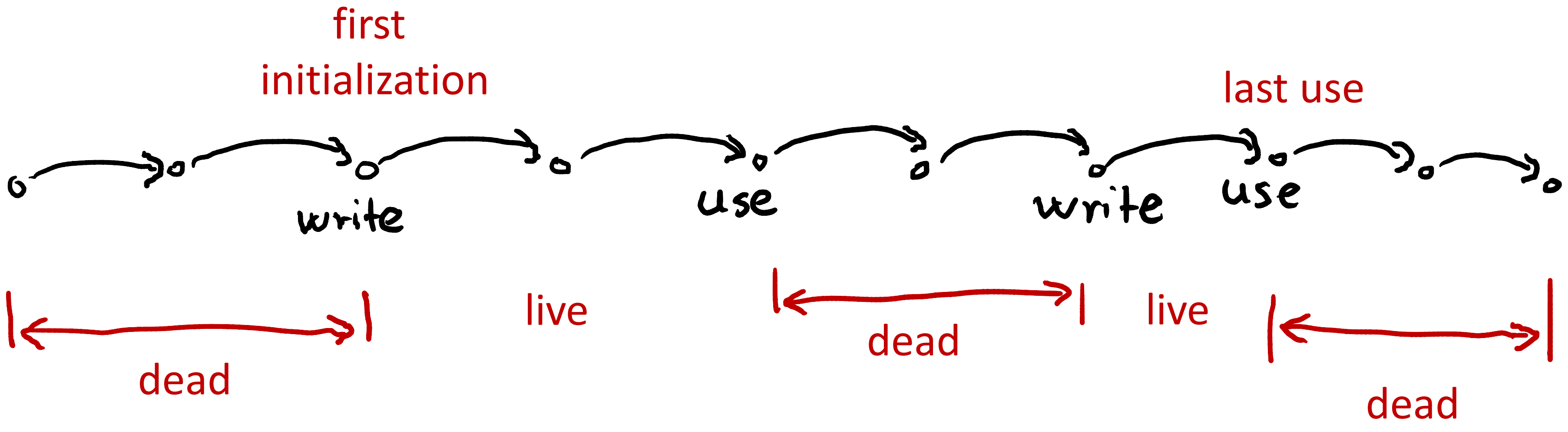
analyze:



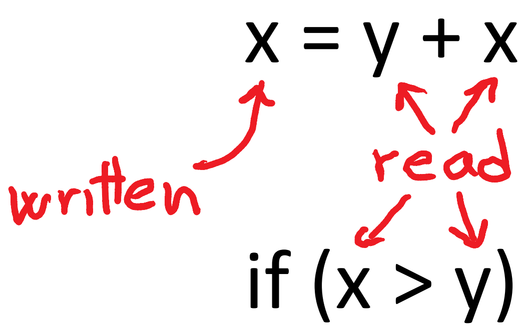
If var occurs anywhere but left-hand side of assignment and has value T, report error

3) Liveness Analysis

Variable is dead if its current value will not be used in the future. If there are no uses before it is reassigned or the execution ends, then the variable is sure dead at a given point.



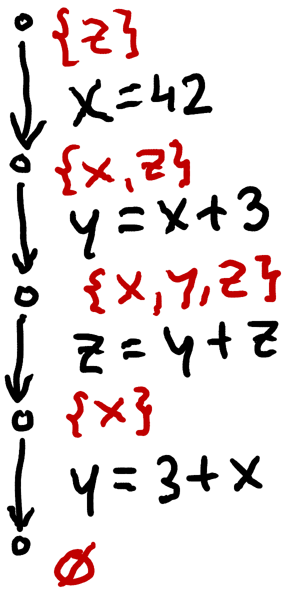
What is Written and What Read



```
x = y + x

if (x > y)
```

Example:



```
{z}
x=42
{x,z}
y=x+3
{x,y,z}
z=y+z
{x}
y=3+x
∅
```

Purpose:

Register allocation:

find good way to decide which variable should go to which register at what point in time.

How Transfer Functions Look

L_i - set of live variables

$$\begin{tikzcd}
\circ \; L_0 \arrow[d, "x = x + y \; (st)"] \\
\circ \; L_2
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\circ \; L_0 = L_1 \cup \{x,y\} \arrow[d, "\text{read}(x,y)"] \\
\circ \; L_1 = L_2 \setminus \{x\} \arrow[d, "\text{write}(x)"] \\
\circ \; L_2
\end{tikzcd}$$

$$L_0 = (L_2 \setminus \{x\}) \cup \{x,y\}$$

Generally

$$L_0 = (L_2 \setminus \text{def}(st)) \cup \text{use}(st)$$

Initialization: Forward Analysis

```
while (there was change)
  pick edge (v1,statmt,v2) from CFG
         such that facts(v1) has changed
  facts(v2)=facts(v2) join transferFun(statmt, facts(v1))
}
```

Liveness: Backward Analysis

```
while (there was change)
  pick edge (v1,statmt,v2) from CFG
         such that facts(v2) has changed
  facts(v1)=facts(v1) join transferFun(statmt, facts(v2))
}
```

Example

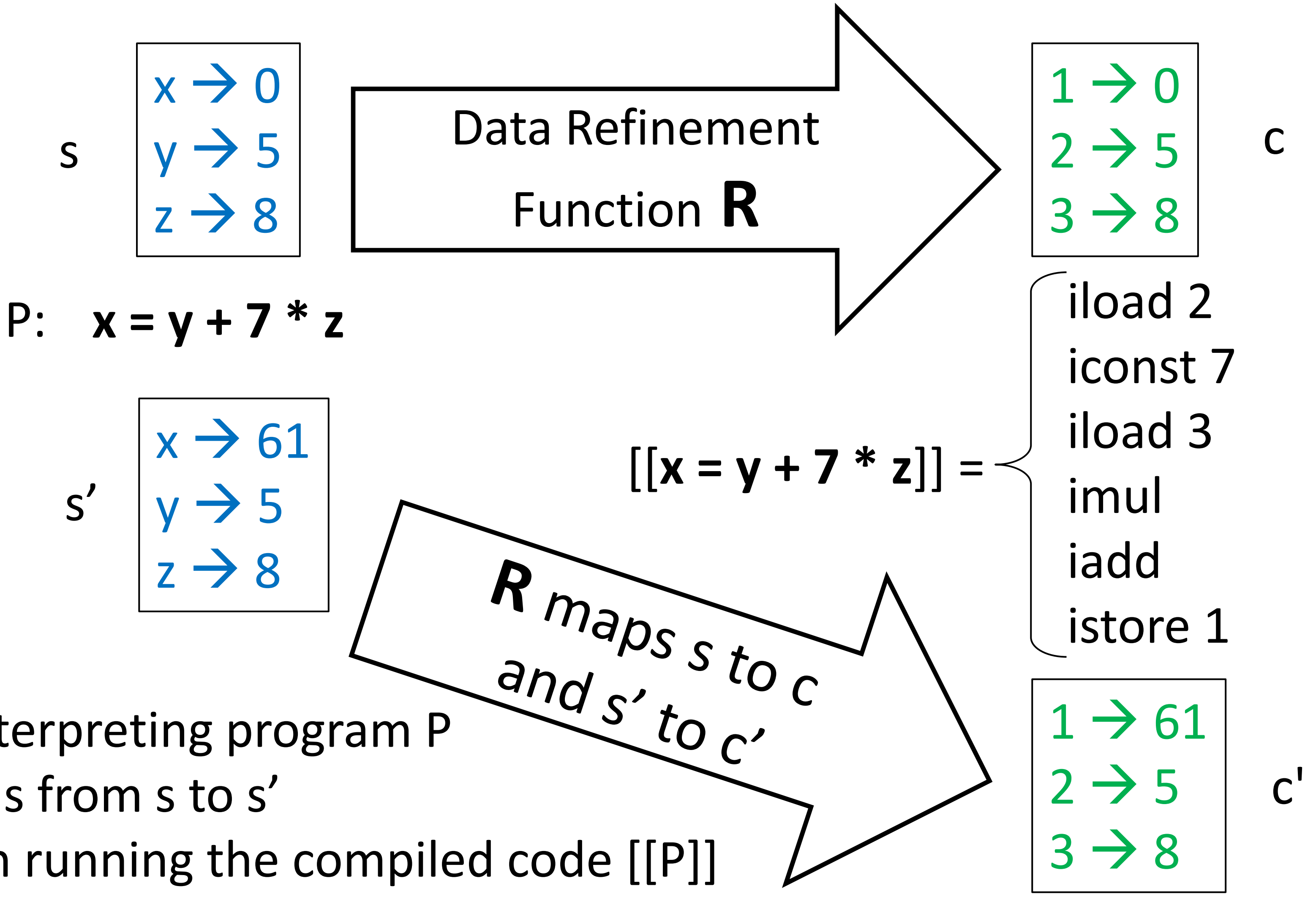
```
x = m[0]
y = m[1]
xy = x * y
z = m[2]
yz = y*z
xz = x*z
res1 = xy + yz
m[3] = res1 + xz
```

4) Data Representation Overview

Original and Target Program have Different Views of Program State

- Original program:
  - local variables given by names (any number of them)
  - each procedure execution has fresh space for its variables (even if it is recursive)
  - fields given by names
- Java Virtual Machine
  - local variables given by slots (0,1,2,…), any number
  - intermediate values stored in operand stack
  - each procedure **call** gets fresh slots and stack
  - fields given by names and object references
- **Machine code:** program state is a large arrays of bytes and a finite number of registers

Compilation Performs Automated Data Refinement



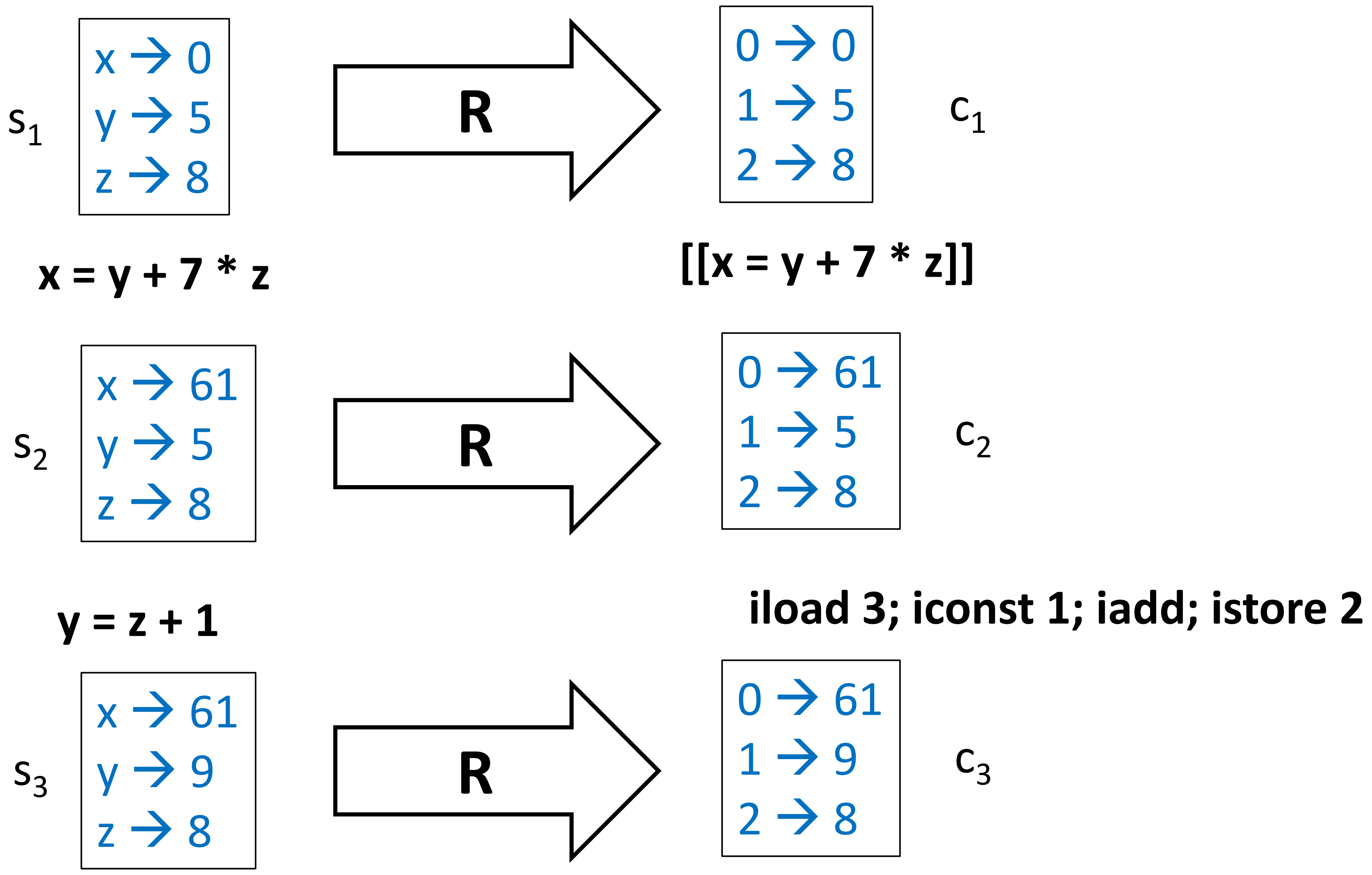
P: **x = y + 7 * z**

[[**x = y + 7 * z**]] =

```
iload 2
iconst 7
iload 3
imul
iadd
istore 1
```

If interpreting program P
leads from s to s’
then running the compiled code [[P]]
leads from R(s) to R(s’)

Inductive Argument for Correctness



(R may need to be a relation, not just function)

A Simple Theorem

$P : S \rightarrow S$	is a program meaning function
$P_c : C \rightarrow C$	is meaning function for the compiled program
$R : S \rightarrow C$	is data representation function

Let $s_{n+1} = P(s_n)$, $n = 0,1,\ldots$ be interpreted execution

Let $c_{n+1} = P(c_n)$, $n = 0,1,\ldots$ be compiled execution

$$\begin{tikzcd}
s_0 \arrow[r, "P"] \arrow[d, "R"'] & s_1 \arrow[r] \arrow[d, dashed, "R"'] & s_2 \arrow[r] \arrow[d, dashed, "R"'] & s_3 & \cdots & s_n \arrow[d, dashed, "R"] \\
c_0 \arrow[r, "P_c"'] & c_1 \arrow[r] & c_2 \arrow[r] & c_3 & \cdots & c_n
\end{tikzcd}$$

Theorem: If

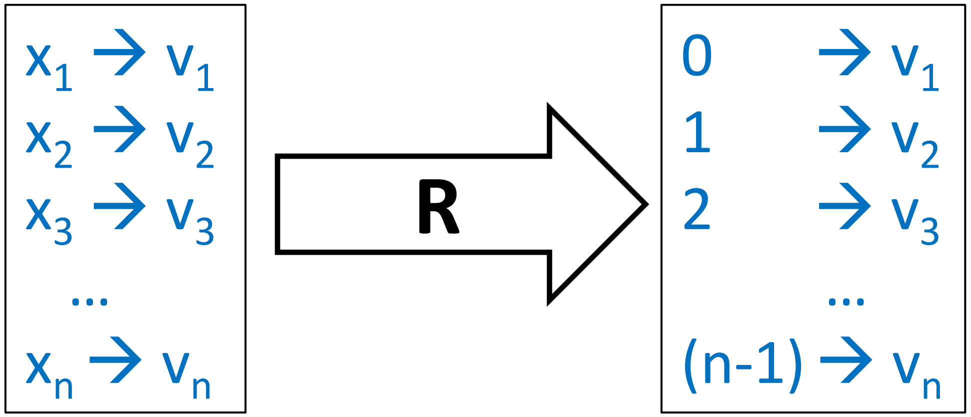
- $c_0 = R(s_0)$
- for all s, $P_c(R(s)) = R(P(s))$

then $c_n = R(c_n)$ for all n.

Proof: immediate, by induction. R is often called **simulation relation.**

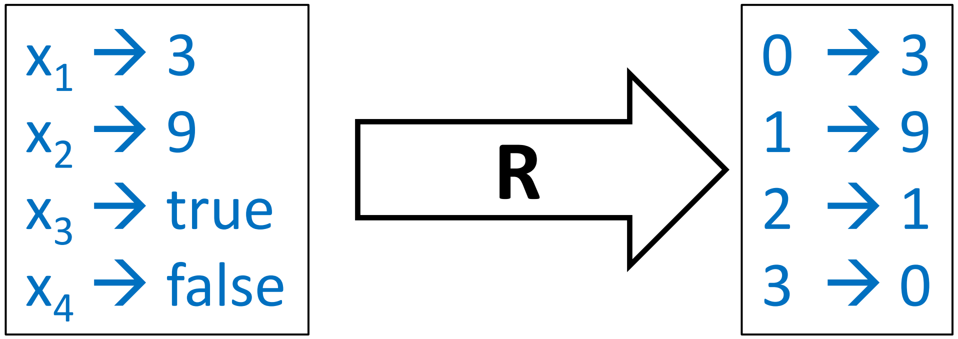
Example of a Simple R

- Let the received, the parameters, and local variables, in their order of declaration, be $x_1, x_2 \ldots x_n$
- Then R maps program state with only integers like this:



R for Booleans

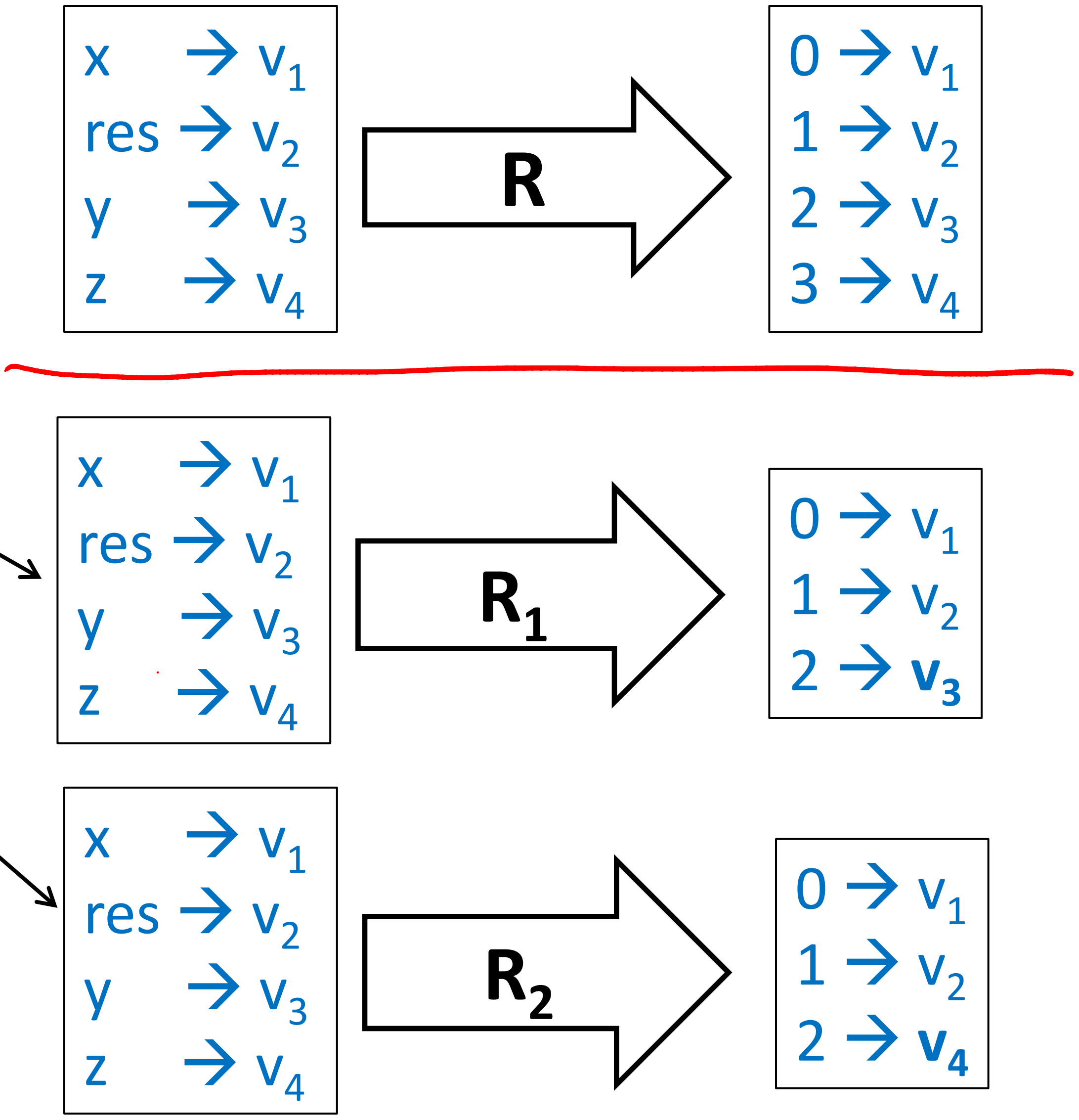
- Let the received, the parameters, and local variables, in their order of declaration, be $x_1, x_2 \dots x_n$
- Then R maps program state like this, where x_1 and x_2 are integers but x_3 and x_4 are Booleans:



R that depends on Program Point

```
def main(x:Int) {
  var res, y, z: Int
  if (x>0) {
    y = x + 1
    res = y
  } else {
    z = -x - 10
    res = z
  }
  …
}
```

Map y,z to same slot.
Consume fewer slots!



Packing Variables into Memory

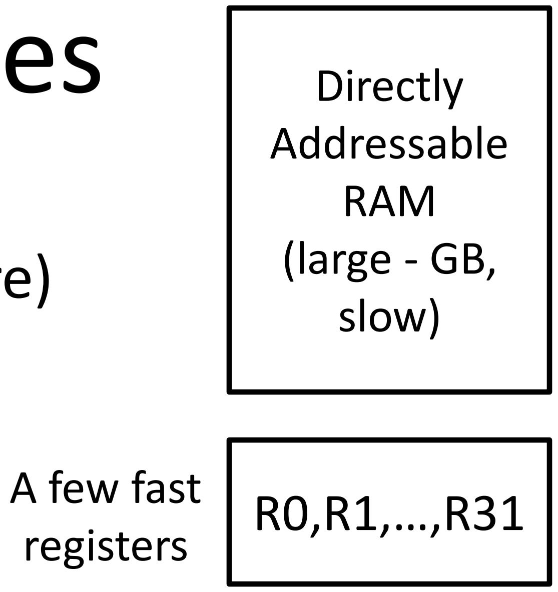
- If values are not used at the same time, we can store them in the same place
- This technique arises in
  - **Register allocation**: store frequently used values in a bounded number of fast registers
  - ‘malloc’ and ‘free’ manual memory management: *free* releases memory to be used for later objects
  - Garbage collection, e.g. for JVM, and .NET as well as languages that run on top of them (e.g. Scala)

Register Machines

Better for most purposes than stack machines

- closer to modern CPUs (RISC architecture)
- closer to control-flow graphs
- simpler than stack machine

Example: ARM architecture



Basic Instructions of Register Machines

$R_i \leftarrow \mathrm{Mem}[R_j]$	load
$\mathrm{Mem}[R_j] \leftarrow R_i$	store
$R_i \leftarrow R_j * R_k$	compute: for an operation *

Efficient register machine code uses as few loads and stores as possible.

State Mapped to Register Machine

Both dynamically allocated heap and stack expand

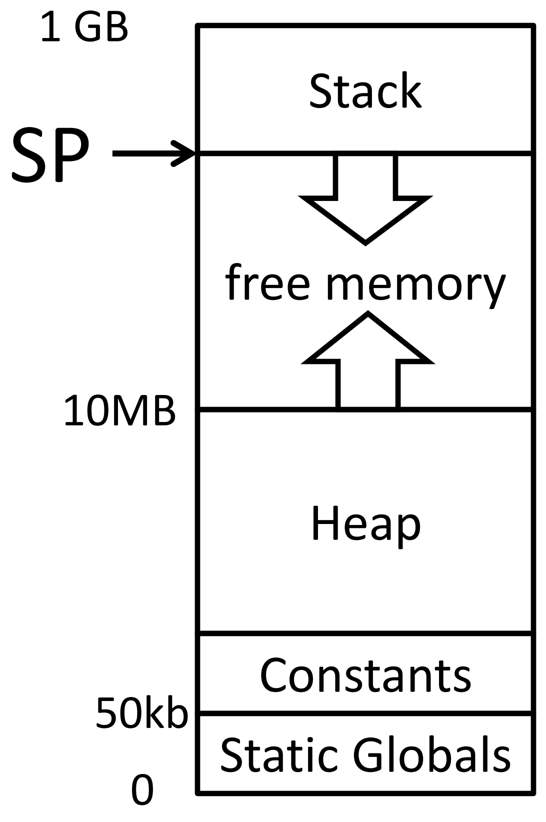
- heap need not be contiguous can request more memory from the OS if needed
- stack grows downwards

Heap is more general:

- Can allocate, read/write, and deallocate, in any order
- Garbage Collector does deallocation automatically
  - Must be able to find free space among used one, group free blocks into larger ones (compaction),…

Stack is more efficient:

- allocation is simple: increment, decrement
- top of stack pointer (SP) is often a register
- if stack grows towards smaller addresses:
  - to allocate N bytes on stack (**push**): **SP := SP - N**
  - to deallocate N bytes on stack (**pop**): **SP := SP + N**



Exact picture may depend on hardware and OS

JVM vs General Register Machine Code

JVM:

```
imul
```

Register Machine:

```
R1 ← Mem[SP]
SP = SP + 4
R2 ← Mem[SP]
R2 ← R1 * R2
Mem[SP] ← R2
```

5) Register Allocation

How many variables?

x,y,z,xy,xz,res1

Do we need 6 distinct registers if we wish to avoid load and stores?

```
x = m[0]
y = m[1]
xy = x * y
z = m[2]
yz = y*z
xz = x*z
res1 = xy + yz
m[3] = res1 + xz
```

```
x = m[0]
y = m[1]
xy = x * y
z = m[2]
yz = y*z
y = x*z            // reuse y
x = xy + yz        // reuse x
m[3] = x + y
```

Idea of Graph Coloring

- Register Interference Graph (RIG):
  - indicates whether there exists a point of time where both variables are live
  - if so, we draw an edge
  - we will then assign different registers to these variables
  - finding assignment of variables to K registers corresponds to coloring graph using K colors!

Graph Coloring Algorithm

Simplify
If there is a node with less than K neighbors, we will always be able to color it!
so we can remove it from the graph
This reduces graph size (it is incomplete)
Every planar can be colored by at most 4 colors (yet can have nodes with 100 neighbors)

Spill
If every node has K or more neighbors, we remove one of them
we mark it as node for potential spilling
then remove it and continue

Select
Assign colors backwards, adding nodes that were removed
If we find a node that was spilled, we check if we are lucky that we can color it
if yes, continue
if no, insert instructions to save and load values from memory
restart with new graph (now we have graph that is easier to color, we killed a variable)