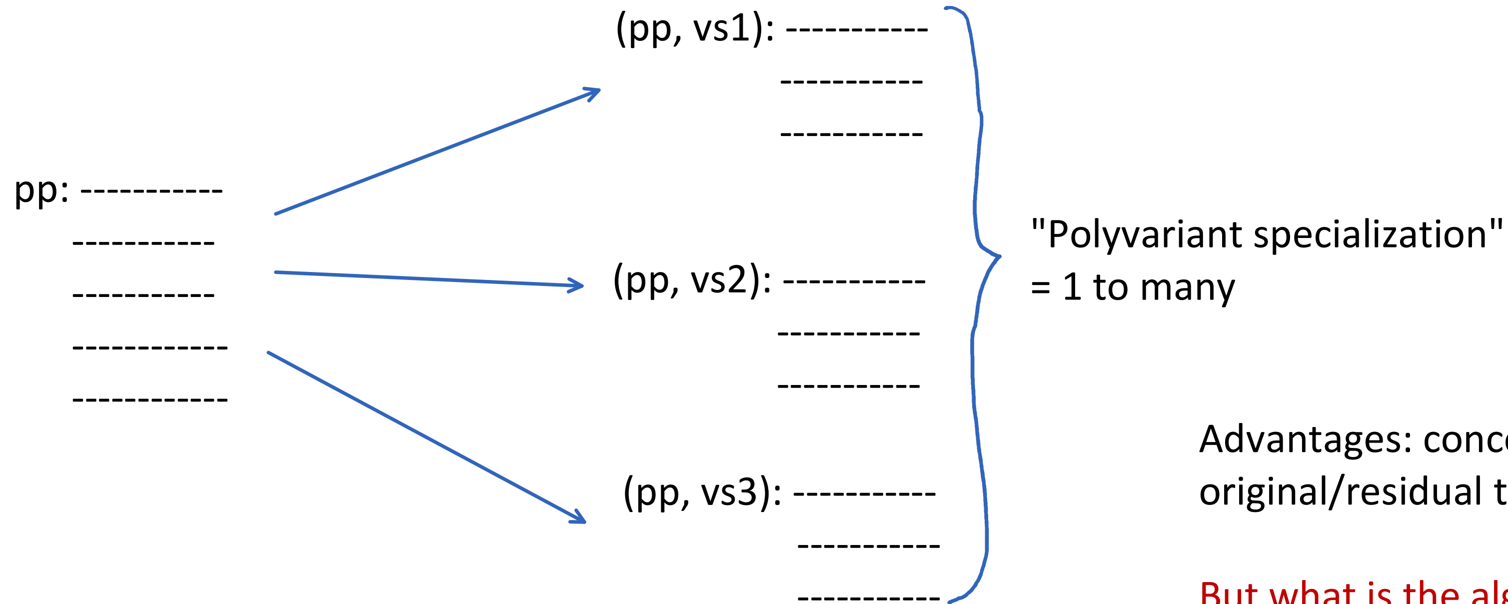
Review

Thursday, March 26, 2020 10:52 PM

state = (pp, (values of supplied vars, values of delayed vars))
$\simeq$ ((pp, values of supplies vars), value of delayed vars)

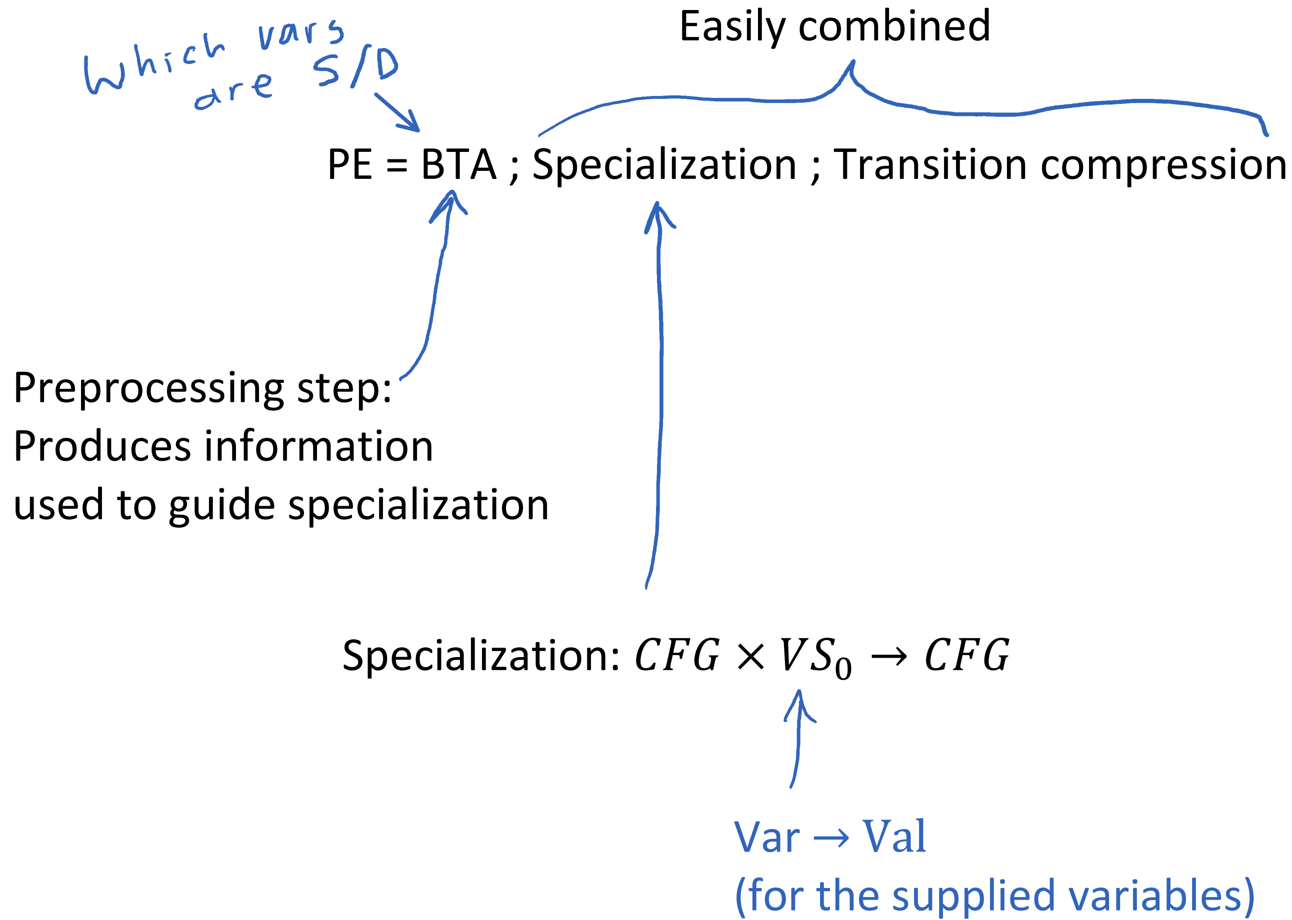


Advantages: concordance of original/residual traces

But what is the algorithm to perform such splitting? (Today's lecture . . .)

Overview of today's lecture

Thursday, March 26, 2020 10:55 PM



Binding-Time Analysis (BTA)

Thursday, March 26, 2020 11:02 PM

Input: Given CFG and an initial division (classification of inputs as S or D)
Output: A uniform, congruent division

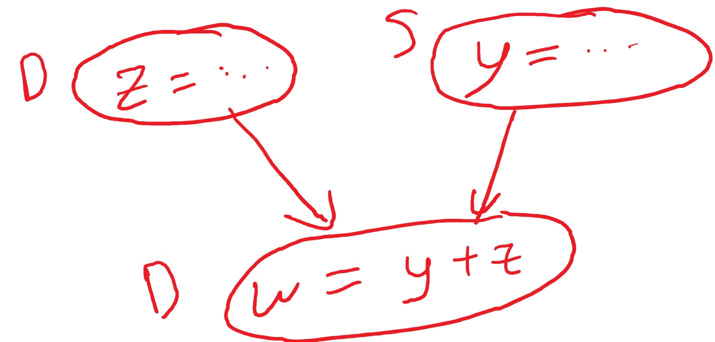
"division" : a classification of the variables as S or D

"uniform" : each variable v has the same classification at every point in the program
non-uniform: v is S at some points; D at others

"congruent" : variables classified S can only depend on variables classified S
i.e., if a variable v depends on a variable w classified D, then v must be D, as well

$y \mapsto S, z \mapsto D$

Examples: D = taint

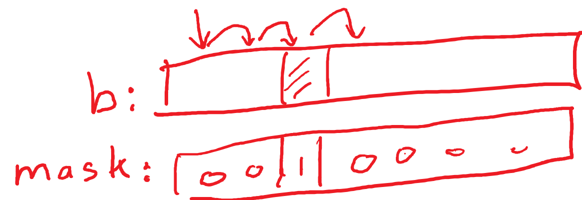


Example: Compute the product of a and b by interpreting the bits of b

Thursday, March 26, 2020 11:10 PM

Source-code form:

```
unsigned int mult(unsigned int a, unsigned int b) {
  unsigned int answer = 0;
  unsigned int mask = 020000000000    // 32-bit
  while (mask > 0) {
    answer = answer << 1;
    if (mask & b) {   // examine a bit of b
      answer = answer + a;
    }
    mask = mask >> 1;
  }
  return answer;
}
```



IR form:

```
          read a
          read b

begin:    answer = 0
          mask = 010
          goto loop

loop:     if mask > 0 goto body else end_loop

body:     answer = answer << 1
          if (mask & b) goto addin else shift

addin:    answer = answer + a
          goto shift

shift:    mask = mask >> 1
          goto loop

end_loop: return answer
```

Finding a congruent division, method 1

Thursday, March 26, 2020 11:08 PM

BTA via dataflow analysis:

a. Find a non-uniform division (per-program-point view)

Propagate division on the CFG

$\langle S/D, S/D, S/D, S/D \rangle$

mash a b answer

Initialization:

$\langle S, S, S, S \rangle$ at all points

except start ?

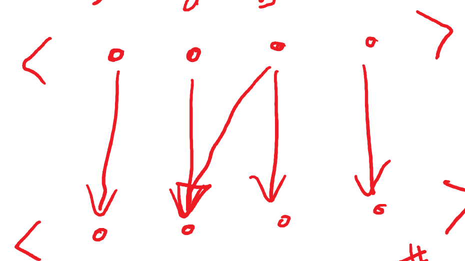
$\langle S, \underline{D}, S, S \rangle$ ... D: initial taint

lub, ⊕, ⊔

D
|
S

Transformers

Assignments:



$a = a + b$

$+^{\#}$	S	D
S	S	D
D	D	D

⊕ (Combine)

$\langle \ldots, S, \ldots \rangle$ $\langle \ldots, D, \ldots \rangle$

$\langle \ldots, D, \ldots \rangle$

$S \sqcup D = D$

Finding a congruent division, method 2

Thursday, March 26, 2020 11:23 PM

Find a uniform division using a dependence graph (per-variable view)

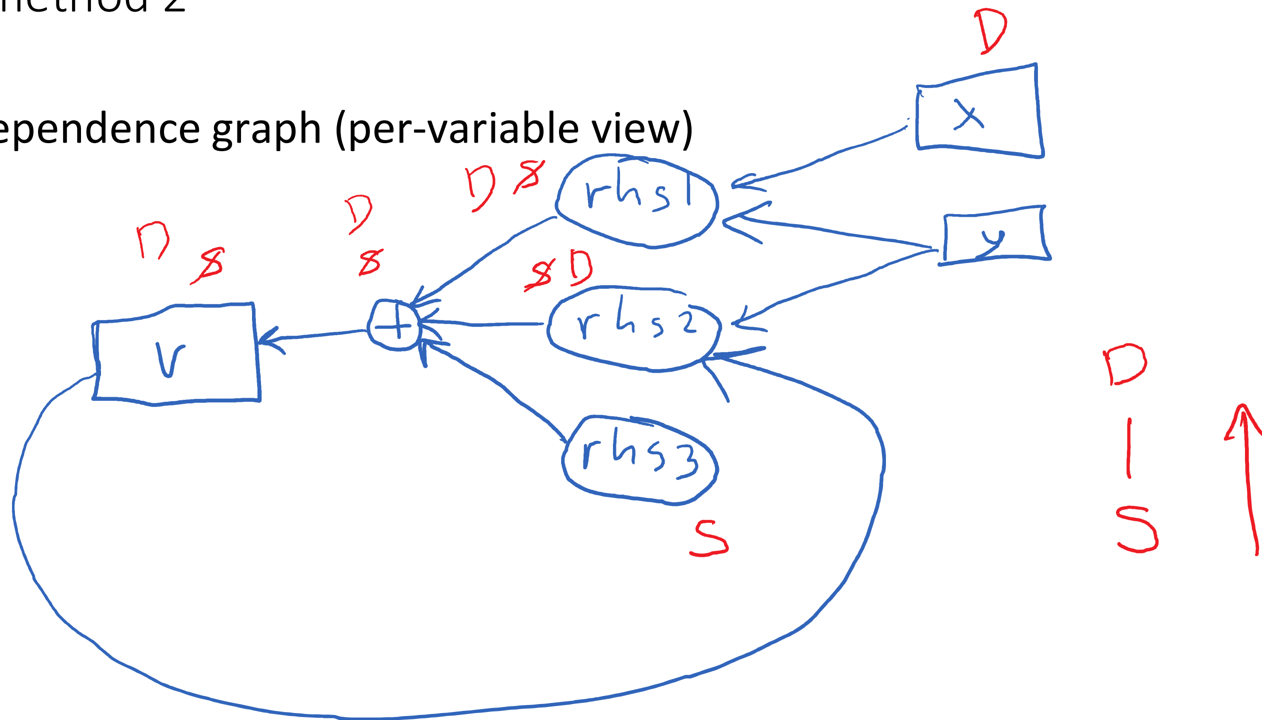
v := rhs1 x+y

⋮

v := rhs2 y*v

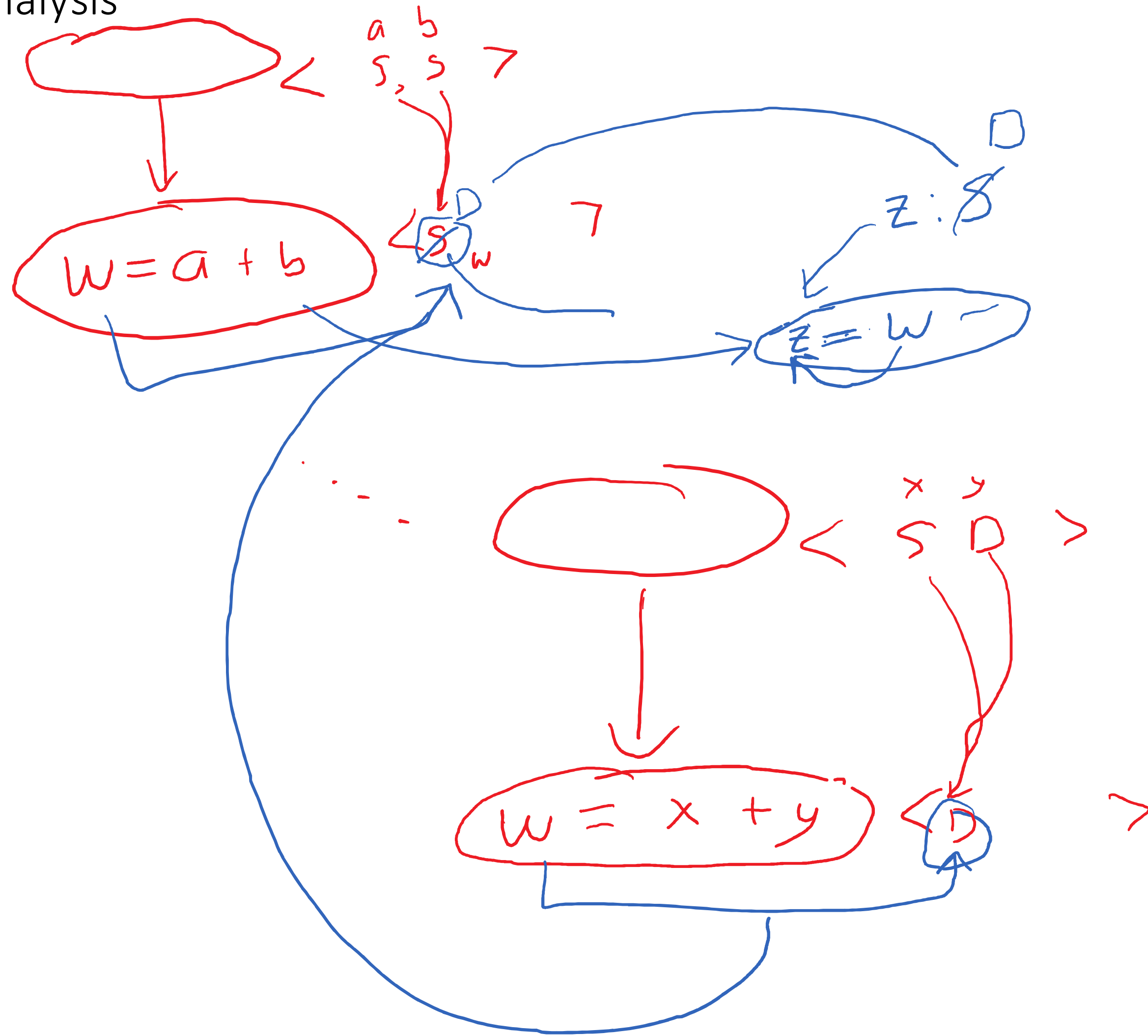
⋮

v := rhs3



Need additional passes of dataflow analysis

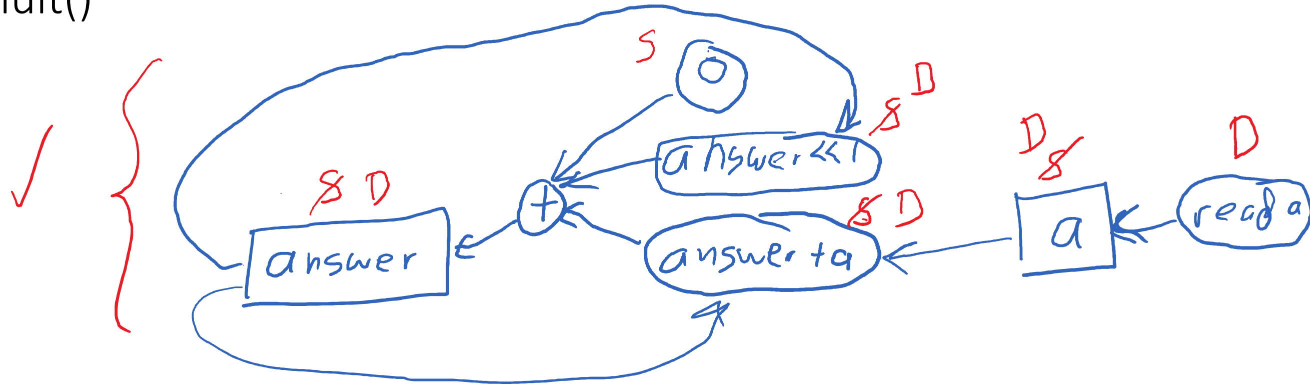
Friday, March 27, 2020 11:22 AM

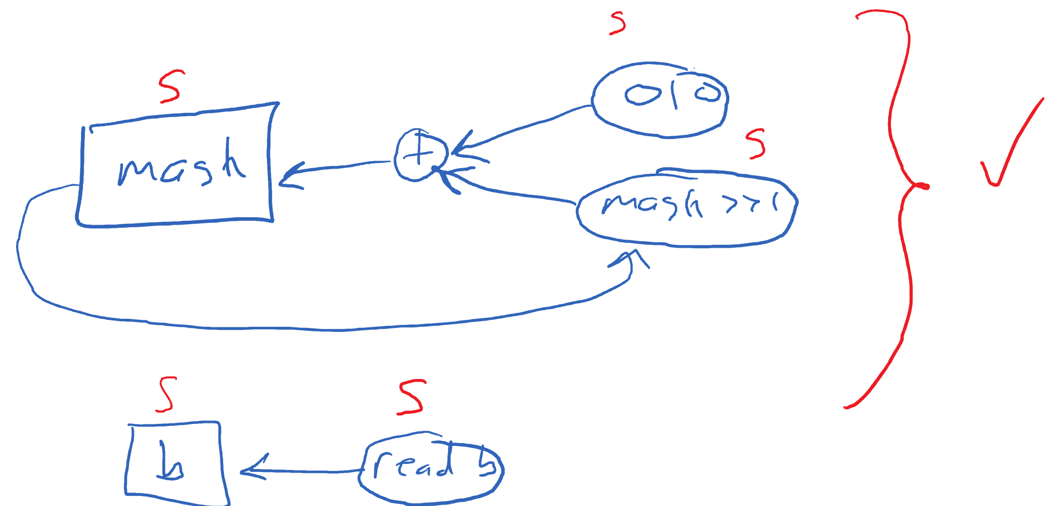


Example: BTA (method 2) for mult()

Friday, March 27, 2020 11:36 AM

```
            read a
            read b

begin:      answer = 0
            mask = 010
            goto loop

loop:       if mask > 0 goto body else end_loop

body:       answer = answer << 1
            if (mask & b) goto addin else shift

addin:      answer = answer + a
            goto shift

shift:      mask = mask >> 1
            goto loop

end_loop: return answer
```





a : D   answer: D   b: s   mask: S

Specialization Algorithm

Thursday, March 26, 2020 11:51 PM

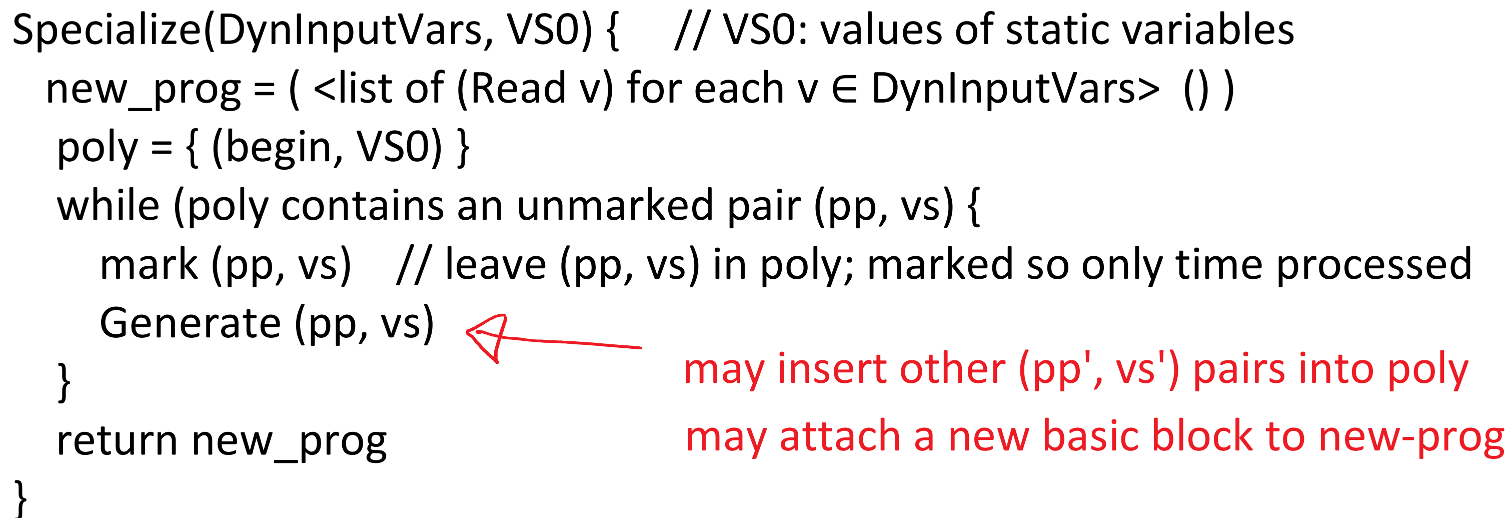
Global variables:

prog: CFG (in IR form) of the original program

new_prog: CFG (in IR form) of the residual program

poly: set of pairs of the form (program-point, StaticVars → Val)

[each pair can be marked or unmarked]



```
Specialize(DynInputVars, VS0) {     // VS0: values of static variables
  new_prog = ( <list of (Read v) for each v ∈ DynInputVars>  () )
   poly = { (begin, VS0) }
   while (poly contains an unmarked pair (pp, vs) {
      mark (pp, vs)    // leave (pp, vs) in poly; marked so only time processed
      Generate (pp, vs)
   }
   return new_prog
}
```

Generate(pp, vs)

Friday, March 27, 2020   12:12 AM

Generate(pp, vs) {
  new_block = empty block; pp_init = pp; vs_init = vs
  for (command = Lookup(pp, prog); command != null; command = Next(command) {

Command type		Perform action	Append to new_block	Insert into poly
x := exp	x: D	residual_exp = simplify(exp,vs)	"x := " << residual_exp	---
	x: S	vs = vs[x↦eval(exp,vs)]	---	---
return exp	---	residual_exp = simplify(exp,vs)	"return " << residual_exp	---
goto pp'	---	---	"goto (pp'," << vs << ")"	(pp', vs)
if exp goto pp' else pp''	exp: D	residual_exp = simplify(exp,vs)	"if " << residual_exp "goto (pp'," << vs << ")" "else (pp''," << vs << ")"	(pp', vs) (pp'', vs)
	exp:S & eval(exp,vs) = T	---	"goto (pp'," << vs << ")"	(pp', vs)
	exp:S & eval(exp,vs) = F	---	"goto (pp''," << vs << ")"	(pp'', vs)

  }
  Insert new_block into new_prog, with tag (pp_init, vs_init)
}

Example: mult with mask: S, b: S, a: D, answer: D and b = 9

Friday, March 27, 2020 12:41 AM

Residual program:

mask → b →

```
                read a
(begin, ?, 011): answer = 0
                goto (loop, 010, 011)

(loop, 010, 011): goto (body, 010,011)
(body, 010,011): answer = answer << 1
                 goto (addin,010,011)
(addin,010,011): answer = answer + a
                 goto (shift,010,011)
(shift,010,011):  goto (loop,04,011)

(loop,04,011):    goto (body, 04,011)
(body, 04,011):   answer = answer << 1
                  goto (shift,04,011)
(shift,04,011):   goto (loop,02,011)

(loop,02,011):    ...
...
(end_loop,0,011): return answer
```

9 is 011 (octal) and 1001 (binary)

VS0 = (?, 011)

```
          read a
       read b
begin:    answer = 0
          mask = 010
          goto loop

loop:     if mask > 0 goto body else end_loop

body:     answer = answer << 1
          if (mask & b) goto addin else shift

addin:    answer = answer + a
          goto shift

shift:    mask = mask >> 1
          goto loop

end_loop: return answer
```

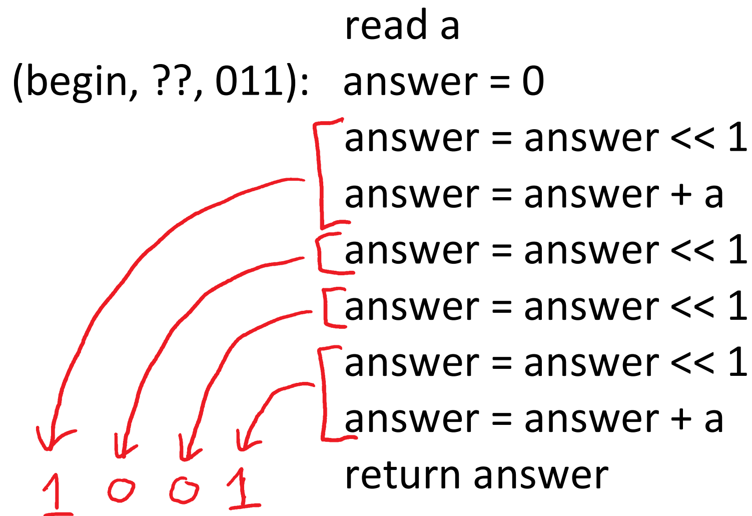
poly = { (begin, (?, 011)), (loop, 010, 011) }
(body, 010, 011)

mask : S
b : S
a : D
answer : D

VS = (010, 011)
(04, 011)

Compress Transitions

Friday, March 27, 2020 12:55 AM



Could simplify further by building "the dag for this basic block" (a standard compiler technique)
Emit code:

```
return ((((a << 1) << 1) << 1) +a)
```

Compress transitions on-the-fly

Friday, March 27, 2020 12:12 AM

Generate(pp, vs) {

new_block = empty block; pp_init = pp; vs_init = vs

for (command = Lookup(pp, prog); command != null; command = Next(command) {

Command type		Perform action	Append to new_block	Insert into poly
x := exp	x: D	residual_exp = simplify(exp,vs)	"x := " << residual_exp	---
	x: S	vs = vs[x↦eval(exp,vs)]	---	---
return exp	---	residual_exp = simplify(exp,vs)	"return " << residual_exp	---
goto pp'	---	command = Lookup(pp',prog)	---	---
if exp goto pp' else pp''	exp: D	residual_exp = simplify(exp,vs)	"if " << residual_exp "goto (pp'," << vs << ")" "else (pp''," << vs << ")"	(pp', vs) (pp'', vs)
	exp:S & eval(exp,vs) = T	command = Lookup(pp',prog)	---	---
	exp:S & eval(exp,vs) = F	command = Lookup(pp'',prog)	---	---

}

Insert new_block into new_prog, with tag (pp_init, vs_init)

}