state = (pp, (values of supplied vars, values of delayed vars))
≈ ((pp, values of supplies vars), value of delayed vars)

"Polyvariant specialization"
= 1 to many

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

PE = BTA; Specialization; Transition compression

Easily combined

Preprocessing step:
Produces information used to guide specialization

Specialization: $CFG \times VS_0 \rightarrow CFG$

Var $\rightarrow$ Val
(for the supplied variables)
**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: \[ D = \text{tan}t \]
Example: Compute the product of $a$ and $b$ by interpreting the bits of $b$

Source-code form:

```c
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 $a$
- $b = 1001$

```
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

BTA via dataflow analysis:

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

Propagate division on the CFG

\[ <s/d, s/d, s/d, s/d> \]

Initialization:
\[ <s, s, s, s> \] at all points except start

\[ <s, d, s, s> \] lbd, \oplus, l

\[ <..., s, ...> <..., d, ...> \]

\[ l/b, \oplus, u \]

\[ s, u, d \]

\[ D: initial taint \]

\[ o = a + b \]

\[ \begin{array}{c}
    \text{Assignments:} \\
    \begin{array}{c}
        a \\
        b \\
        o
    \end{array}
    \\
    \begin{array}{c}
        \# \\
        \text{SD} \\
        s \\
        s \\
        D \\
        D \\
        D
    \end{array}
\end{array} \]

\[ \Theta (\text{Combine}) \]

\[ <..., D, ...> \]

\[ s\uparrow o = D \]
Find a uniform division using a dependence graph (per-variable view)

\[ v := \text{rhs1} \quad x + y \]

\[ v := \text{rhs2} \quad y \times v \]

\[ v := \text{rhs3} \]

Diagram:

- Variables: \( x \) and \( y \)
- Expressions: \( x + y \), \( y \times v \), \( \text{rhs1} \), \( \text{rhs2} \), \( \text{rhs3} \)
- Operations: Addition, Multiplication
- Dependences:
  - \( \text{rhs1} \) depends on \( x \)
  - \( \text{rhs2} \) depends on \( y \)
  - \( \text{rhs3} \) depends on \( v \)
Need additional passes of dataflow analysis
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

Example: BTA (method 2) for mult()
Specialization Algorithm

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) {
    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      | exp: D residual_exp = simplify(exp,vs) | "if " << residual_exp |
        | goto pp'    | exp:S & eval(exp,vs) = T | "goto (pp'," << vs << ")" |
        | else pp''   | exp:S & eval(exp,vs) = F | "else (pp'", << vs << ")" |
        |             | --- | "goto (pp''," << vs << ")" | (pp', vs) |
        |             | --- | "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

9 is 011 (octal) and 1001 (binary)
VS0 = (?, 011)

Residual program:

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

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

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

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

mask: S
b: S
a: D
answer: D
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)) {

<table>
<thead>
<tr>
<th>Command type</th>
<th>Perform action</th>
<th>Append to new_block</th>
<th>Insert into poly</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := exp</td>
<td>x: D residual_exp = simplify(exp,vs) vs = vs[x↦eval(exp,vs)]</td>
<td>&quot;x := &quot; &lt;&lt; residual_exp</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>x: S</td>
<td></td>
<td>---</td>
</tr>
<tr>
<td>return exp</td>
<td>--- residual_exp = simplify(exp,vs)</td>
<td>&quot;return &quot; &lt;&lt; residual_exp</td>
<td>---</td>
</tr>
<tr>
<td>goto pp'</td>
<td>--- command = Lookup(pp',prog)</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>if exp goto pp' else pp&quot;</td>
<td>exp: D residual_exp = simplify(exp,vs)</td>
<td>&quot;if &quot; &lt;&lt; residual_exp &quot;goto (pp&quot;,&quot; &lt;&lt; vs &lt;&lt; &quot;)&quot; &quot;else (pp&quot;&quot;,&quot; &lt;&lt; vs &lt;&lt; &quot;)&quot;</td>
<td>(pp', vs) (pp&quot;, vs)</td>
</tr>
<tr>
<td></td>
<td>exp:S &amp; eval(exp,vs) = T command = Lookup(pp',prog)</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>exp:S &amp; eval(exp,vs) = F command = Lookup(pp&quot;,prog)</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

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

}