Removal of interpretation overhead
in-line substitution of functions called by int
elimination of cases not needed to interpret q
stores of q -- part of the information is known
e.g., binding list
Goal: Ensure that there is a concordance between the residual-program states and the subject-program states

Subject-program states

(\text{label}, (\text{vs}, \text{vd}))

values of supplied variables

Residual-program states

((\text{label}, \text{vs}), \text{vd})

values of delayed variables

"compound label" (conceptually)

Concordance:

Subject-program trace

Residual-program trace

The trace visits label \text{l1} three times

The trace visits label \text{l1} twice

pc == \text{l1} + the value of \text{vs} ([3,99,17] or [6,97,13])

pc == (\text{l1}, [3,99,17])

pc == (\text{l1}, [6,97,13])

Theme: "Conversion of data to control"
("control" = position in the code)
What does a specialized program look like?

Subject program $P$

Residual program $p_s = \left[ \left[ pe \right] \right][p, s]$
1. "surface syntax" of the language of subject programs

   L = A simple imperative flow-chart language with int and list data

2. "deep syntax" of the language of subject programs

   s-expression representation of a program's control-flow graph (CFG)
   + algebraic data type to represent expressions

   Algebraic datatype for representing L expressions
   exp ::=
   | ConstExpr(constant)
   | IdentExpr(identifier)
   | Compound(op exp exp)

   constant ::= <integer constants>
   identifier ::= [a-zA-Z]+
   operator ::= + | * | cons | hd | tl | ...

3. meta-language in which to describe the partial-evaluation algorithm

   o pidgin Algol
   o tables of cases
   o informal graph diagrams
   o <hand-waving> + <smoke & mirrors>

   meta-language permits deconstructing expressions via pattern matching:

   cases e of
   ConstExpr(c): ... expression involving e, c, ...
   IdentExpr(i): ... expression involving e, i, ...
   Compound(o, a, b): ... expression involving e, o, a, b ....

   end

Constructs of L

assignment
if cond goto label else label'
go to label
read of initial data
return final value
print

Also, as syntactic sugar
begin ... end
while ... do ... od
repeat ... until ...

Data types Operators
integers         plus, <, >, =, ...
s-expressions    hd, tl, cons, nil, isnil
simplify(e, store) = // store is a map from names to values
cases e of
    ConstExpr(c): e
    IdentExpr(i): DefinedIn(i,store) ? ConstExpr(Lookup(i,store)) : e
    Compound(op, a, b):
        let v1 = simplify(a, store) and v2 = simplify(b, store) in
        cases v1 of
            ConstExpr(c1):
                cases v2 of
                    ConstExpr(c2): ConstExpr(funcof(op)(c1,c2))
                    default: Compound(op,v1,v2)
            default: Compound(op,v1,v2)
Example of an L-program (surface and deep syntax)

Surface syntax:

```
read(N)
begin: i := 1
    sum := 0
goto loop
loop: if i > N goto end else body
body: sum := sum + i
    i := i + 1
goto loop
end: return sum
```

Deep syntax:

```
( ( (Read N) )
  ( (Block begin
      (Assign i ConstExpr(1))
      (Assign sum ConstExpr(0))
      (goto loop)
    )
  (Block loop
      (Cond (Less i N)
        end
tbody
      )
    )
  (Block body
      (Assign sum Compound(+, IdentExpr(sum), IdentExpr(i)))
      (Assign i Compound(+, IdentExpr(i), ConstExpr(1)))
    )
  (Block end
    (Return sum)
  )
)
```

(Singleton) list of read statements

List of blocks
For partial evaluation, need >= 2 read statements

begin: goto q

q: if y < 3 goto r else s

r: y := y+1
z := z+1
 goto q

s return z

A trace:

Suppose that the input is y: 1, z: c, where c is some specific value

(begin, (1,c))
(q, (1, c))
(r, (1, c))
(q, (2, c+1))
(r, (2, c+1))
(q, (3, c+2))
(s, (3, c+2))
state = (pp, (values of supplied vars, values of delayed vars))
≈ ((pp, values of supplies vars), value of delayed vars)
Example: Specialize program w.r.t. $y \mapsto 1$

begin: goto q
q: if $y < 3$ goto r else s
r: $y := y+1$
   $z := z+1$
   goto q
s: return $z$

Current partial state: $y \mapsto 3$
Worklist: { ... }

(q,5): ...
(s,5): ...
(r,5): ...
(q,6): ...
(r,6): ...
.
.

read(y)
read(z)

read(z)
(begin, 1): goto (q,1)
(q,1): goto (r,1)
(r,1): $z := z+1$
goto (q,2)

(q,2): goto (r,2)
(r,2): $z := z+1$

(q,3): goto (s,3)
(s,3): return $z$

Trace (w.r.t. $z \mapsto c$)

Trace of the original program

Current state: $z \mapsto c+2$

((begin,1), c)
((q,1), c)
((r,1), c)
((q,2), c+1)
((r,2), c+1)
((q,3), c+2)
((s,3), c+2)
((begin,1), c)
((q,1), c)
((r,1), c)
((q,2), c+1)
((r,2), c+1)
((q,3), c+2)
((s,3), c+2)
Lots of gotos to gotos
Most correspond to actions in the original program on supplied quantities "swallowed by the partial evaluator" (i.e., performed at PE-time -- in particular, "y := y+1")

Compress the goto transitions

read(z)
(r,1):  z := z+1
(r,2):  z := z+1
(s,3):  return z

Trace (w.r.t. z↦c)

((r,1), c)
((r,2), c+1)
((s,3), c+2)

Not as easy to make the correspondence with the trace of the original program
1: Binding-time analysis (BTA)
2: Specialization

BTA:

division: labeling of variables/statements into S and D

uniform: each variable has the same S/D classification at all
program points (think: "type")

read(z)     // D

...

z := 27      // S, but makes the division non-uniform

congruence: Variables classified S, can only depend on variables
classified S.
- congruence analysis ~ taint analysis
- D leads to D

y := y+1     // S <- S
z := z+1      // D <- D

w := y+z     // D <- S+D