Example: +, 0, - abstract domain

\[
\begin{align*}
\text{l}_0 &: (T, T, T, T) \\
x &= 1; \\
y &= -1; \\
i &= 0; \\
\text{while (i < 2) } & \{ \\
x &= x + 1; \\
y &= y - 1; \\
z &= x*y; \\
i &= i + 1; \\
\} \\
\text{l}_1 &: (T, T, T, T) \\
i &> 0 \\
\text{l}_2 &: (T, T, T, T) \\
\text{l}_3 &: (T, T, T, T)
\end{align*}
\]
Interprocedural Analysis

Computer Science and Artificial Intelligence Laboratory
MIT

Nov 15, 2011
The easy way 1

```
int x;
main()
{
  x=2;
  f(x);x=2;
  x=3;
  f(x);x=2;
}
```

Pros:
- Very precise

Cons:
- Prohibitively expensive
  - With recursive calls you have to inline infinitely
  - Even without them inlining is exponential

Inline everything!
The easy way 2

Pros:
- Easy to implement
- Efficient

Cons:
- Too imprecise
- Introduces many fake paths into the program

A call is just another form of control flow
A better approach

“Precise interprocedural dataflow analysis via graph reachability”, by Reps, Horwitz and Sagiv

Key ideas:
- valid paths require matching call-return edges
- If we can filter out invalid paths in “easy way 2”, we can regain the precision
- If we can summarize the effect of a procedure, we can do all of this efficiently
Running example

program main
begin
  declare x, y: integer
  read(x)
  call P(x, y)
end

procedure P(a, b: integer)
begin
  if (a > 0) then
    read(b)
    a := a - b
    call P(a, b)
    print(a, b)
  fi
end
Running example

Give call nodes unique ids
Label call and return edges as \( (id, \text{and}, id) \) Respectively

Valid paths can be defined with a grammar

**Same level valid path:**

\[
\text{matched} \rightarrow (i \ \text{matched} \ )_i \ \text{matched} \\
| \ \varepsilon
\]

**Valid path:**

\[
\text{valid} \rightarrow \text{valid} (i \ \text{matched} \ | \ \text{matched}
\]
Consider a path \( p = n_0, n_1, \ldots, n_k, n \) to a node \( n \)
- (note that for all \( i \) \( n_i \in \text{pred}(n_{i+1}) \))

The solution must take this path into account:
\[
f_p (in_0) = (f_{nk}(f_{nk-1}(\ldots f_{n1}(f_{n0}(in_0)) \ldots)) \leq in_n
\]

So the solution must have the property that
\[
\forall \{f_p (in_0) : p \text{ is a path to } n\} \leq in_n
\]

and ideally
\[
\forall \{f_p (in_0) : p \text{ is a path to } n\} = in_n
\]

This is the Meet over Paths Solution (MOP)
A distributive transfer function satisfies
\[ f(x \lor y) = f(x) \lor f(y) \]

Distributivity preserves precision

If framework is distributive, then worklist dataflow analysis algorithm produces the MOP solution
- For all \( n \):
  \[ \lor\{f_p (\text{in}_0) . \ p \text{ is a path to } n\} = \text{in}_n \]
Meet over all *valid* paths solution

Let IVP(n1, n2) be the set of valid paths from n1 to n2

The meet over all *valid* paths solution (MVP) satisfies

\[ \lor \{ f_p (\text{in}_0) \cdot p \in \text{IVP}(n0, n) \} = \text{in}_n \]

How do we compute this?
Cool idea

Distributive transfer functions over a bit-vector lattice can be represented as a graph
Example

Reaching Definitions

1: \( x = a + b \)  
2: \( y = c + d \)  
3: \( b = a + c \)

4: \( x = a + c \)

In general:
  - Add arrow from \( \emptyset \) to the gen set
  - Remove arrows to the kill set
  - Straight arrows for everything else
Key observations

Graphs are composable
- We can summarize the effect of an entire procedure in a single transfer function

Dataflow analysis becomes a graph reachability problem
Dataflow problem as graph reachability