Methodologies for Software Processes

Lecture 4- Dataflow Analysis

(our slides are taken from other courses that use "Principles of Program Analysis" as textbook)

Dataflow analysis:
A General Framework

Dataflow Analysis

- Compile-time reasoning about run-time values of variables or expressions
- At different program points
  - Which assignment statements produced value of variable at this point?
  - Which variables contain values that are no longer used after this program point?
  - What is the range of possible values of variable at this program point?

Program Representation

- Control Flow Graph
  - Nodes N – statements of program
  - Edges E – flow of control
    - pred(n) = set of all predecessors of n
    - succ(n) = set of all successors of n
  - Start node n_0
  - Set of final nodes N_{final}

Program Points

- One program point before each node
- One program point after each node
- Join point – point with multiple predecessors
- Split point – point with multiple successors

Basic Idea

- Information about program represented using values from algebraic structure
- Analysis produces a value for each program point
- Two flavors of analysis
  - Forward dataflow analysis
  - Backward dataflow analysis

Forward Dataflow Analysis

- Analysis propagates values forward through control flow graph with flow of control
  - Each node n has a transfer function f_n
    - Input – value at program point before node
    - Output – new value at program point after node
  - Values flow from program points after predecessor nodes to program points before successor nodes
  - At join points, values are combined using a merge function
- Canonical Example: Reaching Definitions

Backward Dataflow Analysis

- Analysis propagates values backward through control flow graph against flow of control
  - Each node n has a transfer function f_n
    - Input – value at program point after node
    - Output – new value at program point before node
  - Values flow from program points before successor nodes to program points after predecessor nodes
  - At split points, values are combined using a merge function

– Canonical Example: Live Variables

Representing the property of interest

- Dataflow information will be lattice values
  - Transfer functions operate on lattice values
  - Solution algorithm will generate increasing sequence of values at each program point
  - Ascending chain condition will ensure termination
- Will use $\vee$ to combine values at control-flow join points

Transfer Functions

- Transfer function f_n: $P \rightarrow P$ for each node n in control flow graph
- f_n models effect of the node on the program information

Transfer Functions

Each dataflow analysis problem has a set F of transfer functions $f: P \to P$

- Identity function $i \in F$
- F must be closed under composition:

  $\forall f,g \in F$. the function $h(x) = f(g(x)) \in F$
- Each $f \in F$ must be monotone: $x \leq y$ implies $f(x) \leq f(y)$

- Sometimes all $f \in F$ are distributive: $f(x \vee y) = f(x) \vee f(y)$
- Distributivity implies monotonicity

Forward Dataflow Analysis

- Simulates execution of program forward with flow of control
- For each node n, have
  - in_n – value at program point before n
  - out_n – value at program point after n
  - f_n – transfer function for n (given in_n, computes out_n)
- Require that solution satisfy
  - $\forall n.\ out_n = f_n(in_n)$
  - $\forall n \neq n_0.\ in_n = \vee \{ out_m \ .\ m \text{ in } pred(n) \}$
  - $in_{n0} = \mathrm{I}$,

  where I summarizes information at start of program

Worklist Algorithm for Solving Forward Dataflow Equations

for each n do $out_n := f_n(\perp)$

$in_{n0} := I$; $out_{n0} := f_{n0}(I)$

worklist := N - { n_0 }

while worklist $\neq \emptyset$ do

  remove a node n from worklist

  $in_n := \vee \{ out_m \,.\, m \text{ in } pred(n) \}$

  $out_n := f_n(in_n)$

  if out_n changed then

    worklist := worklist $\cup$ succ(n)

Correctness Argument

- Why result satisfies dataflow equations?
- Whenever process a node n,
  the algorithm ensures that $out_n = f_n(in_n)$
- Whenever out_m changes, the algorithm puts succ(m) on worklist.
  Consider any node $n \in$ succ(m).
  It will eventually come off worklist and the algorithm will set

  $$in_n := \vee \{ out_m \,.\, m \text{ in } pred(n) \}$$

  to ensure that $in_n = \vee \{ out_m \,.\, m \text{ in } pred(n) \}$
- So final solution will satisfy dataflow equations

Termination Argument

- Why does algorithm terminate?
- Sequence of values taken on by in_n or out_n is a chain. If values stop increasing, worklist empties and algorithm terminates.
- If the lattice enjoys the ascending chain property, the algorithm terminates
  - Algorithm terminates for finite lattices
  - For lattices without ascending chain property, we may use widening operator

Widening Operators

- Detect lattice values that may be part of infinitely ascending chain
- Artificially raise value to least upper bound of chain
- Example:
  - Lattice is set of all subsets of integers
  - Could be used to collect possible values taken on by variable during execution of program
  - Widening operator might raise all sets of size n or greater to TOP (likely to be useful for loops)

Reaching Definitions

- P = powerset of set of all definitions in program (all subsets of set of definitions in program)
- $\vee = \cup$ (order is $\subseteq$)
- $\perp = \emptyset$
- $I = in_{n0} = \perp$
- F = all functions f of the form $f(x) = a \cup (x-b)$
  - b is set of definitions that node kills
  - a is set of definitions that node generates
- General pattern for many transfer functions
  - $f(x) = GEN \cup (x\text{-}KILL)$

Does Reaching Definitions Satisfy the Framework Constraints?

- $\subseteq$ satisfies conditions for $\leq$
  - $x \subseteq y$ and $y \subseteq z$ implies $x \subseteq z$ (transitivity)
  - $x \subseteq y$ and $y \subseteq x$ implies $y = x$ (asymmetry)
  - $x \subseteq x$ (idempotence)

- F satisfies transfer function conditions
  - $\lambda x.\emptyset \cup (x - \emptyset) = \lambda x.x \in F$ (identity)
  - Will show $f(x \cup y) = f(x) \cup f(y)$ (distributivity)

$$
\begin{aligned}
f(x) \cup f(y) &= (a \cup (x - b)) \cup (a \cup (y - b)) \\
&= a \cup (x - b) \cup (y - b) = a \cup ((x \cup y) - b) \\
&= f(x \cup y)
\end{aligned}
$$

Does Reaching Definitions Framework Satisfy Properties?

- What about composition?

  Given $f_1(x) = a_1 \cup (x\text{-}b_1)$ and $f_2(x) = a_2 \cup (x\text{-}b_2)$

  Must show $f_1(f_2(x))$ can be expressed as $a \cup (x - b)$

$$
\begin{aligned}
f_1(f_2(x)) &= a_1 \cup ((a_2 \cup (x\text{-}b_2)) - b_1) \\
&= a_1 \cup ((a_2 - b_1) \cup ((x\text{-}b_2) - b_1)) \\
&= (a_1 \cup (a_2 - b_1)) \cup ((x\text{-}b_2) - b_1)) \\
&= (a_1 \cup (a_2 - b_1)) \cup (x\text{-}(b_2 \cup b_1))
\end{aligned}
$$

  Let $a = (a_1 \cup (a_2 - b_1))$ and $b = b_2 \cup b_1$

  Then $f_1(f_2(x)) = a \cup (x – b)$

General Result

All GEN/KILL transfer function frameworks satisfy

- Identity
- Distributivity
- Composition

properties

Available Expressions

- P = powerset of set of all expressions in program (all subsets of set of expressions)
- $\vee = \cap$ (order is $\supseteq$)
- $\perp$ = P
- I = $in_{n0} = \emptyset$
- F = all functions f of the form f(x) = a $\cup$ (x-b)
  - b is set of expressions that node kills
  - a is set of expressions that node generates
- Another GEN/KILL analysis

Concept of Conservatism

- Reaching definitions use $\cup$ as join
  - Optimizations must take into account all definitions that reach along ANY path
- Available expressions use $\cap$ as join
  - Optimization requires expression to reach along ALL paths
- Optimizations must conservatively take all possible executions into account. Structure of analysis varies according to way analysis is used.

Backward Dataflow Analysis

- Simulates execution of program backward against the flow of control
- For each node n, have

  in_n – value at program point before n

  out_n – value at program point after n

  f_n – transfer function for n (given out_n, computes in_n)
- Require that solution satisfies

  $\forall n.\ in_n = f_n(out_n)$

  $\forall n \notin N_{final}.\ out_n = \vee \{ in_m\ .\ m \text{ in } succ(n) \}$

  $\forall n \in N_{final} = out_n = O$

  Where O summarizes information at end of program

Worklist Algorithm for Solving Backward Dataflow Equations

for each n do $in_n := f_n(\perp)$

for each $n \in N_{final}$ do $out_n := O$; $in_n := f_n(O)$

worklist := $N - N_{final}$

while worklist $\neq \emptyset$ do

    remove a node n from worklist

    $out_n := \vee \{ in_m \;.\; m \text{ in } succ(n) \}$

    $in_n := f_n(out_n)$

    if in_n changed then

        worklist := worklist $\cup$ pred(n)

Live Variables

- P = powerset of set of all variables in program (all subsets of set of variables in program)
- $\vee = \cup$ (order is $\subseteq$)
- $\perp = \emptyset$
- $\mathrm{O} = \emptyset$
- F = all functions f of the form f(x) = a $\cup$ (x-b)
  - b is set of variables that node kills
  - a is set of variables that node reads

DFA of non-distributive properties

The general pattern of Dataflow Analysis

$$GA_{\mathbf{i}}(p) = \begin{cases} \iota & \text{if } p \in E \\ \oplus \{ GA_o(q) \mid q \in F \} & \text{otherwise} \end{cases}$$

$$GA_o(p) = f_p (GA_{\mathbf{i}}(p))$$

where :

E is the set of initial/final points of the control-flow diagram

ι specifies the initial values

F is the set of successor/predecessor points

$\oplus$ is the combination operator

f is the transfer function associated to node p

Distributive properties

- Monotonicity of a function implies that
-

$$f(x \cup y) \supseteq f(x) \cup f(y)$$

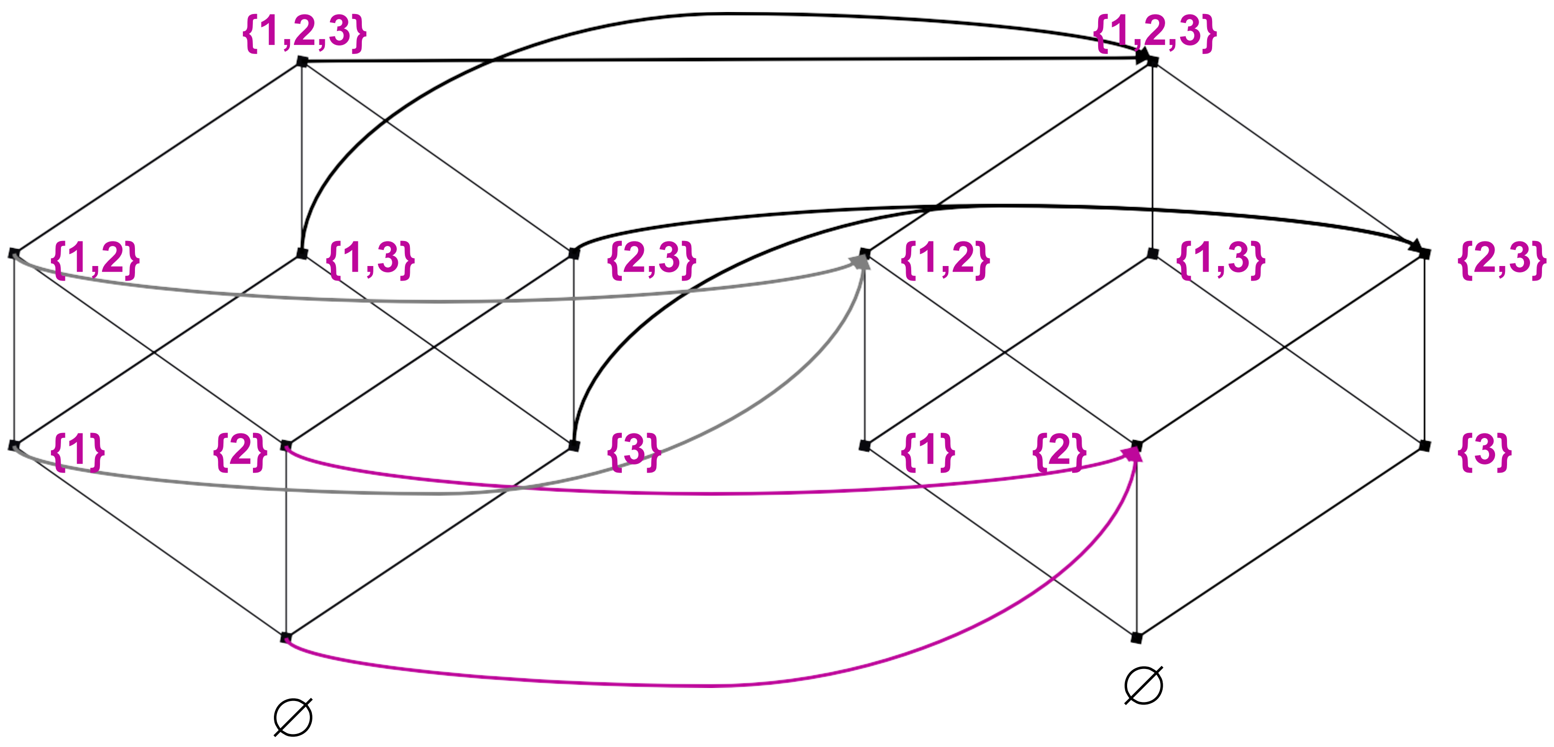
- A function is said distributive a stronger condition hold:

$$f(x \cup y) = f(x) \cup f(y)$$

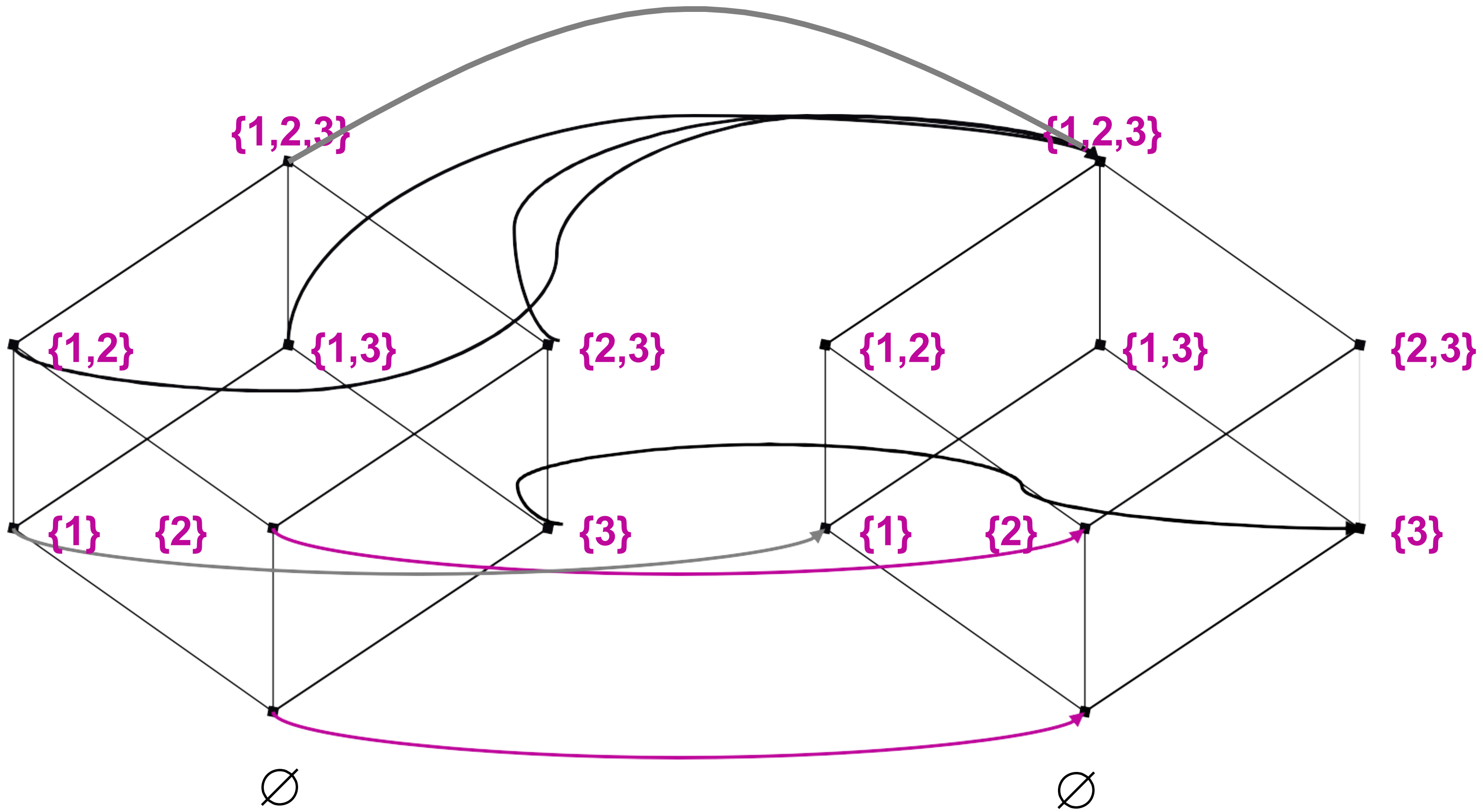
- In general, a dataflow analysis is said distributive if the transfer functions satisfy

$$f(lub(x,y)) = lub(f(x), f(y))$$

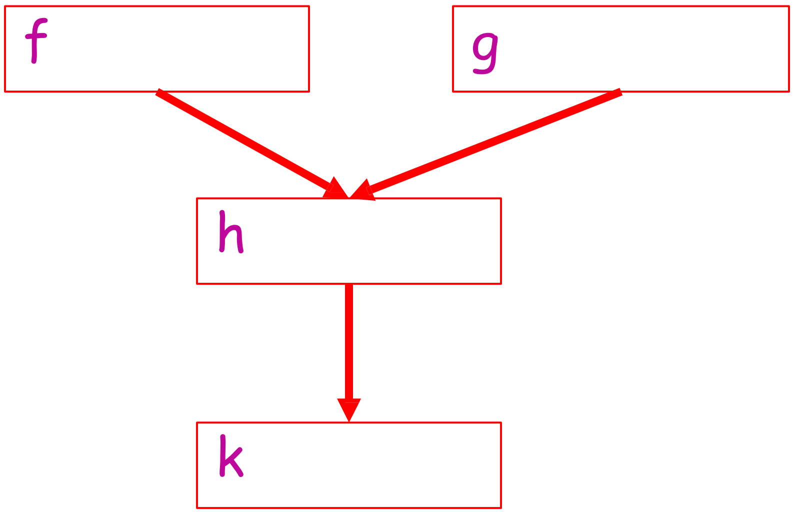
Example: f distributive



Example: f not distributive



Why distributivity is important



$k(h(f(0) \cup g(0))) =$

$k(h(f(0)) \cup h(g(0))) =$

$k(h(f(0))) \cup k(h(g(0)))$

The overall analysis is equal to the lub of the analyses on the different pathes.

DFA of a distributive property

- If the property is distributive, then the minimal solution of the equation system is equivalent to combining the result of the analyses along all the pathes (including infinite pathes).
- In this case the combination operator (least upper bound) does not introduces further loss of accuracy

Which properties are distributive?

- The distributive properties are usually “easy”
- They mainly concern the structure of the program (not the actual values assigned to the variables)
  - E.g., live variables, available expressions, reaching definitions, very busy expressions
  - These properties concern HOW the program pursues the computation, not the actual values of the variables

Non-distributive properties

- They deal with WHAT a program computes
  - E.g.: has the output always the same constant value? Is a variable always assigned a positive number?

- Example: Constant Propagation Analysis

For each program point, we want to know if a variable is always assigned to exactly the same constant value.

It is a forward and definite property.

Constant Propagation Analysis

- Consider the set: $(\mathrm{Var} \to Z^{\top})_{\perp}$

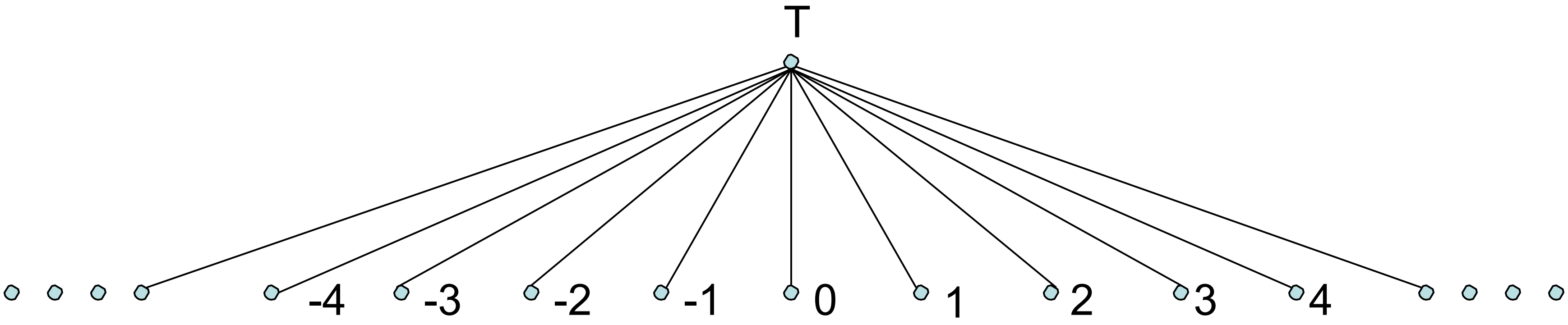
– Var is the set of variables occurring in the program

– $Z^{\top} = Z \cup \{\top\}$ partially ordered by:

for all $n \in Z$: $n \leq_{CP} \top$

for all $n_1, n_2 \in Z$: $(n_1 \leq_{CP} n_2) \Leftrightarrow (n_1 = n_2)$

$Z^{\top}$



- L= $Z \cup \{T\}$
- forall $n \in Z$: $n \leq T$

The lattice (Var $\rightarrow$ $Z^{\top}$)$_{\perp}$

- In $Z^{\top}$, the top element T says that a variable is not always assigned to the same constant value (i.e. it may be assigned to different values).

- An element σ : Var $\rightarrow$ $Z^{\top}$ is a partial function given a variable x, σ (ξ) tells us if x is a constant or not, and in the positive case (if σ (x) is different from T) what is its value.

- The bottom element $\perp$ is added to complete the lattice.

The order in $(Var \to Z^{\top})_{\perp}$

- A partial order in $(Var \to Z^{\top})_{\perp}$

$$\text{forall } \sigma \in (Var \to Z^{\top})_{\perp} : \quad \perp \le \sigma$$

$$\text{forall } \sigma_1, \sigma_2 \in (Var \to Z^{\top})_{\perp} : (\sigma_1 \le \sigma_2)$$

$$\Leftrightarrow (\text{forall } x \in dom(\sigma_1) : \sigma_1(x) \le_{CP} \sigma_2(x))$$

- The least upper bound :

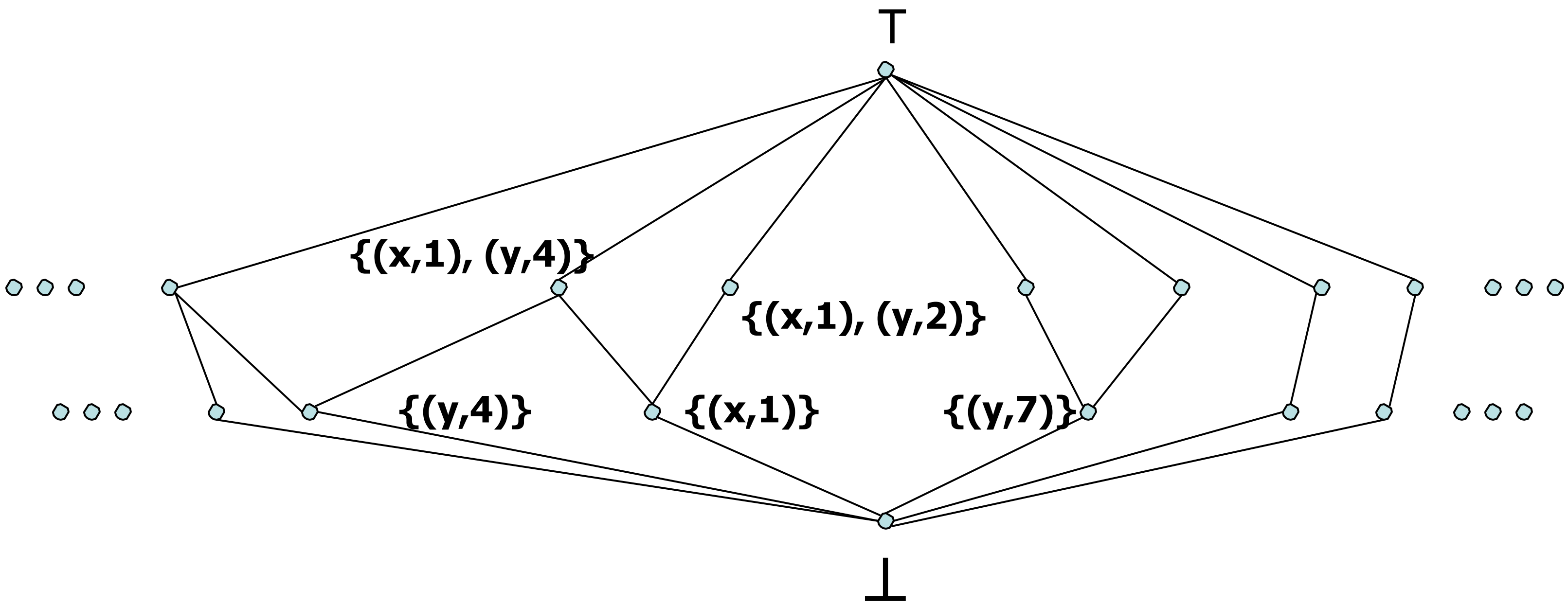
Means equality when $\sigma_i(x)$ are in Z !

$$\text{Forall } \sigma \in (Var \to Z^{\top})_{\perp} : lub(\perp, \sigma) = lub(\sigma, \perp) = \sigma$$

$$\text{Forall } \sigma_1, \sigma_2 \in (Var \to Z^{\top})_{\perp}$$

$$\text{Forall } x \in Var : lub(\sigma_1, \sigma_2)(x) = lub(\sigma_1(x), \sigma_2(x))$$

$(\{x,y\} \rightarrow Z^{\top})_{\perp}$



Expression evaluation

- In order to specify the transfer functions, we have to evaluate an expression given a state σ in $(\text{Var} \to Z^{\top})_{\perp}$

$\mathcal{A}$: $(\text{AExp} \times (\text{Var} \to Z^{\top})_{\perp}) \to Z^{\top}{}_{\perp}$

$\mathcal{A}(x,\sigma) = \perp$ if $\sigma = \perp$
$\quad\sigma(x)$ otherwise

$\mathcal{A}(n,\sigma) = \perp$ if $\sigma = \perp$
$\quad n$ otherwise

$\mathcal{A}(a_1 \text{ op } a_2, \sigma) = \mathcal{A}(a_1,\sigma)\ \underline{\text{op}}\ \mathcal{A}(a_2,\sigma)$

(where $\underline{\text{op}}$ is the corresponding operation of op on $Z^{\top}{}_{*}$: e.g. 4 $\underline{\text{op}}$ 2 = 6)

Transfer functions

- For Constant Propagation Analysis the set of transfer functions is a subset of

  $\mathcal{F} = \{ \text{f} : (\text{Var} \to Z^{\top})_{\perp} \to (\text{Var} \to Z^{\top})_{\perp} \mid \text{f monotone}\}$

- The transfer functions f_l are defined by:

  if λ is the label of an assignment $[x := a]^{\lambda}$

$$f_{\ell}(\sigma) = \begin{cases} \perp & \text{if } \sigma = \perp \\ \sigma[x \to \mathcal{A}(a,\sigma)] & \text{otherwise} \end{cases}$$

  if λ is the label of another statement: $f_{\lambda}(\sigma) = \sigma$

Example

- $[x:=10]^1$; $[y:=x+10]^2$; $([\text{while } x<y]^3\ [y:=y-1]^4)$; $[z:=x-1]^5$

- The minimal solution of the Constant Propagation Analysis of this program is:
-
- $CP_{\texttt{entry}}(1) = \emptyset$
- $CP_{\texttt{exit}}(1) = \{(x \to 10)\}$

  $CP_{\texttt{entry}}(2) = \{(x \to 10)\}$

$CP_{\texttt{exit}}(2) = \{(x \to 10), (y \to 20)\}$

$CP_{\texttt{entry}}(3) = CP_{\texttt{exit}}(3) = CP_{\texttt{entry}}(4) = CP_{\texttt{exit}}(4) = \{(x \to 10), (y \to \mathbf{T})\}$

$CP_{\texttt{entry}}(5) = \{(x \to 10), (y \to \mathbf{T})\}$

$CP_{\texttt{exit}}(5) = \{(x \to 10), (y \to \mathbf{T}), (z \to 9)\}$

Non-distributivity

- In order to show that Constant Propagation Analysis is non distributive, just consider the transfer function f_l corresponding to the statement $[y := x * x]^l$

  consider two states $\sigma_1(x) = 1$ and $\sigma_2(x) = -1$
  in this case:
  $$lub(\sigma_1, \sigma_2)(x) = T$$
  and then
  $$f_l\,(lub(\sigma_1, \sigma_2))(y) = T$$
  whereas
  $$f_l\,(\sigma_1)(y) = 1 = f_l\,(\sigma_2)(y)$$

Interprocedural analysis

Interprocedural Optimizations

- Until now, we have only considered optimizations "within a procedure"
- Extending these approaches outside of the procedural space involves similar techniques:
  - Performing interprocedural analysis
    - Control flow
    - Data flow
  - Using that information to perform interprocedural optimizations

What makes this difficult?

```
procedure joe(i,j,k)
    l ← 2 * k
    if (j = 100)
        then m ← 10 * j
        else m ← i
    call ralph(l,m,k)
    o ← m * 2
    q ← 2
    call ralph(o,q,k)
    write q, m, o, l
```

```
procedure main
    call joe( 10, 100, 1000)

procedure ralph(a,b,c)
    b ← a * c / 2000
```

Since j = 100 this always executes the then clause

and always m has the value 1000

What value is printed for q?
Did ralph() change it?

What happens at a procedure call?

Use worst case assumptions about side effects…

leads to imprecise <u>intra</u>procedural information

leads to explosion in <u>intra</u>procedural def-use chains

What makes this difficult?

```
procedure joe(i,j,k)
    l ← 2 * k
    if (j = 100)
        then m ← 10 *
        j else m ← i
```

```
procedure main
    call joe( 10, 100, 1000)

procedure ralph(a,b,c)
    b ← a * c / 2000
```

Since j = 100 this always executes the then clause

and always m has the value 1000

With perfect knowledge, the compiler could replace this with

write 2, 1000, 2000, 2000

and the rest is dead !

```
write q, m, o, l
```

What value is printed for q? Did ralph() change it?

What happens at a procedure call?

- Use worst case assumptions about side effects
- Leads to imprecise <u>intra</u>procedural information

Leads to explosion in <u>intra</u>procedural def-use chains

The general pattern of Dataflow Analysis

$$GA_i(p) = \begin{cases} \iota & \text{if } p \in E \\ \oplus \{ GA_o(q) \mid q \in F \} & \text{otherwise} \end{cases}$$

$$GA_o(p) = f_p(GA_i(p))$$

where :

E is the set of initial/final points of the control-flow diagram

ι specifies the initial values

F is the set of successor/predecessor points

$\oplus$ is the combination operator

f is the transfer function associated to node p

Procedure calls

- We can label a procedure call
  by: $[\text{call } p(a,z)]^{l_c}_{l_r}$

  where:

  a is an input parameter

  z is an output parameter

  l_c is a label corresponding to the entrance into p

  l_r is a label corresponding to the exit out of p

Flow

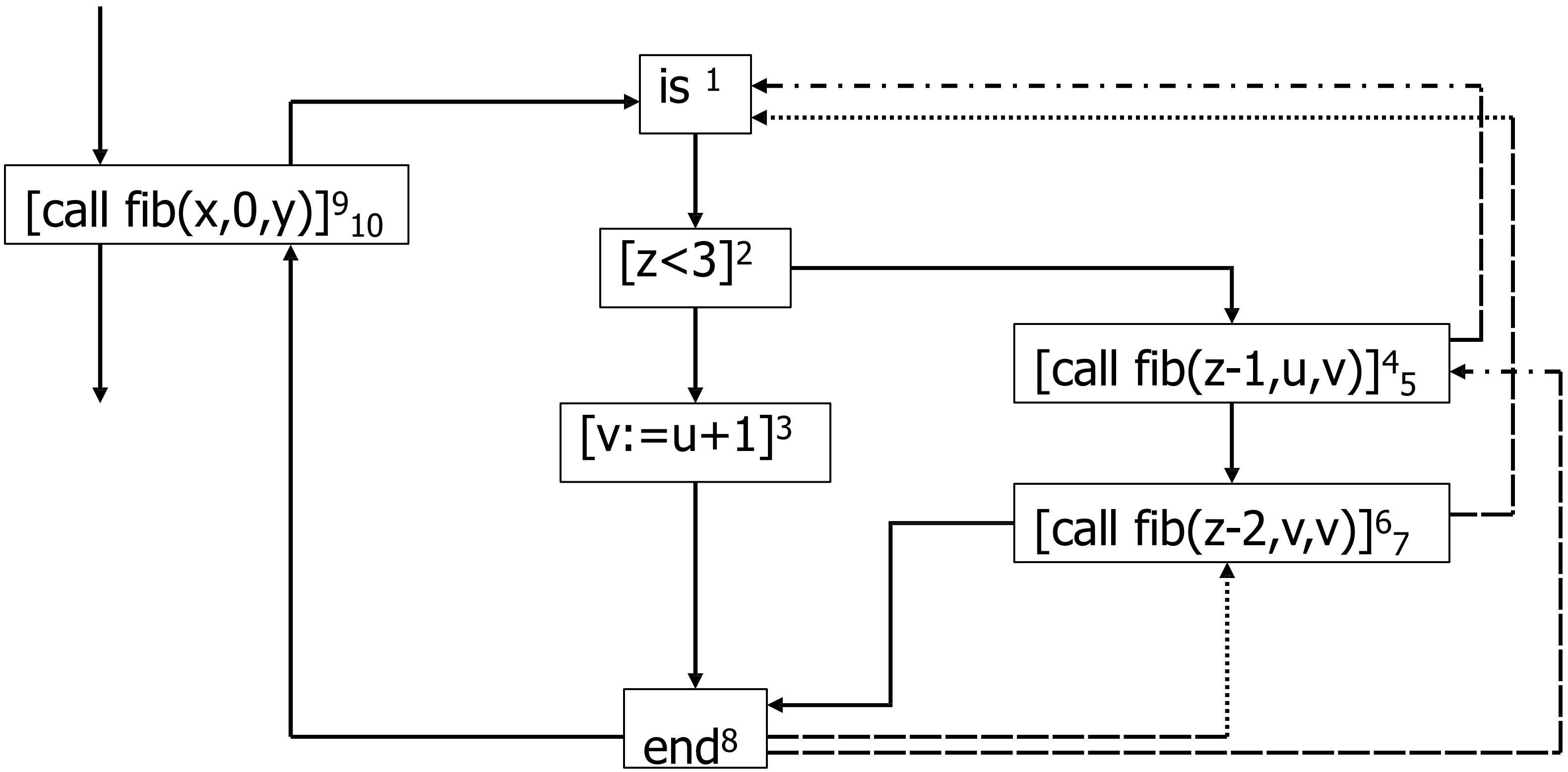
- In the intraprocedural analysis we considered a flow as a set of pairs (p,q) corresponding to an edge in the control flow graph
- We can now consider the call $[\text{call } p(a,z)]^{l_c}_{l_r}$ and a procedure declaration $\text{proc } p(\text{val } x, \text{res } y) \text{ is}^{l_{in}} \text{ S end}^{l_{out}};$
- In the interprocedural graph we should then consider also:
  - $(l_c; l_{in})$ the flow from the call l_c, and the entry label l_{in}
  - $(l_{out}; l_r)$ the flow from the exit label l_{out} to the calling procedure l_r.

Example

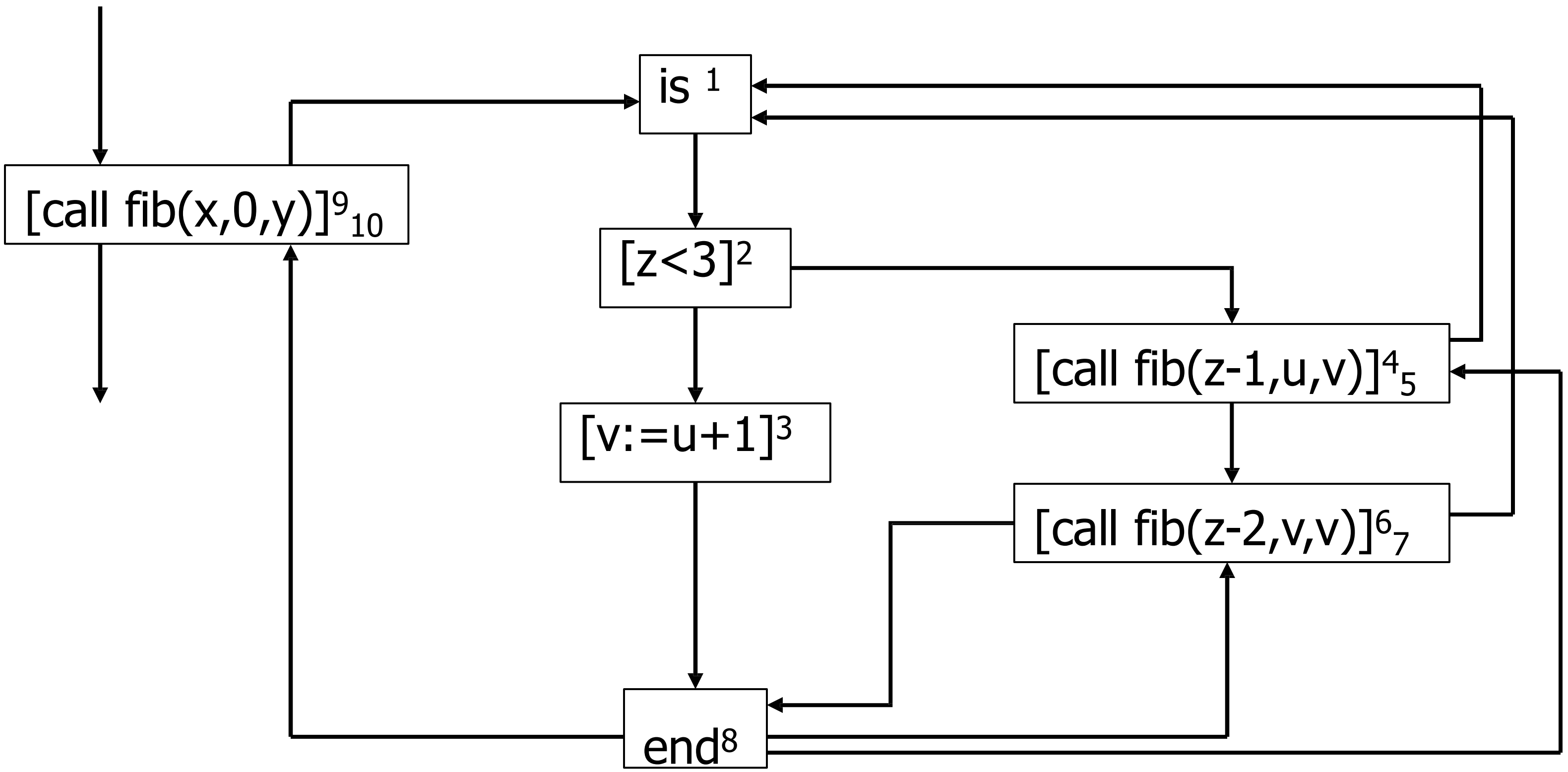
proc p(val x, res y) is$^{l_{in}}$ S end$^{l_{out}}$;

```
proc fib(val: z,u; res: v) is¹
          if [z<3]²
                    then [v:=u+1]³
          else
                    [call fib(z-1,u,v)]⁴₅ ; [call fib(z-2,v,v)]⁶₇
    end⁸;
    [call fib(x,0,y)]⁹₁₀
```

The flow graph



The resulting flattened flow graph



A naive approach

- We may simply extend the dataflow equations using the extended flow

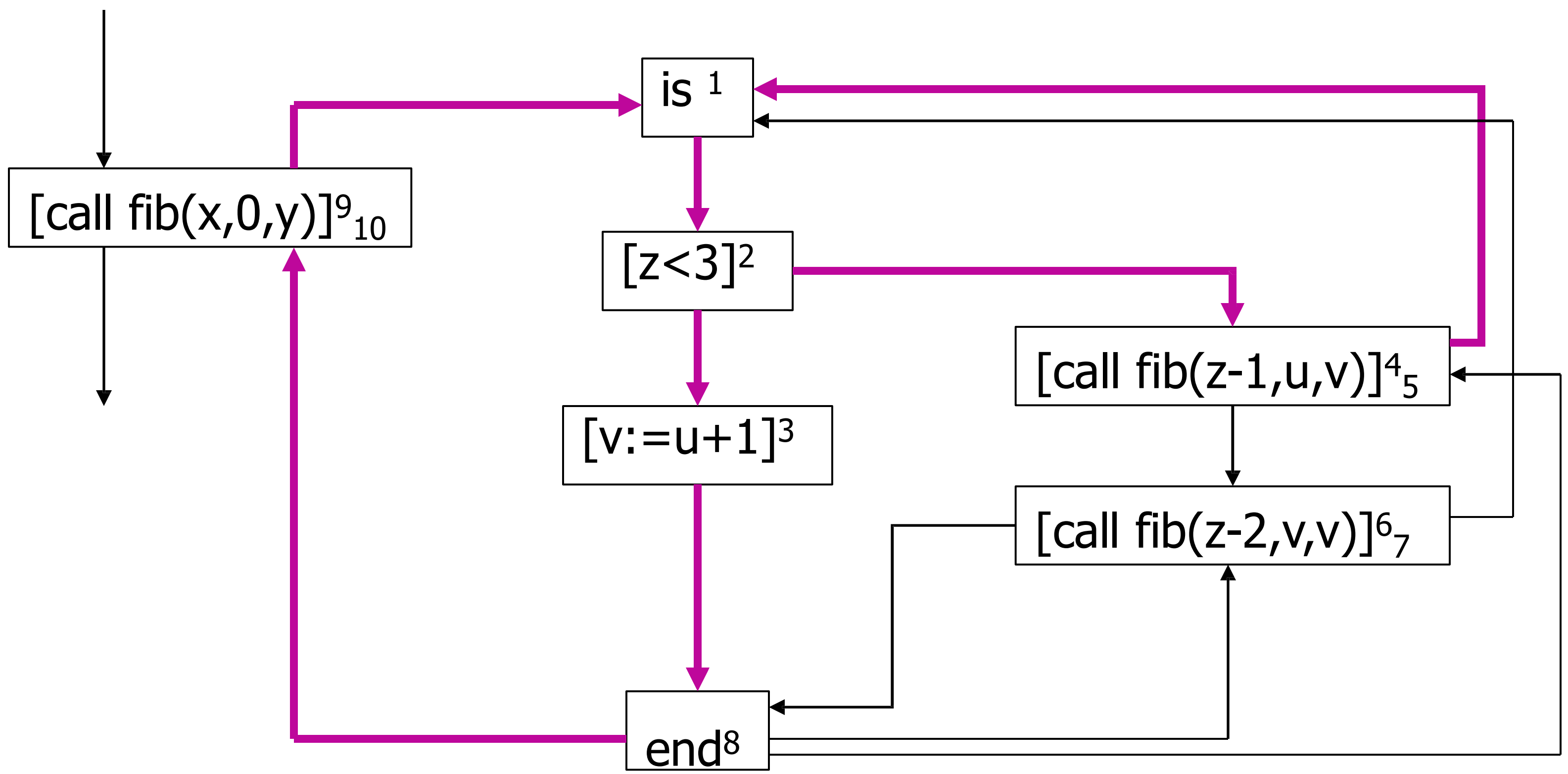
$$GA_{\mathbf{i}}(l) = \begin{cases} \iota & \text{if } l \in E \\ \text{lub}\,\{\, GA_{o}(l') \mid (l', l) \in F \text{ or } (l'; l) \in F \,\} & \text{otherwise} \end{cases}$$

$$GA_{o}(l) = f_{l}\,(\, GA_{\mathbf{i}}(l)\,)$$

Correctness and Accuracy issues

- As we consider all possible paths $(l', l) \in F$ and $(l'; l) \in F$ the analysis is still correct
- However, the analysis also consider the path [9, 1, 2, 4, 1, 2, 3, 8, 10] that does not correspond to any actual computation of the program.
- This deeply affects the accuracy of the analysis

Spurious paths



- The path [9, 1, 2, 4, 1, 2, 3, 8, 10] never occurs in the actual computations

Inter-flow

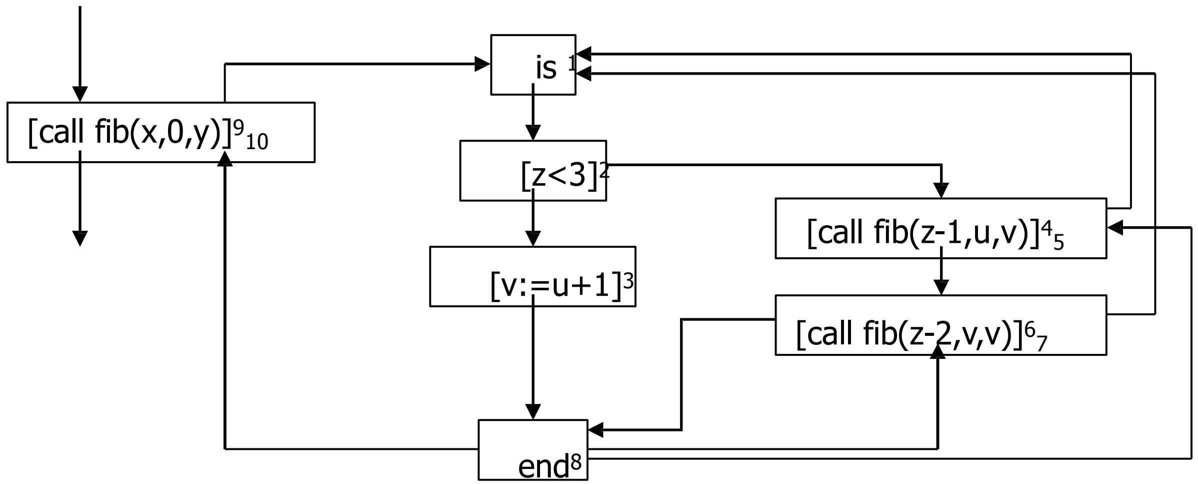
We may define a notion of inter-flow:

inter-flow = $\{(l_c,\ l_{in}\ , l_{out}\ , l_r)$ | the program contains both

[call p(a,z)]$^{l_c}_{l_r}$

and proc p(val x, res y) is$^{l_{in}}$ S end$^{l_{out}}$

Flow and inter-flow



- flow= {(1,2), (2,3), (2,4), (3,8), (4;1), (5,6), (6;1), (7,8), (8;5), (8;7), (8;10), (9;1)}
- Inter-flow= {(9,1,8,10), (4,1,8,5), (6,1,8,7)}

Extending the general framework

$EA_o(l) = f_l\ (\ EA_i(l)\ )$

for all labels l that do not appear as a first or last element of an inter-flow tuple

$EA_i(l) = \bigsqcup \{\ EA_o(l') \mid (l', l) \in F \text{ or } (l'; l) \in F \} \sqcup \iota^l_E$

for all labels l

Moreover, for each inter-flow tuple $(l_c, l_{in}, l_{out}, l_r)$ we introduce the equations:

$EA_o(l_c) = f_{l_c}(\ EA_i(l_c)\ )$

$EA_o(l_r) = f_{l_c, l_r}(\ EA_i(l_c), EA_i(l_r)\ )$