Procedure Syntax for *While*

Previously we had a top-level statement for a *While* program

$$
\begin{array}{lcl}
P & ::= & D\ S \\
D & ::= & \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{l_n} S\ \texttt{end}^{l_x} D \mid \epsilon \\
S & ::= & [x := a]^l \mid \\
 & & [\texttt{call}\ p(a, z)]^{l_c}_{l_r} \mid \\
 & & \ldots
\end{array}
$$

Note that the syntax in the book and in PAG are a bit different.

- no keyword `call` in PAG
- no explicit result parameters and no parameter "mode" descriptions.

Interprocedural Flow Graph (Super Graph)

For completeness we can extend each of the fundamental components of the flow graph for calls

$$
\begin{aligned}
init([\texttt{call}\ \ p(a,z)]^{l_c}_{l_r}) &= l_c \\
final([\texttt{call}\ \ p(a,z)]^{l_c}_{l_r}) &= l_r \\
blocks([\texttt{call}\ \ p(a,z)]^{l_c}_{l_r}) &= \{[\texttt{call}\ \ p(a,z)]^{l_c}_{l_r}\} \\
labels([\texttt{call}\ \ p(a,z)]^{l_c}_{l_r}) &= \{l_c, l_r\} \\
flow([\texttt{call}\ \ p(a,z)]^{l_c}_{l_r}) &= \{(l_c; l_n), (l_x; l_r)\}
\end{aligned}
$$

Inter-procedural edges (IF) are distinguished from intra-procedural edges

$$
\begin{aligned}
\text{inter-flow}_* \quad = \quad & \{(l_c, l_n, l_x, l_r) \mid \\
& P_* \text{ contains } [\texttt{call}\ \ p(a,z)]^{l_c}_{l_r} \text{ and} \\
& \texttt{proc}\ \ p(\texttt{val}\ \ x, \texttt{res}\ \ y)\ \ \texttt{is}^{l_n} S\ \ \texttt{end}^{l_x}\}
\end{aligned}
$$

Example Super Graph

Consider the simple program

```
[0] program test
[1]   proc a(p) is
[2]     y = p;
[3]   end
    begin
[4]   x := 1;
[5]   a(x);

[7]   x := 2;
[8]   a(x);

[10] end
```

Its flow graph can be displayed through PAG/xvcg

What is IF for this example?

Naive Approach to Inter-procedural Analysis

Ignore procedure call semantics.

- This is safe because all valid inter-procedural paths are paths of the super graph.

Its very imprecise

- Lots of invalid inter-procedural paths will be considered.
- e.g., call to `a` at line `[8]` may return after line `[5]`

Naive Reaching Definitions

```
0: program test, { []}
1: proc a, { [(x,7),(y,2),(x,4),]}
2: y := p, { [(x,7),(y,2),(x,4),]}
3: end (a), { [(y,2),(x,7),(x,4),]}
4: x := 1, { []}
5: call a(x), { [(x,4),]}
6: return call a(x), { [(y,2),(x,7),(x,4),]}
7: x := 2, { [(y,2),(x,7),(x,4),]}
8: call a(x), { [(x,7),(y,2),]}
9: return call a(x), { [(y,2),(x,7),(x,4),]}
10: end (test), { [(y,2),(x,7),(x,4),]}
```

`(x,7)` can't possibly reach the output of line `[5]` (*definition of* `x` *hasn't happened yet!*)

`(x,4)` can't reach the output of line `[8]` (*definition on line* `[7]` *should kill it!*)

Inter-procedurally Valid Paths

Procedure calls execute on a stack (in a nested fashion)

```
program test
  proc a(x) is
    if (x>0) then a(x-1);
  end
begin
  a(2);
end
```

Think about how this execution proceeds

Inter-procedurally Valid Paths

Progression of the stack of activation records

```
test
test;a(2)
test;a(2);a(1)
test;a(2);a(1);a(0)
test;a(2);a(1)
test;a(2)
test
```

This kind of behavior can be characterized by a context-free grammar

Valid Path Derivations

One can write a grammar that yields the following derivation steps

$$
\begin{array}{ll}
\mathsf{VP}_* \longrightarrow \mathsf{VP}_{l_1,l_2} & l_1 \in E \text{ and } l_2 \in \mathsf{Lab}_* \\
\mathsf{VP}_{l_1,l_2} \longrightarrow l_1 & l_1 = l_2 \\
\mathsf{VP}_{l_1,l_3} \longrightarrow l_1, \mathsf{VP}_{l_2,l_3} & (l_1, l_2) \in F \\
\mathsf{VP}_{l_c,l} \longrightarrow l_c, \mathsf{CP}_{l_n,l_x}, \mathsf{VP}_{l_r,l} & (l_c, l_n, l_x, l_r) \in \mathsf{IF} \\
\mathsf{VP}_{l_c,l} \longrightarrow l_c, \mathsf{VP}_{l_n,l} & (l_c, l_n, l_x, l_r) \in \mathsf{IF}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{CP}_{l_1,l_2} \longrightarrow l_1 & l_1 = l_2 \\
\mathsf{CP}_{l_1,l_3} \longrightarrow l_1, \mathsf{CP}_{l_2,l_3} & (l_1, l_2) \in F \\
\mathsf{CP}_{l_c,l} \longrightarrow l_c, \mathsf{CP}_{l_n,l_x}, \mathsf{CP}_{l_r,l} & (l_c, l_n, l_x, l_r) \in \mathsf{IF}
\end{array}
$$

Valid Path Derivations

The derivation steps described by the last rule

$$\mathsf{CP}_{l_c,l} \longrightarrow l_c, \mathsf{CP}_{l_n,l_x}, \mathsf{CP}_{l_r,l} \text{ whenever } (l_c, l_n, l_x, l_r) \in \mathsf{IF}$$

assures

- nested matching of $(l_c; l_n)$ and $(l_x; l_r)$ in a valid path
- those edges will only occur in a valid path if their partner does

The two $\mathsf{VP}_{l_c,l}$ rules handle the case where l follows the return from the call and l precedes the return from the call, respectively.

Meet over Valid Paths (MVP)

Recall the definition of MOP at l by $\bigsqcup$ing over all elements of $path_\circ(l)$

It should be clear that the notion of valid paths is a restriction of $path_\circ$

We can define restricted paths and the restricted solution analagous to MOP

$$
\begin{array}{rcl}
vpath_\circ(l) & = & \{[l_1, \ldots, l_{n-1}] \mid n \geq 1 \wedge \\
 & & l_n = l \wedge [l_1, \ldots, l_{n-1}] \text{is a valid path}\} \\
\text{MVP}_\circ(l) & = & \bigsqcup \{ f_{\vec{l}}(\iota) \mid \vec{l} \in vpath_\circ(l) \}
\end{array}
$$

As you might expect $MVP_\circ \sqsubseteq MOP_\circ \sqsubseteq MFP_\circ$

We'll obviously have to approximate the MVP solution

Approaches to Inter-Procedural Analysis

Calculate transfer functions for procedures (Sharir & Pnueli)

Inlining procedure flow graphs

- blow up size of graph
- problems with recursion (k-limiting)
- records top of call stack (naturally)

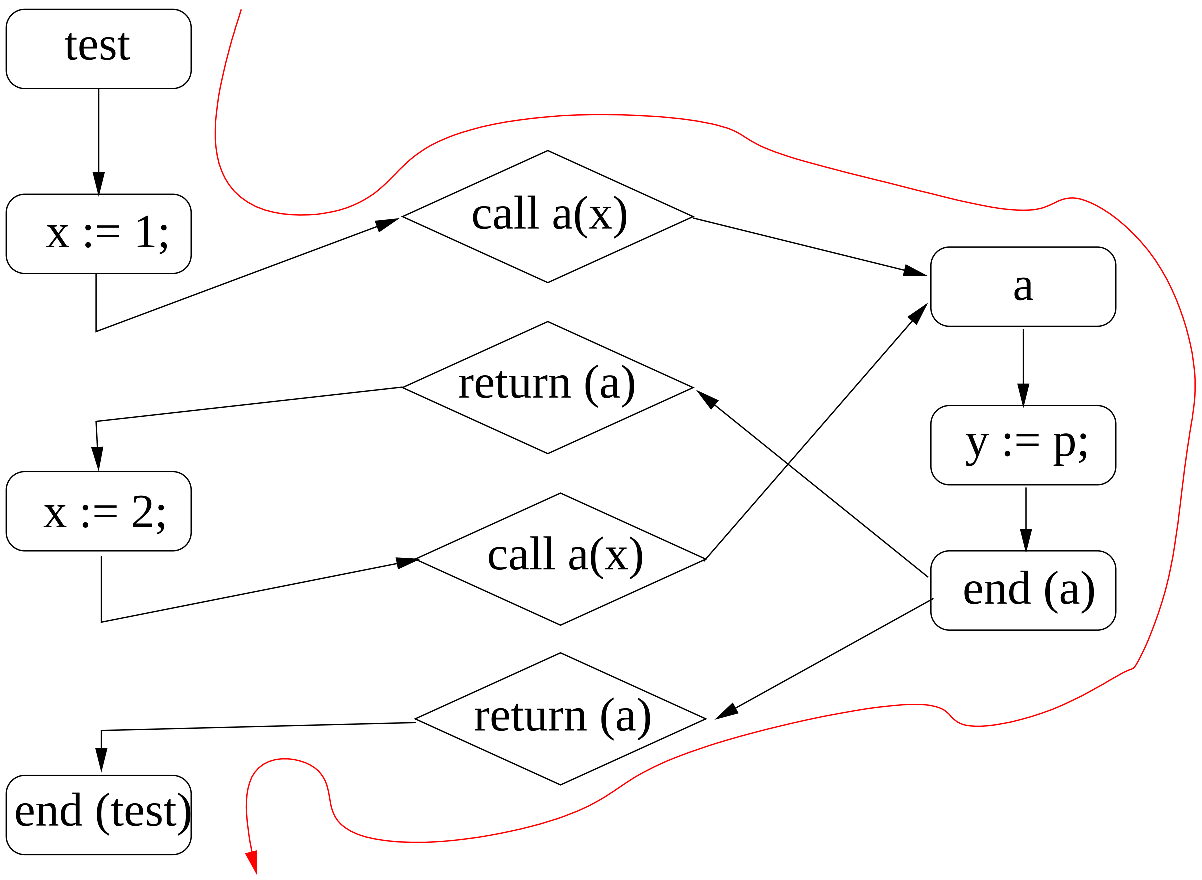
Record information about paths to judge validity

- blow up size lattice
- problems with recursion (k-limiting)
- records top of bottom stack (naturally)

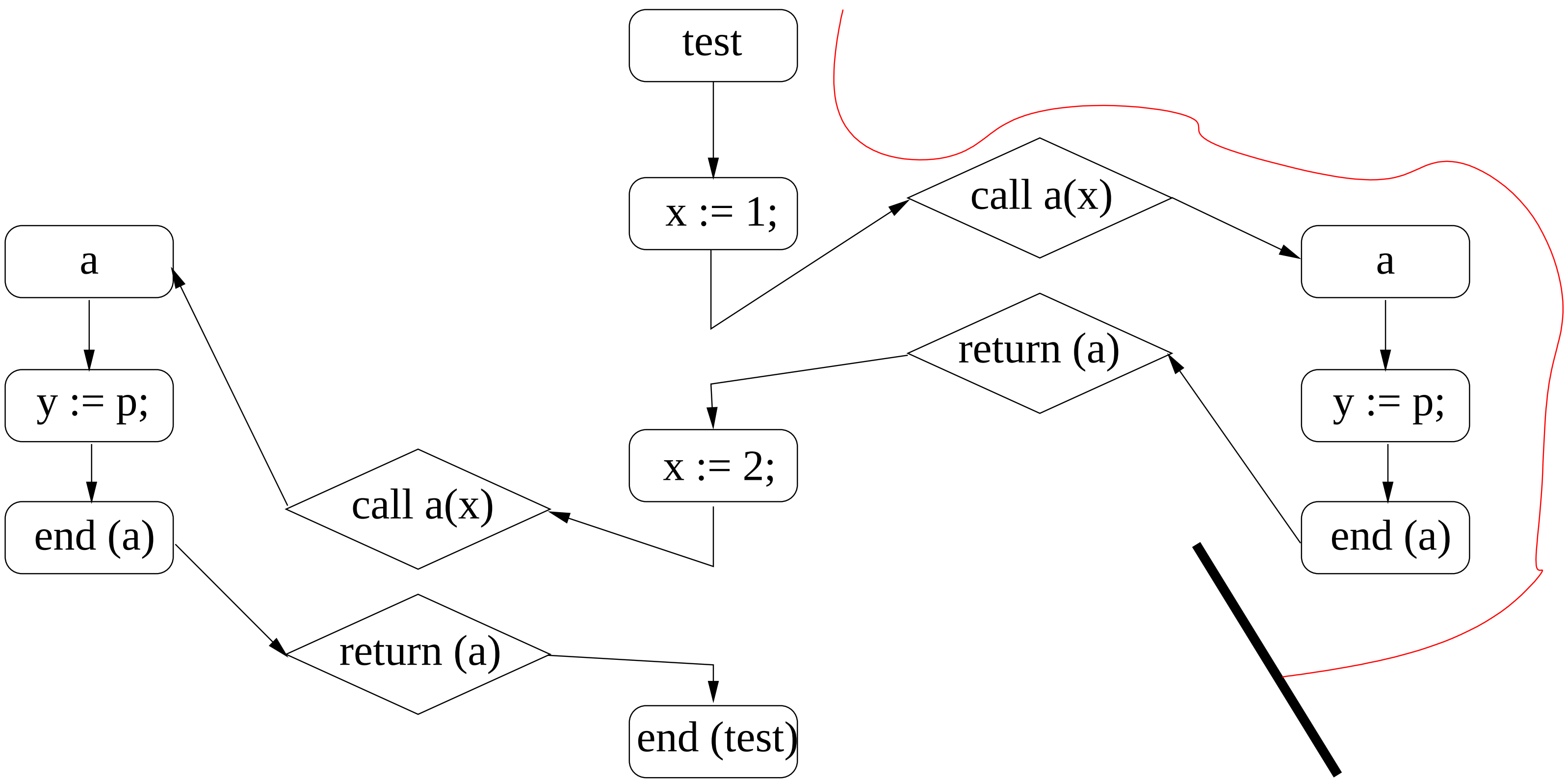
Inlining procedure flow graphs

```
program test
  proc a(p) is
    y = p;
  end
begin
  x := 1;
  a(x);
  x := 2;
  a(x);
 end
```

Super-graph



Inlined procedure flow graph



Explicitly representing calling context

Given a monotone framework $(L, \mathcal{F}, F, E, \iota, f.)$

Construct a new framework that explicitly associates calling context, $\delta \in \Delta$, with data flow facts

- $\widehat{L} = \Delta \to L$
- $\widehat{\mathcal{F}} \supseteq \{\widehat{f_l} \mid \widehat{f_l}(\widehat{l})(\delta) = f_l(\widehat{l}(\delta))\}$
- $\widehat{\iota}$ will map the initial context to ι
- $\widehat{f}.$ will map the functions to labeled statements as before, but will add special functions for local, call and return edges.

The edge relation , F, and extremal values, E, are unchanged.

Intuition

Context will be use to discriminate values that flow to a point along different paths.

In principle, one could record any property of the path to perform this discrimination.

For inter-procedural analysis

- we want to discriminate valid from invalid paths
- we modify the transfer functions for call/return edges since this is where the procedure call semantics come into play
- the rest of the transfer functions are essentially the same

Example : Signs Analysis

For each variable, determine whether it is > 0, $= 0$, or < 0 whenever a given program point is reached (along all paths).

The framework defines a forward problem consisting of

- $L_{sign} = \mathcal{P}(\textsf{Var}_* \to \{+, 0, -\})$
- $f_l^{sign}(Y) = \bigcup\{\phi_l^{sign}(\sigma^{sign}) \mid \sigma^{sign} \in Y\}$ for assignment $[\texttt{x := a}]^l$
- $f_l^{sign}(Y) = Y$ for skips and tests
- ι_{sign} maps all variables to 0

where

- σ^{sign} maps variables to their possible sign.
- $\phi_l^{sign}(\sigma^{sign}) = \{\sigma^{sign}[x \mapsto s] | s \in \mathcal{A}_{sign}[\![a]\!](\sigma^{sign})\}$

- $\mathcal{A}_{sign}$ takes an expression and a signs map and returns the set of signs that expression could evaluate to

Interprocedural Signs Analysis

The given a domain of contexts Δ the enhanced framework consists of

- $\widehat{L_{sign}} = \Delta \rightarrow L_{sign}$ or equivalently
- $\widehat{L_{sign}} = \mathcal{P}(\Delta \times L_{sign})$
- $\widehat{f_l^{sign}}(Z) = \bigcup\{\{\delta\} \times \phi_l^{sign}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z)\}$ for assignment `[x := a]`^{l}
- $\widehat{f_l^{sign}}(Z) = Z$ for skips and tests

This handles the basic transfer functions that simply propogate the context information, δ, assoociated with the incoming signs facts.

Procedure Nodes and Edges

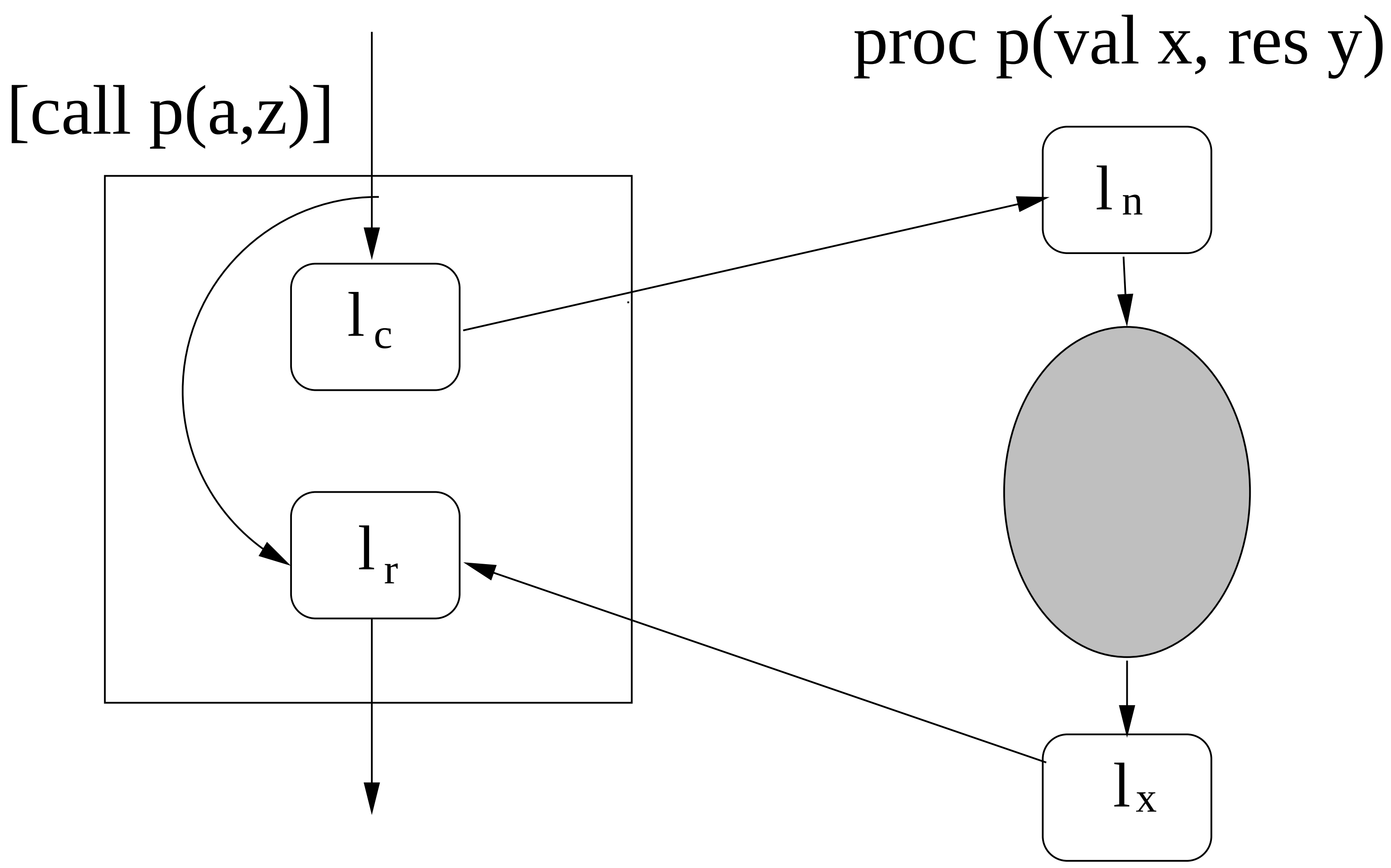
We need to define the transfer functions for each node participating in an inter-procedural edge in the super graph.

For $(l_c, l_n, l_x, l_r) \in$ IF we have three edges

- $(l_c; l_n)$ the *call* edge
- $(l_x; l_r)$ the *return* edge
- $(l_c; l_r)$ the *local* edge (from PAG)

We will take the transfer functions for l_n and l_x to be identity here.

Interprocedural Edges



Call Nodes

Here we wish to define a transfer function that modifies the context information to record the fact that a call happened

They may also modify the data flow facts of the original problem (e.g., for RD)

$\widehat{f^1_{l_c}}$ defines transfer across the call node

The output is propogated to the begin node of the procedure body.

In the discussion that follows, the value propogated along the local edge is the input value to the call node.

Return Nodes

Here we wish to filter the propogation of data flow facts by insuring that only those facts associated with legal contexts at this return point may pass.

To do this, we often need to know the associated call node for this return.

> Unfortunately, PAG doesn't give us access to this directly as it implements inter-procedural analysis internally.

$\widehat{f^2_{l_c,l_r}}$ defines transfer across the return node

To exploit call site information this function takes two $\widehat{L_{sign}}$ values (from local and return edges)

To ignore local information simply define the function as

$$\widehat{f^2_{l_c,l_r}}(\widehat{l},\widehat{l'}) = \widehat{f_{l_r}}(,\widehat{l'})$$

Call Strings

Record the sequence of call sites on the stack of activation records

$\Delta =$ Lab$_*^*$ are *call strings*

> This overloading of $*$ is unfortunate, it means that we have strings of length 0 or greater of labels from the program.

Now the extremal value is defined as

$$\widehat{\iota} = \begin{cases} \iota & \text{if } \delta = \Lambda \\ \perp & \text{otherwise} \end{cases}$$

where Λ is the empty call string

Transfer Functions for IF Nodes

$$\widehat{f^1_{l_c}}(\widehat{l})(\delta') = \begin{cases} f^1_{l_c}(\widehat{l}(\delta)) & \text{if } \delta' = [\delta, l_c] \\ \perp & \text{otherwise} \end{cases}$$

The idea here is to append l_c to the context as the values are propogated across the call node.

Transfer Functions for IF Nodes

$$\widehat{f^2_{l_c,l_r}}(\widehat{l}, \widehat{l'})(\delta') = f^2_{l_c,l_r}(\widehat{l}(\delta), \widehat{l'}([\delta, l_c]))$$

The idea here is to require that the context before the call is equal to the context flowing out of the procedure body, less l_c.

Interprocedural Signs Analysis

We define the call and return transfer functions to complete the analysis

$$\begin{aligned}\widehat{f_{l_c}^{sign1}}(Z) &= \bigcup\{\{\delta'\} \times \phi_{l_c}^{sign1}(\sigma^{sign}) \mid (\delta, \sigma^{sign}) \in Z) \wedge \\ &\quad \delta' = [\delta, l_c]\}\end{aligned}$$

Note there is an error in the book that is corrected here, i.e., the book says $(\delta', \sigma^{sign}) \in Z)$

$$\begin{aligned}\phi_{l_c}^{sign1}(\sigma^{sign}) &= \{\sigma^{sign}[x \mapsto s][y \mapsto s'] | s \in \mathcal{A}_{sign}[\![a]\!](\sigma^{sign})\} \wedge \\ &\quad s' \in \{+, 0, -\}\}\end{aligned}$$

Interprocedural Signs Analysis

Recall that the value flowing across the local edge here is the "input" value to the call node, it has context δ, and the context $\delta' = [\delta, l_c]$ is associated with the value propogated along the call edge.

$$\begin{aligned}\widehat{f^{sign2}_{l_c,l_r}}(Z, Z') \quad &= \quad \bigcup\{\{\delta\} \times \phi^{sign2}_{l_c,l_r}(\sigma_1^{sign}, \sigma_2^{sign}) \mid \\ &\qquad (\delta, \sigma_1^{sign}) \in Z) \wedge (\delta', \sigma_2^{sign}) \in Z' \wedge \delta' = [\delta, l_c]\}\end{aligned}$$

$$\begin{aligned}\phi^{sign2}_{l_c,l_r}(\sigma_1^{sign}, \sigma_2^{sign}) \quad &= \quad \{\sigma_2^{sign}[x \mapsto \sigma_1^{sign}(x)] \\ &\qquad [y \mapsto \sigma_1^{sign}(y)][z \mapsto \sigma_2^{sign}(y)]\}\end{aligned}$$

PAG Call Strings

Recursion implies that one must approximate call strings

PAG uses call strings of length k

- last k calls are recorded
- all of the mechanism for call strings is hidden
- the special processing of return nodes, i.e., that their transfer functions take two arguments, is hidden

Context-sensitive Reaching Definitions

Running: `fcc -M3 1 file.while`

```
0: program test, { []}
1: proc a, { [(x,7),(y,2),(x,4),]}
2: y := p, { [(x,7),(y,2),(x,4),]}
3: end (a), { [(x,7),(y,2),(x,4),]}
4: x := 1, { []}
5: call a(x), { [(x,4),]}
6: return call a(x), { [(y,2),(x,4),]}
7: x := 2, { [(y,2),(x,4),]}
8: call a(x), { [(x,7),(y,2),]}
9: return call a(x), { [(y,2),(x,7),]}
10: end (test), { [(y,2),(x,7),]}
```

`(x,7)` eliminated from input of `[6]`

`(x,4)` eliminated from input of `[9]`