APA
Interprocedural Dataflow Analysis

Jurriaan Hage
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Department of Information and Computing Sciences, Universiteit Utrecht

May 1, 2012

1. The While language with procedures

Procedural programming

- Any sensible programming language supports procedures or functions in some form.
- The main complications that will arise are:
    - How do we propagate analysis information into and out of procedures?
    - A procedure can be jumped to from arbitrarily many locations.
        - Do we join the results over all possible callers?
    - How do we "know" where to return?
        - What if we blindly propagate a single analysis result to all return locations?
- We focus on forward analysis.

Adding procedures to While

- Extend the `While`-language with procedures
- A program takes the form: `begin` D_* S_* `end`
- D_* is a sequence of procedure declarations:

  `proc p(val x, res y) is`$^{\ell_n}$ `S end`$^{\ell_x}$
- `x` and `y` are formal parameters and local to `p`
- A procedure call is a statement: $[\texttt{call p(a,z)}]^{\ell_c}_{\ell_r}$
- `a` is passed by-value and can be any arithmetic expression
- `z` is call-by-result: it can only be used to pass the result back

Information about programs

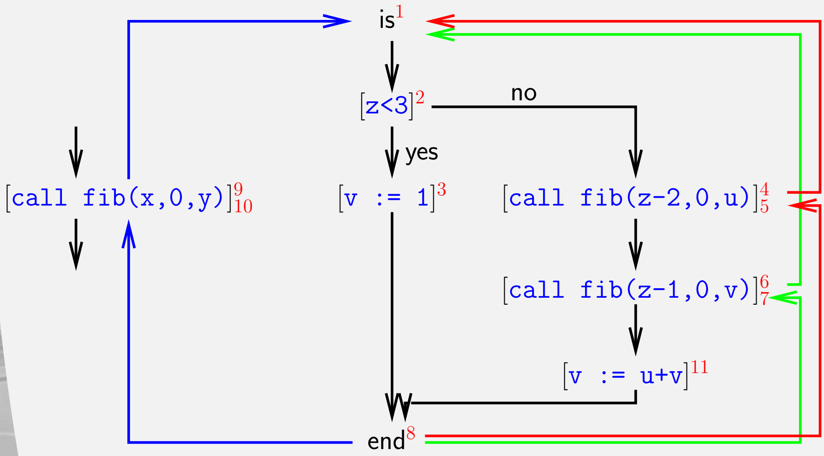
- New block types: `is`, `end` and `call (...)`
- Entry and exit labels attached to `is` and `end`
- Call and return labels attached to `call`
- Add new kind of flow:
  - $(\ell_c; \ell_n)$ for procedure call/entry
  - $(\ell_x; \ell_r)$ for procedure exit/return
- Assume all programs are statically correct:
  - only calls to existing procedures,
  - all labels and procedure names unique.

An example program

```
begin proc fib(val z, u, res v) is$^1$
        if [z<3]$^2$ then [v := 1]$^3$
        else ([call fib(z-2,0,u)]$^4_5$;
              [call fib(z-1,0,v)]$^6_7$;
              [v := v+u]$^{11}$)
    end$^8$;
    [call fib(x,0,y)]$^9_{10}$
end
```

- Syntax more restrictive than examples imply.
- Mimicking local variables: add by-value parameters (like `u`)
- Variables `x` and `y` have global scope
- The scope of `u, v, z` is limited to the body of `fib`.

The flow graph



Meet over all valid paths: MVP

- Generalize the utopian $\mathsf{MOP}_\circ$ and $\mathsf{MOP}_\bullet$ solutions to the more precise $\mathsf{MVP}_\circ$ and $\mathsf{MVP}_\bullet$.
  - Later we consider how to adapt monotone frameworks.
- Paths up to ℓ:

  $\mathsf{vpath}_\circ(\ell) =$

  $$\{[\ell_1, \ldots, \ell_{n-1}] \mid n \geq 1, \ell_n = \ell, [\ell_1, \ldots, \ell_n] \text{ a valid path}\}$$
- $\mathsf{MVP}_\circ(\ell) = \bigsqcup\{f_{\overrightarrow{\ell}}(\iota) \mid \overrightarrow{\ell} \in \mathsf{vpath}_\circ(\ell)\}$
- Similarly for the closed case, $\mathsf{MVP}_\bullet(\ell)$.
- But what is a valid path?

Unbalance and poisoning

```
begin proc neg(val z, res u) is$^1$
        $[$u := -z$]^2$)
    end$^3$;
    $[$call neg(-1,p)$]_6^5$;
    $[$call neg(1,n)$]_8^7$
end
```

- Suppose we treat $(5;1)$ like $(5,1)$?
- Suppose we want to track the signs of all variables.
- Poisoning: information about the first call to `neg` also flows to the second call. Reasonable?
- $\text{path}_\circ$ and $\text{path}_\bullet$ do not always pair call labels correctly with the label of the return.
- Valid paths, on the other hand, are balanced.
- $[5,1,2,3,8]$ is not valid, but $[5,1,2,3,6]$ is.

Use valid paths and context instead

- Issues when defining valid paths
  - Consider only balanced executions.
  - During analysis we only consider finite prefixes of these,
  - including finite prefixes of infinite ones.

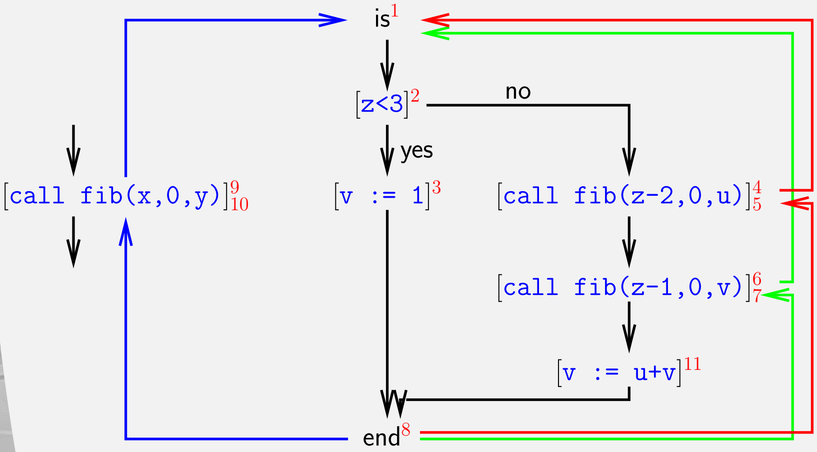
```
begin proc infinite(val n, res x) is^2
            [call infinite(0,x)]^3_4;
        end^5;
        [call infinite(0,x)]^1_6
end
```

- Context can be used to enforce balance:
  - it can simulate/abstract behaviour of a call stack.
- The amount of abstraction determines complexity and precision.

Interprocedural flows

- The previous slides motivate a need to distinguish interprocedural and intraprocedural flow.
- For the fibonacci program:
  $\texttt{flow}(S_*) = \{(1,2),(2,3),(3,8),(2,4),(4;1),(8;5),(5,6),(6;1),(8;7),(7,11),(11,8),(9;1),(8;10)\}$
- Interprocedural:
  $\texttt{inter-flow}(S_*) = \{(\mathbf{9},\mathbf{1},\mathbf{8},\mathbf{10}),(\mathbf{4},\mathbf{1},\mathbf{8},\mathbf{5}),(\mathbf{6},\mathbf{1},\mathbf{8},\mathbf{7})\}$
  4-tuples of call and corresponding return information.
- $(9,1,8,5) \notin \texttt{inter-flow}(S_*)$
- $\texttt{init}(S_*) = 9$ and $\texttt{final}(S_*) = \{10\}$
- Backward variants exist: $\texttt{flow}^R$ and $\texttt{inter-flow}^R$

The flow graph again



Intermediate summary

- Changes to the programming language have now been made.
    - syntax,
    - scoping rules,
    - MOP is generalized to MVP
- Now come the changes to the monotone framework
    - reuse as much as possible of intraprocedural monotone framework,
    - transfer functions for the new statements,
    - distinguish between certain execution paths via context.

2. Embellished Monotone Frameworks

Towards embellished monotone frameworks

- From monotone framework to embellished monotone framework.
- We proceed by example.
  - Define a monotone framework for Detection Of Signs Analysis.
  - Specify the form of transfer functions for calls, entries, exits and returns.
  - Change it to include context so that data flows along balanced paths,
  - by lifting the original transfer functions so that they include context,
  - and making sure that procedure call and return imply a context change.
- Context can be "anything", but we restrict ourselves later so that context helps us to analyze only valid paths.

Detection of Sign Analysis

- Let $(L, \mathcal{F}, F, E, \iota, \lambda\ell.f_\ell)$ be an instance of a monotone framework for Detection of Sign Analysis (Exercise 2.15)
- Detection of Signs gives for each program point what signs each variable may have at that program point.
- Beware: my notation differs from that in NNH.

Detection of Sign Analysis - the lattice

- The complete lattice L consists of sets of functions
- More precisely: elements of $\mathcal{P}(\textbf{Var}_* \rightarrow S)$ with $S = \{-, 0, +\}$
- Each function describes a set of executions leading to a certain program point.
- Example: $\{g, h\} \in L$ with $g(x) = g(y) = +$, and $h(x) = +$ and $h(y) = -$
- In other words,
  - there are executions where x and y are both positive and
  - there are executions where x is positive and y is negative.

Detection of Sign Analysis example

- Assume **Var**$_*$ $= \{x, y\}$,
  $g(x) = +$ and $g(y) = +$, and
  $h(x) = +$ and $h(y) = -$.
- Consider the effect of $[$`x := x+y`$]^\ell$ on g:
  - the function g' which maps both x and y to $+$ (so $g = g'$)
- The effect of $[$`x := x+y`$]^\ell$ on h is
  - map y to $-$, but x to $-, 0$ or $+$
  - the result is described by three functions, h_-, h_0 and h_+, defined as $h_i(y) = -$ and $h_i(x) = i$ (for all i).
- The set $\{g, h\}$ is thus mapped to $\{g', h_-, h_0, h_+\}$.

Relational vs. independent

- Recall: the set $\{g, h\}$ was mapped to $\{g, h_-, h_0, h_+\}$.
    - g tells us y can be mapped to $+$, the h_i that y maps to $-$.
    - The h_i tell us that x can map to any one of the $\{0, -, +\}$.
- Analysis is relational: we store combinations of x and y.
- To save on resources, merge the functions to a set of signs for each variable: x has signs $\{0, -, +\}$ and y has $\{+, -\}$
    - Thereby becoming an independent attributes analysis.
- This value also represents the previously known to be impossible $[x \mapsto -, y \mapsto +]$ and $[x \mapsto 0, y \mapsto +]$.
- The independent attribute analysis is really weaker,
    - but also less resource consuming.

Interpreting expressions

- $\mathcal{A}_s : \textbf{AExp} \to (\textbf{Var}_* \to S) \to \mathcal{P}(S)$ gives all possible signs of an expression, when given a sign for each variable.
- $\mathcal{A}_s[\![\texttt{x+y}]\!][x \mapsto +, y \mapsto +] = \{+\}$
- $\mathcal{A}_s[\![\texttt{x+y}]\!][x \mapsto +, y \mapsto -] = \{0, +, -\}$

Transfer functions

- Transfer function for $[\texttt{x := a}]^\ell$ maps sets of functions to sets of functions:

$$f_\ell(Y) = \bigcup\{\phi_\ell(\sigma) \mid \sigma \in Y\}$$

  where $Y \in L$ and $\phi_\ell(\sigma) = \{\sigma[x \mapsto s] \mid s \in \mathcal{A}_s[\![\texttt{a}]\!](\sigma)\}$

- Functions may "split up":

$$\phi_\ell([x \mapsto +, y \mapsto -]) =$$

$$\{[x \mapsto -, y \mapsto -], [x \mapsto 0, y \mapsto -], [x \mapsto +, y \mapsto -]\}$$

- Finally $f_\ell(Y)$ collects everything:

$$\{[x \mapsto +, y \mapsto +], [x \mapsto -, y \mapsto -],$$

$$[x \mapsto 0, y \mapsto -], [x \mapsto +, y \mapsto -]\}$$

Adding context to the lattice

- Add context to get an embellished monotone framework $(\widehat{L}, \widehat{\mathcal{F}}, F, E, \widehat{\iota}, \lambda\ell.\widehat{f_\ell})$
- The complete lattice L becomes $\Delta \to L$:
  $$\mathcal{P}(\textbf{Var}_* \to S) \text{ becomes } \Delta \to \mathcal{P}(\textbf{Var}_* \to S)$$
- "Omit" context by taking Δ a one element set.
- For each $\delta \in \Delta$ we may have a different value in L.
  - δ serves as an index.
- $\widehat{L}$ is a complete lattice (page 398 of NNH) .
- In the book they use $\mathcal{P}(\Delta \times (\textbf{Var}_* \to S)) \cong \Delta \to \mathcal{P}(\textbf{Var}_* \to S)$. We don't.

Lifting the transfer functions

- We have a transfer function $f_\ell : L \to L$.
- Lift pointwise to $\widehat{f_\ell} : (\Delta \to L) \to (\Delta \to L)$:

$$\widehat{f_\ell}(\widehat{l}) = \lambda\delta \to f_\ell(\widehat{l}(\delta)) \text{ for } \widehat{l} \in \widehat{L}$$

- Or simply, $\widehat{f_\ell}(\widehat{l}) = f_\ell \circ \widehat{l}$
- In words, apply old transfer function independently, i.e., pointwise, for each value in Δ.
- Example:
  $\widehat{f_\ell}([\delta_1 \mapsto \{g\}, \delta_2 \mapsto \{h, g\}]) =$
  $[\delta_1 \mapsto f_\ell(\{g\}), \delta_2 \mapsto f_\ell(\{g, h\})] =$
  $[\delta_1 \mapsto \{g\}, \delta_2 \mapsto \{h_0, h_-, h_+, g\}]$.

Data flow in the new set-up

- Information flows along dataflow graph edges:

$$A_\circ(\ell) = \bigsqcup \{A_\bullet(\ell') \mid (\ell', \ell) \in F \vee (\ell'; \ell) \in F\} \sqcup \widehat{\iota_E^\ell}$$

- So for procedure entry labels, we take the join over all callers.
- How do we tell different calls apart? By using context.
- Transfer almost as usual:

$$A_\bullet(\ell) = \widehat{f}_\ell(A_\circ(\ell))$$

- Call and return are somewhat different.

What happens for a (forward) procedure call?

- Assume a call to procedure p: $(\ell_c, \ell_n, \ell_x, \ell_r) \in \texttt{inter-flow}(S_*)$.
- Two transfer functions: f_{ℓ_c} and f_{ℓ_n}.
- f_{ℓ_n} is the same for every call to p.
  - In NNH always identity function.
- f_{ℓ_c} can be different for each call to p.
- "Chronologically":
  - transfer value at call $A_\bullet(\ell_c) = f_{\ell_c}(A_\circ(\ell_c))$
  - compute $A_\circ(\ell_n)$ by joining $A_\bullet$ for all calls to p.
  - transfer value at entry: $A_\bullet(\ell_n) = f_{\ell_n}(A_\circ(\ell_n))$
    - Often the identity function
  - value ready to flow through p.
- f_{ℓ_c} is typically a function that knows about context.

What happens at procedure return?

- Procedure return encompasses the real difference:

$$A_\bullet(\ell_r) = \widehat{f^2_{\ell_c,\ell_r}}(A_\circ(\ell_c), A_\circ(\ell_r))$$

- Transfers information from inside the procedure and from before the call to just after the call.
- Note: $A_\circ(\ell_r)$ is (normally) just $A_\bullet(\ell_x)$.
- Information before a call can be passed directly to after the call.
  - Instead of propagating it through the call.
- $\widehat{f^2_{\ell_c,\ell_r}}$ may ignore one (or both) arguments.
- For a backward analysis, the transfer functions change arity: the one for call becomes binary, the one for return becomes unary.

Call strings as context

- Context intends to keep analyses of separate calls separated
- Call string: list of addresses from which a call was made.
  - Abstraction of the call stack: $\Delta = [\mathbf{Lab}_*]$
- For `fib`: Λ, $[4], [6], [9], [4,4], \ldots, [9,9], [4,4,4], \ldots$
  - Generate only when needed.
- Call-string abstracts an execution into the labels of calls seen during execution without seeing the corresponding return: $[1,6,5,8,3,2,1,4,2,1,9]$ becomes $[6,9]$
- Procedure call labels are added to the front (stack like).

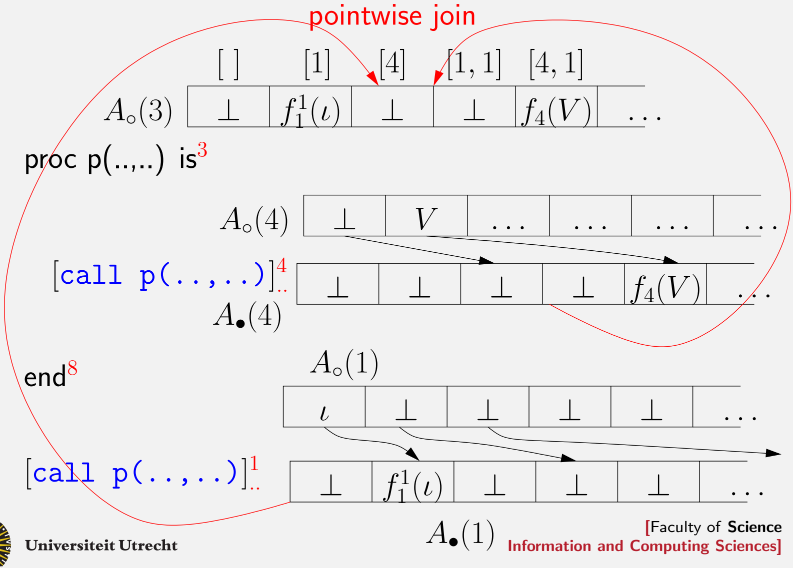
Call strings as context

- Call string: list of addresses from which a call was made.
- For $(\ell_c, \ell_n, \ell_x, \ell_r)$ we define

$$\widehat{f^1_{\ell_c}}(\widehat{l})(\ell_c : \delta) = f^1_{\ell_c}(\widehat{l}(\delta)) \qquad \text{and} \qquad \widehat{f^1_{\ell_c}}(\widehat{l})(\Lambda) = \bot$$

- f_1 computes the effect of a call
- and $\widehat{f_1}$ selects where the effect values should go.
- Valid paths simulated by the transferring between "corresponding" call strings.

Call strings example snapshot



pointwise join

	[ ]	[1]	[4]	[1, 1]	[4, 1]	
$A_\circ(3)$	$\perp$	$f_1^1(\iota)$	$\perp$	$\perp$	$f_4(V)$	...

proc p(..,..) is^3

$A_\circ(4)$	$\perp$	V	...	...	...	...

`[call p(..,..)]`$^4_{..}$

$A_\bullet(4)$	$\perp$	$\perp$	$\perp$	$\perp$	$f_4(V)$	...

end^8

$A_\circ(1)$	ι	$\perp$	$\perp$	$\perp$	$\perp$	...

`[call p(..,..)]`$^1_{..}$

$A_\bullet(1)$	$\perp$	$f_1^1(\iota)$	$\perp$	$\perp$	$\perp$	...

Call strings as context, return

- Similarly, for procedure return:
  $\widehat{f^2_{\ell_c,\ell_r}}(\widehat{l},\widehat{l'})(\delta) = f^2_{\ell_c,\ell_r}(\widehat{l}(\delta),\widehat{l'}(\ell_c:\delta))$
- We use two values:
  - from before the call, which is under the same context as the return,
  - from inside the procedure, which is under the extended call string.

Detection of Signs: procedure calls

- Assume $[\texttt{call p(a,x)}]_{\ell_r}^{\ell_c}$ and

  $\texttt{proc p(val x, res y) is}^{\ell_n}\ \texttt{S end}^{\ell_x}$
- A call consists of two assignments `x := a` and `y := ?`.
  - The context-less transfer function mimicks those.
- For $\sigma = [x \mapsto +, z \mapsto -]$ and `a = -x` we ought to obtain

  $$\begin{aligned}\phi_{\ell_c}(\sigma) = \{&[x \mapsto -, y \mapsto -, z \mapsto -],\\ &[x \mapsto -, y \mapsto 0, z \mapsto -],\\ &[x \mapsto -, y \mapsto +, z \mapsto -]\}\end{aligned}$$
- Semantics says value of y is undefined (instead of 0).
- New x “shadows” the old.
- In general, unshadow when returning.

Detection of Signs: procedure calls

- Assume $[\texttt{call p(a,x)}]^{\ell_c}_{\ell_r}$ and
  $\texttt{proc p(val x, res y) is}^{\ell_n}\ \texttt{S end}^{\ell_x}$
- For $\sigma = [x \mapsto +, z \mapsto -]$ and $\texttt{a} = \texttt{-x}$ we ought to obtain
  $$\begin{aligned}\phi_{\ell_c}(\sigma) = \{&[x \mapsto -, y \mapsto -, z \mapsto -],\\ &[x \mapsto -, y \mapsto 0, z \mapsto -],\\ &[x \mapsto -, y \mapsto +, z \mapsto -]\}\end{aligned}$$
- $f_{\ell_c}(Z) = \bigcup\{\phi_{\ell_c}(\sigma) \mid \sigma \in Z\}$
- $\phi_{\ell_c}(\sigma) =$
  $$\{\sigma[x \mapsto s][y \mapsto s'] \mid s \in \mathcal{A}_s[\![\texttt{-x}]\!](\sigma) \land s' \in \{0, +, -\}\}$$

Detection of Signs: adding context

- Consider the function $Z \in \widehat{L} = \Delta \to L$

$$Z = [\Lambda \mapsto \sigma_1, \delta_2 \mapsto \sigma_2, \ldots]$$

- We want to obtain

$$[\Lambda \mapsto \bot, [\ell_c] \mapsto f^1_{\ell_c}(\sigma_1), (\ell_c : \delta_2) \mapsto f^1_{\ell_c}(\sigma_2), \ldots]$$

- So $\widehat{f^1_{\ell_c}}(Z)$ is such that for all $\delta \in \Delta$

$$\widehat{f^1_{\ell_c}}(Z)(\delta') = \begin{cases} \bot & \text{if } \delta' = \Lambda \\ f^1_{\ell_c}(Z(\delta)) & \text{if } \delta' = \ell_c : \delta \end{cases}$$

- Warning: in NNH they give the same general formula, but the example of Detection of Signs (2.38) uses different notation.

Call strings of bounded size

- L might have ACC, but $\Delta \to L$ might not
  - Call strings can be arbitrarily long for recursive programs
- Enforce termination by restricting length call strings to $\leq k$
- For every different list of call labels, potentially a different analysis result: quickly exponential.

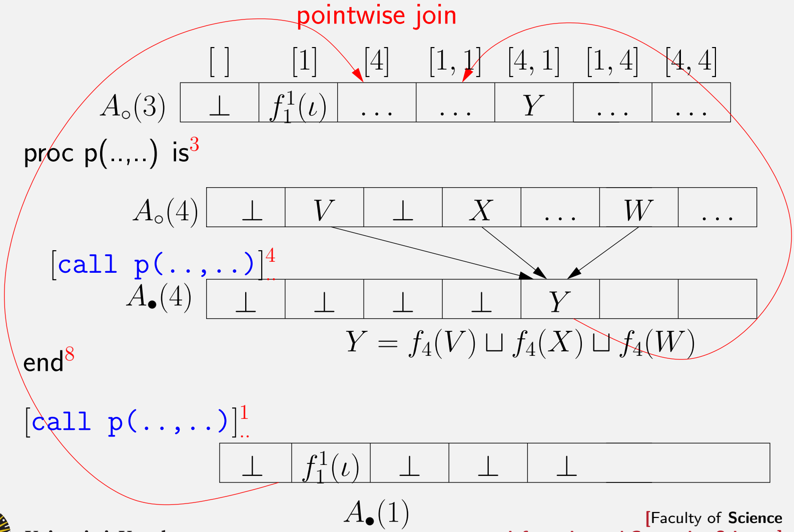
What if we run out of bounds?

- Assume $k = 2$.
- Consider call from 4 either with context $[1,4]$, $[1,1]$ or $[1]$.
- Then in all three cases, the context inside the call will be $[4,1]$.
- To stay sound we must join the transferred analysis results.
- Here's where we gain finiteness at the price of precision.
- In a formula

$$\widehat{f^4_{\ell_c}}(Z)([4,1]) = f^4_{\ell_c}(Z([1,4])) \sqcup f^4_{\ell_c}(Z([1,1])) \sqcup f^4_{\ell_c}(Z([1]))$$

- We can choose the level of detail (value of k) with a known price to pay.
  - Take $k = 0$ to omit context: Δ then equals $\{\Lambda\}$

$k = 2$ bounded call strings, snapshot



pointwise join

	[ ]	[1]	[4]	[1, 1]	[4, 1]	[1, 4]	[4, 4]
$A_\circ(3)$	$\perp$	$f_1^1(\iota)$	...	...	Y	...	...

proc p(..,..) is^3

$A_\circ(4)$	$\perp$	V	$\perp$	X	...	W	...

`[call p(..,..)]`$^4_{..}$

$A_\bullet(4)$	$\perp$	$\perp$	$\perp$	$\perp$	Y		

$$Y = f_4(V) \sqcup f_4(X) \sqcup f_4(W)$$

end^8

`[call p(..,..)]`$^1_{..}$

$\perp$	$f_1^1(\iota)$	$\perp$	$\perp$	$\perp$	

$A_\bullet(1)$

Separate the context from the transfer

- Context is never used to compute the transfer, it only tells you which part of the value to use (and update).
- For different analyses you can use the same kind of context and context change
- In an implementation: decouple the context change from transfer
  - The former selects which values influence a given value.
  - The latter says how.

Flow-sensitive versus flow-insensitive

- Flow-sensitive vs. flow-insensitive: does the result of the analysis depend on the order of statements? Again a matter of cost vs. precision.
- To go from flow-insensitive to flow-sensitive: add program points as a form of context.
- In NNH, flow-sensitivity is hard-coded into the framework.

Final remarks about procedures

- Except for binary transfer functions, the technical changes are slight.
- Conceptually, changes may be bigger.
- For termination, restrict context to finite sets of values.
- Use context to balance cost and precision.
- Simple monotone frameworks can be easily extended to become embellished.
    - A first step in building an analysis.
- Analyzing procedures can be a pain when scoping enters the picture.