

Interprocedural Analysis: Sharir-Pnueli's Call-Strings Approach

Deepak D'Souza

Department of Computer Science and Automation
Indian Institute of Science, Bangalore.

11 September 2024

Outline

- 1 Motivation
- 2 Call-strings method
- 3 Correctness
- 4 Bounded call-string method
- 5 Approximate call-string method

Handling programs with procedure calls

How would we extend an abstract interpretation to handle programs with procedures?

```
main(){  
  x := 0;  
  f();  
  g();  
  print x;  
}
```

```
f(){  
  x := x+1;  
  return;  
}
```

```
g(){  
  f();  
  return;  
}
```

Handling programs with procedure calls

How would we extend an abstract interpretation to handle programs with procedures?

```
main(){                                f(){                                g(){
  x := 0;                               x := x+1;                            f();
  f();                                   return;                            return;
  g();                                   }                                }
  print x;                              }
}
```

Question: what is the collecting state before the `print x` statement in `main`?

Handling programs with procedure calls

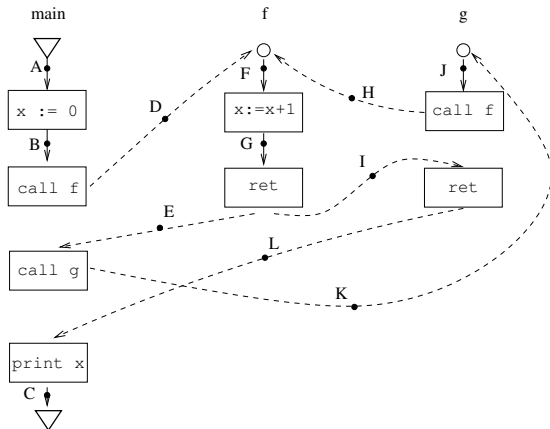
How would we extend an abstract interpretation to handle programs with procedures?

```
main(){                                f(){                                g(){
  x := 0;                             x := x+1;                            f();
  f();                                return;                            return;
  g();                                }                                }
  print x;                            }
}
```

Question: what is the collecting state before the `print x` statement in `main`? Answer: $x \mapsto 2$.

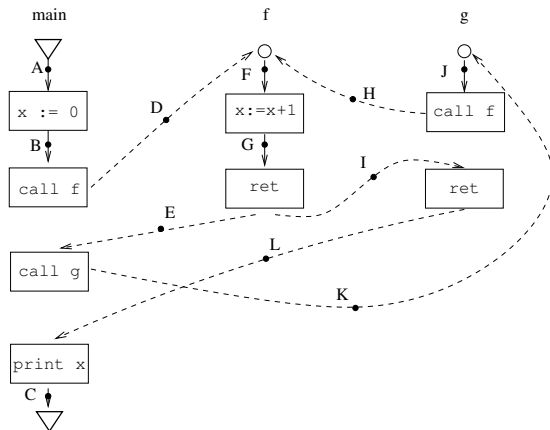
Handling programs with procedure calls

- Add extra edges
 - **call edges**: from call site (call p) to start of procedure p
 - **ret edges**: from return statement (in p) to point after call sites (call p) ("ret sites").



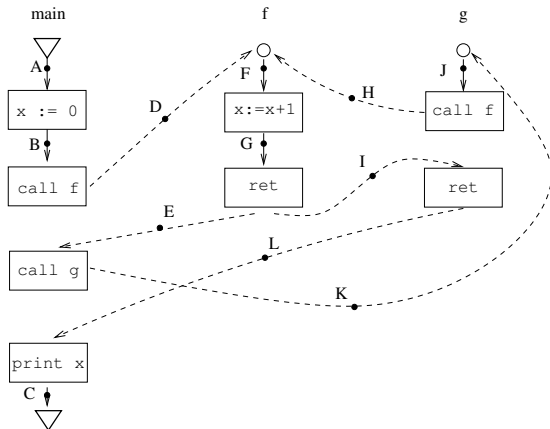
Handling programs with procedure calls

- Assume only global variables.
- Transfer functions for call/return edges?



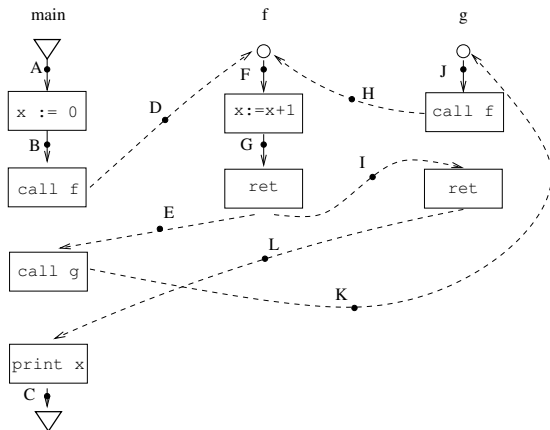
Handling programs with procedure calls

- Assume only global variables.
- Transfer functions for call/return edges? Identity function



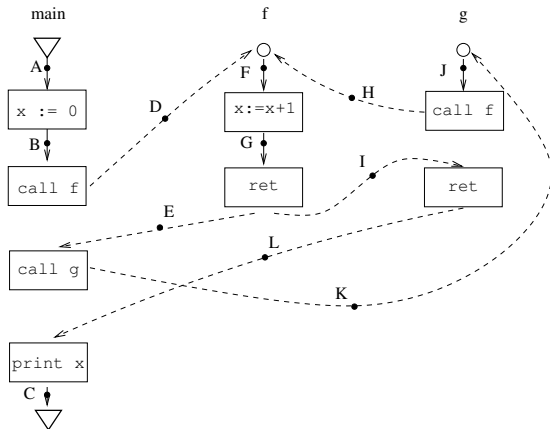
Handling programs with procedure calls

- Assume only global variables.
- Transfer functions for call/return edges? Identity function
- Now compute JOP in this extended control-flow graph.



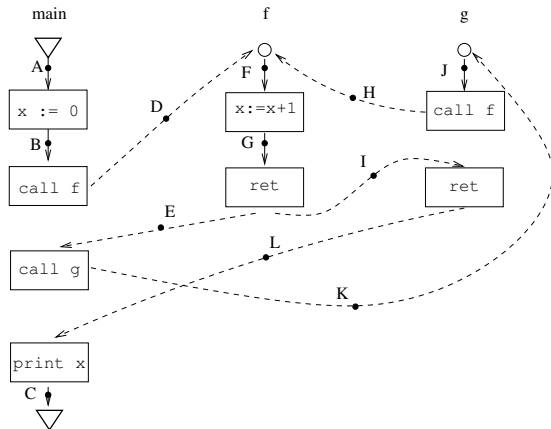
Problem with JOP in this graph

Ex. 1. Actual collecting state at C?



Problem with JOP in this graph

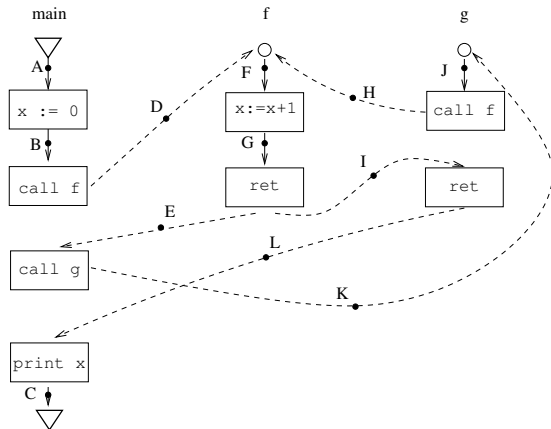
Ex. 1. Actual collecting state at C? $\{x \mapsto 2\}$.



Problem with JOP in this graph

Ex. 1. Actual collecting state at C? $\{x \mapsto 2\}$.

Ex. 2. JOP at C using collecting analysis?

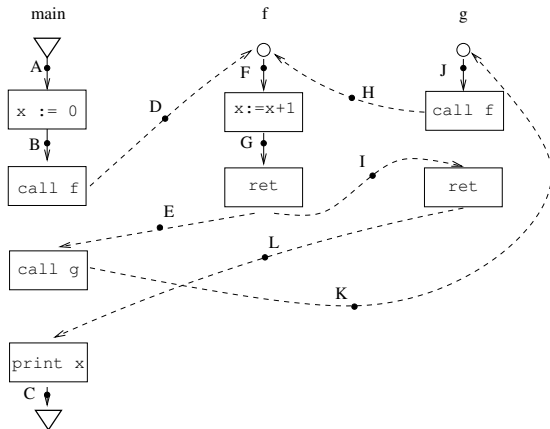


Problem with JOP in this graph

Ex. 1. Actual collecting state at C? $\{x \mapsto 2\}$.

Ex. 2. JOP at C using collecting analysis?

$\{x \mapsto 1, x \mapsto 2, x \mapsto 3, \dots\}$.



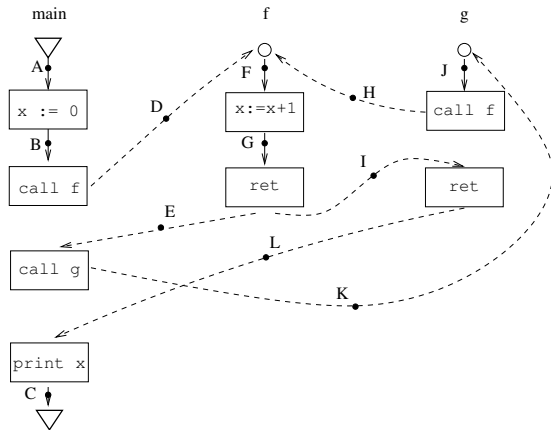
Problem with JOP in this graph

Ex. 1. Actual collecting state at C? $\{x \mapsto 2\}$.

Ex. 2. JOP at C using collecting analysis?

$\{x \mapsto 1, x \mapsto 2, x \mapsto 3, \dots\}$.

- JOP is sound but very **imprecise**.
- Reason: Some paths don't correspond to executions of the program: Eg. ABDFGILC.



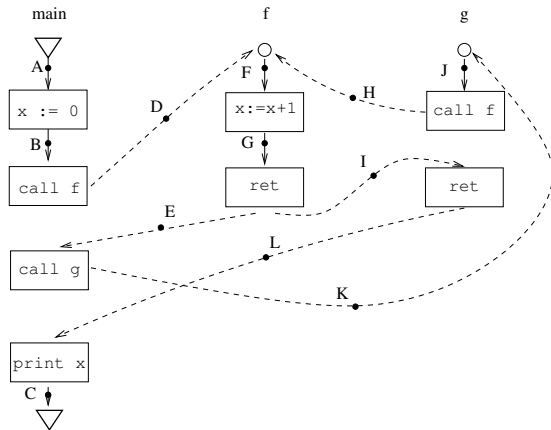
Problem with JOP in this graph

Ex. 1. Actual collecting state at C? $\{x \mapsto 2\}$.

Ex. 2. JOP at C using collecting analysis?

$\{x \mapsto 1, x \mapsto 2, x \mapsto 3, \dots\}$.

- JOP is sound but very **imprecise**.
- Reason: Some paths don't correspond to executions of the program: Eg. ABDFGILC.



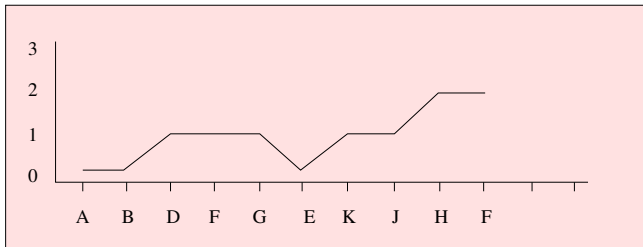
What we want is Join over “Interprocedurally-Valid” Paths (**JVP**).

Interprocedurally valid paths and their call-strings

- Informally a path ρ in the extended CFG G' is inter-procedurally valid if every return edge in ρ “corresponds” to the most recent “pending” call edge.
- For example, in the example program the ret edge E corresponds to the call edge D .
- The call-string of a valid path ρ is a subsequence of call edges which have not been “returned” as yet in ρ .
- For example, $cs(ABDFGEKJHF)$ is “KH”.

Interprocedurally valid paths and their call-strings

- A path $\rho = ABDFGEKJHF$ in $IVP_{G'}$ for example program:



- Associated call-string $cs(\rho)$ is KH .
- For $\rho = ABDFGEK$ $cs(\rho) = K$.
- For $\rho = ABDFGE$ $cs(\rho) = \epsilon$.

Interprocedurally valid paths and their call-strings

More formally: Let ρ be a path in G' . We define when ρ is **interprocedurally valid** (and we say $\rho \in IVP(G')$) and what is its **call-string** $cs(\rho)$, by induction on the length of ρ .

- If $\rho = N$ and N is either an internal edge or a call edge, then $\rho \in IVP(G')$ and $cs(\rho)$ is ϵ or N respectively.
- If $\rho = \rho' \cdot N$ then $\rho \in IVP(G')$ iff $\rho' \in IVP(G')$ with $cs(\rho') = \gamma$ say, and one of the following holds:
 - 1 N is an internal edge.
In this case $cs(\rho) = \gamma$.
 - 2 N is a call edge.
In this case $cs(\rho) = \gamma \cdot N$.
 - 3 N is ret edge, and γ is of the form $\gamma' \cdot C$, and N corresponds to the call edge C .
In this case $cs(\rho) = \gamma'$.
- We denote the set of call-strings that can occur along **initial** IVP paths in G' by Γ .
- Call-strings of example program are given by regexp $\epsilon + OL^*$.

Join over interprocedurally-valid paths (JVP)

- Let P be a given program, with extended CFG G' .
- Let $path_{I,N}(G')$ be the set of paths from the initial point I to point N in G' .
- Let $\mathcal{A} = ((D, \leq), f_{MN}, d_0)$ be an abstract interpretation for P .
- Then we define the **join over all interprocedurally valid paths (JVP)** at point N in G' to be:

$$\bigsqcup_{\rho \in path_{I,N}(G') \cap IVP(G')} f_{\rho}(d_0).$$

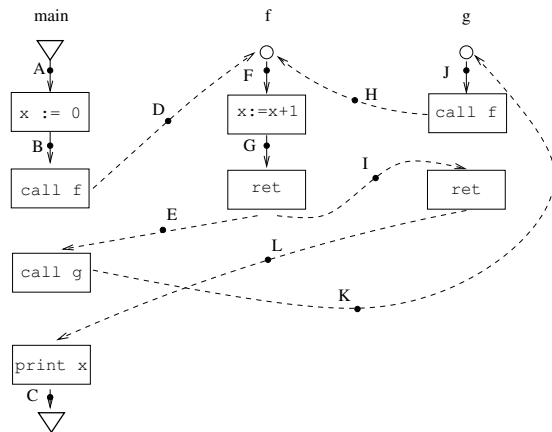
Sharir and Pnueli's approaches to interprocedural analysis



Micha Sharir and Amir Pnueli: Two approaches to interprocedural data flow analysis, in *Program Flow Analysis: Theory and Applications* (Eds. Muchnick and Jones) (1981).

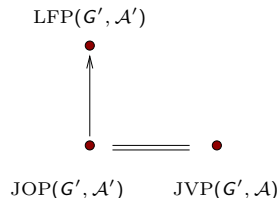
One approach to obtain JVP: Call-Strings

- Find JOP over same graph, but modify the abs int.
- Modify transfer functions for call/ret edges to detect and invalidate invalid edges.
- Augment underlying data values with some information for this.
- Natural thing to try: “call-strings”.



Overall plan

- Define an abs int $\mathcal{A}'$ which extends given abs int $\mathcal{A}$ with call-string data.
- Show that JOP of $\mathcal{A}'$ on G' coincides with JVP of $\mathcal{A}$ on G' .
- Use Kildall (or any other technique) to compute LFP of $\mathcal{A}'$ on G' . This value over-approximates JVP of $\mathcal{A}$ on G' .



Call-string abs int $\mathcal{A}'$: Lattice $(D', \leq')$

- Elements of D' are maps $\xi : \Gamma \rightarrow D$

$$\xi :$$

ϵ	c_1	$c_1 c_2$	$c_1 c_2 c_2$	
d_0	d_1	d_2	d_3	

- Ordering on D' : $\leq'$ is the pointwise extension of $\leq$ in D . That is

$\xi_1 \leq' \xi_2$ iff for each $\gamma \in \Gamma, \xi_1(\gamma) \leq \xi_2(\gamma)$.

$$\xi' :$$

ϵ	c_1	$c_1 c_2$	$c_1 c_2 c_2$	
d'_0	d'_1	d'_2	d'_3	

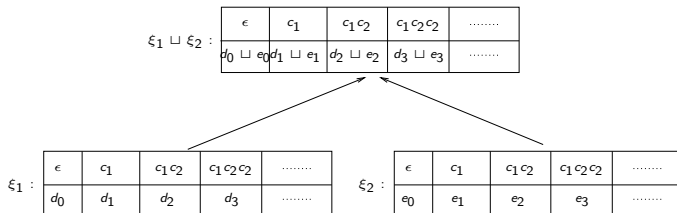


$$\xi :$$

ϵ	c_1	$c_1 c_2$	$c_1 c_2 c_2$	
d_0	d_1	d_2	d_3	

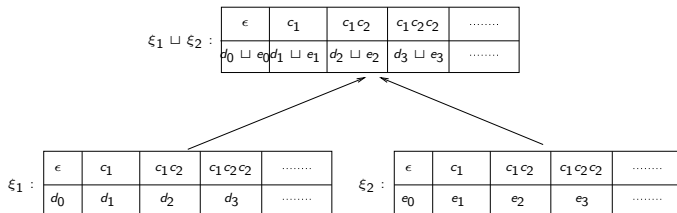
Call-string abs int $\mathcal{A}'$: Lattice $(D', \leq')$

- Induced join is:



Call-string abs int $\mathcal{A}'$: Lattice $(D', \leq')$

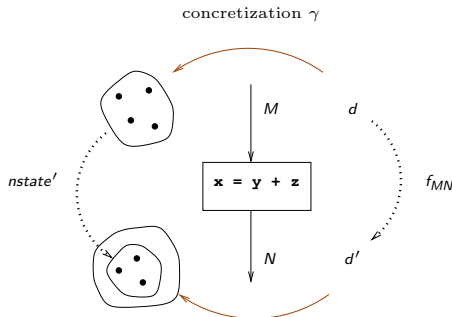
- Induced join is:



- Check that $(D', \leq')$ is also a complete lattice.

Designing a correct transfer function

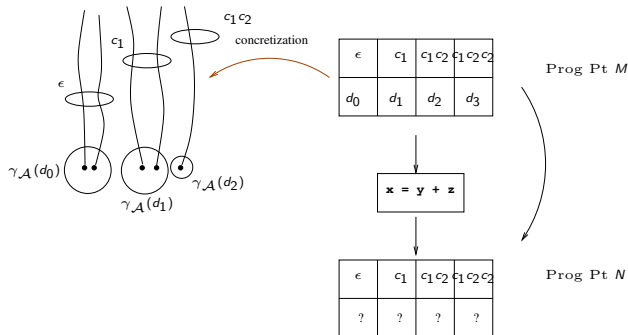
Given components $(D, \leq)$ and abstraction (α) and concretization (γ) functions for elements of D , your transfer functions must satisfy (in addition to monotonicity):



If $\gamma(d)$ were the **only** concrete states that could arise at M , then $\gamma(d')$ should over-approximate the resulting states at N (i.e. $nstate'_{MN}(\gamma(d))$).

Designing correct transfer functions in $\mathcal{A}'$

- A call-string table ξ at program point N represents the set of concrete states (say a concrete state here is of the form (s, γ) where s is a standard concrete state and γ is a call-string of the execution giving rise to s) obtained by concretizing $\xi(\gamma)$ and tagging each concrete state with γ , for each $\gamma \in \Gamma$.
- The transfer functions of $\mathcal{A}'$ should keep this meaning in mind.



Call-string abs int $\mathcal{A}'$: Initial value ξ_0

- Initial value ξ_0 is given by

$$\xi_0(\gamma) = \begin{cases} d_0 & \text{if } \gamma = \epsilon \\ \perp & \text{otherwise.} \end{cases}$$

$\xi_0 :$

ϵ	c_1	c_1c_2	$c_1c_2c_2$	
d_0	$\perp$	$\perp$	$\perp$	

Call-string abs int $\mathcal{A}'$: transfer functions

- Transfer functions for non-call/ret edge N :

$$f'_{MN}(\xi) = f_{MN} \circ \xi.$$

- Transfer functions for call edge N :

$$f'_{MN}(\xi) = \lambda\gamma. \begin{cases} \xi(\gamma') & \text{if } \gamma = \gamma' \cdot N \\ \perp & \text{otherwise} \end{cases}$$

- Transfer functions for ret edge N whose corresponding call edge is C :

$$f'_{MN}(\xi) = \lambda\gamma. \xi(\gamma \cdot C)$$

- Transfer functions f'_{MN} are monotonic (resp. distributive) if each f_{MN} is monotonic (resp. distributive).

Transfer functions f'_{MN} for example program

- Non-call/ret edge B :

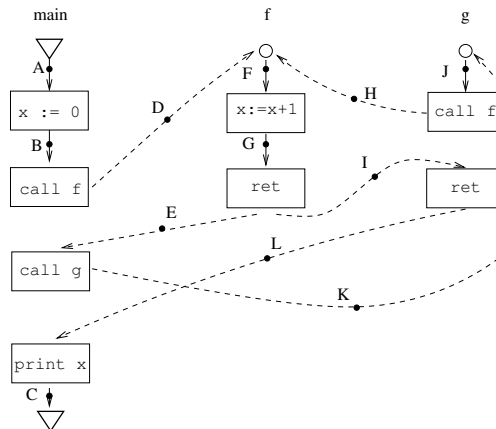
$$\xi_B = f_{AB} \circ \xi_A.$$

- Call edge D :

$$\xi_D(\gamma) = \begin{cases} \xi_B(\gamma') & \text{if } \gamma = \gamma' \cdot D \\ \perp & \text{otherwise} \end{cases}$$

- Return edge E :

$$\xi_E(\gamma) = \xi_G(\gamma \cdot D).$$

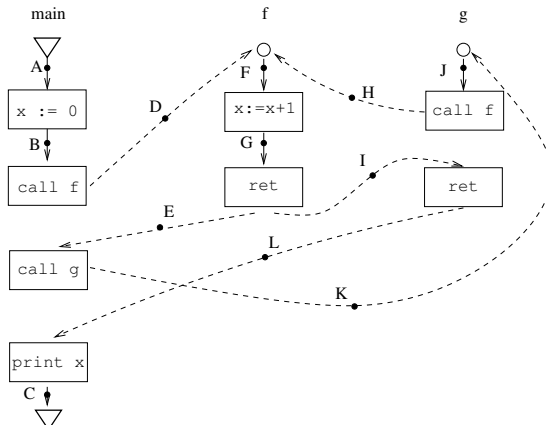


Exercise 1

Let $\mathcal{A}$ be the standard collecting state analysis. For brevity, represent a set of concrete states as $\{0, 1\}$ (meaning the 2 concrete states $x \mapsto 0$ and $x \mapsto 1$). Assume an initial value $d_0 = \{0\}$.

Show the call-string tagged abstract states (in the lattice $\mathcal{A}'$) along the paths

- 1 ABDFGEKJHFGIL (interprocedurally valid)
- 2 ABDFGIL (interprocedurally invalid).



Correctness claim

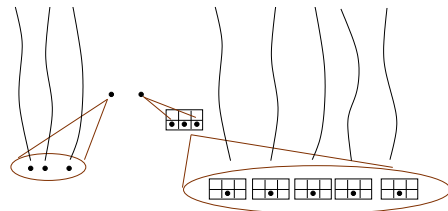
Assumption on $\mathcal{A}$: Each transfer function satisfies $f_{MN}(\perp) = \perp$.

Claim

Let N be a point in G' . Then

$$JVP_{\mathcal{A}}(N) = \bigsqcup_{\gamma \in \Gamma} JOP_{\mathcal{A}'}(N)(\gamma).$$

Proof: Use following lemmas to prove that LHS dominates RHS and vice-versa.



IVP Paths reaching N

Paths reaching N

Correctness claim: Lemma 1

Lemma 1

Let ρ be a path in $IVP_{G'}$. Then

$$f'_\rho(\xi_0) = \lambda\gamma. \begin{cases} f_\rho(d_0) & \text{if } \gamma = cs(\rho) \\ \perp & \text{otherwise.} \end{cases}$$

ϵ	c_1	$cs(\rho)$	$c_1 c_2 c_2$	
$\perp$	$\perp$	d	$\perp$	

Proof: by induction on the length of ρ .

Correctness claim: Lemma 2

Lemma 2

Let ρ be a path **not in** $IVP_{G'}$. Then

$$f'_\rho(\xi_0) = \lambda\gamma.\perp.$$

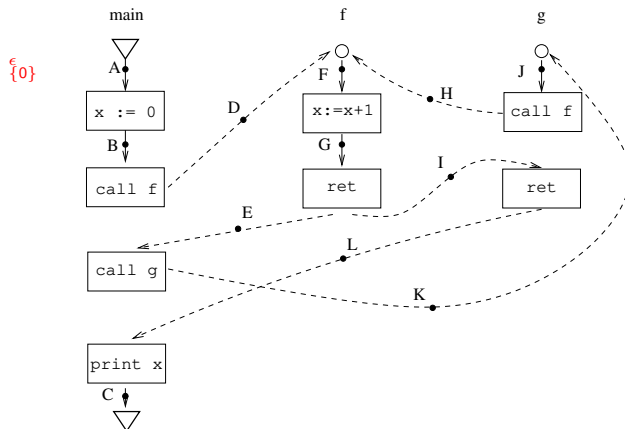
ϵ	c_1	c_2	$c_1c_2c_2$	
$\perp$	$\perp$	$\perp$	$\perp$	

Proof:

- ρ must have an invalid prefix.
- Consider smallest such prefix $\alpha \cdot N$. Then it must be that α is valid and N is a return edge not corresponding to $cs(\alpha)$.
- Using previous lemma it follows that $f'_{\alpha.N}(\xi_0) = \lambda\gamma.\perp$.
- But then all extensions of α along ρ must also have transfer function $\lambda\gamma.\perp$.

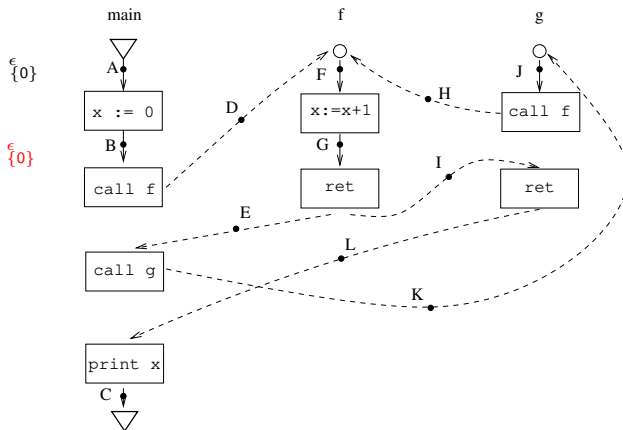
Exercise 2

Use Kildall's algo to compute the LFP of the $\mathcal{A}'$ analysis for the example program. Start with initial value $d_0 = \{0\}$.



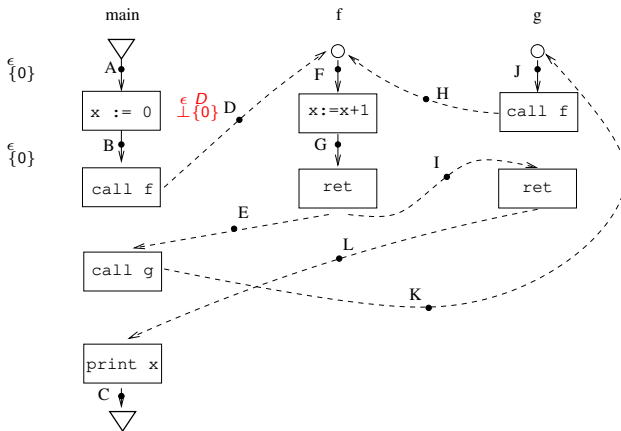
Exercise 2

Use Kildall's algo to compute the LFP of the $\mathcal{A}'$ analysis for the example program. Start with initial value $d_0 = \{0\}$.



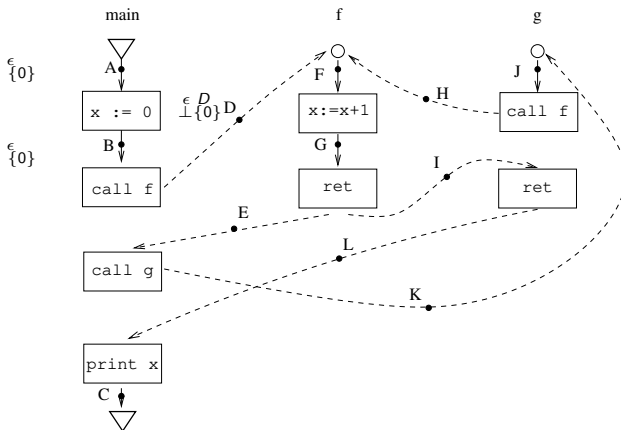
Exercise 2

Use Kildall's algo to compute the LFP of the $\mathcal{A}'$ analysis for the example program. Start with initial value $d_0 = \{0\}$.



Exercise 2

Use Kildall's algo to compute the LFP of the $\mathcal{A}'$ analysis for the example program. Start with initial value $d_0 = \{0\}$.

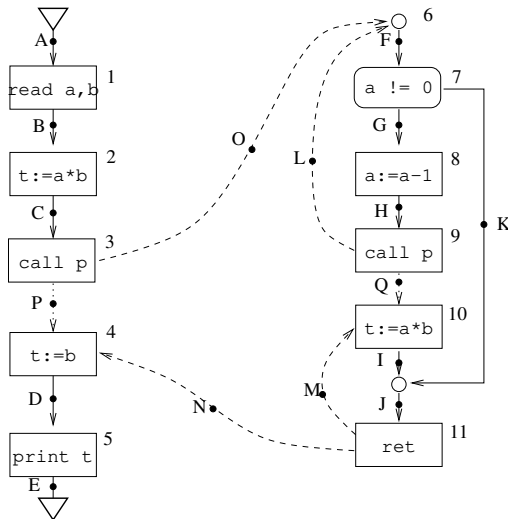


Computing JOP for abs int $\mathcal{A}'$

- Problem is that D' is infinite in general (even if D were finite). So we cannot use Kildall's algo to compute an over-approximation of JOP (it may not terminate when the program has recursive procedures).

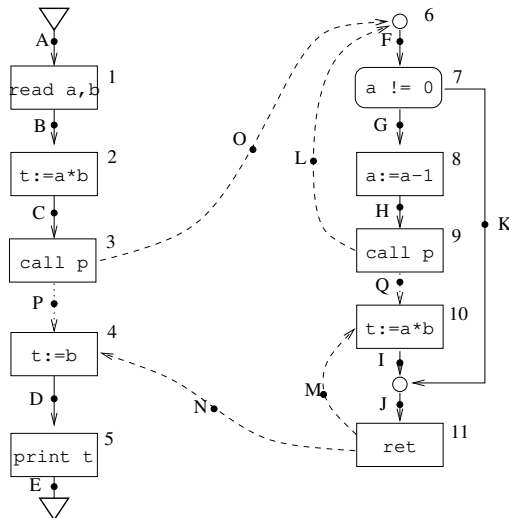
Available expressions

- An expression (like " $a * b$ ") is **available** along an execution if there is a point where the expression is evaluated and thereafter none of the constituent variables (like a and b) are written to.
- An expression is **available** at a point N in a program, if along every execution reaching N , the expression is available.
- Is $a * b$ available at program point D ?

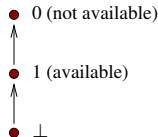


Available expressions

- An expression (like " $a * b$ ") is **available** along an execution if there is a point where the expression is evaluated and thereafter none of the constituent variables (like a and b) are written to.
- An expression is **available** at a point N in a program, if along every execution reaching N , the expression is available.
- Is $a * b$ available at program point D ? Yes.

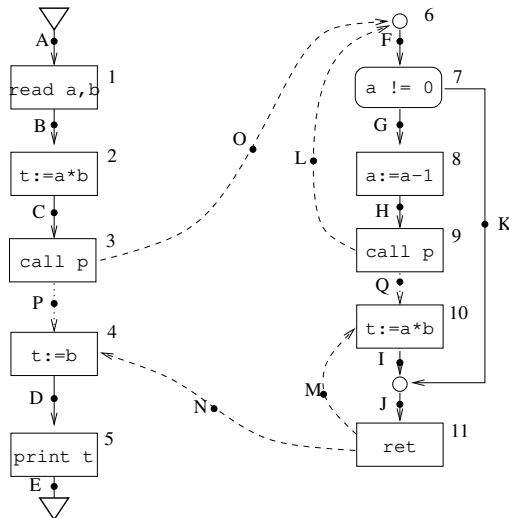


Available expressions analysis

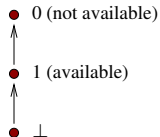


Lattice for Av-Exp analysis for $a * b$.

- “0” concretizes to the set $States \times \{A, NA\}$; while “1” concretizes to $States \times \{A\}$. “ $\perp$ ” concretizes to $\emptyset$.
- JOP of analysis says $a * b$ is not available at program point N .

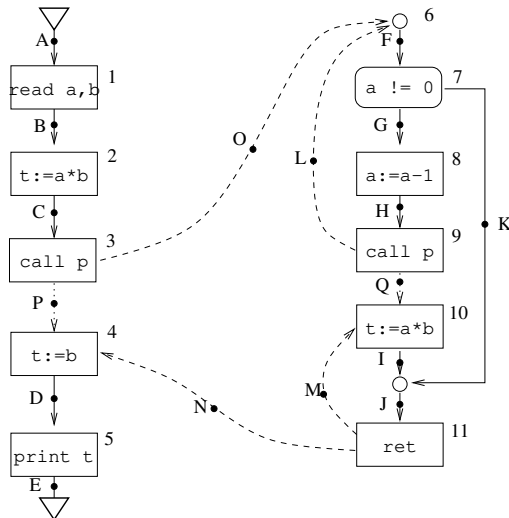


Available expressions analysis



Lattice for Av-Exp analysis for $a * b$.

- “0” concretizes to the set $States \times \{A, NA\}$; while “1” concretizes to $States \times \{A\}$. “ $\perp$ ” concretizes to $\emptyset$.
- JOP of analysis says $a * b$ is not available at program point N .
- JVP says it is available.

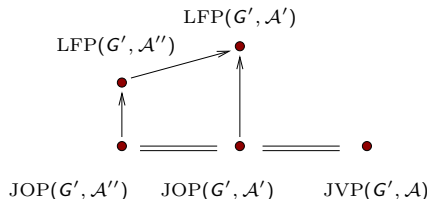


Computing JOP for abs int $\mathcal{A}'$

- We give two methods to **bound** the number of call-strings we need to consider, when underlying lattice $(D, \leq)$ is finite.
 - Give a bound on largest call-string needed.
 - Use “approximate” call-strings.

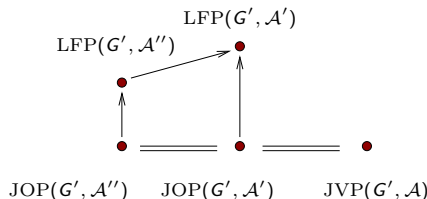
Bounded call-string method for finite underlying lattice D

- Possible to bound length of call-strings in Γ we need to consider.
- For a number l , we denote the set of call-strings (for the given program P) of length at most l , by Γ_l .
- Define a new analysis $\mathcal{A}''$ (M -bounded call-string analysis) in which call-string tables have entries only for Γ_M for a certain constant M , and transfer functions ignore entries for call-strings of length more than M .
- We will show that $\text{JOP}(G', \mathcal{A}'') = \text{JOP}(G', \mathcal{A}')$.



LFP of $\mathcal{A}''$ is more precise than LFP of $\mathcal{A}'$

- Consider any fixpoint V' (a vector of tables) of $\mathcal{A}'$.
- Truncate each entry of V' to (call-strings of) length M , to get V'' .
- Clearly V' dominates V'' .
- Further, observe that V'' is a **post-fixpoint** of the transfer functions for $\mathcal{A}''$.
- By Knaster-Tarski characterisation of LFP, we know that V'' dominates $\text{LFP}(\mathcal{A}'')$.



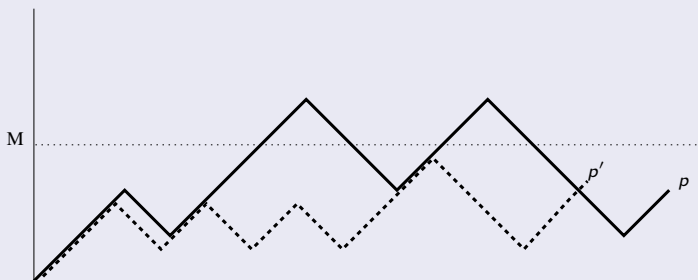
Sufficiency (or safety) of bound

Let k be the number of call sites in P .

Claim

For any path p in $IVP(r_1, N)$ with a prefix q such that $|cs(q)| > k|D|^2 = M$ there is a path p' in $IVP(r_1, N)$ with $|cs(q')| \leq M$ for each prefix q' of p' , and $f_p(d_0) = f_{p'}(d_0)$.

Paths with bounded call-strings



Proving claim

Claim

For any path p in $IVP(r_1, N)$ such that for some prefix q of p , $|cs(q)| > M = k|D|^2$, there is a path p' in $IVP_{\Gamma_M}(r_1, N)$ with $f_{p'}(d_0) = f_p(d_0)$.

- Sufficient to prove:

Subclaim

For any path p in $IVP(r_1, N)$ with a prefix q such that $|cs(q)| > M$, we can produce a **smaller** path p' in $IVP(r_1, N)$ with $f_{p'}(d_0) = f_p(d_0)$.

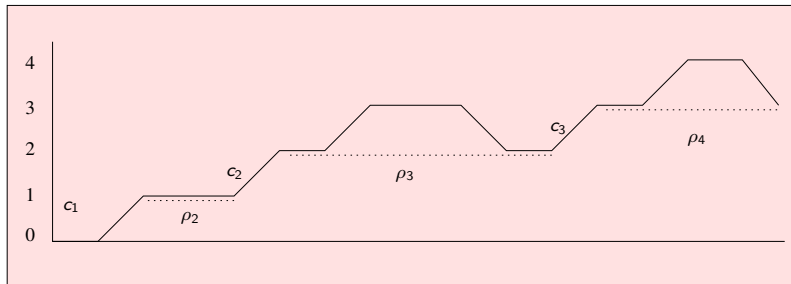
- ...since if $|p| \leq M$ then $p \in IVP_{\Gamma_M}$.

Proving subclaim: Path decomposition

A path ρ in $IVP(r_1, n)$ can be decomposed as

$$\rho_1 \parallel (c_1, r_{p_2}) \parallel \rho_2 \parallel (c_2, r_{p_3}) \parallel \sigma_3 \parallel \cdots \parallel (c_{j-1}, r_{p_j}) \parallel \rho_j.$$

where each ρ_i ($i < j$) is a **valid and complete** path from r_{p_i} to c_i , and ρ_j is a **valid and complete** path from r_{p_j} to n . Thus $c_1, \dots, c_{j-1}$ are the unfinished calls at the end of ρ .



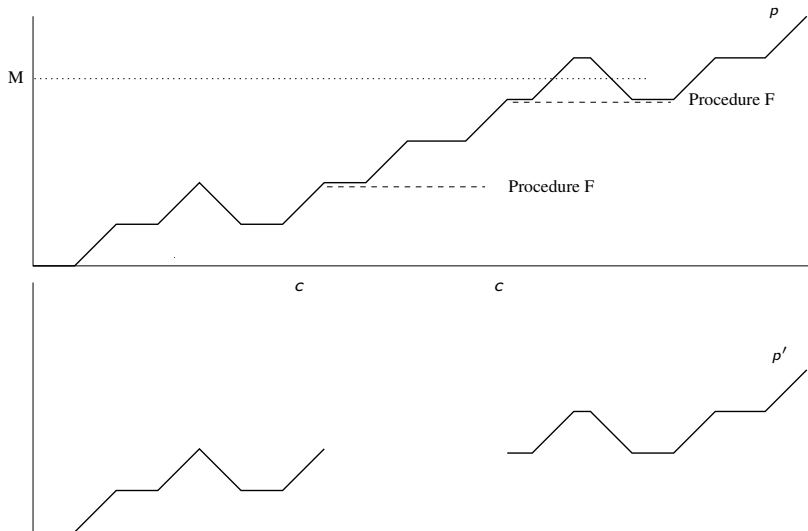
Proving subclaim

- Let p_0 be the first prefix of p where $|cs(p_0)| > M$.
- Let decomposition of p_0 be

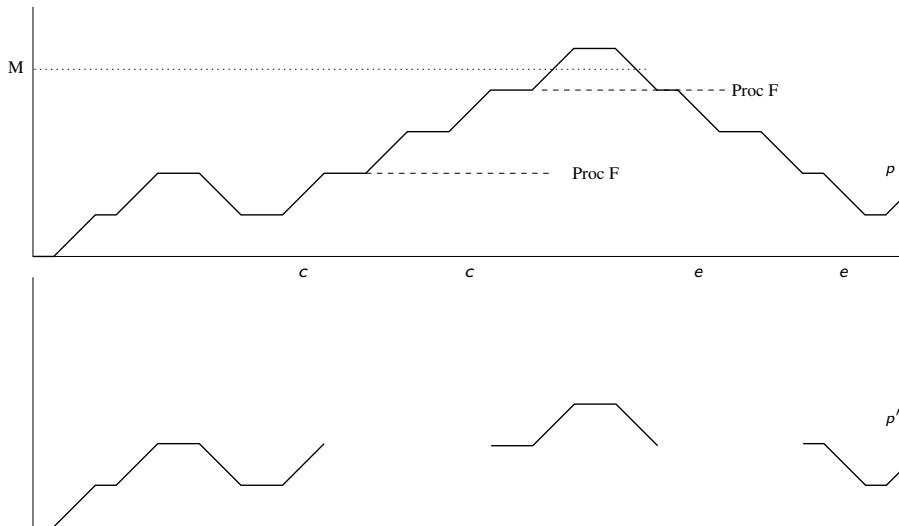
$$\rho_1 \|(c_1, r_{p_2})\| \rho_2 \|(c_2, r_{p_3})\| \sigma_3 \|\cdots\| (c_{j-1}, r_{p_j})\| \rho_j.$$

- Tag each unfinished-call c in p_0 by $(c, f_{q \cdot c}(d_0), f_{q \cdot cq'e}(d_0))$ where e is corresponding return of c in p .
- If no return for c in p tag with $(c, f_{q \cdot c}(d_0), \perp)$.
- Number of distinct such tags is $k \cdot |D|^2$.
- So there are two calls qc and $qcq'c$ with same tag values.

Proving subclaim – tag values are $\perp$



Proving subclaim – tag values are not $\perp$



Approximate (suffix) call-string method

We assume WLOG that the *main* procedure does **not** call itself.

Idea:

- Consider only call-strings of length $\leq l$, for some $1 \leq l$.
- For $l = 2$, call-strings can be of the form “ c_1 ” or “ c_1c_2 ” etc, but not “ $c_1c_2c_3$ ”. So each table ξ is now a finite table.
- Transfer functions for non-call/ret edges remain same.
- Transfer functions for call edge C : Shift each γ entry to $\gamma \cdot C$ if $|\gamma \cdot C| \leq l$; else shift it to $\gamma' \cdot C$ where γ is of the form $A \cdot \gamma'$ for some call A .

- Transfer functions for ret edge N :

Consider each γ of the form $\gamma' \cdot C$, where N corresponds to call edge C . Let the **first** call in γ be from some procedure p . If there exists a call to procedure p , then shift γ entry to $A \cdot \gamma'$, for **each** call A to procedure p .

If there are **no** calls to procedure p (in which case p must be *main*), shift γ entry to γ' .

From Sharir-Pnueli 1981, p136 of typed version

string. As long as the length of a call string is less than j , update it as in Section 4. However, if q is a call string of length j , then, when appending to it a call edge, discard the first component of q and add the new call block to its end. When appending a return edge, check if it matches the last call in q and, if it does, delete this call from q and add to its start all possible call blocks which call the procedure containing the first call in q . This approximation may be termed a call-string suffix approximation.

Correctness of $\mathcal{A}^\approx$

- Show that $\mathcal{A}^\approx$ is a consistent abstraction of the unbounded call-string analysis $\mathcal{A}'$.
- Use concretization function $\gamma^\approx$ and abstraction function $\alpha^\approx$.
- Argue they form a Galois connection
- Transfer functions over-approximate that of the unbounded call-strings analysis.

ϵ	O	OL	OLL	OL^3	OL^4	$\dots$
d_1	d_2	d_3	d_4	d_4	d_4	d_4

Concretization $\gamma^\approx$



ϵ	O	OL	LL
d_1	d_2	d_3	d_4

ϵ	O	OL	OLL	OL^3	OL^4	$\dots$
d_1	d_2	d_3	d_4	d_5	d_6	$\perp$

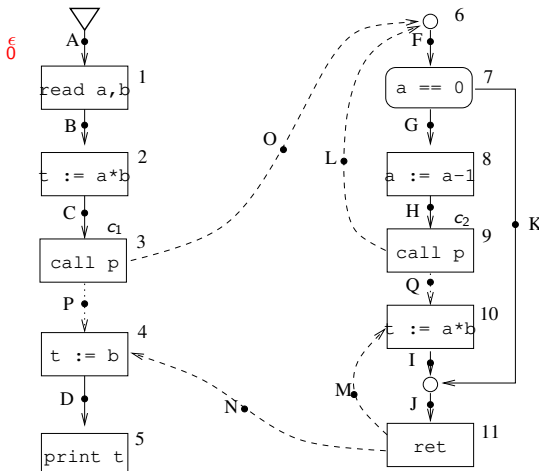


ϵ	O	OL	LL
d_1	d_2	d_3	$d_4 \sqcup d_5 \sqcup d_6$

Abstraction $\alpha^\approx$

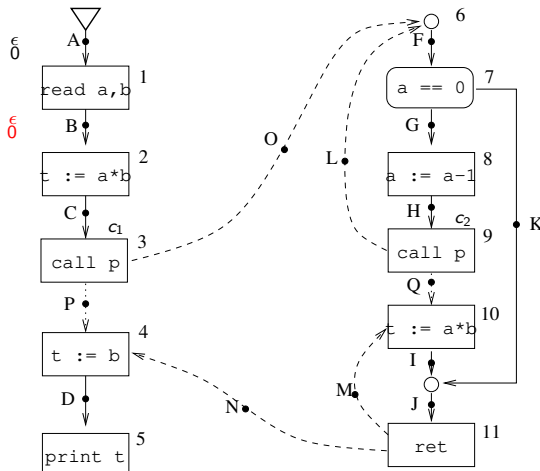
Exercise: approximate call-strings

Assume approximate call-string length of 2. Use Kildall's algo to compute the ξ table values for the example program. Start with initial value $d_0 = 0$.



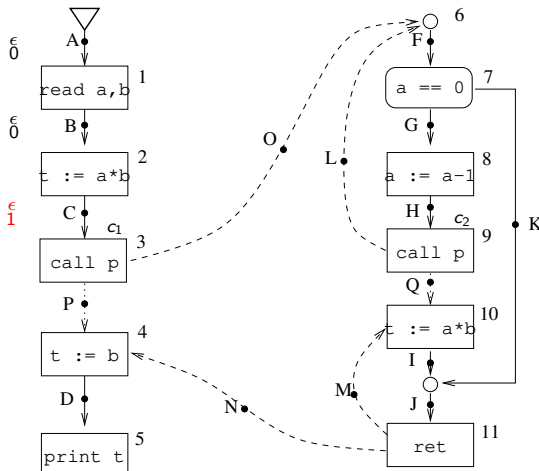
Exercise: approximate call-strings

Assume approximate call-string length of 2. Use Kildall's algo to compute the ξ table values for the example program. Start with initial value $d_0 = 0$.



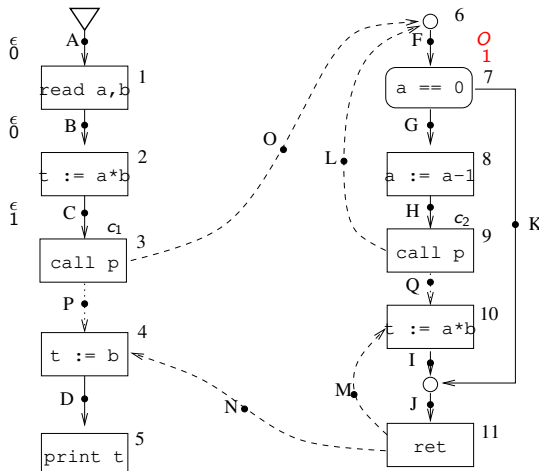
Exercise: approximate call-strings

Assume approximate call-string length of 2. Use Kildall's algo to compute the ξ table values for the example program. Start with initial value $d_0 = 0$.



Exercise: approximate call-strings

Assume approximate call-string length of 2. Use Kildall's algo to compute the ξ table values for the example program. Start with initial value $d_0 = 0$.



Exercise: approximate call-strings

Assume approximate call-string length of 2. Use Kildall's algo to compute the ξ table values for the example program. Start with initial value $d_0 = 0$.

