Interprocedural Analysis and Abstract Interpretation

Outline

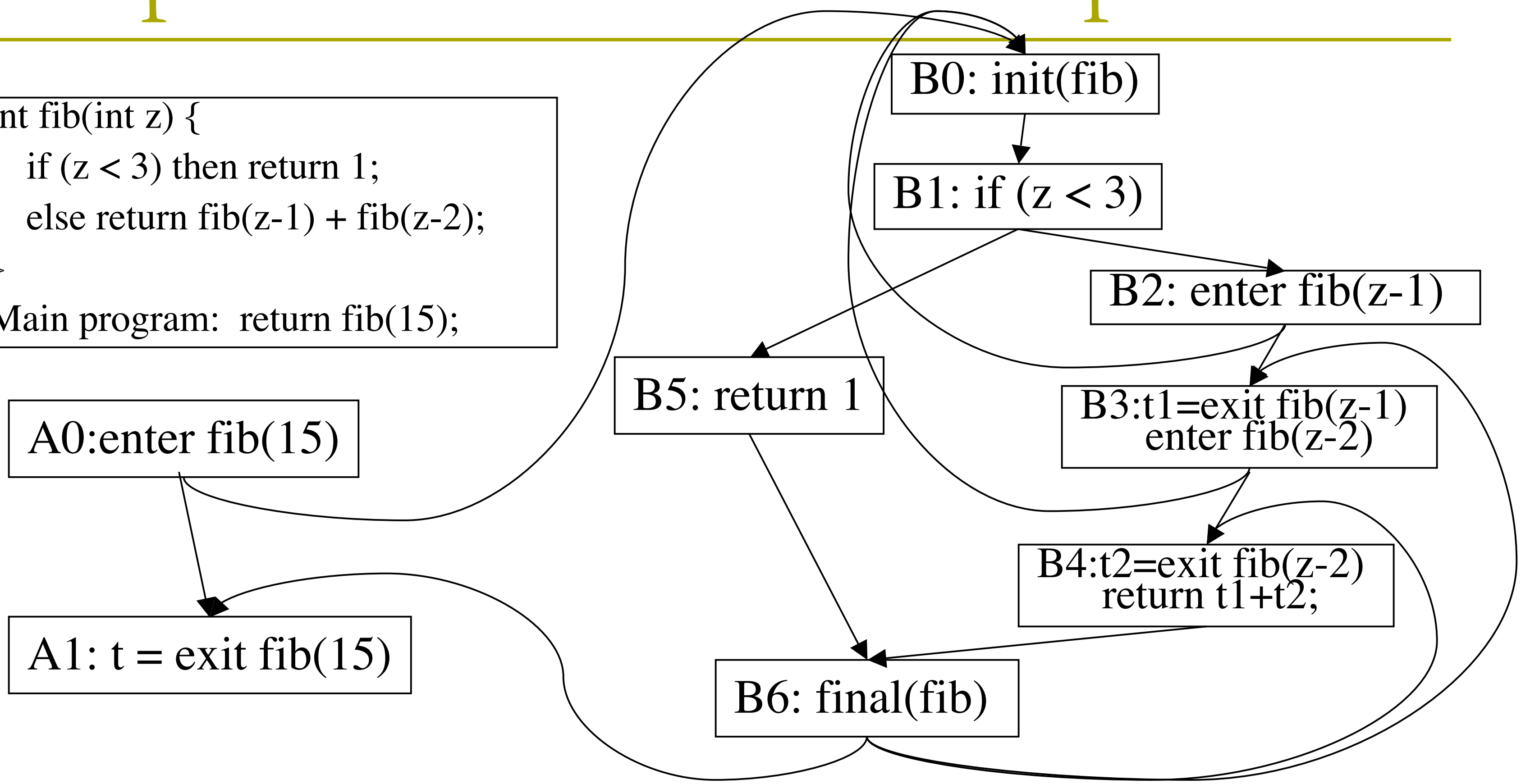
- Interprocedural analysis
  - control-flow graph
  - MVP: “Meet” over Valid Paths
  - Making context explicit
    - Context based on call-strings
    - Context based on assumption sets
- Abstract interpretation

Control-flow graph for a whole program

- At each function definition proc p(x)
  - Create two special CFG nodes:
    - init(p) and final(p)
  - Build CFG for the function body
    - Use init(p) as the function entry node
    - Connect every return node to final(p)
- At each function call to p(x) with
  - Split the original function call into two stmts
    - Enter p(x) (before making the call) and exit p(x) (after the call exits)
  - Connect enter p(x) ->init(p), final(p) -> exit p(x)
  - Connect enter p(x) -> exit p(x) to allow the flow of extra context info
- Three kinds of CFG edges
  - Intra-procedural: internal control-flow within a procedure
  - Procedure calls: from enter p(x) to init(p)
  - Procedure returns: from final(p) to exit p(x)

Interprocedural CFG Example

```
int fib(int z) {
    if (z < 3) then return 1;
    else return fib(z-1) + fib(z-2);
}
Main program:  return fib(15);
```



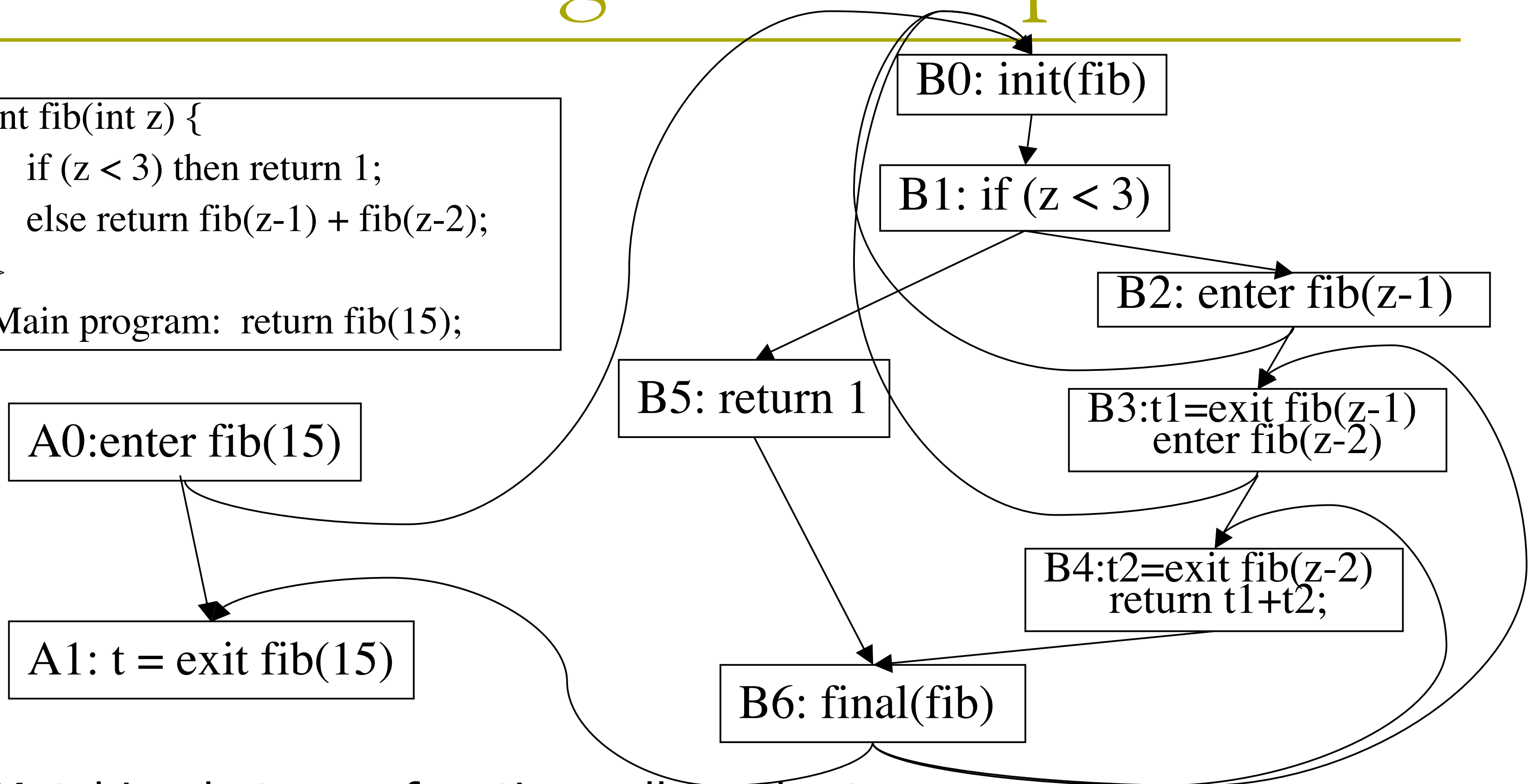
- Problem: matching between function calls and returns

Extending monotone frameworks

- Monotone frameworks consists of
  - A complete lattice $(L,\leq)$ that satisfies the Ascending Chain Condition
  - A set F of monotone transfer functions from L to L that
    - contains the identity function and
    - is closed under function composition
- Transfer functions for procedure definitions
  - For simplicity, both init(p) and final(p) have identity transfer functions
- Transfer functions for procedure calls
  - For procedure entry: assign values to formal parameters
  - For procedure exit: assign return values to outside

Problem: calling context upon return

```
int fib(int z) {
    if (z < 3) then return 1;
    else return fib(z-1) + fib(z-2);
}
Main program:  return fib(15);
```



- Matching between function calls and returns
  - Calculating solutions on non-existing paths could seriously detriment precision
    - E.g. enter fib(z-2) -> init(fib) -> … -> exit fib(z-1) -> …

MVP: “Meet” over Valid Paths

- Problem: matching procedure entries and exits (function calls and returns)
- A complete path must
  - Have proper nesting of procedure entries and exits
  - A procedure always return to the point immediately after it is called
- A valid path must
  - Start at the entry node of the main program
  - All the procedure exits match the corresponding entries
  - Some procedures may be entered but not yet exited
- The MVP solution
  - At each program point t, the solution for t is
    - MVP(t) = Λ { sol(p) : p is a valid path to t }

Making Context Explicit

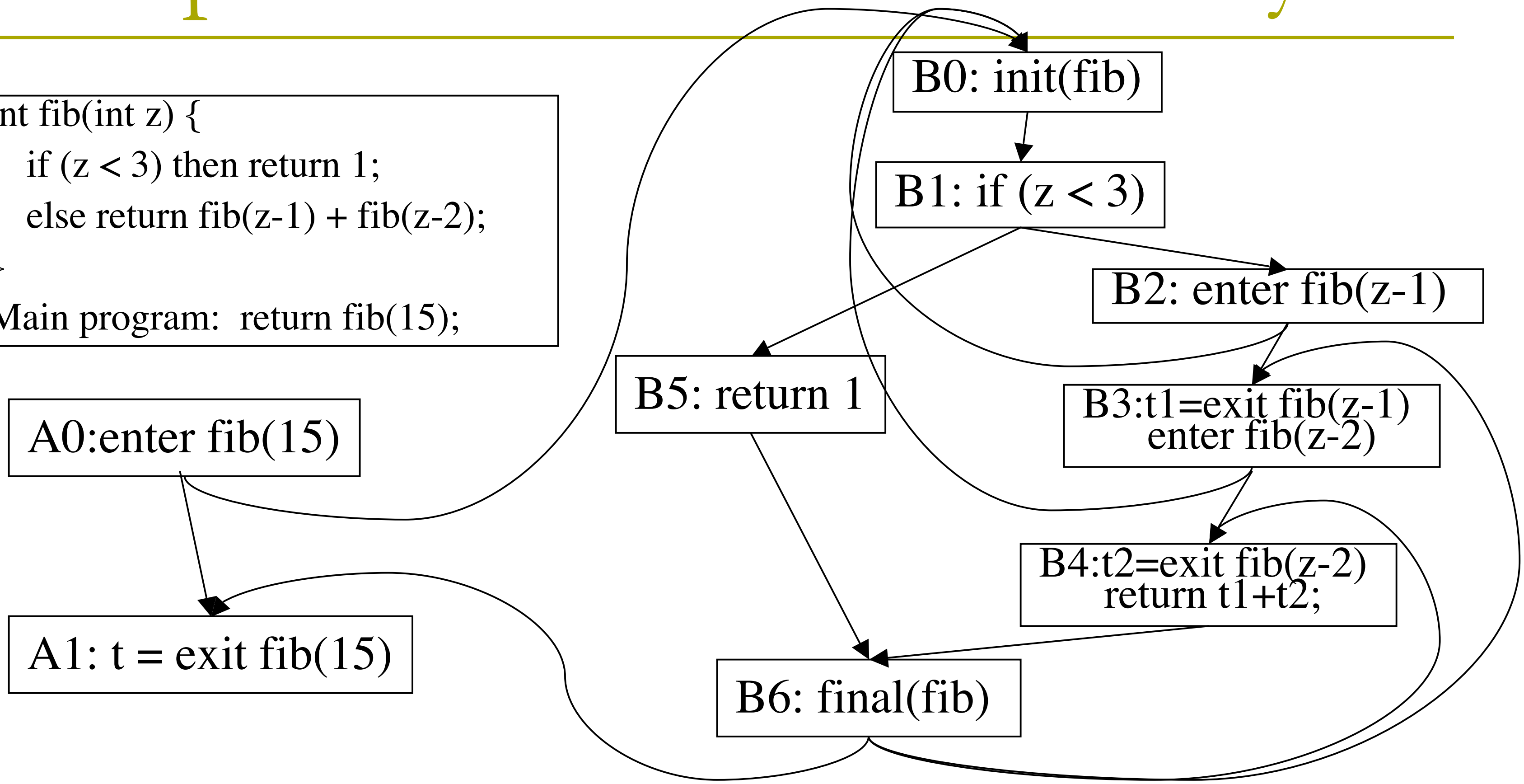
- Context sensitive analysis
  - Maintain separate solutions for different callers of a function
- Extending the monotone framework
  - Starting point (context-insensitive)
    - A complete lattice (L,≤) that satisfies the Ascending Chain Condition
      - L = Power(D) where D is the domain of each solution
    - A set F of monotone transfer functions from L to L
  - Extension
    - L = Power(D * C), where C includes all calling contexts
    - F = L -> L, a separate sub-solution is calculated for each calling context
      - F (procedure entry) : attach caller info. to incoming solution
      - F (procedure exit): match caller info, eliminate solution for invalid paths

Different Kinds of Context

- Call strings --- contexts based on control flow
  - Remember a list of procedure calls leading to the current program point
    - Call strings of unbounded length --- remember all the preceding calls
    - Call strings of bounded length (k) --- remember only the last k calls
- Assumption sets --- contexts based on data flow
  - Assumption sets
    - Use the solution before entering proc p(x) as calling context (e.g., each context makes distinct presumptions about values of function parameters)
  - Large vs. small assumption sets
    - How large is the context: use the entire solution or pick a single constraint from the solution

Example Context-sensitive Analysis

```
int fib(int z) {
    if (z < 3) then return 1;
    else return fib(z-1) + fib(z-2);
}
Main program:  return fib(15);
```



- Range analysis: for each variable reference x, is its value >= or <= a constant value? (i.e, x >= x1; z<=n2)?

Example Range Analysis

Variables: x,z, t1, t2, fib, t; Contexts: A0, B2, B3,none;

Domain: Variables * (<=n, =n, >=n,?,any)

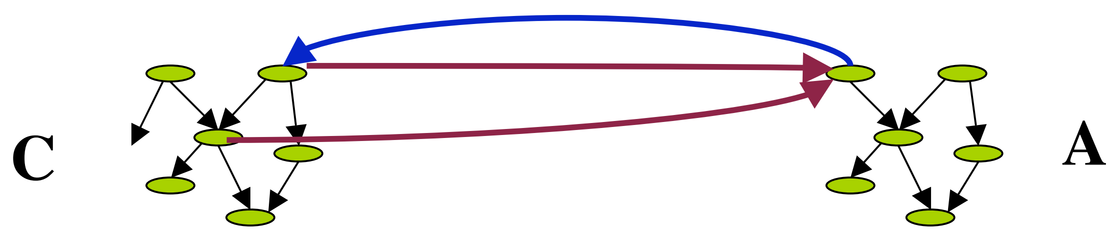
A0	(none)	(none)	(none)	(none)
B0	(none, z=?)	(A0,z=15) (B2/B3, z=?)	(A0,z=15)(B2,z>=2) (B3,z>=1)	(A0,z=15)(B2,z>=2) (B3,z>=1)
B1	(none, z=?)	(A0,z=15) (B2/B3, z=?)	(A0,z=15)(B2,z>=2) (B3,z>=1)	(A0,z=15)(B2,z>=2) (B3,z>=1)
B2	(none, z=?)	(A0,z=15) (B2/B3, z>=3)	(A0,z=15)(B2/B3,z>=3)	(A0,z=15)(B2/B3,z>=3)
B3	(none, z/t1=?)	(A0,z=15,t1=?) (B2/B3,z>=3,t1=?)	(A0,z=15,t1=1) (B2/B3,z>=3,t1=1)	(A0,z=15,t1>=1) (B2/B3,z>=3,t1>=1)
B4	(none, z/t1/t2=?)	(A0,z=15,t1/t2=?)(B2/B3,z>=3,t1/t2=?)	(A0,z=15,t1/t2=1)(B2/B3,z>=3,t1/t2=1)	(A0,z=15,t1/t2>=1)(B2/B3,z>=3,t1/t2>=1)
B5	(none, z=?)	(B2/B3,z<=2)	(B2,z=2) (B3,z<=2)	(B2,z=2) (B3,z<=2)
B6	(none,z/fib=?)	(A0,z=15,fib=?)(B2/B3,z=any,fib=1)	(A0,z=15,fib>=1)(B2/B3,z=any,fib>=1)	(A0,z=15,fib>=1)(B2/B3,z=any,fib>=1)
A1	(none,t=?)	(none,t=?)	(none,t >=1)	(none,t>=1)

Foundations of Abstract Interpretation

- Definition from Wikipedia
  - **abstract interpretation** is a theory of sound approximation of the semantics of computer programs. It can be viewed as a partial execution of a computer program without performing all the calculations.
- Outline
  - Monotone frameworks
    - A complete lattice (L,≤) that satisfies the Ascending Chain Condition
    - A set F of monotone transfer functions from L to L that
      - contains the identity function and
      - is closed under function composition
  - Galois connections, closures,and Moore families
  - Soundness and completeness of operations on abstract data
  - Soundness and completeness of execution trace computation

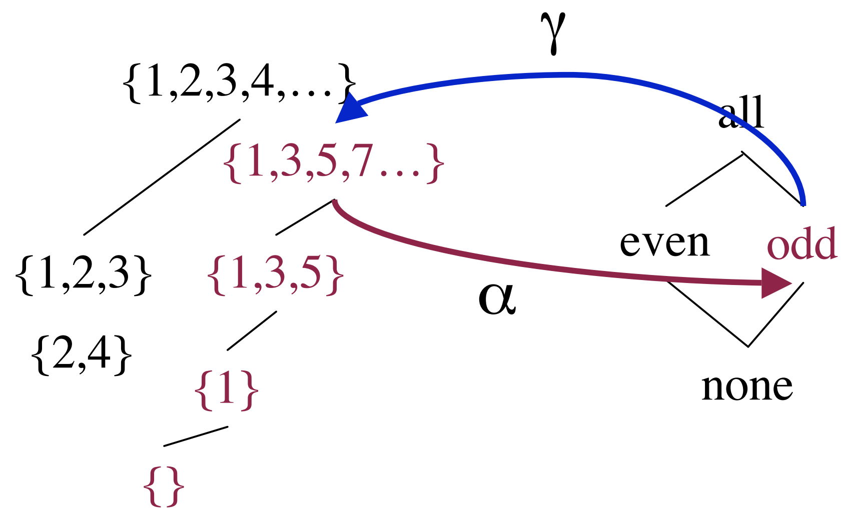
Galois Connections

- Two complete lattices
  - C: the “concrete” (execution) data
    - The execution of the entire program
    - Infinite and impossible to model precisely
  - A: the “abstract” (execution) data
    - Properties (abstractions) of the “concrete” data
    - The solution space (domain) of static program analysis
- For complete lattices C and A, a Galois connection is
  - A pair of monotonic functions, α : C->A, γ : A -> C
  - For all $a \in A$ and $c \in C$: $c \le \gamma(\alpha(c))$ and $\alpha(\gamma(a)) \le a$
  - Is Written as $C<\alpha,\gamma>A$



Galois Connections (2)

- γ and α are inverse maps of each other's image
  - For all $c \in \gamma(A), c=\gamma(\alpha(c))$; for all $a \in \alpha(C), a=\alpha(\gamma(a))$
  - The maps α are "homomorphism" mappings between C and A
- Galois connections are closed under
  - Composition, product, and so on
- Each instruction performs an action f: C->C
  - Can use α and γ to define an abstract transfer function f#: A->A for each f: C->C

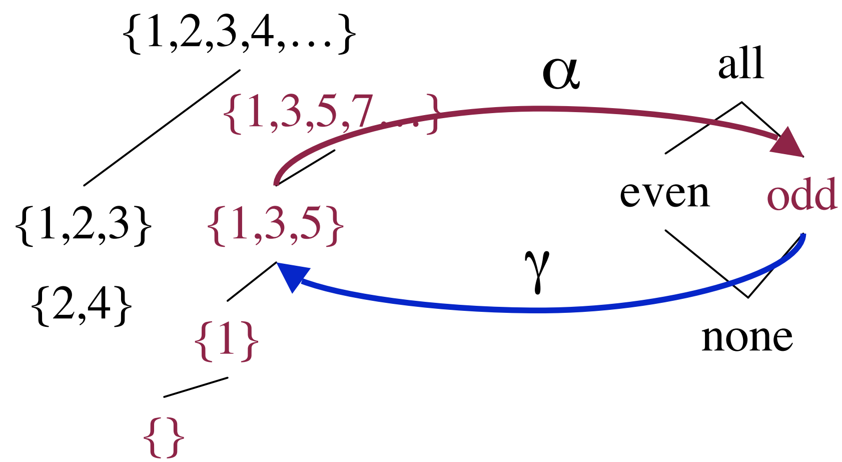


Closure Maps

- For $C<\alpha,\gamma>A$, it is common that $A \subseteq C$. This means A embeds into C as a sub-lattice
  - A's elements name distinguished sets in C
- A closure map defines the embedding of A within C.

Definition: ρ:C->C is a *closure map* if it is

- *Monotonic:* $\forall\, c1,c2 \in C$, $c1 \le c2 \Rightarrow \rho(c1) \le \rho(c2)$;
- *extensive:* $\forall\, c \in C$, $c \le \rho(c)$;
- *idempotent:* $\forall\, c \in C$, $\rho(\rho(c)) = \rho(c)$ (i.e. $\rho * \rho = \rho$)



1) Every Galois connection, $C<\alpha,\gamma>A$ defines a closure map $\alpha \bullet \gamma$;
2) Every closure map, ρ:C->C, defines the Galois connection, $C<\rho,id>\rho(C)$.

Moore Families

- Given C, can we define a closure map on it by choosing some elements of C?
  - *Yes*, if the elements we select are closed under greatest-lower-bounds (meet) operation
  - That is, the new set of elements forms a complete lattice
- **Definition:** $M \subseteq C$ is a *Moore family* iff for all $S \subseteq M$, $(\wedge S) \in M$.
  - We can define a closure map as $\rho(c) = \wedge\{c' \in M \mid c \leq c'\}$.
  - That is, we map each element in C to the closest abstraction (approximation) in M
- For each closure map, ρ:C->C, its image, $\rho(C)$, is a Moore family.

Given C, we can define an abstract interpretation by selecting some $M \subseteq C$ that is a Moore family

Closed Binary Relations

- Often the solution of an analysis is a power set of its domain
  - The Galois connection can be written as Power(D)<α,γ>A
- Given unordered set D and complete lattice A, it is natural to relate the elements in D to those in A by **a binary relation, R ⊆ D * A, s.t.**
  - **(d,a) ∈ R** (or **d R a**, **d |=$_R$ a**) means "d *has property* a".
  - **Example:** D=Int, A={none,neg,pos,zero,nonneg,nonpos,any}.
    - Then 2 R nonneg, 2 R pos, and 2 R any.
- The adjoint function, γ : A->Power(D),can be defined as
  - γ(a) = {d ∈ D | d R a}. E.g., γ (nonneg)={0,1,2,...}.
  - If R defines a Galois collection, then γ(A) defines a Moore family.
- **Proposition: R⊆D*A** defines a Galois connection between (Power(D), A) iff
  - R is *U-closed*: **c R a** and **a ≤ a'** imply **c R a'**;
  - R is *G-closed*: **c R ^ {a | c R a }**

Concrete and Abstract Operations

- Now that we know how to model a solution space via abstraction function α : C -> A,
  - We must model concrete computation steps, f:C->C, by abstract computation steps, f#:A -> A.
- **Example:** we have concrete domain, Nat, and concrete operation, succ: Nat -> Nat, defined as succ(n)=n+1.
  - abstract domain, Parity = {any, even, odd, none}.
  - abstract operation, succ#:Parity -> Parity, defined as

    succ#(even)=odd, succ#(odd)=even, succ#(any)=any, succ#(none)=none,
  - succ# must be consistent (sound) with respect to succ:

    if n Rn a, then succ(n) Rn succ#(a),
    - where Rn $\subseteq$ Nat * Parity relates numbers to their parities (e.g., 2 Rn even, 5 Rn odd, etc.).

Sound Approximation

- Given
  - Galois connection $C<\alpha,\gamma>A$ and
  - functions f : C->C and f#:A-> A,

f# is a *sound approximation* of f iff

  - For all $c \in C$, $\alpha(f(c)) \leq f\#(\alpha(c))$
  - For all $a \in A$, $f(\gamma(a)) \leq \gamma(f\#(a))$
- That is, α defines a "semi-homomorphism" with respect to f and f#

$$\begin{tikzcd} \mathbf{c} \arrow[r, "\alpha"] \arrow[d, "\mathbf{f}"] & \alpha(\mathbf{c}) \arrow[d, "\mathbf{f\#}"] \\ \mathbf{f(c)} \arrow[r, "\alpha"] & \alpha(\mathbf{f(c)}) \leq \mathbf{f\#}(\alpha(\mathbf{c})) \end{tikzcd}$$

Sound Approximation Example

- Given
  - Galois connection Power(Nat)$<\alpha,\gamma>$Parity and
  - Concrete transfer function succ : Nat->Nat, $succ(S) = \{ n + 1 \mid n \in S \}$
  - Abstract transfer function succ#: Parity -> Parity,

    succ#(even)=odd,   succ#(odd)=even

    succ#(any)=any,   succ#(none)=none
- succ# is a *sound approximation* of succ
  - For all $c \in$ Nat, $\alpha(succ(c)) = succ\#(\alpha(c))$

$$
\begin{tikzcd}
\{2,6\} \arrow[r, "\alpha"] \arrow[d, "succ"] & even \arrow[d, "succ\#"] \\
\{3,7\} \arrow[r, "\alpha"] & odd
\end{tikzcd}
$$

Synthesizing f# from f

- Given C<α,γ>A, and function f : C->C, the most precise f#:A->A that is sound with respect to f is
  - f# best (a) = α (f (γ (a)))
- **Proposition:** f# is sound with respect to f iff
  - For all a $\in$ A, f# best(a) $\leq$ f#(a)
  - Of course, f#best has a *mathematical* definition—not an algorithmic one—f#best might not be finitely computable!
- **Parity example continued:**
  - succ#best(even)= α (succ (γ (even))) = α (succ {2n | n≥0 })) = α ({2n+1 | n≥0}) = odd
  - Question: what about other operators on Nat, e.g., *, / ?

Completeness of Approximation(skip)

Given $C<\alpha,\gamma>A$, and function f : C->C,

- Function f#: A->A is sound with respect to f iff
  - For all $c \in C$, $\alpha(f(c)) \leq f\#(\alpha(c))$
  - For all $a \in A$, $f(\gamma(a)) \leq \gamma(f\#(a))$
- Function f#: A->A is *forwards(γ) complete* with respect to f iff
  - For all $a \in A$, $f(\gamma(a)) = \gamma(f\#(a))$
  - That is, $\gamma(A)$ is closed under f : $f(\gamma(A)) \subseteq \gamma(A)$
- Function f#: A->A is *backwards(α) complete* with respect to f iff
  - For all $c \in C$, $\alpha(f(c)) = f\#(\alpha(c))$
  - That is, α partitions C into equivalence classes: $\alpha(c)= \alpha(c')$ implies $\alpha(f(c))=\alpha(f(c'))$
- For an f# to be (forwards or backwards) complete, it must equal $f\#best=\alpha(f(\gamma(a)))$
  - The structure of $C<\alpha,\gamma>A$ and f: C->C determines whether f# is complete.

Transfer Functions and Computation steps

- Each program transition from program point pi to pj has an associated *transfer function*, fij:C->C (or f#ij:A-> A), which describes the associated computation.
  - This defines a computation step of the form, (pi,s) -> (pj,fij(s))
- **Example:**
  - Assignment p0:x=x+1;p1:⋯ has the transfer function

    f01(<…x:n…>) = <…x:n+1…>
  - For multiple transitions in conditionals, attach a transfer function to each possible transition (branch) to “filter” the data that arrives at a program point.

    e.g. p0: cases x≤y: p1:y=y-x;

    y≤x: p2:x=x-y; end
    - fp1(s) = if s[x] ≤ s[y] then s else bot; (filter out s unless s[x] ≤ s[y])
    - fp2(s) = if s[y] ≤ s[x] then s else bot; (filter out s unless s[y] ≤ s[x])

Execution Traces

- An *execution trace* is a (possibly infinite) sequence, (p0,s0)->(p1,s1)->…->(pj,sj)-> …,s.t.
  - for all i≥0: (pi,si) -> $p_{succ(i)}, f_{i,succ(i)}(s_i)$
  - No si equals bot

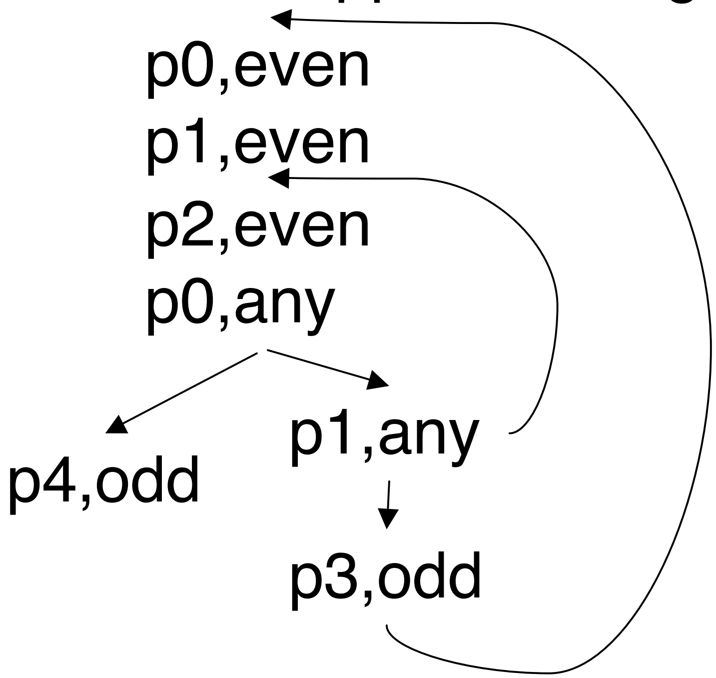
```
P0: while (x != 1) {
P1:  if Even(x)
P2:     x = x div2;
P3:  else
        x = 3*x + 1;
}
P5: exit;
```

Two concrete traces ((pi,v) means (pi,x=v)):

p0,4	p0,6
p1,4	p1,6
p2,4	p2,6
p0,2	p0,3
p1,2	p1,3
p2,2	p2,3
p0,1	p0,10
p4,1	p4,1
	…

Using Approximation to build abstract traces

Abstract over approximating trace:



1. Each concrete transition is generated by an fij;
2. Each abstract transition is generated by the corresponding f#ij.

- Each concrete transition, (pi,s)-> (pj,fij(s)), is reproduced by a corresponding abstract transition, (pi,a)->(pj,f#ij(a)), where s$\in\gamma$(a)
- The traces embedded in the abstract trace tree "cover" (*simulate*) the concrete traces

Shape Analysis

- Goal
  - To obtain a finite representation of the memory storage
- The analysis result can be used for
  - Detection of pointer aliasing
  - Detection of sharing between structures
  - Software development tools
    - Detection of pointer errors, e.g. dereferences of nil-pointers
  - Program verification
    - E.g.,reverse transforms a non-cyclic list to a non-cyclic list

The Concrete Solution Space

- Model the memory (stack and heap)
  - Storage of local variables

    Stack = Var -> (Value ∪ Loc)

    Map each local variable into a value or a unique location
  - The heap storage

    Heap = (Loc * Sel) -> (Value ∪ Loc)

    Map pairs of locations and selectors to values or locations
- Model the operational semantics of programs
  - Program state: State = ProgramPoint * Stack * Heap

    Example: (p1, (x:3,y:Ly), ((Ly,val):5)) is a program state
  - Each statement modifies Stack and Heap of the previous state
    - Stmt: State -> State

Building Abstract Domains

- Given an unordered set, D, of concrete data values, we might ask,
  - *"What are the properties about* D *that I wish to calculate?*
  - *Can I relate these properties* a $\in$ A, *to elements* d $\in$ D *via a UG-closed binary relation,* R: D*A *?*
- Given a set, A, and a binary relation, R: D * A
  - Define γ: A->Power(D) as γ(a) = {d $\in$ D | d R a}
  - Define partial ordering on A: a $\leq$ a' iff γ(a) $\leq$ γ(a')
    - If there are distinct a and a' such that γ(a)=γ(a'), then merge them to force U-closure
  - Ensure that γ(A) is a Moore family by adding greatest-lower-bound elements to A as needed.
    - This forces G-closure
  - Use the existing machinery to define the Galois connection between Power(D) and A

Abstracting the Program State

- Build a binary relation, R_d: Data*AbsData
  - Rv: Value -> AbsValue ; Rl: Loc -> AbsLoc
  - May ignore the values of non-pointer variables.
- Build induced Galois connection, Power(Data)<α,γ>AbsData, we can
  - Build Galois connections that abstract the concrete data

    <xi : vi> Rs <xi : ai> iff vi Rd ai

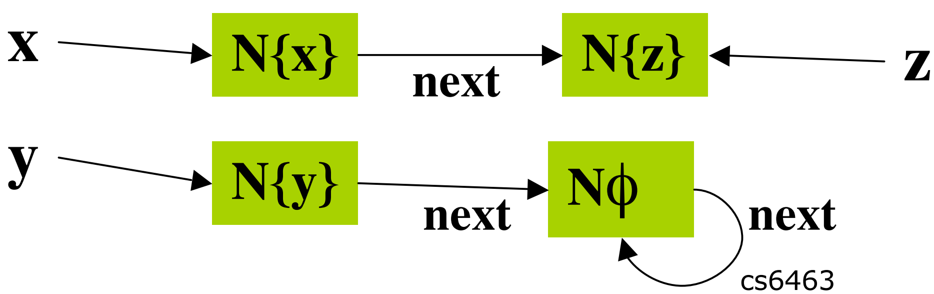
    Example: <x:3, y:4> Rs <x:any, y:any>
  - A program point is abstracted to itself: p Rp p,

    the abstract domain of program points is ProgramPoint ∪ {top, bot} (to make it a complete lattice)
  - Finally, we can relate each concrete state to an abstract one:

    (p,s) Rs (p’,s’) iff p = p’ and s Rs s’

Shape Graphs

- Shape analysis uses a shape graph to abstract the memory storage
  - Graph nodes denote a finite number of abstract locations:
    - Aloc = {Nx I Nx is pointed to by a set of local variables} $\cup$ Nϕ
      - Nx : the node represents all concrete Locations referred to by variables in x
      - Nϕ : abstract summary location (all the other locations)
    - Each graph node abstracts a distinctive set of concrete Locations
      - If variables x and y may be aliased, they must share a single graph node
  - A graph edge sel connect nodes n1 and n2 if n2 is pointed to by n1.sel

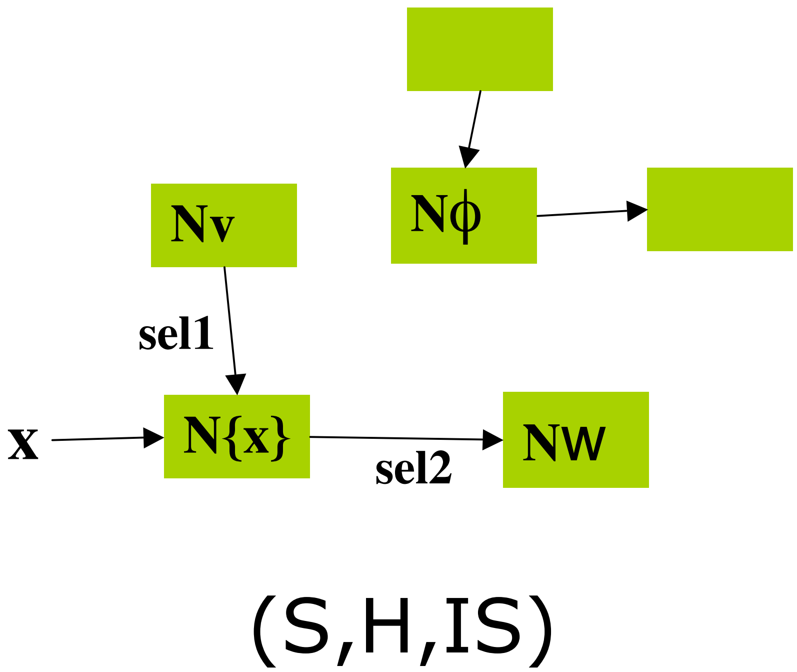


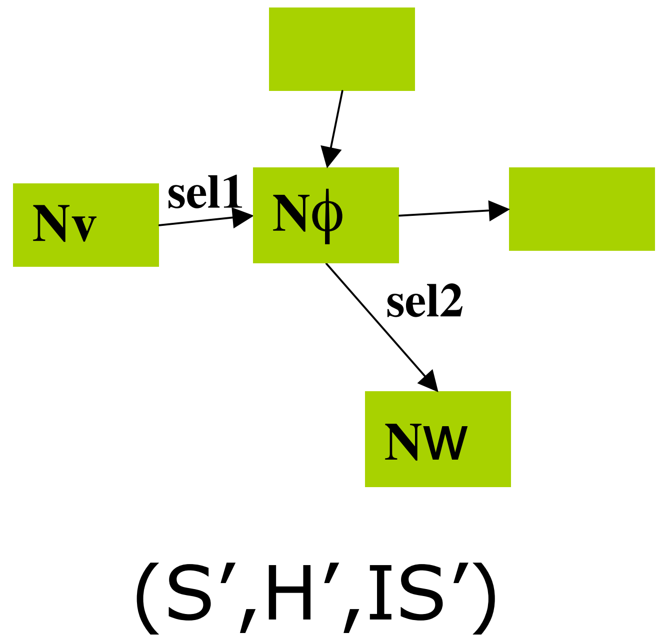
Abstraction of Program States

- Abstraction of memory storage
  - Abstract Stack
    AbsStack = Var -> ALoc
    Map each pointer variable into a unique abstract location (a shape graph node)
  - Abstract heap
    AbsHeap = (ALoc * Sel) -> (ALoc)
    Mapping pairs of abs locations and selectors to abs locations
  - Sharing information
    - IS : ALoc -> { yes, no}
      For each abstract location in the shape graph, is it shared by pointers in the heap?
    - If IS(Nx) = yes, then Nx must have an incoming edge from Nϕ or have more than one incoming edges
- Transfer functions: P(AbsState) -> P(AbsState)
  - Program state: AbsState=ProgramPoint * AbsStack * AbsHeap * IS
  - Each statement modifies mappings in the previous state

Transfer functions(1)

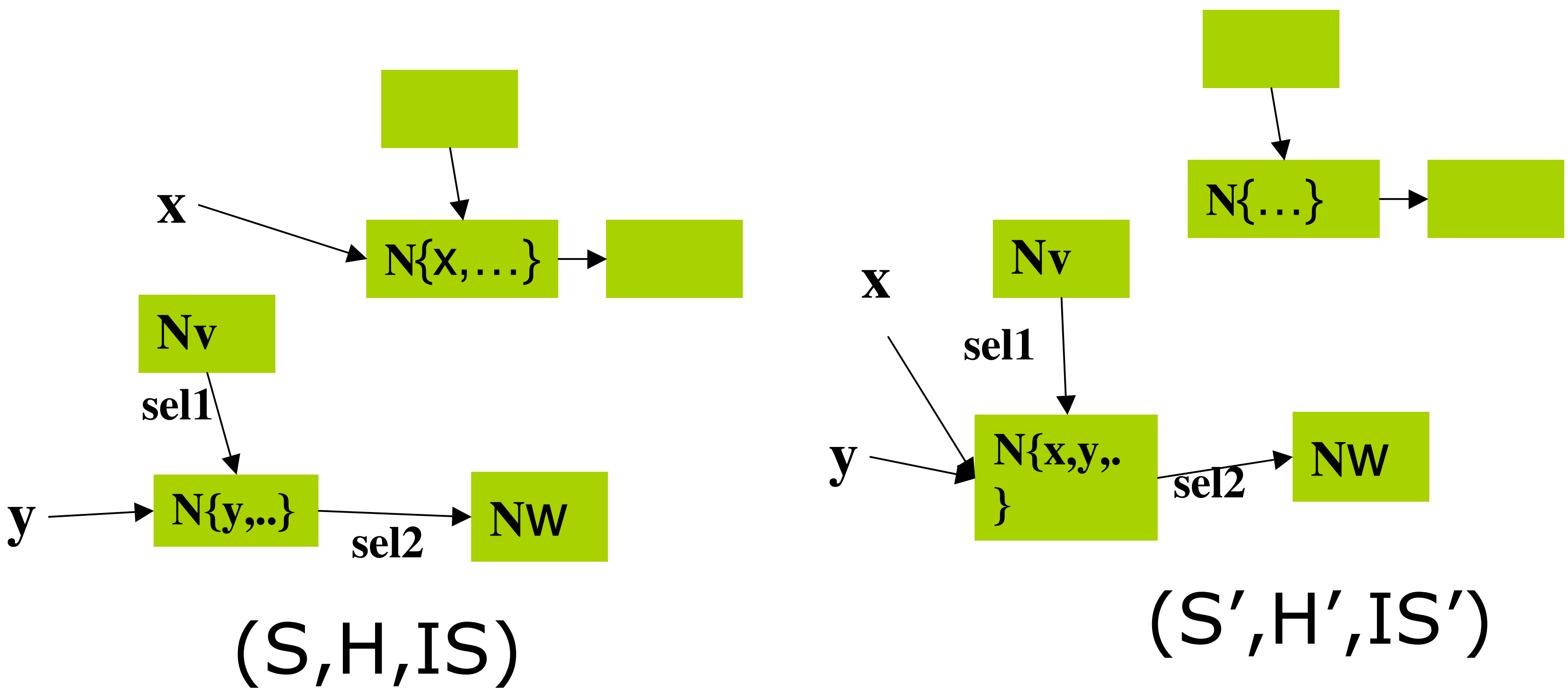
- x = nil
  - F (S,H,IS) = (S′,H′,IS′) where (S′,H′,IS′) is obtained from (S,H,IS) by
    - Removing x from all mappings (killing all previous info. about x)
    - Merging all Nϕ nodes





Transfer functions(2)

- x = y
  - F (S,H,IS) = (S’,H’,IS’) where
    - (S’,H’,IS’) is obtained by modifying mappings for x to be identical to those for y



Transfer functions(3)

- x = y.sel
  - Remove the old binding for x
  - Establish a new binding for x to be the same as y.sel
    - If there is no abstract location defined for y
      - Error: dereference a null pointer
    - If there is an abstract location Ny s.t. S[y] = Ny, but there is no abstract location for (Ny,sel)
      - Error dereference a non-existing field
    - If there exist abstract locations Ny and Nz s.t. S[y] = Ny and H[Ny,sel] = Nz.
      - Modify the mappings so that x points to Nz
      - If Nz = Nϕ, create a new node N{x} for x --- may need to create multiple shape graphs to cover different cases
- Other transfer functions
  - E.g. x.sel = y; x.sel = nil; allocate(x);