

# Principles of Program Analysis



An overview of approaches

# The Nature of static analysis

## --- approximation

---

- Static program analysis --- predict the dynamic behavior of programs without running them
  - Example: at each program execution step, what is the value of each variable?

```
int x, y, z;  
read(&x);  
if (x>0) { y=x; z = 1}  
else { y= - x; z = 2}
```
  - The question cannot be answered precisely b/c the program input is unknown
    - We don't know the value of x, and therefore cannot predict which branch will be taken (whether the value of x is greater than 0)
    - However, we can predict all the possible values for z and that y is  $\geq 0$  at the end of code.
  - Program analysis approach: tries to give approximate answers; tries to prove properties of program entities (variables, functions, types)

# The Nature of Approximation

## --- may and must analysis

---

- Since the behavior of programs cannot be predicted precisely, there are two ways to approximate
  - Over approximation: what *may* happen when all possible inputs are considered?
    - The answer is a superset of what happens at runtime
  - Under approximation: what *must* always happen in spite of different inputs?
    - The answer is a subset of what happens at runtime
- What approximation to use depends on what the results will be used for
  - Should always err on the safe side
    - Example: if we want to remove all useless evaluations in the program, should we find evaluations that may or must be useless?
- The relation between may and must analysis
  - Find all evaluations that are always useless (must analysis)  
<=> find all evaluations that may be useful (may analysis)

# The Precision of Approximation --- How input sensitive is the analysis?

---

- Flow sensitivity: Is solution sensitive to the control flow within a function?
  - Flow-insensitive analysis
    - Example: what variables may be accessed by a code?
    - Solution: find all the variables that appear in the code
  - Flow sensitive analysis
    - Example: what values a variable may have at each program point
    - A different solution must be found for each program point
- Context sensitivity: Is solution sensitive to the calling context of a function?
  - Context-insensitive: a single solution is computed for each function, no matter who calls the function
  - Context-sensitive: different solutions are computed for different chains of callers
- Path sensitivity? Is solution sensitive to different execution paths of a program?
  - Path sensitive: different solutions are computed for different paths from program entry to each statement

# Scopes of Program Analysis

---

- What code are examined to find the solution?
  - Local analysis
    - Operate on a straight-line sequence of statements (a basic block)
    - Often used as basis for more advanced analysis approaches
  - Regional analysis
    - Operate on code with limited control flow, e.g., loops, conditionals
    - Useful for special-purpose optimizations (e.g., loop optimizations)
  - Global (intra-procedural) analysis
    - Operate on a single procedure/subroutine/function
    - Required by most flow-sensitive analysis problems
  - Whole-program (inter-procedural) analysis
    - Operate on an entire program (all sources must be available)
    - Required by context and path sensitive analysis

# Common Approaches to Program Analysis

---

- A family of techniques
  - Data flow analysis: operate on control-flow graph
    - Define a set of data to evaluate at entry and exit of each basic block
    - evaluate the flow of data between pred/succ basic blocks
  - Constraint based analysis
    - For each program entity to be analyzed, define a set of constraints involving information of interest
    - Solve the constraint system via mathematical approaches
  - Abstract interpretation
    - Define a set of data to evaluate at each program point; Map each statement/construct to a finite sequence of semantic actions
    - Statically interpret each instruction in program execution order
  - Type and effect systems
    - Categorize different properties into a collection of types/groups
    - Infer the type/group of each program entity from how it is used
- Techniques differ in algorithmic methods, semantic foundations, language paradigms

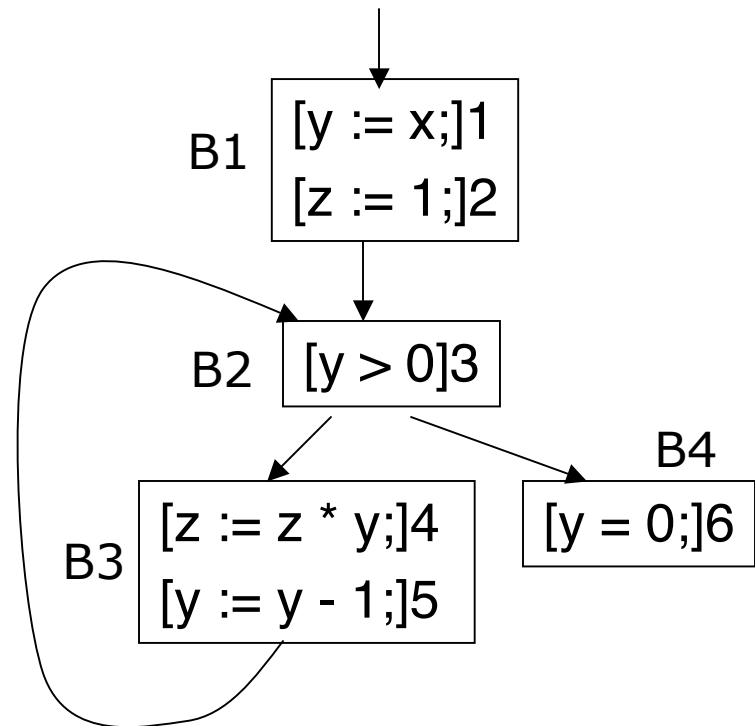
# Example Dataflow Analysis

## Reaching Definitions

Example program with labels

```
[y := x;]1
[z := 1;]2
while [y > 0]3 {
  [z := z * y;]4
  [y := y - 1;]5
}
[y = 0;]6
```

Control-flow graph

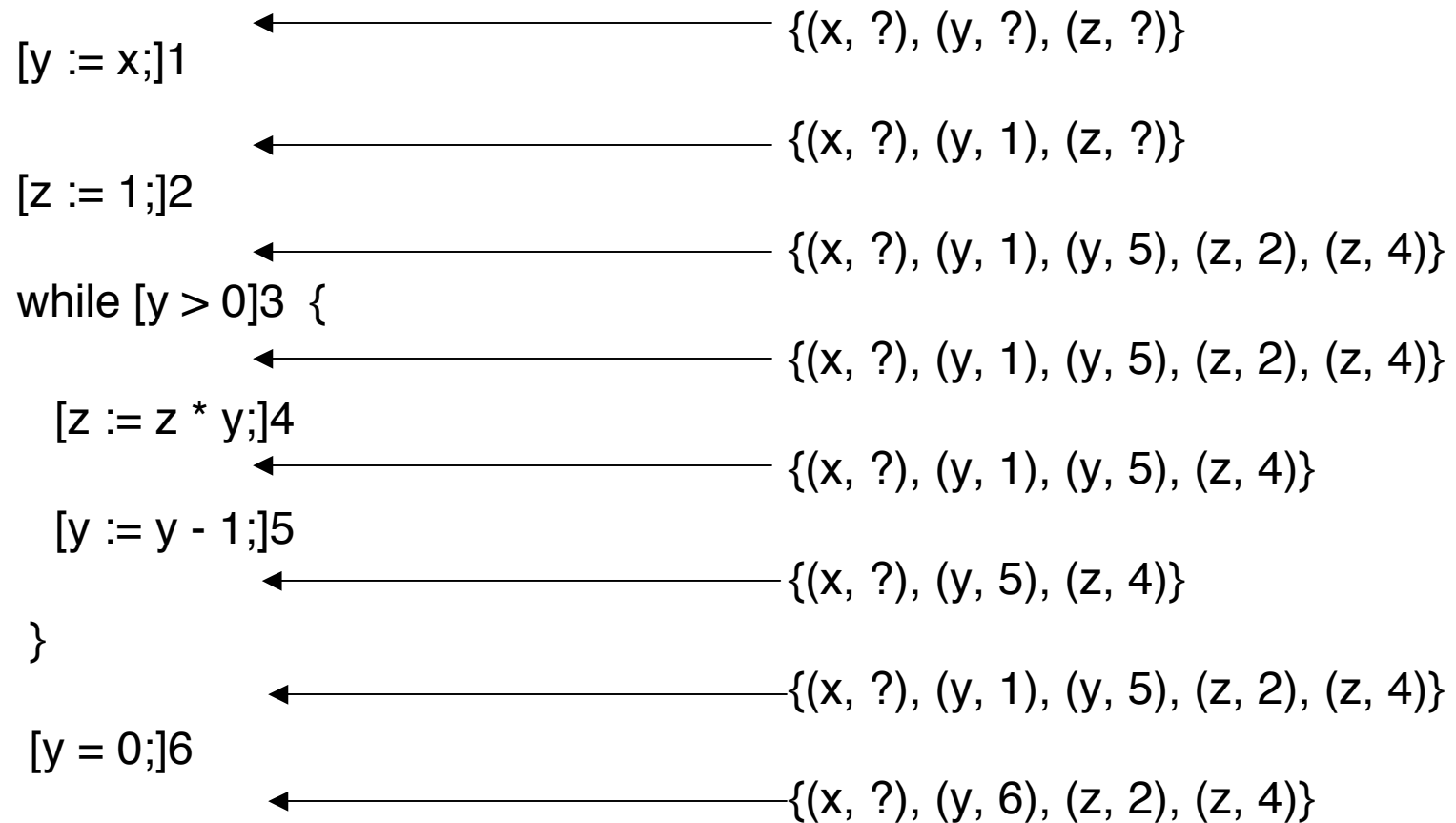


An assignment  $[x := a]i$  reaches  $j$  if there is an execution path from entry to  $j$  where  $x$  was last assigned at  $i$

# Reaching Definition Analysis

## The best solution

---



# Solving the data-flow problem

## Reaching definitions

---

- Domain of analysis
  - The set of all definition points in a procedure/function
  - A definition point  $d$  of variable  $v$  reaches CFG point  $p$  iff there is a path from  $d$  to  $p$  along which  $v$  is not redefined
  - At any CFG point  $p$ , what definition points can reach  $p$ ?
- Reaching definition analysis can be used in
  - Building data-flow graphs
    - Provide info where each operand is defined
  - SSA (static single assignment) construction
    - A representation that encodes both control and data flow of a procedure
- For each basic block  $n$ , let
  - $DEDef(n)$  = definition points whose variables are not redefined in  $n$
  - $DefKill(n)$  = definitions obscured by redefinition of the same name in  $n$

Goal: evaluate all definition points that can reach entry of  $n$

- $RD(n) = \bigcup_{m \in \text{pred}(n)} (DEDef(m) \cup (RD(m) - DefKill(m)))$

# Dataflow Analysis Algorithm

## Computing Reaching Definitions

---

- For each basic block  $n$ , compute
  - $DEDef(n)$  = definition points whose variables are not redefined in  $n$
  - $DefKill(n)$  = definitions obscured by redefinition of the same name in  $n$

Goal: evaluate all definition points that can reach entry of  $n$

$$RD(n) = \bigcup_{m \in \text{pred}(n)} (DEDef(m) \cup (RD(m) - DefKill(m)))$$

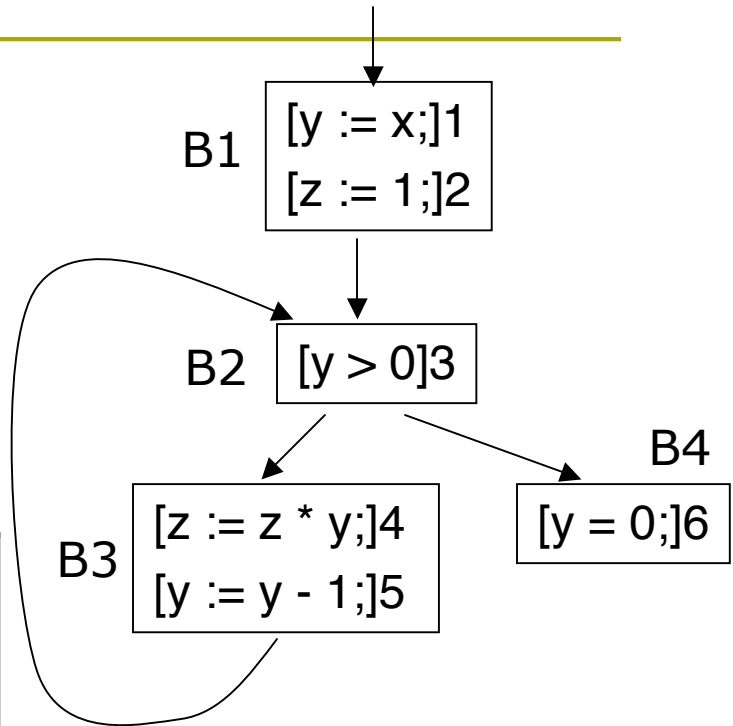
```
for each basic block  $bi$ 
  compute  $DEDef(bi)$  and  $DefKill(bi)$ 
   $RD(bi) := \emptyset$ 
for (changed := true; changed; )
  changed = false
  for each basic block  $bi$ 
    old =  $RD(bi)$ 
     $RD(bi) = \bigcup_{m \in \text{pred}(bi)} (DEDef(m) \cup (RD(m) - DefKill(m)))$ 
    if ( $RD(bi) \neq old$ ) changed := true
```

# Example solution: reaching definition analysis

```

[y := x;]1
[z := 1;]2
while [y > 0]3 {
  [z := z * y;]4
  [y := y - 1;]5
}
[y = 0;]6
    
```

	DEDef	DefKill	RD	RD	RD
B1	1,2	5,6,4	∅	∅	∅
B2	∅	∅	∅	1,2,4,5	1,2,4,5
B3	4,5	1,2,6	∅	1,2,4,5	1,2,4,5
B4	6	1	∅	1,2,4,5	1,2,4,5



Domain: 1 2 4 5 6  
           y z z y y

# Example Constraint-based Analysis

## Loop dependence analysis

---

### Example code

```
for (i=0;i<N;++i) {  
    for (j=0;j<N;++j) {  
        C[i*N+j]=(C[(i-1)*N+j]+C[i*N+j-1])/2;  
    }  
}
```

A loop iteration  $(i,j)$  depends on another iteration  $(i',j')$  if it uses the value computed by  $(i',j')$  or if it writes to a common location written by  $(i',j')$

If a loop iteration  $(i,j)$  depends on iteration  $(i',j')$ , the ordering of the two iterations cannot be switched.

# Loop dependence analysis

## Solving a system of equations

---

### Example code

```
for (i=0;i<N;++i) {  
    for (j=0;j<N;++j) {  
        C[i*N+j]=(C[(i-1)*N+j]+C[i*N+j-1])/2;  
    }  
}
```

- A loop iteration  $(i,j)$  depends on another iteration  $(i',j')$  if  $(i',j')$  computes the value used by  $(i,j)$ , that is
  - If  $(C[i'*N+j'], C[(i-1)*N+j])$   
 $(C[i'*N+j'], C[i*N+j-1])$   
or  $(C[i'*N+j'], C[i*N+j])$  refer to the same location.
  - That is, if  $i'*N+j' = (i-1)*N+j$ ,  $i'*N+j' = i*N+j-1$ ,  
or  $i'*N+j' = i*N+j$

# Example abstract interpretation analysis

## Points-to analysis

### Example program with labels

```
struct Cell {
  int val;
  struct Cell* next;
} *h, *t, *p;
[h = t = NULL;]1
for (int [i=0]2; [i<N]3; [++i]4) {
  [p = new Cell(i, NULL);]5
  if ([h == NULL]6)
    [h = t = p;]7
  else {
    [t->next = p; t = p;]8
  }
}
```

What locations can each pointer variable point to? (can they point to the same location?)

- Define the data to evaluate
  - A set of locations for each pointer variable
  - Keep track of constant values for non-pointer variables
- Define a semantic action for each statement
  - Modifies the location set of pointer variables
  - Allocate new locations
    - Limit the number of locations for each stmt
  - Control flow (conditionals, loops, and function calls)
    - Assume all branches are taken when not sure

# Abstract interpretation of points-to locations

[h = t = NULL;]1	→	h -> 0 t -> 0 p -> ?
[i=0;]2	→	h -> 0 t -> 0 p -> ?
if [i<N]3;	→	h -> 0 t -> 0 p -> ?
[p = new Cell(i,NULL);]5	→	h -> 0 t -> 0 p -> new[5]
if ([h == NULL]6)	→	h -> 0 t -> 0 p -> new[5]
[h = t = p;]7	→	h ->new[5] t ->new[5] p -> new[5]
[++i]4	→	h ->new[5] t ->new[5] p -> new[5]
if [i<N]3;	→	h ->new[5] t ->new[5] p -> new[5]
[p = new Cell(i,NULL);]5	→	h ->{0,new[5]} t ->{0,new[5]} p -> {?,new[5]}
if ([h == NULL]6)	→	h ->{0,new[5]} t ->{0,new[5]} p -> new[5]
else {[t->next = p; t = p;]8	→	h ->{0,new[5]} t ->{0,new[5]} p -> new[5]
[++i]4 if [i<N]3;	→	h ->{0,new[5]} t ->new[5] p -> new[5]
	→	Exit loop if evaluation has stopped changing h ->{0,new[5]} t ->{0,new[5]} p -> {?,new[5]}

# Abstract Interpretation

---

```
AbstractInterpretation(op)
  if (is_assignment(op))
    modify_memory_from_assignment(memory(op), op)
  else if (is_conditional(op)) then
    AbstractInterpretation(cond(op));
    AbstractInterpretation(tree_branch(op));
    AbstractInterpretation(false_branch(op));
  else if (is_loop(op)) then
    repeat
      start_monitor_all_changes(memory(stmts(op)))
      AbstractInterpretation(stmts(op))
    until nothing changes in memory(stmts(op))
  else if (is_procedural_call(op)) then
    setup_parameters_and_return(op);
    AbstractInterpretation(body(op));
  else ...
```

# Example Solution

## Abstract Interpretation

```

struct Cell {
  int val;
  struct Cell* next;
} *h, *t, *p;
[h = t = NULL;]1
for (int [i=0]2; [i<N]3; [++i]4) {
  [p = new Cell(i, NULL);]5
  if ([h == NULL]6)
    [h = t = p;]7
  else {
    [t->next = p; t = p;]8
  }
}
  
```

Domain: h,t,p

0	h->? t->? p->?	
1	h->0 t->0 p->?	
2	h->0 t->0 p->?	
3	h->0 t->0 p->?	h->{0,new[5]} t->{0,new[5]} p->{?,new[5]}
5	h->0 t->0 p->new[5]	h->{0,new[5]} t->{0,new[5]} p->new[5]
6	h->0 t->0 p->new[5]	h->{0,new[5]} t->{0,new[5]} p->new[5]
7	h->new[5] t->new[5] p->new[5]	h->new[5] t->new[5] p->new[5]
8		h->new[5] t->new[5] p->new[5]
4	h->new[5] t->new[5] p->new[5]	h->new[5] t->new[5] p->new[5]

# Example type and effect analysis

## Points-to analysis

### Example program with labels

```
struct Cell {
  int val;
  struct Cell* next;
} *h, *t, *p;
[h = t = NULL;]1
for (int [i=0]2; [i<N]3; [++i]4) {
  [p = new Cell(i,NULL);]5
  if ([h == NULL]6)
    [h = t = p;]7
  else {
    [t->next = p; t = p;]8
  }
}
```

What locations can each pointer variable point to? (can they point to the same location?)

- The type domain: locations
  - Each statement that allocates a new location
  - Each variable that has a location
- Examine each statement and infer a type (a group of locations) for each pointer variable
  - Each pointer variable can have only a single type, no matter where it appears
    - Flow insensitive
- If a distinct type is inferred for each expression, then analysis is flow sensitive

# Applying type and effect approach to points-to analysis

## Example program with labels

```
struct Cell {
  int val;
  struct Cell* next;
} *h, *t, *p;
[h = t = NULL;]1
for (int [i=0]2; [i<N]3; [++i]4) {
  [p = new Cell(i,NULL);]5
  if ([h == NULL]6)
    [h = t = p;]7
  else {
    [t->next = p; t = p;]8
  }
}
```

- The type domain includes
  - NULL, new[5]
- Examine the program text and union all types (locations) for each variable
  - [h=t=NULL]1
    - H->NULL; t->NULL;
  - [p = new Cell(i,NULL);]5
    - P-> new[5]
  - [h = t = p;]7 and [t = p;]8
    - Type(p) is a subset of Type(h)
    - Type(p) is a subset of Type(t)
- Result:
  - h=> {NULL,new[5]}
  - t=> {NULL, new[5]}
  - p=> new[5]
- Key: define typing rules

# Type Inference based points-to analysis

Flow-insensitive type inference:

For each pointer variable  $v$   
do

$\text{Type}(v) = \{\}$

For each operation that  
assigns a new set of  
locations  $L$  to pointer  $v$   
do

$\text{Type}(v) = \text{Type}(v) \cup L$

0	$h \rightarrow \{\}$ $t \rightarrow \{\}$ $p \rightarrow \{\}$
1	$h \rightarrow \{0\}$ $t \rightarrow \{0\}$ $p \rightarrow \{\}$
2	$h \rightarrow \{0\}$ $t \rightarrow \{0\}$ $p \rightarrow \{\}$
3	$h \rightarrow \{0\}$ $t \rightarrow \{0\}$ $p \rightarrow \{\}$
4	$h \rightarrow \{0\}$ $t \rightarrow \{0\}$ $p \rightarrow \{\text{new}[5]\}$
5	$h \rightarrow \{0\}$ $t \rightarrow \{0\}$ $p \rightarrow \{\text{new}[5]\}$
6	$h \rightarrow \{0\}$ $t \rightarrow \{0\}$ $p \rightarrow \{\text{new}[5]\}$
7	$h \rightarrow \{0, \text{new}[5]\}$ $t \rightarrow \{0, \text{new}[5]\}$ $p \rightarrow \{\text{new}[5]\}$
8	$h \rightarrow \{0, \text{new}[5]\}$ $t \rightarrow \{0, \text{new}[5]\}$ $p \rightarrow \{\text{new}[5]\}$