Principles of Program Analysis

An overview of approaches beyond loop analysis and optimizations

The Nature of static analysis --- approximation

- Static program analysis --- predict the dynamic behavior of programs without running them
  - At each 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}
```

  - Cannot be answered precisely as 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 >= 0 at the end of code.
- Program analysis tries to
  - Give approximate answers
  - Prove properties of variables, functions, types

The Nature of Approximation --- may and must analysis

- There are two ways to approximate behavior of programs
  - 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 is problem specific
  - 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 program control flow?
  - 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?
  - Context-insensitive
    - A single solution is computed for each function, no matter who calls it
  - Context-sensitive
    - Different solutions are computed for different chains of callers
- Path sensitivity? Is solution sensitive to execution paths?
  - Path sensitive: different solutions are computed for different paths from program entry to each statement

Scopes of Program Analysis

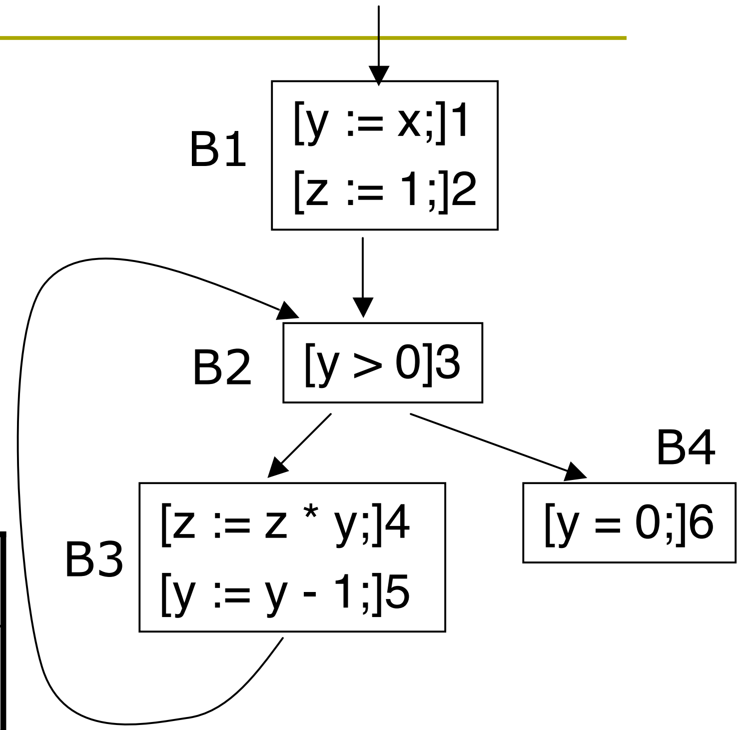
- What code are examined to find the solution?
  - Local analysis
    - Operate on a straight-line sequence of statements (a basic block)
    - Often used as basis for more advanced analysis approaches
  - Regional analysis
    - Operate on code with limited control flow, e.g., loops, conditionals
    - 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
  - 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 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

Foundation of data-flow analysis---Lattices

- An ordered set (L, ≤, V, Λ) is a lattice
  - If x Λ y and x V y exist for all x,y∈L
  - The join operation V: x V y is the least element >= x and y
  - The meet operation Λ : x Λ y is the greatest element <= x and y
- An lattice (L,≤, Λ) is a complete lattice if
  - Each subset Y ⊆ L has a least upper bound and a greatest lower bound
    - LeastUpperBound(Y) = $\bigvee_{m \in Y} m$; GreatestLowerBound(Y) = $\bigwedge_{m \in Y} m$
  - All finite lattices are complete
  - Example lattice that is not complete: the set of all integers I
    - For any x, y∈I, x Λ y = min(x,y), x V y = max(x,y)
    - But LeastUpperBound(I) does not exist
  - Example infinite complete lattice I U {\infty, -\infty}
- Each complete lattice has
  - A top element: the least element
  - A bottom element: the greatest element

Termination of Dataflow Analysis

- A complete lattice L satisfies the finite ascending chain condition if each ascending chain of L eventually stabilizes
  - A set S is a chain if $\forall x,y \in S.\ y \leq x$ or $x \leq y$
  - If $l1 \leq l2 \leq l3 \leq \ldots$, then there is an upper bound $ln = ln+1=ln+2\ldots$
  - This means starting from an arbitrary element $e \in L$, one can only increase e by a finite number of times before reaching an upper bound
- Application to Dataflow Analysis: dataflow information will be lattice values
  - Transfer functions operate on lattice values
  - Solution algorithm will generate increasing sequence of values at each program point
  - Ascending chain condition will ensure termination
- Can use $\vee$ (join) or $\wedge$ (meet) to combine values at control-flow join points

Constraint based Analysis Example: control-flow analysis

- The problem
  - For each function call, what functions may be invoked?
- Syntax-directed analysis
  - Reformulate the analysis specification
    - Construct a finite set of constraints based on structural induction
  - Compute the least solution of the set of constraints
- Each constraint has the form

  (sol1 $\subseteq$ sol2) or ({t} $\subseteq$ sol) or ({t} $\subseteq$ sol1 => sol2 $\subseteq$ sol3)

  - Each sol is either C(l) (l is an expression, e.g., a call site) or P(x) (x is a function parameter/function pointer)
  - Each t is a function definition

Constraint-based Analysis

- For each expression/statement, compute a set of constraints
  - Function definition

    Cond[(fundef(f,x->e0))l] = Cond[e0] ∪

    { {fundef(f,x->e0)}⊆ C(l) } ∪ { fundef(f,x->e0) ⊆ P(f) }
  - Function call (allow functions to return functions as results)

    Cond[((e1)l_1 (e2)l_2)l_3] = Cond[e1] ∪ Cond[e2] ∪

    { {t} ∈ C(l_1)=>C(l_2) ⊆ P(x) ∀ t = (fundef(f,x->e_0) } // parameter

    ∪ { {t} ∈ C(l_1)=> C(l_0) ⊆ C(l_3) ∀ t = (fundef(f,x->e_0)} // result
  - If conditional

    Cond [(if (e0)l_0 then (e1)l_1 else (e2)l_2)l_3] =

    Cond[e0] ∪ Cond[e1] ∪ Cond[e2] ∪ {C(l_2) ⊆ C(l_3)} ∪ {C(l_2) ⊆ C(l_3) }

Solving the constraints

- Input: a set of constraints for the entire program
- Output: the least solution (C,P) to the constraints
- Idea: equivalent to finding the least fixed point of a monotone function defined by the constraints
  - Straight-forward iterative algorithm has n^5 cost, where n is the size of the program (expression)
  - A more sophisticated algorithm takes n^3 complexity
- The graph-based algorithm
  - Build a graph where
    - Each node n corresponds to a unique C(ℓ) or P(x) =>val(n)
    - Add an edge from node n1 to n2 if any change to val(n1) may require modifications to val(n2)
  - Use a worklist to keep track of nodes to change

Example abstract interpretation: 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
    }
 }
```

- 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

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

Abstract interpretation of points-to locations

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

- h -> 0 t -> 0 p -> ?
- h -> 0 t -> 0 p -> ?
- h -> 0 t -> 0 p -> ?
- h -> 0 t -> 0 p -> new[5]
- h -> 0 t -> 0 p -> new[5]
- h ->new[5] t ->new[5] p -> new[5]
- h ->new[5] t ->new[5] p -> new[5]
- h ->{0,new[5]} t ->{0,new[5]} p -> {?,new[5]}
- h ->{0,new[5]} t ->{0,new[5]} p -> new[5]
- h ->{0,new[5]} t ->{0,new[5]} p -> new[5]
- 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 points 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
    Type(v) = {}
For each operation that
    assigns a new set of
    locations L to pointer v
    do
        Type(v) = Type(v) ∪ L
```

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