Program analysis

Mooly Sagiv

html://www.cs.tau.ac.il/~msagiv/courses/wcc08.html

Outline

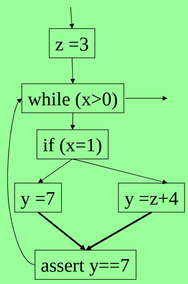
- What is (static) program analysis
- Example
- Undecidability
- An Iterative Algorithm
- Properties of the algorithm
- The theory of Abstract Interpretation

Abstract Interpretation Static analysis

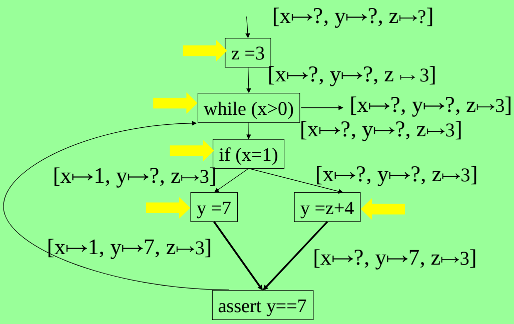
- Automatically identify program properties
  - No user provided loop invariants
- Sound but incomplete methods
  - But can be rather precise
- Non-standard interpretation of the program operational semantics
- Applications
  - Compiler optimization
  - Code quality tools
    - Identify potential bugs
    - Prove the absence of runtime errors
    - Partial correctness

Control Flow Graph(CFG)

```
z = 3
while (x>0) {
    if (x = 1)
      y = 7;
    else
      y =z + 4;
    assert y == 7
}
```



Constant Propagation



Memory Leakage

```
List reverse(Element *head)
{
    List rev, n;
    rev = NULL;
    while (head != NULL) {
          n = head → next;
          head → next = rev;
          head = n;
          rev = head;
    }
    return rev;
}
```

potential leakage of address pointed to by head

Memory Leakage

```
Element* reverse(Element *head)
{
  Element *rev, *n;
  rev = NULL;
  while (head != NULL) {
      n = head → next;
      head → next = rev;
      rev = head;
      head = n;
  }
  return rev; }
```

♘No memory leaks

A Simple Example

```
void foo(char *s )
{
    while ( *s != ' ' )
          s++;
    *s = 0;
}
```

Potential buffer overrun:
offset(s) ≥ alloc(base(s))

A Simple Example

```
void foo(char *s) @require string(s)
{
    while ( *s != ' '&& *s != 0)
          s++;
    *s = 0;
}
```

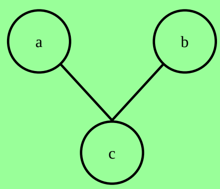
♘No buffer overruns

Example Static Analysis Problem

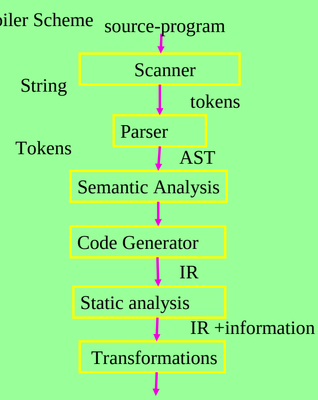
- Find variables which are live at a given program location
- Used before set on some execution paths from the current program point

A Simple Example

```
/* c  */
L0:     a := 0
/* ac */
L1:     b := a + 1
/* bc */
        c := c + b
/* bc */
        a := b * 2
/* ac */
        if c < N goto L1
/* c  */
        return c
```



Compiler Scheme



Undecidability issues

- It is impossible to compute exact static information
- Finding if a program point is reachable
- Difficulty of interesting data properties

Undecidabily

- A variable is live at a given point in the program
  - if its current value is used after this point prior to a definition in some execution path
- It is undecidable if a variable is live at a given program location

Proof Sketch

Pr

L: x := y

Is y live at L?

Conservative (Sound)

- The compiler need not generate the optimal code
- Can use more registers (“spill code”) than necessary
- Find an upper approximation of the live variables
- Err on the safe side
- A superset of edges in the interference graph
- Not too many superfluous live variables

Conservative(Sound) Software Quality Tools

- Can never miss an error
- But may produce false alarms
  - Warning on non existing errors

Data Flow Values

- Order data flow values
  - $a \sqsubseteq b \Leftrightarrow$ a “is more precise than” b
  - In live variables
    - $a \sqsubseteq b \Leftrightarrow a \subseteq b$
  - In constant propagation
    - $a \sqsubseteq b \Leftrightarrow$ a includes more constants than b
- Compute the least solution
- Merge control flow paths optimistically
  - $a \sqcup b$
  - In live variables
    - $a \sqcup b = a \cup b$

Transfer Functions

- Program statements operate on data flow values conservatively

Transfer Functions (Constant Propagation)

- Program statements operate on data flow values conservatively
- If a=3 and b=7 before "z = a + b;"
  - then a=3, b =7, and z =10 after
- If a=? and b=7 before "z = a + b;"
  - then a=?, b =7, and z =? After
- For x = exp
  - CpOut = CpIn [x ↦ [[exp]](CpIn)]

Transfer Functions LiveVariables

- If a and c are potentially live after "a = b *2"
  - then b and c are potentially live before
- For "x = exp;"
  - LiveIn = Livout – {x} ∪ arg(exp)

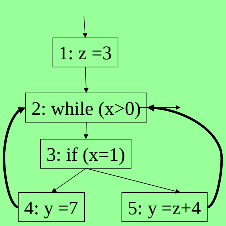
Iterative computation of conservative static information

- Construct a control flow graph(CFG)
- Optimistically start with the best value at every node
- “Interpret” every statement in a conservative way
- Forward/Backward traversal of CFG
- Stop when no changes occur

Pseudo Code (forward)

```
forward(G(V, E): CFG, start: CFG node, initial: value){
   // initialization
  value[start]:= initial
  for each v ∈ V – {start} do value[v] := ⊥
  // iteration
   WL = V
    while WL != {} do
        select and remove a node v ∈WL
        for each u ∈ V such that (v, u) ∈ E do
              value[u] := value[u] ⊔ f(v, u)(value[v])
              if value[u] was changed WL := WL ∪ {u}
```

Constant Propagation



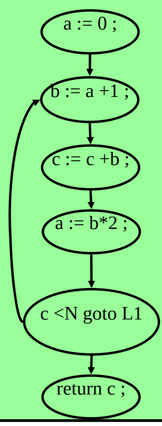
N	Val	WL
	v[1]=[x↦?, y↦?, z ↦?]	{1, 2, 3, 4, 5}
1	v[2]=[x↦?, y↦?, z ↦3]	{2, 3, 4, 5}
2	v[3]=[x↦?, y↦?, z ↦3]	{3, 4, 5}
3	v[4] =[x↦?, y↦?, z↦3] v[5] =[x↦?, y↦?, z↦3]	{4, 5}
4		{5}
5		{}

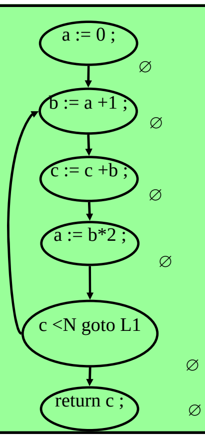
Only values before CFG are shown

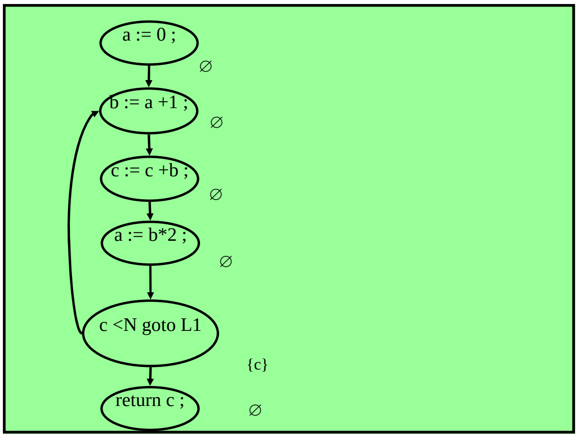
Pseudo Code (backward)

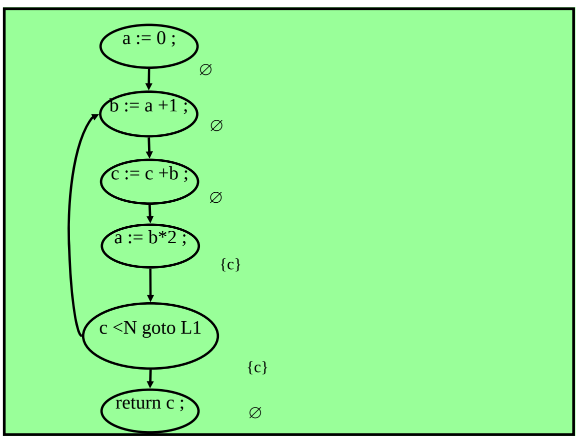
```
backward(G(V, E): CFG, exit: CFG node, initial: value){
  // initialization
  value[exit]:= initial
  for each v ∈ V – {exit} do value[v] := ⊥
  // iteration
  WL = V
  while WL != {} do
    select and remove a node v ∈WL
    for each u ∈ V such that (u, v) ∈ E do
      value[u] := value[u] ⊔ f(v, u)(value[v])
      if value[u] was changed WL := WL ∪ {u}
```

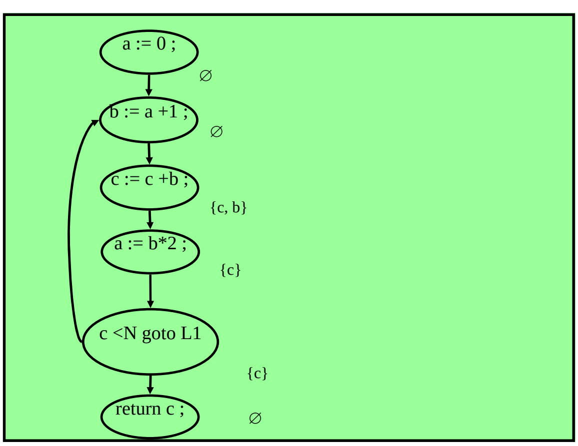
```
/* c  */
L0:      a := 0
/* ac */
L1:      b := a + 1
/* bc */
         c := c + b
/* bc */
         a := b * 2
/* ac */
         if c < N goto L1
/* c  */
         return c
```

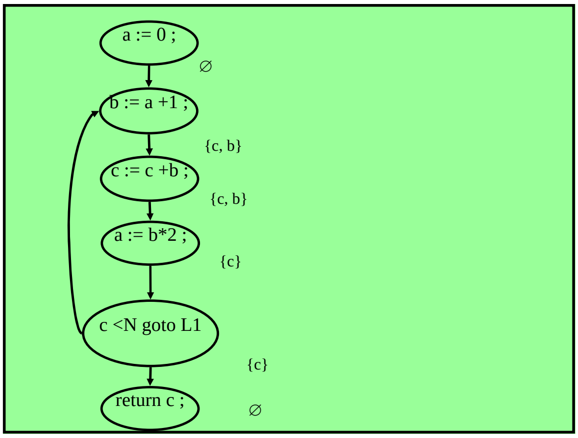


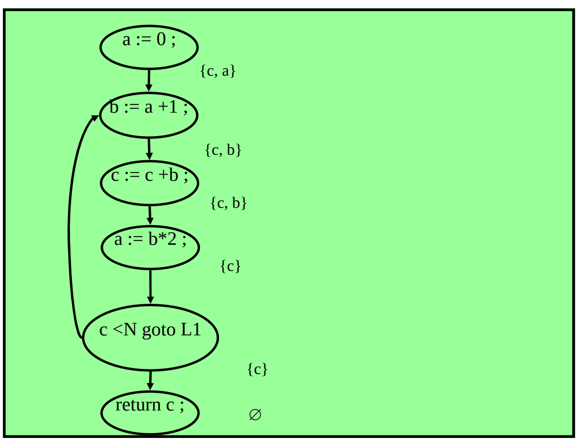


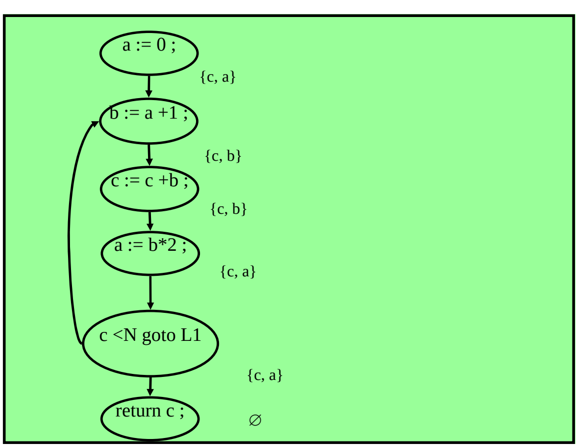

a := 0 ;
∅
b := a +1 ;
∅
c := c +b ;
∅
a := b*2 ;
∅
c <N goto L1
{c}
return c ;
∅


a := 0 ;
∅
b := a +1 ;
∅
c := c +b ;
∅
a := b*2 ;
{c}
c <N goto L1
{c}
return c ;
∅


a := 0 ;
∅
b := a +1 ;
∅
c := c +b ;
{c, b}
a := b*2 ;
{c}
c <N goto L1
{c}
return c ;
∅


a := 0 ;
∅
b := a +1 ;
{c, b}
c := c +b ;
{c, b}
a := b*2 ;
{c}
c <N goto L1
{c}
return c ;
∅


a := 0 ;
{c, a}
b := a +1 ;
{c, b}
c := c +b ;
{c, b}
a := b*2 ;
{c}
c <N goto L1
{c}
return c ;
∅


a := 0 ;
{c, a}
b := a +1 ;
{c, b}
c := c +b ;
{c, b}
a := b*2 ;
{c, a}
c <N goto L1
{c, a}
return c ;
∅

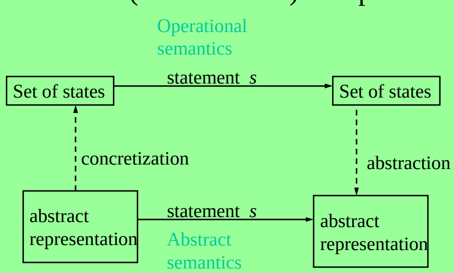
Summary Iterative Procedure

- Analyze one procedure at a time
  - More precise solutions exit
- Construct a control flow graph for the procedure
- Initializes the values at every node to the most optimistic value
- Iterate until convergence

Abstract Interpretation

- The mathematical foundations of program analysis
- Established by Cousot and Cousot 1979
- Relates static and runtime values

Abstract (Conservative) interpretation

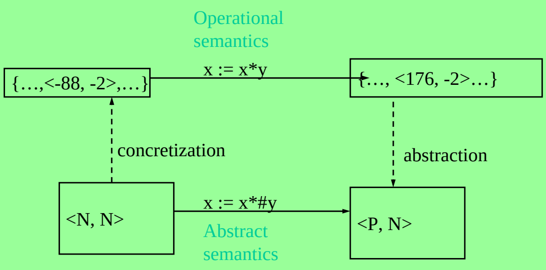


Example rule of signs

- Safely identify the sign of variables at every program location
- Abstract representation {P, N, ?}
- Abstract (conservative) semantics of *

*#	P	N	?
P	P	N	?
N	N	P	?
?	?	?	?

Abstract (conservative) interpretation



Example rule of signs

- Safely identify the sign of variables at every program location
- Abstract representation {P, N, ?}
- α(C) = if all elements in C are positive
  then return P
  else if all elements in C are negative
  then return N
  else return ?
- γ(a) = if (a==P) then
  return{0, 1, 2, … }
  else if (a==N)
  return {-1, -2, -3, …, }
  else return Z

Example Constant Propagation

- Abstract representation
  – set of integer values and and extra value “?” denoting variables not known to be constants
- Conservative interpretation of +

+#	?	0	1	2
?	?	?	?	?
0	?	0	1	2
1	?	1	2	3
2	?	2	3	4

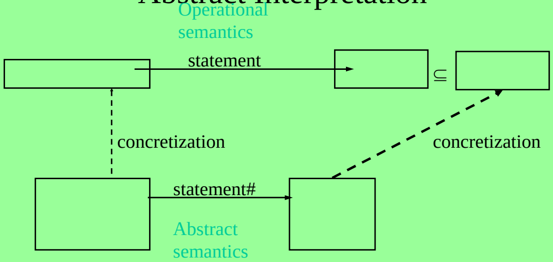
Example Program

```
x = 5;
y = 7;
if (getc())
    y = x + 2;
z = x +y;
```

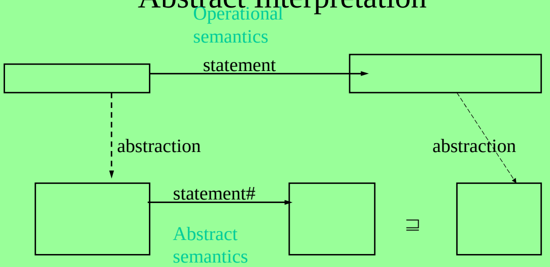
Example Program (2)

```
if (getc())
    x= 3 ; y = 2;
   else
   x =2; y = 3;
z = x +y;
```

Local Soundness of Abstract Interpretation



Local Soundness of Abstract Interpretation



Some Success Stories Software Quality Tools

- The prefix Tool identified interesting bugs in Windows
- The Microsoft SLAM tool checks correctness of device driver
  - Driver correctness rules
- Polyspace checks ANSI C conformance
  - Flags potential runtime errors

Summary

- Program analysis provides non-trivial insights on the runtime executions of the program
  - Degenerate case – types (flow insensitive)
- Mathematically justified
  - Operational semantics
  - Abstract interpretation (lattice theory)
- Employed in compilers
- Will be employed in software quality tools