Techniques for debugging and repair programs

Wei Le

October 3, 2023

Agenda

- Debugging and repair programs
  - identify the location of a bug
  - understand the root cause of a program
  - develop a patch that can pass all the tests
- Dependency and slicing
- Delta-debugging
- Program repair

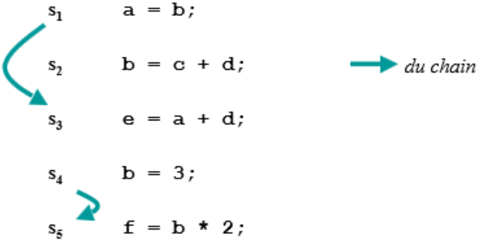
Dependency and slicing

- Dependencies: *control dependency* and *data dependency*
- *Dependency graphs*
- *Program slicing* (chopping, dicing, path slicing, thin slicing, executable slicing)
- *Taint analysis*

Data dependency

- Two statements are *data dependent*: the definition of a variable in a statement reaches the use of the same variable at another statement
- Data dependency specifies the constraints on the order in which statement may be executed
- How to automatically compute data depencencies:
  - Du chains
  - SSA representation
  - PDG

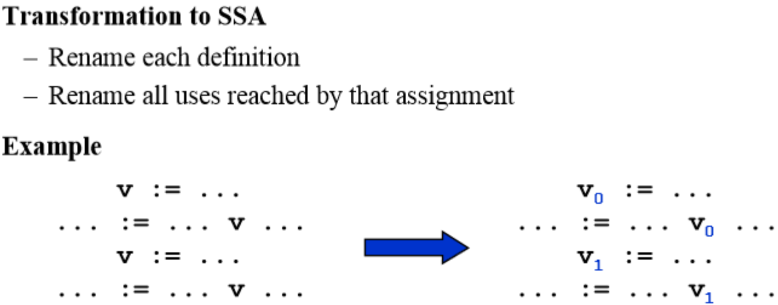
DU chains: def-use chains (link each def to uses)



```
s1    a = b;
s2    b = c + d;
s3    e = a + d;
s4    b = 3;
s5    f = b * 2;
```

- pro: fast to get data dependencies
- con: must be computed and updated, space overhead

SSA: static single assignment



Transformation to SSA

– Rename each definition

– Rename all uses reached by that assignment

Example

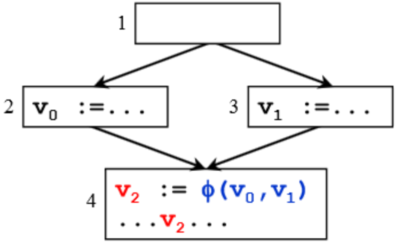
$$
\begin{array}{rcl}
\begin{array}{l}
\quad\;\; \mathtt{v} \;\; \mathtt{:=} \;\; \mathtt{...} \\
\mathtt{...} \;\; \mathtt{:=} \;\; \mathtt{...} \;\; \mathtt{v} \;\; \mathtt{...} \\
\quad\;\; \mathtt{v} \;\; \mathtt{:=} \;\; \mathtt{...} \\
\mathtt{...} \;\; \mathtt{:=} \;\; \mathtt{...} \;\; \mathtt{v} \;\; \mathtt{...}
\end{array}
& \Longrightarrow &
\begin{array}{l}
\quad\;\; \mathtt{v}_0 \;\; \mathtt{:=} \;\; \mathtt{...} \\
\mathtt{...} \;\; \mathtt{:=} \;\; \mathtt{...} \;\; \mathtt{v}_0 \;\; \mathtt{...} \\
\quad\;\; \mathtt{v}_1 \;\; \mathtt{:=} \;\; \mathtt{...} \\
\mathtt{...} \;\; \mathtt{:=} \;\; \mathtt{...} \;\; \mathtt{v}_1 \;\; \mathtt{...}
\end{array}
\end{array}
$$

SSA

Merging Definitions

– φ-functions merge multiple reaching definitions

Example



SSA

Transformation to SSA

- rename variables
- place ϕ function

- each value produced in the program is represented using a variable
- pro: allow analyses and transformations to be simpler and more efficient
- con: may not be executable (requires extra translations to and from); space and time overhead

Data Dependency Graphs

Data dependency graphs [1987:ferrante]: node is the statement, edge is the data dependency relation

It is acyclic unless there is a loop in the program

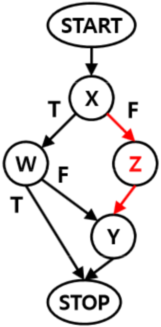
Control Dependency

Intuitively, control dependency between two statements exists if one statement "controls"/"determine" the execution of the other (e.g. through if- or while-statements).

Control Dependency

Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X iff

- There exists a directed path **P** from X to Y with any Z in P post-dominated by Y and
- X is not post-dominated by Y

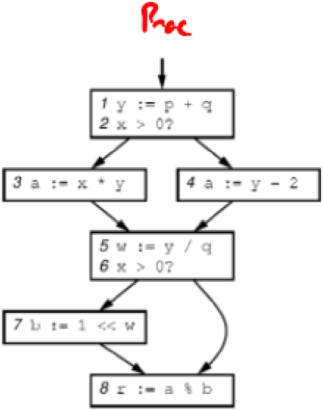


*If **Y** is control dependent on **X** then **X** must have **two exits**.*

Control Dependency

- the first statement is *control dependent* on the entry of the program
- from Condition 1: Y is not control dependent on any node(s) between X and Y; that is, X is the first node Y is control dependent on

Control Dependency: Example

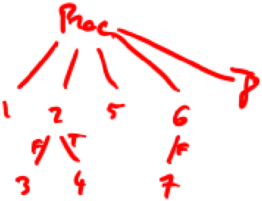


Control dependence relation

3 depends on 2

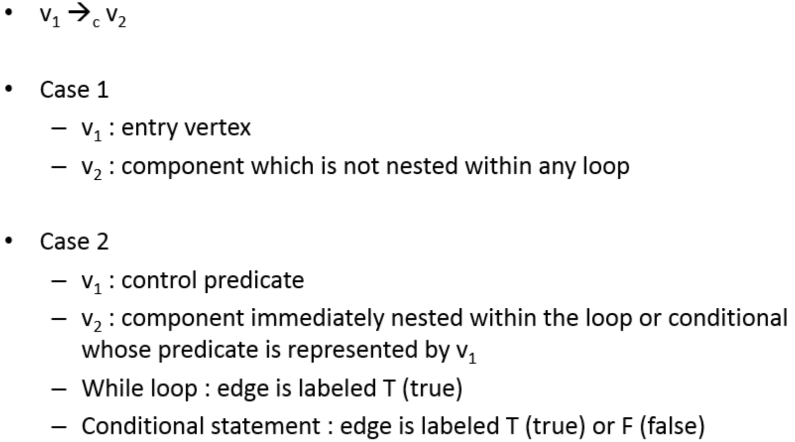
4 " " 2

7 " " 6



Control Dependency Graph

node is the statement, edge is the control dependency relation



- $v_1 \rightarrow_c v_2$
- Case 1
  - – v_1 : entry vertex
  - – v_2 : component which is not nested within any loop
- Case 2
  - – v_1 : control predicate
  - – v_2 : component immediately nested within the loop or conditional whose predicate is represented by v_1
  - – While loop : edge is labeled T (true)
  - – Conditional statement : edge is labeled T (true) or F (false)

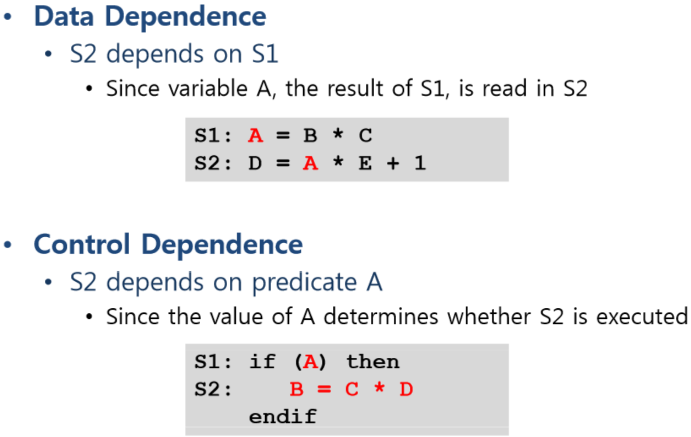
Program Dependence Graphs (PDG)

Node: statements

Edge: control and data dependency edges

Data Dependency + Control Dependency [1987:Ferrante:TOPLAS]

Dependency is transitive



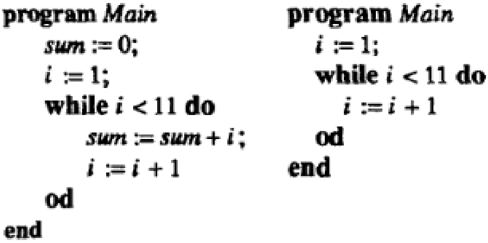
- **Data Dependence**
  - S2 depends on S1
    - Since variable A, the result of S1, is read in S2

```
S1: A = B * C
S2: D = A * E + 1
```

- **Control Dependence**
  - S2 depends on predicate A
    - Since the value of A determines whether S2 is executed

```
S1: if (A) then
S2:     B = C * D
    endif
```

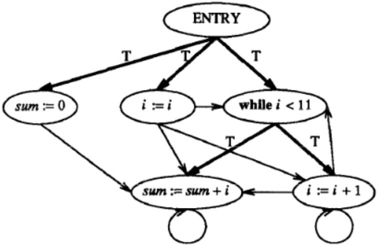
Program Dependence Graphs (PDG): Example

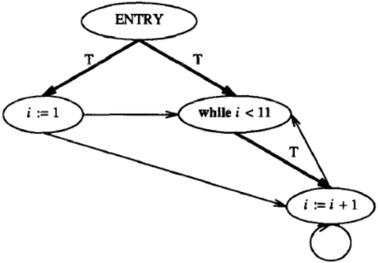


```
program Main
    sum := 0;
    i := 1;
    while i < 11 do
        sum := sum + i;
        i := i + 1
    od
end
```

```
program Main
    i := 1;
    while i < 11 do
        i := i + 1
    od
end
```

Program Dependence Graphs (PDG): Example





Construct PDG

General approaches

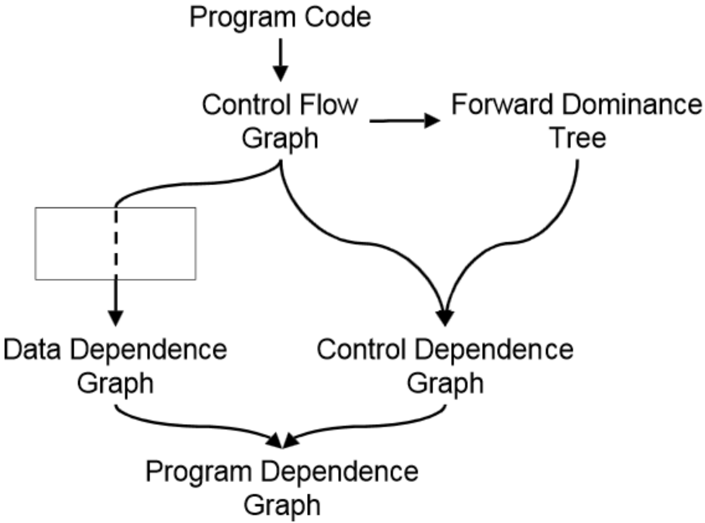
- Data dependence: def-use relations
- Control dependence: control flow graphs, post dominator analysis

Tools: Frama-C (for C), Code Surfer (C/C++), Atlas

Relevant papers:

- The program dependence graph and its use in optimization, 1987, TOPLAS
- Efficiently computing static single assignment form the control dependence graph, 1991, TOPLAS
- Interprocedural slicing using dependence graphs, 1988, PLDI

Construct PDG



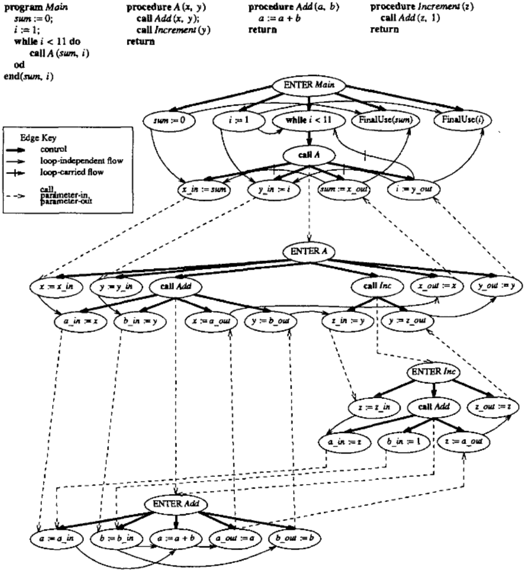
System Dependence Graphs (optional)

System Dependence Graphs (SDG)

SDG: an interprocedural dependence graph representation – a collection of method dependence graphs [1988:Horwitz, 1996:Larsen] (also used by Horwitz, Reps, Binkley)

- Program dependence graph: Represents the system's main program
- Procedure dependence graphs: Represent the system's auxiliary procedures
- Some additional edges
  - Edges that represent direct dependence between a call site and the called procedure
  - Edges that represent transitive dependence due to calls

System Dependence Graphs (SDG)



```
program Main
   sum := 0;
   i := 1;
   while i < 11 do
        call A (sum, i)
   od
end(sum, i)

procedure A (x, y)
   call Add (x, y);
   call Increment (y)
return

procedure Add (a, b)
   a := a + b
return

procedure Increment (z)
   call Add (z, 1)
return
```

System Dependence Graphs (SDG)

Five new vertices for SDG

- Call-site vertex
- Actual-in:
  - Control dependent on call-site vertex
  - Copy values of actual parameters to call temporaries
- Actual-out:
  - Control dependent on call-site vertex
  - Copy from return temporaries
- Formal-in:
  - Control dependent on procedure's entry vertex
  - Copy value of formal parameters from call temporaries
- Formal-out:
  - Control dependent on procedure's entry vertex
  - Copy to return temporaries

System Dependence Graphs (SDG)

Three new edges for SDG

- Call edge
  - Call-site → Procedure-entry
  - From each call-site vertex to the corresponding procedure-entry vertex
- Parameter-in edge
  - Actual-in → Formal-in
  - From each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure
- Parameter-out edge
  - Formal-out → Actual-out
  - From each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site

Program Slicing

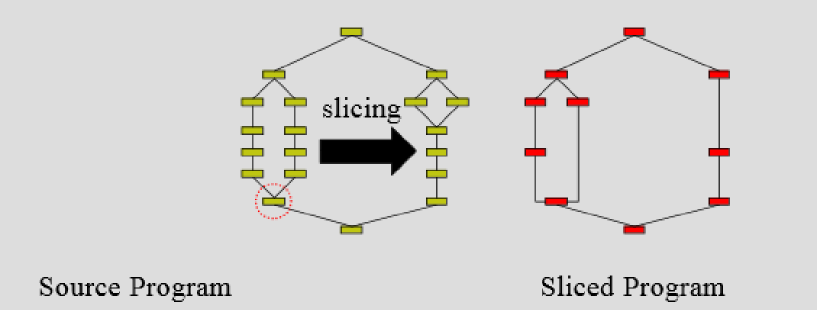
Origin of the Idea

Analysis technique introduced by Mark Weiser in his PHD thesis (1979)

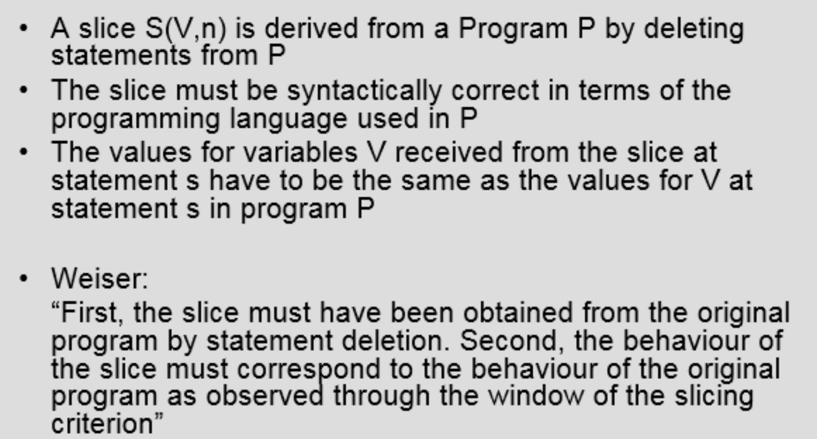
- Idea derived when he was observing experienced programmers debugging a program
- Result: Every experienced programmer uses slicing to debug a program

Program Slicing: Intuitive Understanding

Intuitively, the slice of a program with respect to program point *p* and variable *x* consists of all statements and predicates of the program that might affect the value of *x* at point *p*



Program Slicing: Intuitive Understanding



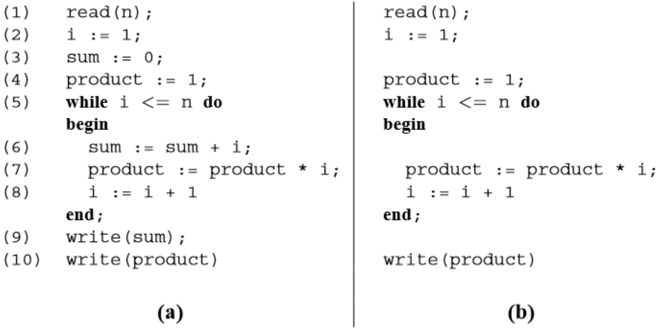
- A slice S(V,n) is derived from a Program P by deleting statements from P
- The slice must be syntactically correct in terms of the programming language used in P
- The values for variables V received from the slice at statement s have to be the same as the values for V at statement s in program P

- Weiser:
  "First, the slice must have been obtained from the original program by statement deletion. Second, the behaviour of the slice must correspond to the behaviour of the original program as observed through the window of the slicing criterion"

Program Slicing: Definition

[1981:Weiser:ICSE] [1995:Tip]

- *(Backward) slice* of v at S is the set of statements involved in computing v's value at S
- A *slicing criterion* of a program P is a tuple $\langle i, V \rangle$, where i is a statement in P and V is a subset of the variables in P



```
(1)    read(n);
(2)    i := 1;
(3)    sum := 0;
(4)    product := 1;
(5)    while i <= n do
       begin
(6)      sum := sum + i;
(7)      product := product * i;
(8)      i := i + 1
       end;
(9)    write(sum);
(10)   write(product)
```

(a)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product)
```

(b)

(a) An example program. **(b)** A slice of the program w.r.t. criterion (10, product).

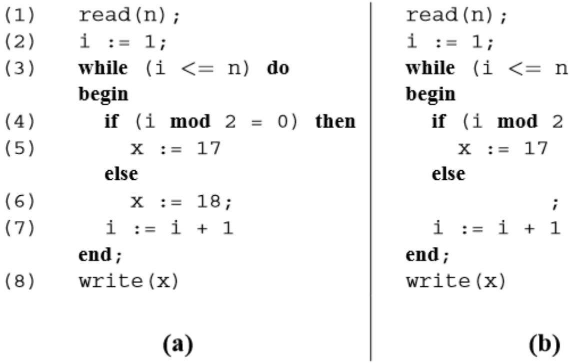
Program Slicing: Definition

static slice and *dynamic slice*:

- *static slice* is computed without making assumptions regarding a program's input (for all possible inputs and paths)
- the computation of *dynamic slice* relies on a specific test case

Program Slicing: Definition

What is the static slice for the program on the left?



```
(1)    read(n);
(2)    i := 1;
(3)    while (i <= n) do
       begin
(4)      if (i mod 2 = 0) then
(5)        x := 17
         else
(6)        x := 18;
(7)      i := i + 1
       end;
(8)    write(x)
```

(a)

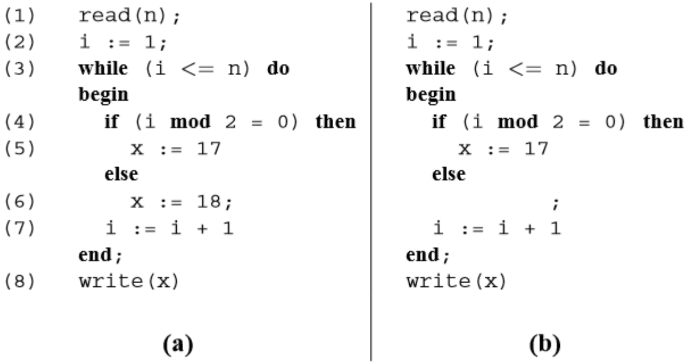
```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else
             ;
  i := i + 1
end;
write(x)
```

(b)

(a) Another example program. **(b)** Dynamic slice w.r.t. criterion (n = 2, 8^1, x)

Program Slicing: Definition

What is the static slice for the program on the left?



```
(1)   read(n);
(2)   i := 1;
(3)   while (i <= n) do
      begin
(4)     if (i mod 2 = 0) then
(5)       x := 17
        else
(6)       x := 18;
(7)     i := i + 1
      end;
(8)   write(x)
```

(a)

```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else
          ;
  i := i + 1
end;
write(x)
```

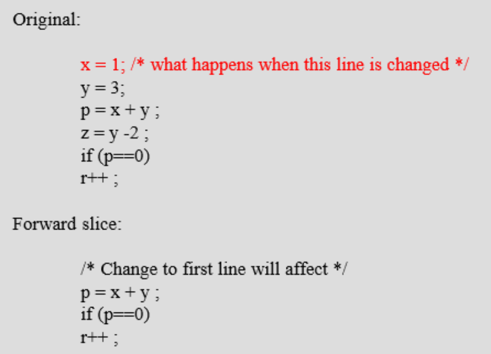
(b)

(a) Another example program. **(b)** Dynamic slice w.r.t. criterion (n = 2, 8^1, x)

Here, the static slice is the entire program

Program Slicing: Definition

Forward slice of a program with respect to a program point p and variable x consists of all statements and predicates of the program that might be affected by the value of x at point p



Original:

```
        x = 1; /* what happens when this line is changed */
        y = 3;
        p = x + y ;
        z = y -2 ;
        if (p==0)
        r++ ;
```

Forward slice:

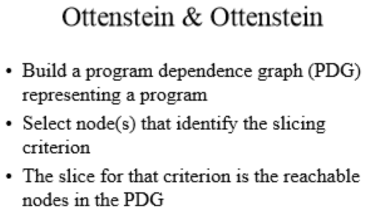
```
        /* Change to first line will affect */
        p = x + y ;
        if (p==0)
        r++ ;
```

Compute Slice

- A data slice is obtained by only taking data dependence into account; a control slice consists of the set of control predicates surrounding a language construct.
- *Data slice*: nodes that v transitively data dependent on – finding transitive data dependence on the data dependence graph
- *Control slice*: nodes that v transitive control dependent on – finding transitive control dependence on the control dependence graph
- *Program slice*: The closure of all data and control slices w.r.t. an expression on the PDG is the slice w.r.t. the set of variables used in the expression.

Compute Slice: Algorithms:

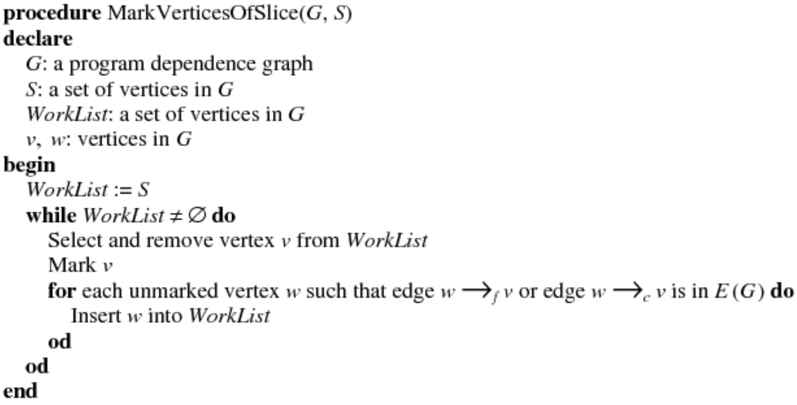
Reachability on PDG



Ottenstein & Ottenstein

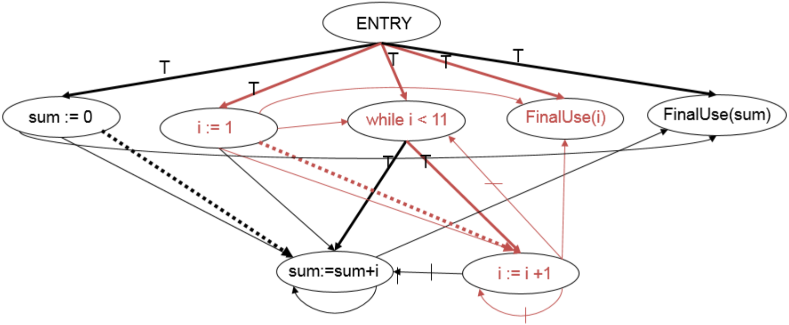
- Build a program dependence graph (PDG) representing a program
- Select node(s) that identify the slicing criterion
- The slice for that criterion is the reachable nodes in the PDG

Compute Slice: Algorithms



```
procedure MarkVerticesOfSlice(G, S)
declare
  G: a program dependence graph
  S: a set of vertices in G
  WorkList: a set of vertices in G
  v, w: vertices in G
begin
  WorkList := S
  while WorkList ≠ ∅ do
    Select and remove vertex v from WorkList
    Mark v
    for each unmarked vertex w such that edge $w \longrightarrow_f v$ or edge $w \longrightarrow_c v$ is in $E(G)$ do
      Insert w into WorkList
    od
  od
end
```

Slicing on FinalUse(i)

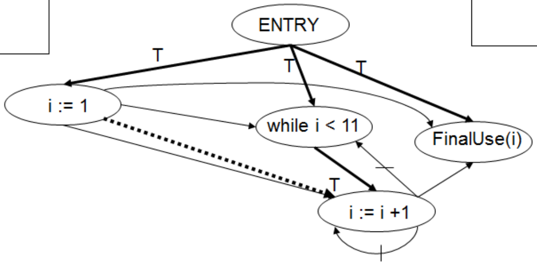


Slicing on FinalUse(i)

```
program Main
  sum := 0;
  i := 1;
  while i < 11 do
   sum := sum +i;
   i := i+1
  od
End(sum,i)
```

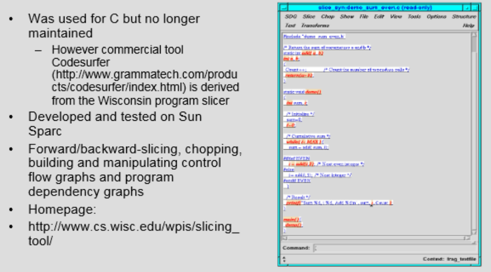
Slice on *FinalUse(i)*

```
program Main
   i := 1;
   while i < 11 do
       i := i+1;
   od
End(i)
```



Compute Slice: Tools

- Code Surfer (academic license)

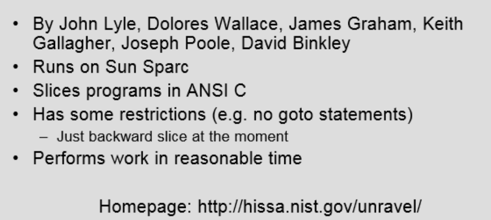


- Was used for C but no longer maintained
  - However commercial tool Codesurfer (http://www.grammatech.com/products/codesurfer/index.html) is derived from the Wisconsin program slicer
- Developed and tested on Sun Sparc
- Forward/backward-slicing, chopping, building and manipulating control flow graphs and program dependency graphs
- Homepage:
- http://www.cs.wisc.edu/wpis/slicing_tool/

slice_syn:demo_sum_even.c (read-only)
SDG Slice Chop Show File Edit View Tools Options Structure
Text Transforms Help

```
#include "demo_sum_even.h"

/* Return the sum of parameters a and b */
static int add( a , b)
int a, b;
{
 Count++;            /* Count the number of procedure calls */
 return(a+b);
}

static void demo()
{
 int sum, i;

 /* Initialize */
 sum=0;
 i=0;

 /* Cumulative sum */
 while( i< MAX ){
  sum = add( sum, i);

#ifdef EVEN
  i = add(i,2); /* Next even integer */
#else
  i = add(i,1); /* Next integer */
#endif EVEN
 }

 /* Result */
 printf("Sum %d, i %d, Add %d\n", sum, i, Count );
}

main(){
 demo();
}
```

- Unravel



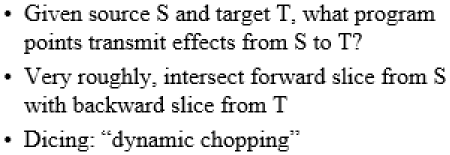
- By John Lyle, Dolores Wallace, James Graham, Keith Gallagher, Joseph Poole, David Binkley
- Runs on Sun Sparc
- Slices programs in ANSI C
- Has some restrictions (e.g. no goto statements)
  - Just backward slice at the moment
- Performs work in reasonable time

Homepage: http://hissa.nist.gov/unravel/

Chopping and Dicing (optional)

Chopping and dicing: Combining two slices

Program chopping

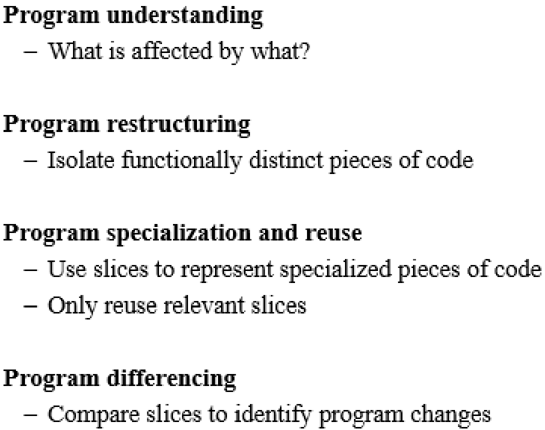


- Given source S and target T, what program points transmit effects from S to T?
- Very roughly, intersect forward slice from S with backward slice from T
- Dicing: “dynamic chopping”

Chopping and dicing: Combining two slices

- *Program dicing* [lyle:weiser:1987] [chen:1993] – used for fault localization
- a method for combining the information of different slices
- a program computes a correct value for variable x and an incorrect value for variable y, the bug is likely to be found in statements that are in the slice w.r.t. y, but not in the slice w.r.t. x.
- A static program dice is the set difference of the static slice of an incorrect variable and the static slice of a correct variable.

Program Slicing: Applications



Program understanding

- What is affected by what?

Program restructuring

- Isolate functionally distinct pieces of code

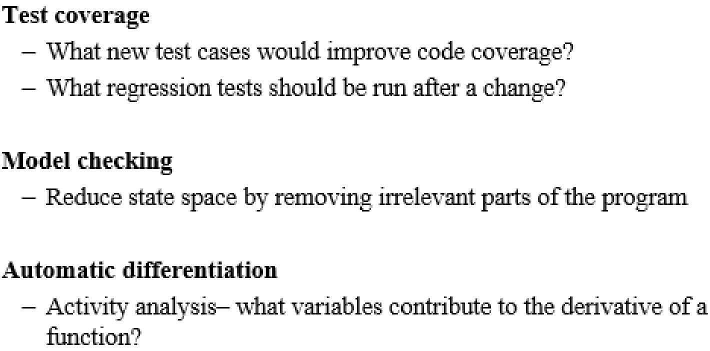
Program specialization and reuse

- Use slices to represent specialized pieces of code
- Only reuse relevant slices

Program differencing

- Compare slices to identify program changes

Program Slicing: Applications



Test coverage

- What new test cases would improve code coverage?
- What regression tests should be run after a change?

Model checking

- Reduce state space by removing irrelevant parts of the program

Automatic differentiation

- Activity analysis– what variables contribute to the derivative of a function?

Program Slicing: Applications

backward slicing [1990:Horwitz:PLDI]

- Debugging
- Understand complicated code
- Isolate individual computation threads within a program, automatic parallelization
- Automatically integrating program variants (merge commits) [1987:Horwitz:POPL]
- ...

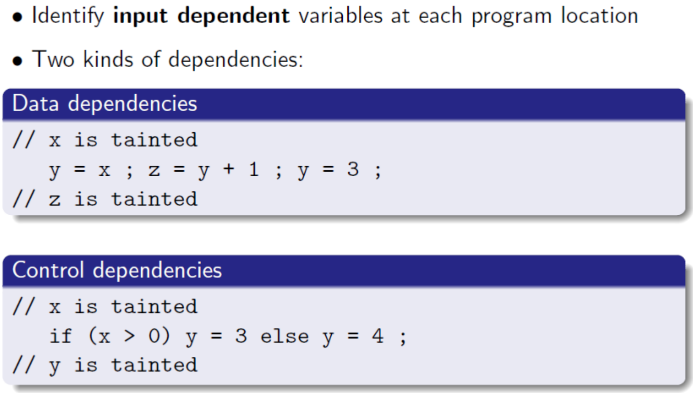
Program Slicing: Applications

Forward slicing

- Show how a value computed is being used subsequently
- Inspect the parts of a program that may be affected by a proposed modification, to check that there are no unforeseen effects on the program's behavior [1995:Tip]
- Taint analysis
- ...

Taint Analysis

- information flows from object x to object y, denoted $x \to y$, whenever information stored in x is transferred to, object y. (forward slicing)



- Identify **input dependent** variables at each program location
- Two kinds of dependencies:

Data dependencies

```
// x is tainted
   y = x ; z = y + 1 ; y = 3 ;
// z is tainted
```

Control dependencies

```
// x is tainted
   if (x > 0) y = 3 else y = 4 ;
// y is tainted
```

Dynamic Slicing

See Xiangyu Zhang's Slides

References and Further Reading

- Path slicing
- Thin slicing
- All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)
- Certification of programs for secure information flow