

IA159 Formal Methods for Software Analysis

Program Slicing and Points-to Analysis

Jan Strejček

Faculty of Informatics
Masaryk University

focus

- slicing via dependence graphs
- points-to analysis
- static single assignment (SSA)
- data dependencies
- control dependencies

source

- M. Chalupa: [Program Slicing and Symbolic Execution for Verification](#), PhD thesis, 2021.

Program slicing reduces a given program by removing statements that are irrelevant for a given **slicing criterion**.

Program slicing reduces a given program by removing statements that are irrelevant for a given **slicing criterion**.

A typical **slicing criterion** is a specific statement or a set of statements. Sliced program should preserve all executions of these statements, i.e., preserve the reachability of these statements and all data they process.

Program slicing reduces a given program by removing statements that are irrelevant for a given **slicing criterion**.

A typical **slicing criterion** is a specific statement or a set of statements. Sliced program should preserve all executions of these statements, i.e., preserve the reachability of these statements and all data they process.

- introduced in *M. D. Weiser: Program Slicing, ICSE 1981*
- the approach based on **dependence graphs** presented in *K. J. Ottenstein and L. M. Ottenstein: The Program Dependence Graph in a Software Development Environment, SDE 1984*

Applications of program slicing

- program debugging
- code comprehension
- code optimization including automatic parallelization
- software verification
- ...

Applications of program slicing

- program debugging
- code comprehension
- code optimization including automatic parallelization
- software verification
- ...

a typical application in software verification (implemented in **Symbiotic**)

- 1 find potentially erroneous statements by a cheap analysis
- 2 slice the program to preserve all executions of these statements
- 3 verify the sliced program

Which statements are irrelevant for the `assert`?

```
1  z = z + 3;
2  if (z > 0) {
3    x = z + 1;
4    z = 3 * x;
5  } else {
6    y = z + 5;
7    x = x * x - z;
8  }
9  if (x > y)
10   z = x - 1;
11  assert(x > 0);
```


Which statements are irrelevant for the `assert`?

```
1  z = z + 3;
2  if (z > 0) {
3    x = z + 1;
4    z = 3 * x;
5  } else {
6    y = z + 5;
7    x = x * x - z;
8  }
9  if (x > y)
10   z = x - 1;
11  assert(x > 0);
```

```
1  z = z + 3;
2  if (z > 0) {
3    x = z + 1;
4
5  } else {
6
7    x = x * x - z;
8  }
9
10
11  assert(x > 0);
```

Basic slicing algorithm

- 1 build a **dependence graph** for the given program
 - nodes are statements
 - edges correspond to **data** and **control dependencies**
- 2 sliced program corresponds to the nodes that are backward reachable from the slicing criterion(s)

intuitive meanings

- a statement r is **data dependent** on a statement w if there exists a program execution where r reads a value from a memory that has been written by w
- a statement n is **control dependent** on a statement b if b is the closest point where a program execution may go some way that misses n
- in practice, we compute overapproximations

Simple dependency graph

```
1 z = z + 3;  
2 if (z > 0) {  
3   x = z + 1;  
4   z = 3 * x;  
5 } else {  
6   y = z + 5;  
7   x = x * x - z;  
8 }  
9 if (x > y)  
10   z = x - 1;  
11 assert(x > 0);
```

Simple dependency graph

```
1 z = z + 3;
2 if (z > 0) {
3   x = z + 1;
4   z = 3 * x;
5 } else {
6   y = z + 5;
7   x = x * x - z;
8 }
9 if (x > y)
10  z = x - 1;
11 assert(x > 0);
```

1: z = z + 3

2: if (z > 0)

3: x = z + 1

6: y = z + 5

4: z = 3 * x

7: x = x * x - z

9: if (x > y)

10: z = x - 1

11: assert(x > 0)

w  r

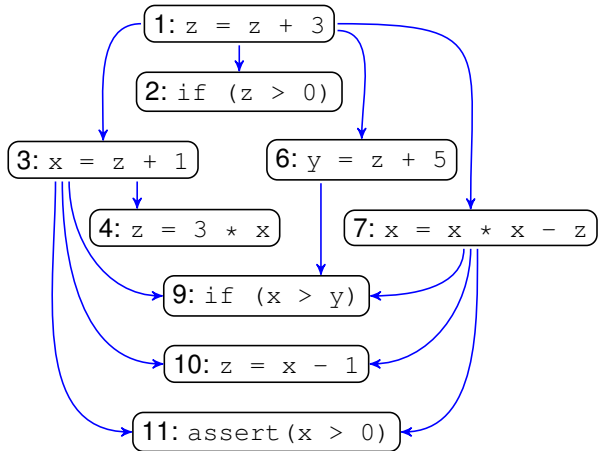
b  n

r is **data dependent** on w if there exists a program execution where r reads a value from a memory that has been written by w

n is **control dependent** on b if b is the closest point where the program may go some way that misses n

Simple dependency graph

```
1 z = z + 3;
2 if (z > 0) {
3   x = z + 1;
4   z = 3 * x;
5 } else {
6   y = z + 5;
7   x = x * x - z;
8 }
9 if (x > y)
10  z = x - 1;
11 assert(x > 0);
```



w  r

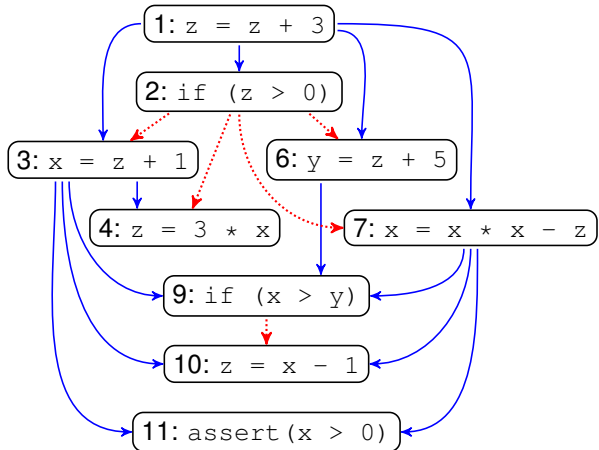
b  n

r is **data dependent** on w if there exists a program execution where r reads a value from a memory that has been written by w

n is **control dependent** on b if b is the closest point where the program may go some way that misses n

Simple dependency graph

```
1 z = z + 3;
2 if (z > 0) {
3   x = z + 1;
4   z = 3 * x;
5 } else {
6   y = z + 5;
7   x = x * x - z;
8 }
9 if (x > y)
10  z = x - 1;
11 assert(x > 0);
```



$w \rightarrow r$

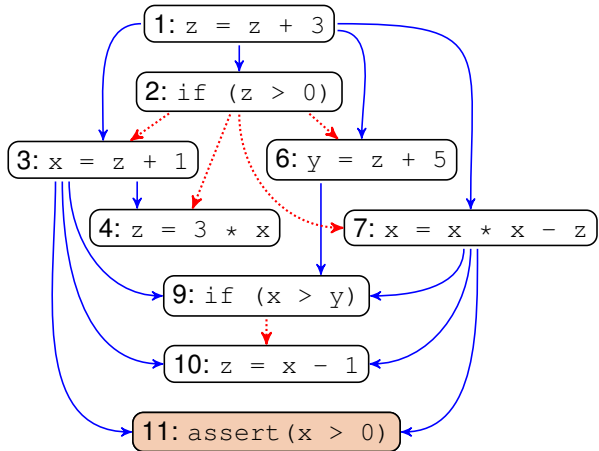
$b \dashrightarrow n$

r is **data dependent** on w if there exists a program execution where r reads a value from a memory that has been written by w

n is **control dependent** on b if b is the closest point where the program may go some way that misses n

Simple dependency graph

```
1 z = z + 3;
2 if (z > 0) {
3   x = z + 1;
4   z = 3 * x;
5 } else {
6   y = z + 5;
7   x = x * x - z;
8 }
9 if (x > y)
10  z = x - 1;
11 assert(x > 0);
```



$w \rightarrow r$

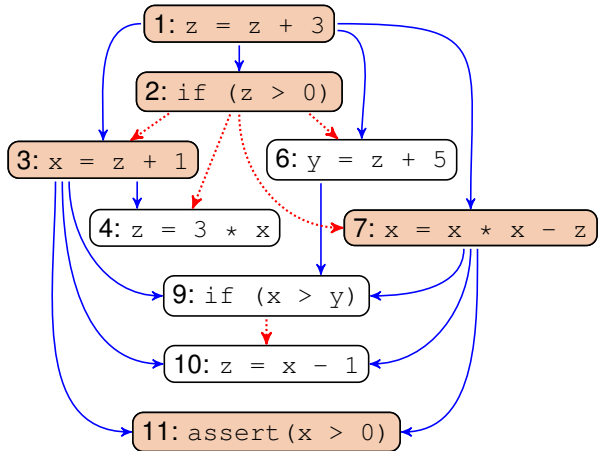
$b \dashrightarrow n$

r is **data dependent** on w if there exists a program execution where r reads a value from a memory that has been written by w

n is **control dependent** on b if b is the closest point where the program may go some way that misses n

Simple dependency graph

```
1 z = z + 3;
2 if (z > 0) {
3   x = z + 1;
4   z = 3 * x;
5 } else {
6   y = z + 5;
7   x = x * x - z;
8 }
9 if (x > y)
10  z = x - 1;
11 assert(x > 0);
```



$w \rightarrow r$

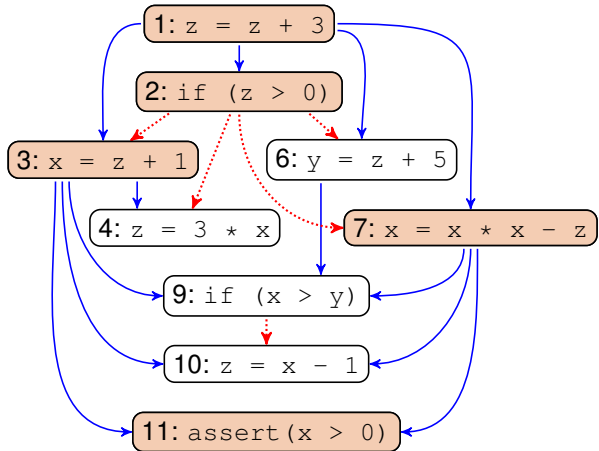
$b \dashrightarrow n$

r is **data dependent** on w if there exists a program execution where r reads a value from a memory that has been written by w

n is **control dependent** on b if b is the closest point where the program may go some way that misses n

Simple dependency graph

```
1 z = z + 3;  
2 if (z > 0) {  
3   x = z + 1;  
4  
5 } else {  
6  
7   x = x * x - z;  
8 }  
9  
10  
11 assert (x > 0);
```



$w \rightarrow r$

$b \dashrightarrow n$

r is **data dependent** on w if there exists a program execution where r reads a value from a memory that has been written by w

n is **control dependent** on b if b is the closest point where the program may go some way that misses n

Points-to analysis aka pointer analysis

How data dependencies look like?

```
1 int x;  
2 int *p;  
3 int *q;  
4 x = 5;  
5 p = &x;  
6 q = p;  
7 *q = 7;  
8 assert(x > 6);
```

How data dependencies look like?

```
1 int x;  
2 int *p;  
3 int *q;  
4 x = 5;  
5 p = &x;  
6 q = p;  
7 *q = 7;  
8 assert(x > 6);
```

⋮

4: x = 5

5: p = &x

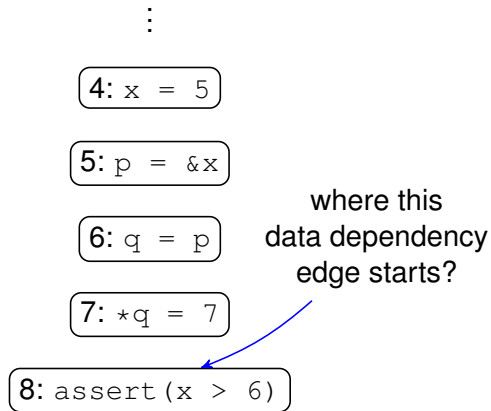
6: q = p

7: *q = 7

8: assert(x > 6)

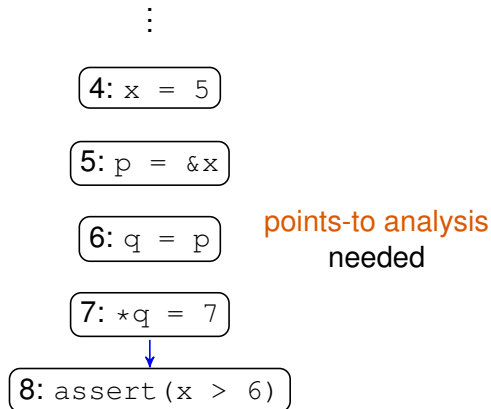
How data dependencies look like?

```
1 int x;  
2 int *p;  
3 int *q;  
4 x = 5;  
5 p = &x;  
6 q = p;  
7 *q = 7;  
8 assert(x > 6);
```



How data dependencies look like?

```
1 int x;  
2 int *p;  
3 int *q;  
4 x = 5;  
5 p = &x;  
6 q = p;  
7 *q = 7;  
8 assert(x > 6);
```



Points-to analysis

- assigns to each pointer p the **points-to set** that contains all memory locations p may point to
- **memory locations** are abstractions of concrete objects located in memory during program execution
 - often identified with allocation statements like `1: int x` or `35: malloc(128)`
 - can represent more concrete objects, e.g., for `malloc` in cycle

Points-to analysis

- assigns to each pointer p the **points-to set** that contains all memory locations p may point to
- **memory locations** are abstractions of concrete objects located in memory during program execution
 - often identified with allocation statements like `1:int x` or `35:malloc(128)`
 - can represent more concrete objects, e.g., for `malloc` in cycle
- we use two additional memory locations
 - `null` representing a pointer value NULL
 - `unknown` saying that the pointer can point anywhere
- additionally, it tracks which memory locations represent one concrete memory object and which are abstract
- can be computed by an abstract interpretation

Points-to analysis

- can be **flow sensitive** or **insensitive**
 - flow sensitive analysis assigns a points-to set to a pointer and a program location (more precise but more expensive)
 - flow insensitive analysis used mainly for programs in **static single assignment (SSA)** form

```
1 int y;  
2 int *data = malloc(40);  
3 ...  
4 int *p = &y;  
5 if (y > 2) {  
6     p = NULL;  
7 } else {  
8     p = data + 2;  
9 }  
10 int *q = p;
```

Points-to analysis

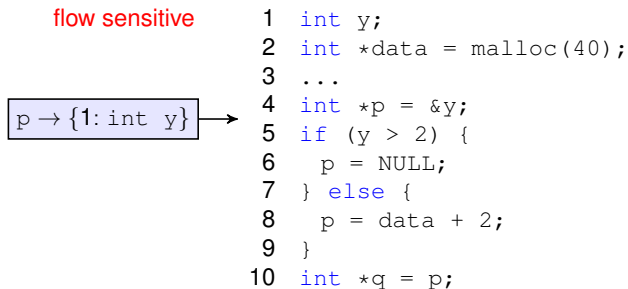
- can be **flow sensitive** or **insensitive**
 - flow sensitive analysis assigns a points-to set to a pointer and a program location (more precise but more expensive)
 - flow insensitive analysis used mainly for programs in **static single assignment (SSA)** form

flow sensitive

```
1 int y;  
2 int *data = malloc(40);  
3 ...  
4 int *p = &y;  
5 if (y > 2) {  
6     p = NULL;  
7 } else {  
8     p = data + 2;  
9 }  
10 int *q = p;
```

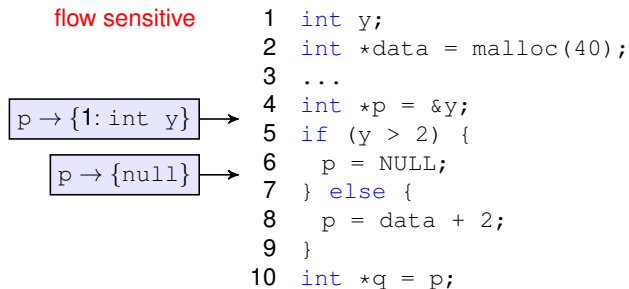
Points-to analysis

- can be **flow sensitive** or **insensitive**
 - flow sensitive analysis assigns a points-to set to a pointer and a program location (more precise but more expensive)
 - flow insensitive analysis used mainly for programs in **static single assignment (SSA)** form



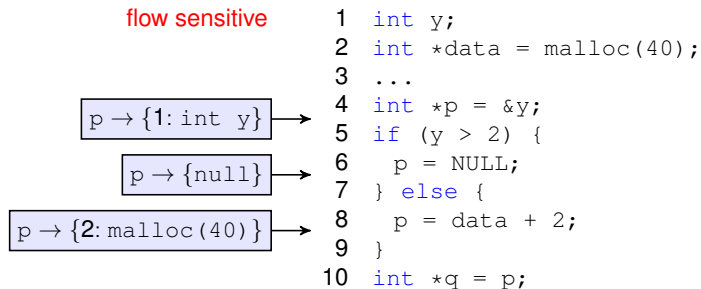
Points-to analysis

- can be **flow sensitive** or **insensitive**
 - flow sensitive analysis assigns a points-to set to a pointer and a program location (more precise but more expensive)
 - flow insensitive analysis used mainly for programs in **static single assignment (SSA)** form



Points-to analysis

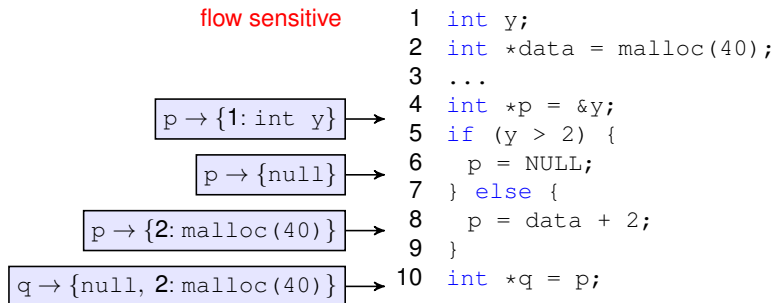
- can be **flow sensitive** or **insensitive**
 - flow sensitive analysis assigns a points-to set to a pointer and a program location (more precise but more expensive)
 - flow insensitive analysis used mainly for programs in **static single assignment (SSA)** form



Points-to analysis

- can be **flow sensitive** or **insensitive**
 - flow sensitive analysis assigns a points-to set to a pointer and a program location (more precise but more expensive)
 - flow insensitive analysis used mainly for programs in **static single assignment (SSA)** form

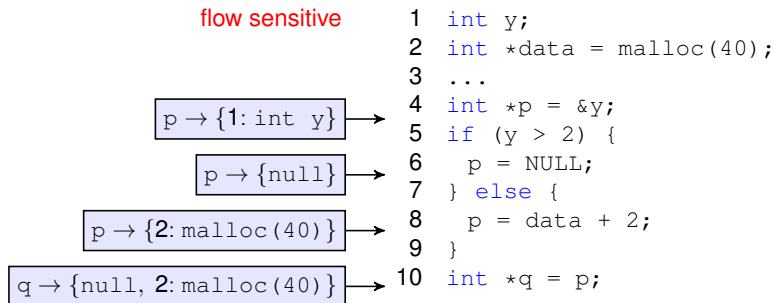
flow sensitive



Points-to analysis

- can be **flow sensitive** or **insensitive**
 - flow sensitive analysis assigns a points-to set to a pointer and a program location (more precise but more expensive)
 - flow insensitive analysis used mainly for programs in **static single assignment (SSA)** form

flow sensitive



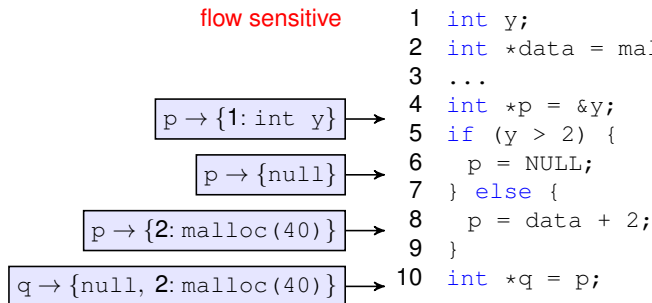
```
1 int y;  
2 int *data = malloc(40);  
3 ...  
4 int *p = &y;  
5 if (y > 2) {  
6     p = NULL;  
7 } else {  
8     p = data + 2;  
9 }  
10 int *q = p;
```

flow insensitive

Points-to analysis

- can be **flow sensitive** or **insensitive**
 - flow sensitive analysis assigns a points-to set to a pointer and a program location (more precise but more expensive)
 - flow insensitive analysis used mainly for programs in **static single assignment (SSA)** form

flow sensitive



$p, q \rightarrow \{1: \text{int } y, \text{null}, 2: \text{malloc}(40)\}$

flow insensitive

Points-to analysis

- can be **flow sensitive** or **insensitive**
 - flow sensitive analysis assigns a points-to set to a pointer and a program location (more precise but more expensive)
 - flow insensitive analysis used mainly for programs in **static single assignment (SSA)** form
- can be **field insensitive** or **sensitive**
 - field sensitive analysis tracks also **offsets**
 - field sensitive analysis is more precise but more expensive

flow sensitive
field insensitive

$p \rightarrow \{1: \text{int } y\}$

$p \rightarrow \{\text{null}\}$

$p \rightarrow \{2: \text{malloc}(40)\}$

$q \rightarrow \{\text{null}, 2: \text{malloc}(40)\}$

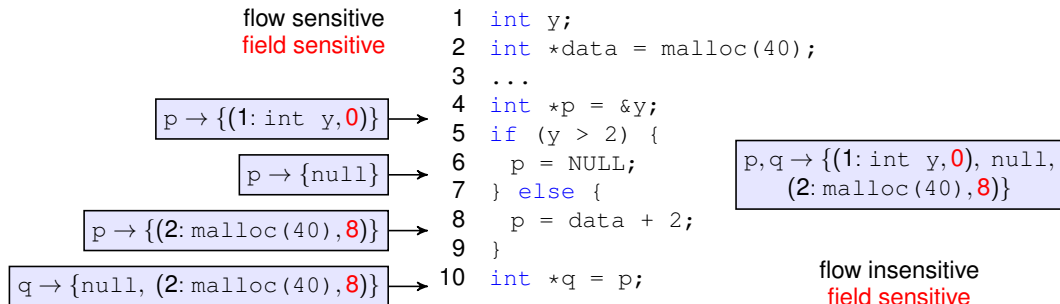
```
1 int y;
2 int *data = malloc(40);
3 ...
4 int *p = &y;
5 if (y > 2) {
6     p = NULL;
7 } else {
8     p = data + 2;
9 }
10 int *q = p;
```

$p, q \rightarrow \{1: \text{int } y, \text{null}, 2: \text{malloc}(40)\}$

flow insensitive
field insensitive

Points-to analysis

- can be **flow sensitive** or **insensitive**
 - flow sensitive analysis assigns a points-to set to a pointer and a program location (more precise but more expensive)
 - flow insensitive analysis used mainly for programs in **static single assignment (SSA)** form
- can be **field insensitive** or **sensitive**
 - field sensitive analysis tracks also **offsets**
 - field sensitive analysis is more precise but more expensive



- a popular algorithm for points-to analysis presented in *L. O. Andersen: Program Analysis and Specialization for the C Programming Language, PhD thesis, 1994*
- applications of points-to analysis
 - can prove that a program is **memory safe**, i.e., it contains no invalid pointer dereference and no invalid memory deallocation
 - can be used for computation of **data dependencies**
 - can help to identify functions called via a function pointer
 - ...

Static single assignment (SSA)

Static single assignment (SSA)

- a program form with only one assignment statement for each variable
- the assignment statement can be evaluated repeatedly
- special instructions called ϕ -nodes added

Static single assignment (SSA)

- a program form with only one assignment statement for each variable
- the assignment statement can be evaluated repeatedly
- special instructions called ϕ -nodes added

```
1  x = input ();
2  z = x + 3;
3  if (z > 0) {
4    x = z + 1;
5    z = 3 * x;
6  } else {
7    z = z + 5;
8  }
9
10
11 z = z + x;
```

Static single assignment (SSA)

- a program form with only one assignment statement for each variable
- the assignment statement can be evaluated repeatedly
- special instructions called ϕ -nodes added

```
1 x = input();
2 z = x + 3;
3 if (z > 0) {
4   x = z + 1;
5   z = 3 * x;
6 } else {
7   z = z + 5;
8 }
9
10
11 z = z + x;
```

```
1 x1 = input();
2 z1 = x1 + 3;
3 if (z1 > 0) {
4   x2 = z1 + 1;
5   z2 = 3 * x2;
6 } else {
7   z3 = z1 + 5;
8 }
9 x3 =  $\phi(x_2, x_1)$ ;
10 z4 =  $\phi(z_2, z_3)$ ;
11 z5 = z4 + x3;
```

- simplifies static analysis
 - without SSA, x may have different values in different locations
 - with SSA, x_i has the same value everywhere
 - flow-insensitive analyses provide better results for programs in SSA
- used in many verification tools and also in compilers
- LLVM IR also uses SSA (sort of)

Data dependence

Consider a fixed **control flow graph (CFG)** with nodes V . We assume that for each node $n \in V$, we have sets:

- $sdef(n)$ of memory locations that **must** be written by n
- $wdef(n)$ of memory locations that **may** be written by n
- $ref(n)$ of memory locations that **may** be read by n

Consider a fixed **control flow graph (CFG)** with nodes V . We assume that for each node $n \in V$, we have sets:

- $sdef(n)$ of memory locations that **must** be written by n
 - $wdef(n)$ of memory locations that **may** be written by n
 - $ref(n)$ of memory locations that **may** be read by n
-
- $null, unknown \notin sdef(n)$ and $null \notin wdef(n) \cup ref(n)$
 - $sdef(n) \subseteq wdef(n)$
 - $sdef(n)$ contains only memory locations that represent one concrete object each time n is executed
 - the sets can be computed by a field-sensitive points-to analysis

Definition (data dependence)

Let V be the set of nodes of a CFG. A node $n_r \in V$ is **data dependent** on a node $n_w \in V$ if there is a path $n_w = n_1, n_2, \dots, n_k = n_r$ in the CFG such that

- $\text{unknown} \notin wdef(n_w) \cup ref(n_r)$ and $wdef(n_w) \cap ref(n_r) \not\subseteq \bigcup_{1 < i < k} sdef(n_i)$ or
- $\text{unknown} \in wdef(n_w)$ and $ref(n_r) \not\subseteq \bigcup_{1 < i < k} sdef(n_i)$ or
- $\text{unknown} \in ref(n_r)$ and $wdef(n_w) \not\subseteq \bigcup_{1 < i < k} sdef(n_i)$.

Definition (reaching definition)

Consider a node n_w and $e \in wdef(n_w)$. We say that the **definition** of e at n_w **reaches** a node n if there is a path $n_w = n_1, n_2, \dots, n_k = n$ and $e \notin \bigcup_{1 < i < k} sdef(n_i)$.

Definition (reaching definition)

Consider a node n_w and $e \in wdef(n_w)$. We say that the **definition** of e at n_w **reaches** a node n if there is a path $n_w = n_1, n_2, \dots, n_k = n$ and $e \notin \bigcup_{1 < i < k} sdef(n_i)$.

- reaching definitions can be computed by an abstract interpretation
- `unknown` $\in wdef(n_w)$ reaches all nodes reachable from n_w

Definition (reaching definition)

Consider a node n_w and $e \in wdef(n_w)$. We say that the **definition** of e at n_w **reaches** a node n if there is a path $n_w = n_1, n_2, \dots, n_k = n$ and $e \notin \bigcup_{1 < i < k} sdef(n_i)$.

- reaching definitions can be computed by an abstract interpretation
- `unknown` $\in wdef(n_w)$ reaches all nodes reachable from n_w

Theorem

If n_r is data dependent on n_w , then

- *the definition of some $e \in wdef(n_w)$ at n_w reaches n_r and $e \in ref(n_r)$, or*
- *`unknown` $\in wdef(n_w) \cup ref(n_r)$ and $wdef(n_w) \neq \emptyset$ and $ref(n_r) \neq \emptyset$.*

- the previous theorem allows to compute an overapproximation of data dependencies with use reaching definitions
- computation is relatively slow because it computes more information than needed
- there are faster algorithms, e.g., **byte-memory SSA** algorithm presented in *M. Chalupa: Program Slicing and Symbolic Execution for Verification, PhD thesis, 2021*

Control dependence

Which statements are irrelevant for the `assert`?

```
1 unsigned int i,n;  
2 n = input();  
3 i = 0;  
4 while (i < n) {  
5     i++;  
6 }  
7 assert(false);
```

```
1 unsigned int i,n;  
2 n = input();  
3 i = 0;  
4 while (i >= n) {  
5     i++;  
6 }  
7 assert(false);
```

Which statements are irrelevant for the `assert`?

```
1  
2  
3  
4  
5  
6  
7 assert(false);
```

```
1 unsigned int i, n;  
2 n = input();  
3 i = 0;  
4 while (i >= n) {  
5     i++;  
6 }  
7 assert(false);
```

Which statements are irrelevant for the `assert`?

```
1  
2  
3  
4  
5  
6  
7 assert(false);
```

```
1 unsigned int i, n;  
2 n = input();  
3 i = 0;  
4 while (i >= n) {  
5     i++;  
6 }  
7 assert(false);
```

- removing a potentially non-terminating cycle can transform an unreachable code into a reachable
- line 7 on the right is unreachable if `input()` always returns 0

Two notions of control dependence

Intuitively, a statement n is **control dependent** on a statement b if b is the closest point where the program may go some way that misses n .

Two notions of control dependence

Intuitively, a statement n is **control dependent** on a statement b if b is the closest point where the program may go some way that misses n .

weak control dependence

- assumes that every execution is finite
- an instance: **standard control dependence**

Two notions of control dependence

Intuitively, a statement n is **control dependent** on a statement b if b is the closest point where the program may go some way that misses n .

weak control dependence

- assumes that every execution is finite
- an instance: **standard control dependence**

strong control dependence

- sensitive to program non-termination: there can be a dependence between two statements if one can infinitely delay the execution of the other
- an instance: **non-termination sensitive control dependence**

Standard control dependence

An **exit-CFG** is a CFG with a unique **exit** node that is reachable from every other node.

Standard control dependence

An **exit-CFG** is a CFG with a unique **exit** node that is reachable from every other node.

Definition (post-dominance)

Given an exit-CFG, its node b **post-dominates** a node a if b is on every path from a to exit. If $a \neq b$, we say that b **strictly post-dominates** a .

Standard control dependence

An **exit-CFG** is a CFG with a unique **exit** node that is reachable from every other node.

Definition (post-dominance)

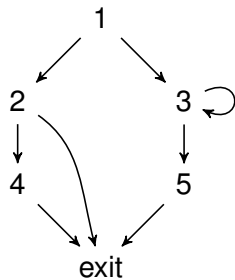
Given an exit-CFG, its node b **post-dominates** a node a if b is on every path from a to exit. If $a \neq b$, we say that b **strictly post-dominates** a .

Definition (standard control dependence)

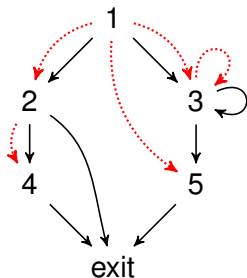
Given an exit-CFG, we say that node n is **standard control dependent (SCD)** on node b if

- 1 there exists a non-trivial path π from b to n with any node on π (excluding b) post-dominated by n and
- 2 b is not strictly post-dominated by n .

Standard control dependence



Standard control dependence



- the SCD relation for an exit-CFG (V, E) can be computed in time $\mathcal{O}(|E|)$ using the algorithm of *J. Ferrante et al.: The Program Dependence Graph and Its Use in Optimization, TOPLAS 1987*
- each CFG can be transformed into an exit-CFG

Non-termination sensitive control dependence

- **predicate nodes** in CFG are nodes corresponding to branching statements
- **maximal path** is a path that cannot be further prolonged, i.e., it is infinite or it ends in a node without any successor

Non-termination sensitive control dependence

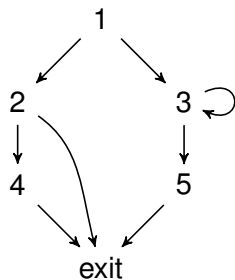
- **predicate nodes** in CFG are nodes corresponding to branching statements
- **maximal path** is a path that cannot be further prolonged, i.e., it is infinite or it ends in a node without any successor

Definition (non-termination sensitive control dependence)

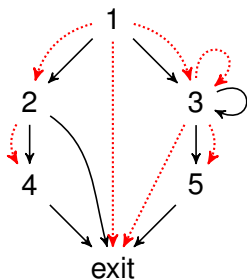
Given a CFG, a node n is **non-termination sensitive control dependent (NTSCD)** on a predicate node p if p has two successors s_1 and s_2 such that

- 1 all maximal paths from s_1 contain n and
- 2 there exists a maximal path from s_2 that does not contain n .

Non-termination sensitive control dependence



Non-termination sensitive control dependence



- the NTSCD relation for a CFG (V, E) can be computed in time $\mathcal{O}(|V|^2)$ using the algorithm of *M. Chalupa et al.: [Fast Computation of Strong Control Dependencies](#), CAV 2021*
- NTSCD treats every program cycle as potentially non-terminating

- we used **program dependence graphs** for programs without procedure calls
- there are also **system dependence graphs** for programs with procedure calls
- both NTSCD and SCD have applications in program slicing for software verification: SCD leads to smaller sliced programs and can only lead to produce false alarms, but not to false negatives
- there are other notions of control dependence, e.g., **decisive order dependence (DOD)**
- points-to analysis and slicer for LLVM implemented in **DG**
`https://github.com/mchalupa/dg`