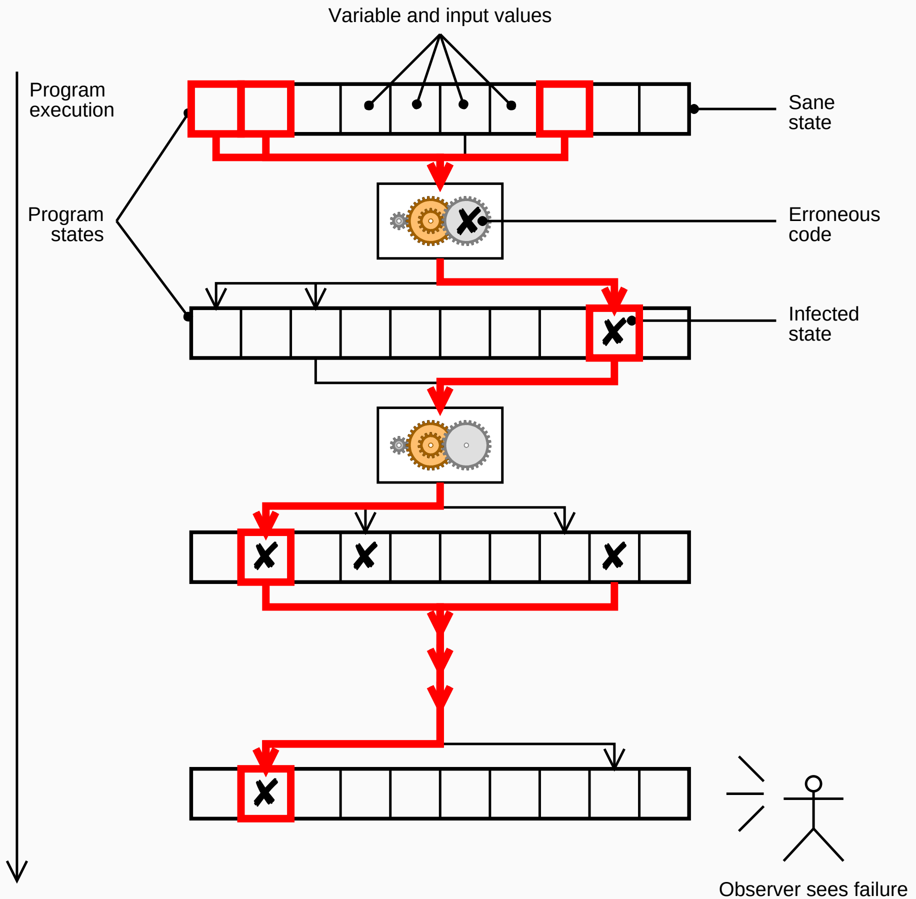
Isolating Value Origins

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Isolating Origins



Thinking Backwards

Besides temporal and spatial focusing, one can also *trace back* the origin of an infection.

> *Something impossible occurred,*
> *and the only solid information is that it really did occur.*
> *So we must think backwards from the result*
> *to discover the reasons.*
>
> — *Brian W. Kernighan and Rob Pike,*
> *THE PRACTICE OF PROGRAMMING*

Dependencies

Consider the following piece of code:

```
if (p)
    z = x * f(y);
```

Assume we find that z is infected (say, z = 0 holds).

These are the potential causes for z being zero:

- x may be zero
- f() may have returned zero
- p may be true (instead of false)

Hence, the value of z *depends* on the statements in which these values have been set.

Program Slicing

Again, we'd like to automate this as much as possible.

Basic idea: *Follow the dependencies through the program to narrow down the set of potential value sources.*

This pattern is called PROGRAM SLICING.

A *program slice* is a subset of the program's statements.

We distinguish two kinds of program slices:

Backward slice The statements that *may have affected* a specific variable

Forward slice The statements that *may be affected* by a specific variable

Backward Slice

Backward slice: All statements that *may have affected* a variable at a specific place in the program.

Program

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Backward slice for (mul, 13)

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }

  write(mul);
}
```

Main use: *Where does this value come from?*

Forward Slice

Forward slice: All statements that *may be affected* by a variable at a specific place in the program.

Program

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Forward slice for (b, 6)

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Main use: *Where does this value go to?*

Dependencies

The relationship between a program state at a specific statement and the statements that may have caused it is called a *dependency*.

There are two kind of dependencies:

Control dependency Some statement A is *control dependent* on a statement B if A might affect if or how often B is executed.

Data dependency Some statement A is *data dependent* on a statement B if A assigns a value to a variable that is being accessed in B.

Such dependencies are computed by *program analysis.*

Dependencies (2)

Consider the following piece of code:

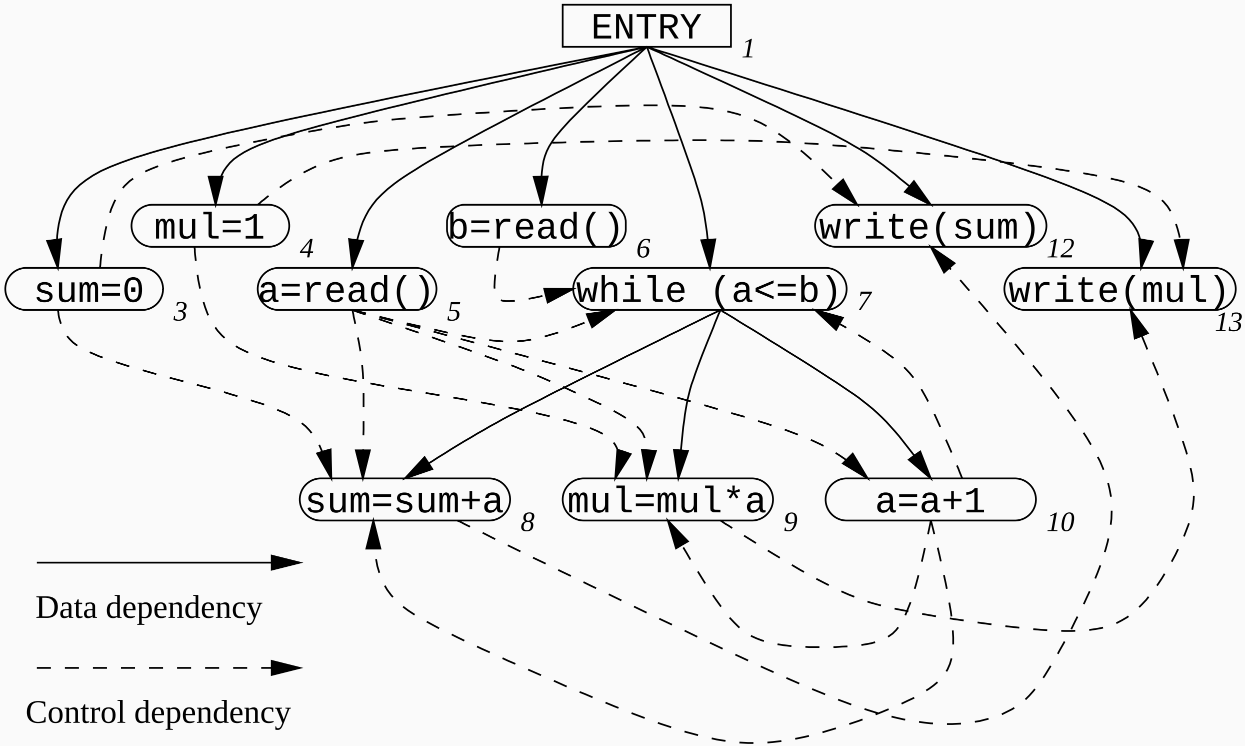
```
if (p)
    z = x * f(y);
```

What does z depend upon?

- z is *control dependent* on `if (p)`
- z is *data dependent* on the assignment(s) of x
- z is *data dependent* on the return value of `f()`

Program slicing collects these dependencies in a *program dependency graph.*

Program Dependency Graph



Dependencies and Causes

Dependencies are only *potential causes.*

Let us extend the code somewhat:

```
x = 0;
if (p)
    z = x * f(y);
```

Obviously, `f()` can no longer be a cause for x being zero, because there is no alternate value that `f()` could return.

Dependencies and Causes (2)

What's "obvious" depends on the *smartness* of the analysis:

```
q = !p; x = p && q;
if (p)
    z = x * f(y);
```

"Obviously", x is always zero—and hence, only p and x are potential causes for z being zero.

It is obvious, too, though, that there are computational limits on the dependencies we can compute.

Conservative Approximation

Program analysis methods are *conservative,* because they conserve the program semantics:

In doubt, a program analysis tool will always produce a dependency.

Only if it can be *proven* that there is *no way* a statement might influence the state, then there is no dependency.

```
q = !p; x = p && q;
if (p)
    z = x * f(y);
```

z does not depend on a and b (unless `f()` would access them); hence, there is no dependency to any assignments of a or b.

Conservative Approximation (2)

The produced dependencies are always *approximations* of the full program behavior, as in *points-to analysis:*

```
T *p, *q, *r;
int main() {
    p = new T;
    f();
    g(&p);
    p = new T;
    ? = *p;
}
```

```
void f() {
    q = new T;
    g(&q);
    r = new T;
}
```

```
void g(T **fp) {
    T local;
    if (...)
        *fp = &local;
    ...
}
```

Simple approximation (faster): *A pointer can point to anything whose address is taken*—p may point to all new T's, p, q and `local` (which thus all depend on p)

Smarter approximation (more precise): consider *data flow* and *control flow*—pointer p points to the last new T.

Limits of Program Slicing

Since program analysis is approximative, the possible influences *multiply* the further you move away from the variable in question ("slicing is short-sighted").

Nonetheless, there will always be a substantial amount of code that *cannot* influence the variable in any (legal) way—which means that debugging is considerably easened.

An open problem is *complexity:*

- associated program analysis is expensive if smart
- intraprocedural slicing is expensive if smart

Dynamic Slicing

One way to overcome the "short-sightedness" of program slicing is to consider *only one specific run* (rather than all possible runs)—for instance, the run we want to understand.

This is called *dynamic* (rather than *static*) slicing, because the analysis *executes* the program in question.

Using a concrete run as reference, we can easily determine

- where pointers point to
- which statements where executed at all
- the sequence of executed statements

All this makes the program analysis *more precise*.

Static vs. Dynamic Slicing

Static slice for s

```
n = read();
a = read();
x = 1;
b = a + x;
a = a + 1;
i = 1;
s = 0;
while (i <= n) {
    if (b > 0)
        if (a > 1)
            x = 2;
    s = s + x;
    i = i + 1;
}
write(s);
```

Dynamic slice

```
n = read(); // 2
a = read(); // 0
x = 1;
b = a + x;
a = a + 1;
i = 1;
s = 0;
while (i <= n) {
    if (b > 0)
        if (a > 1)
            x = 2;
    s = s + x;
    i = i + 1;
}
write(s);
```

Computing Dynamic Slices

Basic idea: *Attach actual sources to every value*

1. Compute a *definition/use table* which stores the defined and used variable values.
2. Assign a dynamic slice to each value definition (initially empty).
3. Execute the program.
4. Whenever a value is defined, assign its slice the *union* of all used slices.
5. At the end of the execution, all definitions will be assigned a slice that holds all value sources.

Def/Use Table

Code	Def	Use
`n = read();`	n	
`a = read();`	a	
`x = 1;`	x	
`b = a + x;`	b	a, x
`a = a + 1;`	a	a
`i = 1;`	i	
`s = 0;`	s	
`while (i <= n) {`	p8	i, n
`    if (b > 0)`	p9	b, p8
`        if (a > 1)`	p10	a, p9
`            x = 2;`	x	p10
`    s = s + x;`	s	s, x, p8
`    i = i + 1;`	i	i
`}`		
`write(s);`	o14	s

Computing the Slices

$$DynSlice(d) = \bigcup_i (DynSlice(u_i) \cup line(u_i))$$

Code	Def	Use	DynSlice
`1 n = read();`	n		
`2 a = read();`	a		
`3 x = 1;`	x		
`4 b = a + x;`	b	a, x	**2, 3**
`5 a = a + 1;`	a	a	**2**
`6 i = 1;`	i		
`7 s = 0;`	s		
`8 while (i <= n) {`	p8	i, n	**6, 1**
`9     if (b > 0)`	p9	b, p8	**2, 3, 6, 1, 4, 8**
`10         if (a > 1)`	p10	a, p9	**2, 3, 6, 1, 4, 8, 5, 9**
`12     s = s + x;`	s	s, x, p8	**6, 1, 7, 3, 8**
`13     i = i + 1;`	i	i	**6, 1, 8**
`8 while (i <= n) {`	p8	i, n	6, 1, **8, 13**
`9     if (b > 0)`	p9	b, p8	2, 3, 6, 1, 4, 8, **13**
`10         if (a > 1)`	p10	a, p9	2, 3, 6, 1, 4, 8, 5, 9, **13**
`12     s = s + x;`	s	s, x, p8	6, 1, 7, 3, 8, **13, 12**
`13     i = i + 1;`	i	i	6, 1, 8, **13**
`8 while (i <= n) {`	p8	i, n	6, 1, 8, 13
`14 write(s);`	o14	s	**6, 1, 7, 3, 8, 13, 12**

Efficiency

Dynamic slicing (as presented here) is quite efficient:

- Case study: While a static slice contains 58% of the statements, a dynamic slice cuts this down to 5%
- Set unions can be implemented with (nearly) constant complexity
- Program execution is slowed down by instrumentation ($\sim$ 2–10 times slower)

What's in a Slice?

A dynamic slice may not contain the erroneous statement:

Static slice for s

```
n = read();
a = read();
x = 1;
b = a + x;
a = a + 1;
i = 1;
s = 0;
while (i <= n) {
    if (b > 0)
        if (a > 1)
            x = 2;
    s = s + x;
    i = i + 1;
}
write(s);
```

Dynamic slice

```
n = read(); // 2
a = read(); // 0
x = 1;
b = a + x;
a = a + 1; // error?
i = 1;
s = 0;
while (i <= n) {
    if (b > 0) // true
        if (a > 1) // false
            x = 2;
    s = s + x;
    i = i + 1;
}
write(s);
```

Dynamic vs. Relevant Slicing

A *relevant slice* includes *conditional* (static) dependencies:

Dynamic slice for s

```
n = read(); // 2
a = read(); // 0
x = 1;
b = a + x;
a = a + 1;
i = 1;
s = 0;
while (i <= n) {
    if (b > 0) // true
        if (a > 1) // false
            x = 2;
    s = s + x;
    i = i + 1;
}
write(s);
```

Relevant slice

```
n = read(); // 2
a = read(); // 0
x = 1;
b = a + x;
a = a + 1; // error may also be here...
i = 1;
s = 0;
while (i <= n) {
    if (b > 0)
        if (a > 1)  // ... or here
            x = 2;
    s = s + x;
    i = i + 1;
}
write(s);
```

Approach: include *static dependencies* for *alternative control flows* (like `(a > 1)`)

Slices and Erroneous Statements

The idea of relevant slices is certainly useful:

- Include all statements that in the slice that, if altered, may change the variable value in question.

But this alteration must still preserve the original definition/use!

If we allow *arbitrary* alterations (i.e. the statement can be changed to anything else), then *every statement* can be a cause for the variable value.

This is especially true for *missing statements* that can be inserted anywhere.

Consequence: *Slices need not include "the" error* (but are helpful in understanding how the error came to be!)

Dicing

Dynamic slices can be very useful if *two program runs* exist:

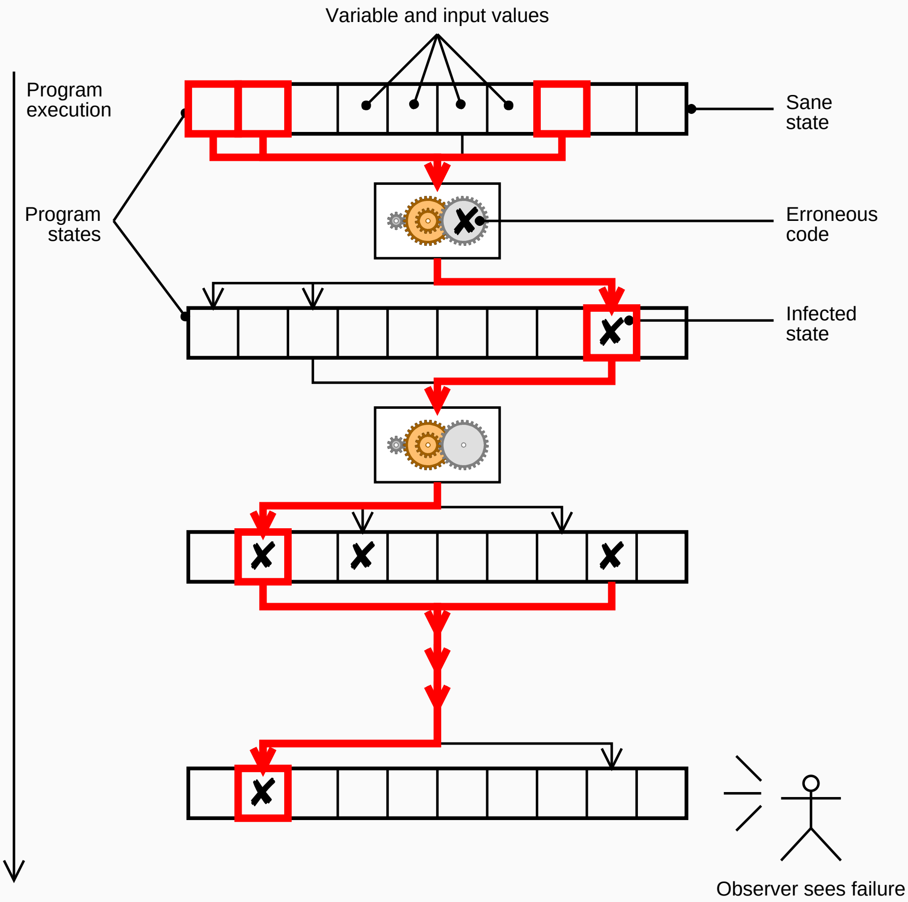
- A run where the failure occurs
- A run where the failure *does not* occur

Basic idea:

1. Compute the slice for the *passing* value
2. Compute the slice for the *failing* value
3. Examine the *difference* (the *dice*) between the two slices

The difference should contain all statements that influence *only* the failing value—i.e. potentially erroneous statements.

Isolating Origins



Concepts

- A *static slice* contains all statements that may affect (or may be affected by) a variable value at a specific statement.
- A *dynamic slice* is specific to a concrete program run.
- A *relevant slice* also includes control flow alternatives.
- A slice need not include the erroneous statement...
- ...but is helpful in tracing down value origins.
- A *dice* is the difference between a passing and a failing slice; it can pinpoint potentially erroneous statements.

References

- Frank Tip, *A survey of program slicing techniques*, Journal of programming languages 3, 121–189 (1995).
  http://citeseer.nj.nec.com/tip95survey.html
- T. Gyimóthy, Árpád Beszédes and Istvan Forgács, *An efficient relevant slicing method for debugging*, Proc. ESEC/FSE 99.
  http://doi.acm.org/10.1145/318773.319248
  (free access from .cs.uni-sb.de)