

## Static Analysis Tools

Lecture 18  
CS 169

## Announcements

- Discussion sessions this week
- Homework due Wed
- A total of 6 homeworks this semester
  - See the schedule online

## Static Analysis

- Analyze the program at compile time
  - Without running the program
- Two sorts
  - Syntactic analyses
    - Did you follow the indentation standard?
  - Semantic analyses
    - Deeper understanding of the program
    - Type systems
    - And others . . .

## Types for Data Races

- Discipline for preventing races
  - Each shared global is protected by one lock
  - The lock must be held at each access
- Example:

```
lock balanceLock;
int balance; /*@ protected by balanceLock @*/
...
synchronized(balanceLock) { balance = balance - 1 }
```
- We adapt the type checking algorithm to check that we follow the locking discipline

## Type Environment for Data Races

- At an occurrence of a variable, we must know:
  - What locks are supposed to protect the variable?
    - We use a variable environment
    - $V$  contains pairs  $(x, L)$ , where  $L$  is the declared lock for  $x$
    - A variable appears in  $V$  only if it is protected
  - What locks are held?
    - We add another element of context: the locks held
- $check(V, LS, E)$ 
  - $E$  obeys the lock discipline, assuming the locking declarations in  $V$ , and that the locks in  $LS$  are held

## Typing Rules for Race Detection

### Rules for expressions

- ```
check(V, LS, n)
- Always
check(V, LS, E1 + E2)
- If check(V, LS, E1) && check(V, LS, E2)
check(V, LS, x)
- If  $x \notin V$ , or
- If  $(x, lock) \in V$  and  $lock \in LS$ 
```

## Typing Rules for Race Detection (II)

$\text{scheck}(V, LS, S)$  - check a statement

Rules:

$\text{scheck}(V, LS, x = E)$

- If  $\text{check}(V, LS, E)$ , and
- $x \notin V$  or  $[(x, \text{lock}) \exists V \text{ and } \text{lock} \exists LS]$

$\text{scheck}(V, LS, \text{synchronized}(\text{lock}) S)$

- If  $\text{lock} \notin LS$  &&  $\text{scheck}(V, LS + \text{lock}, S)$

- $V$  is initialized based on global declarations

Prof. Brewer CS 169 Lecture 18

7

## Type Checking Example

- Consider the code

|                                  |            |           |
|----------------------------------|------------|-----------|
|                                  | V          | LS        |
|                                  | {}         | {}        |
| int b; /*@ protected by bLock */ | (b, bLock) | {}        |
| synchronized(bLock) {            | (b, bLock) | { bLock } |
| temp = b                         | (b, bLock) | { bLock } |
| b = temp - 1                     | (b, bLock) | { bLock } |
| }                                | (b, bLock) | { bLock } |
| b = 0                            | (b, bLock) | { } !!!   |

Prof. Brewer CS 169 Lecture 18

8

## Comments

- This approach actually works in practice
- It does require programmers to declare (and follow) the locking strategy
- It rules out some "smart" locking schemes
  - Lock in the caller and unlock in the callee
  - Intentional races, optimistic locking

Prof. Brewer CS 169 Lecture 18

9

## Splint - Static Analysis for C

Prof. Brewer CS 169 Lecture 18

10

## C Lint

- Modern C compilers check for many questionable constructs

$\text{if}(a = b) \dots$

*should probably be  $a == b$*

$\text{if}(a == 1 \ \& \ b == 2) \dots$

*should probably be  $a == 1 \ \& \ b == 2$*

$\text{if}(x \ll y + 2) \dots$

*should probably be  $x \ll (y + 2)$*

Prof. Brewer CS 169 Lecture 18

11

## C Lint (Cont.)

- Where did this idea come from?
- C Lint
  - Stand-alone program
  - Checked for many such common errors
  - Written in 1979
- Named after the bits of fluff it picks from programs

Prof. Brewer CS 169 Lecture 18

12

## Splint - Philosophy

- Secure programming Lint
  - Follow-up to Lint, LCLint
- Philosophy
  - C programs have lots of bugs
  - Due to language weakness
  - Emphasis on performance over safety (from 1970s)
- Replacing C would be hard, but we can build tools to warn against unsafe programming practices

Prof. Brewer CS 169 Lecture 18

13

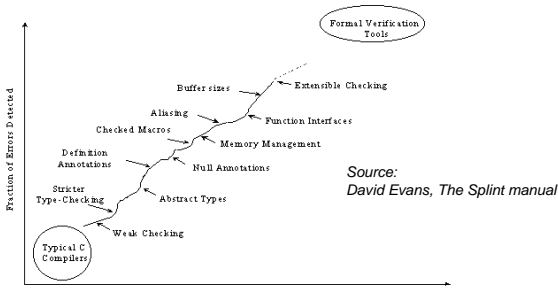
## What Can Splint Detect?

- Subtle type errors
- Abstract type violations
- Dangerous use of macros
- Dereferencing a null pointer
- Memory management errors
- And others ...

Prof. Brewer CS 169 Lecture 18

14

## What Can Splint Do?



Prof. Brewer CS 169 Lecture 18

15

## Stricter Type Checking

- `enum`, `char` are separated from `int`
- Tighter checking of signed/unsigned
- Examples of problems detected:
  - Assign an integer to `char`, or `enum`
  - Assign an unsigned to a signed value

Prof. Brewer CS 169 Lecture 18

16

## Stricter Type Checking (Cont.)

- Splint adds a `bool` type to C
- Checks that predicates have type `bool`
  - E.g., `x == y`, `P && P`, `!P`, but not `x = y`, `P & P`
- Ensures that only `bool` expressions are used in predicates: `if`, `while`, `do`
- Catches many classic errors
- This was fixed in Java

Prof. Brewer CS 169 Lecture 18

17

## Abstract Types

- Abstraction: hide concrete implementation details
  - Easier to understand code
  - Effects of changes can be localized
- Splint uses `typedef` to introduce abstraction

```
typedef /*@abstract@*/ char *MSTRING;
```

  - Clients should use `MSTRING` abstractly
    - without knowledge that it is a synonym for `char*`
  - Can declare functions/modules that have access to the concrete implementation

Prof. Brewer CS 169 Lecture 18

18

## Unused Function Return Values

---

- Many C functions return a special value to signal error
  - It is a mistake to forget to check the return
- Common examples: `read(...)`, `close(...)`
- Splint warns when you do not use the return value of a non-void function

Prof. Brewer CS 169 Lecture 18

19

## Control Flow Issues

---

- C has very loose ordering requirements for evaluation of expression
  - E.g., `y = x++ * x` may result in `y = x2` or `y = x2 + x`
- C allows automatic fallthrough in switch
  - Almost never the intended behavior
- Splint watches for conditionals with no { ... }
  - E.g., `if(...) x++; y ++;`
- Splint detects statement with no effects
  - E.g., `y == *x`

Prof. Brewer CS 169 Lecture 18

20

## Macros are Dangerous

---

- Function-like macros
  - `#define square(x) x * x`
- Macro bodies that are not expressions
  - `#define inc() x++; y ++`
  - `#define incbound() x++; if(x >= 10) x = 0`
- Splint has a number of annotations to control macro checking
  - E.g., `/*@sef@*/` (side-effect free arguments)

Prof. Brewer CS 169 Lecture 18

21

## Splint So Far

---

- Many syntactic checks
- Some type-based checks
- For efficiency, require sufficient information on functions to typecheck the body
  - Forces annotations on function prototypes

Prof. Brewer CS 169 Lecture 18

22

## Flow Sensitivity

---

- Splint described so far is flow insensitive
- Types cannot change
  - The type of a value cannot change
  - The type of a variable is the same for entire scope
- This is often not enough ...

Prof. Brewer CS 169 Lecture 18

23

## Opinions about Splint

---

- Splint is really useful
  - Splint-like tools exist for many languages
- Catches a number of hard-to-find bugs
- But, you must be patient
  - Splint will complain about many things that don't matter, at least in your program
  - You have to wade through the spurious warnings to find the few items of real interest
  - Add annotations

Prof. Brewer CS 169 Lecture 18

24

## Limitations of Static Analyses

---

- Static analyses produce two messages:
  - *Warnings*: I think this is wrong
  - *Errors*: I know this is wrong
- Warnings are far more common
  - *False positive*: a spurious warning
    - There is no bug in the code
    -
  - High false positive rates are not unusual
- Next time: analyses with fewer false positives