Announcements

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

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:
  ```java
  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
    - 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 ∈ V, or
  - If (x, lock) ∈ V and lock ∈ LS

Typing Rules for Race Detection (II)

\texttt{scheck(V, LS, S)} – check a statement

Rules:
\texttt{scheck(V, LS, x = E)}
- If \texttt{check(V, LS, E)}, and
- \( x \not\in V \) or \([x, \text{lock}] \not\in V \) and \( \text{lock} \not\in LS \)
\texttt{scheck(V, LS, synchronized(lock) S)}
- If \( \text{lock} \not\in LS \) && \texttt{scheck(V, LS + lock, S)}

\( V \) is initialized based on global declarations

Type Checking Example

- Consider the code

\begin{verbatim}
int b; /*@ protected by bLock */
synchronized(bLock) {
    temp = b
    b = temp - 1
}
\end{verbatim}

\begin{verbatim}
\begin{array}{c|c}
V & LS \\
\hline
\{\} & \{\} \\
(b, bLock) & \{\} \\
(b, bLock) & \{\text{lock}\} \\
(b, bLock) & \{\text{lock}\} \\
(b, bLock) & \{\} !!! \\
\end{array}
\end{verbatim}

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

Splint – Static Analysis for C

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

What Can Splint Detect?

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

What Can Splint Do?

- 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

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

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
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

Control Flow Issues

- C has very loose ordering requirements for evaluation of expression
  - E.g., \( y \times x++ \times \) may result in \( y = x^2 \) or \( y = x^2 + x \)
- 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., \( y == *x \)

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)

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

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 ...

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
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