Program Checking

Lecture 11
CS195

The Idea

• Software should check its own work
  - When the answer is computed, check that it is the correct answer
• Just as we check our own work...
  - Do sums forwards and backwards

What Does It Mean to Check?

• Say f(x) = y

• How can we verify y = f(x)?
  - Want to do this quickly
  - Completely (not a partial check)

The Definition

• $C$ is a simple checker for $f$ iff:
  • $C$ is asymptotically faster than $f$
  • $C(x,y) = true \iff f(x) = y$
    - With high probability, if $C$ is randomized

Explanation

• Why should $C$ be faster than $f$?

• Two reasons:
  - Practicality
    - Asymptotically, checker has negligible cost
  - Theory
    - Forces the checker to be different from the program

Example: Factoring Integers

• Problem: factor a large integer
  - Believed to be very hard

• Checker
  - Multiply factors together

• More generally, any problem in NP has a PTime checker.
Another Example

- Sort an array of numbers
- How do we check this? Need two things:
  - Elements are ordered
  - Elements are a permutation of original array
  - Compute checksum of both arrays and compare
- Sorting is $O(n \log n)$, but checker is $O(n)$

And Another Example

- Problem:
  - Is $k$ in sorted array $A$?
- Algorithm:
  - Binary search, return yes/no
- Problem:
  - This is not efficiently checkable
- Solution:
  - Change the output to give index into array
  - This gives a constant time check

Interface

- The last example is instructive
  - Sometimes we must augment the output with extra information to enable efficient checking
- This is a general technique
  - What can I add to the output to make it verifiably correct?

And Another Example

- Translation validation
  - Is my compiled program faithful to the original?
  - A very widespread, low-level problem
  - Limits what compiler writers will attempt
  - A research subarea of its own (recent)

Translation Validation Sketch

- A compiler proves to itself that the source and target programs are equivalent
- Make this proof explicit
  - And part of the output
- Proof checking is relatively easy
  - In contrast to proof discovery

Randomness

- Randomization often gives very simple and fast checkers
- Problem: Calculate $A \times B = C$
  - Let $r$ be a randomly chosen small prime
  - Check $((A \mod r) \times (B \mod r)) \mod r = C \mod r$
Why Does this Work?

- \[((A \mod r) \times (B \mod r)) \mod r = C \mod r\]
- If \(A \times B\) does equal \(C\), then the \(=\) clearly holds
- If \(A \times B = C'\), then \(C \mod r \neq C' \mod r\) with high probability
- Small number multiplies/mod can be done fast
- Remember this when we discuss the Pentium bug

Comparison with Other Techniques

- Verification
- Assert
- Testing
- "Fault tolerance"

Verification

- Verification proves correctness for all inputs
  - Before the program is run in production
- Checking proves correctness on one input
  - The one we care about: the current one
- But verification is largely a strawman
  - Full verification is only used in special situations

Assert Programming

- Many programmers use asserts
  - Really, the culture of checking
- But checking is different!
  - Rigorous: checking correctness
  - Time bounds: many asserts are slow

Testing

- Testing is arguably less effective alone than checking used alone
- Checking is
  - Automatic
  - Runs every time
  - Rigorous (says yes/no correctly)
    - At least if checker is correct
- But in reality we want both
  - Checking makes the test suite more effective, and vice versa

Digression: The Pentium Bug

- To produce Pentium, Intel used at least
  - Verification
    - Automatic compilation of high-level equations to lower levels of circuit design
  - Testing
    - Presumably very intensive
- But neither approach found the bug
  - Checking would have found this one
An Example of the Pentium Bug

\[ x = 4195835 \]
\[ y = 3145727 \]
\[ z = x - (x/y) \times y \]

Answer is 0
Pentium gives 256

The Bug

- Pentium uses a fast floating point division algorithm
- Requires a table of constants
  - Three "2" entries were left out of the table
  - Treated as "0"s by the processor

Fault Tolerance

- Multiple different implementations
- Drawbacks
  - Very expensive
  - Slow and/or parallel hardware requirements
  - No assurance distinct implementations aren't correlated
- Example: The space shuttle

Correctors

- Don't just find bugs, fix bugs
- How can we do that?!
- Randomization is the key...

Sketch

- Here's the game:
  - Given \( f \), which computes correct answer with known probability
  - The correcting program uses \( f \) as a subroutine
  - Idea: Use multiple calls to \( f \) to calculate the answer in different ways
  - Constraint: Only allowed a constant factor increase in running time

An Example

- Consider multiplication \( a \times b \)
  - Over a finite field
- Choose random numbers \( r_1, r_2 \)
- Calculate
  \[
  (a - r_1) \times (b - r_2) + (a - r_1) \times (b - r_2) + r_1 \times r_2
  = (a - r_1) \times (b - r_2) \times r_1 + (a - r_1) \times r_2
  = a \times b
  \]
Why Does this Work?

\[(a - r_1) \times (b - r_2) + (a - r_1) \times r_2 + (b - r_2) \times r_1\]

- Each multiplication is a random pair
  - With respect to a, b
  - So each is correct with a known probability \(p\)
  - Sum is wrong bounded above by probability \(4(1 - p)\)
- Repeat trials to increase probability to desired level

Opinions on Correctors

- This sounds like a crazy idea
  - How often is multiplication buggy?
  - How often is my problem a finite field?
- Crazy or not, people are working on it...

Correctors: Historical Example

- But people have tried to build correctors for complex problems
- Consider a historical example where the output is human-generated
  - But could be machine generated
- PL/C was a PL/1 compiler developed at Cornell
  - In the days when compilation was expensive
  - Automatically corrected errors in program
  - Always yielded a valid program "close to" the one the programmer entered

PL/C

- The experience with PL/C was that automatic correction didn’t work
  - The further a program was from a valid program, the more bizarre the output
  - Example: "To be or not to be, that is the question...." Compiles to "begin end;"
- The idea died as compilers got faster

A Use of Correction in the Real World

- There are two kinds of bugs
  - Deterministic
    - These are repeatable—we can find and fix these
  - Non-deterministic
    - Timing bugs
    - Must get lucky to fix one of these
- For a non-deterministic bug, just try again
  - Standard in commercial databases
  - This is a form of automatic correction
    - Note: Requires fail-stop semantics, though

Correctors: Repairing Data Structures

- Write down data structure invariants
  - i.e., asserts
  - But in a nicer specification language
- Example: a file system
  - File system root exists
  - File reference counts consistent with references from directories
  - No block belongs to more than one file
Idea

- Convert each specification into DNF formula
  - Disjunction of conjunctions of basic propositions
- When a violation is detected, pick conjunction that is false
  - Choose based on cost model
  - Repair each of the atomic predicates
- More complicated than it sounds
  - Might not terminate

Autonomic Computing

- Data structure repair is just one facet of a push to make systems "self correcting"
- Buzzword: Autonomic computing
  - Automatic allocation of resources
    - Believable
  - Automatic repair of faults
    - Less believable

Conclusions about Correction

- Not obvious how to apply the idea in full to a complex system
- But a useful idea for specific properties
  - Non-deterministic, but detectable, bugs
  - Perhaps repairing data structures

Back to Checkers

- Each CS community has its view of software engineering
- Programming languages
  - The answer is the compiler
    - i.e., static analysis
- Operating systems
  - The answer is the operating system
    - i.e., dynamic analysis, use LRU
- Theory
  - The answer is asymptotic complexity + randomization

Back to the Definition

- A checker for f is a program that
  - Verifies/refutes that f(x) = y correctly
  - Does so in asymptotically less time than f
- Examine the assumptions underlying this approach

Assumption: The Specification

- Checking f requires we know f’s specification
  - Completely, not just partially
- There is no big system for which we know the full specification
- Partially explains why checking examples are all tiny, neat problems
Assumption: Functions

- Assume programs are input-output functions
- But this is not realistic
  - Most important systems are stream transducers
  - Take a sequence of inputs, produce sequence of outputs
  - Need notion of correct behavior up to a point in time

Assumption: Asymptotic Complexity is Crucial

- Checker must be asymptotically faster
  - Great if it is, but is this required?
- In practice, happy if checker is $\leq 10\%$ of time to compute answer
- But, asymptotic requirement is useful
  - Forces critical thinking in asserts
  - More likely to have orthogonal checker

Note

- Two kinds of bugs:
  - Fail-stop
    - Program dumps core, throws uncaught exception, etc.
  - Malicious
    - Program keeps going, but just produces the wrong answer
- Checking is about the second class only
  - More pernicious than fail-stop
  - But today the world has enough trouble with fail-stop bugs

Checking

- Program checking is a great idea
- Three parts
  - Function produces output that can be checked
    - Independently
  - Checking cost is small relative to computing answer
  - Checking actually checks correctness
- Use this!