Program Checking

Lecture 9
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
  - Double-entry book-keeping
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)
What Does It Mean to Check?

• Say procedure $p(x)$ implements a function $f(x)$
  - if we compute $y := p(x)$
  - then we should have $y = f(x)$

• How can we verify $y = f(x)$?
  - multiple implementations and voting
  - testing
  - static analysis
  - on-line, independent checking
Simple Checkers

• Procedure \( p(x) \) implements function \( f(x) \)

• Have a checking procedure \( C(x,y) \) with
  - \( C(x,y) = \text{true} \iff f(x) = y \)
    • With high probability, if \( C \) is randomized
    - \( C \) is asymptotically faster than \( f \)

• We call \( C \) is a checker for \( f \)
  - \( C \) is also an executable specification for \( f \)
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

• Other examples ... ?
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?
Propositional Satisfiability

• **Complicated, expensive SAT solver finds a satisfying truth assignment for a given boolean formula**

• **Have a separate, linear-time routine to ensure the truth assignment satisfies the original formula**
Theorem Proving

• Automatic Theorem Prover produces a proof of the conjecture
  - Theorem provers are often big, complicated, and untrustworthy

• Have a separate and simple proof checker
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
Proof Carrying Code

• Compiler produces binary executable together with proof that the executable satisfies a given security specification

• Each client downloads executable and quickly checks proof to verify that the code satisfies the security specification
  - no need for trust
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
  – from axiom: \(((A \mod r) \times D) \mod r = (A \times D) \mod r\)

• If \(A \times B = C'\), then \(C \mod r \neq C' \mod r\)
  – With high probability

• Small number multiplies/mod can be done fast

• This is the example application to the Pentium bug
Checking Division

- Given $N$, $D$, division computes $Q$, $R$
  - $N$ numerator
  - $D$ denominator
  - $Q$ quotient
  - $R$ remainder

\[ N = D \times Q + R \]
if and only if
\[ N - R = D \times Q \]

- Reduces to checking multiplication
Comparison with Other Techniques

• The papers draw comparisons to several other approaches:
  • Verification
  • Asserts
  • 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

• But this is largely a strawman
  - Full verification is only used in special situations
Assert Programming

- Many programmers use asserts
  - Really, the culture of checking
  - Paper acknowledges the connection

- But paper focuses on asserts that are
  - more complete
  - more focused on functional behavior
  - particularly efficient
Testing

- Testing and checking both try to find errors
- Checking is
  - Automatic
  - Runs every time
  - Rigorous (says yes/no correctly)
    - At least if checker is correct

- Which is more effective?
- Are they complementary?
  - Checking makes the test suite more effective, and vice versa
- Do we want both?
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 might have found this one
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, or Arianne 5
Correctors

• Don’t just find bugs, fix bugs

• How can we do that?!

• Randomization is the key . . .
Sketch

Here’s the game:

- Given procedure \( p(x) \), which computes correct answer for \( f(x) \) with known probability

- The correcting program uses \( p \) as a subroutine

- Idea: Use multiple calls to \( p \) 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

\[
(a - r_1) \times (b - r_2) + (a - r_1) \times r_2 + r_1 \times (b - r_2) + r_1 \times r_2
= ((a - r_1) + r_1) \times ((b - r_2) + r_2)
= a \times b
\]
Why Does this Work?

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

• 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?

• The idea is probably crazy
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
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
Back to Checkers

• Each CS community has own view of software development

• Programming languages
  – The answer is the compiler, i.e., static analysis

• Operating systems
  – The answer is the OS, i.e., dynamic analysis, caches

• Software engineering
  – The answer is in the development process

• Theory
  – The answer is asymptotic complexity + randomization
Back to the Definition

• A checker for \( f \) is a program \( C \) 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 the examples are all tiny, neat problems

- Would checking partial specifications be useful?
Assumption: Functions

• Assume programs are input-output functions

• But this is not realistic
  – Most important systems are stream transducers
  – Takes input sequence, produces output sequence

• 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 $\cdot 10\%$ of time 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 insidious than fail-stop