CSE 303: Concepts and Tools for Software Development

Dan Grossman

Spring 2007

Lecture 18— Specifications; Profiling (gprof)

Where are We

- Talked about testing, but not what (partially) correct was
- Then another useful tool: a run-time profiler
  - In particular, `gprof`

Specifying Code?

We made a *big* assumption, that we know what the code is *supposed* to do!

Oftentimes, a complete *specification* is at least as difficult as writing the code. But:

- It's still worth thinking about.
- *Partial* specifications are better than none.
- *Checking* specificatins (at compile-time and/or run-time) is great for finding bugs early and "assigning blame".

Full Specification

Often tractable for very simple stuff: "Take an `int` x and return 0 iff there exists ints y and z such that $y * z == x$ (where $x, y, z > 0$ and $y, z < x$)."

What about sorting a doubly-linked list?

- Precondition: Can input be NULL? Can any `prev` and `next` fields be NULL? Must it be a cycle or is "balloon" okay?
- Postcondition: Sorted (how to specify?) – *and* a permutation of the input (no missing or new elements).

And there's often more than "pre" and "post" – time/space overhead, other effects (such as printing), things that may happen in parallel.

Specs should guide programming and testing! Should be *declarative* ("what" not "how") to *decouple* implementation and use.

Pre/post and invariant

Pre- and post-conditions apply to any statement, not just functions

- What is assumed before and guaranteed after

Because a loop "calls itself" its body's post-condition better *imply* the loop's precondition.

- A *loop invariant*

Example: find max (next slide)

Pre/post and invariant

```
// pre: arr has length len; len >= 1
int max = arr[0];
int i=1;
while(i<len) {
  if(arr[i] > max)
    max = arr[i];
  ++i;
}
// post: max >= all arr elements
```

loop-invariant: For all `j<i`, `max>=arr[j]`.

- to show it holds after the loop body, must assume it holds before loop body
- loop-invariant plus `!(i<len)` after body, enough to show post

Partial Specifications

The difficulty of full specs need not mean abandon all hope.

Useful partial specs:

- Can args be NULL?
- Can args alias?
- Are stack pointers allowed? Dangling pointers?
- Are cycles in data structures allowed?
- What is the minimum/maximum length of an array?
- ...

Guides callers, callees, and testers.

Beyond testing

Specs are useful for more than "things to think about while coding" and testing and comments.

Sometimes you can check them dynamically, e.g., with *assertions* (all examples true for C and Java)

- Easy: argument not NULL
- Harder but doable: list not cyclic
- Impossible: Does the caller have other pointers to this object?

assert in C

In C:

```
#include <assert.h>
void f(int *x, int*y) {
  assert(x!=NULL);
  assert(x!=y);
  ...
}
```

- A *macro*; ignore argument if `NDEBUG` defined at time of `#include`, else evaluate and exit with file/line number if zero.

assert in Java

In Java (as of version 1.4):

```
void f(Foo x, Foo y) {
  assert x != null;
  assert x != y : "args to f should not be pointer-equal";
}
```

- By default, ignored.
- At program-start, use command-line options to specify which packages' assertions are *enabled*.

assert style

Many oversimply say "always" check everything you can. But:

- Often not on "private" functions (caller already checked)
- Unnecessary if checked *statically*

"Disabled" in released code because:

- executing them takes time
- failures are not fixable by users anyway
- assertions themselves could have bugs/vulnerabilities

Others say:

- Should leave enabled; corrupting data on real runs is worse than when debugging

Static checking

A stronger type system or other code-analysis tool might take a program and ensure

- Plusses: earlier detection ("coverage" without running program), faster code
- Minus: Potential "false positives" (spec couldn't ever actually be violated, but tool thinks so)

Deep CSE322 fact: Every code-analysis tool proving a non-trivial fact has either false positives (unwarranted warning) or false negatives (missed bug) or both.

Deep real-world fact: That doesn't make them unuseful.

Profilers

A *profiler* monitors and reports (performance) information about a program execution.

They are useful for "debugging correct programs" by learning where programs consume most time and/or space.

"90/10 rule of programs" (and often worse for new programs) – a profiler helps you "find the 10".

But: The tool can be misused and misleading.

What profilers tell you

Different profilers profile different things.

`gprof`, a profiler for code produced by `gcc` is widely available and pretty typical:

- *Call counts:* # of times each function a calls each function b
  - And the simpler fact: # of times a was called
- *Time samples:* # of times the program was executing a when "the profiler woke up to check where the program was".

Neither is quite what you want (as we'll see later), but they're semi-easy and semi-quick to do:

- *Call counts:* Add code to every function call to update a table indexed by function pairs.
- *Time samples:* Use the processor's timer; wake up and see where the program is.

Using `gprof`

- Compile with `-pg` *on the right*.
  - When you create the `.o` (for call counts)
  - When you create the executable (for time samples)
- Run the program (creates (overwrites) `gmon.out`)
- Run `gprof` (on `gmon.out`) to get human-readable results.
- Read the results (takes a little getting used to).