EECS498-008
Formal Verification of Systems Software

Material and slides created by

Jon Howell and Manos Kapritsos

Recursion: exporting ensures

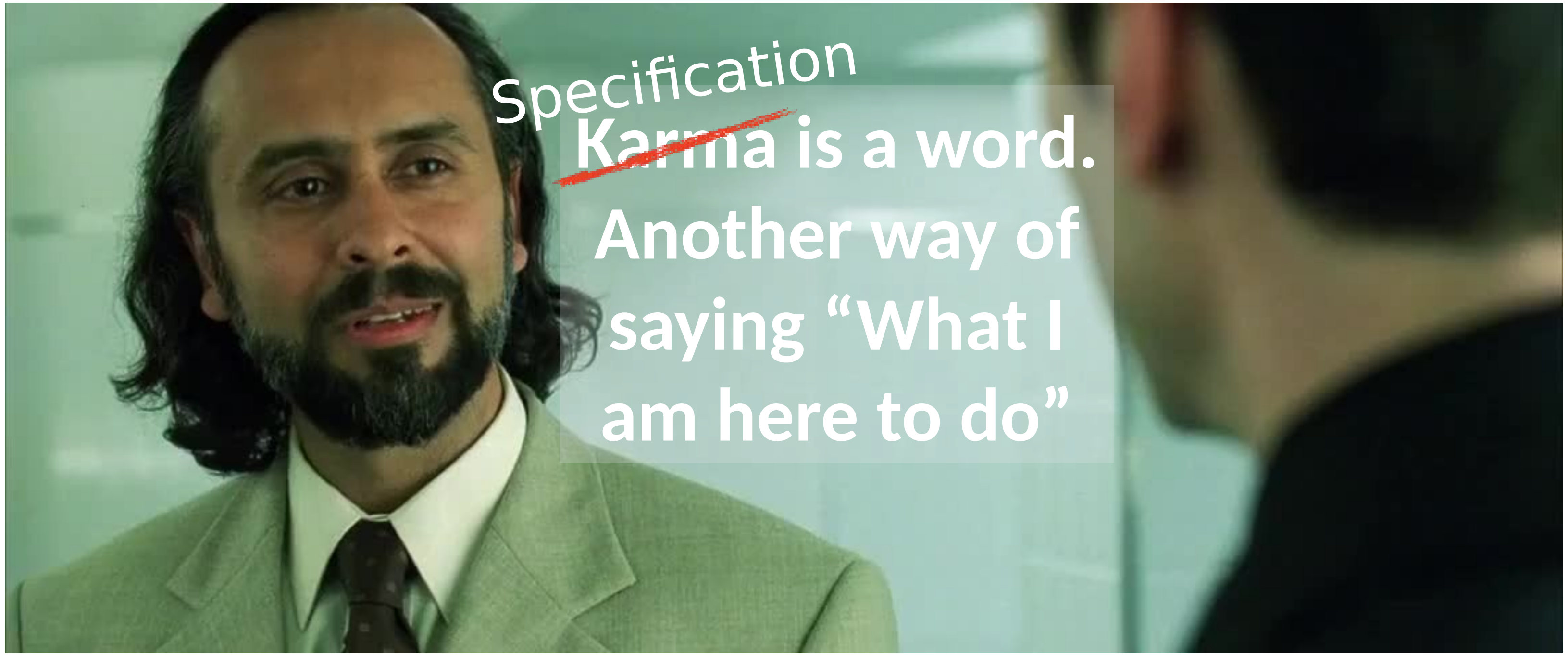
```
function Evens(count:int) : (outseq:seq<int>)
  ensures forall idx :: 0<=idx<|outseq| ==> outseq[idx] == 2 * idx
{
  if count==0 then [] else Evens(count) + [2 * (count-1)]
}
```

```
lemma myLemma(a:seq<int>)
  ensures Foo(a)
{
  myLemma(a[..|a|-1]);
  // proof about last element of a goes here
}
```

Chapter 2 exercises

- Will be released tonight
    - Deadline for PS1 (i.e. Chapters 1 and 2) is September 23, 11:59pm.

Chapter 2: Specification


Specification
~~Karma~~ is a word. Another way of saying "What I am here to do"

How to specify our programs

Attempt #1: Just tell your programmers what you want them to code

Writing is nature's way of letting you know how sloppy your thinking is

-Dick Guindon

How to specify our programs

Attempt #2: Write down an English description (aka a design doc)

Mathematics is nature's way of letting you know how sloppy your writing is

-Leslie Lamport

Formal mathematics is nature's way of letting you know how sloppy your mathematics is

-Leslie Lamport

Formal specification

A way to define formally (i.e. precisely) what your program should do

Before you start writing code, make sure you know what code is supposed to be doing

Before you start writing a proof, make sure you know what you are proving

Specification

A specification defines ***which executions are allowable***

```
lemma Double(x:int) returns (y:int)
    ensures y == 2*x
{
    ...
}
```

(x=1, y=2) ✓

(x=2, y=4) ✓

(x=2, y=2) ✗

(x=-3, y=-6) ✓

(x=-2, y=4) ✗

Ways to specify what the program should do

- C-style assertions

```
y = 2*x;
assert(y==2*x)
```

- Postconditions

```
lemma Double(x:int) returns
(y:int)
    ensures y == 2*x
{
    y := 2*x;
}
```

- Properties/invariants

“At most one node holds the lock at any time”

- Refinement

- Linearizability
- Equivalence to logically centralized service

Specification is trusted

Formal verification: proving that your protocol or implementation meets the spec

You cannot prove that the spec is correct

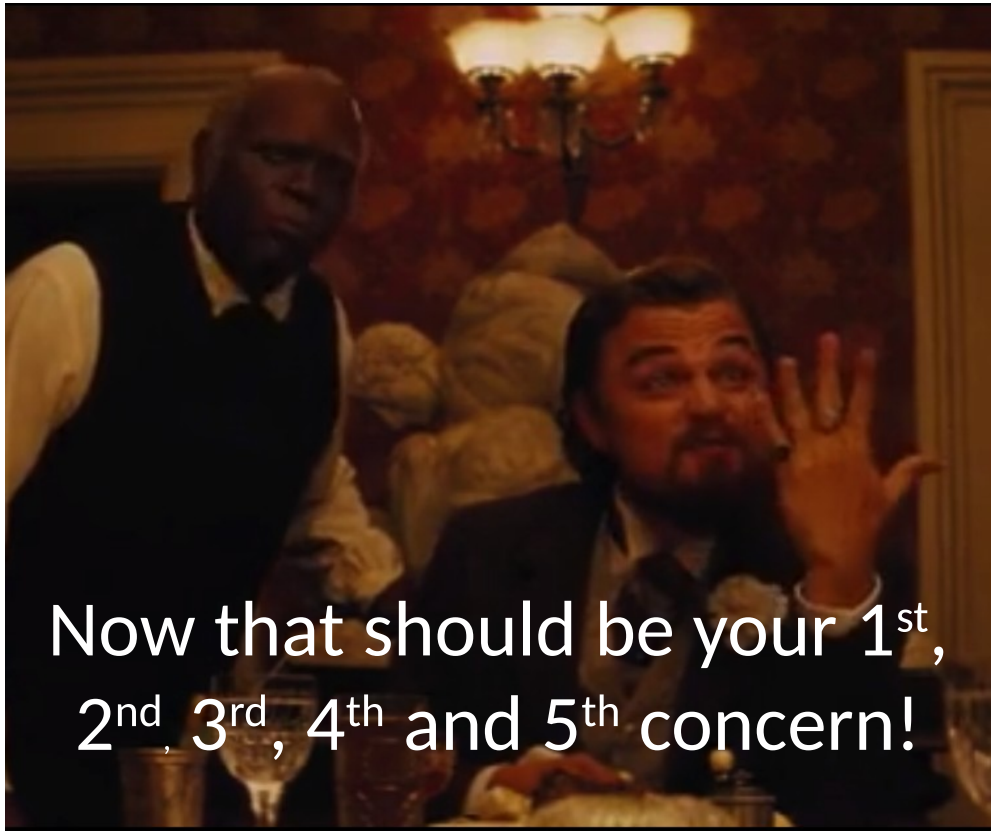
You have to ***trust*** your spec

Your proof is as good as your spec

A wrong spec is one of the few ways to introduce bugs into formally verified code

Check your spec

1. Check your spec!
2. Check your spec!
3. Check your spec!
4. Check your spec!
5. Check your spec!



The benefit of specification

The spec is typically ***much smaller*** than the code

- So we have to inspect a few lines of code only

Dijkstra’s algorithm spec

```
IsShortestPath(g, p) {
      && IsPath(g, p)
      && forall p2 :: IsPath(g, p2) ==> |p| <= |p2|
}
```

A good spec

A good spec is ***correct/complete***

- It precludes all undesirable behaviors

Example: IsMaxIndex

```
predicate IsMaxIndex(a:seq<int>, x:int) {
  && 0 < x < |a|
  && (forall i | 0 < i < |a| :: a[i] <= a[x])
}
```

A good spec (cont.)

A good spec is ***concise***

- It elides every irrelevant concept
- Is simple and easy to read

```
predicate IsMaxIndex(a:seq<int>, x:int) {
  && 0 <= x < |a|
  && (forall i | 0 <= i < |a| :: a[i] <= a[x])
}
```

A good spec (cont.)

A good spec is ***abstract***

- It doesn’t constrain the implementation

Dijkstra’s algorithm spec

```
IsShortestPath(g:Graph, p:Path) {
      && IsPath(g, p)
      && forall p2 :: IsPath(g, p2) ==> |p| <= |p2|

}
```

Edsger W. Dijkstra

- 1972 Turing Award winner
- Inventor of:
  - Dijkstra’s shortest path algorithm
  - Semaphores
  - The THE operating system
  - Banker’s algorithm
- “*Progress is possible only if we train ourselves to think about programs without thinking of them as pieces of executable code.*”

Verification and the “eradication” of bugs

Frequent quote from verification experts

- “We prove that there are no bugs at all…”

Frequent quote from verification skeptics

- “Nonsense! You can still have bugs in your spec”

The truth is somewhere in the middle

- Yes, your spec may have bugs
- But do you prefer inspecting 30 lines for bugs or 30000?