CS 3110

Lecture 17: Verification

Prof. Clarkson

Spring 2015

Today's music: Theme from *Downton Abbey*

Review

Current topic:

- Reasoning about programs

Last two lectures:

- Efficiency

Today: Verification

- How to reason about the correctness of code
- A little bit of formal reasoning

Question #0

Why am I wearing a top hat?

A. Because top hats are cool?
B. Did I binge-watch too much *Downton Abbey*?
C. Is it because we're getting formal?
D. All of the above

Question #0

Why am I wearing a top hat?

A. Because top hats are cool?
B. Did I binge-watch too much *Downton Abbey*?
C. **Is it because we're getting formal?**
D. All of the above

Building Reliable Software

- Suppose you work at (or run) a software company.
- Suppose you've sunk 30+ person-years into developing the "next big thing":
  - Boeing Dreamliner2 flight controller
  - Autonomous vehicle control software for Nissan
  - Gene therapy DNA tailoring algorithms
  - Super-efficient green-energy power grid controller
- How do you avoid disasters?
  - Turns out software endangers lives
  - Turns out to be impossible to build software

Approaches to Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - Static analysis
    ("lint" tools, FindBugs, …)
  - Fuzzers
- Mathematical
  - Sound type systems
  - "Formal" verification

Less formal: Techniques may miss problems in programs

All of these methods should be used!

Even the most formal can still have holes:

- did you prove the right thing?
- do your assumptions match reality?

More formal: eliminate *with certainty* as many problems as possible.

Testing vs. Verification

Testing:

- Cost effective
- Guarantee that program is correct on tested inputs and in tested environments

Verification:

- Expensive
- Guarantee that program is correct on all inputs and in all environments

Edsger W. Dijkstra

(1930-2002)

Turing Award Winner (1972)

For eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness

"Program testing can at best show the presence of errors but never their absence."

Verification

- In the 1970s, scaled to about tens of LOC
- Now, research projects scale to real software:
  - CompCert: verified C compiler
  - seL4: verified microkernel OS
  - Ynot: verified DBMS, web services
- In another 40 years?

Verification of max

```
(* returns: max x y is the maximum of x and y.  *)
val max : int -> int -> int
let max x y = if x>=y then x else y
```

How could we prove that the postcondition holds for any inputs?

Question #1

Which of the following defines "maximum"?

A. (max x y) >= x *and* (max x y) >= y

B. (max x y) = x *or* (max x y) = y

C. A and B

D. None of the above

Question #1

Which of the following defines "maximum"?

A. (max x y) >= x *and* (max x y) >= y

B. (max x y) = x *or* (max x y) = y

C. A and B

D. None of the above

Verification of max

```
(* returns: max x y is the maximum of x and y.
 *    that is:
 *      (max x y) >= x
 *        and
 *      (max x y) >= y
 *        and
 *      (max x y = x) or (max x y = y). *)
val max : int -> int -> int
let max x y = if x>=y then x else y
```

Let's give a proof that **`max`** satisfies its specification...

Verification of max

Expression	Assumptions	Justification
`if x>=y then x else y`	None	(We consider an arbitrary application of `max`)

Verification of max

Expression	Assumptions	Justification
`if x>=y then x else y`	None	(We consider an arbitrary application of `max`)
CASE: `x>=y`		

Verification of max

Expression		Assumptions	Justification
`if x>=y then x else y`		None	(We consider an arbitrary application of `max`)
CASE: `x>=y`			
	`x`	`x>=y`	Since the guard is true, the if expression evaluates to the then branch

Verification of max

Expression		Assumptions	Justification
`if x>=y then x else y`		None	(We consider an arbitrary application of `max`)
CASE: `x>=y`			
	`x`	`x>=y`	Since the guard is true, the if expression evaluates to the then branch
	Postcondition satisfied: `x>=x and x>=y and (x=x or x=y)`		

Verification of max

Expression		Assumptions	Justification
`if x>=y then x else y`		None	(We consider an arbitrary application of `max`)
CASE: `x>=y`			
	`x`	`x>=y`	Since the guard is true, the if expression evaluates to the then branch
	Postcondition satisfied: `x>=x and x>=y and (x=x or x=y)`		
CASE: not `(x>=y)`, i.e., `y>x`			

Verification of max

Expression		Assumptions	Justification
`if x>=y then x else y`		None	(We consider an arbitrary application of `max`)
CASE: `x>=y`			
	`x`	`x>=y`	Since the guard is true, the if expression evaluates to the then branch
	Postcondition satisfied: `x>=x and x>=y and (x=x or x=y)`		
CASE: not `(x>=y)`, i.e., `y>x`			
	`y`	`y>x`	Since the guard is false, the if expression evaluates to the else branch

Verification of max

Expression		Assumptions	Justification
`if x>=y then x else y`		None	(We consider an arbitrary application of `max`)
CASE: `x>=y`			
	`x`	`x>=y`	Since the guard is true, the if expression evaluates to the then branch
	Postcondition satisfied: `x>=x and x>=y and (x=x or x=y)`		
CASE: not `(x>=y)`, i.e., `y>x`			
	`y`	`y>x`	Since the guard is false, the if expression evaluates to the else branch
	Postcondition satisfied: `y>=x and y>=y and (y=x or y=y)`		

Verification of max

Expression		Assumptions	Justification
`if x>=y then x else y`		None	(We consider an arbitrary application of `max`)
CASE: `x>=y`			
	`x`	`x>=y`	Since the guard is true, the if expression evaluates to the then branch
	Postcondition satisfied: `x>=x and x>=y and (x=x or x=y)`		
CASE: not `(x>=y)`, i.e., `y>x`			
	`y`	`y>x`	Since the guard is false, the if expression evaluates to the else branch
	Postcondition satisfied: `y>=x and y>=y and (y=x or y=y)`		
Cases are exhaustive: `x>=y` or `y>x` And in every case, postcondition is satisfied. QED.			

Another implementation of max

```
(* returns: a value z s.t.
 *    z>=x and z>=y and (z=x or z=y) *)
let max' x y = (abs(y-x)+x+y)/2

(* returns: abs x is x if x>=0, otherwise -x *)
val abs : int -> int
```

Modular verification: use only the specs of other functions, not their implementations

Let's give a proof that `max'` satisfies its specification...

Verification of max'

Expression	Assumptions	Justification
`(abs(y-x)+x+y)/2`	None	(We consider an arbitrary application of `max'`)

Verification of max'

Expression	Assumptions	Justification
`(abs(y-x)+x+y)/2`	None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`	n1=x+y	x+y evaluates to some int n1

Verification of max'

Expression	Assumptions	Justification
`(abs(y-x)+x+y)/2`	None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`	n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`	n1=x+y n2=y-x	y-x evaluates to some int n2

Verification of max'

Expression	Assumptions	Justification
`(abs(y-x)+x+y)/2`	None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`	n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`	n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: `y>=x`		

Verification of max'

Expression		Assumptions	Justification
`(abs(y-x)+x+y)/2`		None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`		n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`		n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: `y>=x`			
	`(n2+n1)/2`	n1=x+y n2=y-x y>=x	By the spec of abs, abs(n2) evaluates to n2, because n2=y-x and y>=x

Verification of max'

Expression		Assumptions	Justification
(abs(y-x)+x+y)/2		None	(We consider an arbitrary application of max')
(abs(y-x)+n1)/2		n1=x+y	x+y evaluates to some int n1
(abs(n2)+n1)/2		n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: y>=x			
	(n2+n1)/2	n1=x+y n2=y-x y>=x	By the spec of abs, abs(n2) evaluates to n2, because n2=y-x and y>=x
	n3/2	" n3=n2+n1	n2+n1 evaluates to some int n3

Verification of max'

Expression		Assumptions	Justification
`(abs(y-x)+x+y)/2`		None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`		n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`		n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: `y>=x`			
	`(n2+n1)/2`	n1=x+y n2=y-x y>=x	By the spec of abs, abs(n2) evaluates to n2, because n2=y-x and y>=x
	`n3/2`	" n3=n2+n1	n2+n1 evaluates to some int n3
	`y`	"	n3/2 = (y-x+x+y)/2 = 2y/2 = y

Verification of max'

Expression		Assumptions	Justification
`(abs(y-x)+x+y)/2`		None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`		n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`		n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: `y>=x`			
	`(n2+n1)/2`	n1=x+y n2=y-x y>=x	By the spec of abs, abs(n2) evaluates to n2, because n2=y-x and y>=x
	`n3/2`	" n3=n2+n1	n2+n1 evaluates to some int n3
	`y`	"	n3/2 = (y-x+x+y)/2 = 2y/2 = y
	Postcondition satisfied: `y>=x and y>=y and (y=x or y=y)`		

Verification of max'

Expression	Assumptions	Justification
`(abs(y-x)+x+y)/2`	None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`	n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`	n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: not `(y>=x)`, i.e., `y<x`		

Verification of max'

Expression	Assumptions	Justification
`(abs(y-x)+x+y)/2`	None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`	n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`	n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: not `(y>=x)`, i.e., `y<x`		
`(-n2+n1)/2`	n1=x+y n2=y-x y<x	By the spec of abs, abs(n2) evaluates to -n2, because n2=y-x and y<x

Verification of max'

Expression	Assumptions	Justification
`(abs(y-x)+x+y)/2`	None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`	n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`	n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: not `(y>=x)`, i.e., `y<x`		
`(-n2+n1)/2`	n1=x+y n2=y-x y<x	By the spec of abs, abs(n2) evaluates to -n2, because n2=y-x and y<x
`(n3+n1)/2`	" n3 = -n2	-n2 evaluates to some int n3

Verification of max'

Expression	Assumptions	Justification
`(abs(y-x)+x+y)/2`	None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`	n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`	n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: not `(y>=x)`, i.e., `y<x`		
`(-n2+n1)/2`	n1=x+y n2=y-x y<x	By the spec of abs, abs(n2) evaluates to -n2, because n2=y-x and y<x
`(n3+n1)/2`	" n3 = -n2	-n2 evaluates to some int n3
`n4/2`	" n4=n3+n1	n3+n1 evaluates to some int n4

Verification of max'

Expression	Assumptions	Justification
`(abs(y-x)+x+y)/2`	None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`	n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`	n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: not `(y>=x)`, i.e., `y<x`		
`(-n2+n1)/2`	n1=x+y n2=y-x y<x	By the spec of abs, abs(n2) evaluates to -n2, because n2=y-x and y<x
`(n3+n1)/2`	" n3 = -n2	-n2 evaluates to some int n3
`n4/2`	" n4=n3+n1	n3+n1 evaluates to some int n4
`x`	"	n4/2 = (-(y-x)+x+y)/2 = 2x/2 = x

Verification of max'

Expression	Assumptions	Justification
`(abs(y-x)+x+y)/2`	None	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`	n1=x+y	x+y evaluates to some int n1
`(abs(n2)+n1)/2`	n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: not `(y>=x)`, i.e., `y<x`		
`(-n2+n1)/2`	n1=x+y n2=y-x y<x	By the spec of abs, abs(n2) evaluates to -n2, because n2=y-x and y<x
`(n3+n1)/2`	" n3 = -n2	-n2 evaluates to some int n3
`n4/2`	" n4=n3+n1	n3+n1 evaluates to some int n4
`x`	"	n4/2 = (-(y-x)+x+y)/2 = 2x/2 = x
Postcondition satisfied: `x>=x and x>=y and (x=x or x=y)`		

Verification of max'

Expression	Assumptions	Justification
(abs(y-x)+x+y)/2	None	(We consider an arbitrary application of max')
(abs(y-x)+n1)/2	n1=x+y	x+y evaluates to some int n1
(abs(n2)+n1)/2	n1=x+y n2=y-x	y-x evaluates to some int n2
CASE: y>=x		
…	…	…
CASE: not (y>=x), i.e., y<x		
…	…	…
Cases are exhaustive: y>=x or y<x And in every case, postcondition is satisfied. QED.		

Verification of max'

```
# max' max_int 0;;
- : int = -1

(abs(0-max_int)+max_int+0)/2
=
(abs(-max_int)+max_int)/2
=
(max_int+max_int)/2
=
-2/2
=
-1
```

Question #2

What went wrong?

A. There's a bug in our proof
B. There's a bug in our specification of max
C. There's a bug in our specification of abs
D. There's a bug in our implementation
E. Something else

Question #2

What went wrong?

A. There's a bug in our proof
B. There's a bug in our specification of max
C. There's a bug in our specification of abs
D. There's a bug in our implementation
E. **Something else (mainly this)**

What went wrong?

Unstated, unsatisfied preconditions!

```
(* requires: min_int <= x ++ y <= max_int *)
val (+) : int -> int -> int

(* requires: min_int <= x -- y <= max_int *)
val (-) : int -> int -> int
```

where ++ and -- denote the "ideal" math operators

Where did it go wrong?

- Everywhere we wrote something like "a+b evaluates to some int n"
- We should have been checking the precondition of **`(+)`**
- Same for **`(-)`**
- Clients don't know to guarantee that those preconditions hold!
  - as shown by the example of **`max' max_int 0`**
- So we strengthen the spec of max' by adding a precondition to it

Corrected spec for max'

```
(* returns: a value z s.t.
 *    z>=x and z>=y and (z=x or z=y)
 * requires: min_int/2 <= x <= max_int/2
 *       and min_int/2 <= y <= max_int/2 *)
let max' x y = (abs(y-x)+x+y)/2
```

Let's call that **`requires`** clause PRE for short

Verification of max'

Expression		Assumptions	Justification
`(abs(y-x)+x+y)/2`		PRE	(We consider an arbitrary application of `max'`)
`(abs(y-x)+n1)/2`		" n1=x+y	x+y evaluates to some int n1, and by PRE, that addition can't overflow
`(abs(n2)+n1)/2`		" n2=y-x	y-x evaluates to some int n2, and by PRE, that subtraction can't underflow
CASE: `y>=x`			
	`(n2+n1)/2`	n1=x+y n2=y-x y>=x	By the spec of abs, abs(n2) evaluates to n2, because n2=y-x and y>=x
	`n3/2`	" n3=n2+n1	n2+n1 evaluates to some int n3, and by PRE, that addition can't overflow
	`y`	"	n3/2 = (y-x+x-y)/2 = 2y/2 = y
	Postcondition satisfied: `y>=x and y>=y and (y=x or y=y)`		

Other case is similar; conclusion is the same

Verified max'

```
(* returns: a value z s.t.
 *    z>=x and z>=y and (z=x or z=y)
 * requires: min_int/2 <= x <= max_int/2
 *       and min_int/2 <= y <= max_int/2 *)
let max' x y = (abs(y-x)+x+y)/2
```

Verified max' vs max

```
(* returns: a value z s.t.
 *    z>=x and z>=y and (z=x or z=y)
 * requires: min_int/2 <= x <= max_int/2
 *       and min_int/2 <= y <= max_int/2 *)
let max' x y = (abs(y-x)+x+y)/2

(* returns: a value z s.t.
 *    z>=x and z>=y and (z=x or z=y) *)
let max x y = if x>=y then x else y
```

max' assumes more about its input than max does

...max' has a stronger precondition

Strength of preconditions

Given two preconditions PRE1 and PRE2 such that
PRE1 => PRE2

- (and PRE1 not logically equivalent to PRE2)
- e.g., x>1 => x>0
- PRE1 is stronger than PRE2:
  - assumes more
  - function can be called under fewer circumstances
- PRE2 is weaker than PRE1:
  - assumes less
  - function can be called under more circumstances
- The weakest possible precondition is to assume nothing, but that might make implementation difficult
- The strongest possible precondition is to assume so much that the function can never be called

Verified max' vs max

```
(* returns: a value z s.t.
 *     z>=x and z>=y and (z=x or z=y)
 * requires: min_int/2 <= x <= max_int/2
 *        and min_int/2 <= y <= max_int/2 *)
let max' x y = (abs(y-x)+x+y)/2

(* returns: a value z s.t.
 *     z>=x and z>=y and (z=x or z=y) *)
let max x y = if x>=y then x else y
```

max' assumes more about its input than max does

...max' has a stronger precondition

...max' can be called under fewer circumstances; maybe less useful to clients

Strength of postconditions

Given two postconditions POST1 and POST2 such that POST1 => POST2

- (and POST1 not logically equivalent to POST2)
- e.g., returns a stably-sorted list => returns a sorted list
- POST1 is stronger than POST2:
  - promises more
  - function result can be used under more circumstances
- POST2 is weaker than POST1:
  - promises less
  - function result can be used under fewer circumstances
- The weakest possible postcondition is to promise nothing
- The strongest possible postcondition is to promise so much that the function could never be implemented

Question #3

Which is the stronger postcondition for **`find`**?

```
A: (* returns:  find lst x is an index
    *           at which x is found in lst
    * requires: x is in lst *)

B: (* returns:  find lst x is the first index
    *           at which x is found in lst
    * requires: x is in lst *)

val find: 'a list -> 'a -> int
```

Question #3

Which is the stronger postcondition for **`find`**?

```
A: (* returns:  find lst x is an index
    *           at which x is found in lst
    * requires: x is in lst *)

B: (* returns:  find lst x is the first index
    *           at which x is found in lst
    * requires: x is in lst *)

val find: 'a list -> 'a -> int
```

Satisfaction of specs

- Suppose a client gives us a spec to implement.
- Could we implement a function that meets a different spec, verify that implementation against that other spec, and still make the client happy?
- Analogy: In Java, if you're asked to implement a function that returns a List, could you instead return
  - an Object?
  - an ArrayList?

Satisfaction of specs

- If a client asked for A, could we give them B?
- If a client asked for B, could we give them A?

```
A: (* returns:  find lst x is an index
    *           at which x is found in lst
    * requires: x is in lst *)

B: (* returns:  find lst x is the first index
    *           at which x is found in lst
    * requires: x is in lst *)
```

Satisfaction of specs

- If a client asked for A, could we give them B? **Yes.**
- If a client asked for B, could we give them A? **No.**

```
A: (* returns:  find lst x is an index
    *           at which x is found in lst
    * requires: x is in lst *)

B: (* returns:  find lst x is the first index
    *           at which x is found in lst
    * requires: x is in lst *)
```

Satisfaction of specs

- If a client asked for C, could we give them D?
- If a client asked for D, could we give them C?

```
C: (* returns: a value z s.t.
    *     z>=x and z>=y and (z=x or z=y)
    * requires: min_int/2 <= x <= max_int/2
    *         and min_int/2 <= y <= max_int/2 *)

D: (* returns: a value z s.t.
    *     z>=x and z>=y and (z=x or z=y) *)
```

Satisfaction of specs

- If a client asked for C, could we give them D? Yes.
- If a client asked for D, could we give them C? No.

```
C: (* returns: a value z s.t.
    *     z>=x and z>=y and (z=x or z=y)
    * requires: min_int/2 <= x <= max_int/2
    *           and min_int/2 <= y <= max_int/2 *)
```

```
D: (* returns: a value z s.t.
    *     z>=x and z>=y and (z=x or z=y) *)
```

Question #4

Suppose a client gives us a spec to implement:

```
requires: PRE
returns:  POST
```

Which of the following could we instead implement and still satisfy the client?

A. Weaker PRE and weaker POST
B. Weaker PRE and stronger POST
C. Stronger PRE and weaker POST
D. Stronger PRE and stronger POST
E. None of the above

Question #4

Suppose a client gives us a spec to implement:

```
requires: PRE
returns:  POST
```

Which of the following could we instead implement and still satisfy the client?

A. Weaker PRE and weaker POST
B. Weaker PRE and stronger POST
i.e., assume less and promise more
C. Stronger PRE and weaker POST
D. Stronger PRE and stronger POST
E. None of the above

Refinement

Specification B *refines* specification A if any implementation of B is also an implementation of A

- Any implementation of "find first" is an implementation of "find any", so "find first" refines "find any"
- Any implementation of "max" is an implementation of "max of small ints", so "max" refines "max of small ints"

Refinement and PS's

- We give you a SPEC1 for an exercise
- You refine that to a new SPEC2
  - Weaken the precondition or strengthen the postcondition
- You submit an implementation of SPEC2
- By the definition of refinement, any implementation of SPEC2 is an implementation of SPEC1
  - so you are ☺
- But if you incorrectly refine the spec, then you are ☹
  - (strengthen the precondition or weaken the postcondition)

Refinement and PS's

- We give you a SPEC1 for an exercise
- You implement that
  - You are ☺
- We post a refined SPEC2 on Piazza.
  - Weakens precondition or strengthens postcondition
- An implementation of SPEC1 is not necessarily an implementation of SPEC2!
  - You are ☹
- Which is why one of my commandments to TAs is "Don't refine the spec."
- And why I tell you, "This is unspecified; do something reasonable."

Refinement and verification

How can we verify that SPEC2 refines SPEC1?

- Need to prove that PRE1 => PRE2
  - i.e., PRE2 has a weaker precondition than PRE1
- and that POST2 => POST1
  - ie., POST2 has a stronger postcondition than POST1

Proof

- We worked only somewhat formally today
  - Wrote formulas involving *and*, *or*, =>
  - How do we know we got it right?
- Formal verification: checked by machine
  - maybe machine generates the proof
  - maybe machine only checks the proof
- For that, we need *formal logic* (see CS 4860) and *proof assistants*