Specifications

Prof. Clarkson
Fall 2016

Today's music: *Nice to know you* by Incubus

Review

Previously in 3110:

- Behavioral equivalence
- Proofs of correctness by induction on naturals, lists, trees, ...

Today:

- Verify that a function implementation satisfies its specification

Specification vs. Implementation

Specification ("spec"):

```
(* [max x y] is the maximum of [x] and [y].  *)
val max : int -> int -> int
```

Implementation:

```
let max x y = if x>=y then x else y
```

Specifications

```
(* postcondition:  ...
   precondition :  ...   *)
val f: t1 -> t2
```

- Postcondition: guaranteed to be true of value returned by function
- Precondition: must be true of value passed to function as argument
  - in which case function must not diverge nor raise exception
  - hence if precondition holds, function is guaranteed to evaluate to a value

Specifications

Choices of how to write specification comment for max's precondition:

- *`omit`*
- `precondition: none.`
- `requires: nothing.`
- `assumes: nothing.`
- `...`

Specifications

Choices of how to write specification comment for max's postcondition:

- `[max x y] is the maximum of [x] and [y].`
- `postcondition: [max x y] is the maximum of [x] and [y].`
- `returns: [max x y] is the maximum of [x] and [y].`
- `ensures: [max x y] is the maximum of [x] and [y].`
- `...`

Verification

- Verification: prove that implementation satisfies specification
- Proof gets to assume precondition
- Proof has to establish that postcondition holds
  - Might use behavioral equivalence
  - Might use structural induction
  - …

Question

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. The conjunction of A and B

D. None of the above

Question

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. The conjunction of 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...

```
let max x y = if x>=y then x else y
```

Verification of max

```
Theorem:
(max x y) >= x and (max x y) >= y
  and (max x y = x) or (max x y = y)

Proof:  by case analysis

Case:  x >= y
  Note that max x y ~ x,
    because max x y -->* x when x >= y.
  Substituting x for (max x y) in the theorem,
    we have x >= x and x >= y and (x=x or x=y).
    By math and the assumption that x >= y,
    that holds.
```

```
let max x y = if x>=y then x else y
```

Verification of max

```
Theorem:
(max x y) >= x and (max x y) >= y
  and (max x y = x) or (max x y = y)

Proof:  by case analysis

Case:  x < y
  Note that max x y ~ y,
    because max x y -->* y when x < y.
  Substituting y for (max x y) in the theorem,
    we have y >=x and y >= y and (y=x or y=y).
    By math and the assumption that x < y,
    that holds.
```

```
let max x y = if x>=y then x else y
```

Verification of max

```
Theorem:
(max x y) >= x and (max x y) >= y
  and (max x y = x) or (max x y = y)

Proof:  by case analysis

Case:  x >= y
 ...
Case:  x < y
 ...

Those two cases are exhaustive.

QED
```

Another implementation of max

```
(* (max' x y) >= x and (max' x y) >= y
     and (max' x y = x) or (max' x y = 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

But if we don't have code, can't use ~ and eval...

(in this case we could appeal to math, but we won't)

Instead use specification!

Specification structure

```
(* postcondition:  f x is z where R(z,x)
   precondition :  Q(x)  *)
val f: t1 -> t2
```

e.g.

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

```
R(z,x) = (z=x or z=-x) and z>=0
Q(x) = true
```

Using specifications in proofs

```
(* postcondition:  f x is z where R(z,x)
   precondition :  Q(x)  *)
val f: t1 -> t2
```

New axiom: **specification**

```
if Q(x) then there exists z such that
  f x ~ z and R(z,x)
```

This axiom introduces an assumption about `f`: now someone is obligated to verify `f`!

```
let max' x y = (abs(y-x)+x+y)/2
```

Verification of max'

```
Theorem:
(max' x y) >= x and (max' x y) >= y
  and (max' x y = x) or (max' x y = y)

Proof:  by case analysis

Case:  y-x >= 0  equiv.  y >= x
  Note that abs(y-x) ~ y-x by specification
    and by assumption that y >= x.
  So max' x y ~ (y-x + x + y)/2 ~ (y+y)/2 ~ y.
  Substituting y for (max' x y) in the theorem,
    we have y >= x and y >= y and (y=x or y=y).
    By math and the assumption that y >= x,
    that holds.
```

```
let max' x y = (abs(y-x)+x+y)/2
```

Verification of max'

```
Theorem:
(max' x y) >= x and (max' x y) >= y
  and (max' x y = x) or (max' x y = y)

Proof:  by case analysis

Case:  y-x < 0  equiv.  y < x
  Note that abs(y-x) ~ x-y by specification, math,
    and by assumption that y < x.
  So max' x y ~ (x-y + x + y)/2 ~ (x+x)/2 ~ x.
  Substituting x for (max' x y) in the theorem,
    we have x >= x and x >= y and (x=x or x=y).
    By math and the assumption that y < x,
    that holds.
```

```
let max' x y = (abs(y-x)+x+y)/2
```

Verification of max'

```
Theorem:
(max' x y) >= x and (max' x y) >= y
  and (max' x y = x) or (max' x y = y)

Proof:  by case analysis

Case:  y-x >= 0
 ...
Case:  y-x < 0
 ...

Those two cases are exhaustive.

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

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

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)

We agreed to ignore the limits of machine arithmetic...

Machine arithmetic

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

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

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

- in counterexample, we attempt to compute `max_int+max_int`
- so our implementation of `max'` doesn't guarantee those preconditions hold when it calls `(+)` and `(-)`
- we could add a precondition to `max'` to rule out that behavior...

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

Theorem:
if min_int/2 <= x <= max_int/2
  and min_int/2 <= y <= max_int/2
then max' x y >= x and max' x y >= y
  and (max' x y = x or max' x y = y)

Proof:  omitted.  QED
```

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 ≢ PRE2

- e.g., $x>1 \Rightarrow x>0$
- 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 ≢ 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

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

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

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

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"

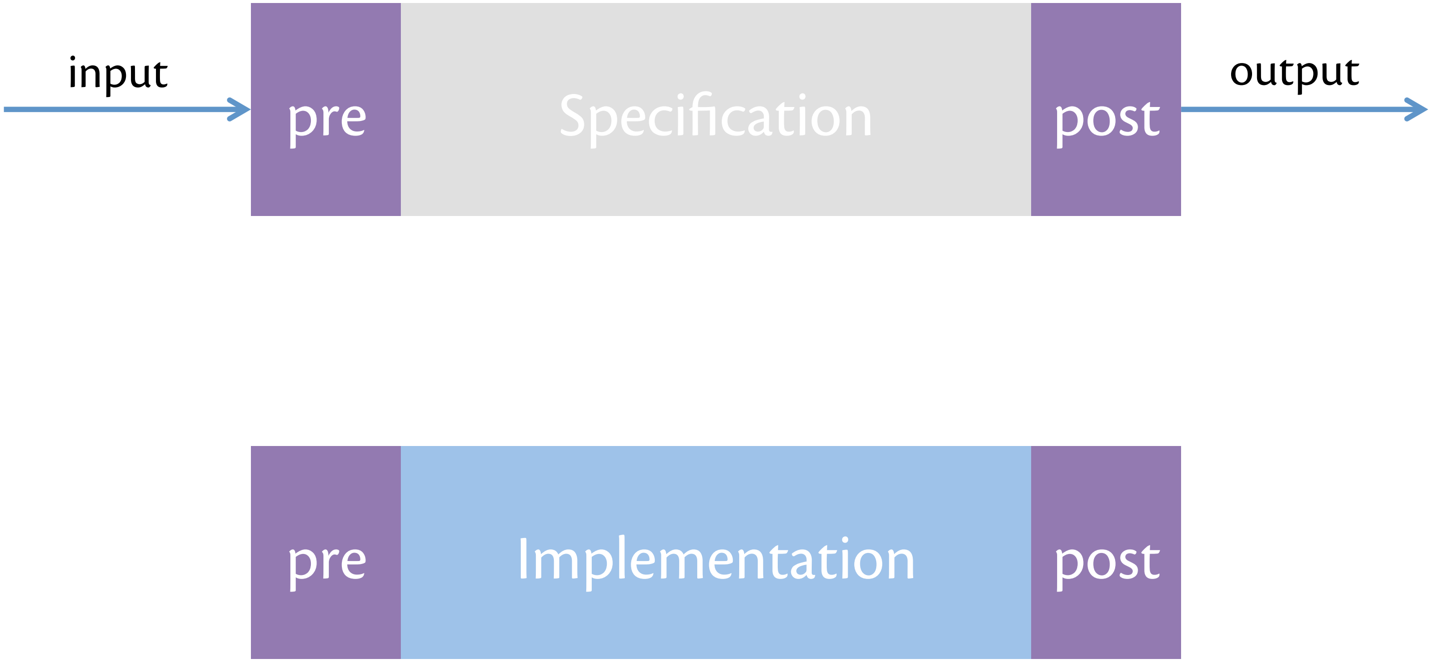
Q: How can we verify that SPEC2 refines SPEC1?

A: Prove that PRE1 ⇒ PRE2 and POST2 ⇒ POST1

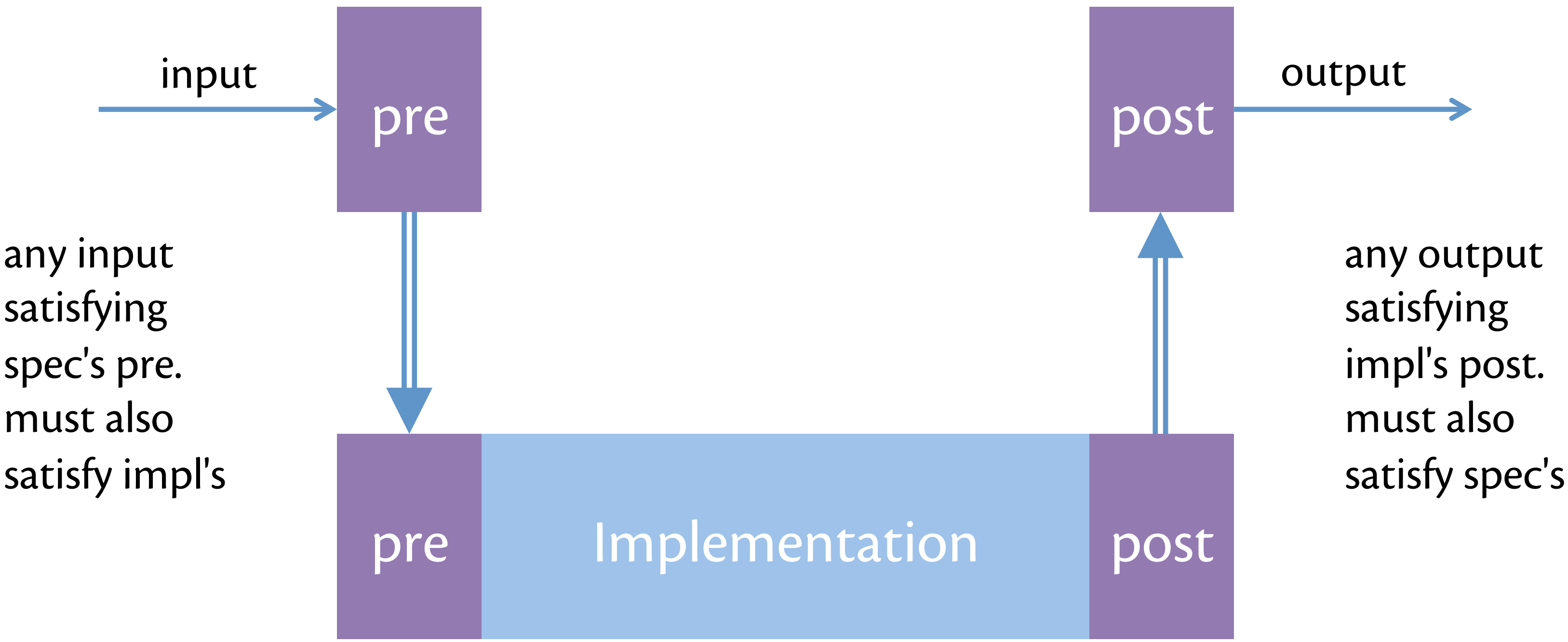
- PRE2 is weaker than or equivalent to PRE1
- POST2 is stronger than or equivalent to POST1

Refinement

Refinement


input
pre
Specification
post
output
pre
Implementation
post

Refinement


input
pre
post
output
any input
satisfying
spec's pre.
must also
satisfy impl's
any output
satisfying
impl's post.
must also
satisfy spec's
pre
Implementation
post

Refinement and assignments

- We give you a SPEC1 for an assignment
- 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
  - maybe assume more: strengthen the precondition
  - or guarantee less: weaken the postcondition
  - then you don't pass our test cases
  - so you are ☹

Refinement and assignments

- We give you a SPEC1 for an assignment
- 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 have to do some reimplementation
  - You are ☹
- In the real world, clients are going to refine specs on you *all the time*
- But we try not to make your life harder than necessary
  - Which is why one of my commandments to TAs is "Don't refine the spec."
  - And why whenever possible I tell you, "This is unspecified; do something reasonable."

Proof

- We worked only somewhat formally today
  - Wrote formulas involving *and, or, implies*
  - 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* and maybe special purpose logics for reasoning about programs (see CS 4110)

Upcoming events

- [Wednesday] MS1 due, no late submissions
- [Thursday-next Thursday] Design review meetings
- [next Thursday] Prelim 2; see Piazza post

This is specified.

THIS IS 3110