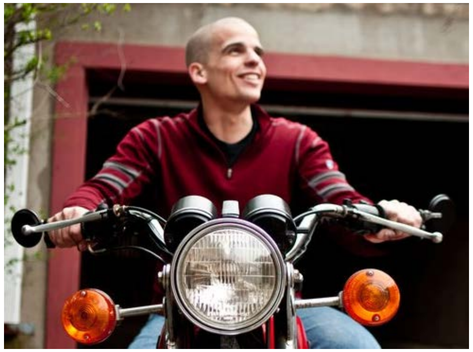
Automating Programming Assessments

What I Learned Porting 15-150 to Autolab

Iliano Cervesato

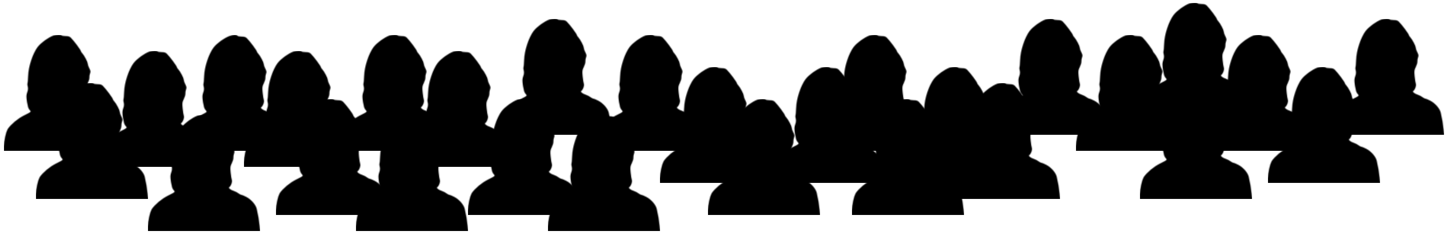
Thanks!

Bill Maynes



Ian Voysey

Jorge Sacchini



Generations of 15-150, 15-210 and 15-212 teaching assistants

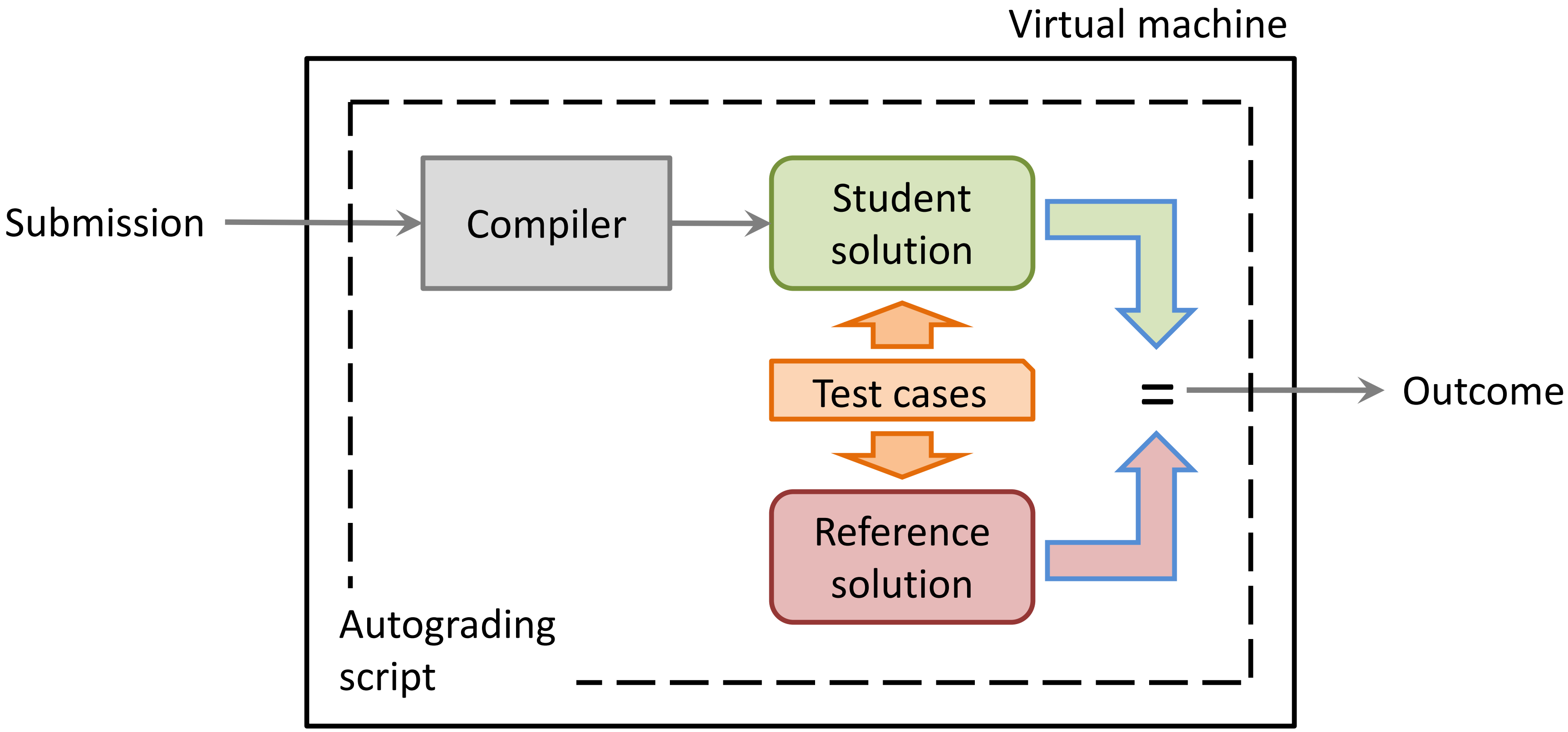
Outline

- Autolab
- The challenges of 15-150
- Automating Autolab
  - ➢ Test generation
- Lessons learned

AUTOLAB

- Tool to automate assessing programming assignments
  - ➢ Student submits solution
  - ➢ Autolab runs it against reference solution
  - ➢ Student gets immediate feedback
    - » Learns from mistakes while on task
- Used in 80+ editions of 30+ courses
- Customizable

How Autolab works, typically



The promises of Autolab

- Enhance learning
  - By pointing out errors while students are on task
  - *Not when the assignment is returned*
    - *Students are busy with other things*
    - *They don’t have time to care*
- Streamline the work of course staff … maybe
  - Solid solution must be in place from day 1
  - Enables automated grading
    - Controversial

15-150

Use the mathematical structure of a problem to program its solution

- Core CS course
- Programming and theory assignments

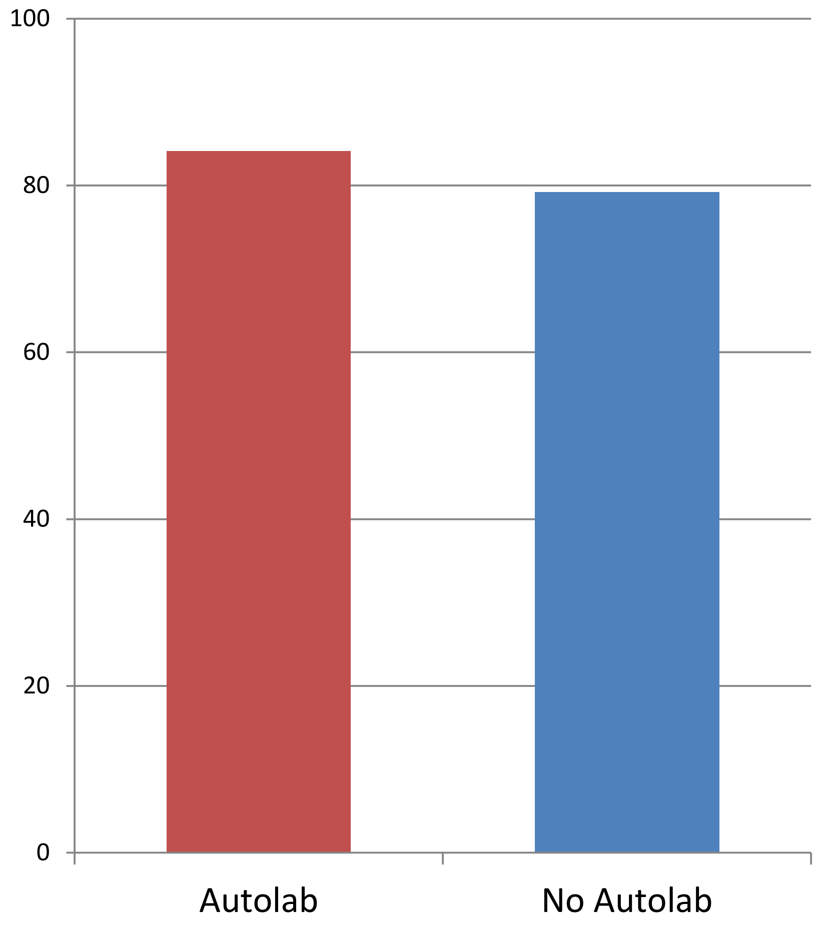
- Pittsburgh (x 2)
  - 150-200 students
  - 18-30 TAs

- Qatar
  - 20-30 students
  - 0-2 TAs

Autolab in 15-150

- Used as
  - ➢ Submission site
  - ➢ Immediate feedback for coding components
  - ➢ Cheating monitored via MOSS integration
- Each student has 5 to 10 submissions
  - ➢ Used 50.1% in Fall 2014
- Grade is *not* determined by Autolab
  - ➢ All code is read and commented on by staff

Effects on Learning in 15-150



- Insufficient data for accurate assessment
  - Too many other variables

- Average of the normalized median grade in programming assignments

The Challenges of 15-150

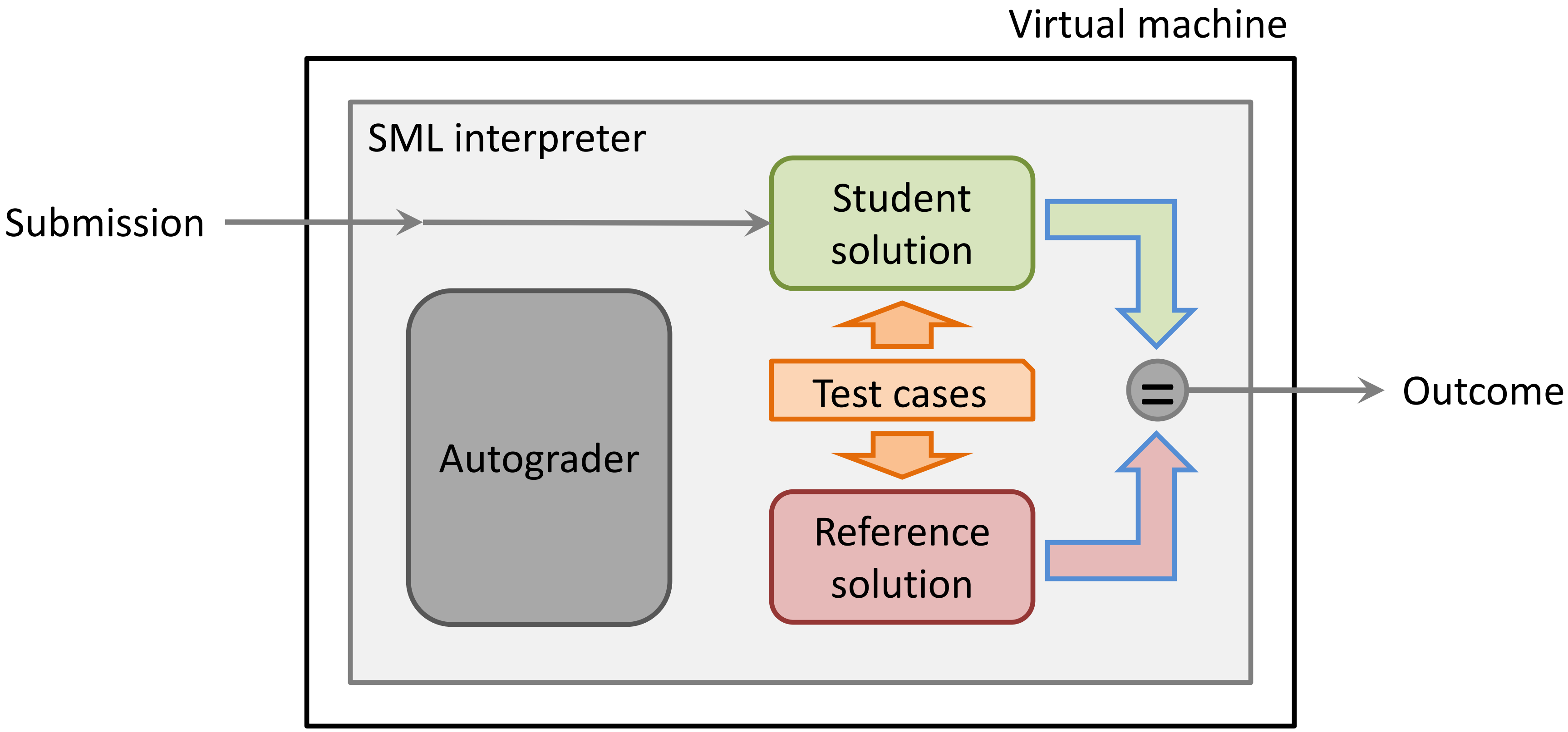
- 15-150 relies on Standard ML (common to 15-210, 15-312, 15-317, …)
    - ➢ Used as an *interpreted* language
        - » no I/O
    - ➢ Strongly typed
        - » No “eval”
    - ➢ Strict module system
        - » Abstract types
- 11, very diverse, programming assignments
    - ➢ Students learn about module system in week 6

Autograding SML code

- Traditional model does not work well
  - Requires students to write unnatural code
  - Needs complex parsing and other support functions
    - But SML already comes with a parser for SML expressions
- Instead, make everything happen *within* SML
  - running test cases
  - establishing outcome
  - dealing with errors

Student and reference code become modules

Running Autolab with SML

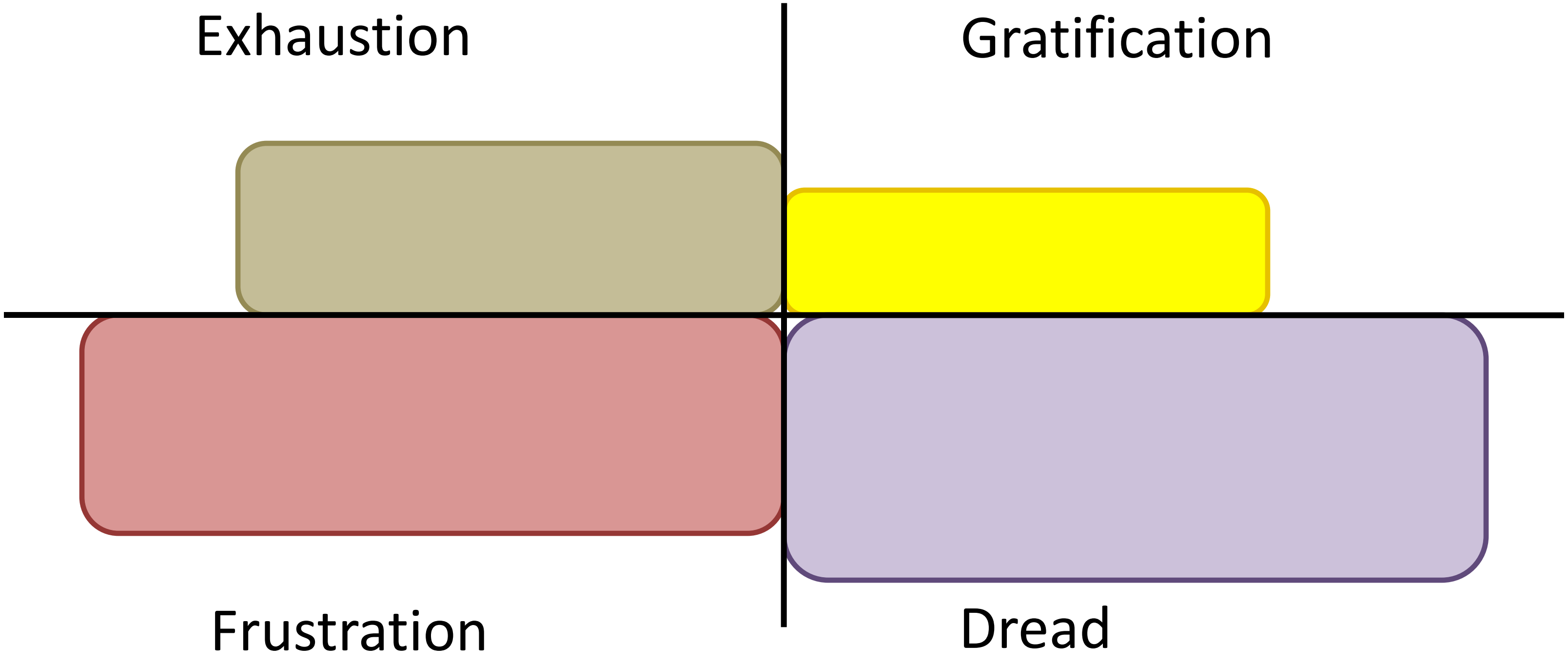


Making it work is non-trivial

- Done for 15-210
  - ➢ But 15-150 has much more assignment diversity
- No documentation
  - ➢ Initiation rite of TAs by older TAs
    - » Cannot work on the Qatar campus!
  - ➢ Demanding on the course staff
- TA-run
  - ➢ Divergent code bases

Too important to be left to rotating TAs

Autograder development cycle



Work of course staff hardly streamlined

What’s in a typical autograder?

```
grader.cm
handin.cm
handin.sml
autosol.cm
autosol.sml
HomeworkTester.sml
xyz-test.sml
aux/
  allowed.sml
  xyz.sig
  sources.cm
  support.cm
```

- A working autograder takes 3 days to write
  - ➢ Each assignment brings new challenges
  - ➢ Tedious, ungrateful job
  - ➢ Lots of repetitive parts
  - ➢ Cognitively complex
- Time taken away from helping students
- Discourages developing new assignments

(*simplified*)

However

```
grader.cm
handin.cm
handin.sml
autosol.cm
autosol.sml
HomeworkTester.sml
xyz-test.sml
aux/
   allowed.sml
   xyz.sig
   sources.cm
   support.cm
```

- Most files can be generated automatically from function types

- Some files stay the same

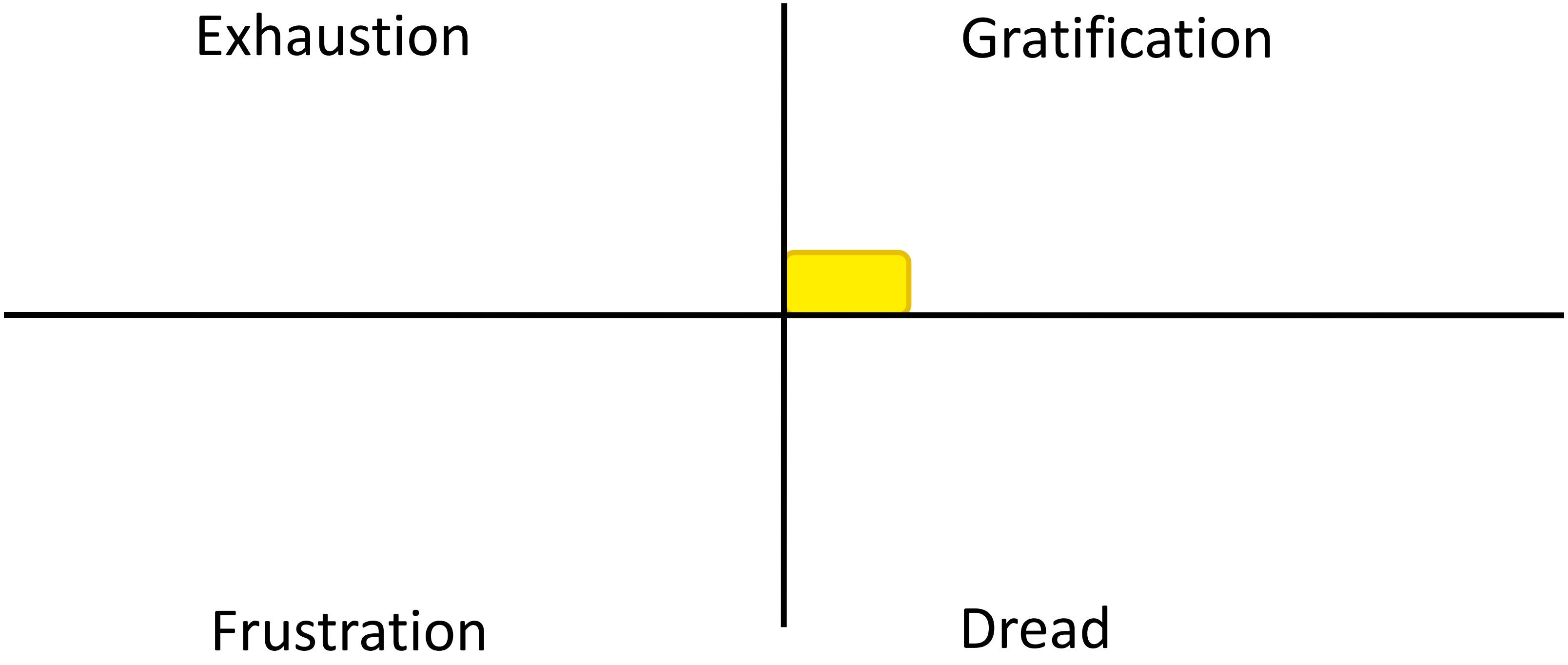
- Others are trivial
  - given a working solution

(*simplified*)

Significant opportunity for automation

- Summer 2013:
  - Hired a TA to deconstruct 15-210 infrastructure
- Fall 2013:
  - Ran 15-150 with Autolab
  - Early automation
- Fall 2014:
  - Full automation of large fragment
  - Documentation
- Summer 2015:
  - Further automation
  - Automated test generation
  - Fall 2015 was loaded on Autolab by first day of class

Is Autolab effortless for 15-150?



Exhaustion

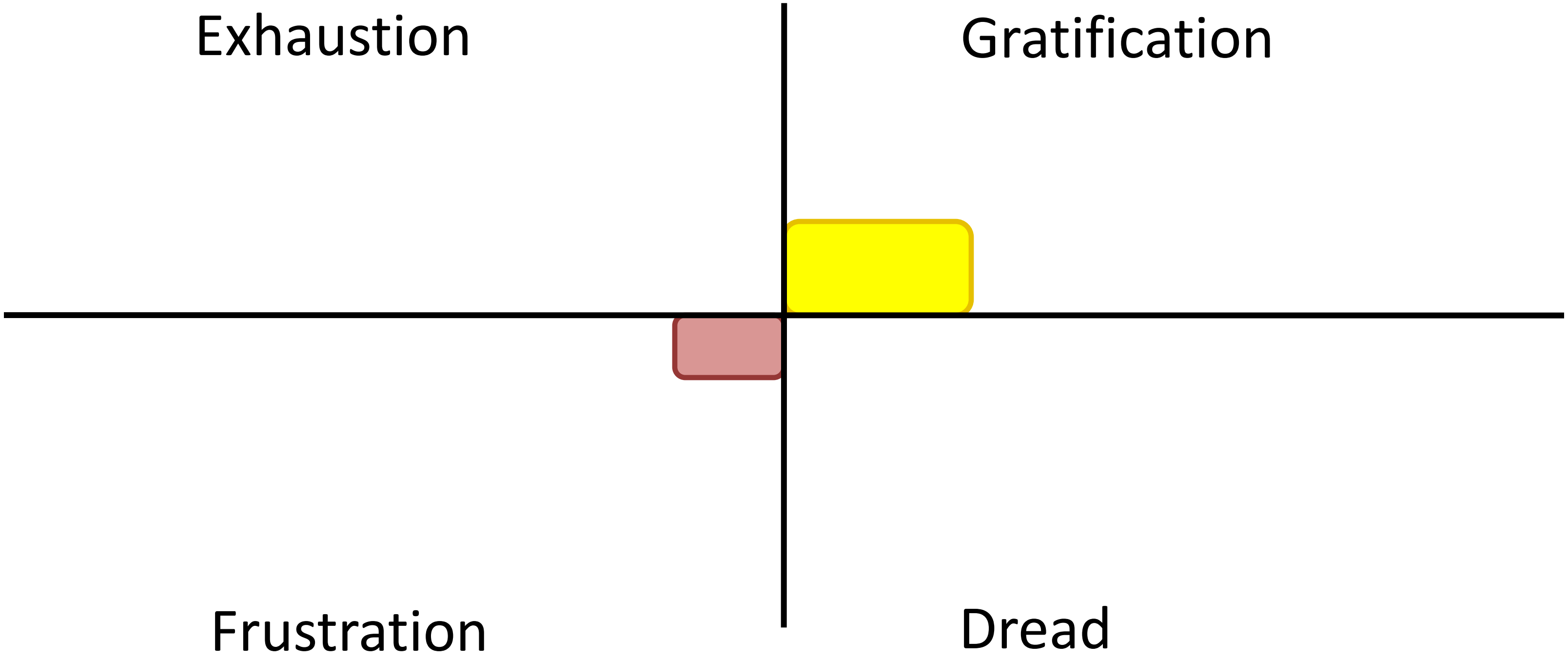
Gratification

Frustration

Dread

Not quite …

… but definitely streamlined



Automate what?

```
(* val fibonacci: int -> int *)
fun test_fibonacci () = OurTester.testFromRef
(* Input to string      *)  Int.toString
(* Output to string     *)  Int.toString
(* output equality      *)  op=
(* Student solution     *)  (Stu.fibonacci)
(* Reference solution   *)  (Our.fibonacci)
(* List of test inputs *)  (studTests_fibonacci @
                             (extra moreTests_fibonacci))
```

Printing

Equality

Tests

Automatically generated

- For each function to be tested,
  - Test cases
  - Equality function
  - Printing functions

Equality and Printing Functions

- Assembled automatically for primitive types
- Generated automatically for user-defined types

  New

  - Trees, regular expressions, game boards, …
- Placeholders for abstract types
  - Good idea to export them!

- Handles automatically
  - Polymorphism, currying, exceptions
  - Non-modular code

Example

```
(* datatype tree = empty | node of tree * string * tree *)
fun tree_toString (empty: tree): string = "empty"
  | tree_toString (node x) =
     "node" ^ ((U.prod3_toString (tree_toString,
                                  U.string_toString,
                                  tree_toString))        x)

(* datatype tree = empty | node of tree * string * tree *)
fun tree_eq (empty: tree, empty: tree): bool =  true
  | tree_eq (node x1, node x2) =
     (U.prod3_eq (tree_eq, op=, tree_eq)) (x1,x2)
  | tree_eq _ = false
```

Automatically generated

New

Test case generation

- Defines randomized test cases based on function input type
  - Handles functional arguments too
- Relies on QCheck library
- Fully automated
  - Works great!

Example

```
(* datatype tree = empty | node of tree * int * tree *)
fun tree_gen (0: int): tree Q.gen =
      Q.choose [Q.lift empty ]
  | tree_gen n =
      Q.choose'[(1, tree_gen  0),
                (4, Q.map node (Q.prod3 (tree_gen (n-1),
                                         Q.intUpto 10000,
                                         tree_gen (n-1)))) ]

(* val Combine : tree * tree -> tree *)
fun Combine_gen n = (Q.prod2 (tree_gen n, tree_gen n))

val Combine1 = Q.toList (Combine_gen 5)
```

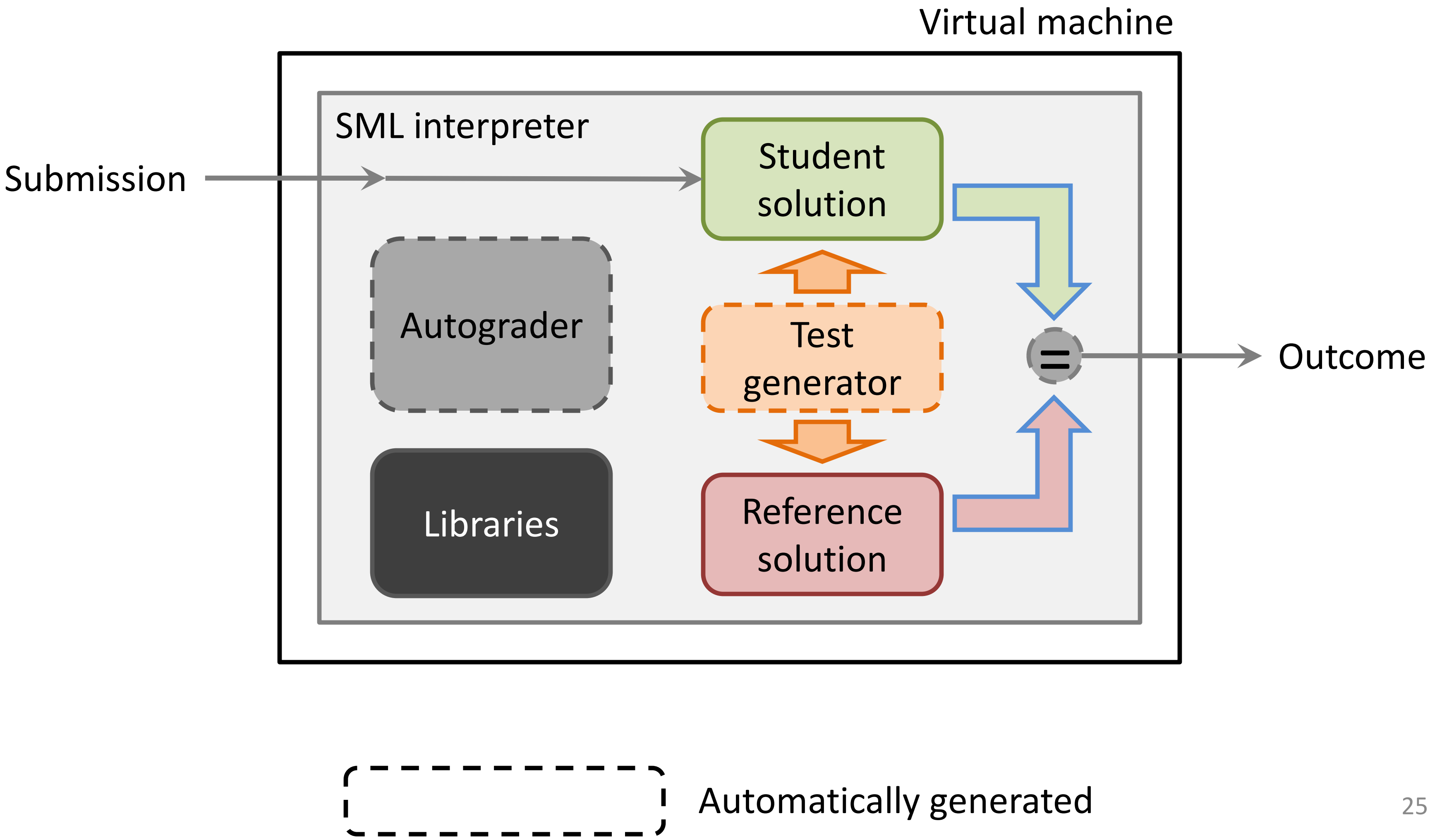
Mostly automatically generated

A more complex example

```
(* val permoPartitions: 'a list -> ('a list * 'a list) list *)
fun test_permoPartitions (a_ts) (a_eq) = OurTester.testFromRef
(* Input to string     *)  (U.list_toString a_ts)
(* Output to string    *)  (U.list_toString
                             (U.prod2_toString
                               (U.list_toString a_ts,
                                U.list_toString a_ts)))
(* output equality     *)  (U.list_eq
                               (U.prod2_eq
                                 (U.list_eq a_eq,
                                  U.list_eq a_q)))
(* Student solution    *)  (Stu.permoPartitions)
(* Reference solution  *)  (Our.permoPartitions)
(* List of test inputs *)  (studTests_permoPartitions @
                            (extra moreTests_permoPartitions))
```

Automatically generated

Current Architecture



Status

- Developing an autograder now takes from 5 minutes to a few hours
  - 3 weeks for all Fall 2015 homeworks, including selecting/designing the assignments, and writing new automation libraries
- Used also in 15-312 and 15-317
- Some manual processes remain

Manual interventions

- Type declarations
  - ➢ Tell the autograder they are shared
- Abstract data types
  - ➢ Marshalling functions to be inserted by hand
- Higher-order functions in return type
    - » E.g., streams
  - ➢ Require special test cases

- Could be further automated
  - ➢ Appear in minority of assignments
  - ➢ Cost/reward tradeoff

Example

```
(* val map : (''a -> ''b) -> ''a set -> ''b set *)
fun test_map (a_ts, b_ts) (b_eq) = OurTester.testFromRef
(* Input to string     *)  (U.prod2_toString
                            (U.fn_toString a_ts b_ts,
                             (Our.toString a_ts) o Our.fromList))
(* Output to string    *)  ((Our.toString b_ts) o Our.fromList)
(* output equality     *)  (Our.eq o (mapPair Our.fromList))
(* Student solution    *)  (Stu.toList o (U.uncurry2 Stu.map)
                             o (fn (f,s) => (f, Stu.fromList s)))
(* Reference solution  *)  (Our.toList o (U.uncurry2 Our.map)
                             o (fn (f,s) => (f, Our.fromList s)))
(* List of test inputs *)  (studTests_map @
                            (extra moreTests_map))
```

Mostly automatically generated

Tweaking test generators

- Invariants
  - ➢ Default test generator is unaware of invariants
    - » E.g., factorial: input should be non-negative
- Overflows
    - » E.g., factorial: input should be less than 43
- Complexity
    - » E.g., full tree better not be taller than 20-25

- Still: much better than writing tests by hand!

About testing

- Writing tests by hand is tedious
  - Students hate it
    - Often skip it even when penalized for it
  - TAs/instructors do a poor job at it
- Yet, testing reveals bugs
- Manual tests are skewed
  - Few, small test values
  - Edge cases not handled exhaustively
  - Subconscious bias
    - Mental invariants

Future Developments

- Better test generation through annotations
  - E.g., 15-122 style contracts
- Automate a few more manual processes
- Overall architecture can be used with other languages
- Let students use the test generators
  - Currently too complex

To autograde or not to autograde?

- So far, Autolab has be an aid to grading
- Could be used to determine grades automatically in programming assignments
  - Impact on student learning?
  - Cheating?
  - Enable running 15-150 with fewer resources

15-150 beyond programming

- Proofs
  - Students don’t like induction, but don’t mind coding
  - Modern theorem provers turn writing a proof into a programming exercise
    - Can be autograded
- Complexity bounds
  - Same path?

Lessons learned

- Automated grading support helped me run a better course
- Writing an autograder generator is a lot more fun than writing an autograder
- Room for further automation
  - ➢ Work really hard to do less work later
- Automated test generation is great!

Questions?

Other pedagogic devices

- Bonus points for early submissions
  - ➢ Encourages good time management
  - ➢ Lowers stress
- Corrected assignments returned individually
  - ➢ Helps correct mistakes
- Grade forecaster
  - ➢ Student knows exactly standing in the course
  - ➢ What-if scenarios