Overview

Applying Formal Methods, Part IV

Overview

Property testing: a case study

- An implementation of the queue data type
- QuickCheck

Queues

- A queue is a first-in, first-out (FIFO) data structure with the enqueue and dequeue operations; the data can be stored in:
  1. A list
  2. Two stacks (how?)
- What are the properties that a queue should have? How do you test if the properties hold?

The End

The Queue Data Type

Representation I

An Abstract Model for Queue

A FIFO data structure with the following supporting operations:

Operation	Actions performed
add (i.e., enqueue)	Add an element to the queue.
remove (i.e., dequeue)	Remove an element from the queue.
isEmpty	Test if the current queue is empty.
front	Return (but not remove) first element from the queue.
empty	Return an empty queue.

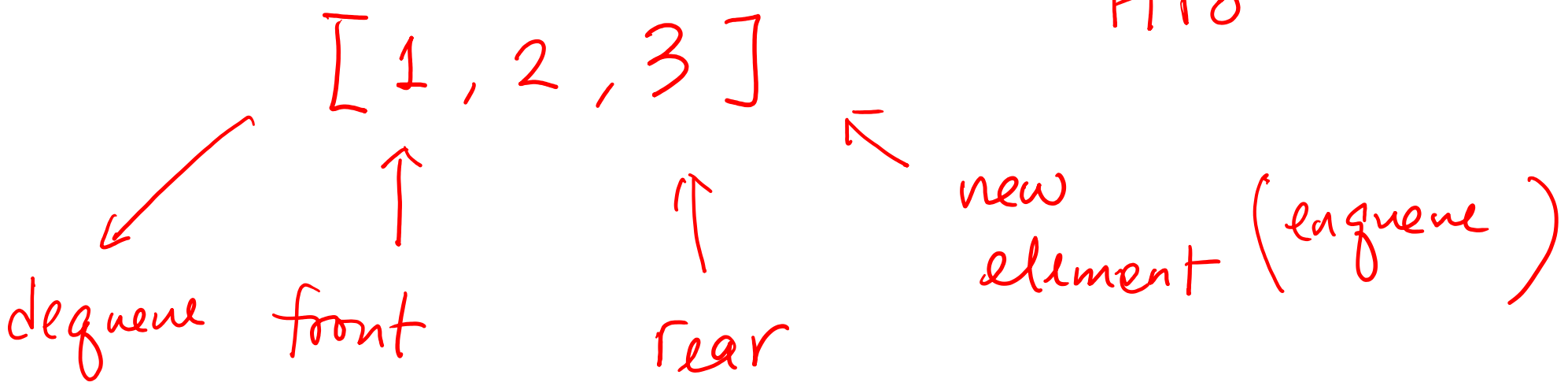
Specification

Data	Haskell representation
q: a List of type a	type Queue a = [a]

Operation	Haskell implementation
enqueue (add)	add x q = q ++ [x]
dequeue (remove)	remove (x:q) = q
isEmpty	isEmpty q = null q
front	front (x:q) = x
empty	empty = [ ]

Queue (as a List)

FIFO



Illustration

- Action: empty queue -> add 1 -> add 2 - > add 3 -> remove
- [ ] (Empty queue) -> add 1 -> [1] -> add 2 -> [1, 2] -> add 3 -> [1, 2, 3] -> remove -> [2, 3]

The End

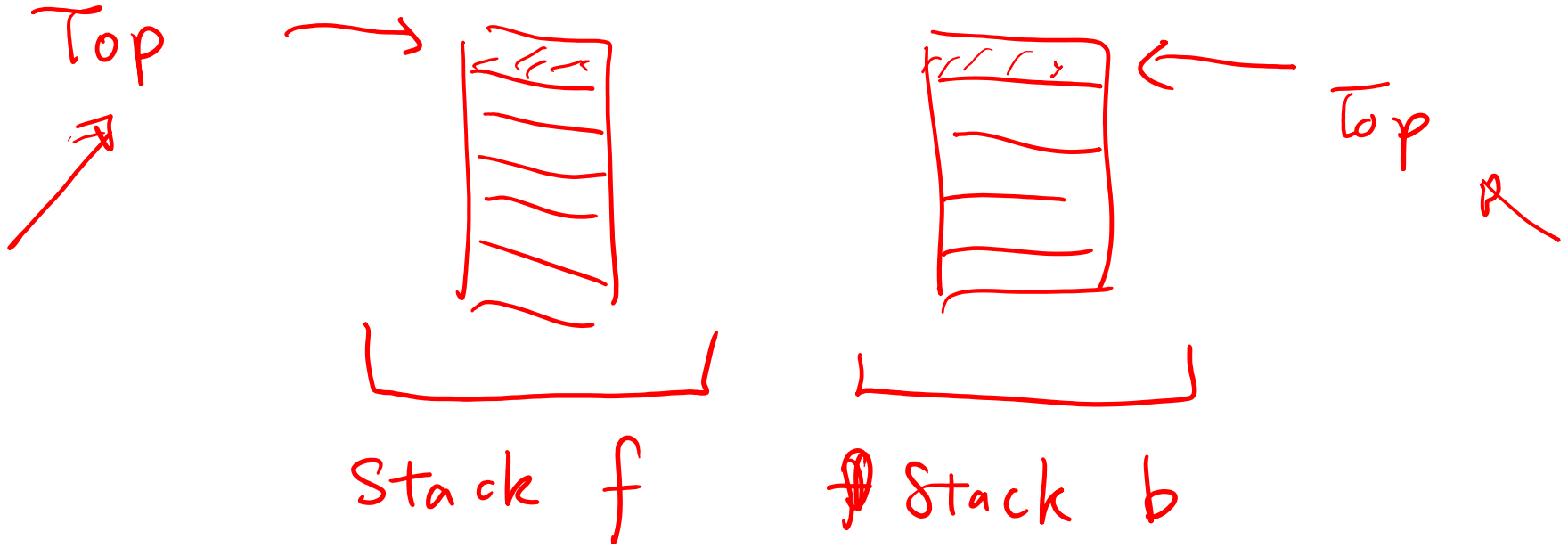
The Queue Data Type

Representation II

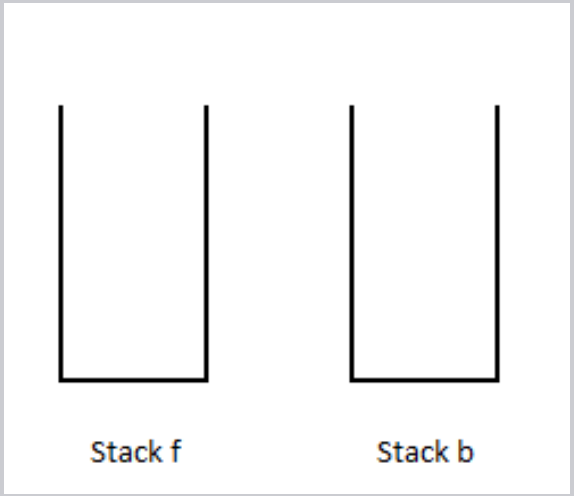
An Alternate Representation

Use two stacks to represent a single queue.

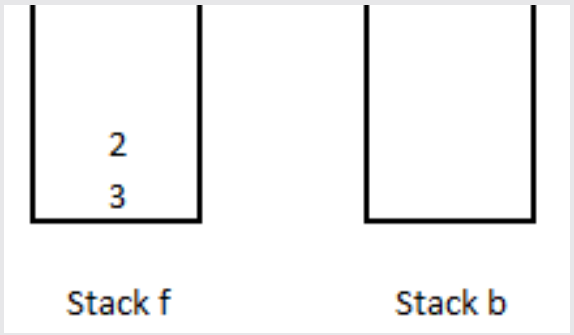
- Motivation: operations (e.g., add) can be more efficient.
- How do you maintain the FIFO property?
- When will elements be moved from one stack to another?


Top
Top
Stack f
Stack b

A Queue via Two Stacks (Illustration)

Queue via two stacks	Rules
	• Empty queue: Both stacks are empty. • Dequeue: Always return the top element from stack f. • Whenever stack f is empty, we pop element from stack b one by one to stack f until stack b is empty. • Enqueue: Always push the new element to stack b and check if stack f is empty. if so, we pop elements from stack b one by one to stack f until stack b is empty.

Action: empty queue -> add 1 -> add 2 - > add 3 -> remove



([ ],[ ]) -> add 1 -> ([ ],[1]) -> (stack f is empty) -> ([1],[ ])
([1],[ ]) -> add 2 -> ([1],[2]) -> add 3 -> ([1],[3,2])
([1],[3,2]) -> remove -> ([ ],[3,2]) -> (stack f is empty) -> ([2,3],[ ])

Specification: First Attempt

Data	Haskell representation
(f , b) : a tuple of lists (2 stacks). f, b is of type [a]	type QueueI a = ([a],[a])

Operation	Haskell implementation (~~second~~ first attempt)
addI (i.e., enqueue)	addI x (f , b) =
removeI (i.e., dequeue)	removeI (x:f , b) =
isEmptyI	isEmptyI (f , b) =
frontI	frontI (x:f, b) =
emptyI	emptyI = ([] , [])

The End

QuickCheck and Property Testing

Introduction

Introduction, Part I

What is QuickCheck?

- An automatic testing tool for Haskell programs
- Specification (prepared by the programmer)
  - Written in Haskell, using tools from QuickCheck library
  - A list of properties that the functions must satisfy
- Testing: check if the properties hold for many randomly generated cases

Introduction, Part II

QuickCheck can:

- Define and execute tests for properties specified
- Define test data generators and show test data distribution
- Assist and guide software design (e.g., debugging), help software documentation, and reuse
- Encourage programmers to (learn to) formulate precise formal specifications and use testing results for software development

Introduction, Part III

QuickCheck provides examples of domain-specific embedded languages (Hutton, Chapter 7); they are:

- A formal specification language to write tests directly into the source code
- A test data generation language to give compact description of many randomly generated tests for each property specified

Introduction, Part III

QuickCheck provides examples of domain-specific embedded languages (Hutton, Chapter 7); they are:

- A formal specification language to write tests directly into the source code
- A test data generation language to give compact description of many randomly generated tests for each property specified

The End

More on the Queue Data Type

Algebraic Specification

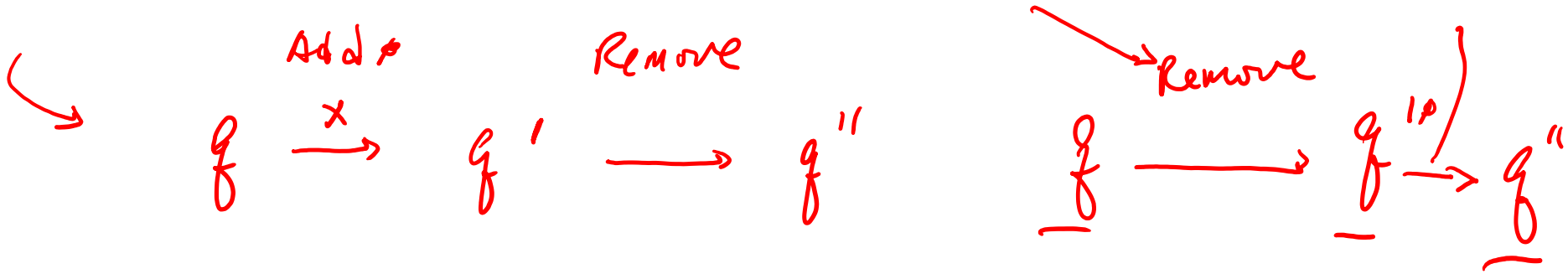
Algebraic Specification

- Any queue can be expressed as an algebraic expression:

  addI x_1 (addI x_2 … (addI x_k emptyI) …)

- Properties of a queue can be expressed as an algebraic expression; for example, for any queue $q \neq \emptyset$ and x of type a

  remove (addI x q) = addI x (remove q)



Specification: Second Attempt

Data	Haskell representation
(f , b) : a tuple of lists (2 stacks). f, b is of type [a]	type QueueI a = ([a],[a])

Operation	Haskell implementation (second attempt)	flipQ function
addI (i.e., enqueue)	addI x (f , b) = flipQ (f , x:b)	flipQ ([ ], b) = (reverse b, [ ]) flipQ q = q if stack f is empty, we pop elements from stack b one by one to stack f until stack b is empty
removeI (i.e., dequeue)	removeI (x:f , b) = flipQ (f , b)	
isEmptyI	isEmptyI (f , b) = null f	
frontI	frontI (x:f, b) = x	
emptyI	emptyI = ([ ], [ ])	

(*)

A property that holds for (f , b) at all times	Transform a queue: representation 2 to representation 1
invariant :: QueueI Integer -> Bool invariant (f , b) = not (null f) | null b	retrieve :: QueueI Integer -> [Integer] retrieve (f , b) = f ++ reverse b

QuickCheck, Part I

- Property: for any queue $q \neq \emptyset$ and x of type a

  removeI (addI x q) = addI x (removeI q)
- Specification: Prop_remove_add x q = invariant q && not (isEmptyI q) ==> removeI (addI x q) = addI x (removeI q)

QuickCheck, Part II

- Results

```
prompt> QuickCheck prop_remove_add
*** Failed! Falsified (after 1 test and 2 shrinks):
0
([0],[0])
```

counter-example

- Reason(s): discussion

x = 0

qerror = ([0],[0])

QuickCheck, Part III

Intermediate steps	Remarks
prompt> invariant qerror True prompt> isEmptyI qerror False prompt> not (isEmptyI qerror) True prompt> removeI (addI 0 qerror) ([0,0],[]) prompt> addI 0 (removeI qerror) ([0],[0])	• ([0,0],[ ]) = ([0],[0]) is false • ([0,0],[ ]) represents the queue [0,0] • ([0],[0]) represents the queue [0,0] retrieve [0,0] Hence, we find out that we should test if two sides of the equation are equivalent (represent the same queue). qerror = ([0],[0]) x = 0

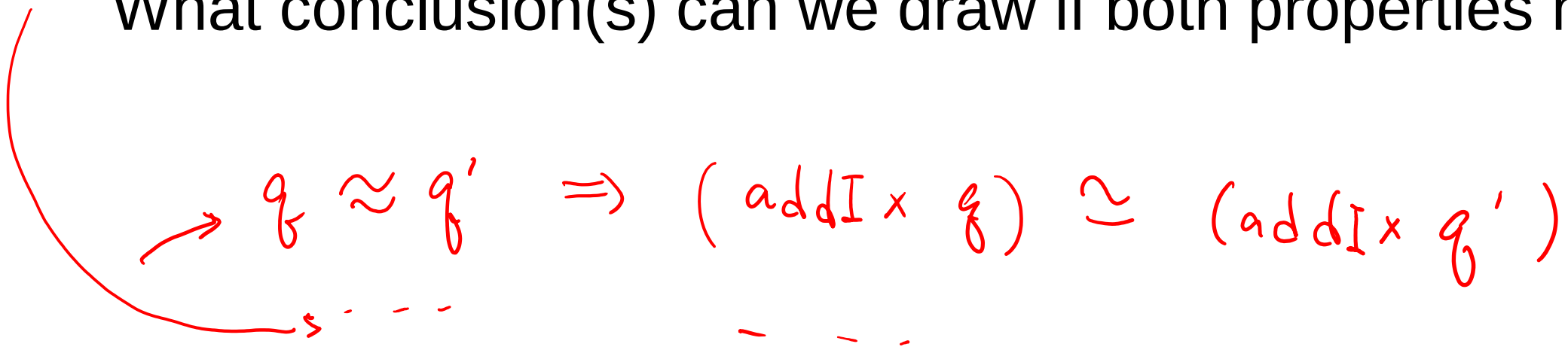
Revised Version

- Definition
  - $q, q' \; are \; equivent \;\; invariant\,(q), invariant\,(q'), and \; retreive\,(q) == retrieve\,(q')$
- QuickCheck property
  - $q \; `equiv` \; q' = \;\; invariant\,(q) \;\&\&\; invariant\,(q') \;\&\&\; retreive(q) == retreive\,(q')$

The Equivalence Property

- $\forall\ q, q'\ and\ \forall\ x,$
  - $q\ `equiv`\ q' \implies (addI\ x\ q)`equiv`\ (addI\ x\ q')$
- $\forall\ q, q'\ (q, q' \neq \emptyset),$
  - $q\ `equiv`\ q' \implies (removeI\ q)`equiv`\ (removeI\ q')$

What conclusion(s) can we draw if both properties hold?



The End

Weekly Summary

Apply Formal Methods IV

Summary

The case study

- Discuss the background on queue implementation(s).
- Walk through the program development for queues.
- Show how to formulate properties (specification) that are written in Haskell and are executable.
- Demonstrate how property testing can guide the software-development process.

The Software Tool QuickCheck

- Is an example of domain specific embedded language that support random testing of program properties
- Encourages using formal methods (e.g., automatic property testing, formal specification, etc.) for program development

Note: Re-implementations of QuickCheck exist for several languages.

The End