COMPSCI 141

Liam O'Conn
University of Edinburgh, LFCS (an

Term 2, 2020

Motivation

We've already see

How do we come up with correctness properties in the first place?

Structure of a Module

A Haskell program will usually be made up of many modules, each of which exports one or more *data types*.
Typically a module for a data type `X` will also provide a set of functions, called *operations*, on

- `X`
- `X`
- to update the data type:

A lot of software can be designed with this structure.

Example (Data Types)

A dictionary data type, with empty, insert and lookup.

Data Invariants

One source of properties is *data invariants*.

Data Invariants

Data invariants a

Whenever we use a

invariants are mai

Example

- That a list of words in a dictionary is always in sorted order.
- That a binary tree satisfies the search tree properties.
- That a date value will never be invalid (e.g. 31/13/2019).

Properties for Data Invariants

For a given data type `X`, we define a *wellformedness predicate*

$$\texttt{wf} :: \texttt{X} \to \texttt{Bool}$$

For a given value x.

Properties

For each operation, if all input values of type `X` satisfy `wf`.

In other words, for each constructor operation c $\texttt{wf}\,(c\ \cdots)$, and for each update operation $u :: \texttt{X} \to \texttt{X}$ we must show $\texttt{wf}\ x \implies \texttt{wf}(u\ x)$

Demo: Dictionary example, sorted order.

Stopping External Tampering

Even with our sorted dictionary example, there's nothing to stop a malicious or clueless programmer fro

Example

The malicious programmer could just add a word directly to the dictionary, unsorted, bypassing our carefully written `insert` function.

We want to prevent this sort of thing from happening.

Abstract Data Types

An *abstract* data type (ADT) is a data type where the implementation details of the type and its associated operations are hidden.


```
newtype Dict
type Word = Strin
type Definit
emptyDict   :: Dict
insertWord :: Word -> Definition -> Dict -> Dict
lookup      :: Word -> Dict -> Maybe Definition
```


If we don't have access to the implementation of the provided operations, which we know preserve our data inv invariants cannot be violated if this module is correct.

Demo: In Haskell, we make ADTs with module headers.

Abstract? Data Types

In general, *abs*

The inverse of abs

Abstract data typ

implementation details are hidden, and we no longer have to re

level of implementation.

Validation

Suppose we had a `sendEmail` function:

```
sendEmail :: String -- email address
          -
          -
```


It is possible to mix th t, it's possible that the given email address is not valid.

Question

Suppose that we wanted to make it impossible to call t first checking that the email address was valid.
How would we accomplish this?

Validation ADTs

We could define a tiny ADT for validated email addresses, where the data invariant is that the contained email address is valid:


```
module EmailADT(Email, checkEmail, sendEmail)
    newtype E

    checkEm
    checkEm
                  | otherwise       = Nothing
```


Then, change the type of `sendEmail`:


```
sendEmail :: Email -> String -> IO()
```

The only way (outside of the `EmailADT` module) to create a value of type `Email` is to use `checkEmail`.

`checkEmail` is an example of what we call a *smart constructor*: a constructor that enforces data invariants.

Reasoning about ADTs

Consider the following, more traditional example of an ADT interface, the unbounded queue:


```
data Queue

emptyQueue :: Q
enqueue :: Int -> Q
front   :: Queue -> Int
dequeue :: Queue -> Queue -- partial
size    :: Queue -> Int
```


We could try to come up with properties that relate these functio without reference to their implementation, such as:

```
dequeue (enqueue x emptyQueue) == emptyQueue
```

However these do not capture functional correctness (usually).

Models for ADTs

We could imagine a simple implementation for queues, just in terms of lists:

```
emptyQueueL = []
enqueueL a  = (++ [a])
frontL      = head
dequeueL    = tail
sizeL       = length
```

But this implementation is $\mathcal{O}(n)$ to enqueue! Unacce

However!

This is a dead simple implementation, and trivial to see that it is cor
better queue implementation, it should always give the same results as this simple one.
Therefore: This implementation serves as a functional correctness specification for our `Queue` type!

Refinement Relations

The typical approach to connect our model queue to our `Queue` type is to define a relation, called a *refinement relation*, that relates a `Queue` to a list and tells us if the two structures represent the same queue conceptually:

```
rel :: Queue -> [In
```

Then, we show tha

```
prop_empty_
```

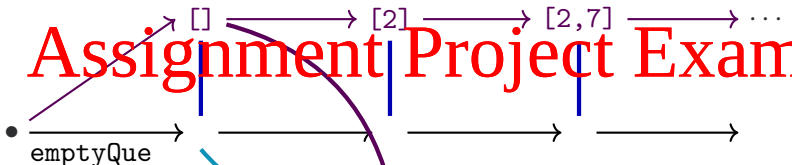
That any query functions for our two types produce equal result
such as for size:

```
prop_size_r fq lq = rel fq lq ==> size fq = size lq
```

And that each of the queue operations preserves our refinement relation, for example for enqueue:

```
prop_enq_ref fq lq x =
  rel fq lq ==> rel (enqueue x fq) (enqueueL x lq)
```

In Pictures




```
prop_enq_ref fq lq x =
  rel fq lq ==> rel (enqueue x fq) (enqueueL x lq)
prop_empty_r = rel emptyQueue emptyQueue
prop_size_r fq lq = rel fq lq ==> size fq == sizeL lq
```

Whenever we use a Queue, we can reason as if it were a list!

Abstraction Functions

These refinement relations are very difficult to use with QuickCheck because the `rel fq lq` preconditions are very hard to satisfy with randomly generated inputs.

For this example, it

the correspondin

Conceptually, our refinement relation is then just:


```
\fq lq → absfun fq =
```

However, we can re-express our properties in a much more QC-friendly format (**Demo**)

Fast Queues

Let's use test-driven development! We'll implement a fast Queue with amortised $\mathcal{O}(1)$ operations.

```
data Queue
```



```
       Int    -- size of the rear
```

We store the rear part of the queue in reverse order, to make enque

Thus, converting from our Queue to an abstract list re

```
toAbstract :: Queue -> [Int]
toAbstract (Q f sf r sr) = f ++ reverse r
```

Data Refinement

These kinds of properties establish what is known as a *data refinement* from the abstract, slow list model to the fast, complex queue implementation.

Refinement and Specifications

In general, all funct

1. all data inv
2. the implem

There is a limit to the amount of abstraction we can do before they b testing (but not necessarily for proving).

Warning

While abstraction can simplify proofs, abstraction does not reduce the fundamental complexity of verification, which is provably hard.

Data Invariants for Queue

In addition to the already-stated refinement properties, we also have some data invariants to maintain for a value `Q f sf r sr`:

1. `length f == sf`
2. `length r == sr`
3. **importan**

We will ensure our ese invariants.

Thus, our `wellformed` predicate is used merely to enfo outputs of our operations:

```
prop_wf_empty = wellformed (emptyQueue)
prop_wf_enq q = wellformed (enqueue x q)
prop_wf_deq q = size q > 0 ==> wellformed (dequeue q)
```

Implementing the Queue

We will generally implement by:

- Dequeue fr
- Enqueue to t
- If necessary
  and appending it to the front.

  This step is slow ($O(n)$) but only happens every g
  average case amortised complexity of $\mathcal{O}(1)$ ti

Amortised Cost

```
enqueue x (Q f sf r sr) = inv3 (Q f sf (x:r) (sr + 1))
```

When we enqueue each of $[1..7]$ to the empty queue in turn:

	Q []	0	[]	0
				0 (
				1
$\rightarrow$	Q [1, 2, 3]	3	[5	
$\rightarrow$	Q [1, 2, 3]	3	[6	
$\rightarrow$	Q [1, 2, 3, 4, 5, 6, 7]	7	[]	0 (

Observe that the slow invariant-reestablishing step $(*)$ happens after 1 step, then 2, then 4...

Extended out, this averages out to $\mathcal{O}(1)$.

Another Example

Consider this ADT interface for a bag of numbers:


```
data Bag
emptyBag    :: Bag
addToBag    :: Int -> Ba
averageBag  :: B
```


Our conceptual a

```
emptyBagA = []

addToBagA x xs = x:xs

averageBagA [] = Nothing
averageBagA xs = Just (sum xs `div` length xs)
```


But do we need to keep track of all that information in our implementation? No!

Concrete Implementation

Our concrete version will just maintain two integers, the total and the count:


```
data Bag = B { total :: Int , count :: Int }
emptyBag :: Bag
emptyBag = B 0 0

addToBag :: Int -> Bag -> Bag
addToBag x (B t c) = B (x + t) (c + 1)

averageBag :: Bag -> Maybe Int
averageBag (B _ 0) = Nothing
averageBag (B t c) = Just (t `div` c)
```

Refinement Functions

Normally, writing an abstraction function (as we did for `Queue`) is a good way to express our refine

write such a functi


```
toAbstract :: B
toAbstract (B t c) = ?????
```

Instead, we will go in the other direction, giving us a :


```
toConc :: [Int] -> Bag
toConc xs = B (sum xs) (length xs)
```

Properties with Refinement Functions

Refinement functions produce properties much like abstraction functions, only with the abstract and concrete layers swapped:


```
prop_ref_em
   toConc emA
```



```
prop_ref_add x ab =
   toConc (addToBagA x ab) == addToBag x (toConc ab)

prop_ref_avg ab =
   averageBagA ab == averageBag (toConc ab)
```

Assignment 1 and Break

Assignment 1 has b

It is due right before t

Advice from Alu

The assignments do not involve much coding, but they do invol

Start early!

Homework

1. Get started o
2. Next progr
3. Last week's
4. This week's quiz is also up, due the following Friday.