COMP3141

Software System Design and Implementation

Data Invariants, Abstraction and Refinement

Liam O'Connor
University of Edinburgh LFCS (and UNSW)

Term 2 2020

Motivation

We've already seen how to prove and test correctness properties of our programs.

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`.

- to construct the data type: $c :: \cdots \rightarrow \mathtt{X}$
- to query information from the data type: $q :: \mathtt{X} \rightarrow \cdots$
- to update the data type: $u :: \cdots \mathtt{X} \rightarrow \mathtt{X}$

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 are properties that pertain to a particular data type.
Whenever we use operations on that data type, we want to know that our data invariants are maintained.

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*

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

For a given value x :: `X`, `wf` x returns true iff our data invariants hold for the value x.

Properties

For each operation, if all input values of type `X` satisfy `wf`, all output values will satisfy `wf`.
In other words, for each constructor operation $c :: \cdots \rightarrow \texttt{X}$, we must show $\texttt{wf}\ (c\ \cdots)$, and for each update operation $u :: \texttt{X} \rightarrow \texttt{X}$ 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 from going in and mucking up our data invariants.

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 = String
type Definition = String
emptyDict  :: Dict
insertWord :: Word -> Definition -> Dict -> Dict
lookup     :: Word -> Dict -> Maybe Definition
```

If we don't have access to the implementation of Dict, then we can only access it via the provided operations, which we know preserve our data invariants. Thus, our data invariants cannot be violated if this module is correct.

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

Abstract? Data Types

In general, *abstraction* is the process of eliminating detail.

The inverse of abstraction is called *refinement*.

Abstract data types like the dictionary above are abstract in the sense that their implementation details are hidden, and we no longer have to reason about them on the level of implementation.

Validation

Suppose we had a sendEmail function

```
sendEmail :: String -- email address
          -> String -- message
          -> IO ()  -- action (more in 2 wks)
```


It is possible to mix the two String arguments, and even if we get the order right, it's possible that the given email address is not valid.

Question

Suppose that we wanted to make it impossible to call sendEmail without 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:

```haskell
module EmailADT(Email, checkEmail, sendEmail)
  newtype Email = Email String

  checkEmail :: String -> Maybe Email
  checkEmail str | '@' `elem` str = Just (Email str)
                 | otherwise      = Nothing
```

Then, change the type of sendEmail:

```haskell
  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 :: Queue
enqueue    :: Int -> Queue -> Queue
front      :: Queue -> Int    -- partial
dequeue    :: Queue -> Queue  -- partial
size       :: Queue -> Int
```


We could try to come up with properties that relate these functions to each other 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! Unacceptable!

However!

This is a dead simple implementation, and trivial to see that it is correct. If we make a 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 -> [Int] -> Bool
```

Then, we show that the refinement relation holds initially:


```
prop_empty_r = rel emptyQueue emptyQueueL
```

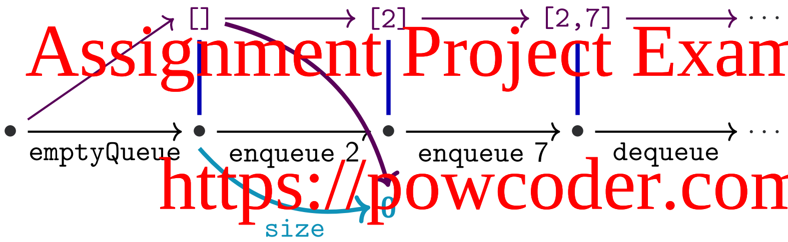
That any query functions for our two types produce equal results for related inputs, such as for size:


```
prop_size_r fq lq = rel fq lq ==> size fq == sizeL 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 ==> ref (enqueue x fq) (enqueueL x lq)
    prop_empty_r = rel emptyQueue emptyQueueL
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's a lot easier if we define an abstraction function that computes the corresponding abstract list from the concrete Queue.


```
toAbstract :: Queue → [Int]
```

Conceptually, our refinement relation is then just:


```
\fq lq → absfun fq == lq
```

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 = Q [Int] -- front of the queue
               Int   -- size of the front
               [Int] -- rear of the queue
               Int   -- size of the rear
```

We store the rear part of the queue in reverse order, to make enqueueing easier.

Thus, converting from our Queue to an abstract list requires us to reverse the rear:

```
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, concrete Queue implementation.

Refinement and Specifications

In general, all functional correctness specifications can be expressed as:

1. all data invariants are maintained, and
2. the implementation is a refinement of an abstract correctness model.

There is a limit to the amount of abstraction we can do before they become useless for 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. **important**: $sf \geq sr$ — the front of the queue cannot be shorter than the rear.

We will ensure our `Arbitrary` instance only ever generates values that meet these invariants.

Thus, our `wellformed` predicate is used merely to enforce these data invariants on the 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 from the front.
- Enqueue to the rear.
- If necessary, re-establish the third data invariant by taking the rear, reversing it, and appending it to the front.

This step is slow ($\mathcal{O}(n)$), but only happens every n operations or so, giving an average case amortised complexity of $\mathcal{O}(1)$ time.

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 `emptyQueue` in turn:

	Q	[]	0	[]	0	
$\rightarrow$	Q	[1]	1	[]	0	$(*)$
$\rightarrow$	Q	[1]	1	[2]	1	
$\rightarrow$	Q	[1, 2, 3]	3	[]	0	$(*)$
$\rightarrow$	Q	[1, 2, 3]	3	[4]	1	
$\rightarrow$	Q	[1, 2, 3]	3	[5, 4]	2	
$\rightarrow$	Q	[1, 2, 3]	3	[6, 5, 4]	3	
$\rightarrow$	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 -> Bag -> Bag
averageBag :: Bag -> Maybe Int
```

Our conceptual abstract model is just a list of numbers:

```
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:

```haskell
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 refinement relation in a QC-friendly way. In this case, however, it's hard to write such a function:


```
toAbstract :: Bag -> [Int]
toAbstract (B t c) = ?????
```

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


```
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_empty =
    toConc emptyBagA == emptyBag

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 been **released**.

It is due right before the **Wednesday Lecture** of **Week 5**.

Advice from Alumni

The assignments do not involve much coding, but they do involve a lot of thinking.

Start early!

Homework

1. Get started on Assignment 1.
2. Next programming exercise is out, due before 3pm Wed Week 5.
3. Last week's quiz is due this Friday. Make sure you submit your answers.
4. This week's quiz is also up, due the following Friday.