COMP3141

Software System Design and Implementation

Lecture 4: Testing Strategies, Abstract Data Types

Johannes Åman Pohjola
University of New South Wales
Term 2 2023

Announcements

Assignment 01: Available since Sunday, due July 3.

Warning

While, you have almost two weeks left to complete the assignment, a full solution will require a good bit of programming and thinking. **Start early!**

Motivation

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

But how do we come up with properties in the first place?!

Motivation

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

But how do we come up with properties in the first place?!

- **Ideal**: Come up with a test suite that guarantees full correctness. I.e. if the tests pass, our specification is satisfied.

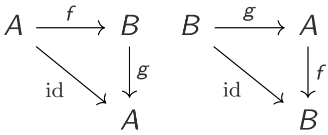
Motivation

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

But how do we come up with properties in the first place?!

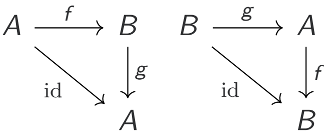
- **Ideal**: Come up with a test suite that guarantees full correctness. I.e. if the tests pass, our specification is satisfied.
- **Reality**: The more properties you test, the harder for a bug to squeeze through.

Invertible Functions I



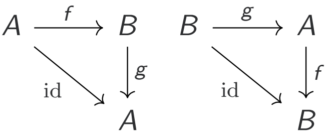
- The **inverse** of a function f is another function g that undoes the operation of f.

Invertible Functions I



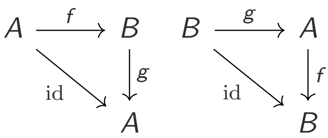
- The **inverse** of a function f is another function g that undoes the operation of f.
- For example, the function `f x = x + 5` has the inverse function `g x = x - 5`, since `g (f x) = x`.

Invertible Functions I



- The **inverse** of a function f is another function g that undoes the operation of f.
- For example, the function `f x = x + 5` has the inverse function `g x = x - 5`, since `g (f x) = x`.
- Many non-mathematical examples: saving an object to disk and (re)loading it;

Invertible Functions I



- The **inverse** of a function f is another function g that undoes the operation of f.
- For example, the function `f x = x + 5` has the inverse function `g x = x - 5`, since `g (f x) = x`.
- Many non-mathematical examples: saving an object to disk and (re)loading it; parsing a JSON string into an object then printing it back as a JSON string.

Invertible Functions II

Whenever you have an invertible function $f : T \rightarrow S$, you can implement the inverse $g : S \rightarrow T$, and write tests

```
prop_inverse_left :: T -> Bool
prop_inverse_left x = g (f x) == x
prop_inverse_right :: S -> Bool
prop_inverse_right x = f (g x) == x
```

These are often called **encode/decode** or **round-tripping** tests.

Invertible Functions III

Round-tripping is a very powerful testing technique. It can catch a large variety of (even subtle) bugs in common scenarios such as:

- Loading data from a SQL database to memory (and vice versa),

Invertible Functions III

Round-tripping is a very powerful testing technique. It can catch a large variety of (even subtle) bugs in common scenarios such as:

- Loading data from a SQL database to memory (and vice versa),
- Making API requests (e.g. transforming to and from JSON),

Invertible Functions III

Round-tripping is a very powerful testing technique. It can catch a large variety of (even subtle) bugs in common scenarios such as:

- Loading data from a SQL database to memory (and vice versa),
- Making API requests (e.g. transforming to and from JSON),
- Parsing user input (say a date from the string `"03 Jul 2023"`), etc.

Invertible Functions III

Round-tripping is a very powerful testing technique. It can catch a large variety of (even subtle) bugs in common scenarios such as:

- Loading data from a SQL database to memory (and vice versa),
- Making API requests (e.g. transforming to and from JSON),
- Parsing user input (say a date from the string `"03 Jul 2023"`), etc.

Issues:

- Many functions are not invertible: for example, `length`.
- You have to implement the inverse. You might not need the inverse for anything else. Worst of all, sometimes the inverse is much more difficult to compute than the function itself! (`show` vs `read`)

Invertible Functions IV

An ideal case (that occurs surprisingly frequently) is when a function is its own inverse. Take for example

```
reverse :: String -> String
not :: Bool -> Bool
```

Invertible Functions IV

An ideal case (that occurs surprisingly frequently) is when a function is its own inverse. Take for example

```haskell
reverse :: String -> String
not :: Bool -> Bool
```

Definition

An **involution** is a function *f* that is its own inverse, i.e., `f (f x) = x` for all `x`.

Invertible Functions IV

An ideal case (that occurs surprisingly frequently) is when a function is its own inverse. Take for example

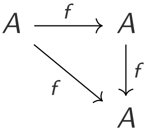
```
reverse :: String -> String
not :: Bool -> Bool
```

Definition

An **involution** is a function f that is its own inverse, i.e., `f (f x) = x` for all `x`.

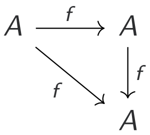
Advantage: You don't have to implement the inverse separately. Whenever you a have a function $f : T \to T$, you should at least think about whether it might be an involution.

Idempotence I



We say that an operation is **idempotent** if performing it twice yields the same result as performing it just once.

Idempotence I



We say that an operation is **idempotent** if performing it twice yields the same result as performing it just once.

Examples:

- After sorting a list (`sort`), sorting it again won't change the result.
- After removing duplicates from the list (`nub`), removing duplicates from the result will yield the same thing.
- Taking absolute values: `abs (abs (-5)) = abs 5 = 5`.
- Calling an elevator: pressing the button twice has the same result as pressing it just once.

Demo: filter twice

Idempotence II

Whenever you expect a function $f : T \to T$ to be idempotent, you can write tests

```
prop_idempotent :: T -> Bool
prop_idempotent x = f (f x) == f x
```

Idempotence II

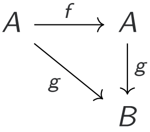
Whenever you expect a function $f : T \to T$ to be idempotent, you can write tests

```
prop_idempotent :: T -> Bool
prop_idempotent x = f (f x) == f x
```

Convenient and easy (no need to implement anything else) but has a **disadvantage**:

- Invertibility usually means high likelihood of full correctness: it only fails if you made the same mistake twice. Idempotence does not give good guarantees of full correctness by itself.

Invariants I



Whenever performing some kind of operation f does not change a given property g of the object, we say that g is an **invariant** of f.

Examples:

- The length of a list does not change after a `map` operation.
- The contents of a list do not change after a `sort` operation.

Invariants II

Whenever you expect a function $f : T \rightarrow T$ to preserve an invariant $g : T \rightarrow S$, you can write tests

```
prop_invariant :: T -> Bool
prop_invariant x = g (f x) == g x
```

Hard to Find, Easy to Test

Often it's much harder to find a solution than to test that it's actually correct. **Demo: prime factors of the integer 4294574089 are [13, 71, 923, 5041]**

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 (cf. A1)
- That a binary tree satisfies the search tree properties. (cf. practice problems)
- 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 $\texttt{wf}\ x \implies \texttt{wf}(u\ x)$

Stopping External Tampering

What's to stop a malicious or clueless programmer from going in and mucking up our data invariants?

Example

If we have an `Email` datatype, which is supposed to only contain valid emails, we can still construct an invalid email directly: `Email "INVALID"`.

We want to prevent this sort of thing from happening. For this, we need modules and abstract data types.

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: $u :: \cdots \rightarrow \mathtt{X} \rightarrow \mathtt{X}$

Abstract Data Types

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

The inverse of abstraction is called *refinement*.

Abstract data types 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

```haskell
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:

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

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

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 Email = Email String

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

Then, change the type of sendEmail:

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

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 Email = Email String

    checkEmail :: String -> Maybe Email
    checkEmail str | '@' `elem` str = Just (Email str)
                   | 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`.

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 Email = Email String

    checkEmail :: String -> Maybe Email
    checkEmail str | '@' `elem` str = Just (Email str)
                   | 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.