Advanced Programming

Testing

Fritz Henglein

Datalogisk Institut, Københavns Universitet (DIKU)

- Abstract data types and modules
- Specifications
- Testing fundamentals
- Specification-driven testing
- Property-based testing by input partitioning
- Randomized testing

Goals

Week 5 (last week):

- concurrency;
- channels and threads;
- thread-based asynchronous programming;
- client-server programming.

Week 6 (this week):

- specification-driven testing;
- randomized property based testing;
- programming with Quickcheck.

Abstract data types and modules revisited

- An *abstract data type (ADT)* is an abstract type together with operations involving the type and a precise description of what the operations do. An ADT consists of
  - an *interface* containing the *declarations* of the type and its operations (functions),
  - their *specification* (what they do, which properties they have); and
  - *examples* illustrating correct and incorrect use of the operations.
- An *implementation* of an ADT contains
  - the *definitions* (source code) of the operations; and
  - *evidence* that it satisfies the ADT's specification (test design documentation and resulting test suite).
- An ADT typically has *multiple possible implementations*.
  - In particular, the implementation types may be different from each other.
  - Example: `s -> (a, s)` and `IORef s -> IO a` for state monads.
- *Module*: Structure with zero, one or more abstract or concrete type declarations.
  - ADT: Module with one abstract type declaration.

Abstract data types

- Programming language support for ADTs and *modular abstraction*:
  - ML-family languages: Signatures (module types), structures (module implementations), functors (module-parameterized structures).
  - Haskell: Type classes, class instances, class constraints in functions.
  - Java-like languages: Interfaces and abstract classes, concrete classes, concrete classes with interface-parameterized constructor(s).

Specifications

- *Specification*: Describes what a (collection of) function(s) does, not how it is implemented (its code).
  - Litmus test: Contains *everything* a user of a module *needs to know* and *nothing else*.
  - Specifications may be
    - inconsistent (contradictory): contain a contradiction;
    - incomplete (often the case): leave room for multiple implementations with different behaviors.
- *Implementation*: Code.
- *Correctness*: Evidence (e.g. test design method and resulting *test suite*) that code may satisfy its specification.
- *Requirements*: Informal description of desired functionality of a program component (what it should do and what it should not do).
  - Requirements may be unclear and leave room for interpretation.
  - A specification *resolves and documents* this ambiguity.

Module interface specification as a contract

- Module interface specification: *Contract* between programmer *using* a module (user) and programmer *implementing* it (implementor).
  - User to implementor: "All I want to know needs to be written in the module specification. How I use your implementation is none of your business. Don't make me look at your code."
  - Implementor to user: "All you need to know is written in the module specification. How I implement it is none of your business. Don't ask to look at my code. (I may change it at any time.)"
  - Implementor and user are *roles*: They *may* be the same person.
- What if specification and implementation are *inconsistent*, as evidenced by a *test failure*?
  - Change the code to meet the specification: Requires no negotiation with user(s).
  - Change the specification to meet the code: Requires negotiation with user(s) since *changes may propagate to whatever code they have written.*
  - Change both, code and specification: Natural part of *exploratory* programming during *requirements elicitation.*
  - Common problem: Most modules are *underspecified*. User tries out code and relies on unspecified behavior of implementation.

Example: State monad specification

- How to (partially) specify an abstract data type? Typical methods:
  1. Properties its functions must satisfy.
     - Usually *universal equational properties*
  2. Reference implementation (executable specification, model implementation)
     - Observable results must be the same as those from a reference implementation
- Example: State monad
  1. Monad laws plus

```
put s >> get          = put s >> return s
put s >> put t        = put t
get   >> get          = get
```

     for all `s, t`.
  2. Coherence with purely functional implementation `FState`

Example: Reference implementation by type conversion

- Idea: Provide conversion functions from/to reference type
- Example:

```
FState s a = s -> (a, s))        -- reference type
IState s a = IORef s -> IO a    -- implementation type

to :: FState s a -> IState s a
to sa ref = do
  s <- readIORef ref
  let (a, t) = sa s
  writeIORef ref t
  return a

from :: IState s a -> (s -> IO (a, s))
from isa s = do
  ref <- newIORef s
  a <- isa ref
  t <- readIORef ref
  return (a, t)
```

Example: Reference implementation by type conversion

- Idea: Check that implementation functions `fI` simulate reference functions `fF`.
- Properties:

```
from (to sa)                     ~ sa            :: FState s a
from getI                        ~ getF          :: FState s s
from (putI s)                    ~ putF s        :: FState s ()
from (returnI a)                 ~ returnF a     :: FState s a
from (to sa `bindI` (to . f))    ~ sa `bindF` f  :: FState s b
```

for all `fstate, s, a, f`.

- Note: Left-hand side operations from new state monad implementation; right-hand side from reference implementation.
- Problems:
  - What does ~ mean?
  - How to generate functions for testing?

Testing

- *Testing* is (partial) verification by
  - executing code on inputs and checking whether the result is evidence of a specification violation;
  - *systematically* constructing the inputs to *maximize their likelihood of demonstrating a specification violation*.
- Terminology:
  - **Formal verification**: Mathematical proof that there exists *no* valid input that results in a specification violation.
  - **Testing**: The *systematic* discipline of finding “nasty” inputs that maximize the likelihood of finding specification violations.
    - Not being able to “break the code” is taken as evidence (not proof) that the code may be correct with respect to its given specification.
  - **Trying out, illustrating, exemplifying**: Applications of the code showing that it satisfies its specification for *some* inputs (if there is a specification) or that it does something useful (if there is no specification, only informal requirements).
- Don’t use the term “test” unless you have *both* specification and code and you are *systematically* looking for specification violations.
  - Use “illustrate” or “exemplify” or “try out” instead.

Testing

- *Unit testing*: Testing a module implementation against its module specification.
  - A *unit* is a basic testable component; it corresponds to the notion of *module* here.
  - Testing is of the implementation including code in standard modules, but excluding bespoke modules it depends on. (These need to be stubbed in unit testing.)
  - NB: These notes are only about unit testing.
- *Integration testing*: Testing a composition of modules that form a *system component* against its specification.
- System testing: Testing a *deployable system* against its *specification* (which is typically just a large and changing test suite).
- *Acceptance testing*: Testing a deployable system against its *business and user requirements*.

Specification-driven testing

- Recall: All forms of testing require *both* specification and code.
  - No specification, no testing.
- *Specification-driven testing*: Analyze specification and systematically
  - find *valid* data with maximal likelihood to expose a specification violation of the code;
  - determine corresponding *expected* outputs *from the specification*.
    - Don't look at the code in this process.
- Test-driven development (TDD):
  - Develop the test suite *before* you have any code.
  - Continuously run the test suite to drive and check the code as you develop it.
- Specification-driven testing is also called *specification-based testing*, *external testing*, and *black-box testing*.
  - "Black-box testing" is not used here since much of questionable value is unfortunately written on the web on that.

Properties

- A *logical statement* is a statement that is either true or false.
- A *(logical) property* of one or more functions is a true logical statement involving the function(s).
- A *universally quantified property* has the form

  $$\forall x, y, \ldots\ Q(x, y, \ldots)$$

  or

  $$\forall x, y, \ldots (P(x, y, \ldots) \Rightarrow Q(x, y, \ldots)).$$

  In words: For all (valid) $x, y, \ldots$ the property $Q(x, y, \ldots)$ is true.
  - Example:

    $$\forall x \in \texttt{Int}\ ((x \geq 0) \Rightarrow \texttt{fib}(x) = \mathit{fib}(x))$$

    where `fib` is some code and *fib* is the “true” mathematically defined Fibonacci function.
- A *(partial) formal specification* is often a conjunction of universally quantified properties.
- *Property-based testing*: Systematic design of *finite* subset of inputs aimed at falsifying a property.

Input partitioning

- *Input partitioning* is a *design method* for constructing *test data* by *inspecting the properties to be checked*.
- Given a universally quantified property, partition its valid inputs into a *finite* set of partitions, that is into pairwise disjoint subsets whose union is the entire set of valid inputs.
- From each partition,
  - choose a *typical* element and one or more *boundary elements*;
  - determine the *expected* output for each element according to the function specification—*not* the code.
- Test suite: The chosen input/expected output pairs. The boundary elements increase the likelihood of finding a specification violation.
- Test execution: Run the function on the inputs in the test suite; check that it produces the corresponding expected outputs.
- Input partitioning is also called *equivalence partitioning* or *equivalence class* partitioning.

Input partitioning: Numbers

- Generally useful partitioning of integral and floating-point numbers.
  - `int` (32-bit 2's-complement integers):
    - Partitions: Negative numbers, $\{0\}$, positive numbers.
    - Values: `-2147483648`, `-137`, `-1`, `0`, `1`, `2377`, `2147483647`.
  - `float` (64-bit IEEE 754 floating-point numbers):
    - Partitions: { Negative infinity }, negative numbers, $\{0\}$, positive numbers, { positive infinity }, { NaN } ("not-a-number").
    - Values: `System.Double.NegativeInfinity`, `2.2250738585072014e-308`, `-1.0`; `-System.Double.Epsilon`, `0.0`, `System.Double.Epsilon`, `1.0`, `47771354343.98989988`, `1.7976931348623158e+308`, `System.Double.PositiveInfinity`, `System.Double.NaN`.
  - Note: F# syntax used!
- Note that most test values are boundary values.

Intermediate summary:
Programming = Specification + coding + verification

- *Programming* consists of
  1. *Specification*: *What* is computed? Which functions, what are their *properties*?
  2. *Coding*: *How* is it computed? How are the functions implemented ("code")?
  3. *Verification*: Which explicit *evidence* is there for the program code meeting its specification (correctness)?
- Verification always requires *both*, program code *and* specification.
  - Program code may be correct with respect to one specification and incorrect with respect to another specification.
- Avoid statements such as "My code works".
  - With respect to which specification?
    - Code is trivially correct with respect to whatever it happens to do—there is nothing to test; it does what it does. But what *does* it do that a code user can rely on?
  - Does it *always* work and can you provide conclusive evidence for that, for *all* valid inputs and in *all* conceivable contexts, or just on *some data* that you happen to have tried out?

Property-based randomized testing

- Consider an implementation of a module specification.
- Let the module specification contain a *partial formal specification* in the form of a *conjunction of universal properties* that involve the module's functions:

$$\forall x, y, \ldots . P_1(x, y, \ldots) \wedge \ldots \wedge \forall x, y \ldots . P_n(x, y, \ldots)$$

  where the P_i are *decidable properties*, that is there are terminating functions f_i returning a Boolean such that $f(x, y, \ldots) == True$ if and only $P(x, y, \ldots)$ holds.
    - The module specification may contain additional requirements.
- *Randomized testing*: For each P_i
    - generate tuples $(x_0, y_0, \ldots), \ldots, (x_n, y_n, \ldots)$ *(pseudo)randomly*.
    - apply $f_i(x_j, y_j, \ldots)$ to all tuples $(x_j, y_j, \ldots)$;
    - If one application *fails* (returns *False*), this *proves* that the specification is *not satisfied* by the module implementation;
    - if all applications *succeed* (return *True*), this is *provides empirical evidence* that the specification *may be satisfied* by the module implementation.

QuickCheck: Defining properties

```
quickCheck :: Testable a => a -> IO ()

class Testable a where
  property :: a -> Property

data Property

forAll :: Testable b => Gen a -> (a -> b) -> Property
(==>)  :: Testable a => Bool -> a -> Property
(===)  :: Eq a => a -> a -> Property
(=/=)  :: Eq a => a -> a -> Property
ioProperty :: Testable a => IO a -> Property
```

- QuickCheck terminology:
  - *Property*: Probabilistic computation that succeeds (returns true) or fails (returns false) with information about the failure.
  - *Testable type*: Type with a default function for turning a value into a property.

QuickCheck: Probabilistic generation of values

```
newtype Gen a = MkGen {unGen :: QCGen -> Int -> a}

class Arbitrary a where
  arbitrary :: Gen a

choose :: (a, a) -> Gen a
oneof :: [Gen a] -> Gen a
frequency :: [(Int, Gen a)] -> Gen a
elements  :: [a] -> Gen a
sized :: (Int -> Gen a) -> Gen a
getSize :: Gen Int
resize :: Int -> Gen a -> Gen a
generate :: Gen a -> IO a
```

- QuickCheck terminology:
  - `Gen  a`: Computations generating a (pseudo)random value of type a.
  - `Arbitrary`: Class of types with a default random value generator for that type.

Input partitioning versus random testing

- Input partitioning: Good at identifying rare boundary elements/corner cases with high falsification potential.
- Random testing: Good at automatically generating large numbers of typical (random) elements.
- QuickCheck can be used to combine both: Custom generators (reflecting outcome of input partitioning)
- Explicit properties are crucial in both!

QuickCheck: More information

- Library randomized testing Haskell: `https://hackage-content.haskell.org/package/QuickCheck-2.16.0.0/docs/Test-QuickCheck.html`.
- QuickCheck manual: `https://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html`

Testing challenges

- Testing imperative code (including object-oriented code):
  - Test design: Partition *state space*. Design *commands* for reaching boundary and typical states. Use *queries* for observing the reached state. Do the observed values have the expected *properties*?
  - Command-Query Separation (CQS) design pattern useful for testability.
- Testing concurrent code:
  - Properties need to be checked *for all possible schedulers* (all interleavings of all concurrently executing threads), not only for the particular scheduler one happens use.
  - Very difficult *and* expensive!
  - Advanced techniques (partially symbolic execution, etc) are beyond the scope of AP.
- Confusing test design with test suite
  - Specification-driven testing is the systematic and documented *process* of analyzing a specification to arrive at a *test suite with high falsification power*.
  - The test suite is the *output of the process*. By itself it says nothing about its falsification power. Only the *test design report* documenting the *systematic process* of construction the test suite does.

Summary

- Abstract data types and modules
- Specifications
- Testing fundamentals
- Specification-driven testing
- Property-based testing by input partitioning
- Randomized testing