Today's Topic

Testing of programs

- Specification-based

- Tool-supported

- Automatically

Questions

How can we gain (sufficient) confidence that...

- our programs are *sound*,

- other people's programs are *sound*?

Answers

- Verification

  - Formal soundness proof (soundness of the specification, soundness of the implementation)

  - High confidence, high effort

- Testing

  - Variants: *systematically* vs. *ad hoc*

  - Controllable effort, undefined quality statement

    Remember:
    "Testing can only show the presence of faults, not their absence" (Dijkstra)

On the other hand...

Observation

Testing is often...

- amazingly successful in disclosing faults

Requirements

Reporting on...

- What has been tested?

- How thoroughly, how comprehensively has been tested?

- How was *success* defined?

Additionally desirable...

- Reproducibility of tests

- Repeated testing after program modifications

Preconditions

Indispensable...

- Specification of the meaning of the program
  - *Informally* (commentary in the program, in a separate documentation)
    ⤳ ...often ambiguous, open to interpretation
  - *Formally*
    ⤳ ...precise semantics, unique

In the following

Specification-based, tool-supported testing in Haskell

- QuickCheck
  - defines a *formal specification language*
    ⤳ ...allows property definition inside the (Haskell) source code
  - defines a *test-case generator language*
    ⤳ ...allows a simple and concise description of a large number of test cases
  - allows automatic testing of all properties specified in a module, including failure reports

Remark

Specification- and test-case generator language are...

- Examples of so-called *domain-specific embedded languages*
  ⤳ ...special strength of functional programming

- implemented as a *combinator library* in Haskell
  ⤳ ...allows to make use of the full expressiveness of Haskell when defining properties and test-case generators

- Part of the standard Haskell-distribution (see module `QuickCheck`)
  ⤳ ...ensures simple and direct usability

Reference

The following presentation is based on...

- Koen Claessen, John Hughes. *Specification-based Testing with QuickCheck*. In Jeremy Gibbons, Oege de Moor (Hrsg.), *The Fun of Programming*. Palgrave MacMillan, 2003.

Other relevant references...

- Koen Claessen, John Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In Proceedings of the ACM SIGPLAN 2000 International Conference on Functional Programming (ICFP 2000), 268 - 279, 2000.

- Koen Claessen, John Hughes. *Testing Monadic Code with QuickCheck*. In Proceedings ACM SIGPLAN 2002 Haskell Workshop, 2002.

Property Definition with QuickCheck 1(2)

First examples...

Inside the program:

```
prop_PlusAssociative :: Integer -> Integer -> Integer -> Bool
prop_PlusAssociative x y z = (x+y)+z == x+(y+z)
```

In Hugs:

```
Main>quickCheck prop_PlusAssociative
OK, passed 100 tests
```

Remark:

- Type specification for prop_PlusAssociative is required because of the overloading of + (otherwise error message)

- Type specification allows a type-specific generation of test-cases

- Property definition in the most simple cases: Predicates, i.e. functions with range type Boolean

Property Definition with QuickCheck 2(2)

In the program:

```
prop_PlusAssociative :: Float -> Float -> Float -> Bool
prop_PlusAssociative x y z = (x+y)+z == x+(y+z)
```

In Hugs (possibly, think of rounding errors):

```
Main>quickCheck prop_PlusAssociative
Falsifiable, after 13 tests:
1.0
-5.16667
-3.71429
```

Remark:

Error report contains:

- Number of tests successfully passed

- Counter example

More Complex Property Definitions with QuickCheck 1(3)

Consider as the property to be checked:

...to insert in a sorted list

(we suppose that a function insert is given)

```
prop_InsertOrdered x xs = ordered (insert x xs)
```

...too strong/naive. Note, xs is not supposed to be sorted.

More Complex Property Definitions with QuickCheck 2(3)

Remedy:

```
prop_InsertOrdered :: Integer -> [Integer] -> Property
prop_InsertOrdered x xs = ordered xs ==> ordered (insert x xs)
```

Remark:

- ordered xs ==>: Adding a *precondition*

  ⤳ ...Test cases, which do not match the precondition, are dropped

- ==>: ...is not a Boolean operator; it is an operator, which affects the selection of test cases

  ⤳ ...Property definitions, which rely on such operators, always have the type Property

More Complex Property Definitions with QuickCheck 3(3)

Possible too:

- Direct quantification over sorted lists

```
prop_InsertOrdered :: Integer -> Property
prop_InsertOrdered x =
          forAll orderedLists $ \xs -> ordered (insert x xs)
```

Going beyond:

- Refinement of the specification such that the result list concides with the argument list (except of the inserted element)

The Operator $

See Standard Prelude:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Remark

- The operator $ is the "apply" operator

- It is useful to avoid the usage of parentheses:

  Example: f (g x) can be written as f $ g x

An Extended Example

...abstract data type for (first-in-first-out) *queues*.

Simple (yet inefficient) implementation:

```
type Queue a = [a]
empty        = []
add x q      = q ++ [x]     -- inefficient!
isEmpty q    = null q
front (x:q)  = x
remove (x:q) = q
```

A More Efficient Implementation

Basic Idea

- Splitting the list in two portions (list front and back)

- List back in reverse order

- This ensures: Efficient access to list front and list back

```
type QueueI a   = ([a],[a])
emptyI          = ([],[])
addI x (f,b)    = (f,x:b)
isEmptyI (f,b)  = null f
frontI (x:f,b)  = x
removeI (x:f,b) = flipQ (f,b)
  where
    flipQ ([],b) = (reverse b, [])
    flipQ q      = q
```

List Representations and Represented Abstract Lists: The Relation

...by means of a *retrieve* function:

```
retrieve :: QueueI Integer -> [Integer]
retrieve (f,b) = f ++ reverse b
```

- ...transforms the (usually many) representations of an abstract list as values of QueueI into the underlying abstract list as values of Queue

- ...the understanding of QueueI and Queue as lists on integers allows us to drop type specifications in the following

Soundness Properties for Functions on QueueI

...by means of retrieve it is checkable, if

- the results of applying the efficient functions on QueueI coincide with those of the abstract functions on Queue

Soundness Properties: First Try

Apparently, the following properties are expected to hold:

```
prop_empty     = retrieve emptyI == empty
prop_add x q   = retrieve (addI x q) == add x (retrieve q)
prop_isEmpty q = isEmptyI q == isEmpty (retrieve q)
prop_front q   = frontI q == front (retrieve q)
prop_remove q  = retrieve (removeI q) == remove (retrieve q)
```

Soundness Properties: Second Try

Check using `QuickCheck` yields:

```
Main>quickCheck prop_isEmpty
Falsifiable, after 4 tests:
([],[-1])
```

Problem:

- The specification of `isEmpty` (as the specifications of `frontI` and `removeI`) implicitly assume that the following *invariant* holds:

  – The front of the list is only empty, if the back of the list is empty, too

Soundness Properties: Corrected Version

We define the following invariant:

```
invariant :: QueueI Integer -> Bool
invariant (f,b) = not (null f) || null b
```

...and add them to all property definitions:

```
prop_empty    = retrieve emptyI == empty
prop_add x q  = invariant q ==>
                       retrieve (addI x q) == add x (retrieve q)
prop_isEmpty q = invariant q ==>
                       isEmptyI q == isEmpty (retrieve q)
prop_front q  = invariant q ==>
                       frontI q == front (retrieve q)
prop_remove q = invariant q ==>
                       retrieve (removeI q) == remove (retrieve q)
```

Soundness Properties: Corrected Version

Check using `QuickCheck` yields:

```
Main>quickCheck prop_isEmpty
OK, passed 100 tests
```

But:

```
Main>quickCheck prop_front
Program error: front ([],[])
```

Problem:

- `frontI` (just as `removeI`) may only be applied to non-empty lists; so far, this is not taken into account.

  – Remedy: Add `not (isEmptyI q)` to the preconditions of the relevant properties

Soundness Properties: Corrected Version

We obtain:

```
prop_empty    = retrieve emptyI == empty
prop_add x q  = invariant q ==>
                       retrieve (addI x q) == add x (retrieve q)
prop_isEmpty q = invariant q ==>
                       isEmptyI q == isEmpty (retrieve q)
prop_front q  = invariant q && not (isEmptyI q) ==>
                       frontI q == front (retrieve q)
prop_remove q = invariant q && not (isEmptyI q) ==>
                       retrieve (removeI q) == remove (retrieve q)
```

Remark:

- Now, all properties pass the test successfully!

Soundness Properties: Still to Resolve

Tested so far:

- Operations on queues behave correctly on representations of queues which satisfy the invariant

    ```
    invariant (f,b) = not (null f) || null b
    ```

We still need to check:

- Operations producing queues do only produce queues, which satisfy this invariant.

Soundness Properties: Still to Resolve

The formulation of appropriate properties for functions producing queues:

```
prop_inv_empty    = invariant emptyI
prop_inv_add x q  = invariant q ==> invariant (addI x q)
prop_inv_remove q = invariant q && not (isEmptyI q) ==>
                              invariant (removeI q)
```

Soundness Properties: Still to Resolve

Check by means of `QuickCheck` yields:

```
Main>quickCheck prop_inv_add
Falsifiable, after 0 tests:
0
([],[])
```

Problem:

- The invariant must hold
  - …not only after applying `removeI`,
  - …but also after applying `addI` to the empty list

Soundness Properties: Still to Resolve

To this end:

- Adjustment of function `addI` as follows:

  ```
  addI x (f,b) = flipQ(f,x:b)  -- instead of: addI x (f,b) = (f,x:b)
  ```

Remark:

- Now, all properties pass the test successfully!

Observation

In the course of developing this example it turned out:

- Testing disclosed (only) one bug in the implementation (this was in function `addI`)

- *But*: Several missing preconditions and a missing invariant in the original definitions of properties were found and added

This is both typical. It is valuable, too:

- The additional conditions and invariants are now explicitly given in the program text

- They add to understanding the program and are valuable as a documentation, both for the program developer and for future users (think of program maintaining!)

Algebraic Specifications

...(often a desired) alternative to the abstract model

An algebraic specification...

- specifies equational constraints the operations ought to comply with

Algebraic Specifications

For the example of queues, for instance, as follows:

```
prop_isEmpty q     = invariant q ==> isEmptyI q == (q == emptyI)
prop_front_empty x = frontI (addI x emptyI) == x
prop_front_add x q = invariant q && not (isEmptyI q) ==>
                          frontI (addI x q) == frontI q
prop_remove_empty x = removeI (addI x emptyI) == emptyI
prop_remove_add x q = invariant q && not (isEmptyI q) ==>
                          removeI (addI x q) == addI x (removeI q)
```

Algebraic Specifications

Check using `QuickCheck` yields:

```
Main>quickCheck prop_remove_add
Falsifiable, after 1 tests:
0
([1],[0])
```

Problem:

- Left hand side yields: ([0,0],[])

- Right hand side yields: ([0],[0])

- Equivalent but not equal!

Algebraic Specifications

Solution:

- Consider instead of "equal" now "equivalent"

```
q 'equiv' q' = invariant q && invariant q' &&
               retrieve q == retrieve q'
```

Then, replacement of

```
prop_remove_add x q = invariant q && not (isEmptyI q) ==>
                        removeI (addI x q) == addI x (removeI q)
```

by

```
prop_remove_add x q = invariant q && not (isEmptyI q) ==>
                        removeI (addI x q) 'equiv' addI x (removeI q)
```

yields the desired result.

Algebraic Specifications

Similar to the previous setting, we have to check:

- All operations producing queues yield results, which are equivalent, if the arguments are.

Considering the operation `addI`, for instance, we have to check:

```
prop_add_equiv q q' x = q 'equiv' q' ==> addI x q 'equiv' addI x q'
```

Algebraic Specifications

Mathematically sound, but inappropriate for fully automatic testing, since:

```
Main>quickCheck prop_add_equiv
Arguments exhausted after 58 tests.
```

Problem and background:

- `QuickCheck` generates lists `q` und `q'` randomly

- Most of the pairs of lists will not be equivalent, and hence be discarded for the actual test

- `QuickCheck` generates a maximum number of candidate arguments only, and then stops, possibly before the number of 100 test cases is met

Enhancing Usability

...of QuickCheck by providing support for

- Quantification over subsets
  - ...by means of *filters*
  - ...by means of *generators*   (type-based, weighted, size controlled,...)

- ...

- test case monitoring

In the following:

$\rightsquigarrow$ ...illustrating this support in terms of examples!

Quantifications over Subsets

For `QuickCheck` holds:

- By default, parameters are quantified over values of the appropriate type

Often, however, it is desired:

- A quantification over subsets of these values

Quantifications over Subsets

`QuickCheck` offers several alternatives for this purpose:

- Representation of subsets in terms of *Boolean functions*, which act as a filter for test cases
  - Adequate, if many elements of the underlying set are members of the relevant subset, too.
  - Inadequate, if only a few elements of the underlying set are members of the relevant subset

- Representation of subsets in terms of *generators*
  - A generator of type `Gen a` yields a random sequence of values of type `a`
  - The property `forall set p` successively checks `p` on randomly generated elements of `set`

Support by QuickCheck

For the effective usage of generators `QuickCheck` supports:

- different variants for the specification of relations such as `equiv`
  - As a Boolean function
    * simple to check equivalency of two values, difficult to generate values which are equivalent
  - As a function from a set of values to another set of equivalent values (generator!)
    * simple to generate equivalent values, difficult to check equivalency of two values

Generators

The generator variant for `equiv`:

```
equivQ :: QueueI a -> Gen(QueueI a)
equivQ q = do k <- choose (0,0 'max' (n-1))
              return (take (n-k) els, reverse (drop (n-k) els))
  where
    els = retrieve q
    n   = length els
```

Remark:

- Definition of `choose` will be given later

Generators

This allows us to check:

- The equivalency of the two variants of `equiv`

To this end check:

```
prop_EquivQ q = invariant q ==>
    forAll (equivQ q) $ \q' -> q 'equiv' q'
```

Remark:

- $ means function application. Using $ allows the omission of parentheses, see the λ expression in the example

- The property which is dual to `prop_EquivQ` cannot be checked by testing

Generators

This allows:

- Reformulating the property that `addI` maps equivalent queues to equivalent queues

```
prop_add_equiv q x = invariant q ==>
    forAll (equivQ q) $ \q' -> addI x q 'equiv' addI x q'
```

Remark:

- Other properties analogously

Defining Generators

...simplified because of the monadic type of `Gen`.

It holds:

- `return a` yields (generates) always `a` and represents the singleton set {a}

- `do {x <- s; e}` can be considered the (generated) set {e | x ∈ s}

Defining Generators

The fundamental function to make a choice:

```
choose :: Random a => (a,a) -> Gen a
```

Remark:

- The function `choose` generates "randomly" an element of the specified domain

- `choose (1,n)` represents the set {1...n}

Applying choose

Using `choose` we can define `equivQ` (as seen above):

```
equivQ :: QueueI a -> Gen(QueueI a)
equivQ q = do k <- choose (0,0 'max' (n-1))
              return (take (n-k) els, reverse (drop (n-k) els))
  where
    els = retrieve q
    n   = length els
```

- Generates a random queue containing the same elements as q

- The number of elements in the remainder of the list will be chosen such that it is properly smaller than the total number of elements of the list (supposed the total number is different from 0)

Type-based Generators

...by means of the overloaded generator `arbitrary`

Example:

```
prop_max_le x y = x <= x 'max' y
```

is equivalent to

```
prop_max_le = forAll arbitrary $ \x -> forAll arbitrary $ \y ->
    x <= x 'max' y
```

Type-based Generators

Another example:

The set $\{y \mid y \geq x \}$ can be generated by

```
atLeast x = do diff <- arbitrary
               return (x + abs diff)
```

because of the equality (which holds for numerical types)

$$\{y \mid y \geq x\} = \{x + abs\ d \mid d \in \mathbb{Z}\}$$

(Remark: Similar definitions for other types are possible.)

Selection

...by means of a generator `oneof`, which can be thought of as set union.

Example: Constructing a sorted list

```
orderedLists = do x <- arbitrary
                  listsFrom x
    where
      listsFrom x = oneof [return [], do y <- atLeast x
                                         liftM (x:) (listsFrom y)]
```

Underlying intuition:

- A sorted list is either empty or the addition of a new head element to a sorted list of larger elements

Weighted Selection

- The `oneof` combinator picks with equal probability one of the alternatives

- This often has an unduly impact on the test case generation (the empty set will be selected too often)

- *Remedy*: A weight function `frequency`, which assigns different weights to the alternatives

```
frequency :: [(Int,Gen a)] -> Gen a
```

Application:
```
listsFrom x = frequency [(1,return []),
                         (4,do y <- atLeast x
                               liftM (x:) (listsFrom y)) ]
```

- A `QuickCheck` generator corresponds to a probability distribution over a set, not the set itself

- The impact of the above assignment of weights is that on average the length of generated lists is 4

The Class Arbitrary

If non-standard generators such as `orderedLists` are used frequently, it is advisable to make this type an instance of `Arbitrary`:

```
newtype OrderedList a = OL [a]


instance (Num a, Arbitrary a) => Arbitrary(OrderedList a) where
   arbitrary = liftM OL orderedLists
```

The re-definition of `insert`

```
insert :: Ord a => a -> OrderedList a -> OrderedList a
```

allows then to generate arguments automatically.

Controlling the Size of Generated Test Data

- Often wise for type-based test data generation

- Explicitly supported by `QuickCheck`

...by means of:
```
sized :: (Int -> Gen a) -> Gen a  -- For defining size aware gen.

sized $ \n -> do len <- choose (0,n) -- Application of sized
                 vector len          -- in the Def. of the default
                                     -- list generator

vector n = sequence [arbitrary | i <- [1..n]] -- generates random list
                                               -- of length n

resize :: Int -> Gen a -> Gen a -- for controlling the size of
                                -- generated values

sized $ \n -> resize (round (sqrt (fromInt n))) arbitrary
       -- Application of resize
```

Generators for user-defined Types

Test data generators for...

- *predefined ("built-in") types*
    - are provided by `QuickCheck`
    - for *user-defined types*, this is not possible

- *user-defined types*
    - have to be provided by the user in terms of defining a suitable instance of class `Arbitrary`
    - require usually, especially in case of recursive types, to control the size of generated test cases

Example: Binary Trees

Consider type `Tree`:

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

The following definition of the test-case generator is apparent:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary =
     frequency [(1,return Leaf),
                (3,liftM3 Branch arbitrary arbitrary arbitrary)]
```

Example: Binary Trees

Note:

- The assignment of weights (1 vs. 3) has been done in order to avoid the generation of all too many trivial trees of size 1

- *Problem*: The likelihood that a generator comes up with a *finite* tree, is only one third

  ⤳ …note that termination is possible only, if all sub-trees generated are finite. With increasing breadth of the trees, the requirement of always selecting the "terminating" branch has to satisfied at ever more places simultaneously

Example: Binary Trees

Remedy:

- Usage of the parameter `size` in order to ensure
  - termination and
  - "reasonable" size

  of the trees generated

Example: Binary Trees

Implementation:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbTree

arbTree 0 = return Leaf
arbTree n | n>0 =
   frequency [(1,return Leaf),
              (3,liftM3 Branch shrub arbitrary shrub)]
     where
       shrub = arbTree (n 'div' 2)
```

Example: Binary Trees

Remark:

- shrub is a *generator* for "small" trees

- shrub is not bounded to a special tree; the two occurrences of shrub will usually generate different trees

- Since the size limit for subtrees is halved, the total size is bounded by the parameter size

- Defining generators for recursive types must usually be handled differently as in this example

Test-Data Monitoring / Test Coverage

In practice, it is meaningful...

- to monitor the test cases generated

- in order to obtain a hint on the quality and the coverage of test cases of a QuickCheck run

For this purpose QuickCheck provides...

- an array of monitoring possibilities

Test-Data Monitoring / Test Coverage

Why is test-data monitoring meaningful?

Reconsider the example of inserting into a sorted list:

```
prop_InsertOrdered :: Integer -> [Integer] -> Property
prop_InsertOrdered x xs = ordered xs ==> ordered (insert x xs)
```

Test-Data Monitoring / Test Coverage

QuickCheck performs the check of prop_InsertOrdered such that...

- lists are generated randomly

- each generated list will be checked, if it is sorted (used test case) or not (discarded test case)

Obviously, it holds...

- the likelihood that a randomly generated list is sorted is the higher the shorter the list is

This introduces the danger that...

- the property prop_InsertOrdered is mostly tested with lists of length one or two

- even a successful test is not meaningful

Test-Data Monitoring / Test Coverage

For monitoring `QuickCheck` provides a...

- combinator `trivial`, where the meaning of "trivial" is user-definable

Example:

```
prop_InsertOrdered :: Integer -> [Integer] -> Property
prop_InsertOrdered x xs = ordered xs ==>
      trivial (length xs <= 2) $ ordered (insert x xs)
```

with

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests (91% trivial)
```

Test-Data Monitoring / Test Coverage

Observation:

- 91% are too many trivial test cases in order to ensure that the total test is meaningful

- The operator `==>` should be used with care in test-case generators

Remedy:

- User-defined generators
  - ⤳ ...as in the example of `prop_InsertOrdered` on slide 14

Test-Data Monitoring / Test Coverage

The combinator `trivial` is...

- instance of a more general combinator `classify`

```
trivial p = classify p "trivial"
```

Test-Data Monitoring / Test Coverage

Multiple application of `classify` allows a more refined test-case monitoring:

```
prop_InsertOrdered x xs = ordered xs =>
   classify (null xs) "empty lists" $
       classify (length xs == 1) "unit lists" $
           ordered (insert x xs)
```

This yields:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
42% unit lists.
40% empty lists.
```

Test-Data Monitoring / Test Coverage

Going beyond, the combinator `collect` allows to keep track on all test cases:

```
prop_InsertOrdered x xs = ordered xs =>
  collect (length xs) $ ordered (insert x xs)
```

This yields:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
46% 0.
34% 1.
15% 2.
5% 3.
```

Conclusions 1(3)

Generally, it holds:

- Formalizing specifications is meaningful (even without a subsequent formal proof of soundness)

Experience shows:

- Specifications provided are often (initially) faulty themselves

Conclusions 2(3)

`QuickCheck` is an effective tool...

- to disclose bugs in
  - programs and
  - specifications

  with little effort.

- to reduce
  - test costs
  - while simultaneously testing more thoroughly

Conclusions 3(3)

Investigations of Richard Hamlet in...

Richard Hamlet. *Random Testing*. In J. Marciniak (Ed.), Encyclopedia of Software Engineering, Wiley, 970-978, 1994

suggest that

- a high number of test cases yields meaningful results even in the case of *random testing*

In principle, it holds:

- The generation of random test cases is "cheap"

Hence, there are many reasons advising...

- the routine use of a tool like `QuickCheck`!

Next lecture...

We will continue in April after the Easter holiday on...

- Thu, April 24, 2008, lecture time: 4.15 p.m. to 5.45 p.m., lecture room on the ground floor of the building Argentinierstr. 8

First and second assignment...

- ...please check out the homepage of the course for details (the due dates will be on Mon, April 28, 2008).

In case of problems with the generation of accounts on machine g0, please contact "Studienassistent" Markus Triska at

- `studasscomplang.tuwien.ac.at`