Programming with monads, laziness, randomness, and QuickCheck

COMP 150 - Applied Functional Programming

October 15, 2012

Higher-order functions and monads

(1) In Sections 3.1 and 3.2 of the QuickCheck paper, you’ll see examples which use polymorphic functions `liftM` and `liftM2`. If we had had more time on “monad day”, these were functions that you might have invented.

(a) At what types are these functions used in the examples?
(b) What are the most general types you can imagine that make sense for `liftM` and `liftM2`?
(c) In the “coin-flip” problem, you are given a monadic computation that returns a floating-point number between 0 and 1, and you need to convert it to a random coin flip. Does either of these functions help with the coin-flip problem?

(2) What higher-order functions can you think of that relate functions, the IO monad, and lists?

Testing predicates

(3) You have written a `Predicate` type class which describes predicates of arbitrary arity and which defines overloaded operations to conjoin, disjoin, and complement predicates, as well as the always-true predicates and the always-false predicates.

Define an `implies` function that is overloaded on predicates of arbitrary arity. Do not extend `Predicate`.

(4) Given your code for operating on predicates of arbitrary arity, try using QuickCheck to test these properties:

```haskell
even === complement odd
(<) === complement (>)
forall p . (even 'conjoin' p) 'implies' p
forall p . (even 'disjoin' p) 'implies' p
```

To universally quantify over a predicate, you will want to use QuickCheck’s special function shrinker, e.g.,

```haskell
property3 (Fun _ p) =
  (even 'conjoin' p) 'implies' p
```

as opposed to the simpler

```haskell
property3_not_useful p =
  (even 'conjoin' p) 'implies' p
```

The version without `Fun` won’t even type-check, because there’s no `Show` instance for a function.

(5) Extend QuickCheck extensional equality to functions of arbitrary arity. In particular, if you have two functions `f` and `g` which both have type `a → b → ⋯ → n → r`, and if argument types `a, b, ⋯, n` are in class `Arbitrary`, and if the result type `r` is in class `Eq` then it should be possible to define an operator `====` such that the expression `f === g` is well typed.

(a) What should the type of `f === g` be?
(b) Define `====` by introducing a new type class and suitable instance declarations.

(6) Use your new extensional equality to test the following properties:

```haskell
even === complement odd
(<) === complement (>)
forall p . (complement p 'conjoin' p) 'implies' falsehood
forall p . (complement p 'disjoin' p) 'implies' truth
forall p . (complement p 'disjoin' truth) 'implies' p
```

The last three properties are polymorphic and overloaded. Please test each property at multiple arities. You will need to use a type signature; if you like, you can follow the example at the end of Section 2.1 of the QuickCheck paper

DVD Packing

(7) What properties can you think of that the DVD-packing algorithm should satisfy?

(a) What are the properties?
(b) Which of the properties can be coded using QuickCheck?
(c) Code them.
(8) To test DVD packing with QuickCheck, what test-case generator would you want to use? (Hint: the example I gave you is not a very challenging test.)

**Bubble search**

(9) Bubble search is interesting because it is a general method for improving any greedy algorithm.

(a) Write a polymorphic type for bubble search that makes the generality of the method manifest.

(b) What, if any, is the role of laziness?

(c) What deterministic properties of bubble search can you write down and test?

(d) What probabilistic properties of bubble search can you write down?

**DVD Packing results**

(10) What results do you get using bubble search for DVD packing? Please analyze your results in depth and explain everything that is going on.

(11) Suppose you are given the goal of packing DVDs such that every DVD except the last is at least 99.5% full.

(a) Is this goal achievable?

(b) How would you tackle the problem?