Formal Methods for Software Development

Till Mossakowski, Lutz Schröder

03./08.11.2004

Monadic QuickCheck

- extension of QuickCheck for monadic (= imperative) programs
- specifications are equations between monadic actions
- equations are interpreted observationally, i.e. $p_1 = p_2$ if p_1 and p_2 admit the same observations in each context

The State Monad

```
newtype State s a =
   State { runState :: s -> (a, s) }

instance Monad (State s) where
        return a = State $ \s -> (a, s)
        m >>= k  = State $ \s -> let
                (a, s') = runState m s
                in runState (k a) s'
```

The Queue Example

```
type Queue = Int
type QueueState a = (Int->[a],Int)
type QS a = State (QueueState a)

initState :: QueueState a

empty   :: QS a Queue
add  :: Queue -> a -> QS a ()
remove  :: Queue -> QS a ()
front   :: Queue -> QS a (Maybe a)
```

Algebraic specification of queues

```
  q <- empty; x <- front q
= q <- empty; x <- return Nothing

  q <- empty; add q m; x <- front q
= q <- empty; add q m; x <- return (Just m)

  add q m; add q n; x <- front q
= add q m; x <- front q; add q n

  q <- empty; add q m; remove q
= q <- empty

  add q m; add q n; remove q
= add q m; remove q; add q n
```

Observational Equivalence

- A context C is a “program with a hole”. We write the result of “filling the hole” with a term e as $C[e]$.
- We write $p \Downarrow o$ when program p computes an observable result o.
- Two terms e and e' are observationally equivalent if, for every context C,

  $C[e] \Downarrow o$ if and only if $C[e'] \Downarrow o$

How to Implement Observational Equivalence with QuickCheck?

- Problem: how to enumerate all contexts?
- Solution: only work with sequences of basic operations of the queue monad
- Represent these as sequences of actions
- Be careful that only well-formed sequences are taken into account (e.g. those not removing something from the empty queue)

Queue Actions as a Datatype

```
data Action a = Add a | Remove |
              Front | Return (Maybe a)
   deriving (Eq, Show)
```

Executing Actions

```
perform :: Queue -> [Action a]
           -> QS a [Maybe a]
perform q [] = return []
perform q (a : as) =
 case a of
  Add n -> add q n >> perform q as
  Remove -> remove q >> perform q as
  Front -> liftM2 (:) (front q)
                     (perform q as)
  Return x -> liftM (x :) (perform q as)
```

Well-formed Action Sequences

```
actions :: (Arbitrary a, Num b)
    => b -> Gen [Action a]
actions n =
  oneof
    ([return [],
      liftM2 (:) (liftM Add arbitrary)
                 (actions (n + 1)),
      liftM (Front :) (actions n)] ++
      if n == 0 then [] else
        [liftM (Remove :) (actions (n - 1))])
```

Counting the Number of Queue Elements

```
delta :: [Action a] -> Int
delta = sum . map deltaAux
  where deltaAux a = case a of
                Add _ -> 1
                Remove -> -1
                _ -> 0
```

Observational Equivalence

```
(===) :: (Eq a,Show a,Arbitrary a) => [Action
c === c' =
  forAll (actions 0) $ \ pref ->
  forAll (actions (delta (pref ++ c)))
      $ \ suff ->
  let observe x =
       fst (runState ( do
                    q <- empty
                    perform q (pref++x++suff)
                 initState )
    in observe c == observe c'
```

Example Properties

```
prop_FrontAdd m n =
  [Add m, Add n, Front] ===
  [Add m, Front, Add n]

prop_AddRemove m n =
  [Add m, Add n, Remove] ===
  [Add m, Remove, Add n]
```

Observational Equivalence, Modified

```
c ==ˆ c′ =
  forAll (actions (delta c)) $ \ suff ->
     let observe x =
           fst (runState ( do
                    q <- empty
                    perform q (x ++ suff ))
                 initState)
       in observe c == observe c′
```

Remaining Properties

```
prop_FrontEmpty =
 [Front] ==^ [Return Nothing]

prop_FrontAddEmpty m =
 [Add m, Front] ==^ [Add m, Return (Just m)]

prop_AddRemoveEmpty m =
 [Add m, Remove] ==^ []
```

Model-based Testing

- The imperative queue is tested against an abstract (functional) model of queues
- using an abstraction function

```
emptyS = []
addS a q = ((), q ++ [a])
removeS (_ : q) = ((), q)
frontS [] = (Nothing, [])
frontS (a : q) = (Just a, a : q)
```

Hoare triples

$$\{p\}\ x \leftarrow e\ \{q\ x\}$$

read: under precondition p, program e delivers a value x and ends in a state such that $q\ x$ holds.

In QuickCheck:

```
pre p
x <- run e
assert q
```

Alternative Definition of Observat. Equality

```
(====) :: (Eq a,Show a,Arbitrary a) =>
      [Action a] -> [Action a] -> Property
c ==== c' = imperative(
  forAllM (actions 0) $ \ pref ->
  forAllM (actions (delta (pref ++ c)))
      $ \ suff ->
  do q <- run empty
     obs1 <- run (perform q (pref++c++suff))
     obs2 <- run (perform q (pref++c++suff))
     assert (obs1==obs2)
  )
```

Dynamic Logic

- $[x \leftarrow p]\varphi$ (read: after all possible runs of p, φ holds)
- $< x \leftarrow p > \varphi$ (read: after some possible run of p, φ holds)
- This makes a difference for non-deterministic programs (e.g. ParSec combinator parser)
- danymic logic has highest degree of flexibility, but not testing support (yet)
- theorem proving support under development

Summary

- QuickCheck allows for testing monadic code
- Pre- and postconditions can be formulated, as in Hoare triples
- Sequences of monadic actions and their well-formedness have to be implemented in an ad-hoc way
- Dynamic logic overcomes this problem, but only provides theorem proving, no testing