COMP3141

Software System Design and Implementation

Data Invariants, Abstraction and Refinement Practice

Christine Rizkallah
CSE, UNSW (and Data61)
Term 2 2019

Recall: Structure, Data Invariants and Refinement

- Structure: a module for a data type `X` will typically provide operations to construct, query, and update `X`.

Recall: Structure, Data Invariants and Refinement

- Structure: a module for a data type `X` will typically provide operations to construct, query, and update `X`.
- Data Invariants: are properties that pertain to a particular data type that the data type's operations should maintain.

Recall: Structure, Data Invariants and Refinement

- Structure: a module for a data type `X` will typically provide operations to construct, query, and update `X`.
- Data Invariants: are properties that pertain to a particular data type that the data type's operations should maintain.
- Data Refinement: for every behaviour exhibited by the concrete implementation, there is a similar behaviour in the abstract model.

Recall: Structure, Data Invariants and Refinement

- Structure: a module for a data type `X` will typically provide operations to construct, query, and update `X`.
- Data Invariants: are properties that pertain to a particular data type that the data type's operations should maintain.
- Data Refinement: for every behaviour exhibited by the concrete implementation, there is a similar behaviour in the abstract model.

Refinement and Specifications

In general, all functional correctness specifications can be expressed as:

1. all data invariants are maintained, and

Recall: Structure, Data Invariants and Refinement

- Structure: a module for a data type `X` will typically provide operations to construct, query, and update `X`.
- Data Invariants: are properties that pertain to a particular data type that the data type's operations should maintain.
- Data Refinement: for every behaviour exhibited by the concrete implementation, there is a similar behaviour in the abstract model.

Refinement and Specifications

In general, all functional correctness specifications can be expressed as:

1. all data invariants are maintained, and
2. the implementation is a refinement of an abstract correctness model.

Editor Example

Consider this ADT interface for a text editor:

```
data Editor
einit :: String -> Editor
stringOf :: Editor -> String
moveLeft :: Editor -> Editor
moveRight :: Editor -> Editor
insertChar :: Char -> Editor -> Editor
deleteChar :: Editor -> Editor
```

Data Invariant Properties

```
prop_einit_ok           s = wellformed (einitA s)
prop_moveLeft_ok      a = wellformed (moveLeftA a)
prop_moveRight_ok     a = wellformed (moveRightA a)
prop_moveInsert_ok x a = wellformed (insertCharA x a)
prop_moveDelete_ok    a = wellformed (deleteCharA a)
```

Editor Example: Abstract Model

Our conceptual abstract model is a string and a cursor position:

Editor Example: Abstract Model

Our conceptual abstract model is a string and a cursor position:

```
einitA s = A s 0
stringOfA (A s _) = s
moveLeftA  (A t c) = A t (max 0 (c-1))
moveRightA (A t c) = A t (min (length t) (c+1))
insertCharA x (A t c) = let (t1, t2) = splitAt c t
                        in A (t1 ++ [x] ++ t2) (c+1)
deleteCharA (A t c) = let (t1, t2) = splitAt c t
                      in A (t1 ++ drop 1 t2) c
```

But do we need to keep track of all that information in our implementation?

Editor Example: Abstract Model

Our conceptual abstract model is a string and a cursor position:

```
einitA s = A s 0
stringOfA (A s _) = s
moveLeftA  (A t c) = A t (max 0 (c-1))
moveRightA (A t c) = A t (min (length t) (c+1))
insertCharA x (A t c) = let (t1, t2) = splitAt c t
                        in A (t1 ++ [x] ++ t2) (c+1)
deleteCharA (A t c) = let (t1, t2) = splitAt c t
                      in A (t1 ++ drop 1 t2) c
```

But do we need to keep track of all that information in our implementation? No!

Concrete Implementation

Our concrete version will just maintain two strings, the `left part` (in reverse) and the `right part` of the cursor:

Concrete Implementation

Our concrete version will just maintain two strings, the `left part` (in reverse) and the `right part` of the cursor:

```
einit s = C [] s
stringOf (C ls rs) = reverse ls ++ rs
moveLeft (C (l:ls) rs) = C ls (l:rs)
moveLeft c = c
moveRight (C ls (r:rs)) = C (r:ls) rs
moveRight c = c
insertChar x (C ls rs) = C (x:ls) rs
deleteChar (C ls (_:rs)) = C ls rs
deleteChar c = c
```

Refinement Functions

Abstraction function to express our refinement relation in a QC-friendly way: such a function:

```
toAbstract :: Concrete -> Abstract
toAbstract (C ls rs) = A (reverse ls ++ rs) (length ls)
```

Properties with Abstraction Functions

```
prop_init_r s =
  toAbstract (einit s) == einitA s
prop_stringOf_r c =
  stringOf c == stringOfA (toAbstract c)
prop_moveLeft_r c =
  toAbstract (moveLeft c) == moveLeftA (toAbstract c)
prop_moveRight_r c =
  toAbstract (moveRight c) == moveRightA (toAbstract c)
prop_insChar_r x c =
  toAbstract (insertChar x c)
  == insertCharA x (toAbstract c)
prop_delChar_r c =
  toAbstract (deleteChar c) == deleteCharA (toAbstract c)
```