COMP3141

Curtis Millar
CSE, UNSW (and Data61)
24 June 2020

Sort Properties

1. `sortFn xs == s`
2. `` x `elem` sor ``
3. `isSorted (`
4. `length xs == length (sortFn xs)`
5. `sortFn xs == insertionSort xs`

Dodgy Sort

1. Satisfy only (
2. Satisfy only (
3. Satisfy only (
4. Satisfy only (1), (2), (3), and (4)

Fractal Art

- Let's take a lo
- Assess your
  1. Is the fu
  2. Is the pi
     image other than recursion depth, size, and colour?
  3. Is it a real attempt to generate a nice image?
- Online form to review peers art & implementation on cou

Data Invariants

- Data invariants are statements that must always be true of a data structure. We generally represent these invariants as a *wellformedness predicate*, a function that tests wheth
- Data invar data type. The output of an


```
constructor :: .. -> X
```

- Data invarants must also be shown to be true for all functio value of a data type. The output of these functions must sati wellformedness predicate only if the input does.


```
fn :: .. -> X -> X
```

Abstract Data Types

- ADTs allow us to encapsulate the implementation of a data type by restricting access to whi
  outside th
- The ability t
  dependant on the language.
- If all the externally visible functions maintain the data inv
  code can construct a value that ever violates them.

Refinement

- A relation from an *implementation* to an *abstract model* or an *abstract specificati*
- If an implem behavior bu

  A refinement is the opposite of an abstraction, which rem
- In this course, the model and implementation will presen interface with different implementation details.

Data Refinement

- We can demonstrate a *refinement relation* between two data types if we can show that the interfaces are the same and they exhibit the same behavior. This is a *data refine*

- We choose w *definition of specificati*
- The other data type then becomes our *impl* at we will actually use in the final system.

- We must show that the implementation is a refinement o specification.

Data Refinement

Refinement and Specifications

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

1. all data invar
2. the implem

There is a limit to the amount of abstraction we can do before they become useless for testing (but not necessarily for proving).

Warning

While abstraction can simplify proofs, abstraction does not reduce the fundamental complexity of verification, which is provably hard.

Editor Example

Consider this ADT interface for a text editor:

```haskell
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_             s = wellfor
prop_moveLe
prop_moveRi
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:

```
einitA s = A s 0
stringOfA (A s _) = s
moveLeftA  (A t c) = A t (m
moveRightA (A t c
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:

```
einit s = C [] s
stringOf (C ls rs
moveLeft (C (l:
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 fun
function:


```haskell
toAbstract :: C
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 == s
prop_moveLe
  toAbstrac
```



```
prop_moveRight_r c =
  toAbstract (moveRight c) == moveRightA (toAbst
prop_insChar_r x c =
  toAbstract (insertChar x c)
  == insertCharA x (toAbstract c)
prop_delChar_r c =
  toAbstract (deleteChar c) == deleteCharA (toAbstract c)
```

Homework

1. Last week's
2. The third pr
3. The first assi
4. This week's quiz is also up, it's due next Friday (in 9 days).

Consultations

- Poll on Piazz
- Tomorro
- Link on cour

- Make sure to join the queue on Hopper. Be ready to share you (`ghci` or `stack repl`) and editor set up.