COMP3141

Software System Design and Implementation

Data Invariants, Abstraction and Refinement Practice

Curtis Millar
CSE, UNSW (and Data61)

24 June 2020

Sort Properties

1. `sortFn xs == sortFn (reverse xs)`
2. `` x `elem` sortFn (xs ++ [x] ++ ys) ``
3. `isSorted (sortFn xs)`
4. `length xs == length (sortFn xs)`
5. `sortFn xs == insertionSort xs`

Dodgy Sort

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

Fractal Art

- Let's take a look at the gallery
- Assess your peers
  1. Is the function which generates the image a recursive function?
  2. Is the picture function given parameters that influence at least two aspects of the 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 course website soon.

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 whether a value is well-formed.
- Data invaraints must be shown to be true for all *constructors* of a data type. The output of any constructor must satisfy the wellformedness predicate.

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

- Data invaraints must also be shown to be true for all functions that transform the value of a data type. The output of these functions must satisfy the 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 which functions can be used construct, query, or transform a value from *outside* the module in which it is defined.
- The ability to restrict access to certain implementation details is generally dependant on the language.
- If all the externally visible functions maintain the data invariants then no external code can construct a value that ever violates them.

Refinement

- A relation from an *implementation* to an *abstract model* or an *abstract specification*.
- If an implementation *refines* a model or specification, it exhibits all of the same behavior but may have additional behaviour or detail.
  A refinement is the opposite of an abstraction, which removes detail.
- In this course, the model and implementaion will present an indistingushable 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 refinement*.
- We choose which data type will be the *abstract model* which is the *definition* or *specification*.
- The other data type then becomes our *implementation*, i.e. the data type that we will actually use in the final system.
- We must show that the implementation is a refinement of the model or specification.

Data Refinement

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.

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:

```
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:

```
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:

```haskell
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)
```

Homework

1. Last week's quiz is due on Friday. Make sure you submit your answers.
2. The third programming exercise is due by the start if my next lecture (in 7 days).
3. The first assignment is due by the start if my next lecture (in 7 days).
4. This week's quiz is also up, it's due next Friday (in 9 days).

Consultations

- Poll on Piazza to register interest. Will not run if there are no votes.
- Tomorrow, 9am to 11am on Blackboard Collaborate.
- Link on course website & Piazza.
- Make sure to join the queue on Hopper. Be ready to share your screen with REPL (`ghci` or `stack repl`) and editor set up.