Testing

Prof. Clarkson
Fall 2016

Today’s music: Wrecking Ball by Miley Cyrus
Review

Previously in 3110:
• architecture and design of large programs
• specification of modules

Today:
• finish up specification
• testing
Review: Sets without duplicates

module ListSetNoDup : Set = struct
  (* the list may never have duplicates *)
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x l =
    if mem x l then l else x :: l
  let size = List.length
end
Review: Sets with duplicates

module ListSetDup : Set = struct
  (* the list may have duplicates *)
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x l = x :: l
  let rec size = function
    | [] -> 0
    | h::t -> size t +
       (if mem h t then 0 else 1 )
end
Review: Set implementations

• Both have the same representation type, 'a list
• But they interpret values of that type differently
  – [1;1;2] is {1,2} in ListSetDup
  – [1;1;2] is not meaningful in ListSetNoDup
  – In both, [1;2] and [2;1] are {1,2}
Review: Abstraction function

- **Abstraction function** (AF) captures designer’s intent in choosing a particular representation of a data abstraction
- Not actually an OCaml function, but a mathematical function
- Maps *concrete values* to *abstract values*

![Diagram](image)

client’s view

implementer’s view

abstract: set

concrete: lists (no dups)
module ListSetNoDup : Set = struct
  (* AF: the list [a1; ...; an] represents
  * the set \{a1, ..., an\}. [] represents
  * the empty set. *)
  type 'a set = 'a list
...
end

module ListSetDup : Set = struct
  (* AF: the list [a1; ...; an] represents
  * the smallest set containing the
  * elements a1, ..., an. [] represents
  * the empty set. *)
  type 'a set = 'a list
...
end

So far nothing other than module name specifies whether duplicates are allowed...
REPRESENTATION INVARIANT
Representation invariant

• **Representation invariant** characterizes which concrete values are *valid* and which are *invalid*
  – “Rep invariant” or "RI" for short
  – Valid concrete values mapped by AF to abstract values
  – Invalid concrete value not mapped by AF to any abstract values
  – Closely related to *class invariants* that you saw in 2110

• RI is a *fact whose truth is invariant* except for limited blocks of code...
  – (much like loop invariants from 2110)
Representation invariant

Operations of data abstraction:

- **Assume that any inputs from client satisfy RI**
  
  - e.g., `ListSetNoDup` operations assume that values passed in contain no duplicates

- **Internally might produce intermediate values that violate RI**
  
  - e.g., `ListSetNoDup` could temporarily be processing a list with duplicates

- **Must produce output to client that satisfy RI**
  
  - e.g., `ListSetNoDup` operations produce values that contain no duplicates
module ListSetNoDup : Set = struct
   (* AF: the list [a1; ...; an] represents
    * the set \{a1,...,an\}. [] represents
    * the empty set. *)
   (* RI: the list contains no duplicates *)
   type 'a set = 'a list
end

module ListSetDup : Set = struct
   (* AF: the list [a1; ...; an] represents
    * the smallest set containing the
    * elements a1, ..., an. [] represents
    * the empty set.
    * RI: none *)
   type 'a set = 'a list
end
Implementing the RI

- **Implement it early**, before any operations are implemented.
- Common **idiom**: if RI fails then raise exception, otherwise return concrete value

``` Ocaml
let rep_ok (x:'a list) : 'a list =
  if has_dups x then failwith "RI"
  else x
```

- When debugging, check **rep_ok** on every input to an operation and on every output...
Checking the RI

module ListSetNoDup : Set = struct
  (* AF: ... *)
  (* RI: ... *)
  type 'a set = 'a list
  let rep_ok = ...
  let empty = rep_ok []
  let mem x l = List.mem x (rep_ok l)
  let add x l =
    if mem x (rep_ok l) then (rep_ok l)
    else rep_ok(x :: l)
  let size l = List.length (rep_ok l)
end

Funny story...this saved a CS 3110 tournament one year
Checking the RI

• Can be expensive!

• For production code, options include...
  – only check “cheap parts” of RI
  – comment out "real" implementation, change `rep_ok` to identity function, let compiler optimize call away
  – use language features for condition compilation (in OCaml, CamlP4 or PPX)
CORRECTNESS OF OPERATIONS
AF and operations

Example: ListSetDup

union \{2,3\}

abstract operation

AF

\{1,2\}

concrete operation

append \{2,3\}

AF

\{1,2;3\}

[1;2]

[1;2;2;3]
AF and operations

abstract operation

implemented operation

*commutative diagram:* both paths lead to the same place
Correctness of operations

Implementation is correct if $AF$ commutes:

$$op_{\text{abs}}(AF(c)) = AF(op_{\text{conc}}(c))$$

- $c$ is a concrete value for which $RI$ holds
- $op_{\text{conc}}$ is the concrete implementation of the operation, e.g. list append
- $op_{\text{abs}}$ is the abstract operation (not implemented), e.g. set union
Recap: Specifying rep. types

• **Q:** How to *interpret* the representation type as the data abstraction?
  • **A:** Abstraction function

• **Q:** How to determine which values of representation type are *meaningful*?
  • **A:** Representation invariant
Black box testing

tester knows nothing about internals of functionality being tested
Black box testing

tester knows nothing about internals of functionality being tested
Glass box testing

Input → tester knows internals of functionality being tested → Output
Glass box testing

tester knows internals of functionality being tested
Black box testing

• Tests are based on the specification

• Advantages:
  – Tester is not biased by assumptions made in implementation
  – Tests are robust w.r.t. changes in implementation
  – Tests can be read and evaluated by reviewers who are not implementers

• Main kinds of black box tests:
  – Typical inputs
  – Boundary cases
  – Paths through spec
Typical inputs

- Common, simple values of a type
  - **int**: small integers like 1 or 10
  - **char**: alphabetic letters, digits
  - **string**: whose length is a small integer and whose characters are typical
  - **'a list**: a small integer number of elements, each of which is a typical value of type 'a
  - **records/tuples**: each field/component with a typical value
  - **variants**: typical constructors, if there is such a thing

- Any example inputs provided by specification
Boundary cases

A QA Engineer walks into a bar.
Orders a beer.
Orders 0 beers.
Orders 999999999 beers.
Orders a lizard.
Orders -1 beers.
Orders a sdfeljknesv.
@sempf
Boundary cases

• aka corner cases or edge cases
• Atypical or extremal values of a type, and values nearby
  – int: 0, 1, -1, min_int, max_int
  – char: '\000', '\255', '\032' (space), '\127' (delete)
  – string: empty string, string with a single character, unreasonably long string
  – 'a list: empty list, list with a single element, list with enough elements to trigger stack overflow on non-tail-recursive functions
  – records/tuples: combinations of atypical values
  – variants: all constructors
Paths through spec

Representative inputs for classes of outputs

(* [is_prime n] is true iff [n] is prime *)
val is_prime: int -> bool

two classes of output:
• true: representative input: n=13
• false: representative output: n=42

compare functions have three classes of output
functions that return variants have several classes of output
Paths through spec

Representative inputs for each way of satisfying the precondition

(* [sqrt x n] is the square root of [x]
 * computed to an accuracy of [n]
 * significant digits
 * requires: x >= 0 and n >= 1 *)

val sqrt : float -> int -> float

x=0.0, n=1,  x=1.0, n=1,  x=0.0, n=2,  x=1.0, n=2
Paths through spec

Representative inputs for each way of (not) raising exception

(* [pos x lst] is the 0-based position of
  * the first element of [lst] that equals [x].
  * raises: Not_found if [x] is not in [lst]. *)

val pos: 'a -> 'a list -> int

x=1, lst=[1],  x=0, lst[1]
Glass box testing

• aka *white box testing*

• **Advantages:**
  – can determine whether a new test case really yields additional information about correctness of implementation
  – can address likely errors that are not apparent from specification

• **Supplements** black-box testing; does not **replace** examination of specification

• **Main kind of glass box test cases:**
  – *paths through implementation* aka *path coverage*
Paths through implementation

All execution paths through implementation are tested

```haskell
let max3 x y z =
    if x > y then
        if x > z then x else z
    else
        if y > z then y else z
```

Testing according to black-box specification might lead to all kinds of inputs

But there are really only four paths through implementation!
Representatives:  3 2 1,   3, 2, 4,   1, 2, 1,   1, 2, 3
Achieving path coverage

• Include test cases for:
  – each branch of each (nested) if expression
  – each branch of each (nested) pattern match

• Particularly watch out for:
  – base cases of recursive function
  – recursive calls in recursive function
  – every place where an exception might be raised
Testing data abstractions

• Some functions of a data abstraction *produce* a value of it
  – *empty* produces an empty set
  – *union* produces a set

• Other functions *consume* a value
  – *size* consumes a dictionary and produces an integer
  – *bindings* consumes a dictionary and produces a list

• For every possible path through spec and impl of producers... test how a consumer handles it
  – e.g. if producers of a set handle sets of size 0, 1, and >1 differently...
  – then test each such set with every consumer

• For every value returned by abstraction, check the RI
Randomized testing

[Barton Miller, 1989, 2000, 2006]
• "It was a dark and stormy night..."
• Generate random inputs and feed them to programs:
  – Crash? hang? terminate normally?
  – Of ~90 utilities in '89, crashed about 25-33% in various Unixes
  – Crash => buffer overflow potential
• Since then, "fuzzing" has become a standard practice for security testing
• Results have been repeated for X-windows system, Windows NT, Mac OS X
  – Results keep getting worse in GUIs but better on command line
Debugging

- Testing reveals a fault in the program
- Debugging reveals the cause of that fault
- "Bug" is misleading
  - it didn't just crawl into the program
  - you or I put it there
- Debugging takes more time than programming
  - get it right the first time!
  - understand exactly why you expect code to work before testing/debugging it
Debugging advice

• Follow the scientific method:
  – formulate a falsifiable hypothesis
  – create an experiment that can refute that hypothesis
  – run that experiment
  – keep a lab notebook

• Find the simplest possible input that causes fault

• Find the first manifestation of inappropriate behavior
Debugging advice

- Invest effort in writing code to help you understand intermediate results
- The bug is probably not where you think it is...ask yourself where it cannot be
- Get someone else to help you: duck debugging
- If all else if failing, doubt your sanity: do you have the right compiler? the right source code
- Don't debug when angry or tired: give it a break; come back refreshed
- Think through the fix carefully: "fixing" a bug often leads to new bugs
Upcoming events

• [next Wed] A3 due
• [Oct 13, two weeks from today] Prelim 1, look for Piazza post soon

This is tested.

THIS IS 3110
Acknowledgment

Parts of this lecture are based on Program Development in Java: Abstraction, Specification, and Object-Oriented Design by Barbara Liskov with John Guttag.