Specifications

Sebastiaan Joosten

University of Minnesota
Driven to Discover®

Motivation

- When choosing a particular data-structure, I care about:
  - How to create the structure
  - What operations I can perform
    - and what these operations do
  - The performance
- If someone else implemented it for me,
  I don’t really care about the data-structure
- An algebraic specification is an abstraction that focuses on the operations, not the data-structure
  (or performance)

Algebraic specifications in ocaml

- Ocaml lets you specify the operations in a signature
- The types give *some* information on what the operations do
- The comments should give the remainder of the information

- Signatures (with comments) are precisely the right abstraction level for specifications!

Example specification (taken from Batteries)

```
module BatDeque: sig .. end
type 'a t = 'a dq
```

A synonym for convenience

Construction

```
val empty : 'a dq
```

The empty deque.

```
val cons : 'a -> 'a dq -> 'a dq
```

`cons x dq` adds `x` to the front of `dq`. O(1)

```
val snoc : 'a dq -> 'a -> 'a dq
```

`snoc x dq` adds `x` to the rear of `dq`. O(1)

Deconstruction

```
val front : 'a dq -> ('a * 'a dq) option
```

`front dq` returns `Some (x, dq')` iff `x` is at the front of `dq` and `dq'` is the rest of `dq` excluding `x`, and `None` if `dq` has no elements. O(1) amortized, O(n) worst case

```
val rear : 'a dq -> ('a dq * 'a) option
```

`rear dq` returns `Some (dq', x)` iff `x` is at the rear of `dq` and `dq'` is the rest of `dq` excluding `x`, and `None` if `dq` has no elements. O(1) amortized, O(n) worst case

Example specifications

- These are abstract specifications for:
  - implementers
  - users
- It’s abstract because it does not say what datatypes are used, just what the consequences of those choices are
- They leave things implicit that we take for granted:
  - after adding to the front of the queue, the remainder stays unchanged
  - after taking from the front of the queue, the remainder stays unchanged
  - etc.
- It’s not so suitable for proving and testing

Here is an alternative specification:

- front empty = None
- rear empty = None
- front (cons x xs) = Some (x, xs)
- rear (snoc x xs) = Some (x, xs)

- This specification is:
  - Algebraic: it consists of equalities (between ocaml terms)
  - Formal: it is written in a set language (here: algebraic+ocaml) with a clear interpretation
  - Great for testing and proving! (But not necessarily intuitive)
  - Incomplete and overly restrictive at the same time :-(

Coming up with a specification: terminology

- A module with an abstract specification will generally have three kinds of operations:
  - Construction: Functions to create things of type t
  - Deconstruction: Functions to observe what things of type t look like
  - Manipulation: Functions that don’t really fit in either category.
- A function of type ‘.. -> t’ is often a Construction function. If the .. do not contain ’t’, it is always a Construnction function.
- A function using things of type ‘t’ as arguments, that does not result in a value with ‘t’ in it, is always a Deconstruction function.
- Remaining functions (something of type t occurs in an argument and in the result) can be of all three kinds (often it’s a matter of opinion).
- Note: A function whose type does not have a t in it, is neither and should not be in the module.

Coming up with a specification

- The only way to observe what a function does, is through the deconstruction functions.
- The implementation should be irrelevant, so the type of both sides of the rule should never be t
- As a consequence, any rule (L = R) will use a deconstruction function.

Coming up with a specification

- Every function / constant should occur in some part of the specification. Together with the rule to use a deconstructor, this makes it easy to come up with these:

- front empty = None
- rear empty = None
- front (cons x xs) = Some (x, xs)
- rear (snoc x xs) = Some (x, xs)

Coming up with a complete specification

- It’s a whole separate science to decide whether a specification is complete or not
- If we can proof what we want to, our specification is ‘complete enough’
- If there’s a wrong implementation that our specification does not rule out, it is not complete enough
- If there’s a correct implementation that our specification does rule out, it is overly restrictive

A wrong implementation

- front empty = None
- rear empty = None
- front (cons x xs) = Some (x, xs)
- rear (snoc x xs) = Some (x, xs)

- Can be satisfied by:
- type ‘a t = ‘a list
- let empty = []
- let cons x xs = x::xs
- let snoc x xs = x::xs
- let front = function [] -> None | (x::xs) -> Some (x, xs)
- let rear = function [] -> None | (x::xs) -> Some (x, xs)

Let’s do the proofs!

A wrong implementation

- front empty
  = {empty def}
  front []
  = {front def}
  match [] with [] -> None | (x::xs) -> Some (x,xs)
  = {match}
  None

- type 'a t = 'a list
- let empty = []
- let cons x xs = x::xs
- let snoc x xs = x::xs
- let front = function [] -> None | (x::xs) -> Some (x, xs)
- let rear = function [] -> None | (x::xs) -> Some (x, xs)

front empty = None !

A wrong implementation

- rear empty
  = {empty def}
  rear []
  = {rear def}
  match [] with [] -> None | (x::xs) -> Some (x,xs)
  = {match}
  None

- type ‘a t = ‘a list
- let empty = []
- let cons x xs = x::xs
- let snoc x xs = x::xs
- let front = function [] -> None | (x::xs) -> Some (x, xs)
- let rear = function [] -> None | (x::xs) -> Some (x, xs)

rear empty = None !

A wrong implementation

- front (cons x xs)
  = {cons def}
  front (x :: xs)
  = {front def}
  match x :: xs with [] -> None | (x::xs) -> Some (x,xs)
  = {match}
  Some (x, xs)

- type ‘a t = ‘a list
- let empty = []
- let cons x xs = x::xs
- let snoc x xs = x::xs
- let front = function [] -> None | (x::xs) -> Some (x, xs)
- let rear = function [] -> None | (x::xs) -> Some (x, xs)

front (cons x xs) = Some (x, xs)

A wrong implementation

- rear (snoc x xs)
  = {snoc def}
  rear (x :: xs)
  = {rear def}
  match x :: xs with [] -> None | (x::xs) -> Some (x,xs)
  = {match}
  Some (x, xs)

- type ‘a t = ‘a list
- let empty = []
- let cons x xs = x::xs
- let snoc x xs = x::xs
- let front = function [] -> None | (x::xs) -> Some (x, xs)
- let rear = function [] -> None | (x::xs) -> Some (x, xs)

rear (snoc x xs) = Some (x, xs)

A wrong implementation

- front empty = None
- rear empty = None
- front (cons x xs) = Some (x, xs)
- rear (snoc x xs) = Some (x, xs)

- type ‘a t = ‘a list
- let empty = []
- let cons x xs = x::xs
- let snoc x xs = x::xs
- let front = function [] -> None | (x::xs) -> Some (x, xs)
- let rear = function [] -> None | (x::xs) -> Some (x, xs)

Why is this implementation wrong?

Completing our specification

- Here’s a property that will fail for our implementation:
- match front xs, front (snoc y xs) with
  | Some (a, _), Some (b, _) -> a = b
  | None, Some (b, _) -> y = b
  | _, _ -> false

- type ‘a t = ‘a list
- let empty = []
- let cons x xs = x::xs
- let snoc x xs = x::xs
- let front = function [] -> None | (x::xs) -> Some (x, xs)
- let rear = function [] -> None | (x::xs) -> Some (x, xs)

Completing our specification

- Here's a property that will fail:
- match front xs, front (snoc y xs) with

```
| Some (a, _), Some (b, _) -> a = b
| None, Some (b, _) -> y = b
| _, _ -> false
```

- Ugly properties like this will make it hard to prove things
- Coming up with 'beautiful' properties is not always possible
- Different function types can help improve things

Our specification is overly restrictive

- Take a look at this property:
- front (cons x xs) = Some (x, xs)

- This requires that the xs on both sides is *identical* according to ocaml's identity.
- All we really require is that:
- front (cons x xs) = Some (x, xs')
  for some xs', such that xs and xs' *behave* the same.

- Let's look at the difference between these things!

A deque implementation

- A common way to implement a queue is by using two stacks (i.e. regular lists), and reversing one when the other is empty:

- type ’a t = ('a list * 'a list)
- let empty = ([], [])
- let cons x (a,b) = (x::a, b)
- let snoc x (a,b) = (a, x::b)
- let front x = function

```
  | (x::xs,ys) -> Some x (xs,ys)
  | ([],ys) -> (match (List.rev ys) with
        [] -> None | (h::tl) -> Some h (tl, []))
```

Two similar queues

- These queues now represent the same data:
- ([1;2;3],[])
- ([1;2],[3])
- ([1],[3;2])
- ([],[3;2;1])

- From the perspective of the user, we might be tempted to say: ([1;2;3],[]) = ([],[3;2;1])
(since the implementation is hidden!)
- From the perspective of the implementer, this is not true!

An optimization

- Our ‘deque’ implementation has a ‘front’ and a ‘rear’, which could lead to this unfortunate behavior:
- rear ([1;2;3;4],[]) = Some (4, ([],[3;2;1]))
- front ([],[3;2;1]) = Some (1, ([2;3],[]))
- rear ([2;3],[]) = Some (3, ([],[2]))
- … we keep reversing the list!

- To avoid this and get O(1) complexity back, the two lists are kept ‘of similar size’
- I won’t discuss the precise conditions here

A possible run

- cons (1, ([2;3;4;5;6],[]))
  = {according to some implementation}
  ([1;2;3],[6;5;4])
- front ([1;2;3],[6;5;4])
  = {according to some implementation}
  Some (1, ([2;3],[6;5;4]))

- This implementation violates:
  front (cons x xs) = Some (x, xs)
- But it satisfies:
  front (cons x xs) = Some (x, xs')
  for *some* xs' where xs and xs' *behave* the same.

Using = more conveniently

- The = is defined in ocaml as structural equality.
  I don't want to change its definition.
  (if I did, I could fix the 'error' in the textbook)
- We can use a different symbol, say ≡, to indicate that two things are equal *in behavior*.
- I'll define it somewhat informally:
  - x ≡ y = eq x y for 'eq' as defined in our module
  - x ≡ y = x = y for x, y : int, float, … (basic built-in type)
    (a, b) ≡ (c, d) = (a = c && b = d)
    Some a ≡ None = false
    Some a ≡ Some b = (a = b)
    … similar for all other exposed types (this is the informal part)

Using ≡ …

- We'd need to prove that we can use ≡ in the same way we have used =
- .. which would require a course in logic
- .. and it would also give us precise conditions that our 'eq' implementation needs to satisfy
- We'll see a clever way of defining an 'eq' function on Wednesday
  (that will satisfy the conditions not mentioned here)

Final remarks

- Coming up with accurate specifications is hard
- It’s good to be pragmatic sometimes:
  - If you can prove what you need to prove, it’s enough
  - If you’re not doing proofs, aim for testability!