CS 3110

Abstraction and Specification

Prof. Clarkson
Fall 2017

Today's music: "A Fifth of Beethoven" by Walter Murphy

Review

Previously in 3110:

- Language features for modularity
- Some familiar data structures

Today:

- Abstraction and specification
- Clients vs. implementers
- Specification of functions

Question

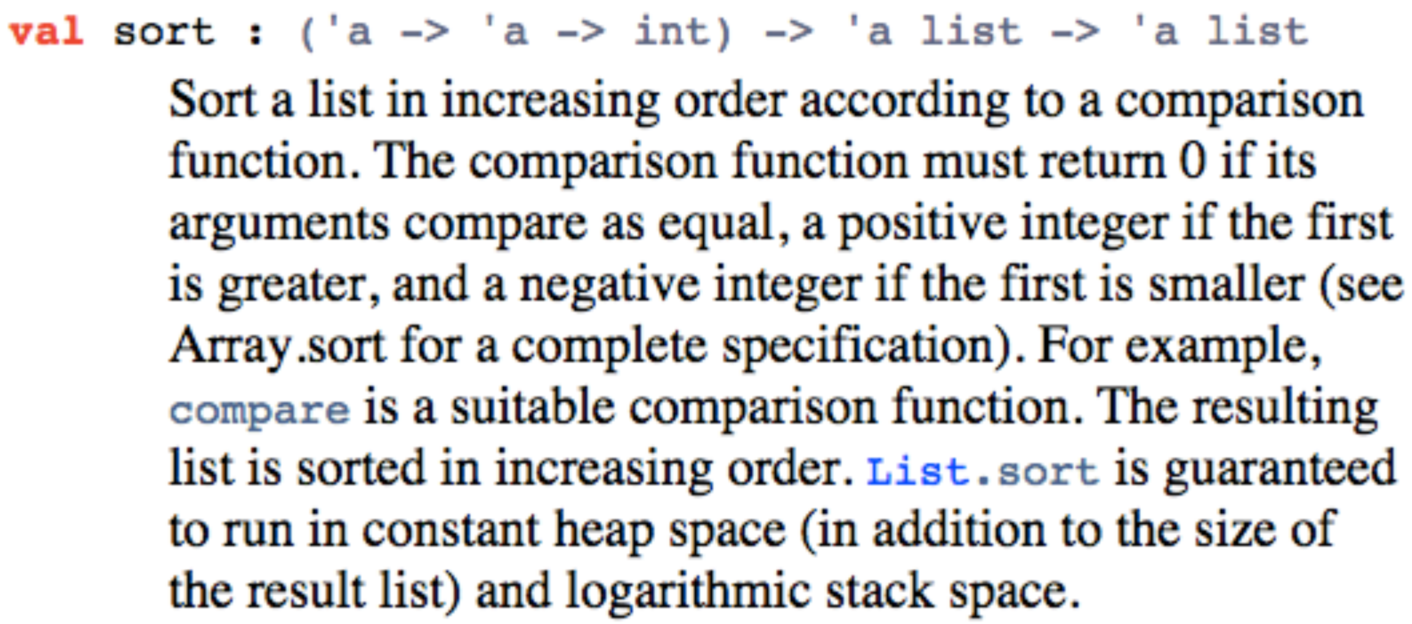
Think about `java.util` (or some other library you've used frequently). How do you usually come to understand the functionality it provides?

A. **By example:** I search until I find code using the library, then tweak the code to do what I want.
B. **By tutorial:** I read the library's tutorial to understand how it works, then I write code inspired by it.
C. **By documentation:** I read the official documentation for functions, classes, etc., in the library, then I write code from scratch.
D. **By implementation:** I download the source code for the library, read it, then write my own code.
E. I never really understood `java.util`.

What if you had to read the implementation?

```
let rec sort n l =
    match n, l with
    | 2, x1 :: x2 :: _ ->
       if cmp x1 x2 <= 0 then [x1; x2] else [x2; x1]
    | 3, x1 :: x2 :: x3 :: _ ->
       if cmp x1 x2 <= 0 then begin
         if cmp x2 x3 <= 0 then [x1; x2; x3]
         else if cmp x1 x3 <= 0 then [x1; x3; x2]
         else [x3; x1; x2]
       end else begin
         if cmp x1 x3 <= 0 then [x2; x1; x3]
         else if cmp x2 x3 <= 0 then [x2; x3; x1]
         else [x3; x2; x1]
       end
    | n, l ->
       let n1 = n asr 1 in
       let n2 = n - n1 in
       let l2 = chop n1 l in
       let s1 = rev_sort n1 l in
       let s2 = rev_sort n2 l2 in
       rev_merge_rev s1 s2 []
…
```

Example specification



```
val sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, `compare` is a suitable comparison function. The resulting list is sorted in increasing order. `List.sort` is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

Exercise: take 2 minutes. Feel free to talk with someone near you. Identify any preconditions and postconditions.

Example specification

- **One-line summary of behavior:** *Sort a list in increasing order according to a comparison function.*
- **Precondition:** *The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, compare is a suitable comparison function.*
- **Postcondition:** *The resulting list is sorted in increasing order.*
- **Promise about efficiency:** *List.sort is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.*

Question

What grade would you give the List.sort specification?

A. It provides pre- and postconditions. They are specific enough for me to understand how to use the function as a client. They do not contain irrelevant details or vague descriptions.
B. Parts of the specification are hard to understand. Some details are missing, or some parts are vague.
C. The specification is confusing or just plain wrong.

Abstraction and specifications

- Abstraction: an entity that results from forgetting information, so that different things can be treated the same
  - E.g., a function or a module
- Specification: a contract between an implementer of an abstraction and a client of an abstraction
  - Describes behavior of abstraction
  - Clarifies responsibilities
  - Makes it clear who to blame

Implementations and specifications

An implementation satisfies a specification if it provides the described behavior

Many implementations can satisfy the same specification

- Client has to assume it could be any of them
- Implementer gets to pick one

Specification

Writing good specs is hard:

- the language and compiler do not demand it
- if you're coding only for yourself, neither do you

Reading specs is also hard:

- requires close attention to detail

ABSTRACTION BY SPECIFICATION

Abstraction by specification

- Document behavior of function
  - Summary of behavior
  - Pre- and post-conditions
  - Sample usages

- Specification is a kind of abstraction:
  - Forgetting about details
  - Use documentation to reason about behavior instead of having to read implementation

Benefits of abstraction by specification

- **Locality:** abstraction can be understood without needing to examine implementation
  - critical in implementing large programs
  - also important in implementing smaller programs in teams

- **Modifiability:** abstraction can be reimplemented without changing implementation of other abstractions
  - update standard libraries without requiring world to rewrite code
  - performance enhancements: write the simple slow thing first, then improve bottlenecks as necessary (cf. A3!)

Good specifications

- **Sufficiently restrictive:** rule out implementations that wouldn't be useful to clients
  - common mistakes: not stating enough in preconditions, failing to identify when exceptions will be thrown, failing to specify behavior at boundary cases

- **Sufficiently general:** do not rule out implementations that would be useful to clients
  - common mistakes: writing operational specifications instead of definitional (saying how, not what), stating too much in a postcondition

Goal is to write specifications that balance being restrictive and general

When to write specifications

- **During design:**
  - as soon as a design decision is made, document it in a specification
  - posing and answering questions about behavior clarifies what to implement

- **During implementation:**
  - update specification during code revisions
  - a specification becomes obsolete only when the abstraction becomes obsolete

Audience of specification

- **Clients**
  - Spec informs what they must guarantee (preconditions)
  - Spec informs what they can assume (postconditions)
- **Implementers**
  - Spec informs what they can assume (preconditions)
  - Spec informs what they must guarantee (postconditions)

SPECIFYING FUNCTIONS

A template for spec. comments

```
(**
 * returns: [f x] is ...
 * example: ...
 * requires: ...
 * raises: ...
 * effects: ...
 *)
val f : t1 ...-> t2
```

From *Abstraction and Specification in Program Development*

(Now *Program Development in Java: Abstraction, Specification, and Object-Oriented Design)*

By Barbara Liskov and John Guttag

Barbara Liskov

b. 1939

Turing Award Winner 2008

For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing.

Requires clause

```
(**
 * returns: [hd lst] is the head of [lst].
 * requires: [lst] is non-empty.
 *)
val hd : 'a list -> 'a
```

- Aka precondition
- Total function: well-defined behavior for all inputs: no requires clause needed
- Partial function: some inputs lead to unspecified behavior: requires clause needed for clients to use function correctly
- Blame is on client for not passing arguments that satisfy requires clause
- Types of arguments are not part of it, because compiler guarantees can never call with arguments of wrong type – Python programmers take note!
- Rather, the value specification (and type in it) is part of entire spec along with comment

OCamldoc

```
(**
 * returns: [f x] is ...
 *)
```

- Double asterisk signifies comment to extract to HTML (or another format)
- Text in square brackets formatted as source code in extracted comments

Returns clause

```
(**
 * returns: [sort lst] is a list
 *   containing the same elements of
 *   [lst], but sorted in ascending order.
 *)
val sort : int list -> int list
```

- Aka postcondition
- “returns:” prefix is optional, but clause itself is required for every function
- Usually phrased in terms of application and specifies an equality - “is”
- Blame is on implementer if function doesn’t guarantee behavior specified in returns clause (unless client violates requires clause)
- Again, type of return value is not part of the clause

Example clause

```
(**
 * example: [sort [1;3;2]] is [1;2;3].
 *)
val sort : int list -> int list
```

- Optional clause, but can be super helpful to humans
- Could provide multiple examples of course
- Examples become natural unit test cases, too

Raises clause

```
(**
 * returns: [hd lst] is the head of [lst].
 * requires: [lst] is non-empty.
 * raises: [Failure "hd"] if [lst] is empty.
 *)
val hd : 'a list -> 'a
```

- A second pre+postcondition
- Specifies when exception **must** be raised: implementer would be at fault if function instead returned a normal value instead of raising exception
- Can make partial function be total

Exceptions

- Standard library has a couple good exceptions pre-defined:
  - `Invalid_argument of string`: argument does not "make sense"
  - `Failure of string`: function is undefined on an argument, client should not pattern match on string which might change
  - When to use which? Unclear; designer of OCaml suggests Failure is a bit of a legacy from earlier design and today it would be better to define your own exceptions
- Always good to define your own exception types for communicating particular errors

Do I need to `assert` the precondition?

- **If stated as requires:** no
  - It's the client's fault when violated
  - Implementer is allowed to catch the machine on fire
  - And checking it might be computationally expensive
- **If stated as raises:** no
  - Implementer must raise exception under given condition
  - But clause says which exception it must be, not necessarily `Assert_failure`
- **Asserting the precondition** is a (great!) defensive programming technique that is independent of specification

Effects clause

```
(**
 * returns: [process_grades gs] is the
 *   number of non-zero grades in [gs]
 * effects: prints the non-zero grades
 *   in [gs] to standard output
 *)
val process_grades : grade list -> int
```

- Another kind of postcondition: guarantees to have some side-effect
- Other kinds of effects: writing to a file, reading input, [later in course] mutation
- If return type is unit, can omit returns clause and have just effects

Where to write specifications

- Put specs where clients will find them
  - In signature
  - Usually in .mli file
- Not where implementer will write code
  - In structure
  - Usually in .ml file
- And don't duplicate them between .ml and .mli!

Upcoming events

- [Wed] A1 due
- [Thur] A2 out

This is abstract.

THIS IS 3110