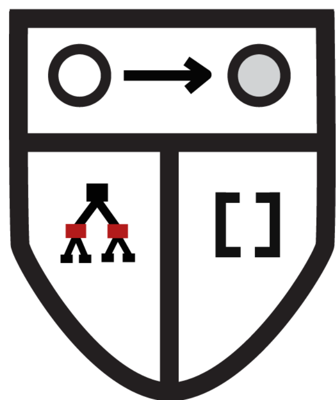


CS 3110

Abstraction Functions and Representation Invariants

Greg Morrisett

Spring 2018

Review

Previously in 3110:

- Abstraction and specification
- Specifying functions

Today:

- Specifying data abstractions
- Representation types
- Abstraction functions
- Representation invariants

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!

Back to: 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)

But the spec isn't **enough** for implementers...

REPRESENTATION TYPES

Example: sets

```
module type Set = sig
  type 'a t
  val empty : 'a t
  val mem   : 'a -> 'a t-> bool
  val add   : 'a -> 'a t-> 'a t
  val size  : 'a t-> int
end
```

Sets without duplicates

```
module ListSetNoDup : Set = struct
  (* the list may never have duplicates *)
  type 'a t = '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
```

Sets with duplicates

```
module ListSetDup : Set = struct
   (* the list may have duplicates *)
   type 'a t = '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
```

Compare 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}
- Interpretation differs because they make different assumptions about what values of that type can be:
  - passed into operations
  - returned from operations
- e.g.,
  - `[1;1;2]` can be passed into and returned from `ListSetDup`
  - `[1;1;2]` should not be passed into or returned from `ListSetNoDup`

Question

Consider this implementation of *set union* with representation type **`'a list`**:

```
let union l1 l2 = l1 @ l2
```

Under which invariant on representation type will that implementation be correct?

A. There may be duplicates in lists
B. There may not be duplicates in lists
C. Both A and B
D. Neither A nor B

Question

Consider this implementation of *set union* with representation type `'a list`:

```
let union l1 l2 = l1 @ l2
```

Under which invariant on representation type will that implementation be correct?

A. There may be duplicates in lists
B. There may not be duplicates in lists
C. Both A and B
D. Neither A nor B

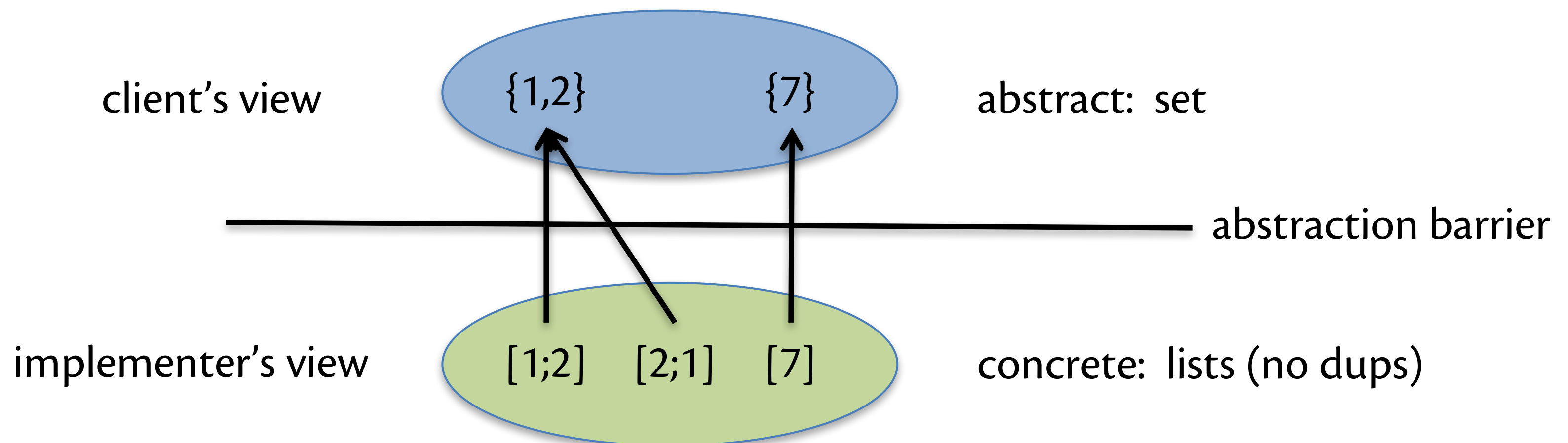
Representation type questions

- **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

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*



AF properties

- *Many-to-one*: many values of concrete type can map to same value of abstract type
  - `[1;2]` maps to {1,2}, as does `[2;1]`
- *Partial:* some values of concrete type do not map to any value of abstract type
  - `[1;1;2]` (in no dups) does not map to any set

Documenting AFs

```
module ListSetNoDup : Set = struct
  (* AF: the list [a1; ...; an] represents
   *    the set {a1,...,an}.  [] represents
   *    the empty set. *)
  type 'a t = '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 t = 'a list
  ...
end
```

Documenting AFs

- You might write:
  - `(* Abstraction Function: comment *)`
  - `(* AF: comment *)`
- You write it FIRST
  - It's the number one decision you have to make while implementing a data abstraction
  - It gives meaning to representation
  - It dictates what values are necessary in a module, or what fields are necessary in an object, or what

Implementing AFs

- Mostly you don't
  - Would need to have an OCaml type for abstract values
  - If you had that type, you'd already be done...
- But sometimes you do something similar:
  - **`string_of_X`** or **`to_string`** or **`format`**
  - quite useful for debugging

Duplicates?

```
module ListSetNoDup : Set = struct
  (* AF: the list [a1; ...; an] represents
   *    the set {a1,...,an}.  [] represents
   *    the empty set. *)
  type 'a t = '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 t = 'a list
  ...
end
```

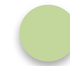
So far, nothing other than name of module specifies whether duplicates are allowed…

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)
  - RI is implicitly part of pre- and post-conditions
  - operations may violate it temporarily (e.g., construct a list with duplicates then throw out the duplicates)

Representation invariant

concrete
input



concrete
output

RI holds

RI holds

RI maybe violated

Documenting RI

```
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 t = '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 t = '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

```
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 t = 'a list
  let rep_ok = ...
  let empty = rep_ok []
  let mem x l = List.mem x (rep_ok l)
  let add x l =
    let l' = rep_ok l in
    if mem x l' then l'
    else rep_ok(x :: l')
  let size l = List.length (rep_ok l)
end
```

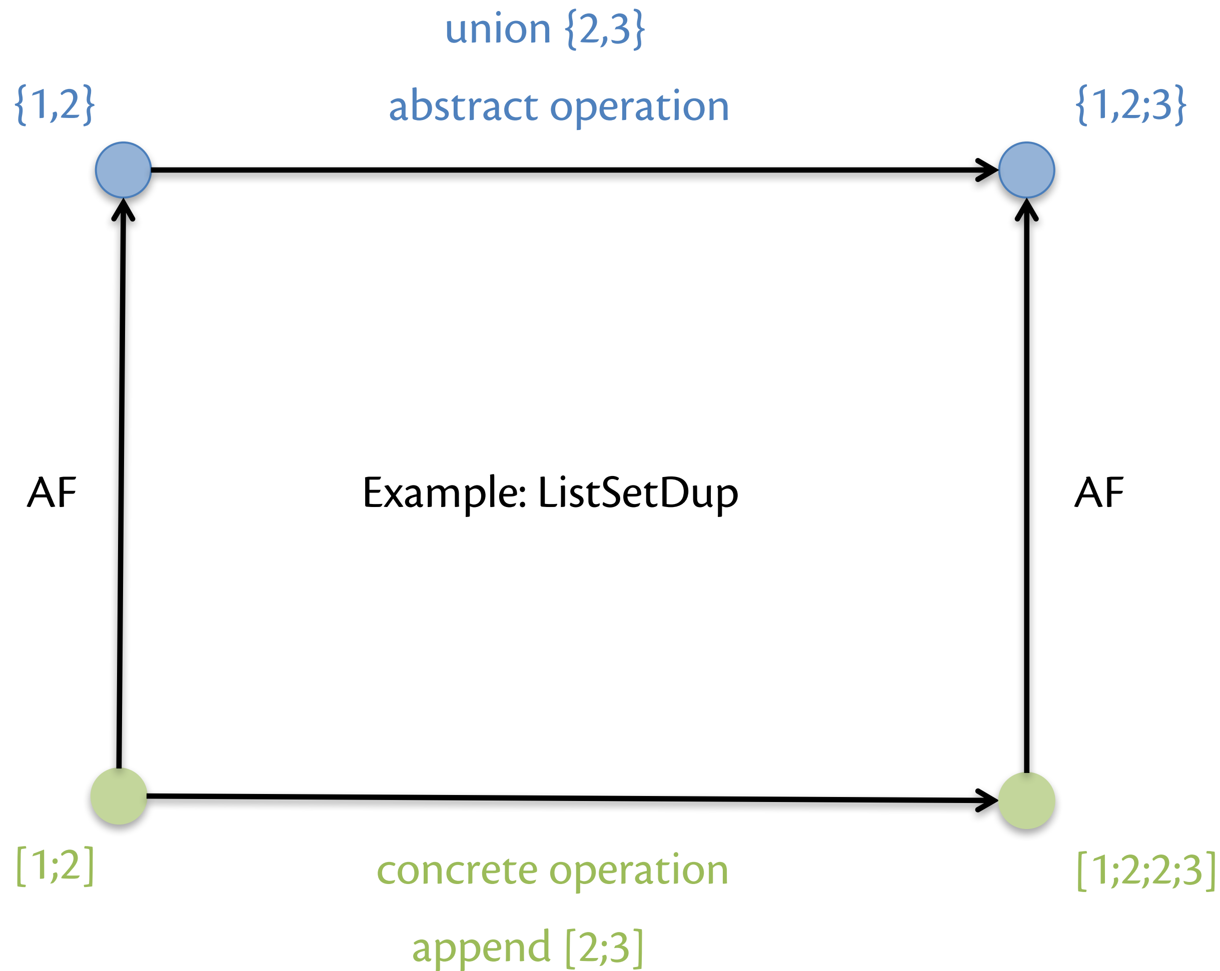
Funny story…this saved a CS 3110 tournament one year

Checking the RI

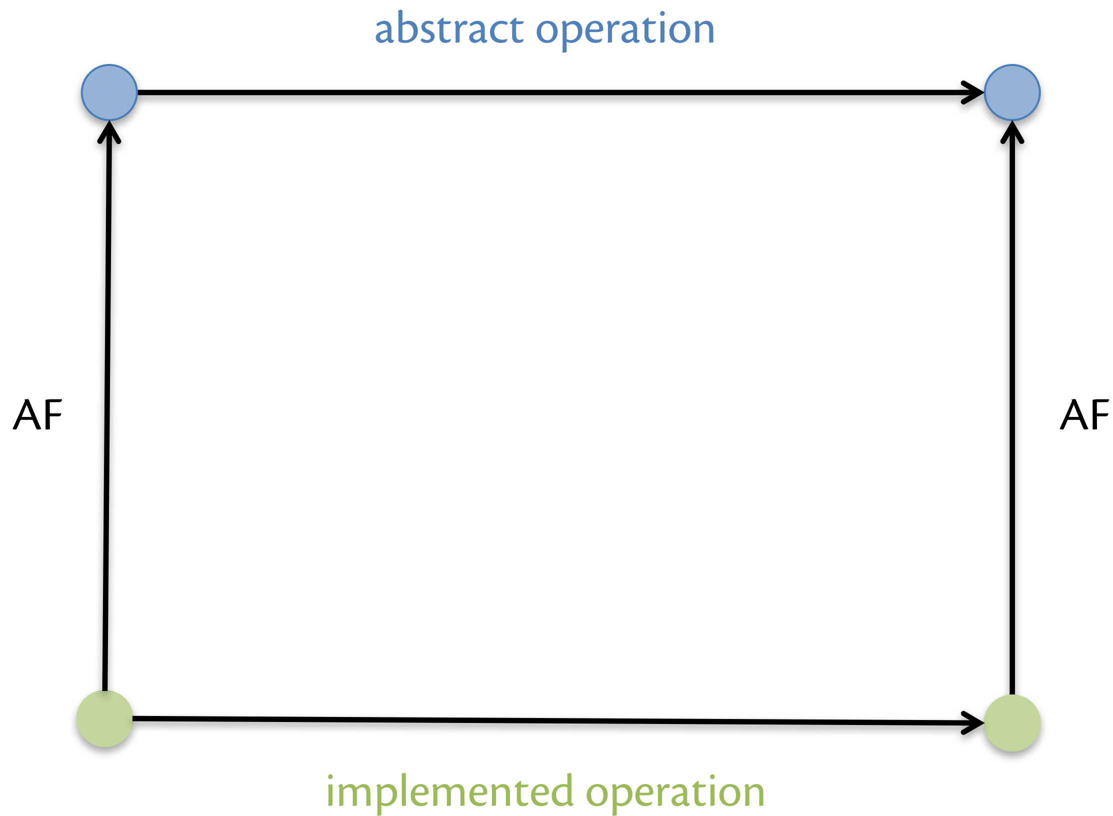
- Can be expensive!
- For production code, options include...
  - only check “cheap parts” of RI
  - comment out "real" implementation, change **`rep_ok`** to identity function, let compiler optimize call away
  - use language features for condition compilation (in OCaml, CamlP4 or PPX)

CORRECTNESS OF OPERATIONS

AF and operations



AF and operations



commutative diagram: both paths lead to the same place

Correctness of operations

Implementation is correct if AF commutes:

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

- c is a concrete value **for which RI holds**
- op_{conc} is the concrete implementation of the operation, e.g. list append
- op_{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

Upcoming events

- [March 9th] A2 due