CS 3110

Lecture 11: Documenting abstractions

Prof. Clarkson

Fall 2014

Today's music: *In C* by Terry Riley

Review

This week:

- Programming in the large
  - Modules, signatures, functors
  - Modularity, abstraction, specification

- Today:
  - Documentation for clients: specifications
  - Documentation for implementers: abstraction functions and representation invariants

Question #1

How much of PS3 have you finished?

A. None
B. About 25%
C. About 50%
D. About 75%
E. I'm done!!!

PS3 Quadtree Requirements

- We left a lot *unspecified*
  - You get to make implementation choices
  - This was deliberate!
  - (Early drafts left even more unspecified)
- Programming assignments in 3110 become increasingly less specified
- Programming assignments in 4000-level classes are even less specified
- Programming projects IRL might be entirely unspecified
- Success in programming (and in life) depends increasingly on you:
  - figuring out what's important and concentrating on that
  - making reasonable and defensible choices
  - being creative ☺

Review: example specification

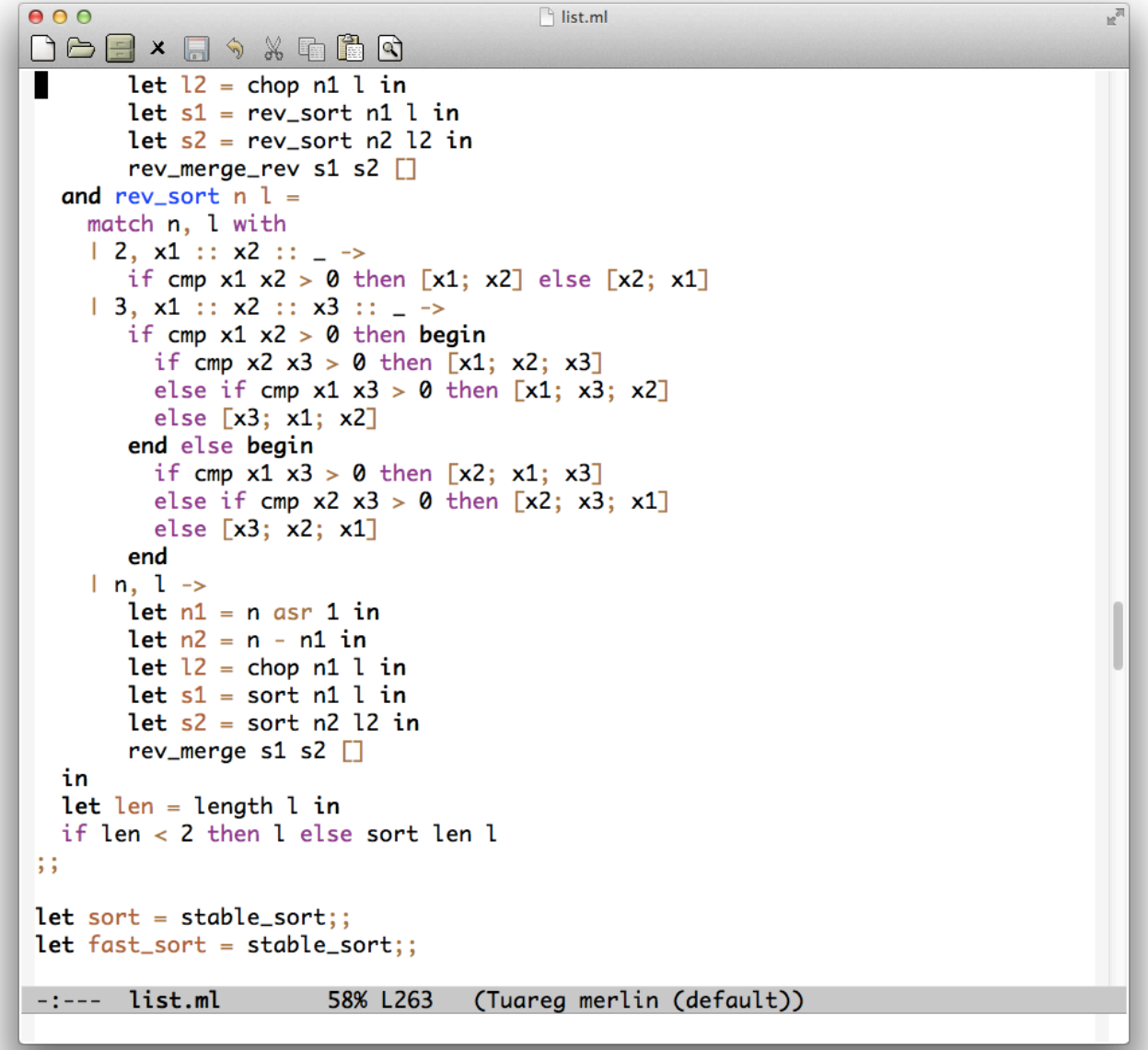
```
val sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

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.

Review: 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 behavior:** List.sort is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

What if you had to read the implementation?



Specifications

A **specification** is 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

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

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

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
- **Sufficiently clear:** easy for clients to understand behavior
  - common mistakes: verbosity, omission of details and examples, lack of structure
  - best case: client reads spec and comes away confused
  - worst case: client read spec, thinks they understand it, but they don't hence can't use abstraction correctly

Goal is to write specifications that are restrictive AND general AND clear

When to write specifications

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

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

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
```

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 size l = match l with
      [] -> 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 #2

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

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

Under which assumptions about representation type will that implementation be correct?

A. There are no duplicates in lists
B. There could be duplicates in lists
C. Both A and B
D. Neither A nor B

Question #2

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

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

Under which assumptions about representation type will that implementation be correct?

A. There are no duplicates in lists

B. There could be duplicates in lists

C. Both A and B

D. Neither A nor B

Representation type questions

- How to interpret the representation type as the data abstraction?

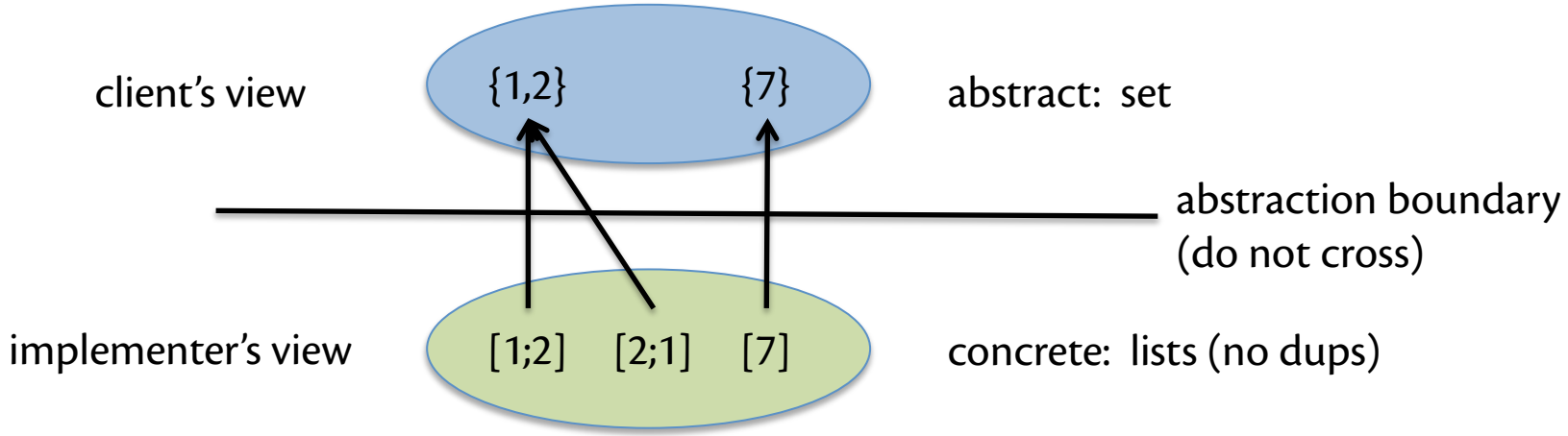
  ...abstraction function

- How to determine which values of representation type are meaningful?

  ...representation invariant

Abstraction function

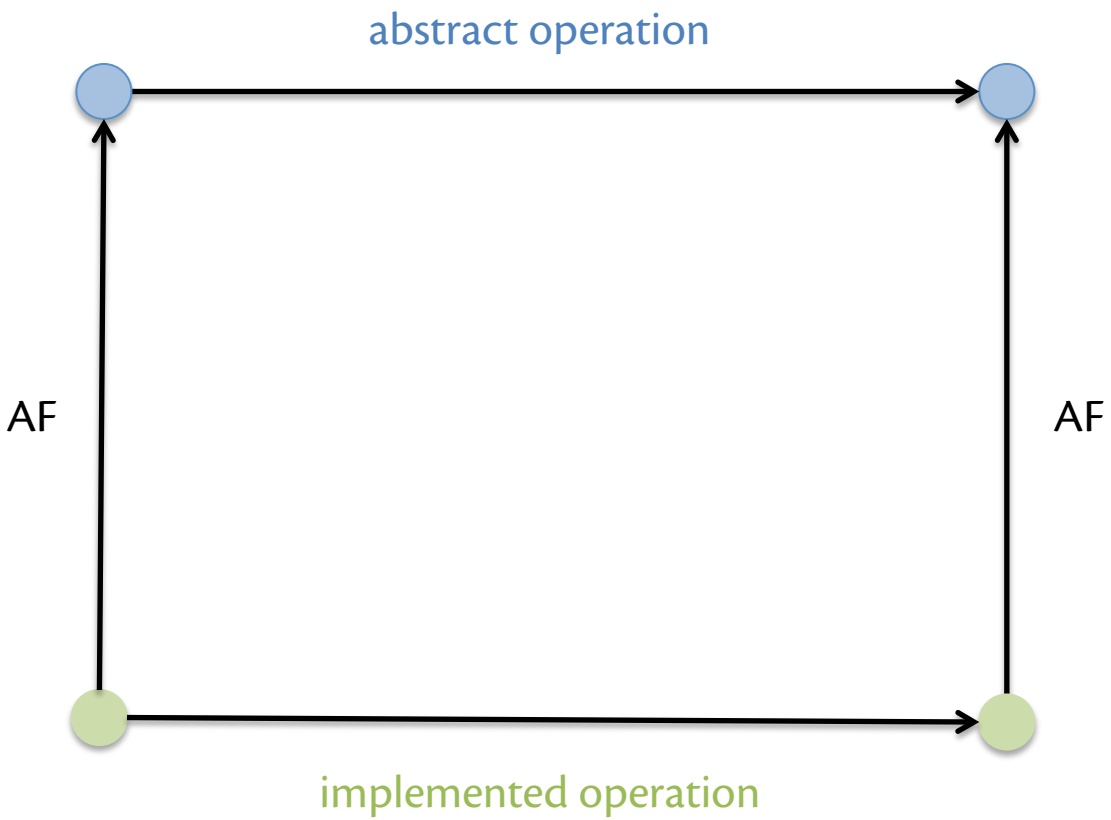
- **Abstraction function** (AF) captures designer's intent in choosing a particular representation of a data abstraction
- Not actually an OCaml function, but a mathematical function
- Maps *concrete values* to *abstract values*



AF properties

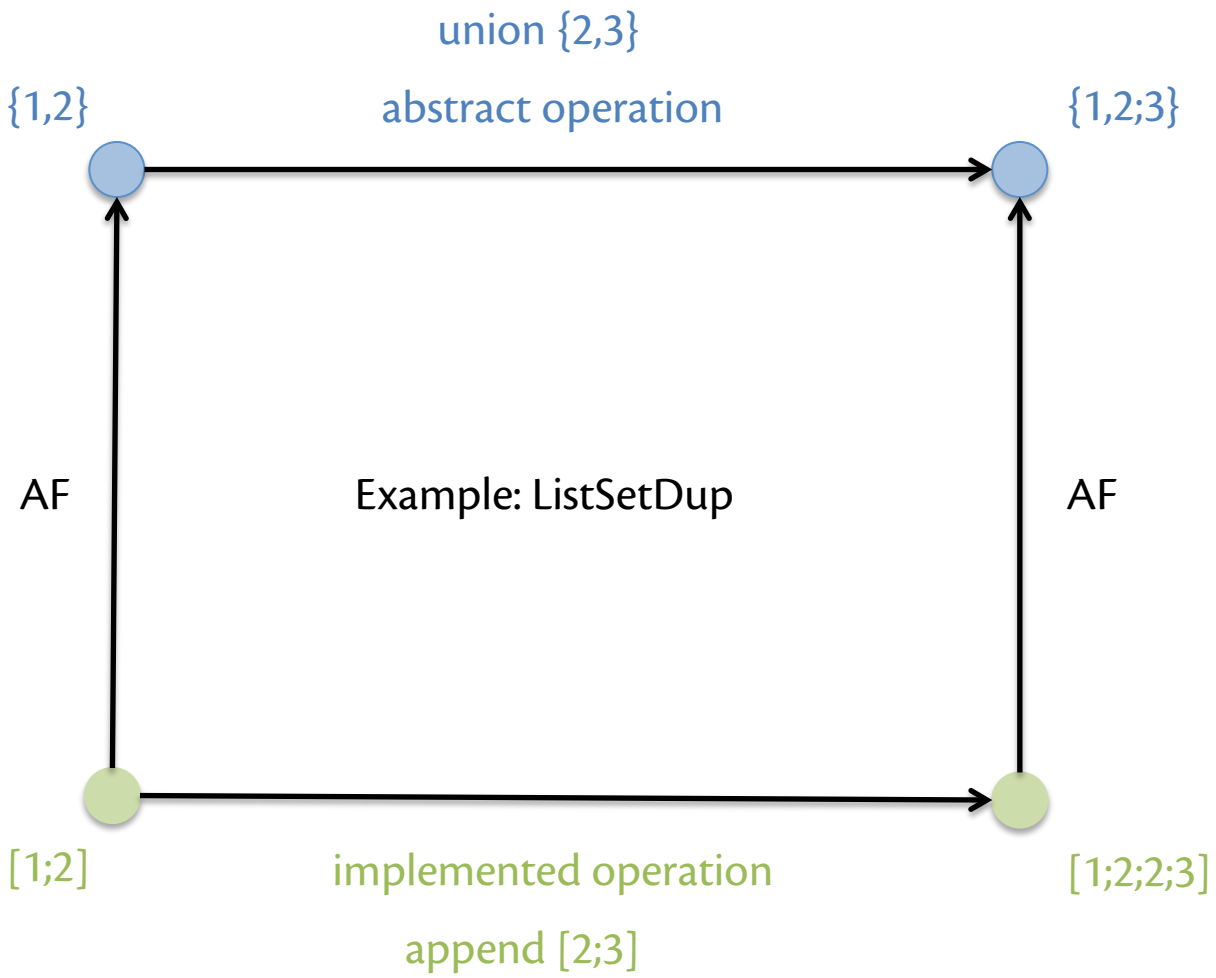
- *Many-to-one*: many values of concrete type can map to same value of abstract type
  - `[1;2]` maps to {1,2}, as does `[2;1]`
- *Partial:* some values of concrete type do not map to any value of abstract type
  - `[1;1;2]` (in no dups) does not map to any set

AF and operations



commutative diagram: both paths lead to the same place

AF and operations



commutative diagram: both paths lead to the same place

Correctness of operations

AF gives us a way to characterize correctness of operation implementation:

$$op_A(AF(c)) = AF(op_C(c))$$

"Commutative": AF commutes with op

Documenting AFs

```
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
```

Documenting AFs

- You might write:
  - Abstraction Function: *comment*
  - AF: *comment*
  - *comment*
- You write it FIRST
  - It's the number one decision you have to make while implementing a data abstraction
  - It dictates what fields are necessary in an object, or what values are necessary in a module
  - It gives meaning to representation
- (A large part of quadtree problem writeup on PS3 is explaining the AF! ...how to interpret regions, quadrants, etc.)

Question #3

Which of the following are part of the AF for quadtrees?

A. A quadtree is a data structure that provides a sparse representation of 2D space.
B. The first quadtree in a Node is the north-eastern quadrant (I) of the region, ..., and the fourth is the south-eastern quadrant (IV).
C. To find an object near point (x, y), a quadtree is traversed starting from the root, walking down the appropriate sequence of child nodes that contain the point until a leaf node is reached.
D. A Leaf is never separated if doing so would cause the size of the region to become too small.

Question #3

Which of the following are part of the AF for quadtrees?

A. A quadtree is a data structure that provides a sparse representation of 2D space.

B. The first quadtree in a Node is the north-eastern quadrant (I) of the region, ..., and the fourth is the south-eastern quadrant (IV).

C. To find an object near point (x, y), a quadtree is traversed starting from the root, walking down the appropriate sequence of child nodes that contain the point until a leaf node is reached.

D. A Leaf is never separated if doing so would cause the size of the region to become too small.

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
  - string_of_X
  - toString()
  - Matrix.show
  - ...all really useful for debugging
    - example: crypto library

Representation invariant

- Recall: AF may be partial
  - [1;1;2] is not a valid ListSetNoDup
  - You might have decided some tuples don't represent valid quadtrees in your implementation
- **Representation invariant** characterizes which concrete values are *valid* and which are *invalid*
  - "Rep invariant" or RI for short
  - Valid concrete values will be mapped by AF to abstract values
  - Invalid concrete value will not be mapped by AF to abstract values
    - CANNOT meaningfully apply AF to values that don't satisfy RI
  - Many Piazza questions about quadtrees on PS3 are about RI

    ...which we mostly let you define ☺

Representation invariant

Operations of data abstraction...

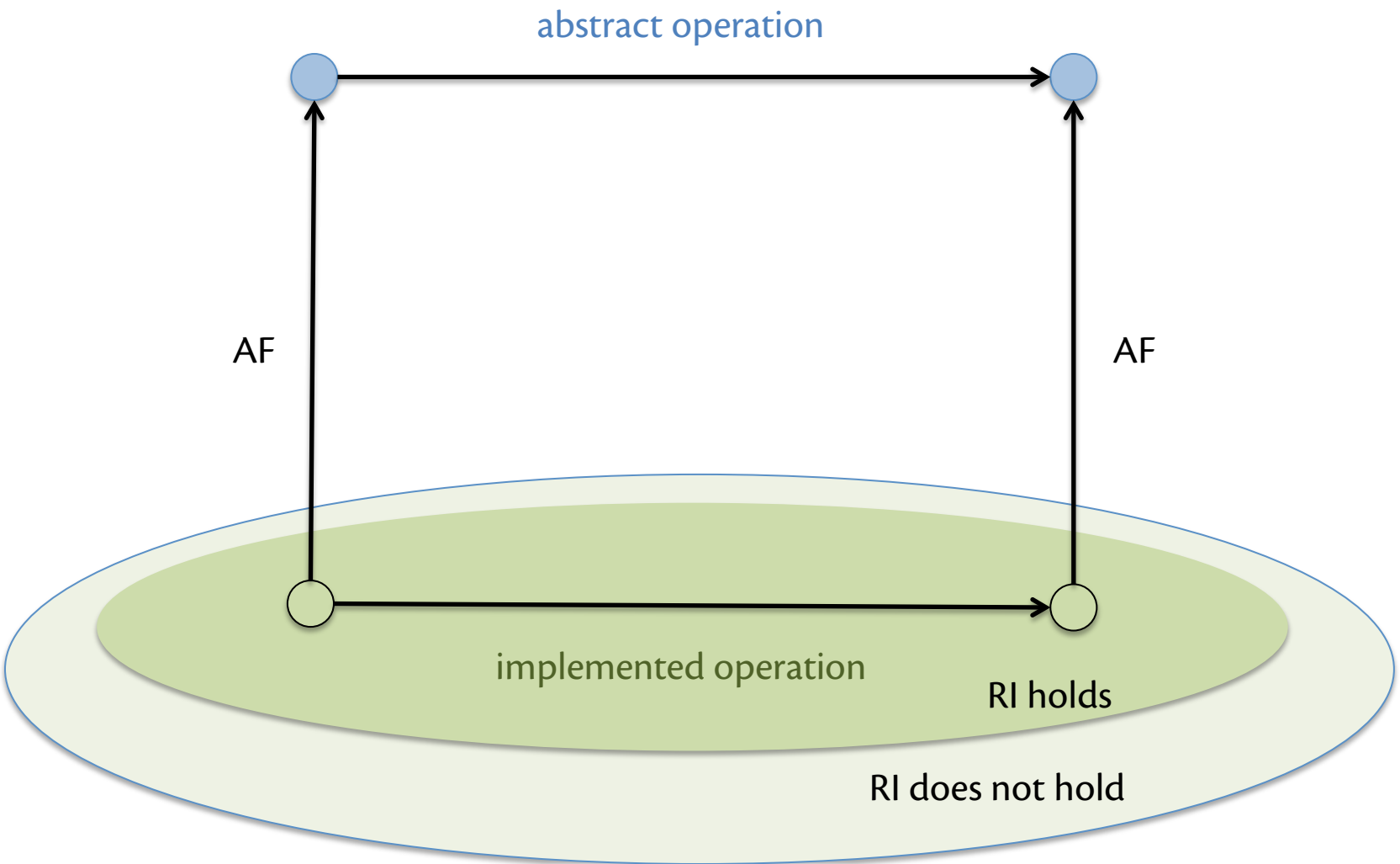
- Assume that any inputs from client satisfy RI
  - e.g., **`ListSetNoDup`** operations assume that values passed in contain no duplicates
- Internally might produce intermediate values that violate RI
  - e.g., **`ListSetNoDup`** could temporarily be processing a list with duplicates
- Must produce output to client that satisfy RI
  - e.g., **`ListSetNoDup`** operations produce values that contain no duplicates
- Hence RI is a fact whose truth is *invariant* except for exactly the period of time when data abstraction operates on values

Representation invariant

Imagine a different person implements each operation of a data abstraction

- The RI is what they agree on
- Needs to express all the relevant constraints so the people don't have to talk to one another

RI with commutative diagram



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 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
```

Implementing the RI

- Great habit to cultivate
- Implement it EARLY, before any operations are implemented
- Common **idiom**: if RI fails then raise exception, otherwise return concrete value

```
let repOK (x:'a list) : 'a list =
  if has_dups x then raise "RI failure"
  else x
```

- When debugging, check repOK on every input to an operation and on every output
- Effectively happened on PS2:
  - Matrices could be invalid
  - Operations were supposed to raise MatrixFailure
  - So your solutions were essentially checking a repOK!

Checking the RI

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

Funny story...this saved the CS 3110 tournament one year

Checking the RI

- Can be expensive!
- For production code:
  - only check "cheap parts" of RI
  - change repOK to identity function, let compiler optimize call away
    - but keep the real repOK around in a comment!
  - use language feature for condition compilation (or various assertion libraries)

Please hold still for 1 more minute

WRAP-UP FOR TODAY

Prelim 1

- Everything from day 1 through tomorrow's recitation (inclusive) is covered by Prelim 1
- Will post a sample prelim on Piazza
- Will have review session in recitation day before prelim
- Will cancel lecture day of prelim

Upcoming events

- **PS3 due Thursday**

This is invariant.

THIS IS 3110