CS 3110

Abstraction and Specification

Nate Foster

Spring 2020

Today's music: Never Change by JAY-Z

CLICKER QUESTION 1

OCaml

Review

Previously in 3110:

- OCaml module system

Today:

- Abstraction and specification
- Specifying functions
- Specifying data abstractions
- Abstraction functions and representation invariants

Abstraction

(verb: abstract)

Forgetting information, so that different things can be treated as the same

(noun)

Artifacts that result from that process

Specification

(noun)
Intended behavior of abstraction

(verb: specify)
The act of creating such an artifact

Audience of specification

- Clients
  - What they must guarantee (preconditions)
  - What they can assume (postconditions)

- Implementers
  - What they can assume (preconditions)
  - What they must guarantee (postconditions)

Benefits of specifications

- **Locality:** understand abstraction without needing to read implementation
- **Modifiability:** change implementation without breaking client code
- **Accountability:** clarify who is to blame

Specifications are contracts

Satisfaction

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

What if spec is ambiguous?

Ambiguity is a fact of life.

Do the most reasonable thing you can.

Probably not 🔥

Who wrote it?

- **You:** improve it
- **Client:** seek clarification

but if you make 500 requests they probably won't hire you again

SPECIFYING FUNCTIONS

Template

```
(** [f x] is ...
    Example: ...
    Requires: ...
    Raises: ...  *)
val f : t -> u
```

Based on *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.

Data abstraction

- A data abstraction is a specification of operations on a set of values
  - e.g., stacks have push, pop, peek, etc.; we don't know what the values concretely are
- A data structure is an implementation of a data abstraction with a particular representation
  - e.g., `ListStack` implemented `StackSig` with `'a list`, `(::)`, etc.

SPECIFYING DATA ABSTRACTIONS

Demo

CLICKER QUESTION 2

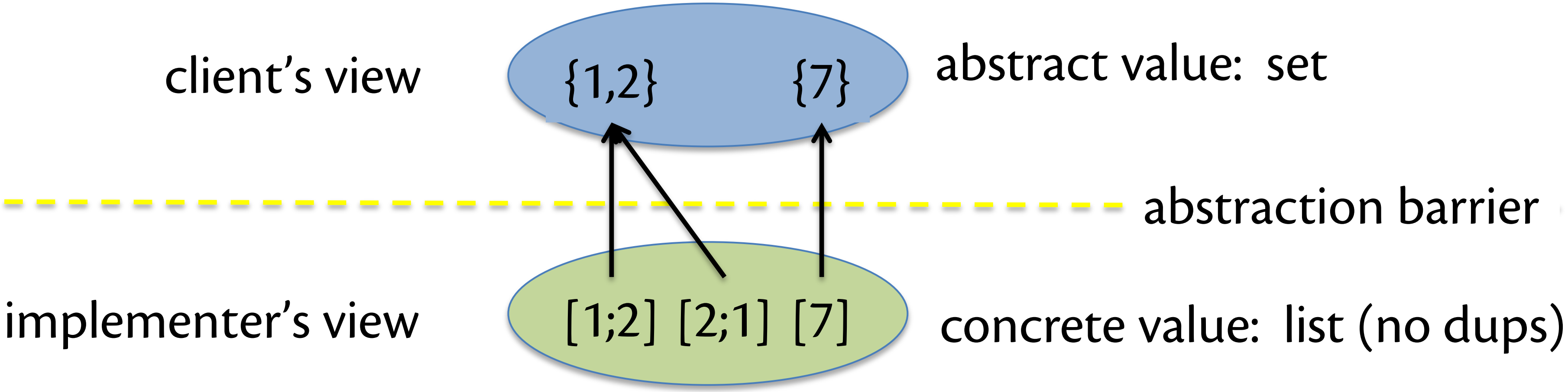
Representation types

- 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

ABSTRACTION FUNCTIONS

Abstraction function



client's view

abstract value: set

abstraction barrier

implementer's view

concrete value: list (no dups)

the black arrows are the abstraction function

Abstraction function

maps

valid concrete values

to

abstract values

Documenting the AF

- Above rep type in implementation you write:

  ```
  (* AF: comment *)
  ```

- Write it first before implementing operations

Demo

Implementing the AF

- `to_string`
- `format`
  (see abstract types section in textbook)

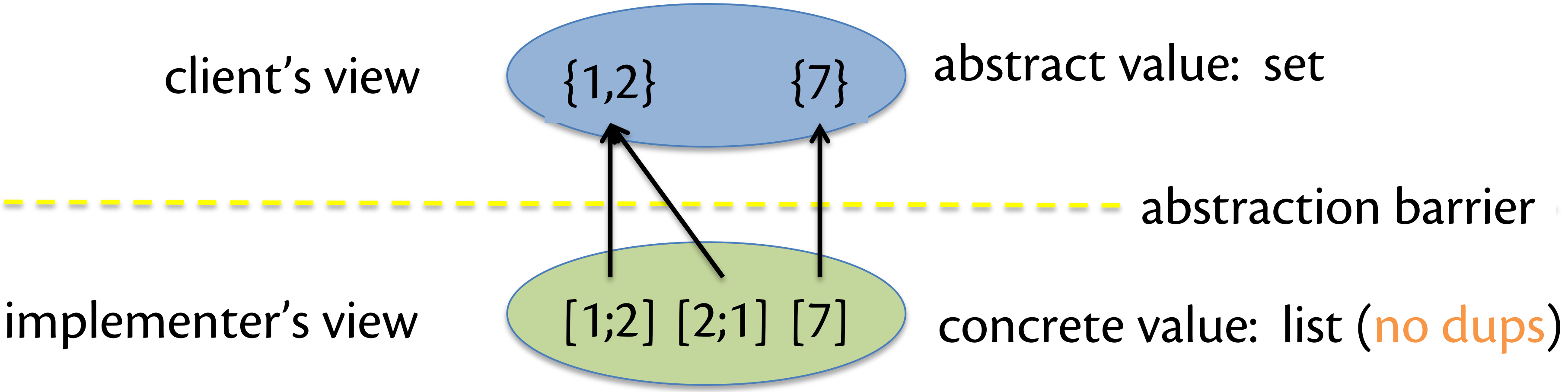
Representation types

- 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

REPRESENTATION INVARIANTS

Abstraction function



client's view

{1,2}

{7}

abstract value: set

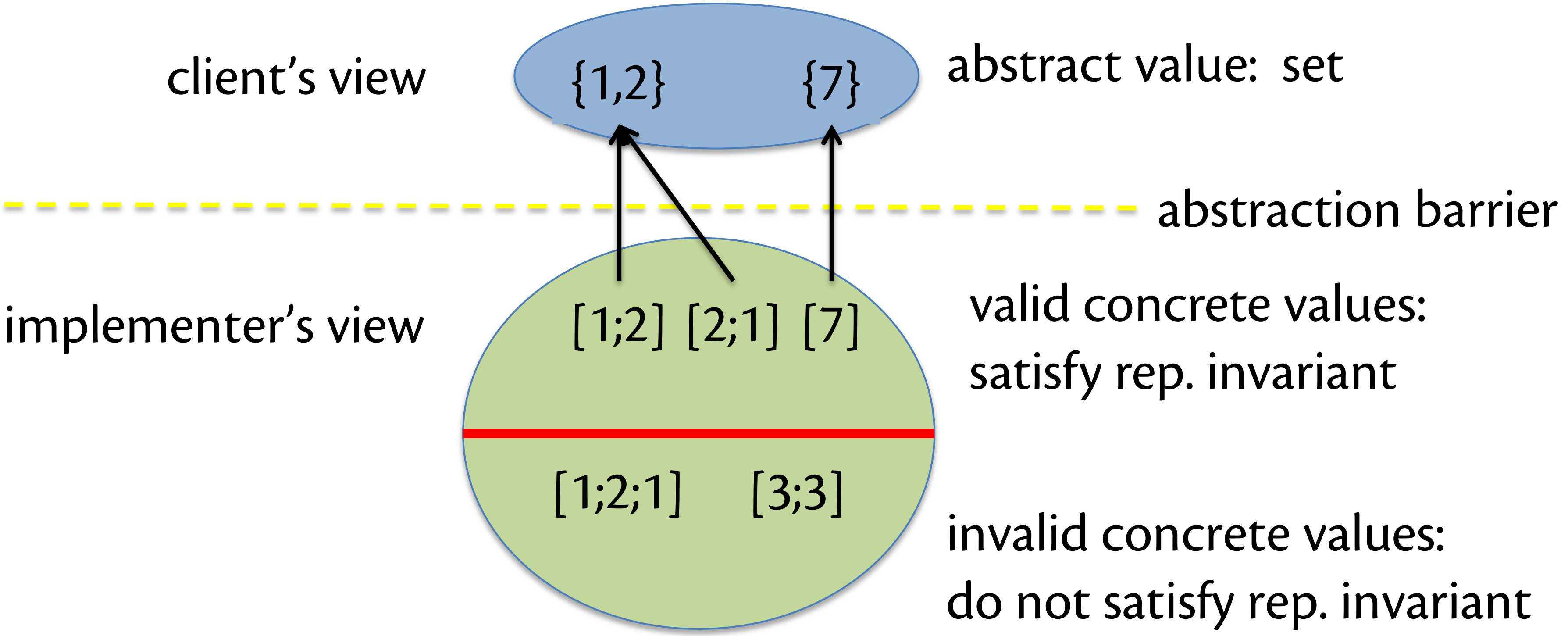
abstraction barrier

implementer's view

[1;2] [2;1] [7]

concrete value: list (no dups)

Representation invariant



client's view

{1,2} {7}

abstract value: set

abstraction barrier

implementer's view

[1;2] [2;1] [7]

valid concrete values:
satisfy rep. invariant

[1;2;1] [3;3]

invalid concrete values:
do not satisfy rep. invariant

the thick red line is the rep. invariant

Rep. invariant

distinguishes

valid concrete values

from

invalid concrete values

Documenting the RI

- Above rep type in implementation you write:

  ```
  (* RI: comment *)
  ```

- Write it first before implementing operations

Demo

Rep. invariant

implicitly part of
every precondition and
every postcondition
in abstraction

Demo

Implementing the RI

Idiom:

- write `rep_ok` function
- call function on every input and output

```
let rep_ok (x : t) : t =
  if (* check RI *) then t
  else failwith “RI”
```

This saved a 3110 final project game tournament one year!

Upcoming events

- [tomorrow night] A2 due
- [Thursday] nothing out!

This is invariant.

THIS IS 3110