3110: Modules

Clarity

- For functional programming, we are only responsible to the existing data structures to make sure they are unchanged by function executions.

Homework Solution - Stacks

(note these solutions, show the core and do not include index checking, add a bunch of checking that next is not None, where appropriate)

```
def Push(value, top_frame):
    return Frame(value=value, next=top_frame)
```

```
def Pop(top_frame):
    return top_frame.next
```

Homework Solution – Linked Lists

```
Def Remove(head, index):
    if index == 1:
        return head.next.next

    new = head.copy()
    new.next = remove(head.next, index-1)
    return new
```

Homework Solutions – Linked Lists

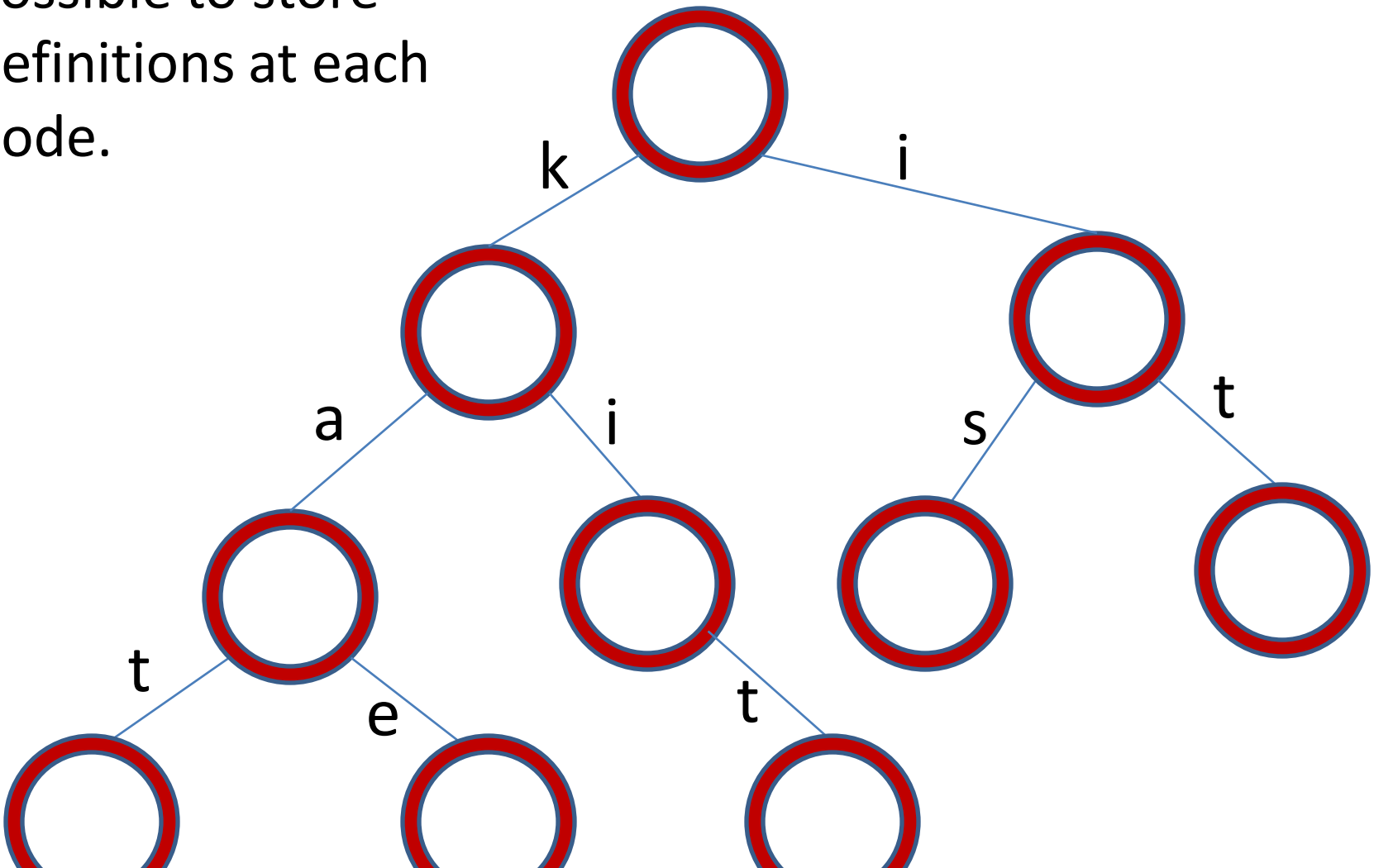
```
def Add(head, index, value):
    new = head.copy()

    if index == 0:
        new.next = Frame(value=value, next=head.next)
    else:
        new.next = Add(head.next, index-1, value)

    return new
```

Tries / Prefix Trees

- The edges from a parent to a child now represent something.
- Consider edges named, and an ordered traversal through the tree, collecting edge names, results in constructing a key.
- The value of that key is then stored in the node.

Trie Ex.

Possible to store definitions at each node.



Project

- You can run with creating and using Trie.

- Instructions for Red-Black Trees to come, you will be asked to implement the add operation for Red-Black Trees.

Signatures

- So far we've been dealing with *interfaces*, a contract between a module and a user.

- *Signatures* of functions define valid parameter types and return types. Python does not have a built in way to do this.

- We'll see how to handle signatures in Python for the purpose of debugging.

Module Design

- We've seen it should be a way to divide the problem into manageable sub-problems.

- Can be designed for ease of implementation, safety, security, etc
  - Designing Power Plant Modules
  - Designing Secure Server Modules

Module Design

- Can be well thought out before you code, but is not necessarily set in stone.

- As you get further into an implementation you observe sub-modules forming. Sometimes not a bad idea to pull them out.

- Ex. In scientific computing, may notice several modules building the same function, best to then pull that function out. Better for debugging and perhaps future programming. This is how personal libraries get started.

Interface

- Is a contract and should be adhered to.
- Not a copying of available function's signatures.
  - Incomprehensible to users
  - Doesn't allow for change of implementation
- Vice Versa, function's signatures should not be a copy of the interface.

Abstraction

- Abstraction is “selective ignorance”
  - Useful in that it allows us to sit back and look at larger pictures rather than insist on detailed knowledge of every line of code.

- Two abstraction mechanisms:
  - **Abstraction by parameterization**
  - **Abstraction by specification**

- 4 kinds of abstractions:
  - **Function abstraction**
  - **Data abstraction**
  - Iteration abstraction
  - Type hierarchy
- [Slide from D. Hou]

Abstraction by Parameterization

- The specification of what computation is occurring is abstracted under types.

- Ex. For Push, the input -> output can be viewed as:

Frame, object -> Frame

Again, Python doesn't check this for us. We'll see how to check it ourselves.

Abstraction by Specification

- This is the intent of the code and the contract of what it will do.

- For Push(old_top_frame, new_top_frame):
  - Requires: new_top_frame to have a next attribute
  - Effects: adds new_top_frame to the top of the stack

Abstraction by Specification

- Seen this before?

```
"""
LinearMove:
    Takes:
        start        - (x_0,y_0) initial position
        finish       - (x_n, y_n) final position
        n_frames - int, number of frames
       Returns:
        a list  of [(x_0, y_0),(x_1, y_1), ..., (x_n, y_n)]
    Smoothly moves a point from start to finish in n_frames.
"""“
```

- Sometimes (Takes/Returns) is (Requires/Effects)

Data Abstraction

- Powerful, let's look at an example Stack:

Stack:

Performs a last in first out queue.

Push: stores a given object and returns a new stack

Pop: returns the most recently Pushed object and a new stack

- Minimal amount of detail necessary. Don't have to know what the object is. Don't have to know how the Stack is implemented.

Want to test modules

- Understand that implementations can be hidden under abstraction, let's take another look at that.

- *Local reasoning* once a function has a specification, we can go ahead and debug that function on it's own. We don't have to worry about the rest of the program, and once debugged the rest of the program doesn't have to worry about the function.

- The rest of this lecture follows very closely the spring lecture notes for lecture 8:

- http://www.cs.cornell.edu/courses/cs3110/2010sp/lectures/lec08.html

A series of examples

- Given the specification that a set is a list of elements and union returns a new list
- In Python could do:

  list_1 + list_2

  But this would cause problems for other functions, especially size of set, because it allows duplicate members.

Try 2

- Give that a set is a list with no duplicates

- In Python:

  list(unique(list_1+list_2))

  Or

  Result = list_1

  Result = Result + filter(lambda a: a not in result, list_2)

Can do better

- Recall using tree structure

So specifications

- The looser/more general the specification the harder the implementation, the easier for the user.
- The tighter/more specific the specification the easier the implementation, the harder for the user.
- We'll add more details to that.

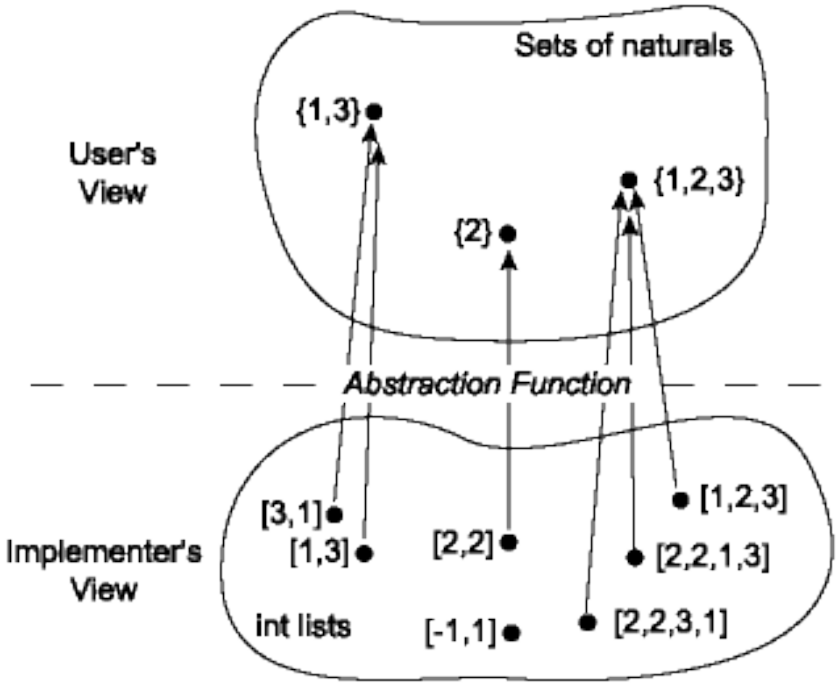
Abstraction Functions

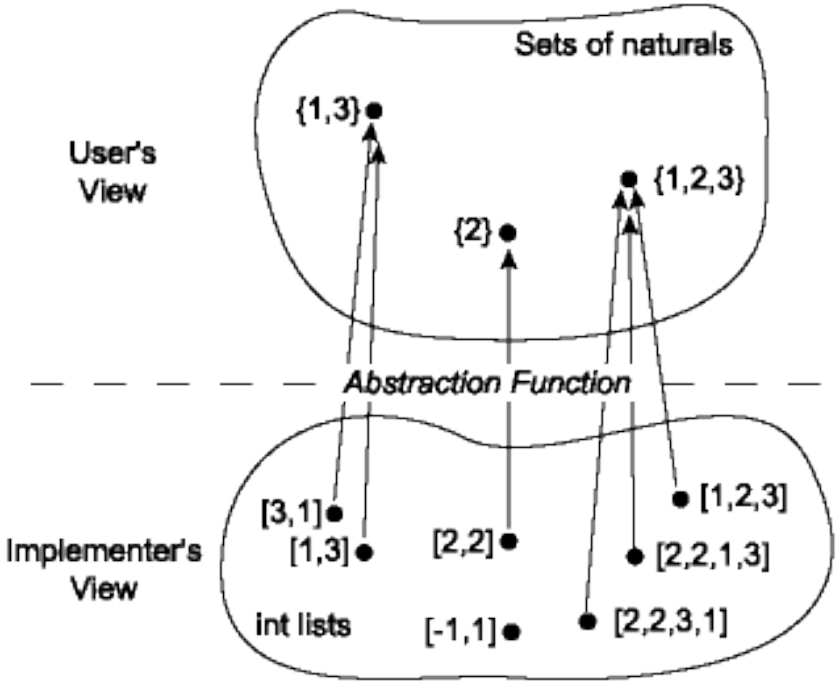
- Let's abstract sets as lists:
- Consider lists/sets:

  [3, 1], [1, 3], [1, 1, 3]

- To implementer, the lists are different (recall union for lists with repetition)
- To user, they all represent the set {1, 3}

- The loss of information between the concrete data for the implementer to the abstract data for the user ([1, 1, 3] -> {1, 3}) is described by the *abstraction function*.





- Note, function is not 1-1 and the inverse is not onto (in math terms)

- Function is *many-to-one* (consider [1, 1] and [1]), going forward and going backward the function is *partial* (consider [‘cat’, ‘hat’]).

Comments

- Hence it is useful to add the abstraction function to the comments.

- For the specification that a list may contain duplicates the comment:

“””

Abstraction function: the list [a1;...;an] represents the smallest set containing all of a1;...;an. The list may contain duplicates. The empty list represents the empty set.

“””

- For the specification that a list may not contain duplicates:

“””

Abstraction function:

the list [a1;...;an] represents the set

{a1;...;an}. [] represents the empty set.

“””

Add this to all comments?

- You should practice it.

- In more advanced code will see it mentioned when the function is not obvious or when it provides clarity.

Commutative Diagrams

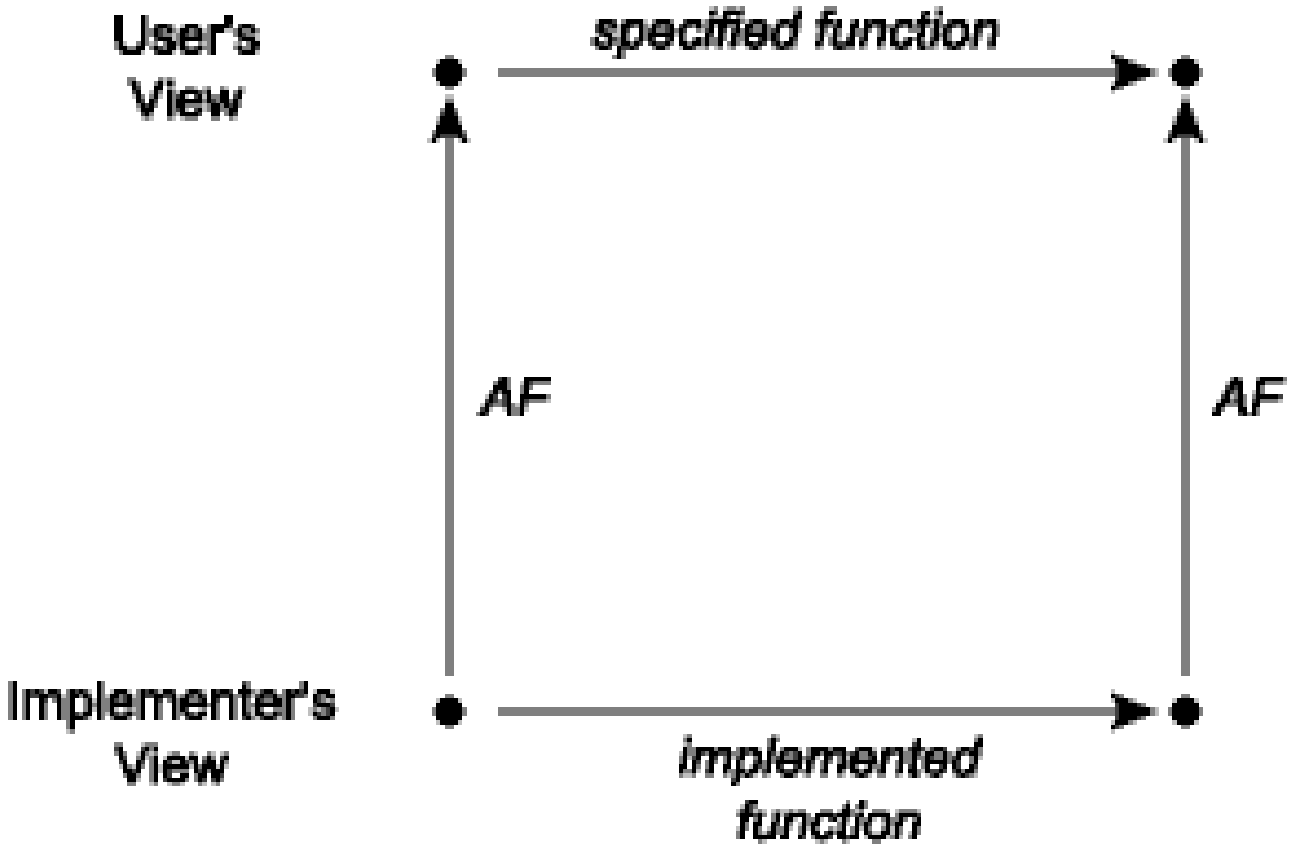
- We can start to ask if a program is correct.
- We can use the abstraction to reason about what the program is performing.
- An example:

Consider Union

- In particular the implementation with duplicates
- Let list_1 = [1, 3] and list_2 = [2, 2]
- The implementation will produce [1,3, 2, 2], which through the abstraction function is {1, 2, 3}

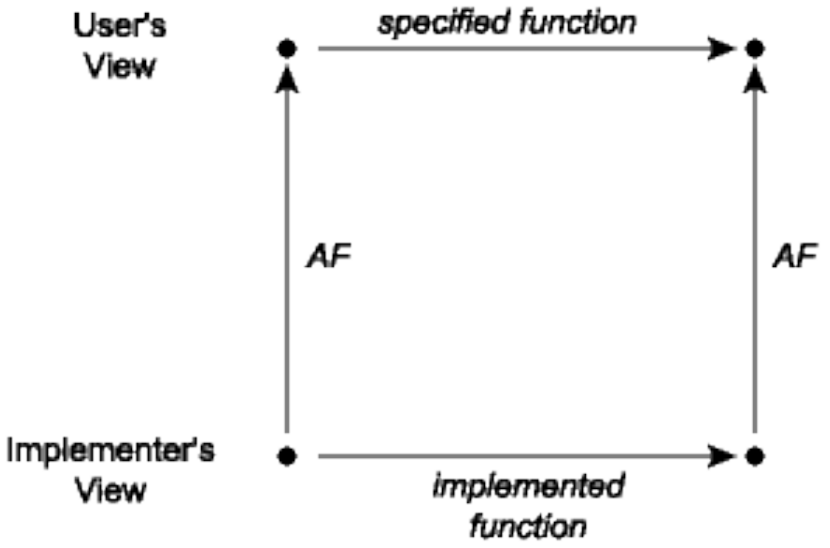
- Alternatively, through the abstraction function list_1 = {1, 3} and list_2 = {2}, applying union's specification we get {1, 2, 3}
- The Same!!

- A picture view of the check, through a commutative diagram:



Not the end all

- Given a list with duplicates, consider calculating the minimum representation of the set, the abstraction function gets rid of the information about duplicates. Ex [1, 1, 3, 3]



Representation Invariants

- Abstraction functions by themselves are not enough.
- Need to be coupled with *representation invariant/rep invariante/RI* this function “defines what concrete data values are valid representations of abstract values”

- Ex.:

“””

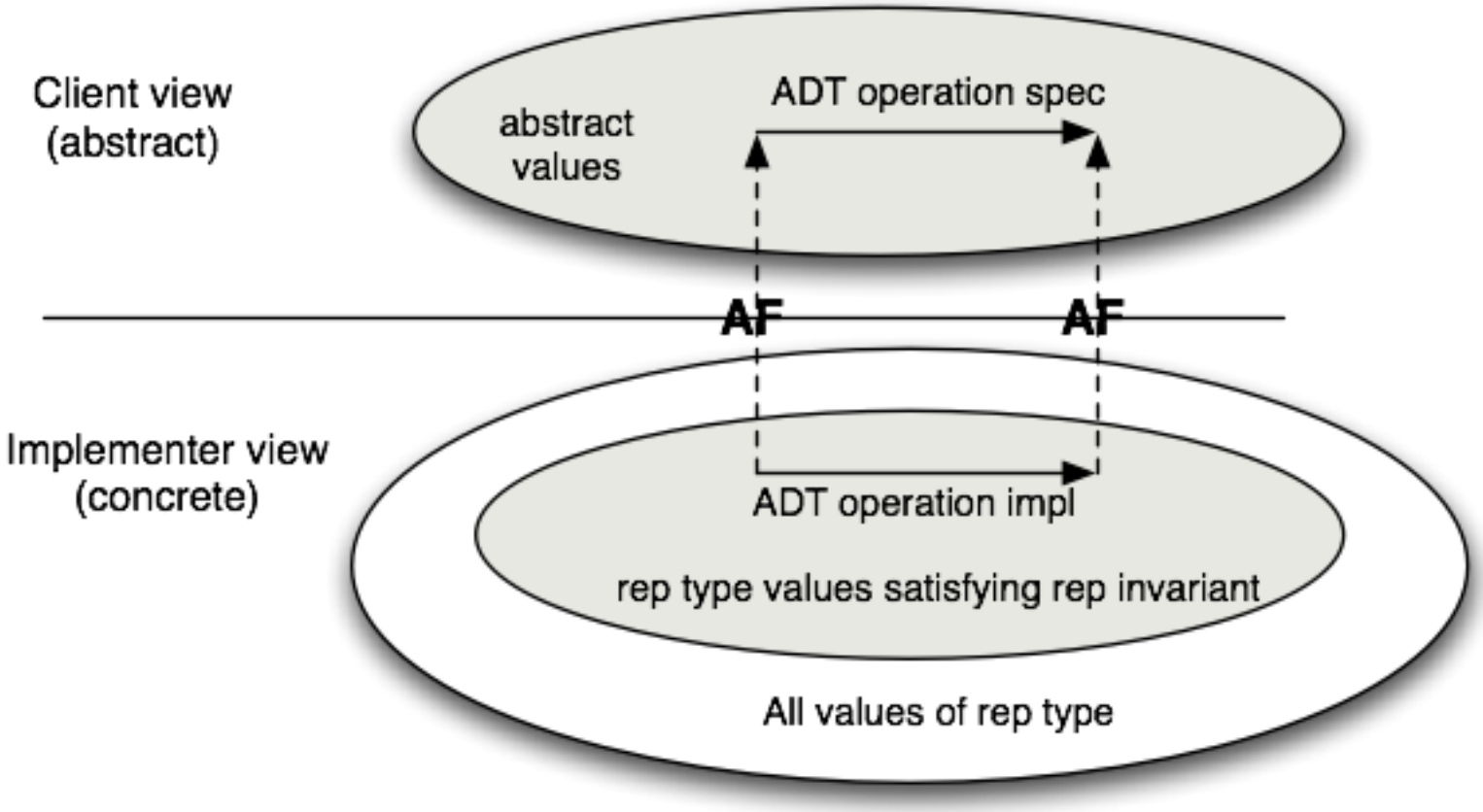
Representation invariant: the list contains no duplicate elements.

“””

Representation Invariant & Abstraction Function

- We have seen that the abstraction function may be a *partial* function. If the implementation produced a value that did not map to an abstract function, we couldn't reason about out program.
- So the RI works with the Abstraction function to make sure the implementation never produces such values.

All together



- The RI must hold otherwise functions that rely on it will break.

- To make sure this holds we write a repOK function to check that the RI is upheld.

repOk

- Checks that the RI is upheld after each function returns.

- Returns identity(same output) if it is and throws an exception otherwise

Production Code

- repOK can be expensive to compute:
- Best use for debugging and leave in comments for debugging future modifications if expensive.

- If cheap, may still be useful in production code.

Principles for Modular Design

Issue	Loose coupling	Tight coupling
Size of interface	*narrow* interface: few operations	*wide* interface: many operations
Complexity	Simple specifications	Complex specifications
Invariants	Local	Global
Pre/post-conditions	Weak, nondeterministic	Strong, deterministic
Correctness	Easier to get right	Harder to get right
Performance	May sacrifice performance	May expose optimizations

Module Invariants