

Efficient and Precise Typestate Analysis by Determining Continuation-Equivalent States

Eric Bodden



TECHNISCHE
UNIVERSITÄT
DARMSTADT

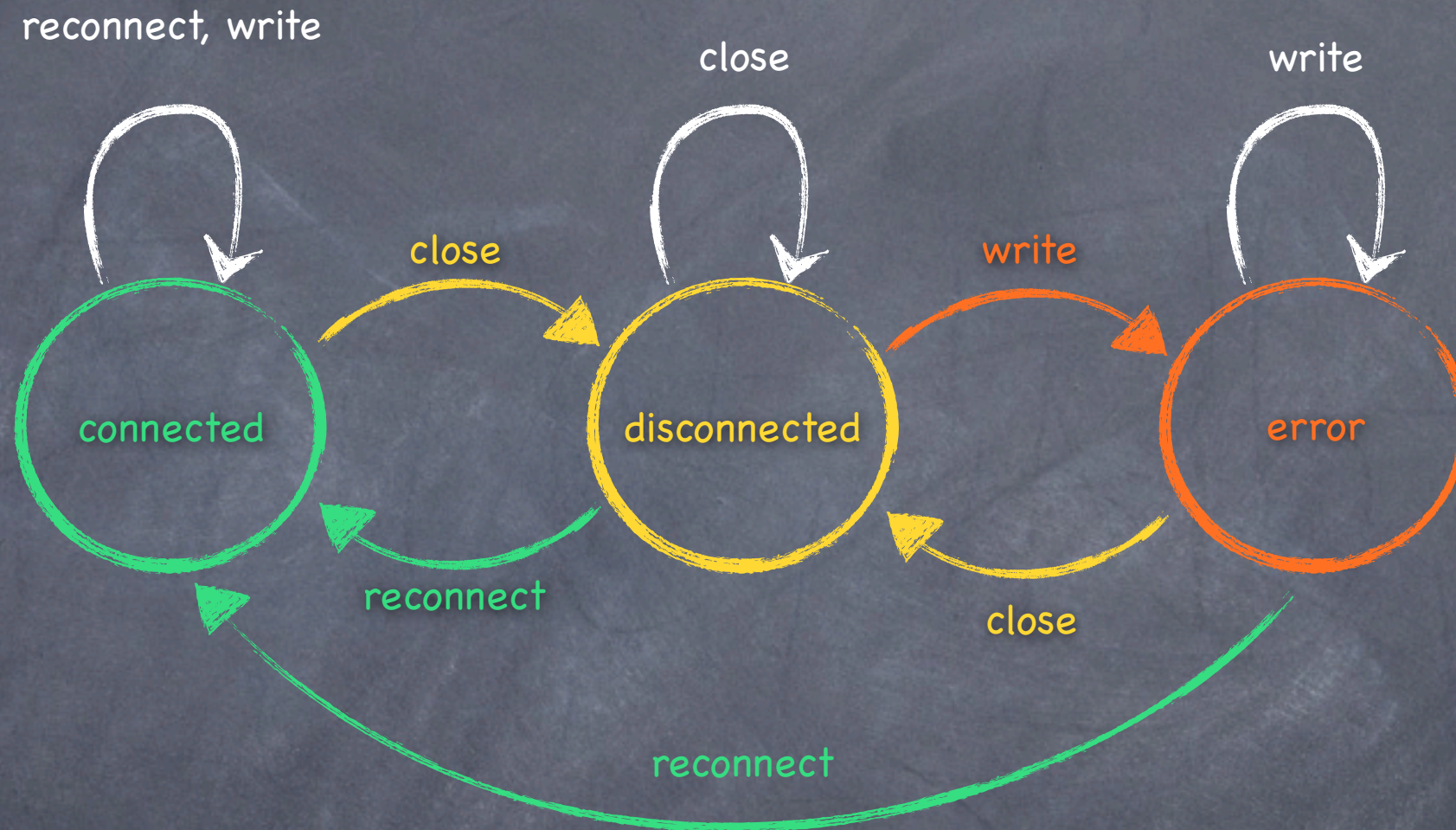


CASED

Finite-state properties

“After closing a connection c ,
don't write to c until c is reconnected.”

Finite-state properties



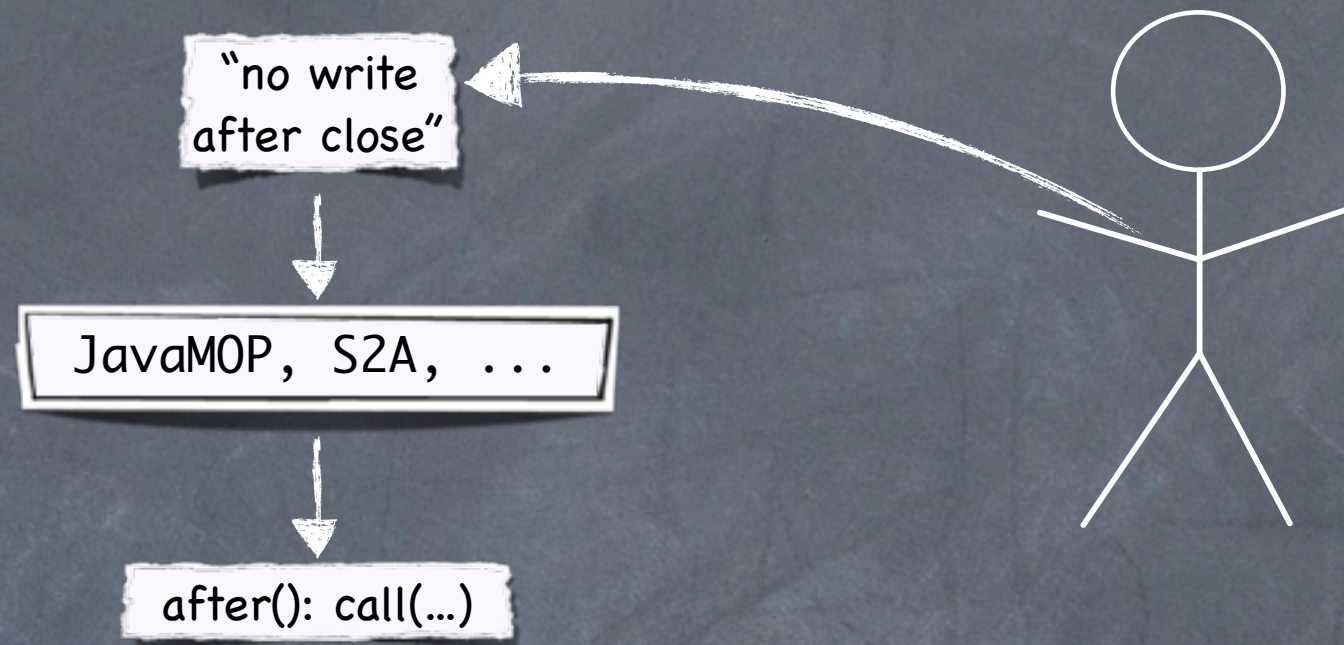
“After closing a connection *c*,
don’t write to *c* until *c* is reconnected.”

Runtime-verifying finite-state properties

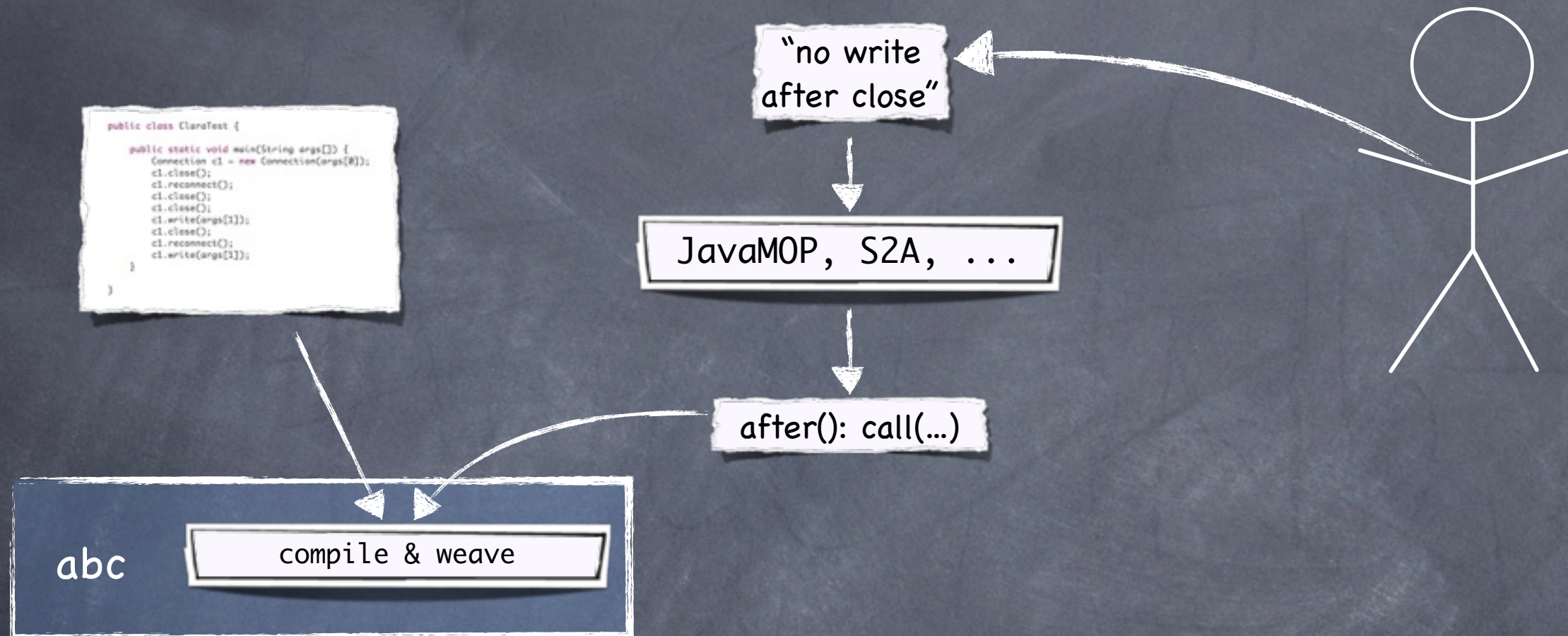
Runtime-verifying finite-state properties



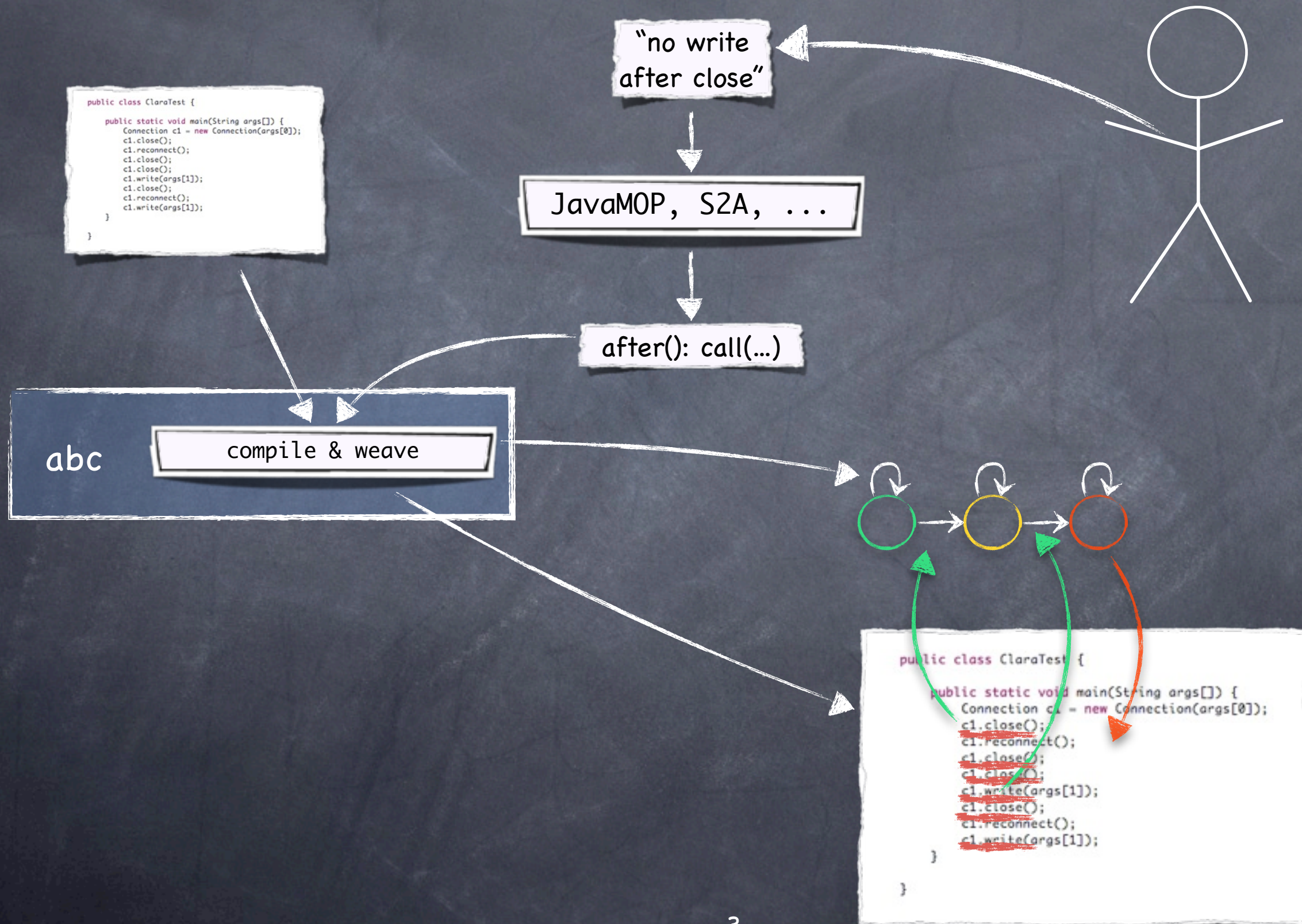
Runtime-verifying finite-state properties



Runtime-verifying finite-state properties



Runtime-verifying finite-state properties



Runtime-verifying finite-state properties

Runtime-verifying finite-state properties



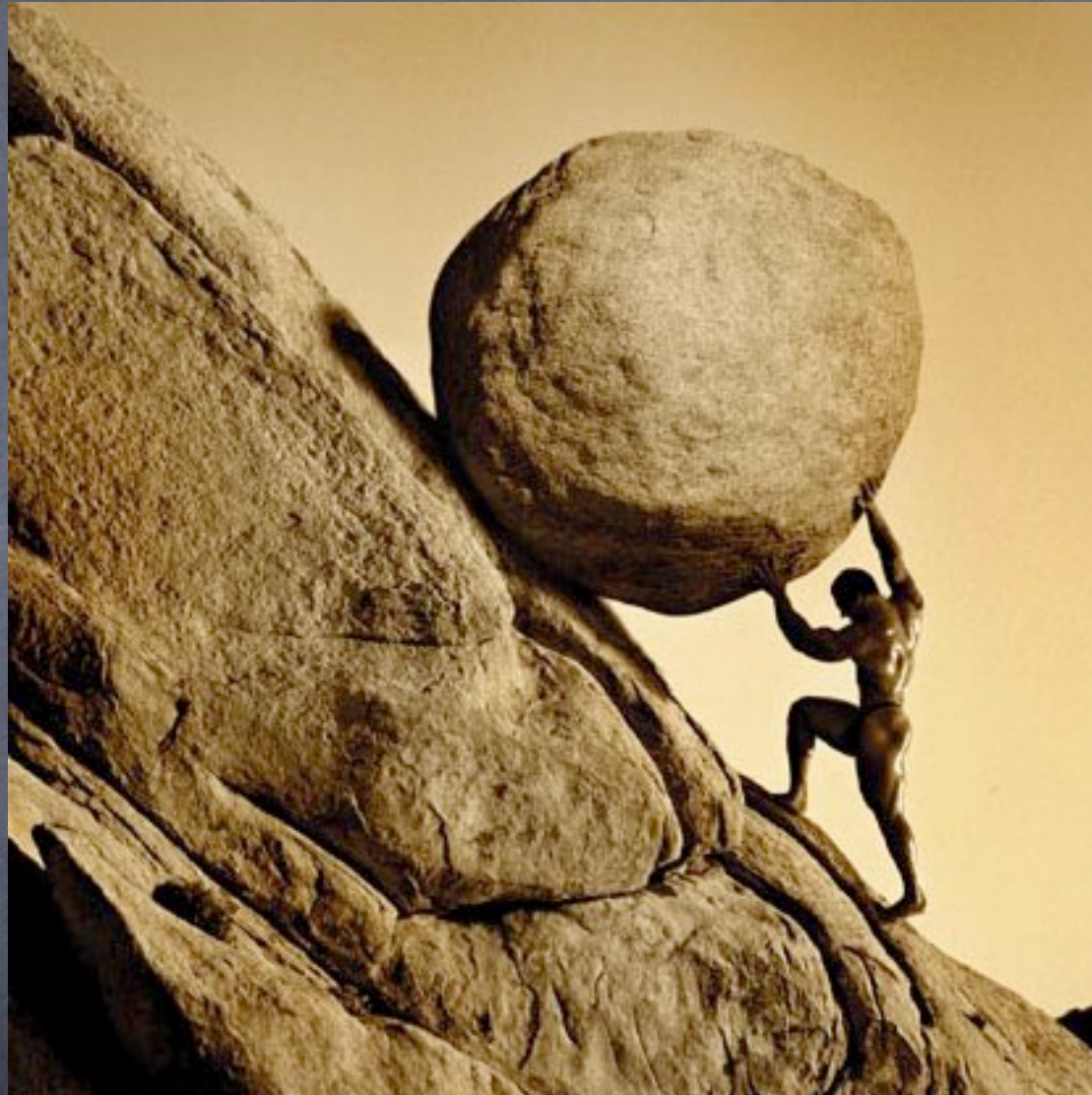
No static guarantees

Runtime-verifying finite-state properties



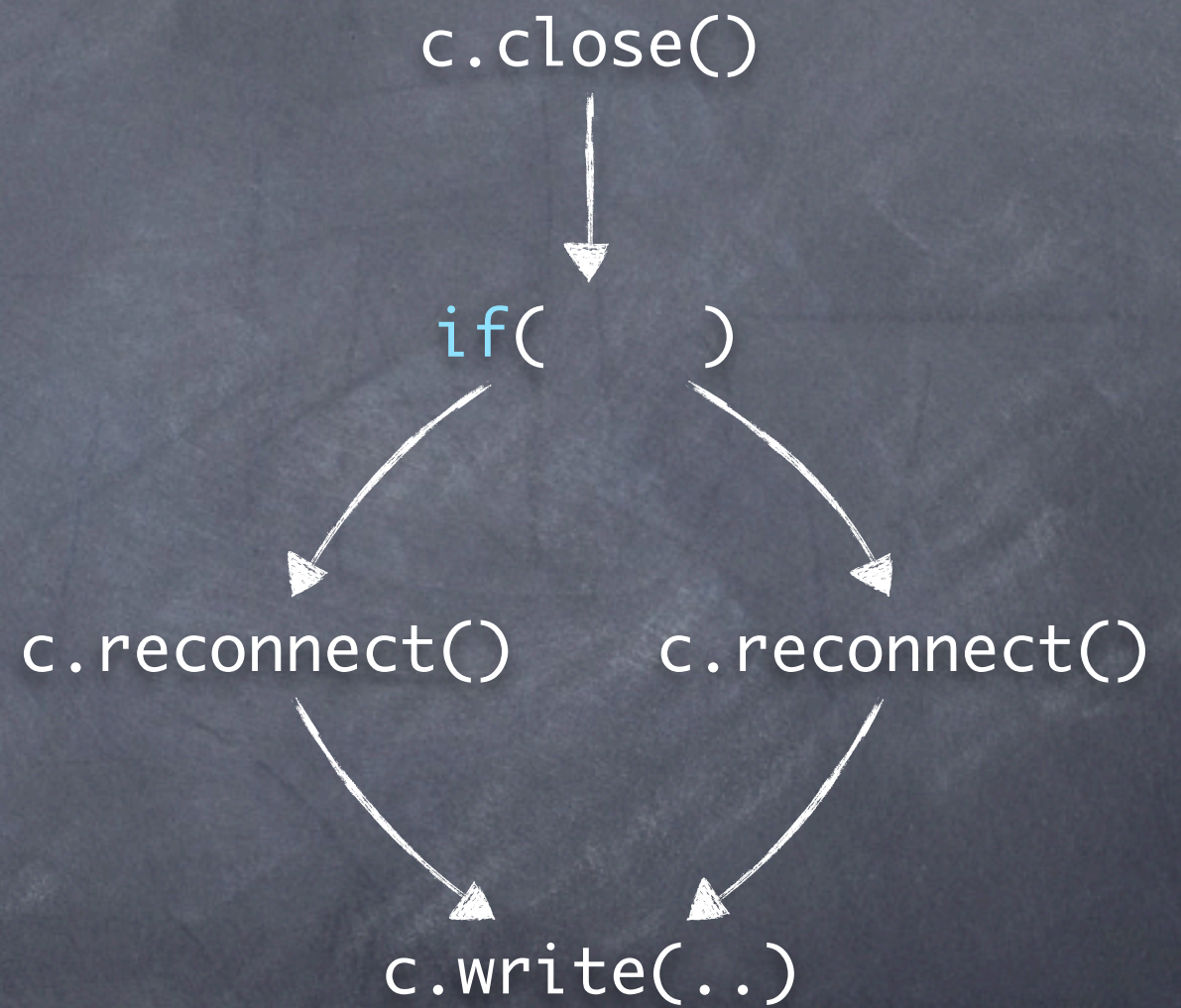
Potentially large runtime overhead

Runtime-verifying finite-state properties

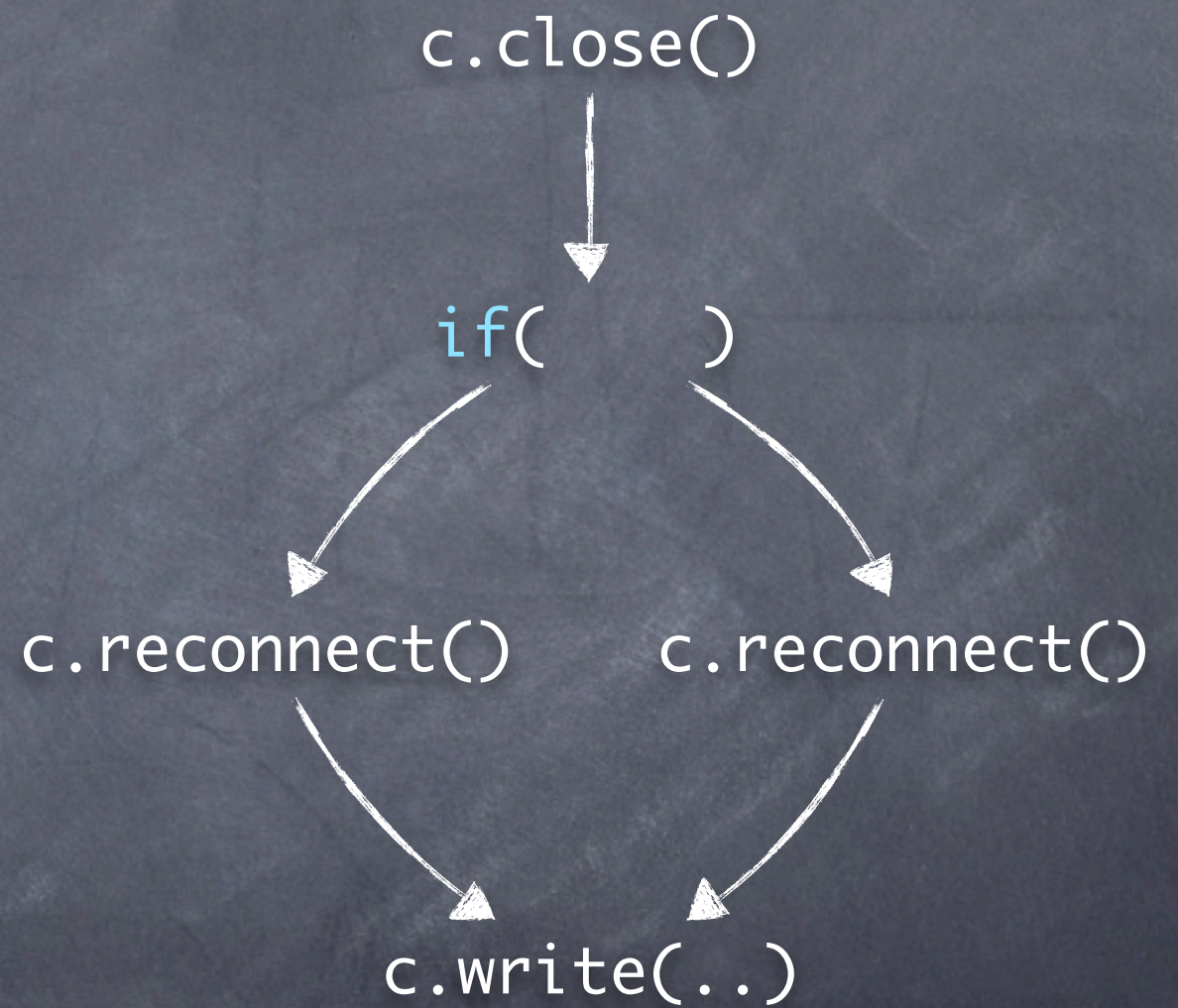
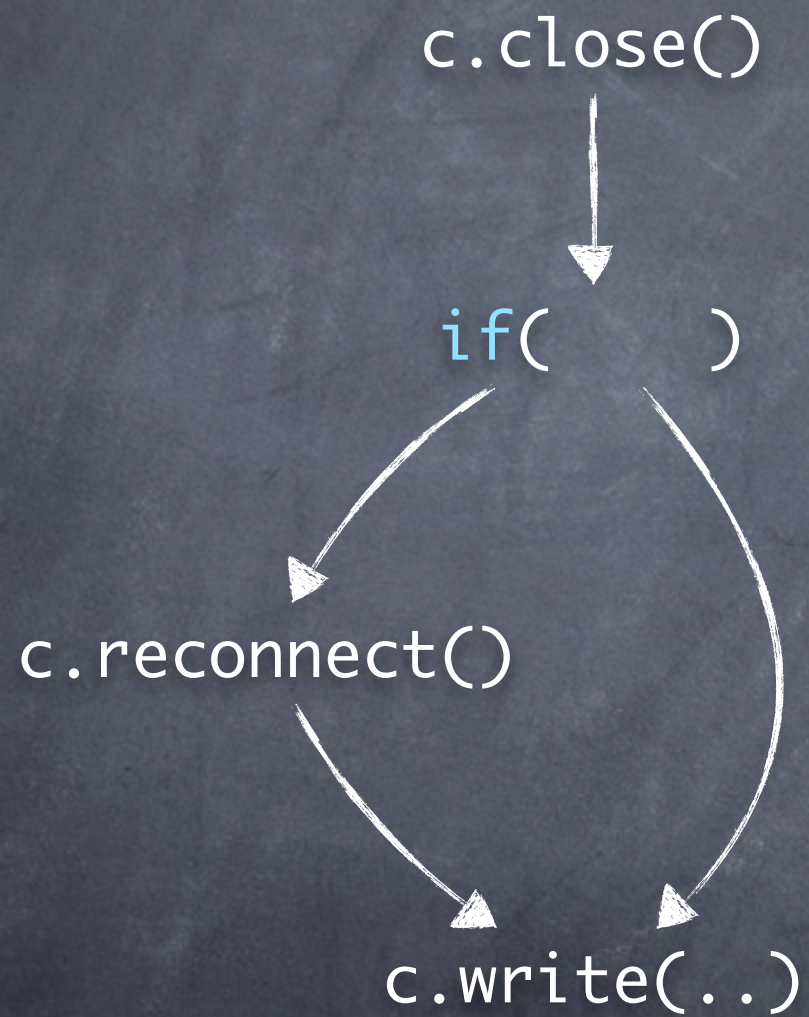


When to finish testing?

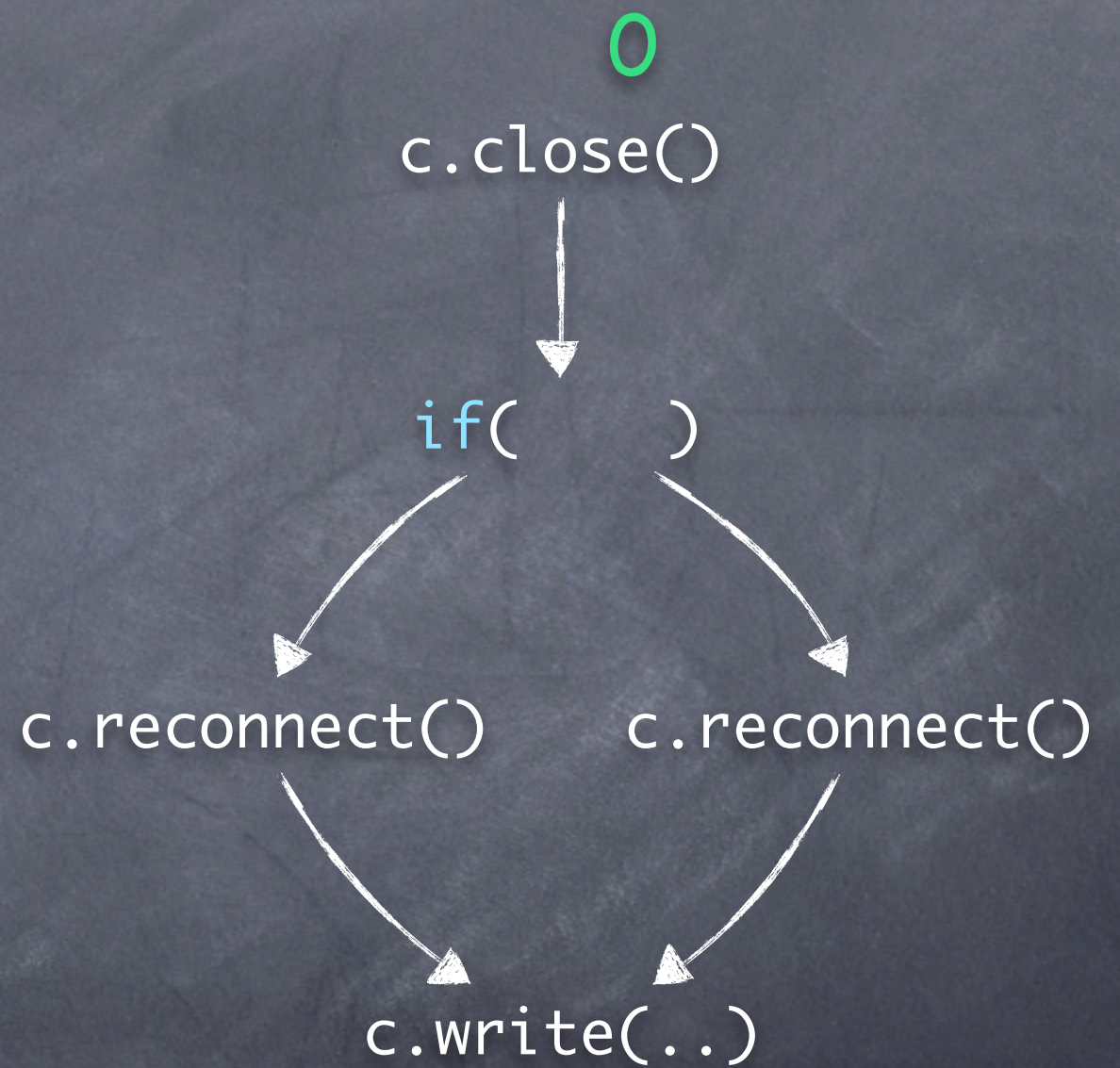
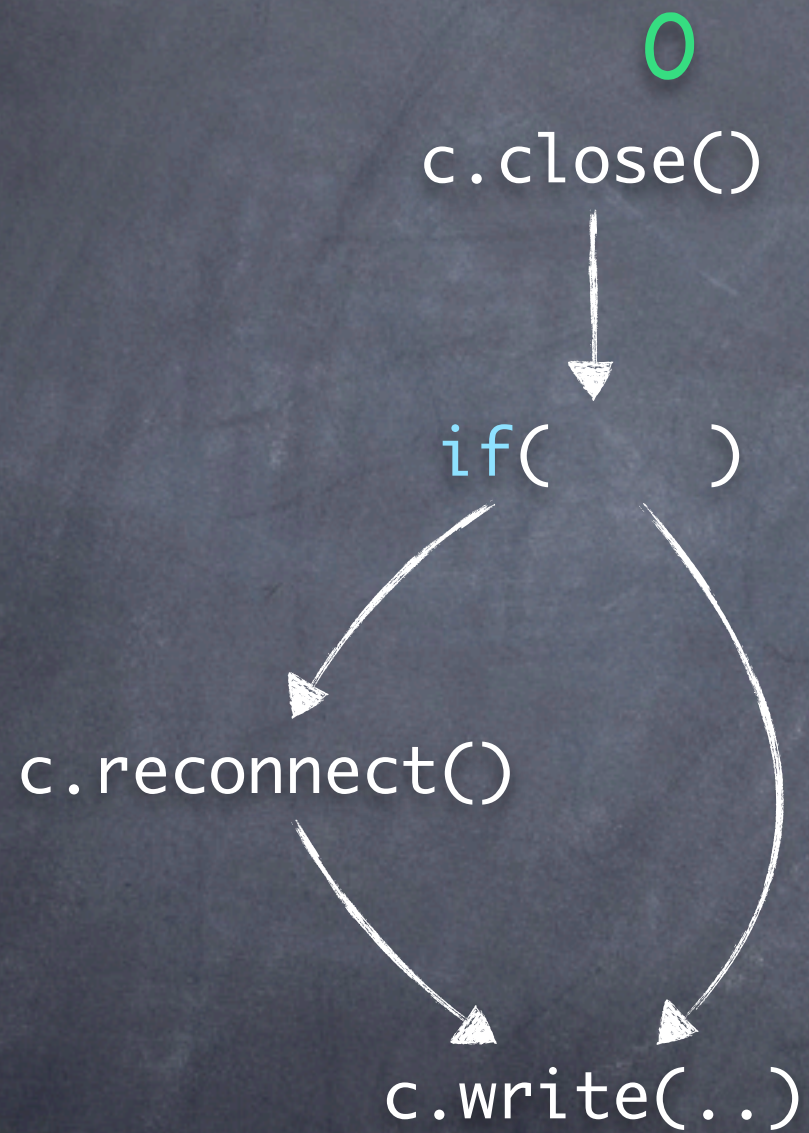
Pure static analysis



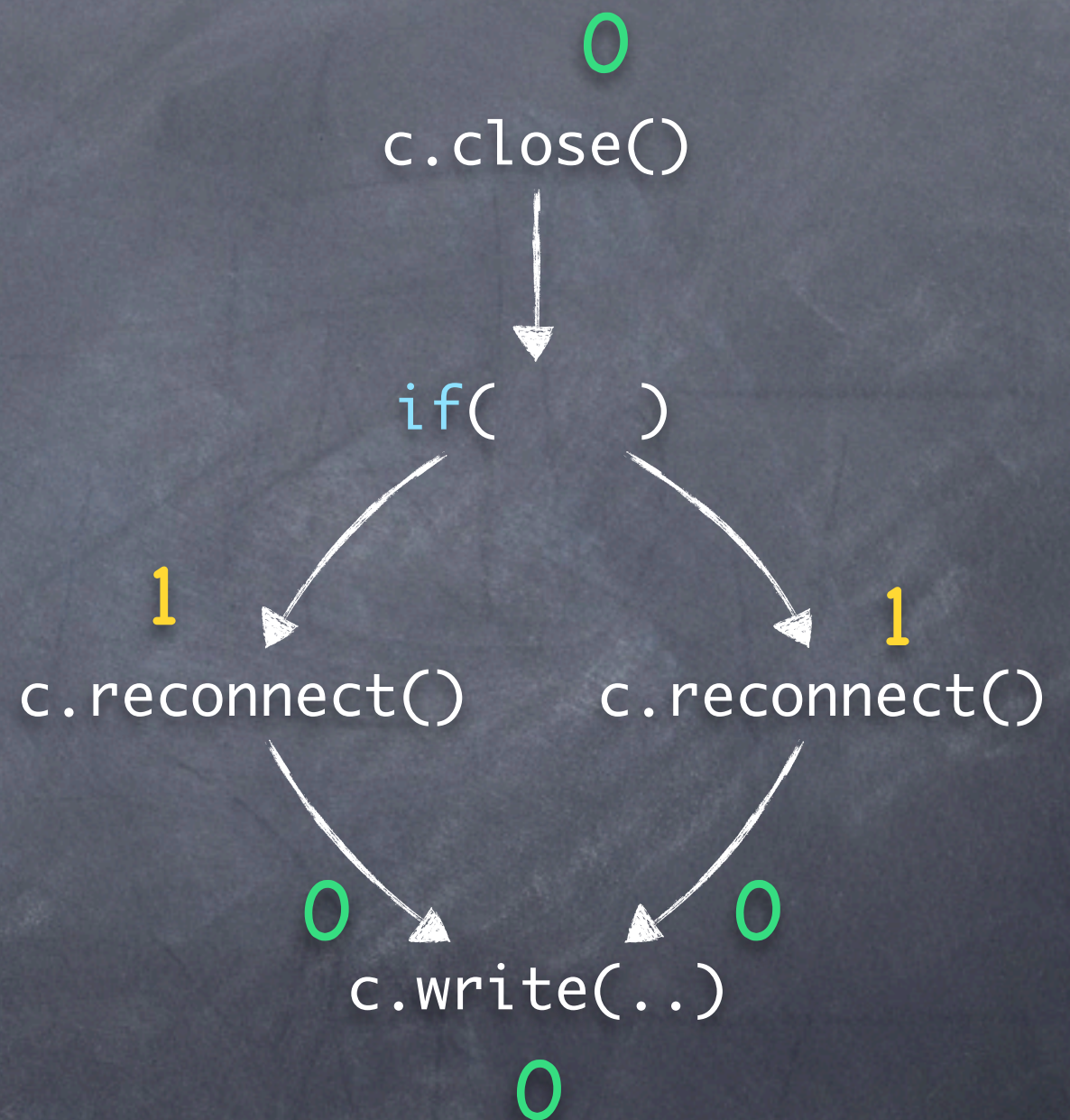
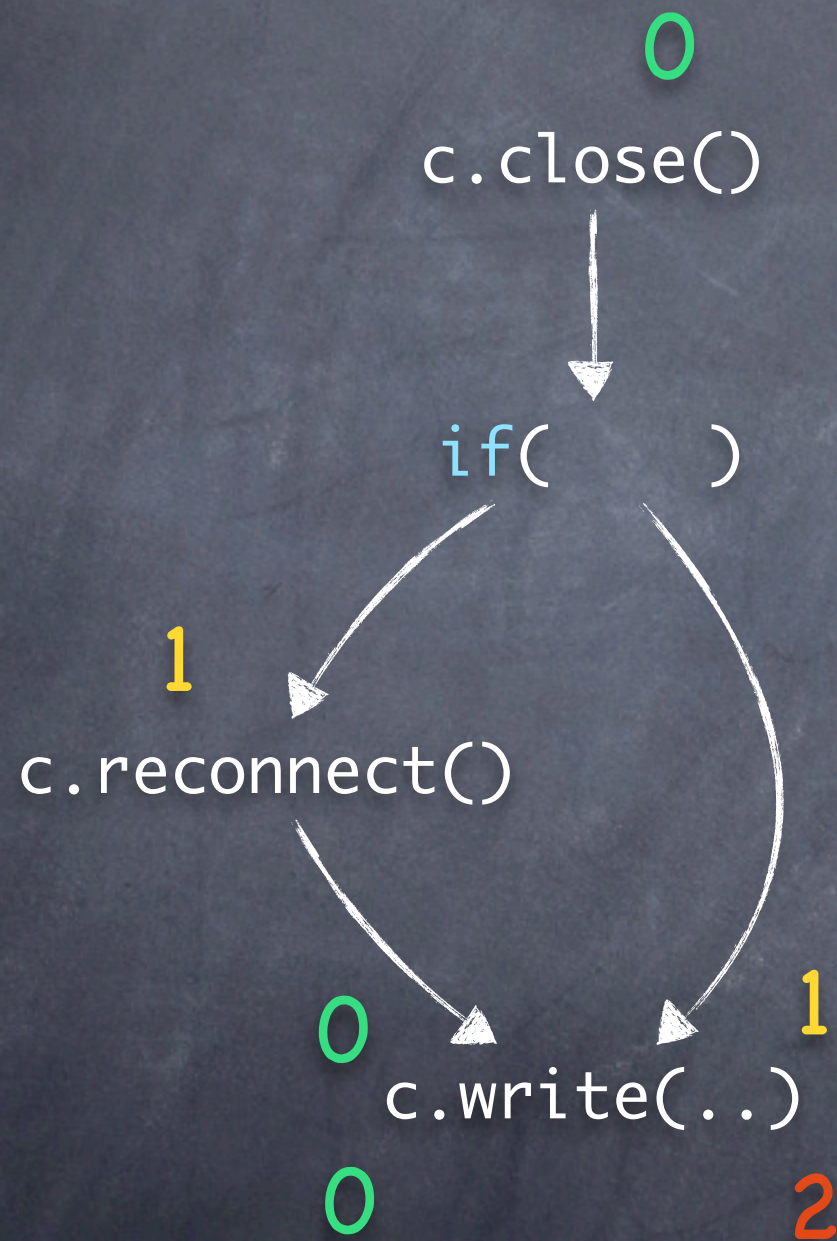
Pure static analysis



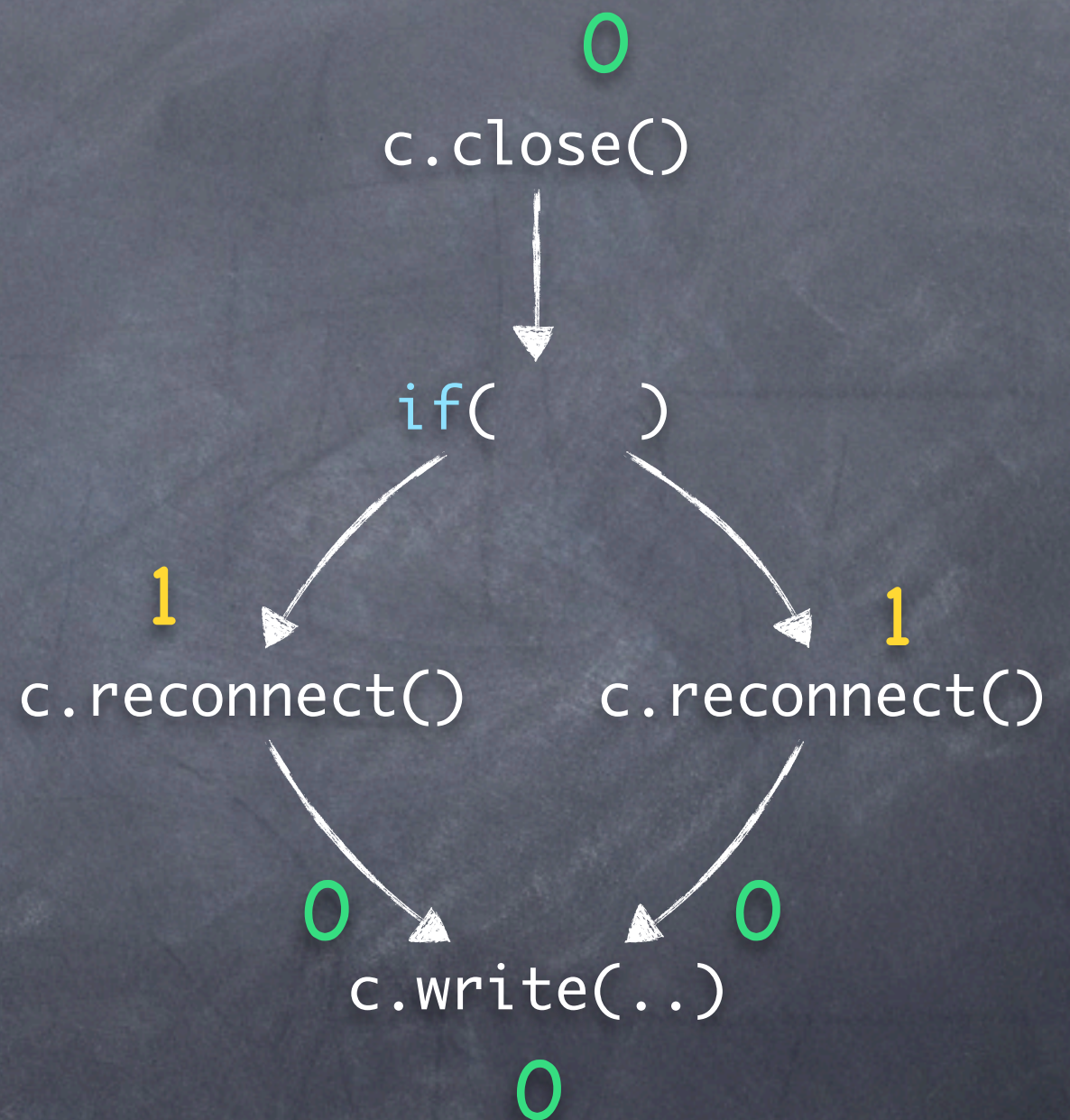
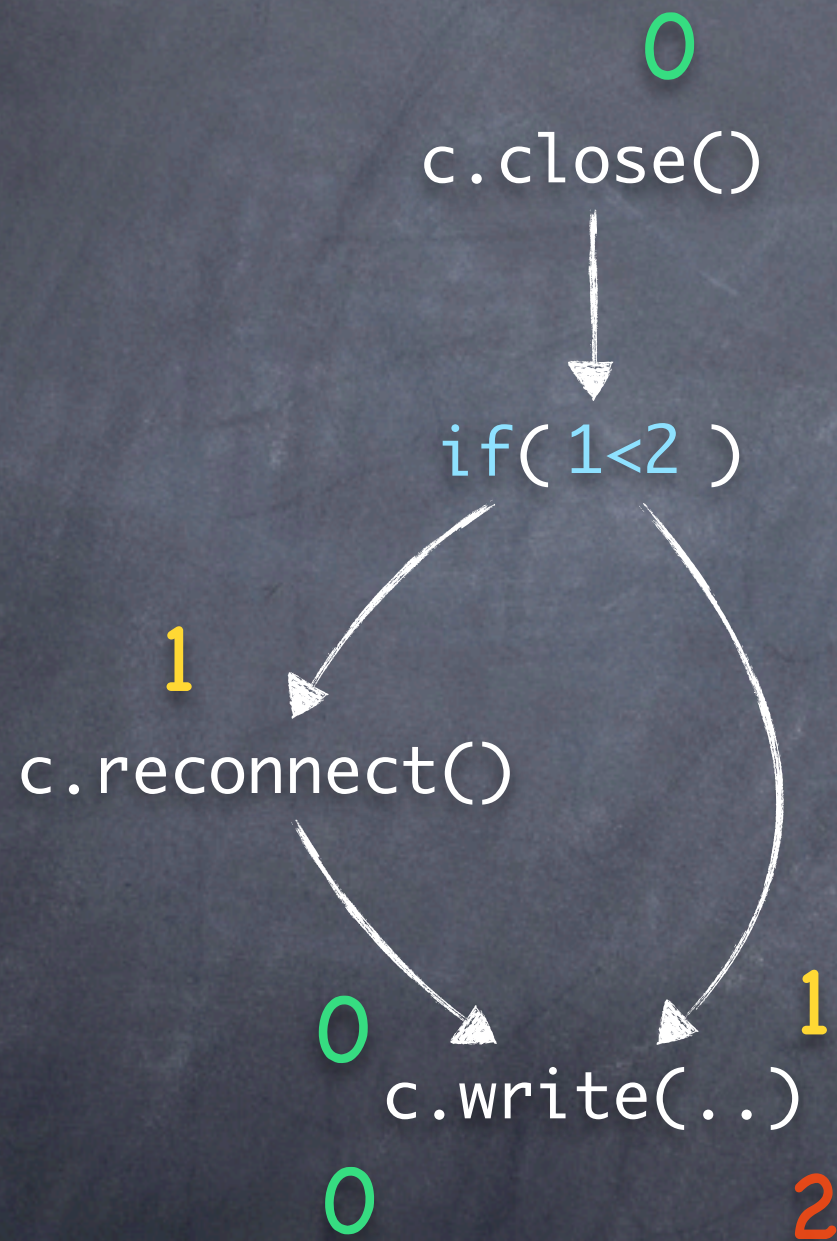
Pure static analysis



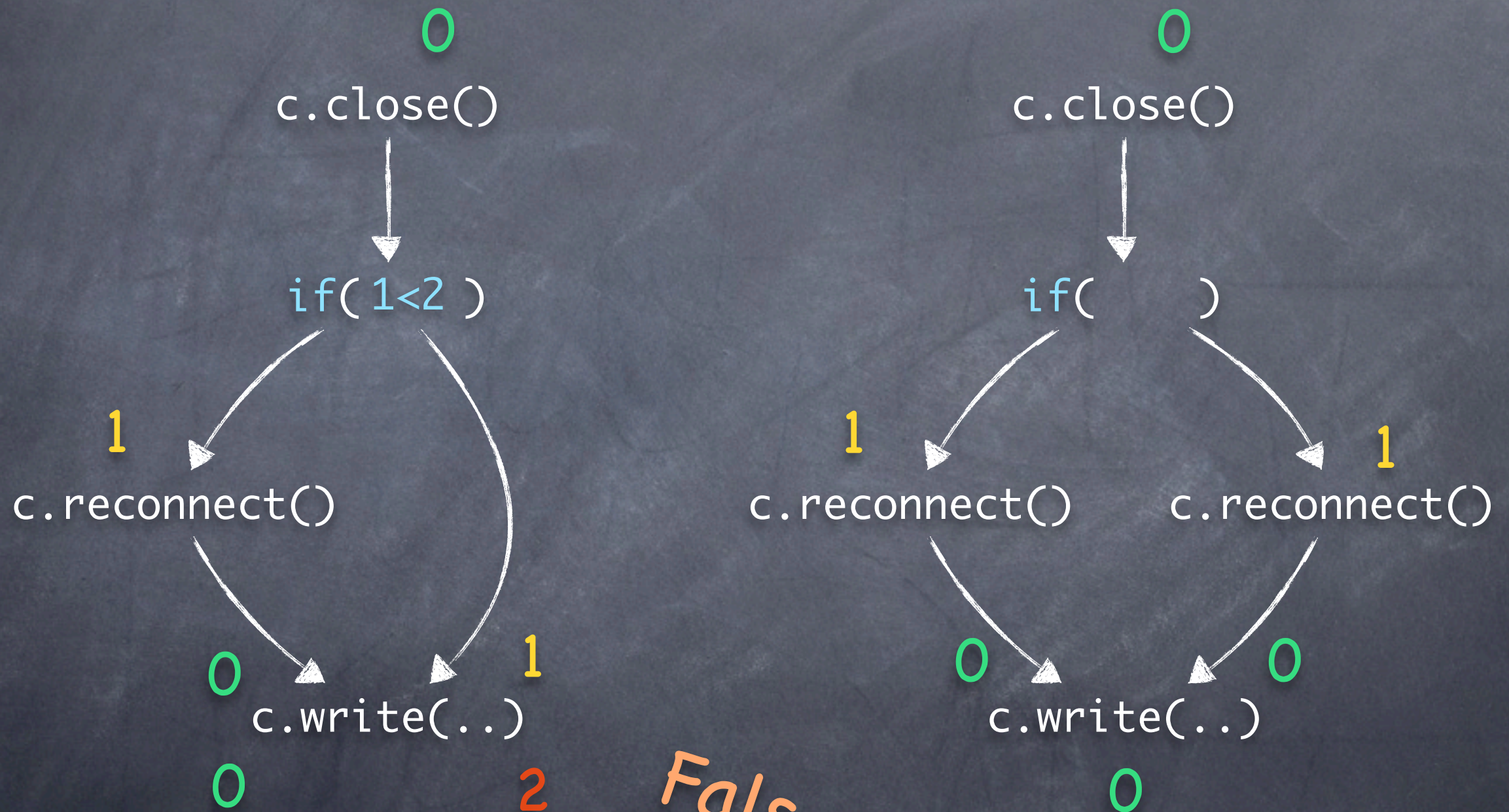
Pure static analysis



Pure static analysis

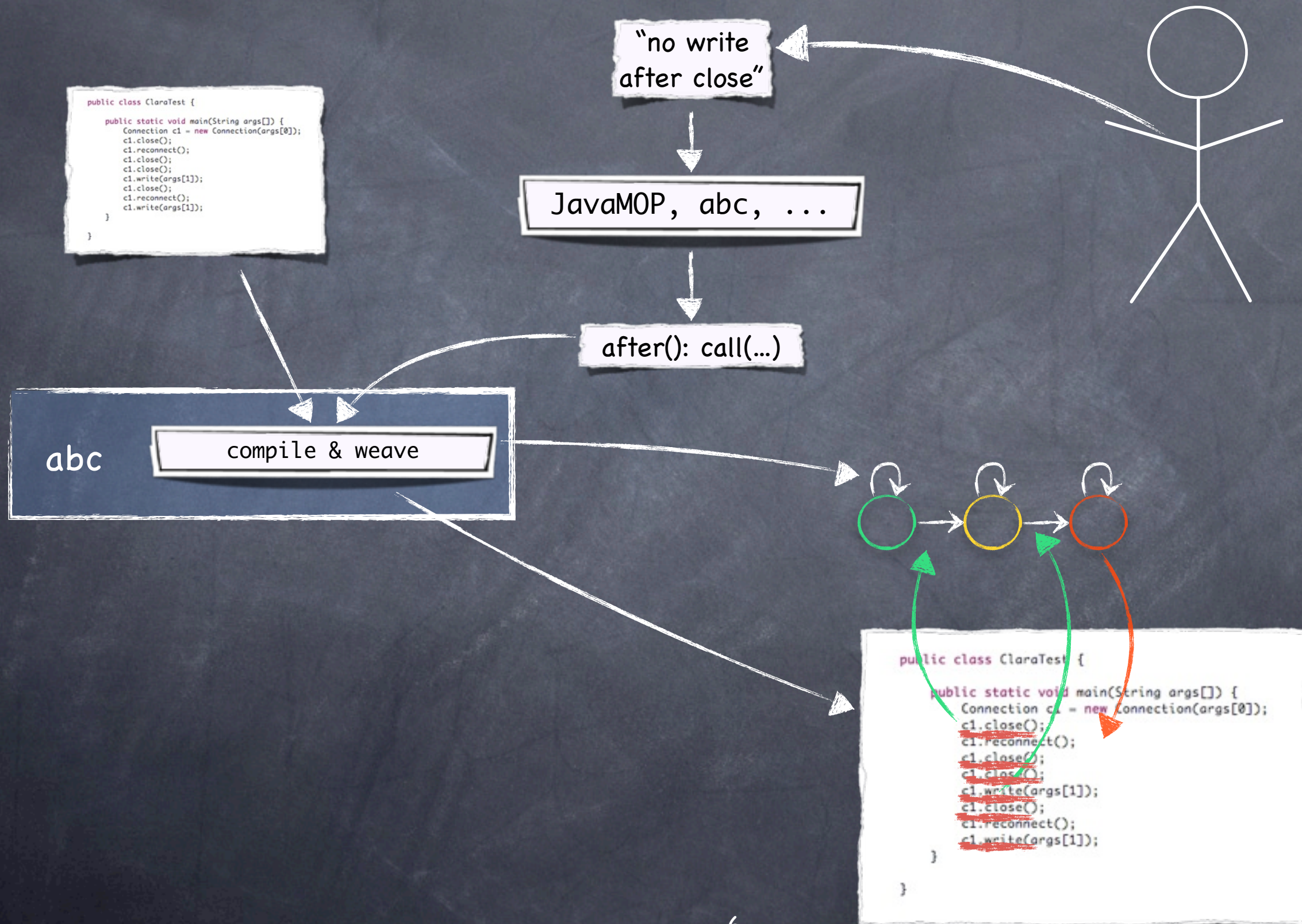


Pure static analysis

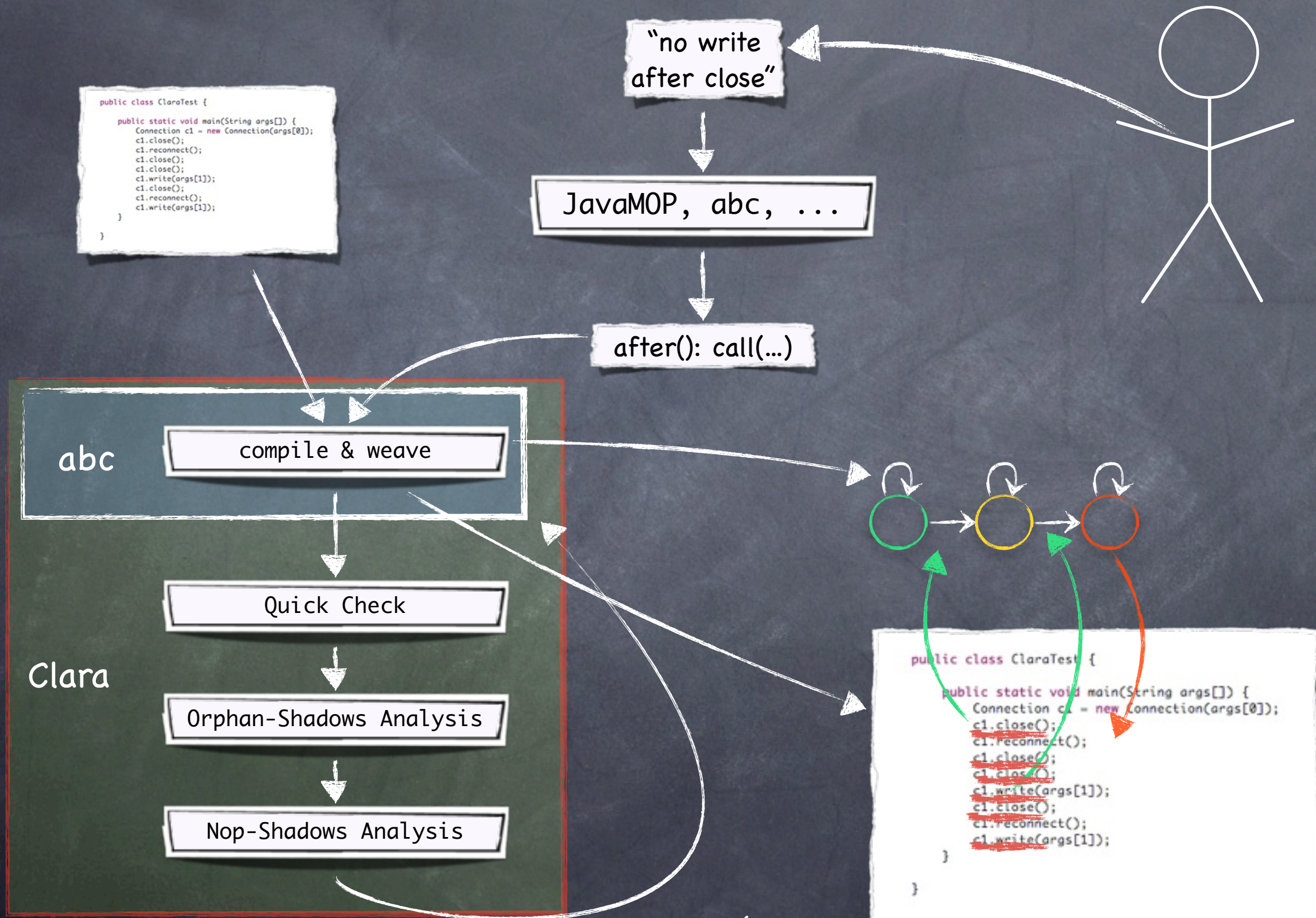


False Positive!

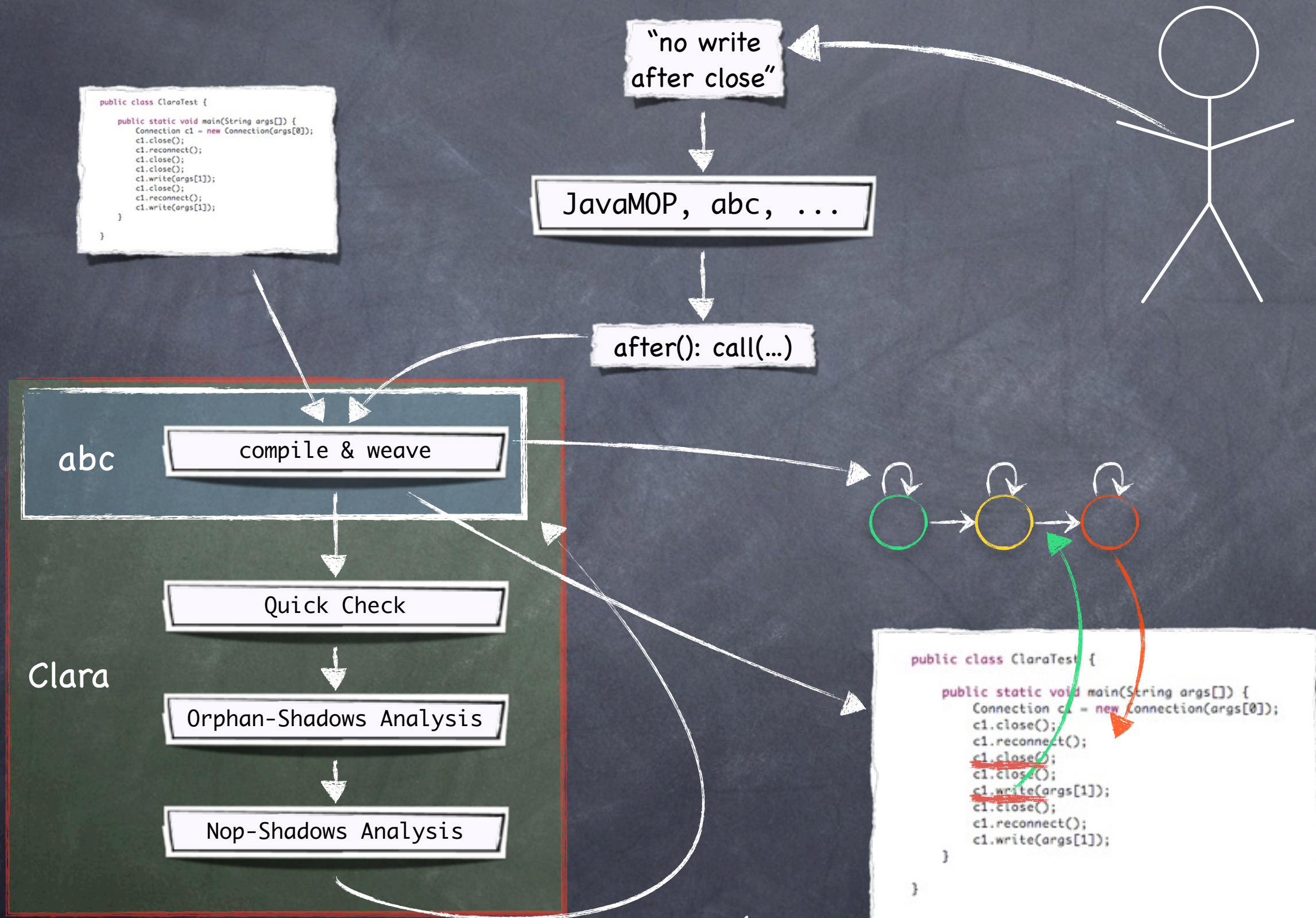
The Clara Framework

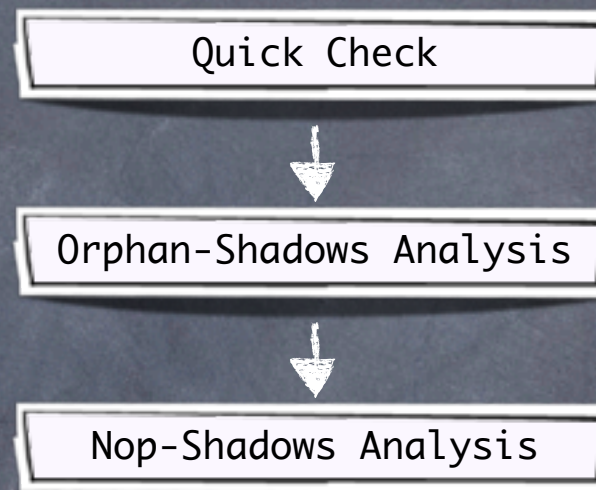


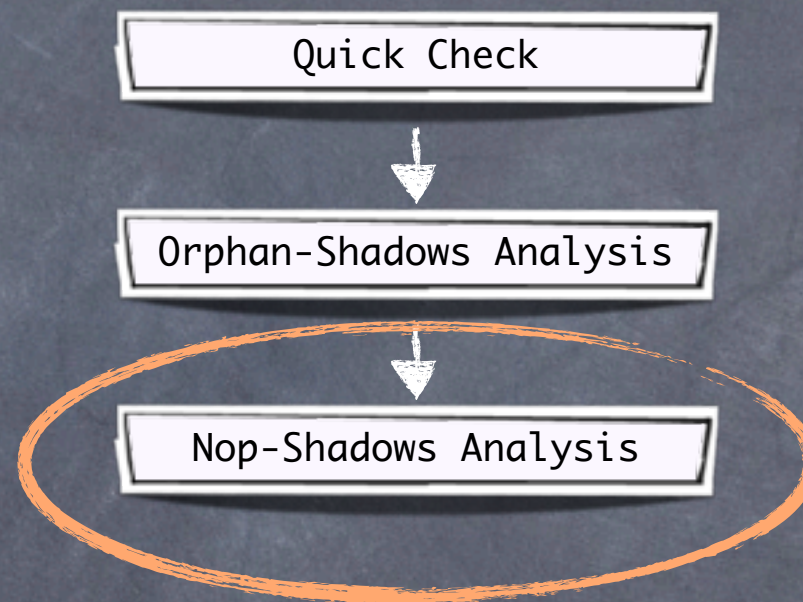
The Clara Framework

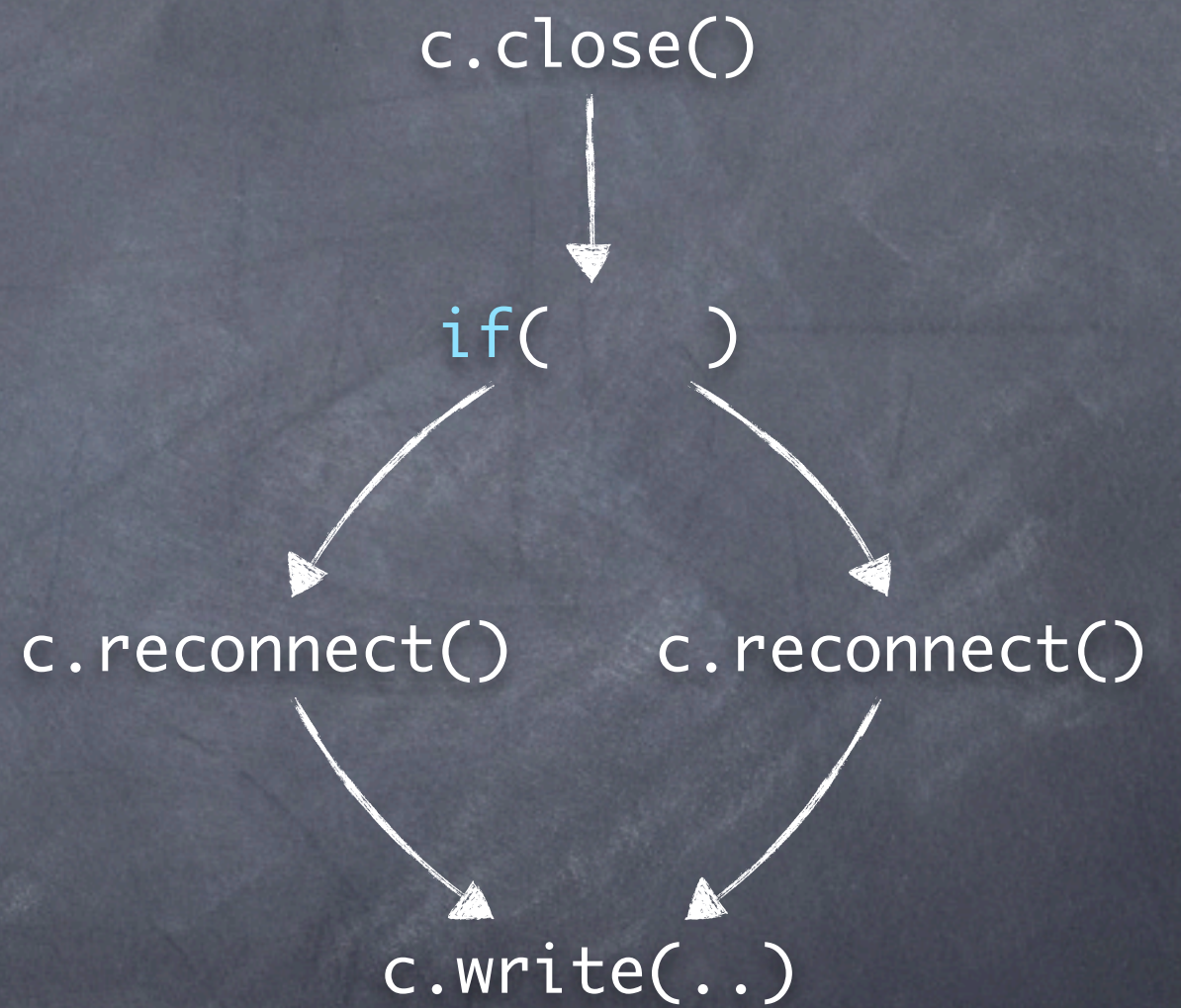
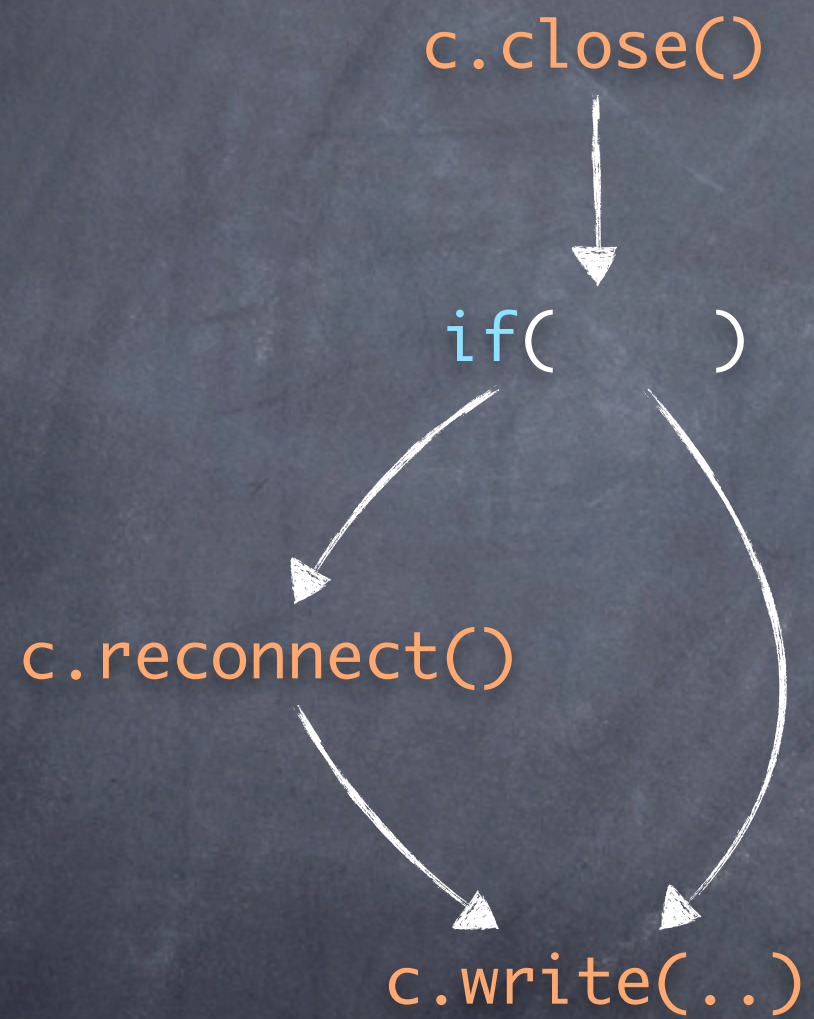


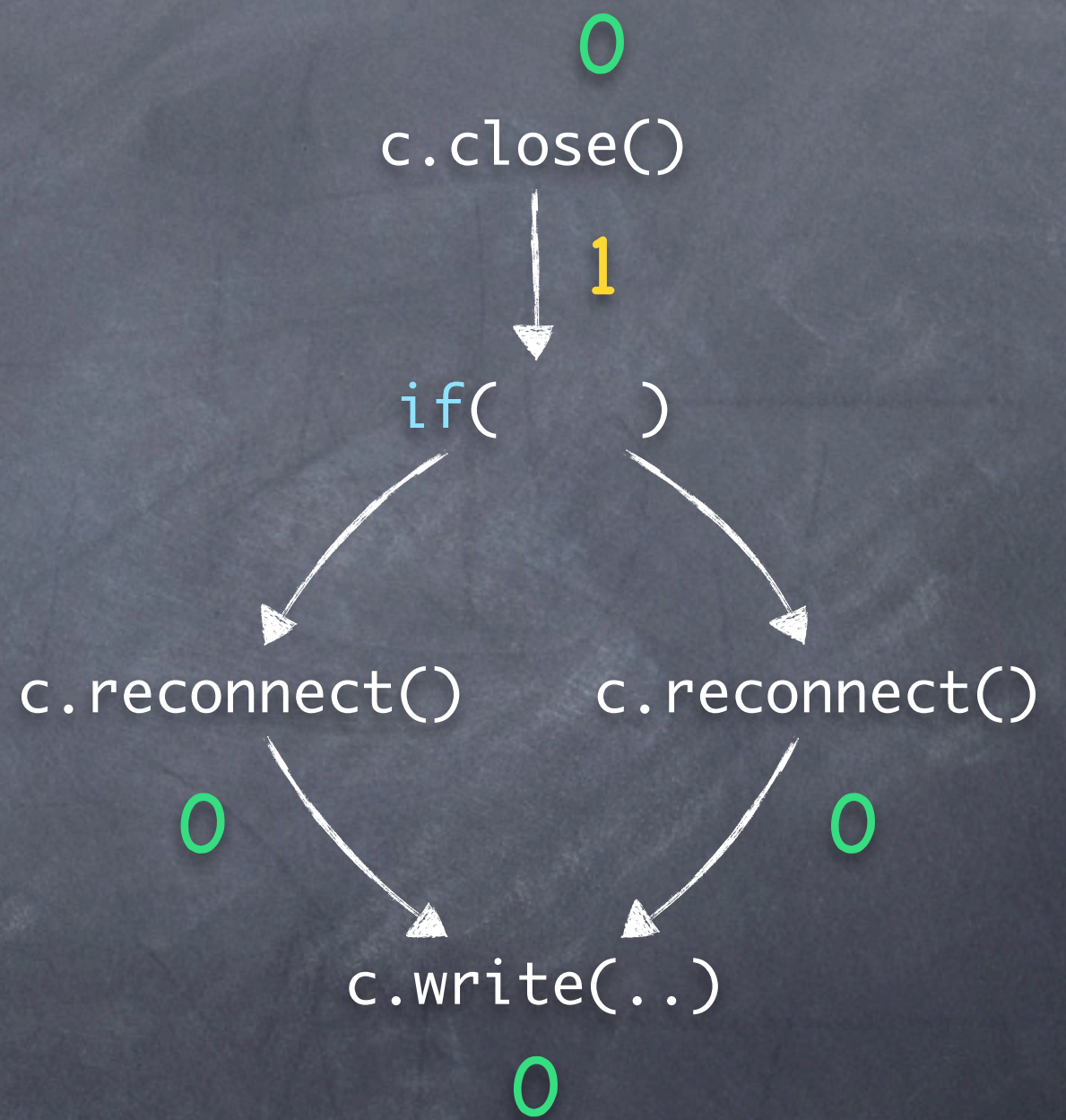
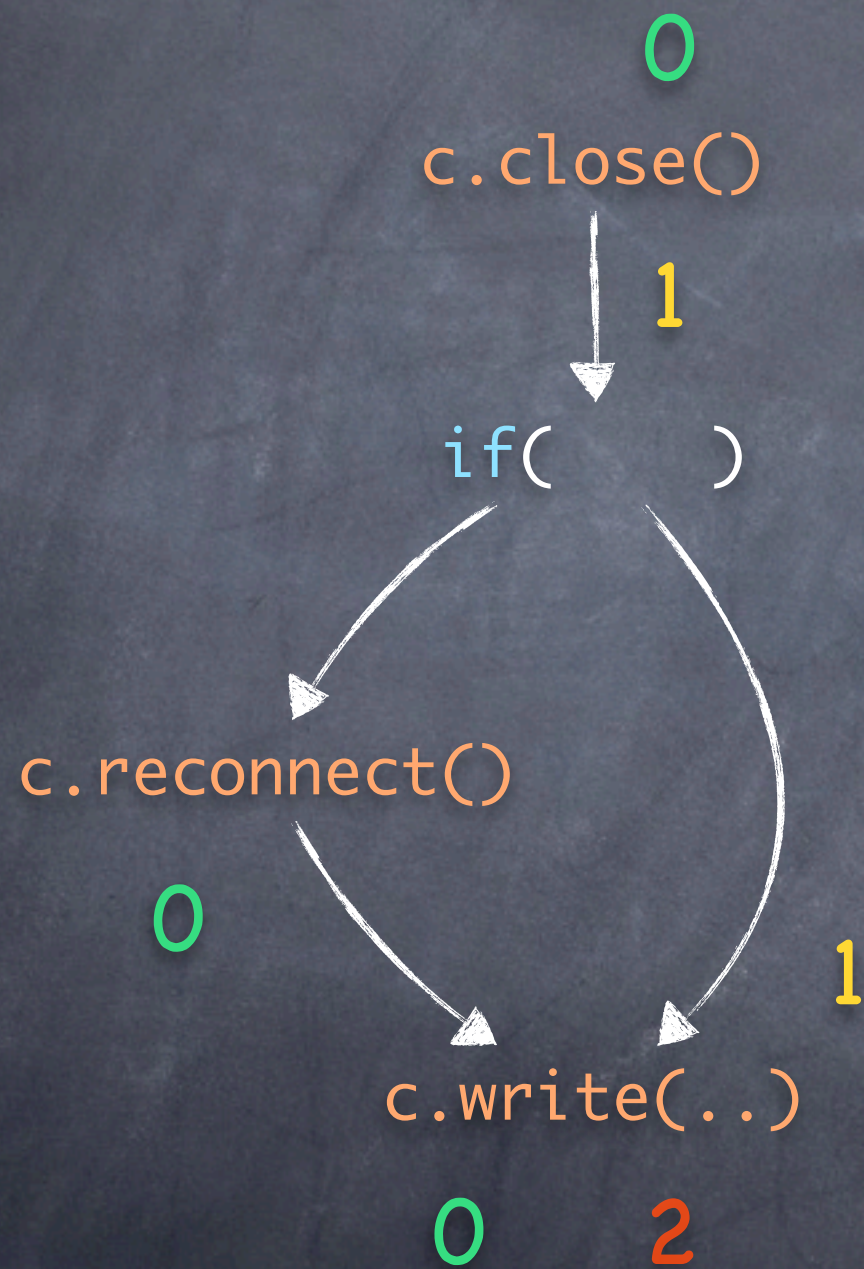
The Clara Framework

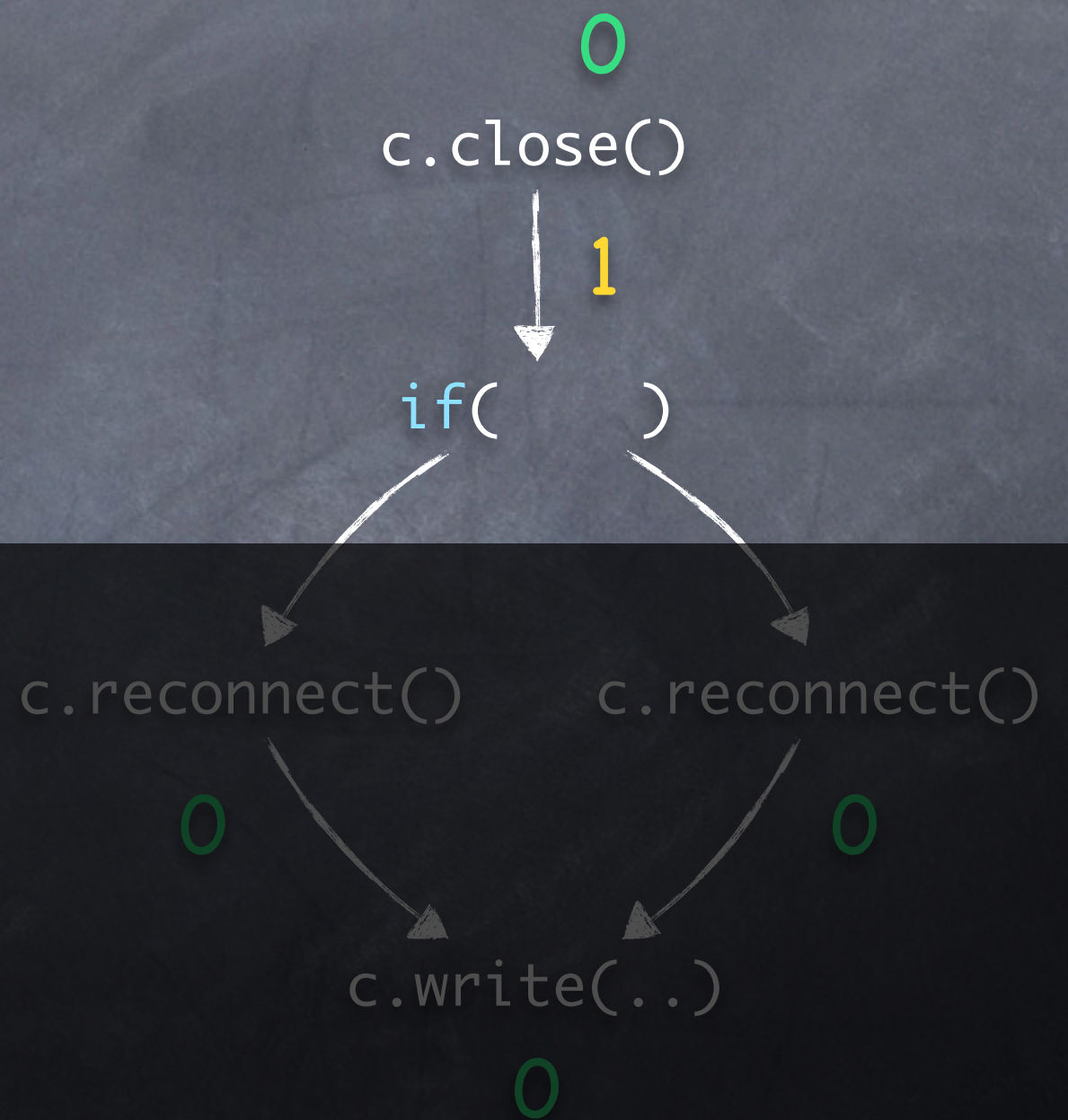
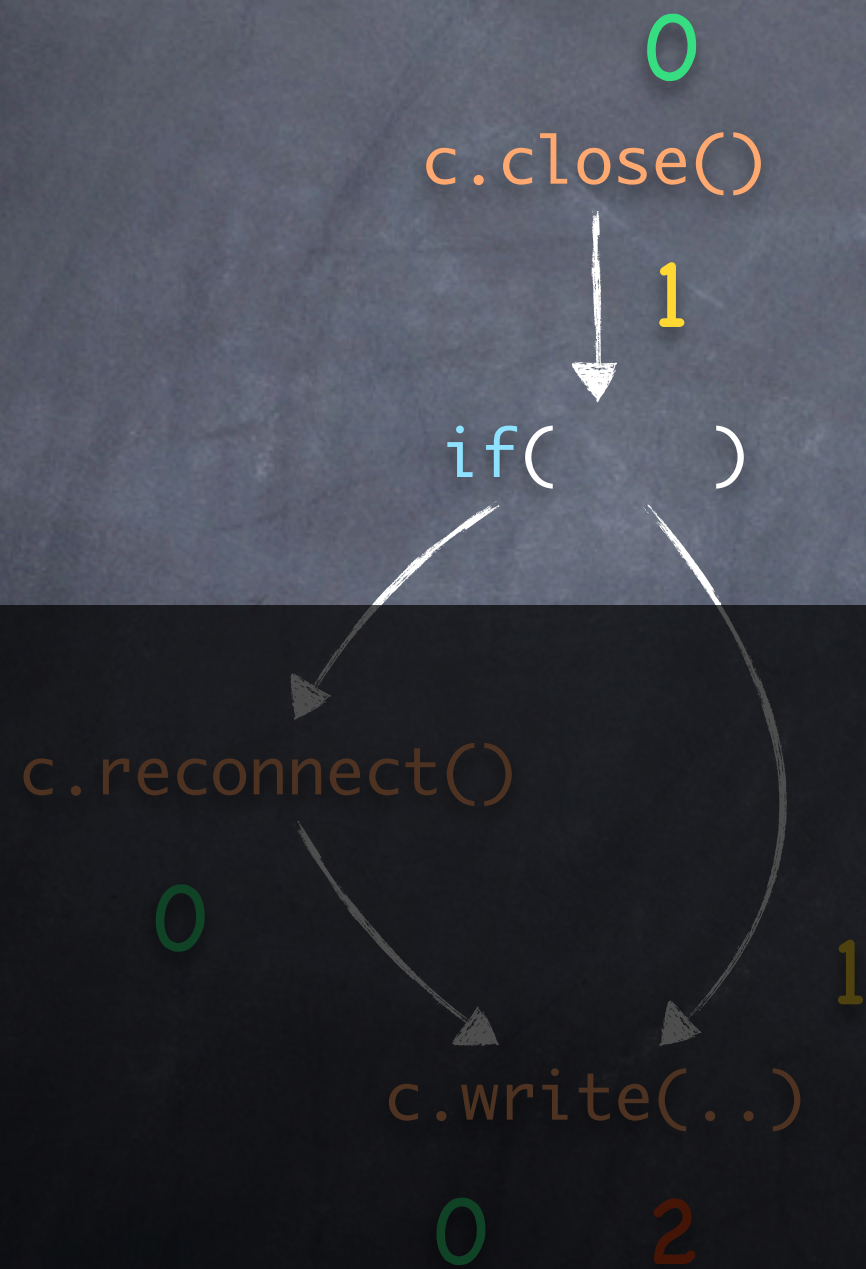












Nop-Shadows Analysis

Idea:

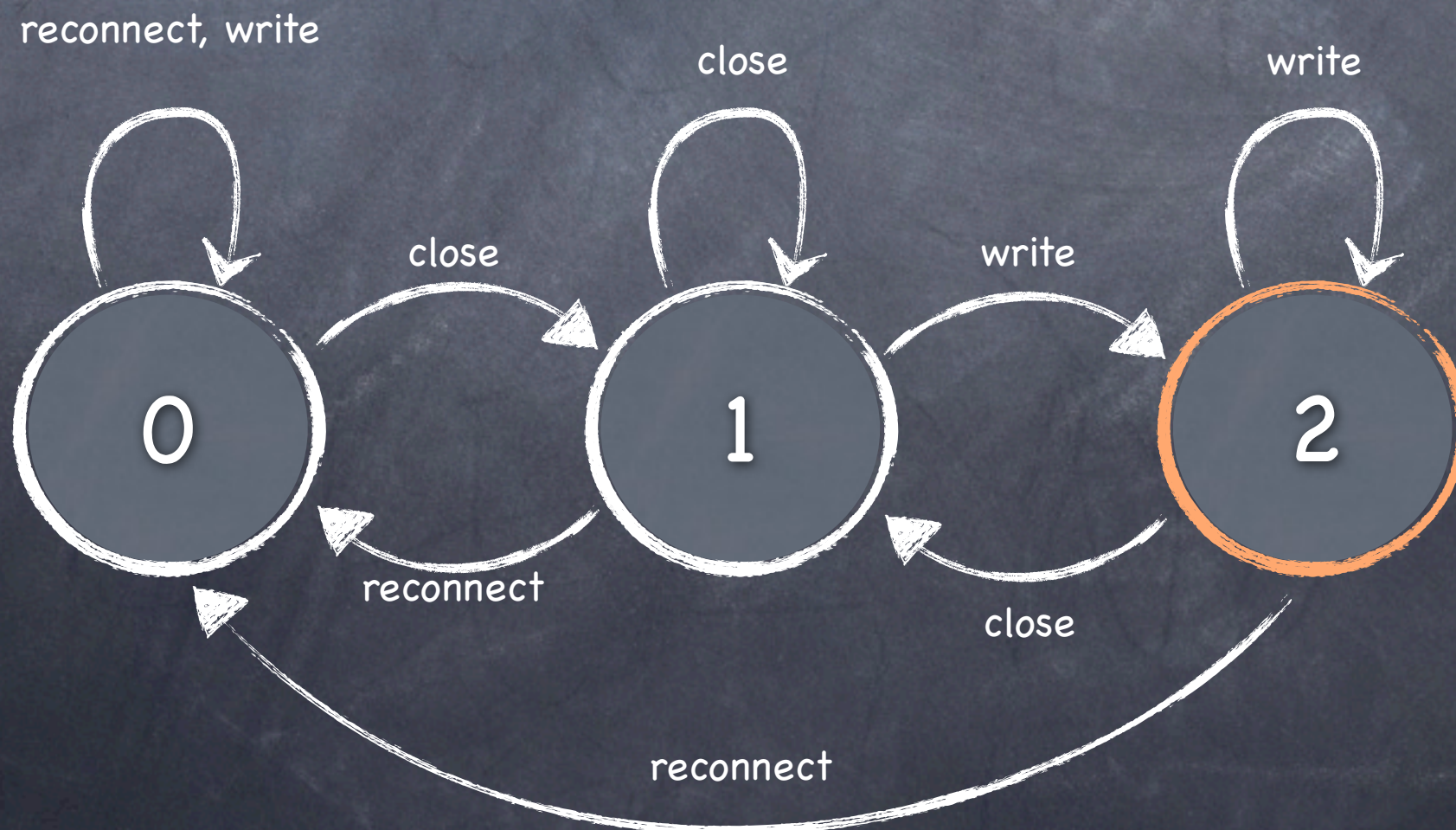
For every transitioning statement s :

- Identify states that are equivalent at s .
- If s may transition only between equivalent states then disable transitions at s .

`c.close()`

`c.reconnect()`

`c.write()`





0

1

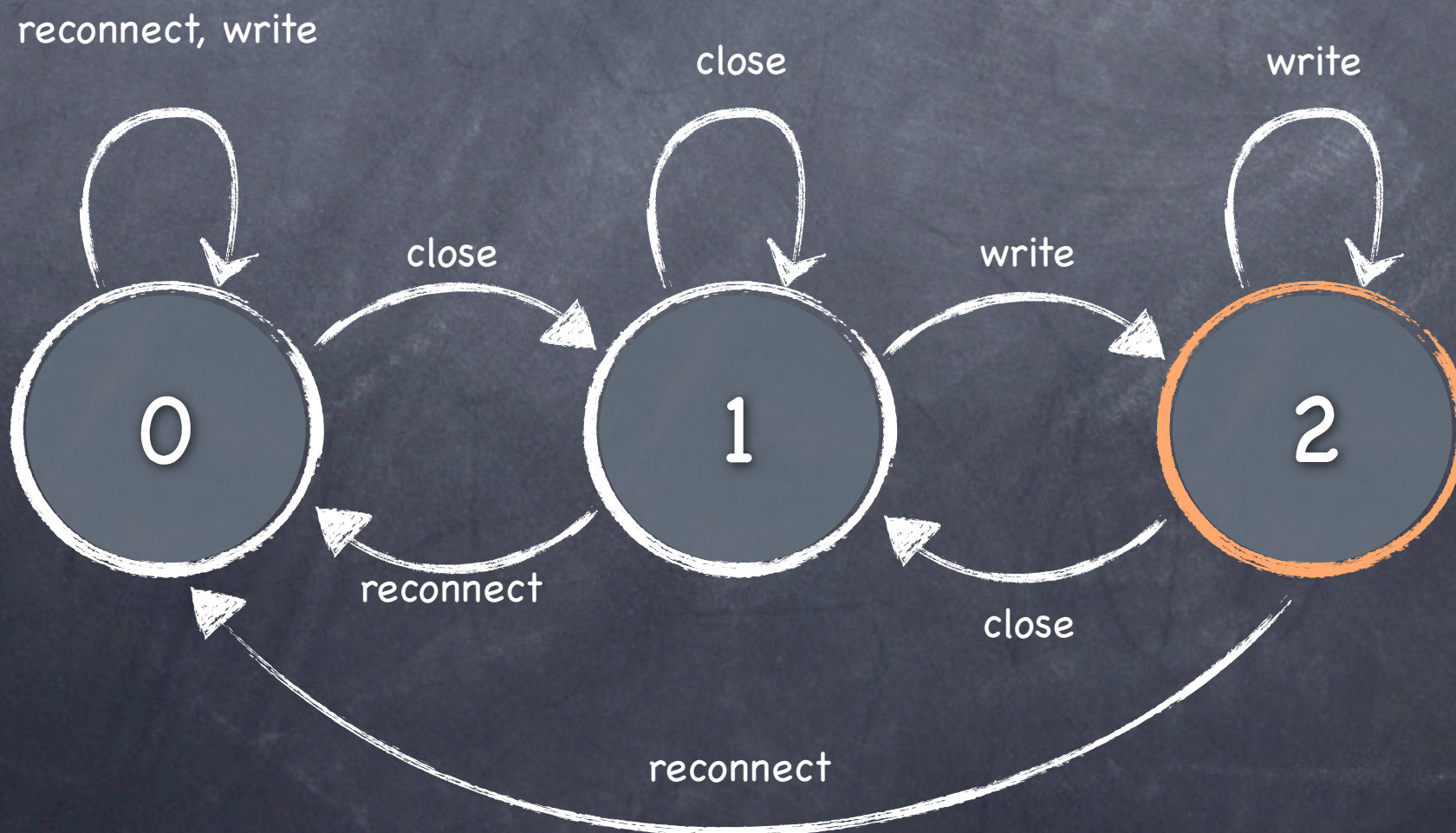
0

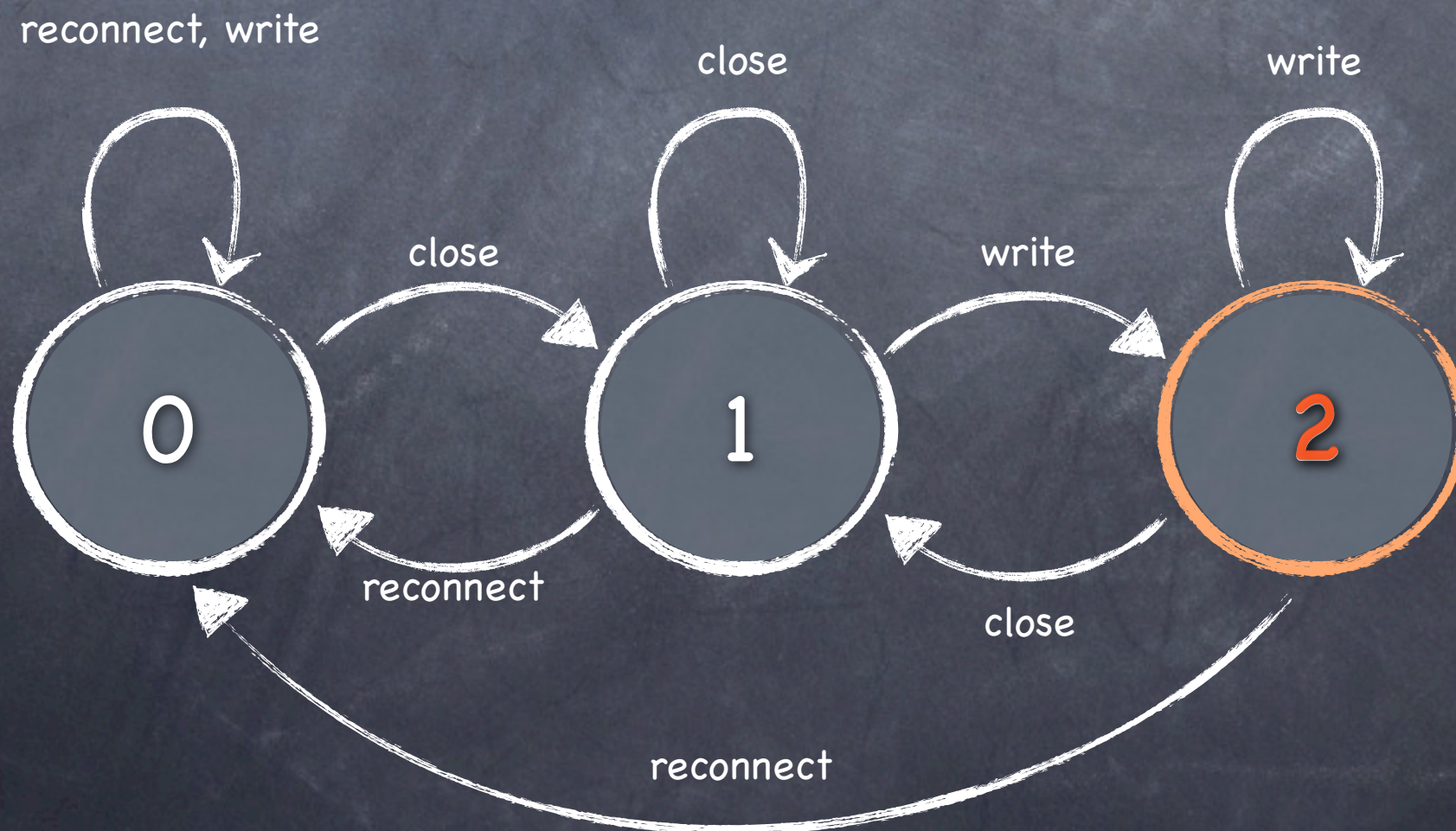
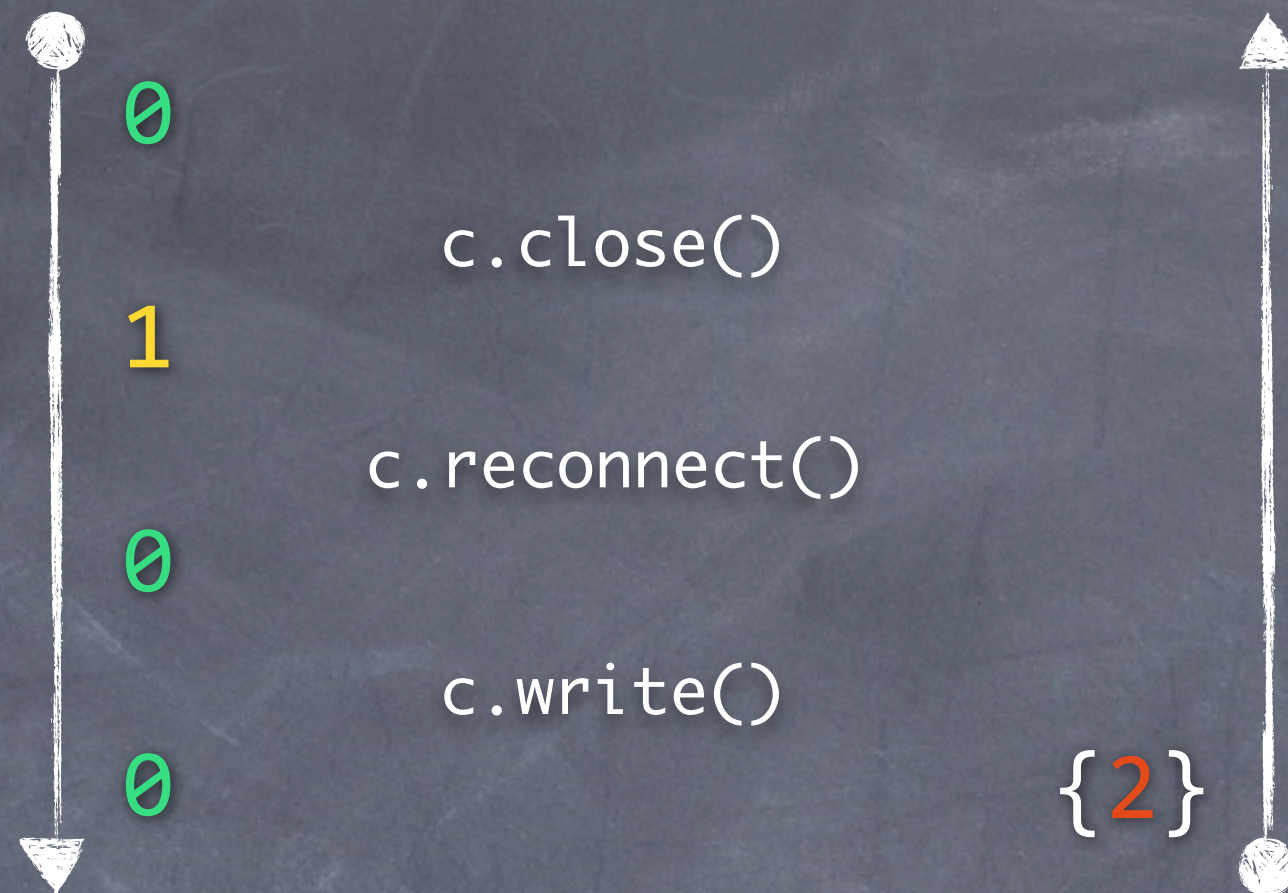
0

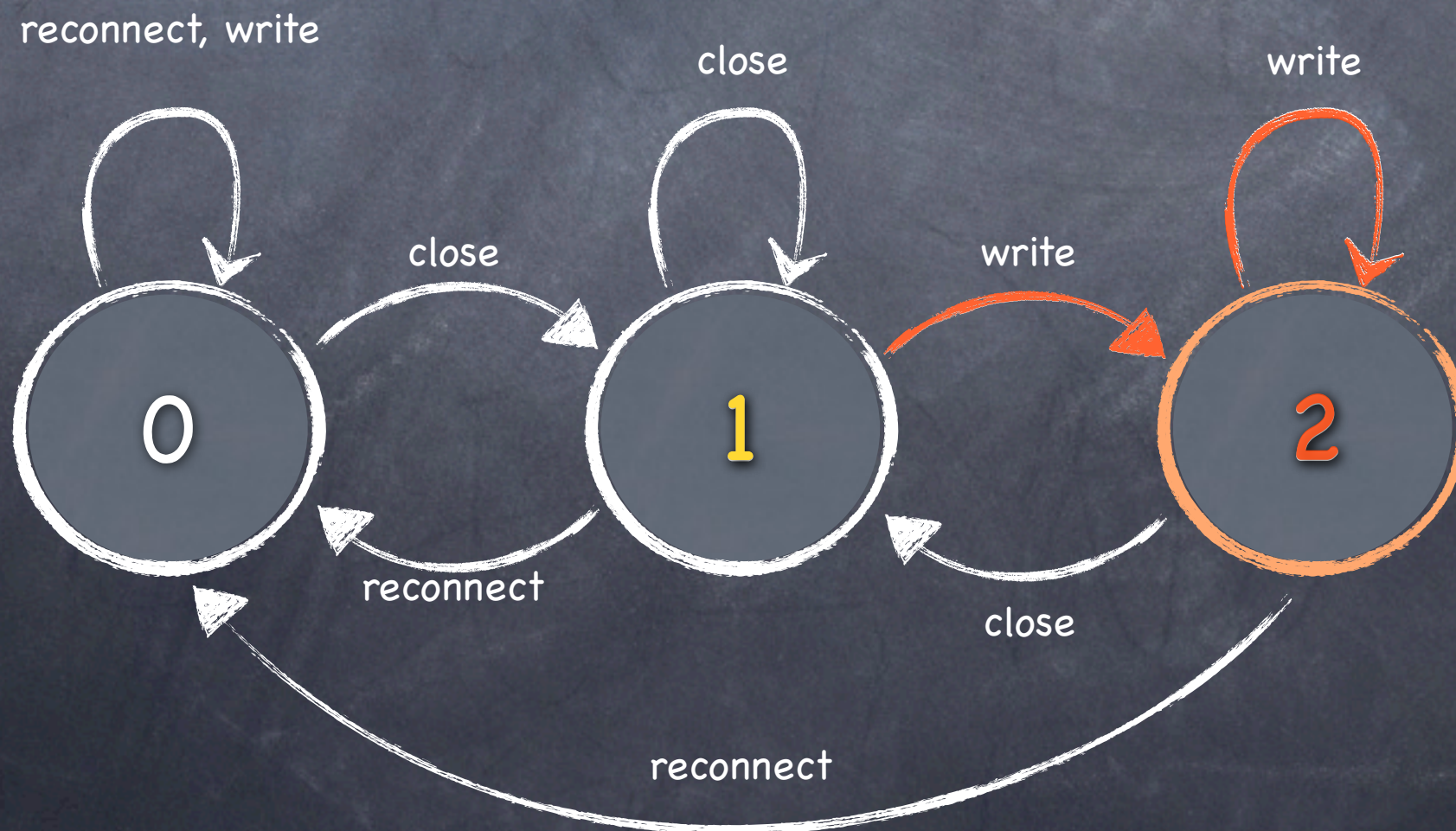
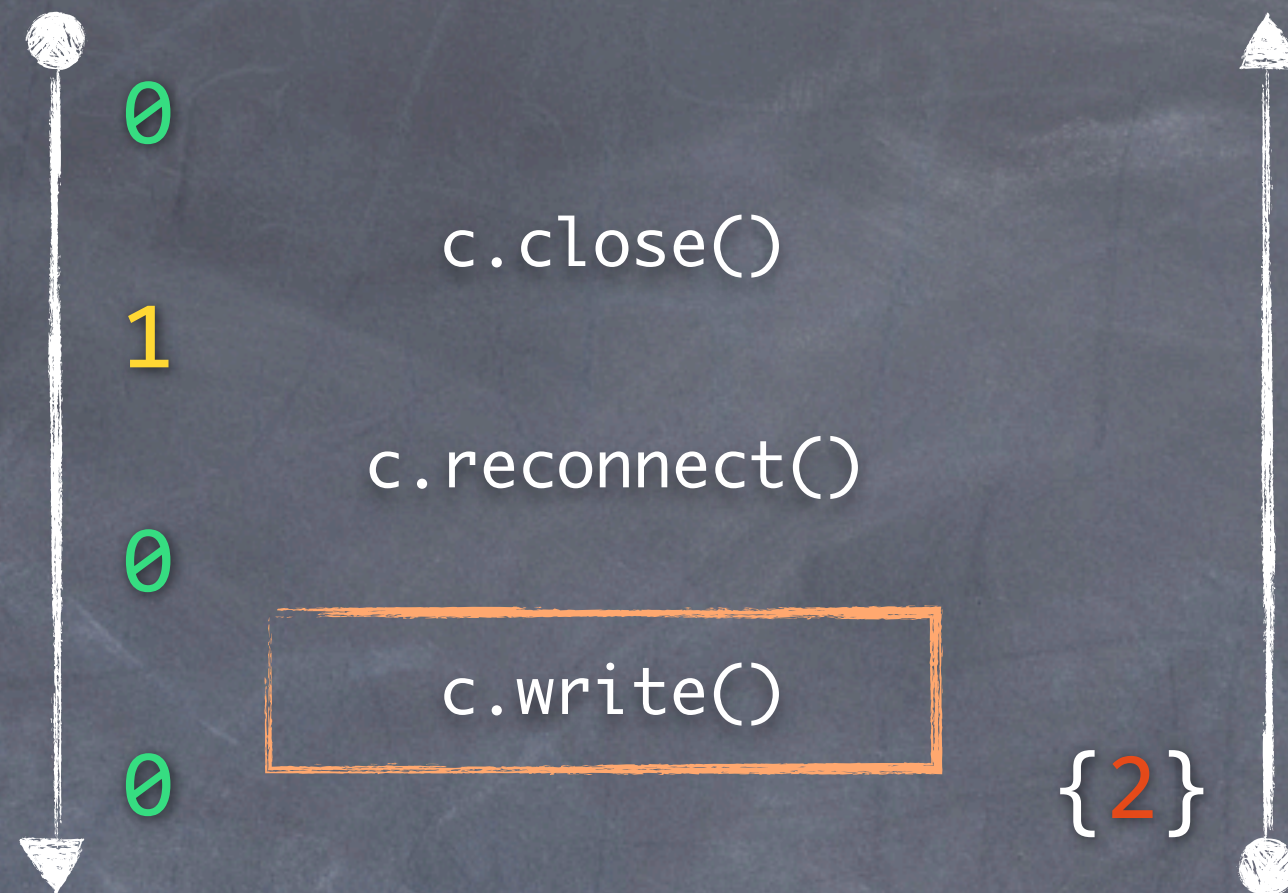
c.close()

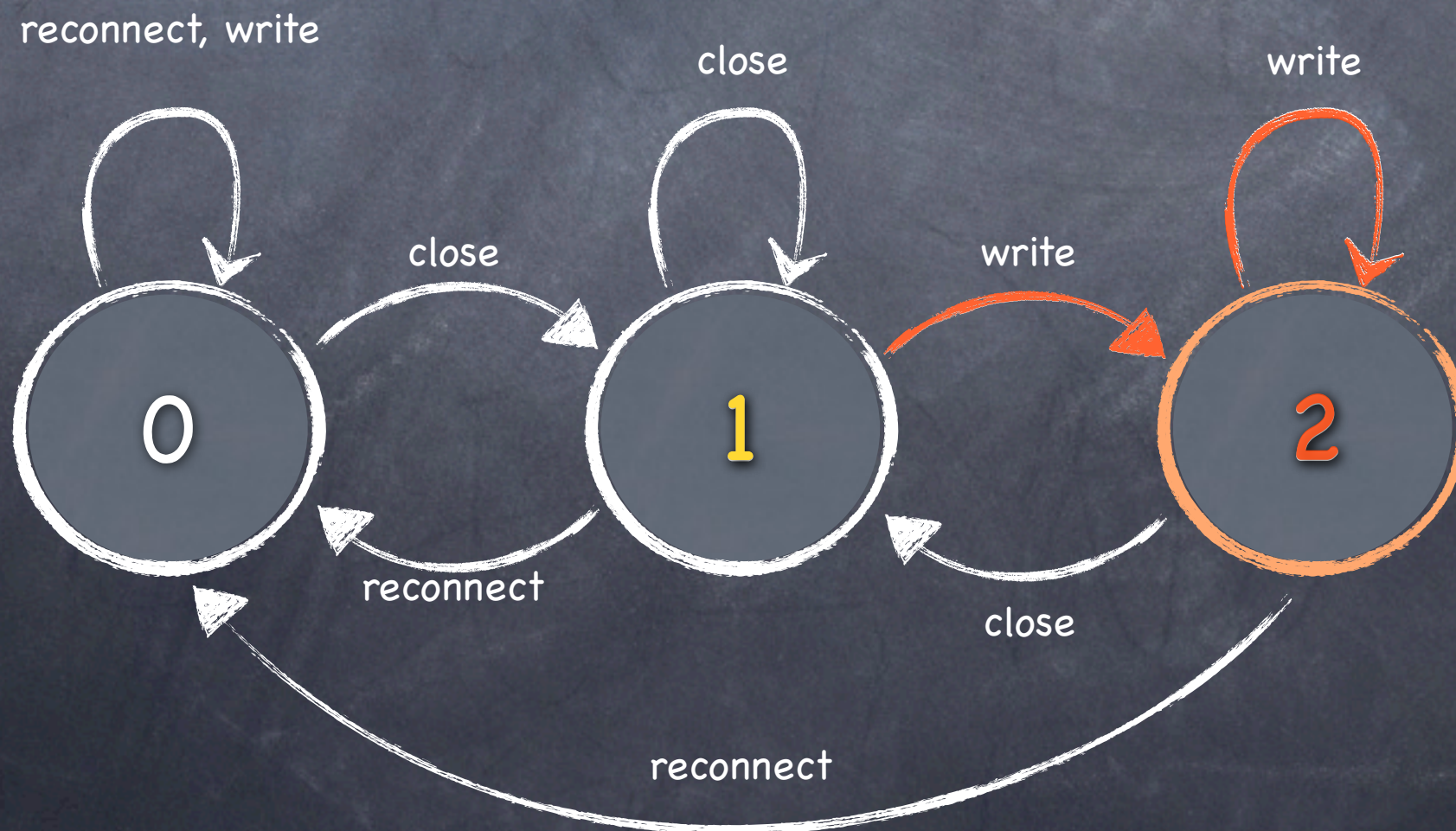
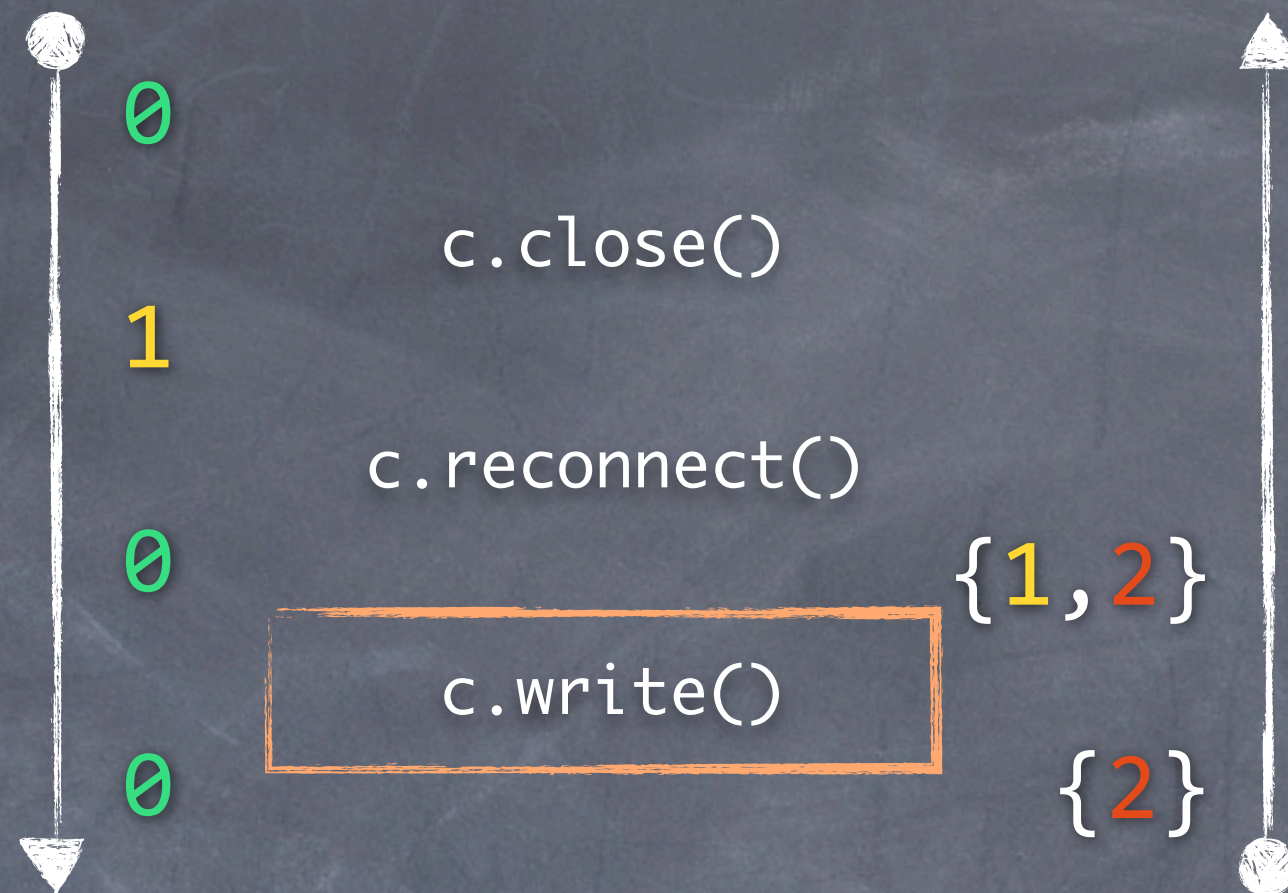
c.reconnect()

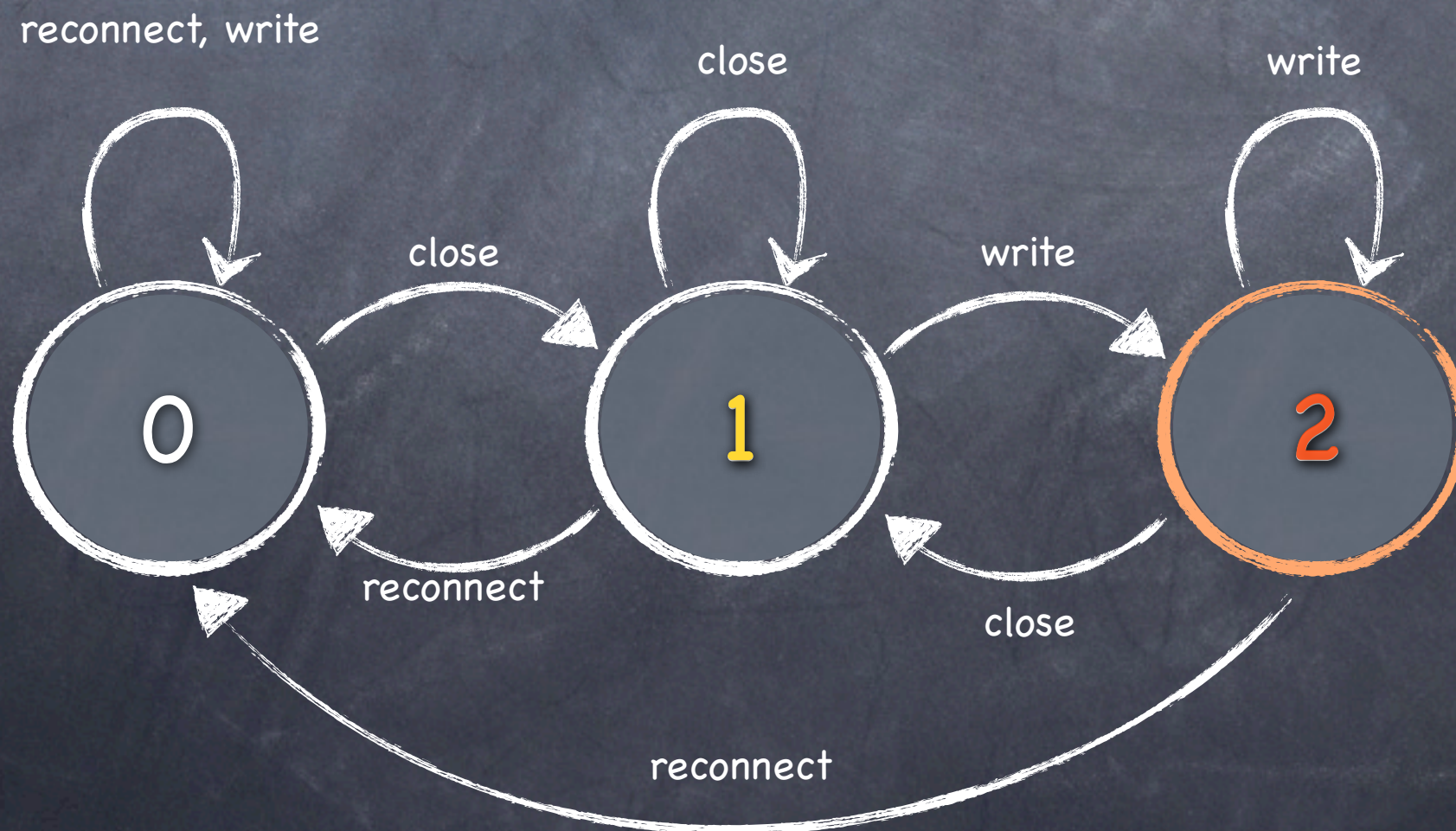
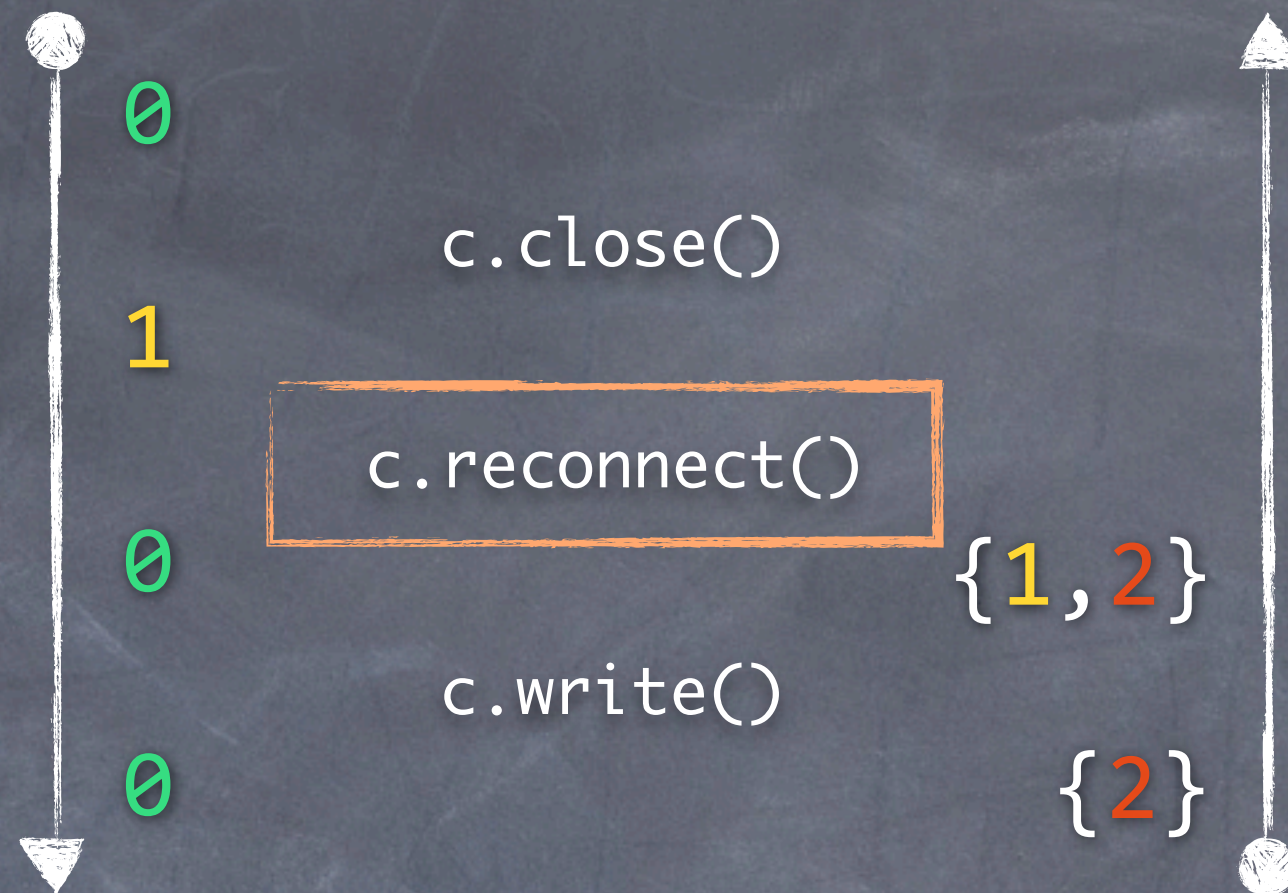
c.write()

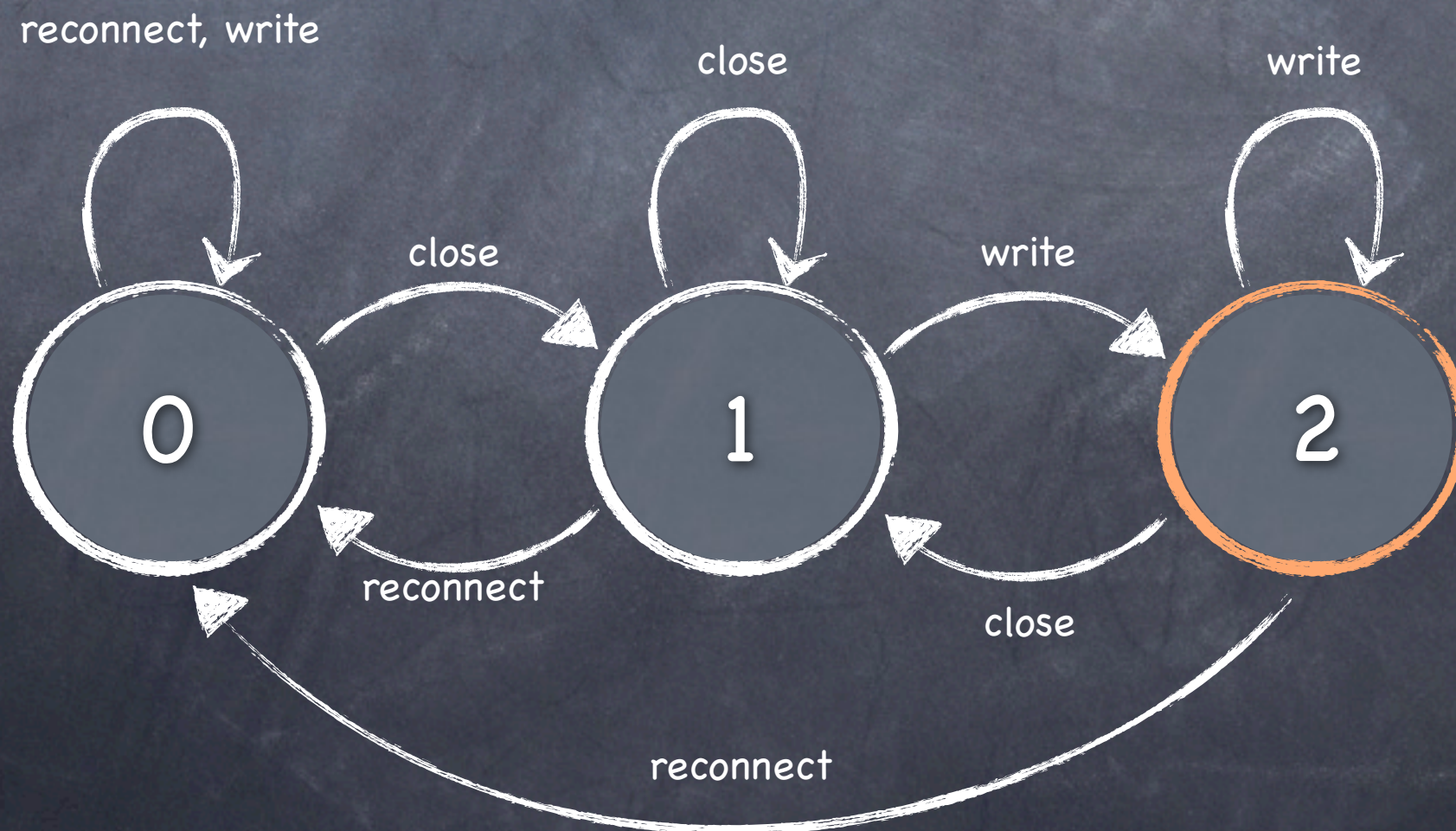
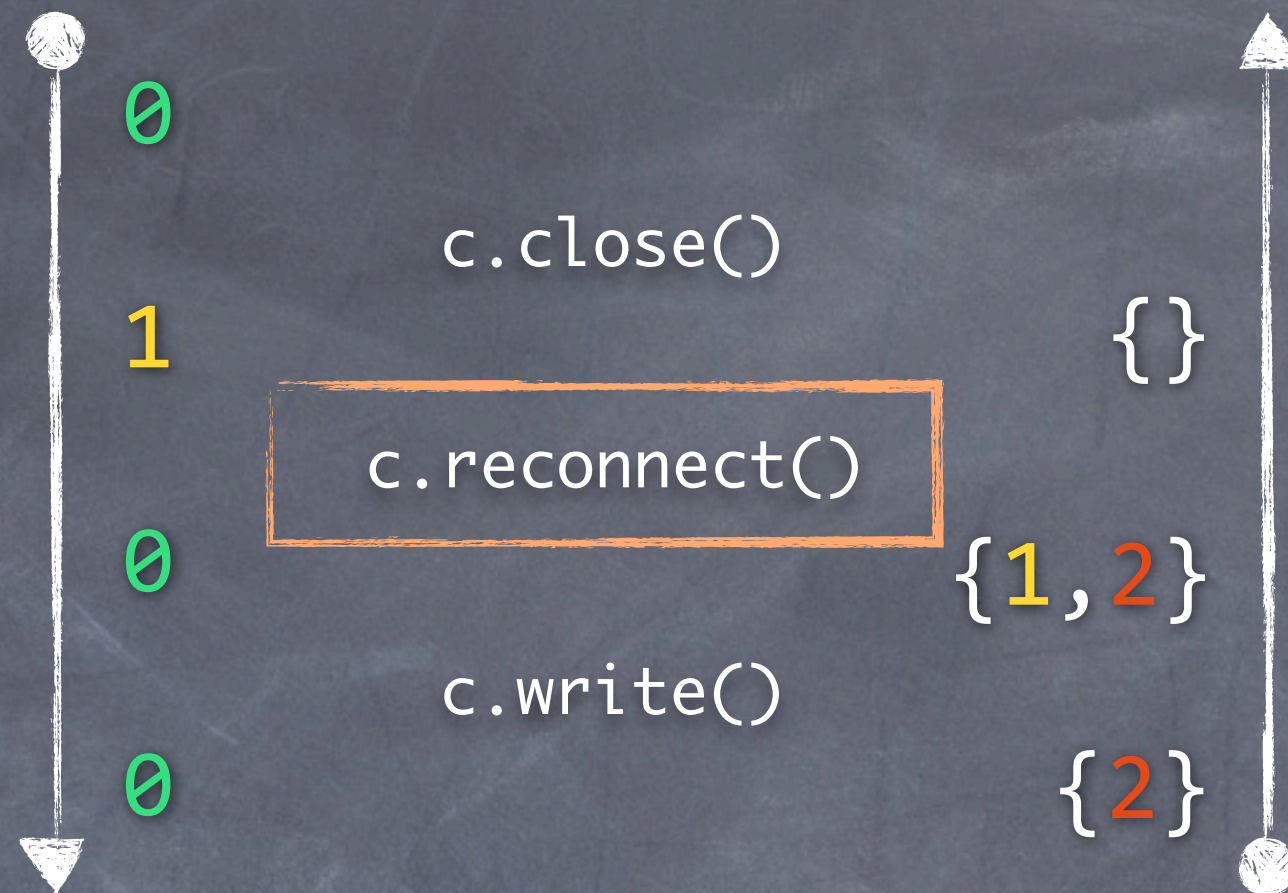


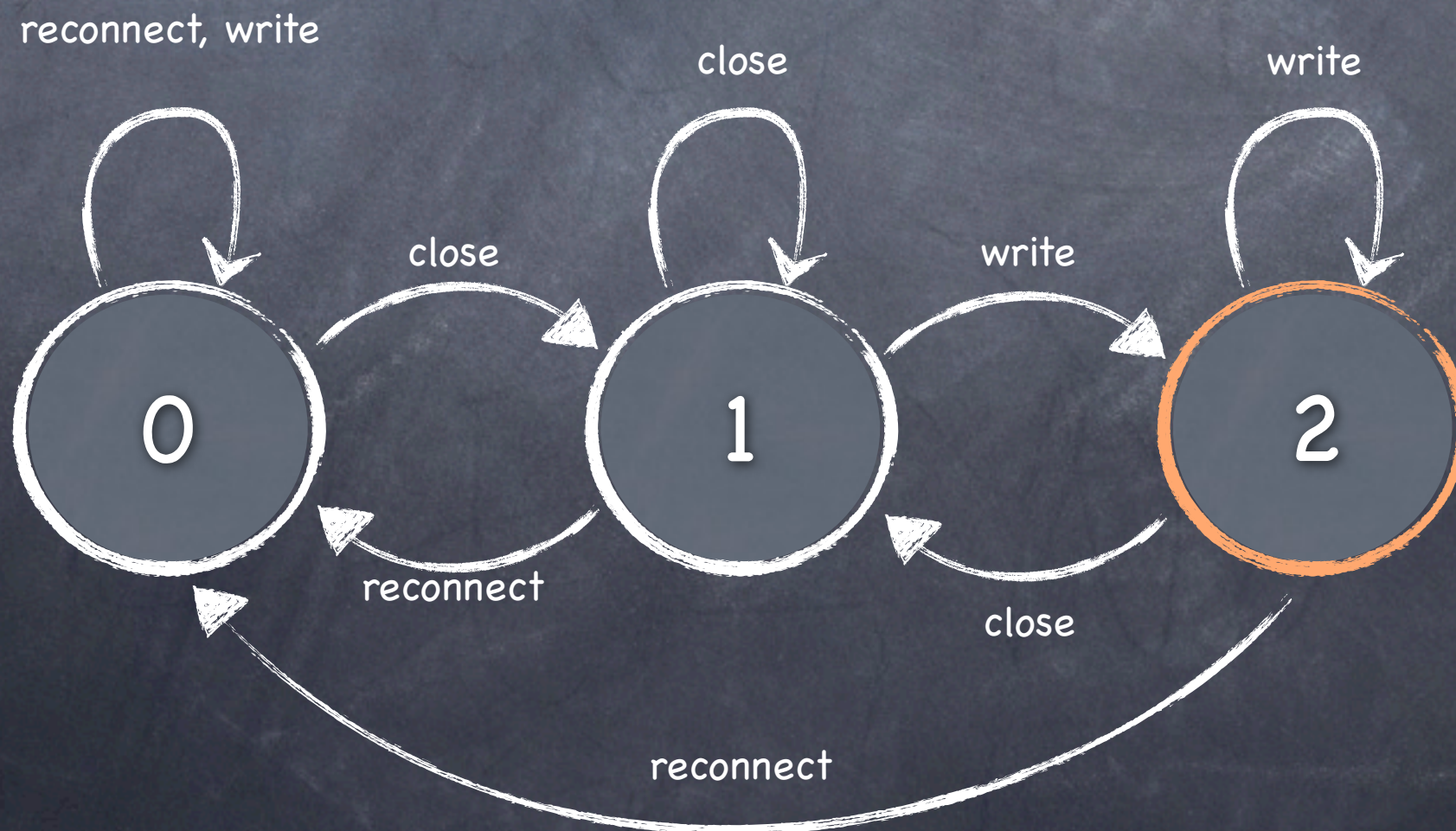
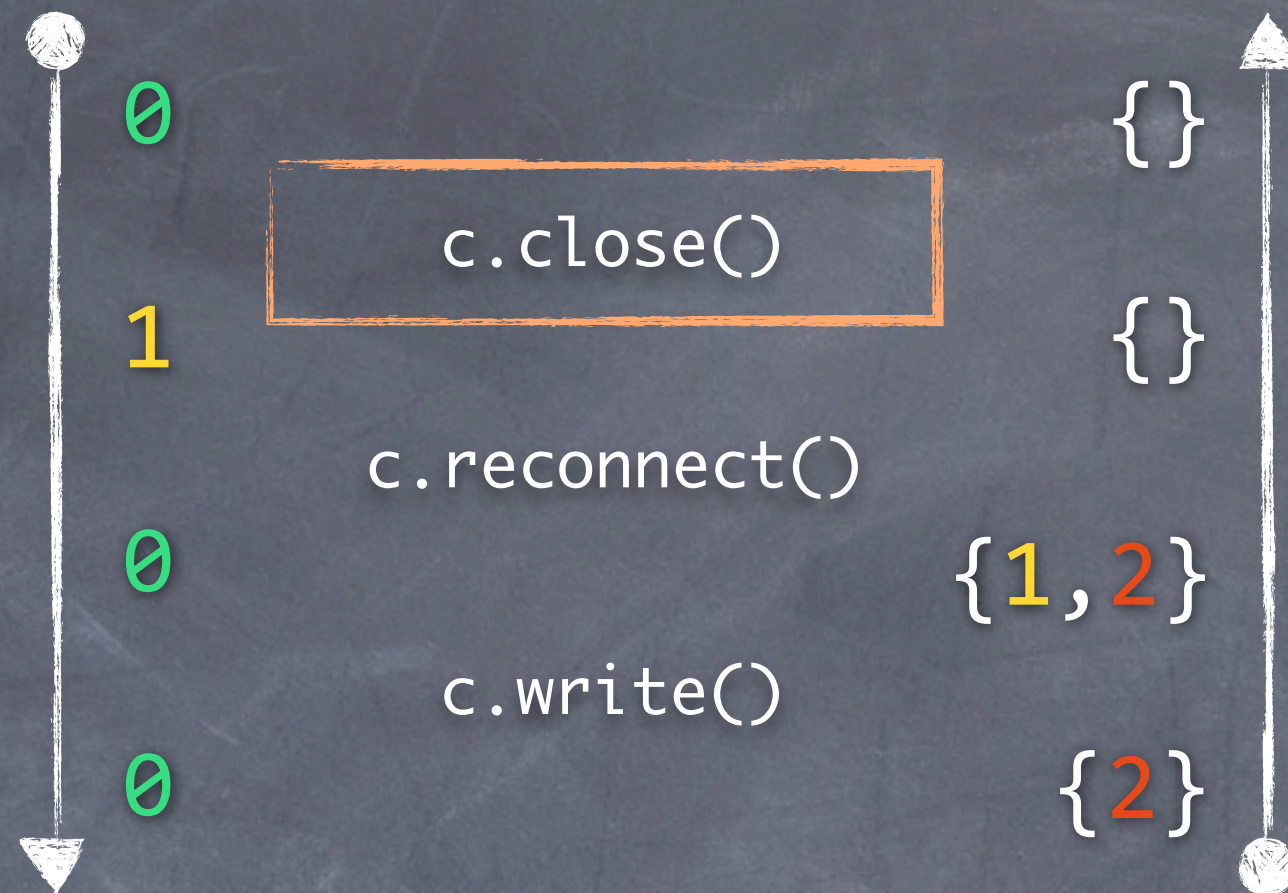


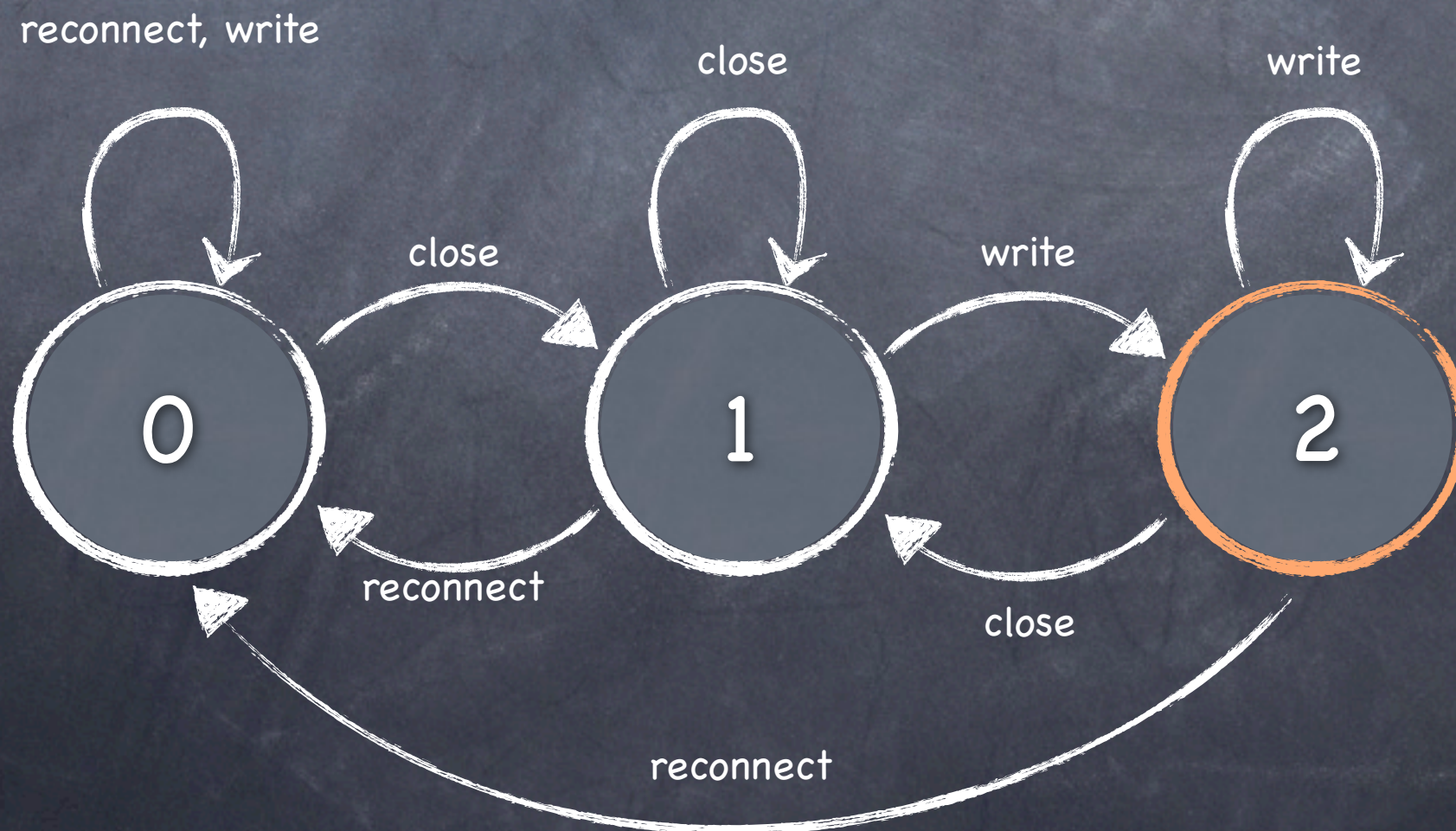
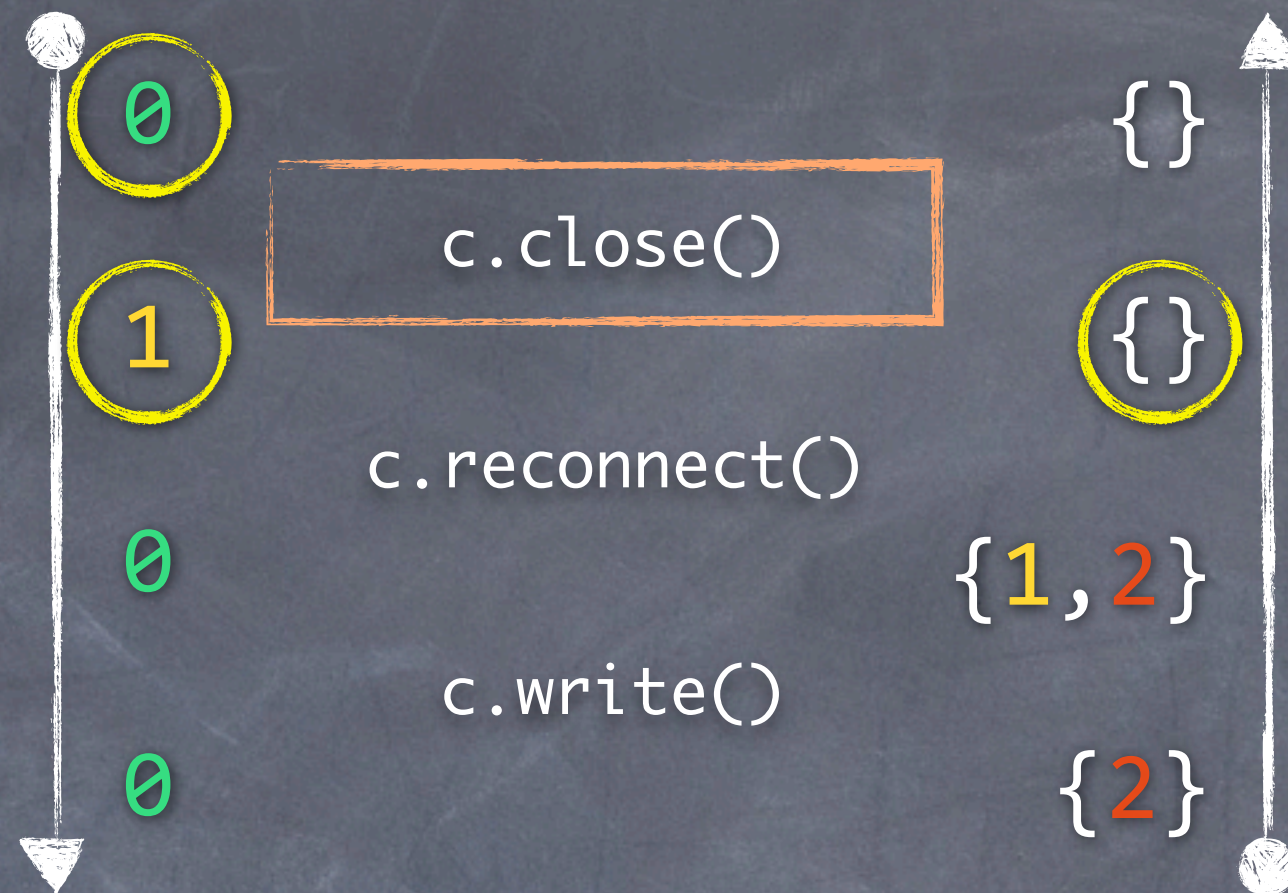


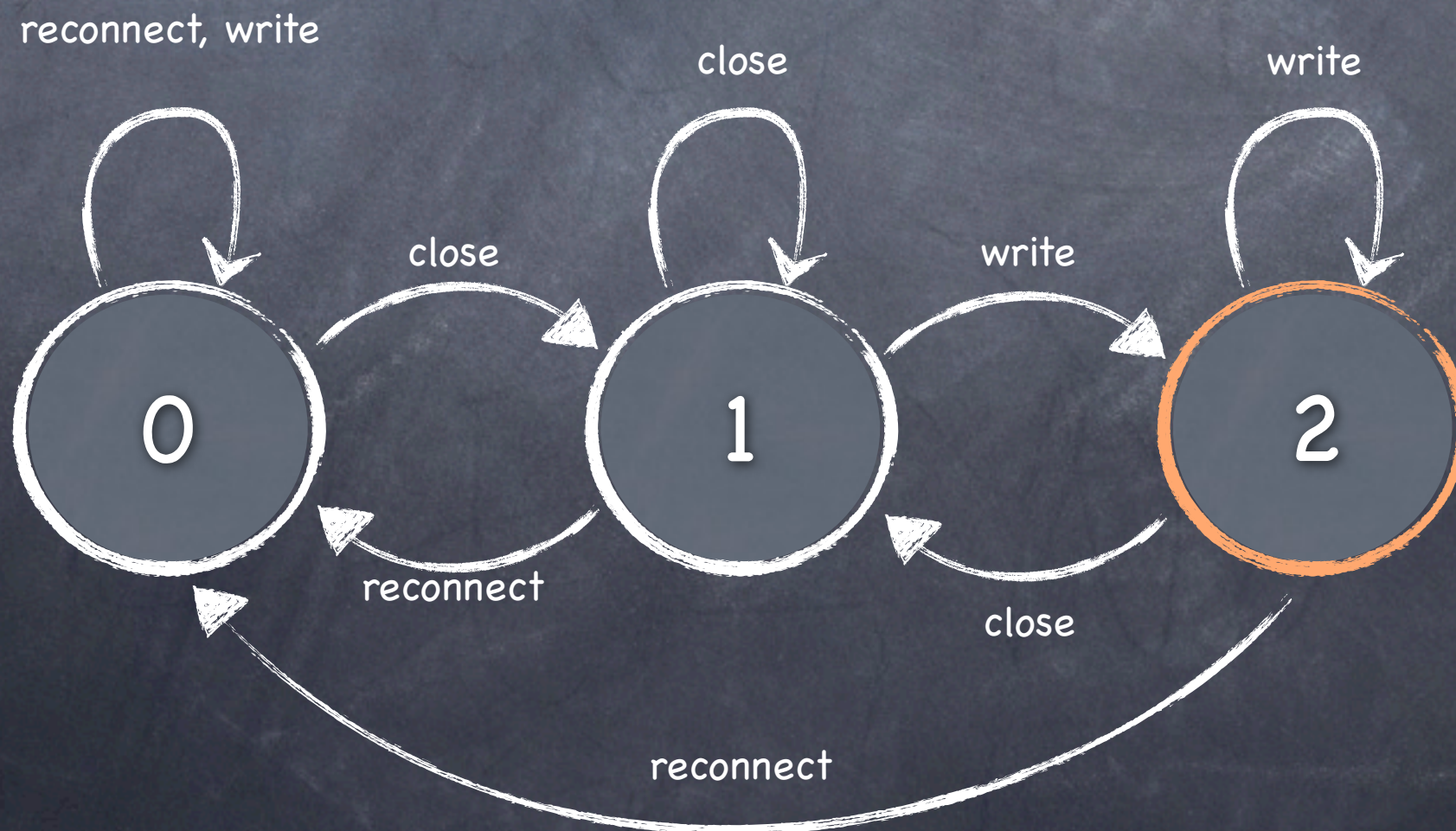
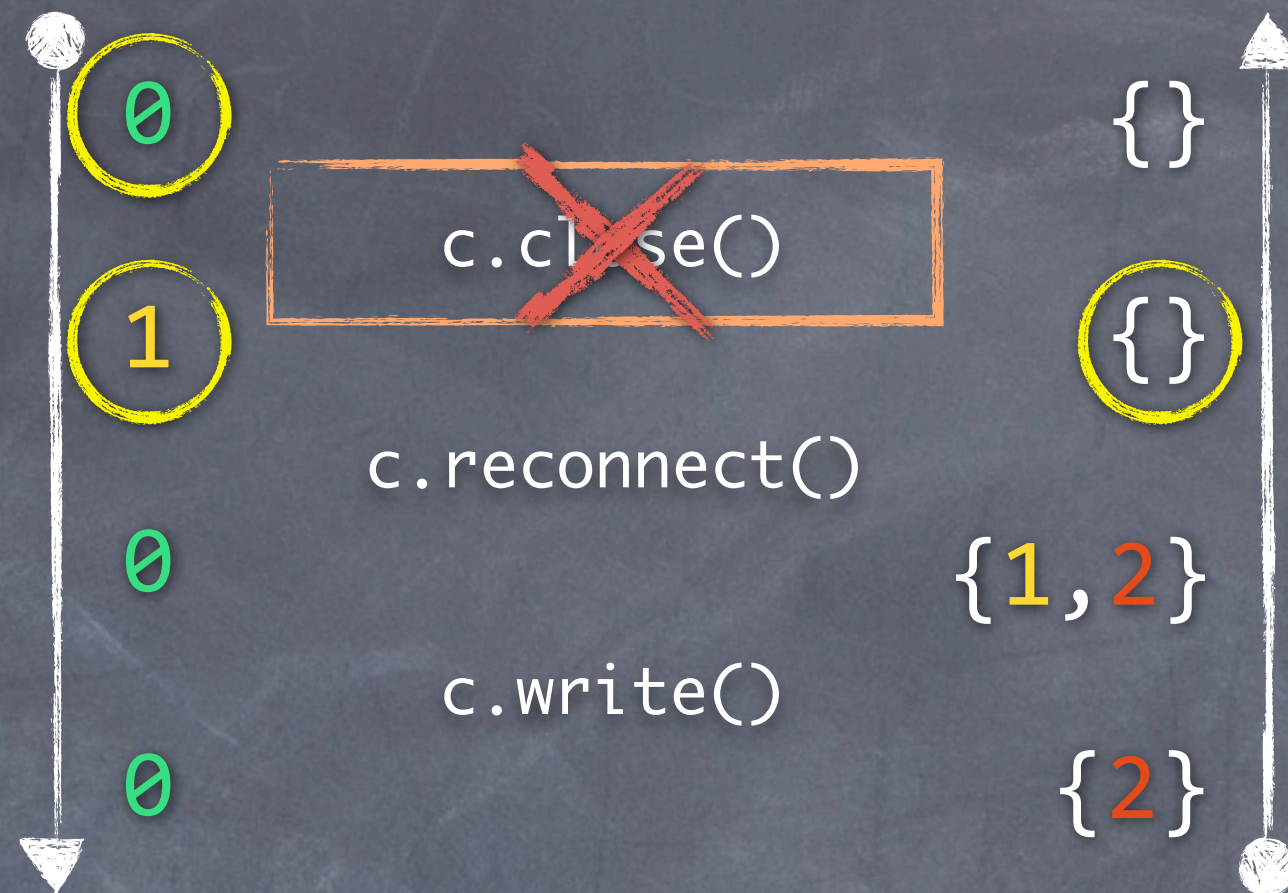


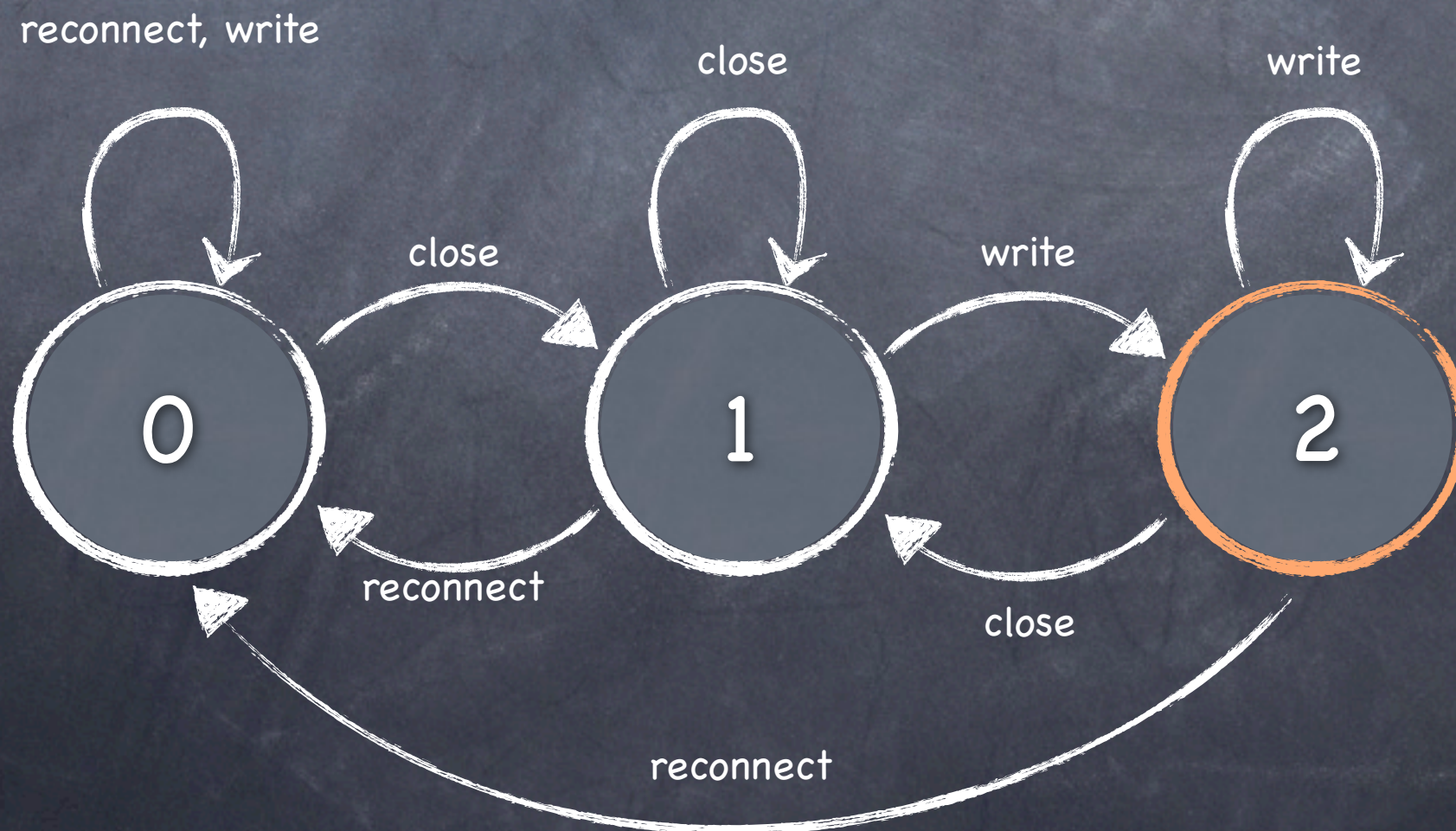
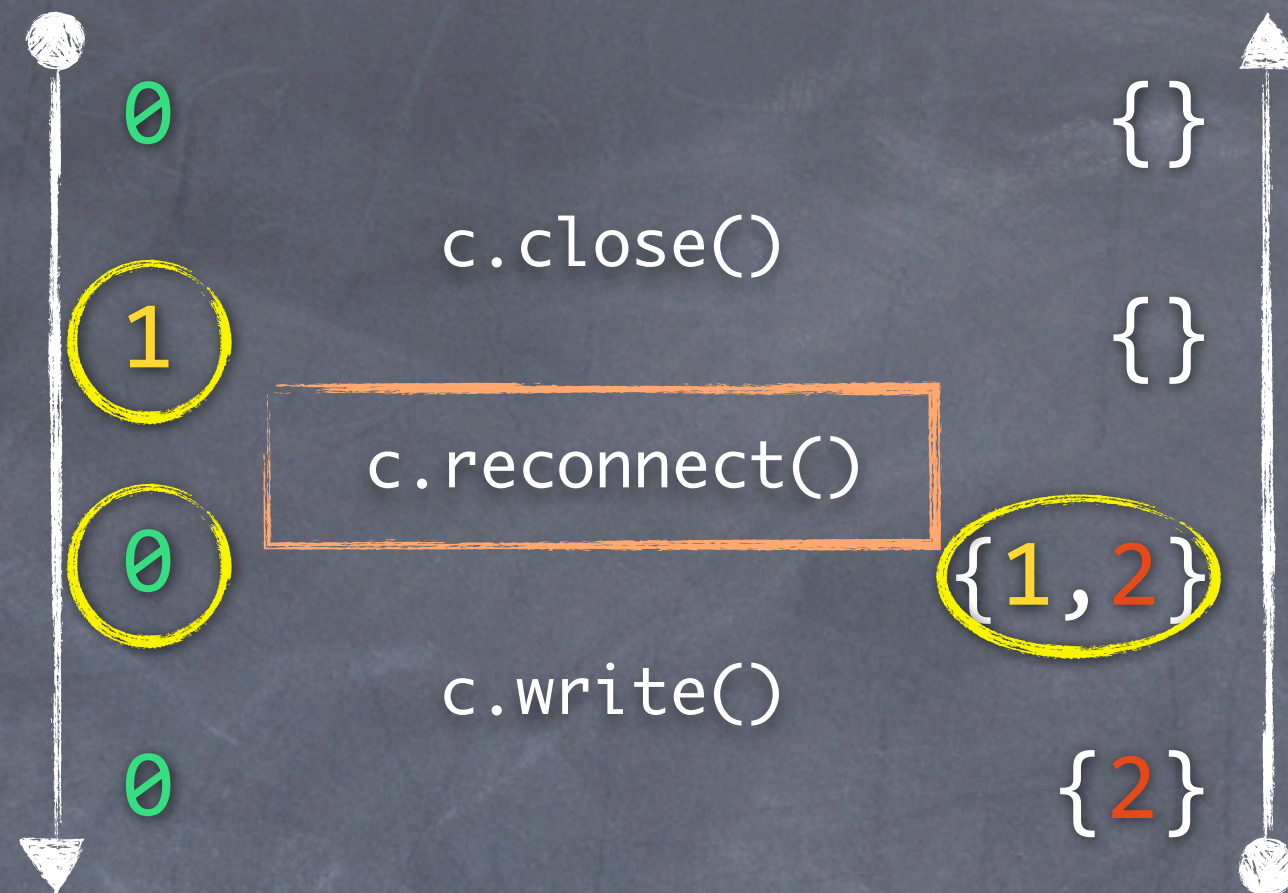


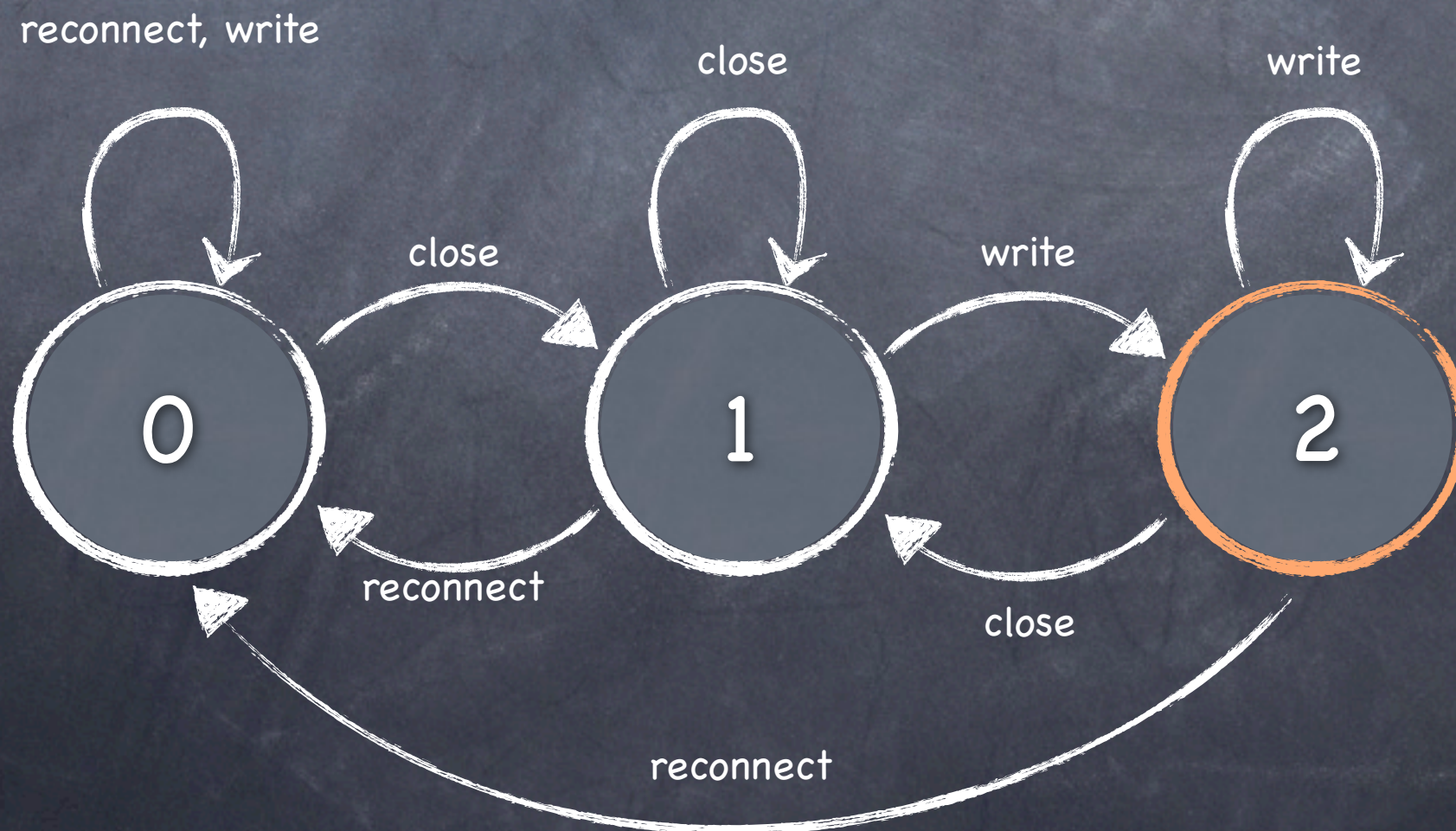
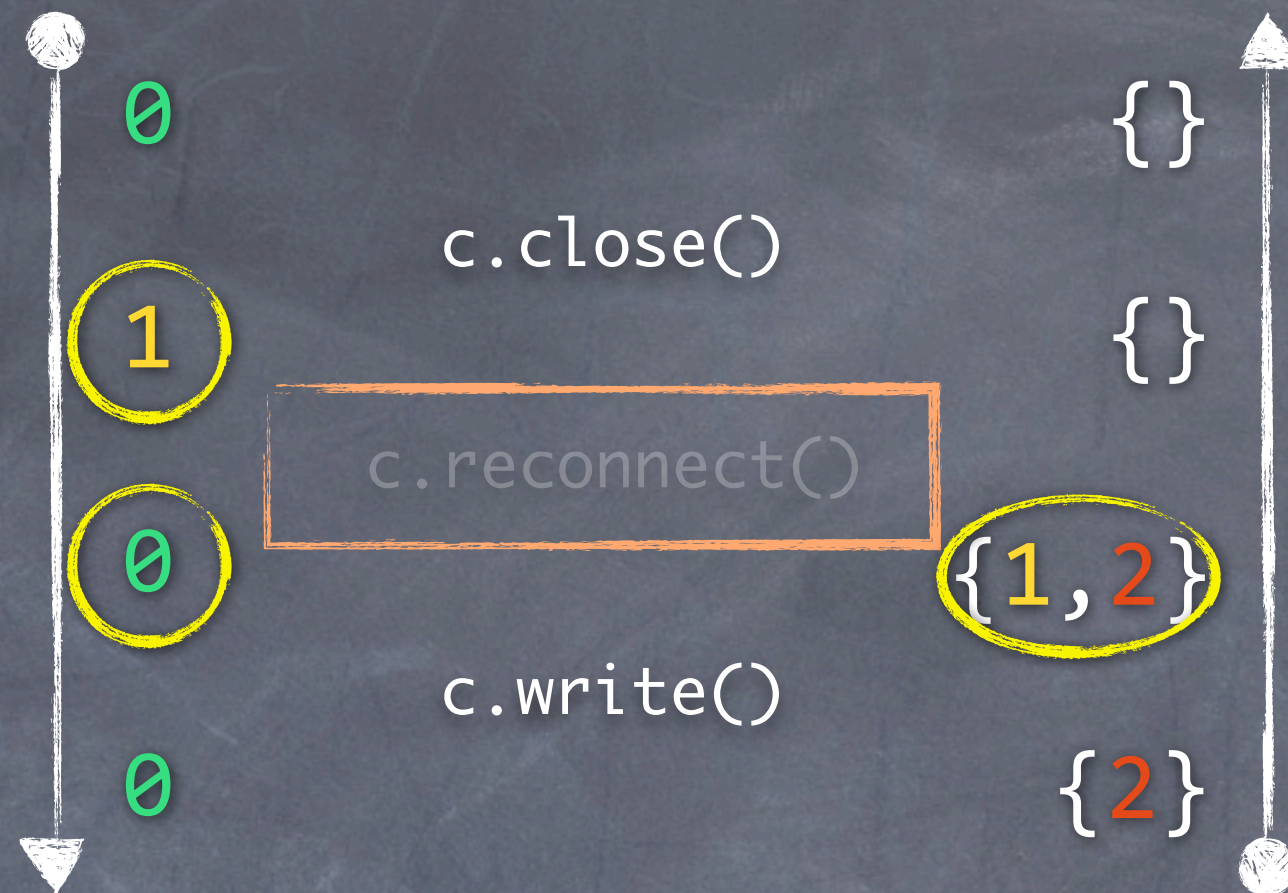


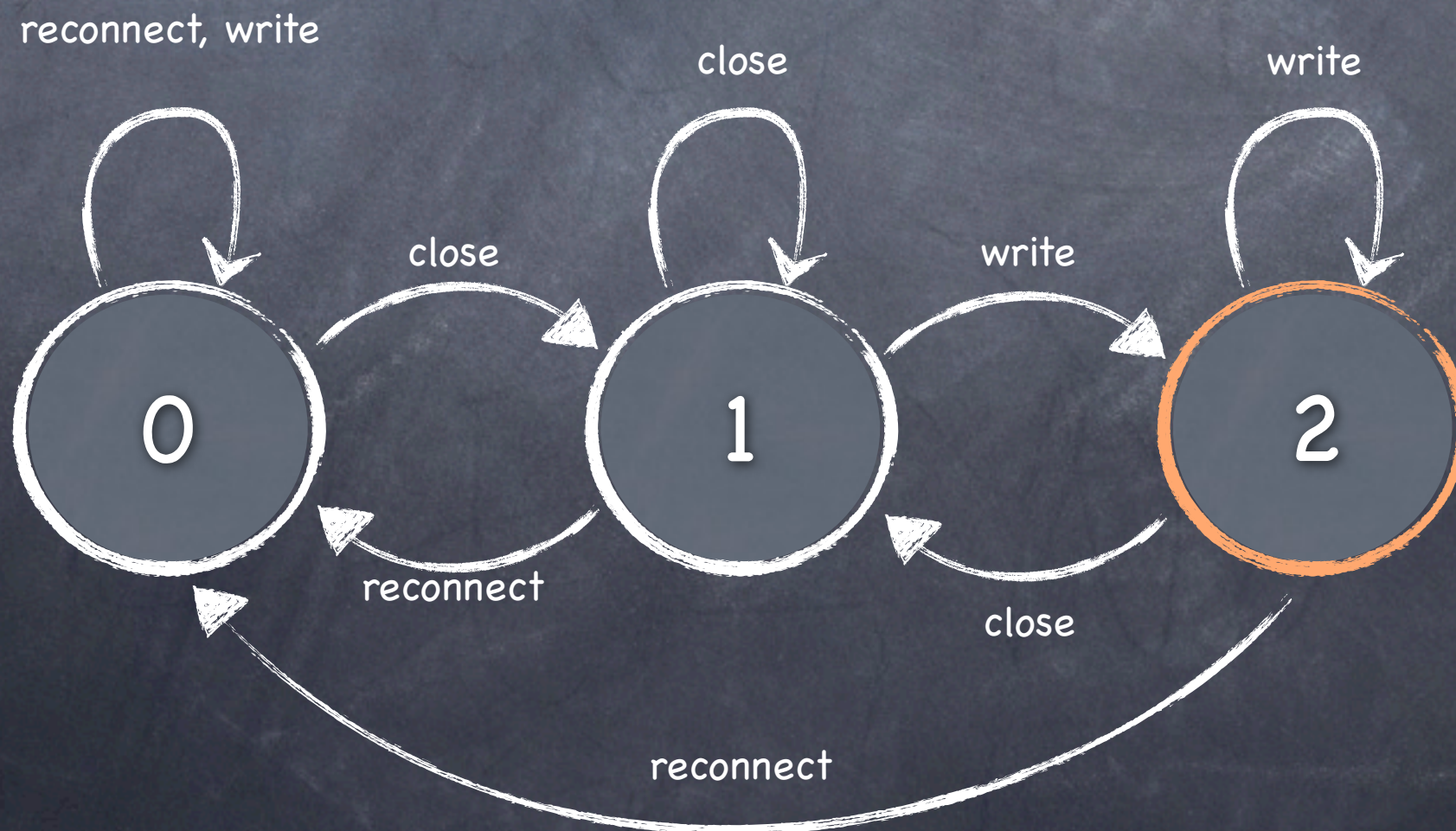
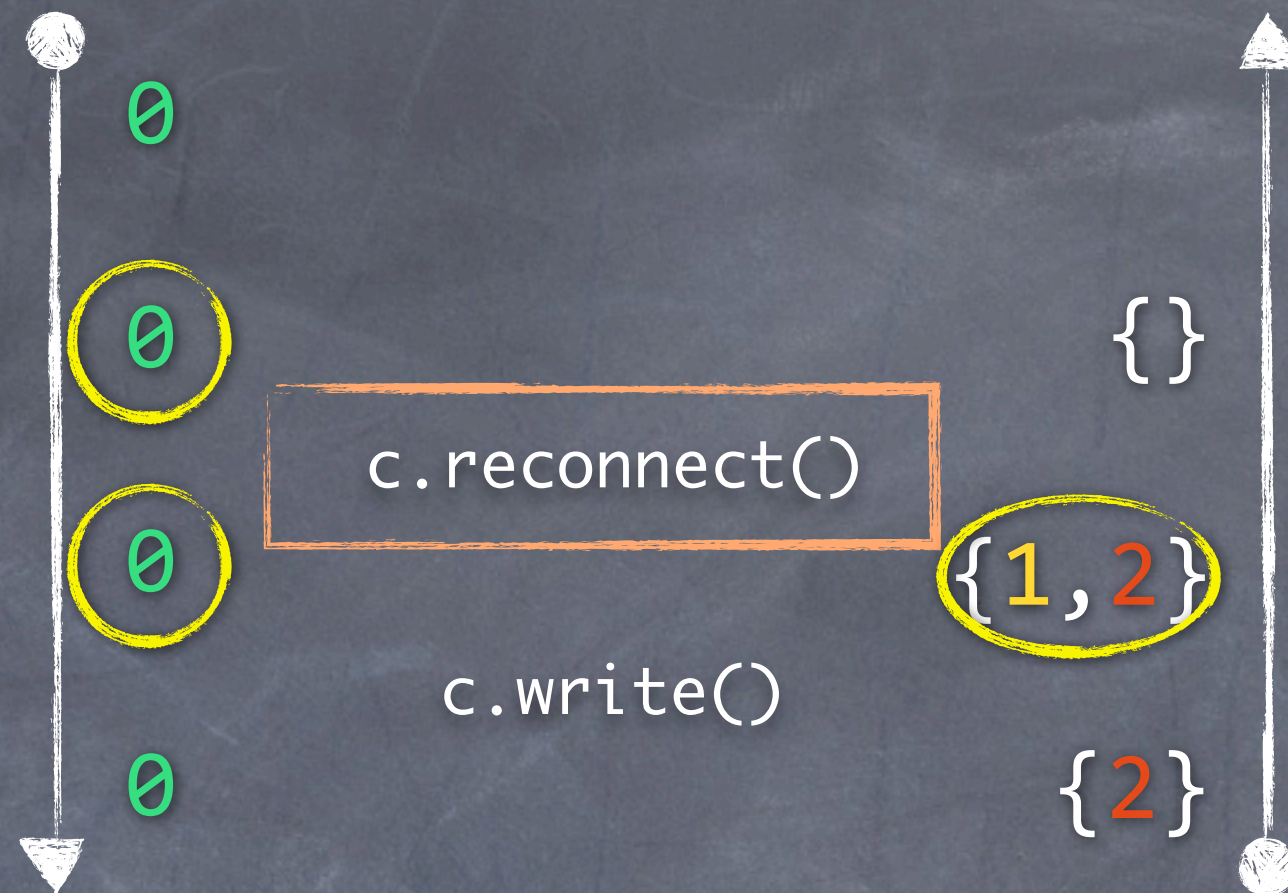


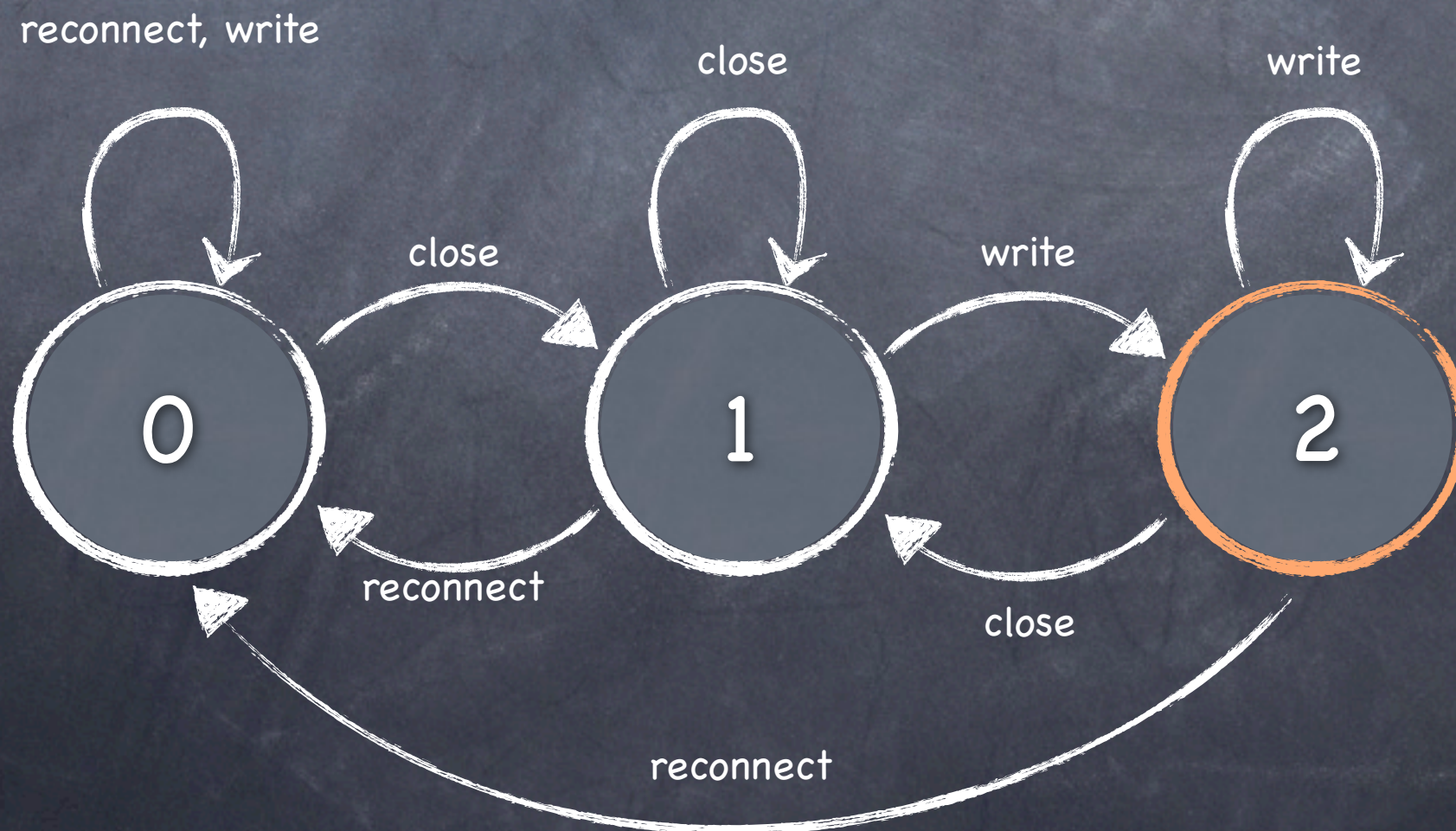
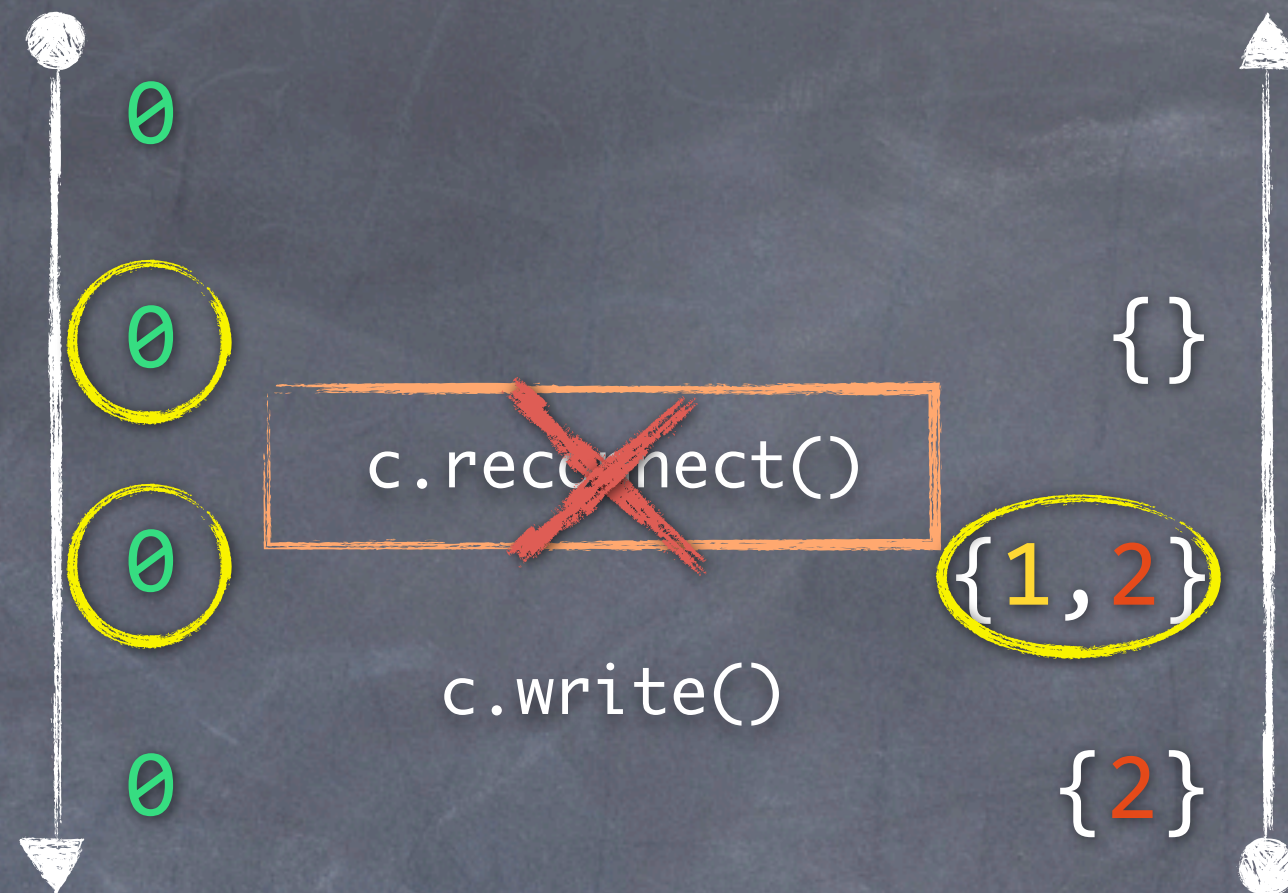


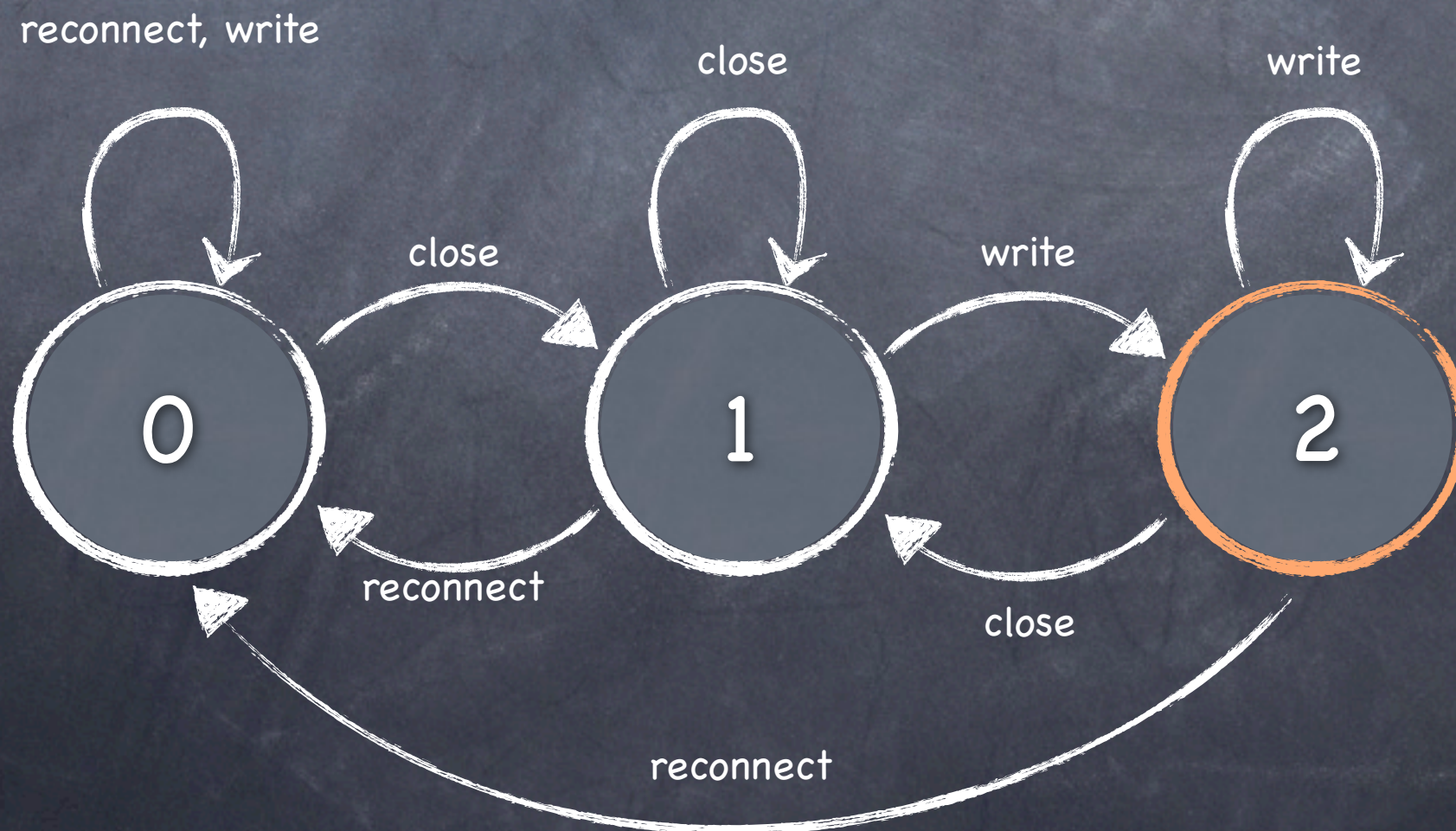
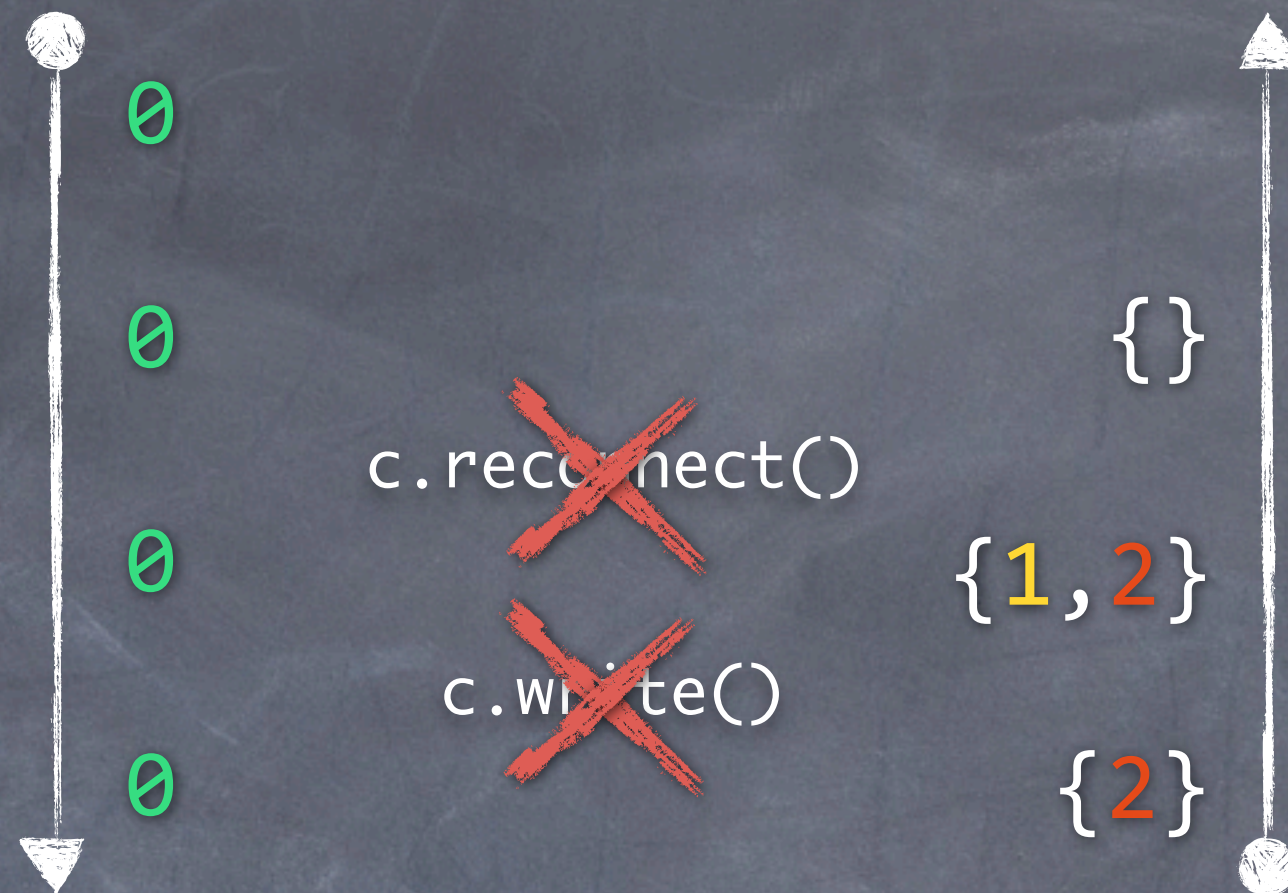


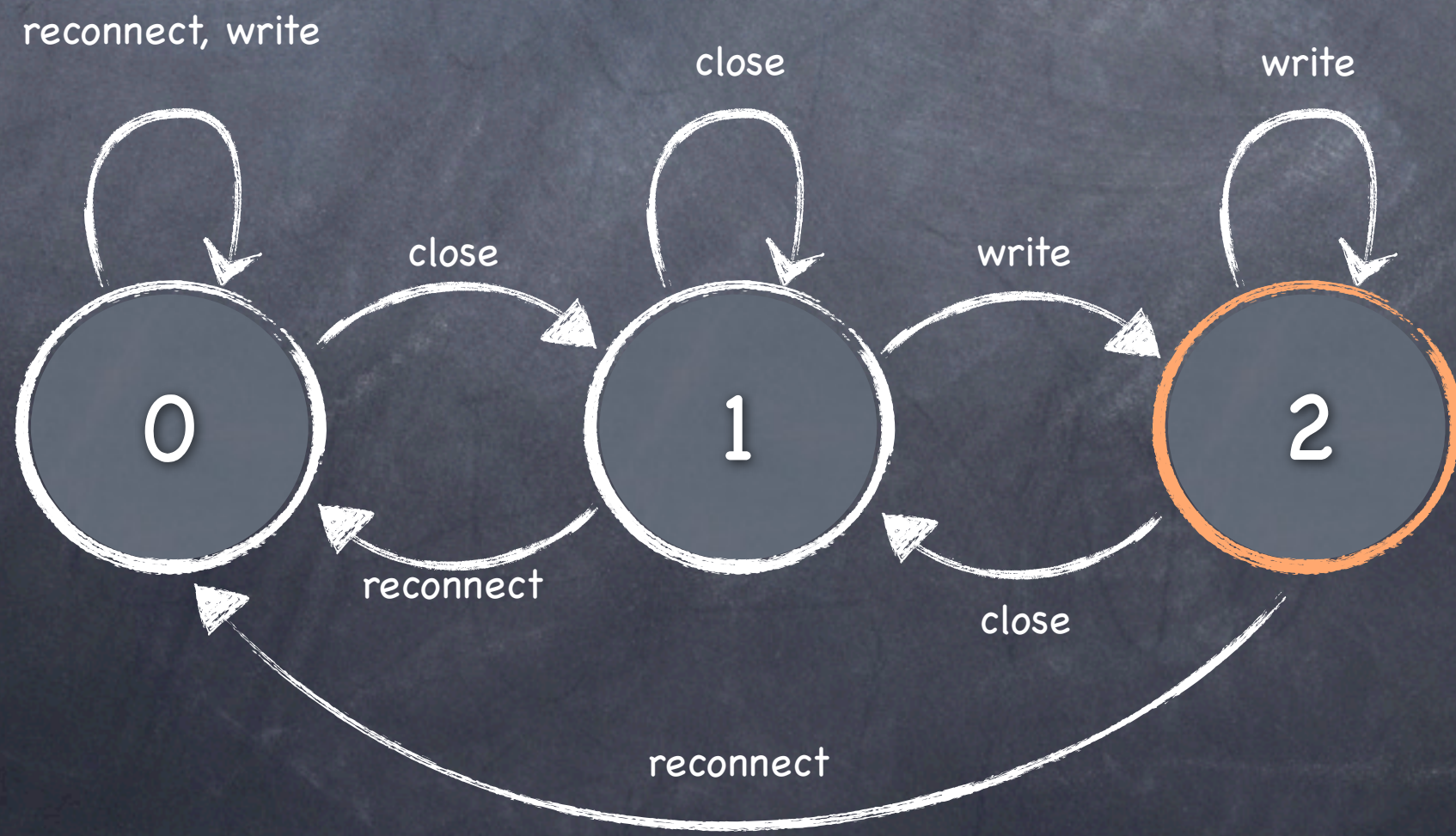




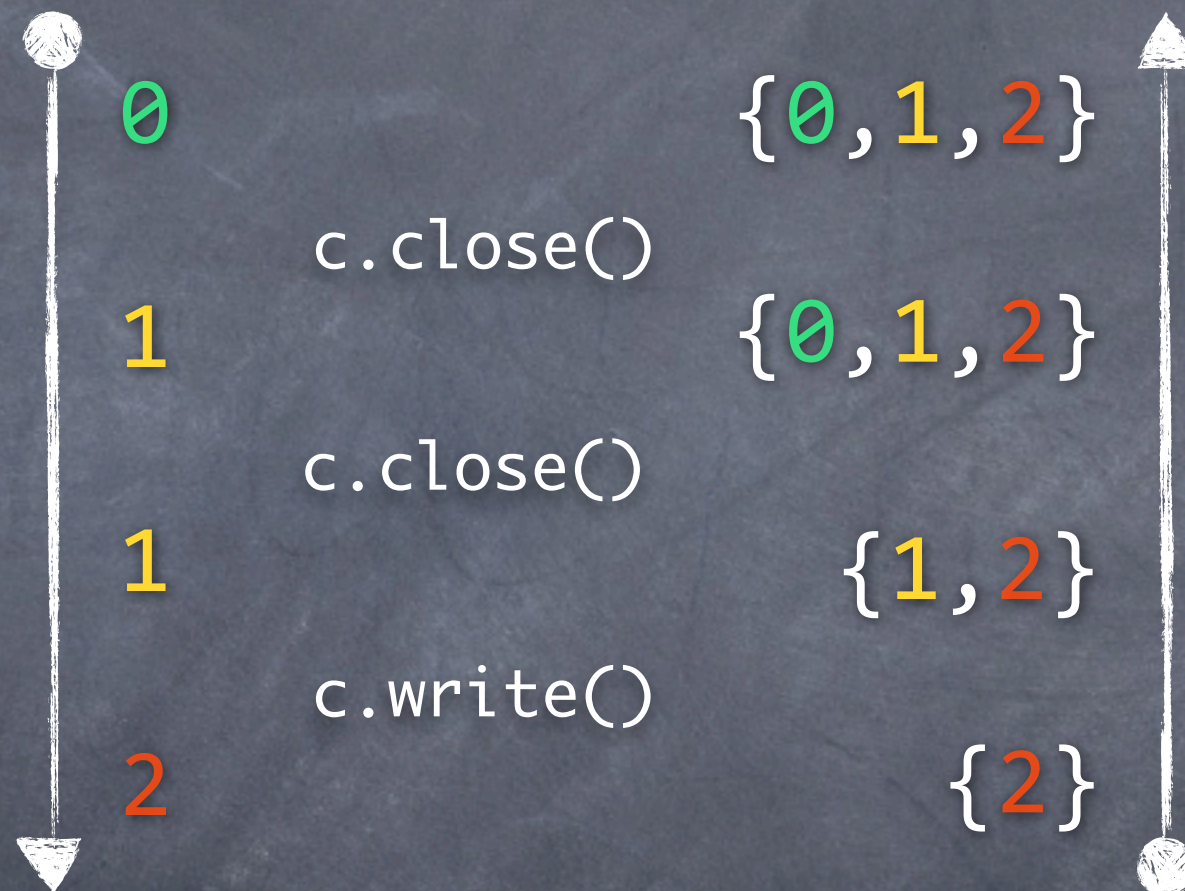




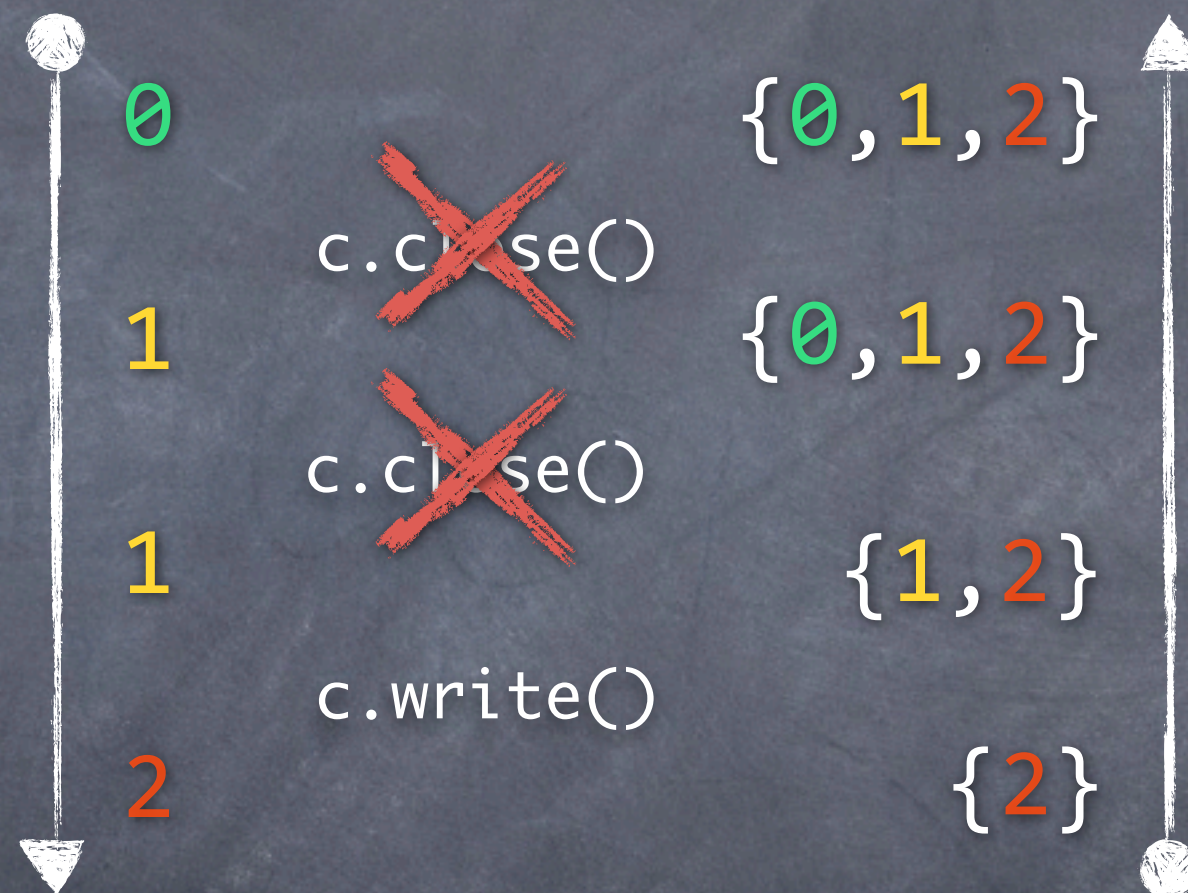




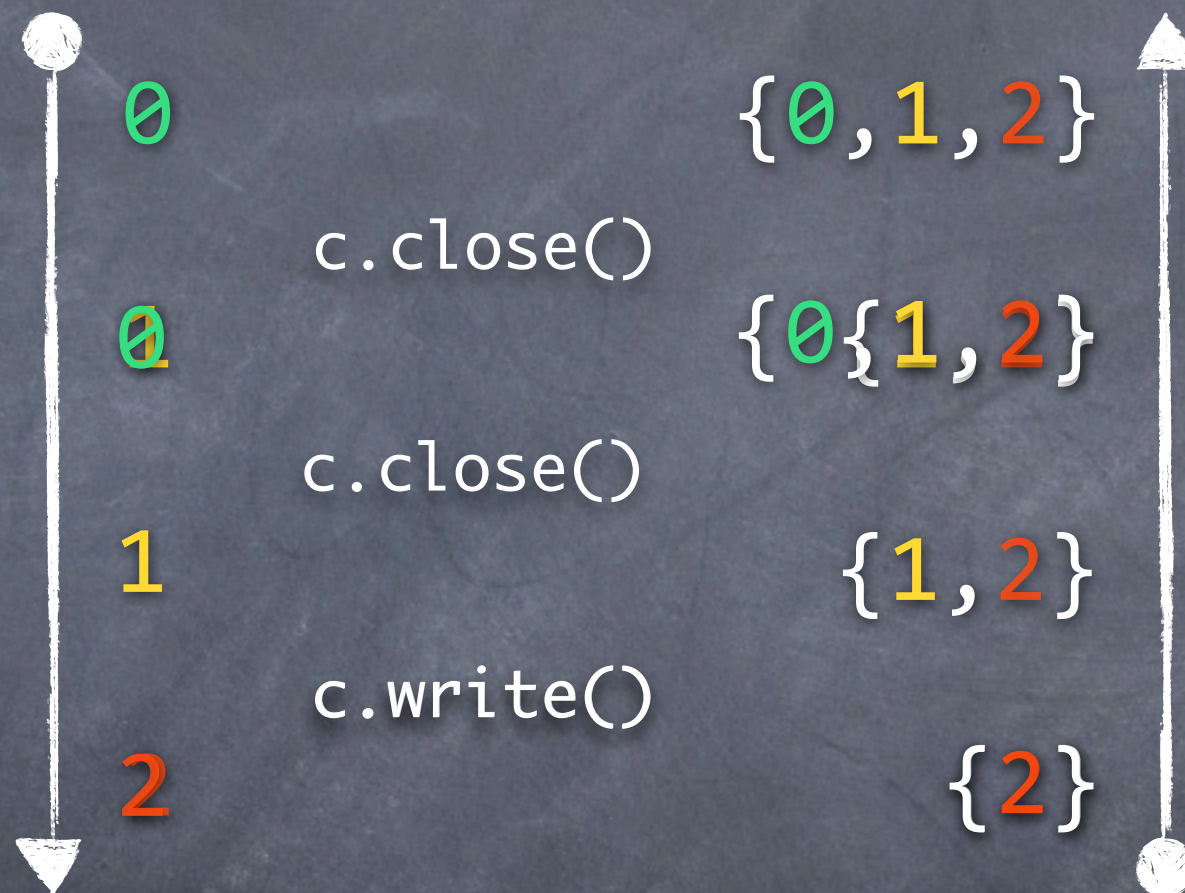
Algorithm is greedy



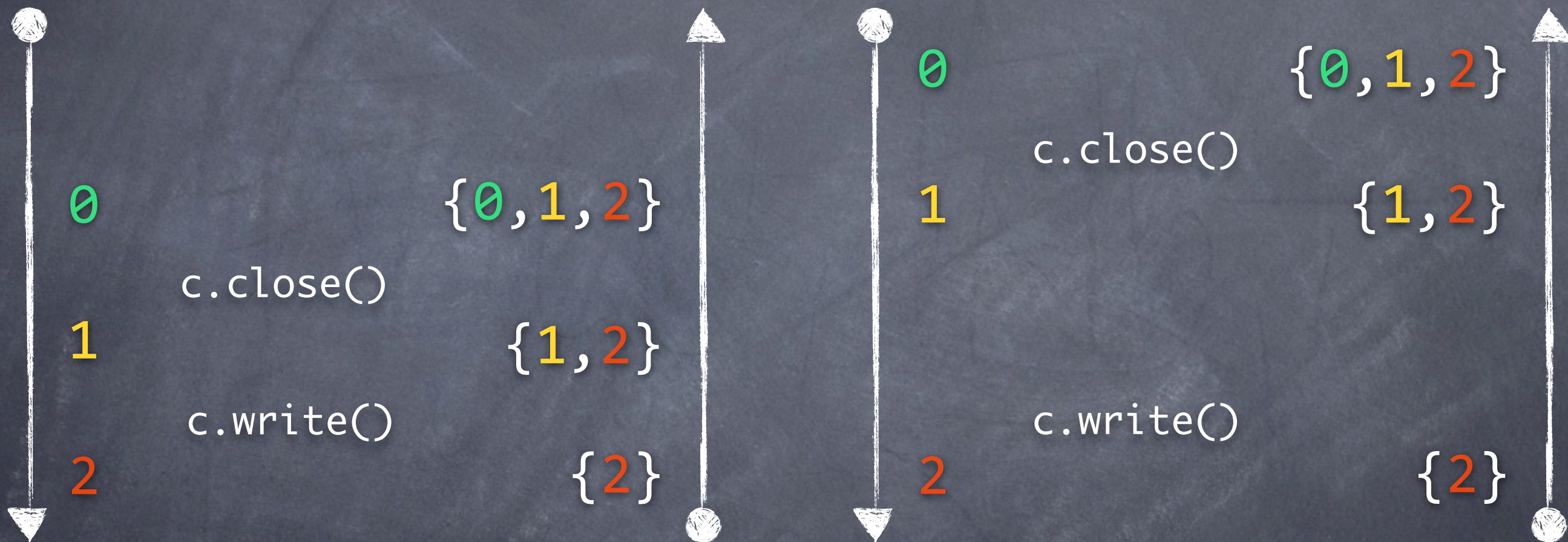
Algorithm is greedy



Algorithm is greedy



Algorithm is greedy




```
c1.close();
```

```
c1.reconnect();
```

```
c1.close();
```

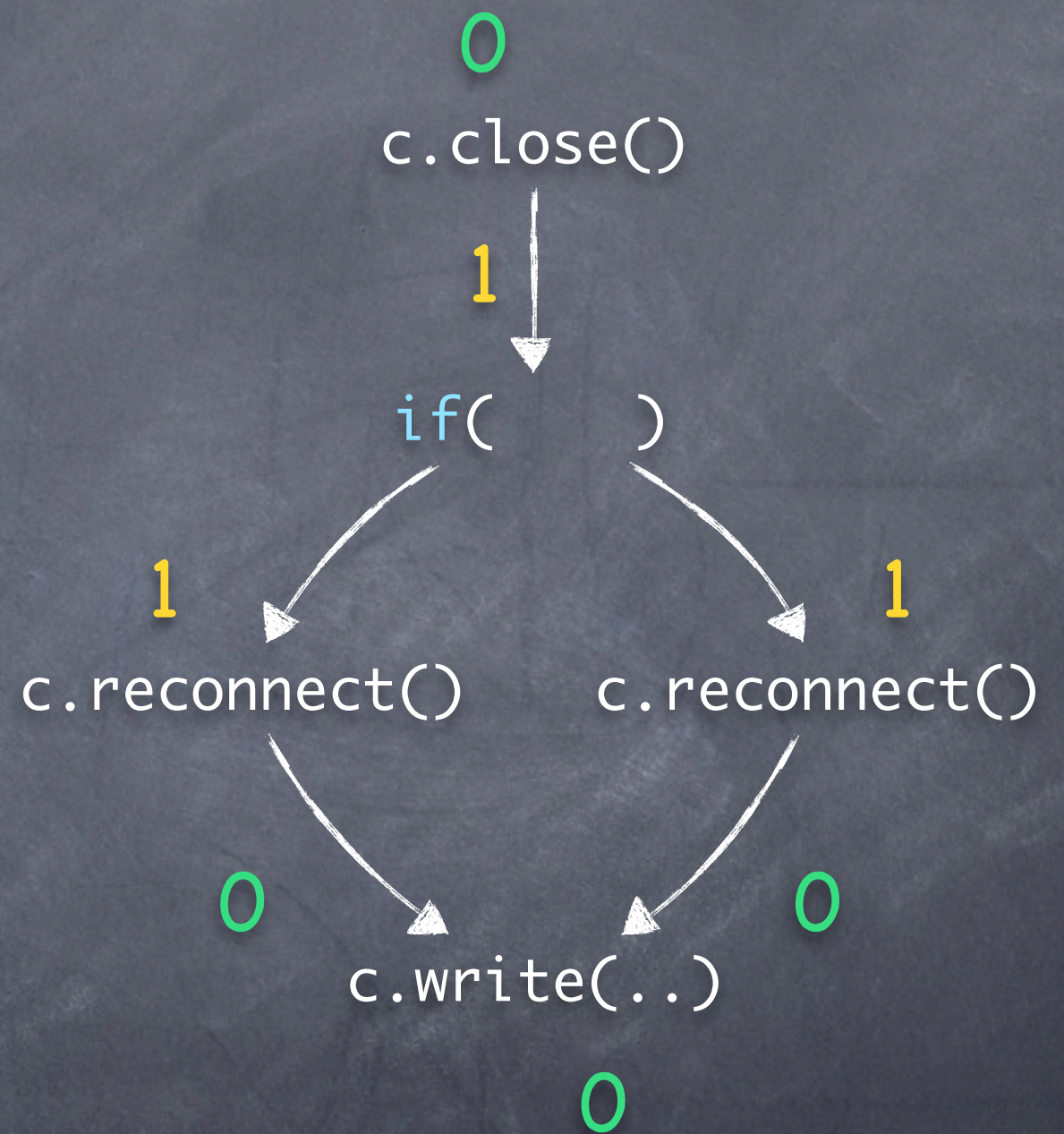
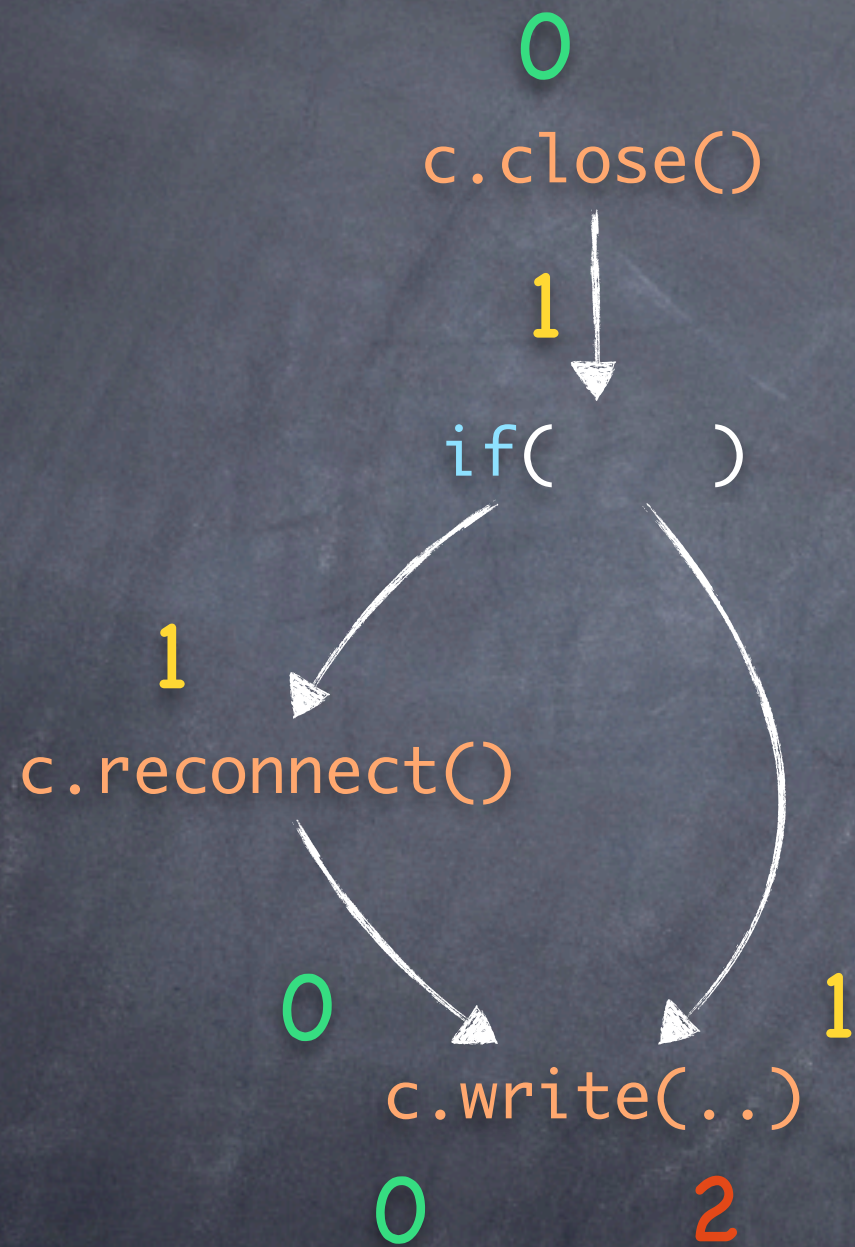
```
c1.close();
```

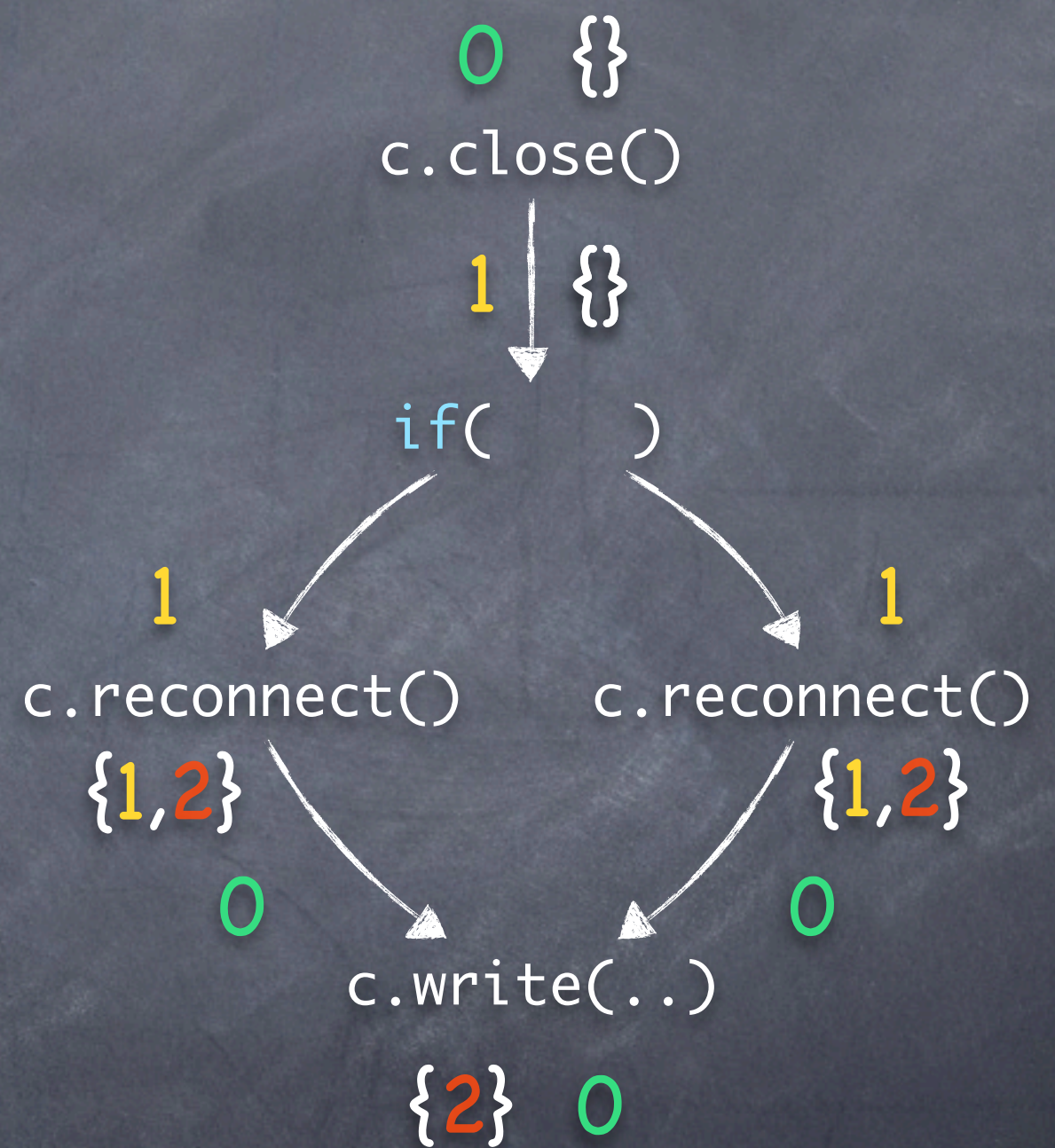
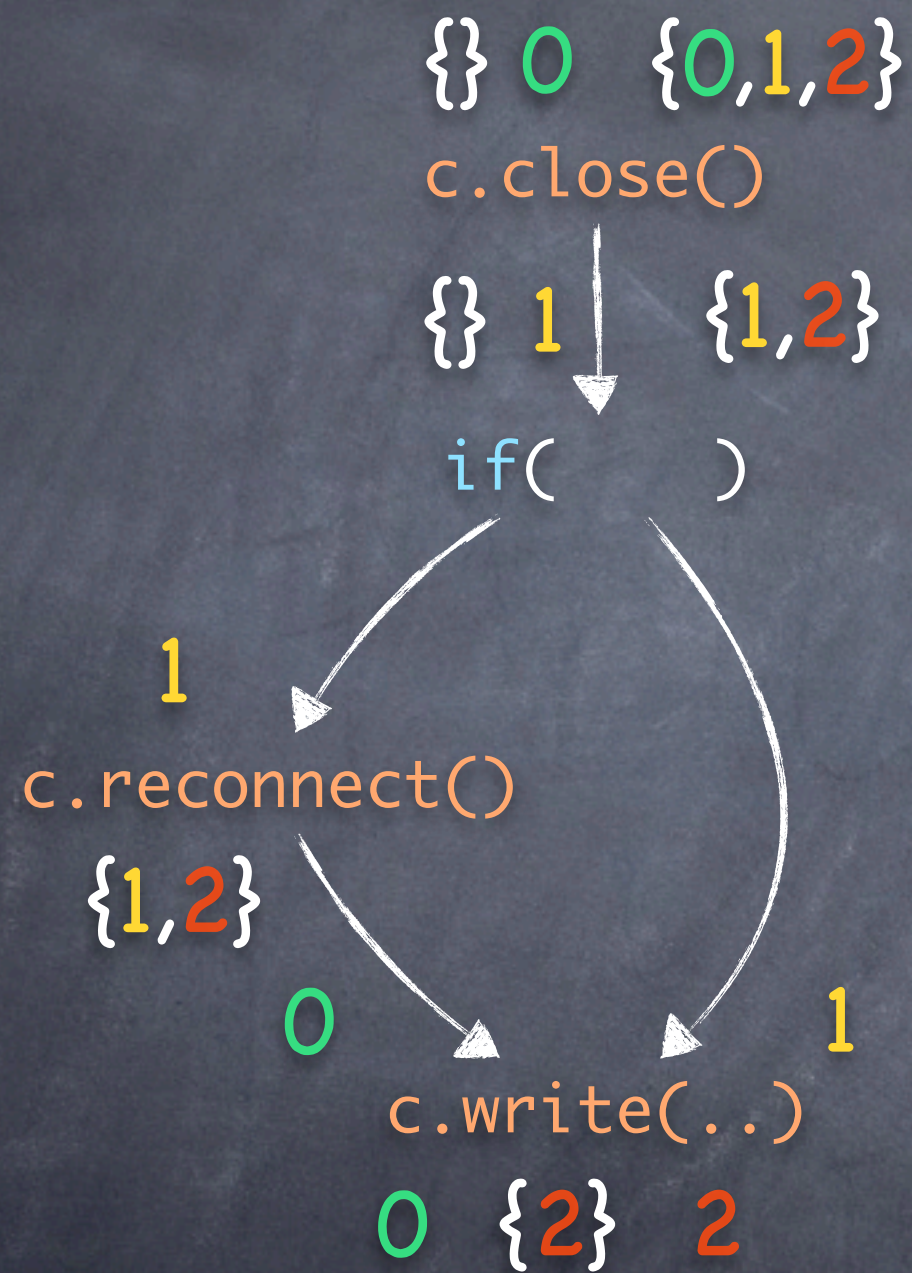
```
c1.write(..);
```

```
c1.close();
```

```
c1.reconnect();
```

```
c1.write(..);
```



`{}` 0 `{0,1,2}`

`c.close()`

`{}` 1 `{1,2}`

`if()`

1

`c.reconnect()`

`{1,2}`

0

`c.write(...)`

0 `{2}` 2

1

0 `{}`

`c.close()`

1 `{}`

`if()`

1

`c.reconnect()`

`{1,2}`

0

`c.write(...)`

`{2}` 0

1

`c.reconnect()`

`{1,2}`

0

`{}` 0 `{0,1,2}`

`c.close()`

`{}` 1 `{1,2}`

`if()`

1

`c.reconnect()`

`{1,2}`

0

`c.write(...)`

0 `{2}` 2

1

0 `{}`
~~`c.close()`~~

1 `{}`

`if()`

1

`c.reconnect()`

`{1,2}`

0

`c.write(...)`

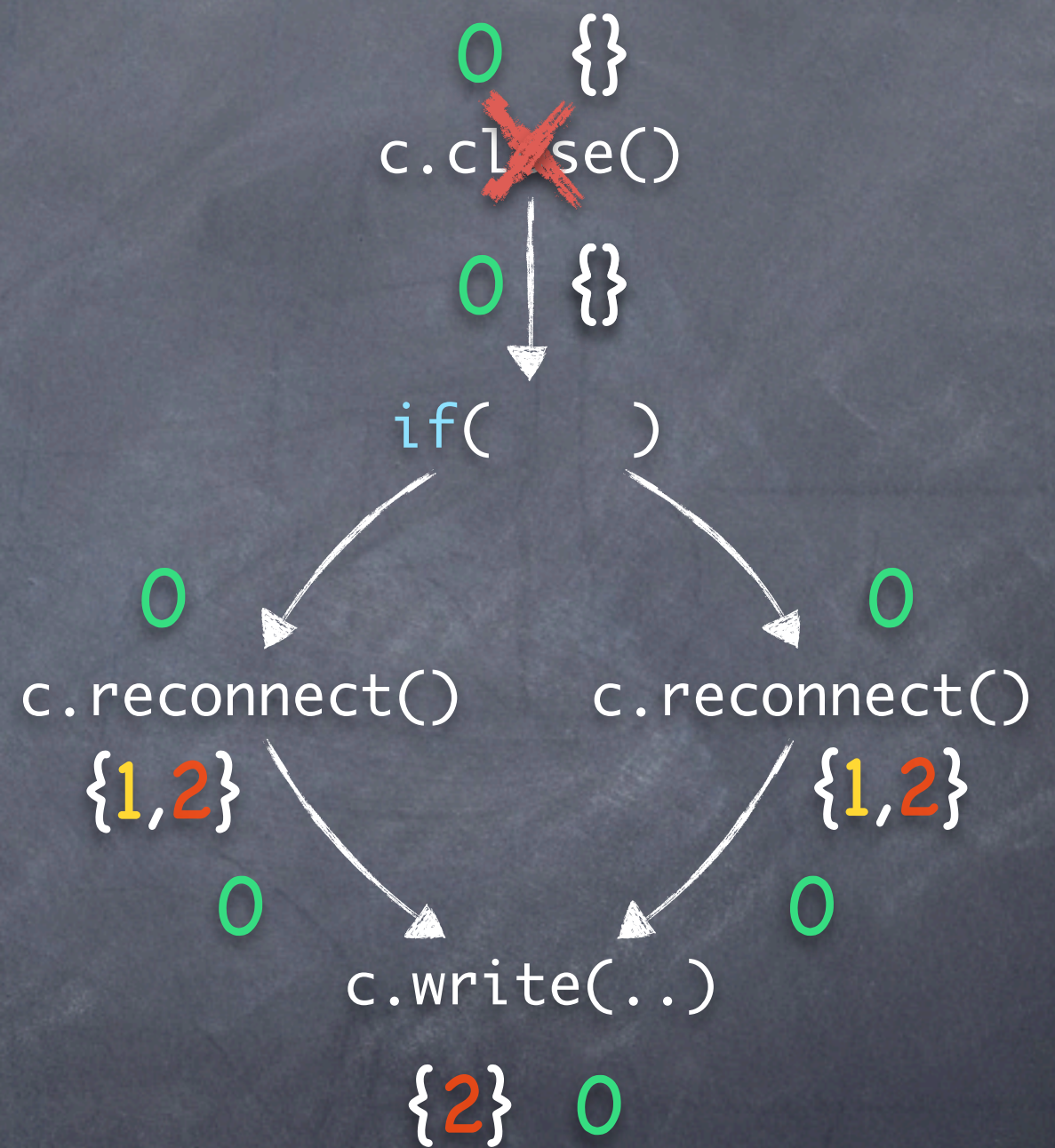
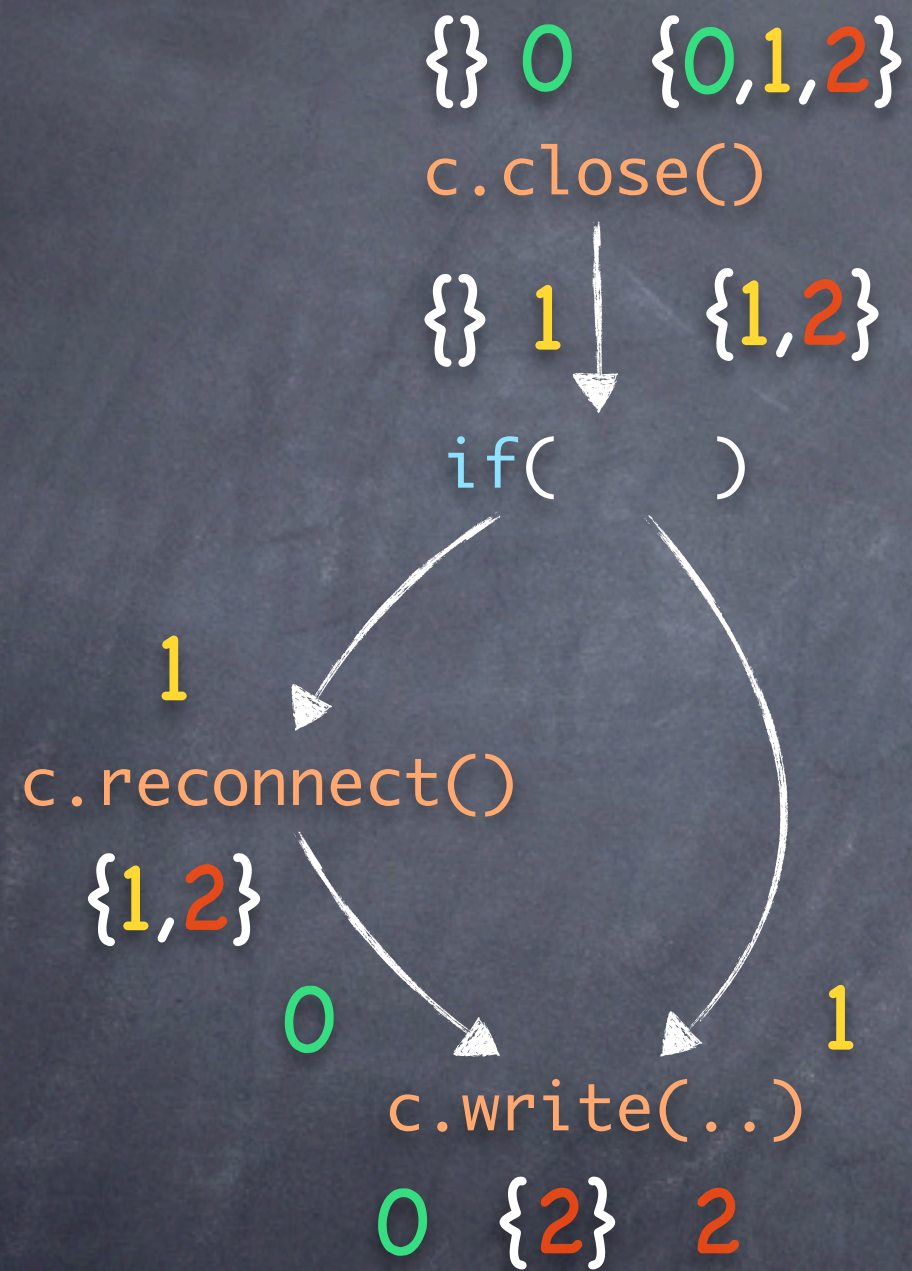
`{2}` 0

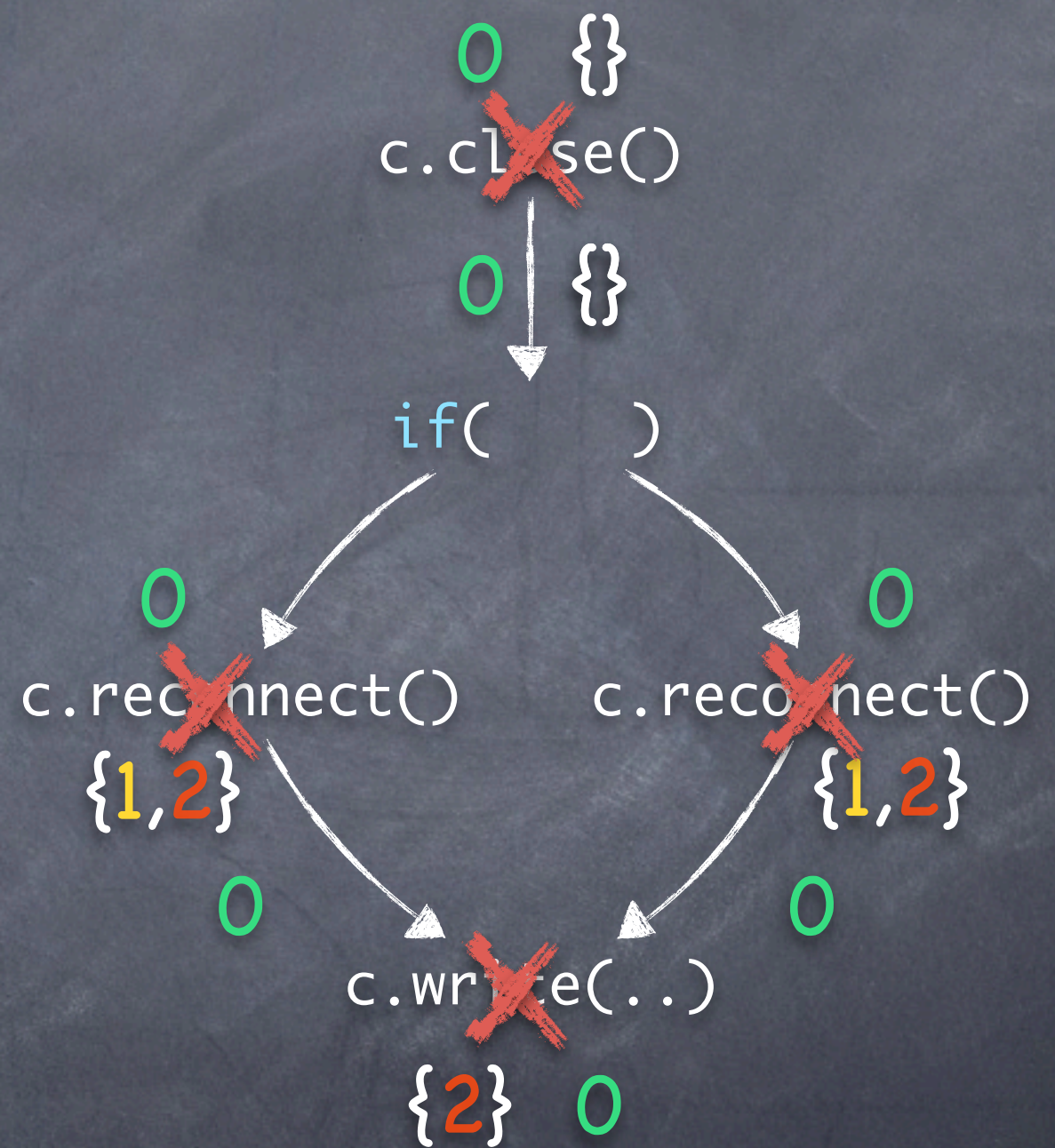
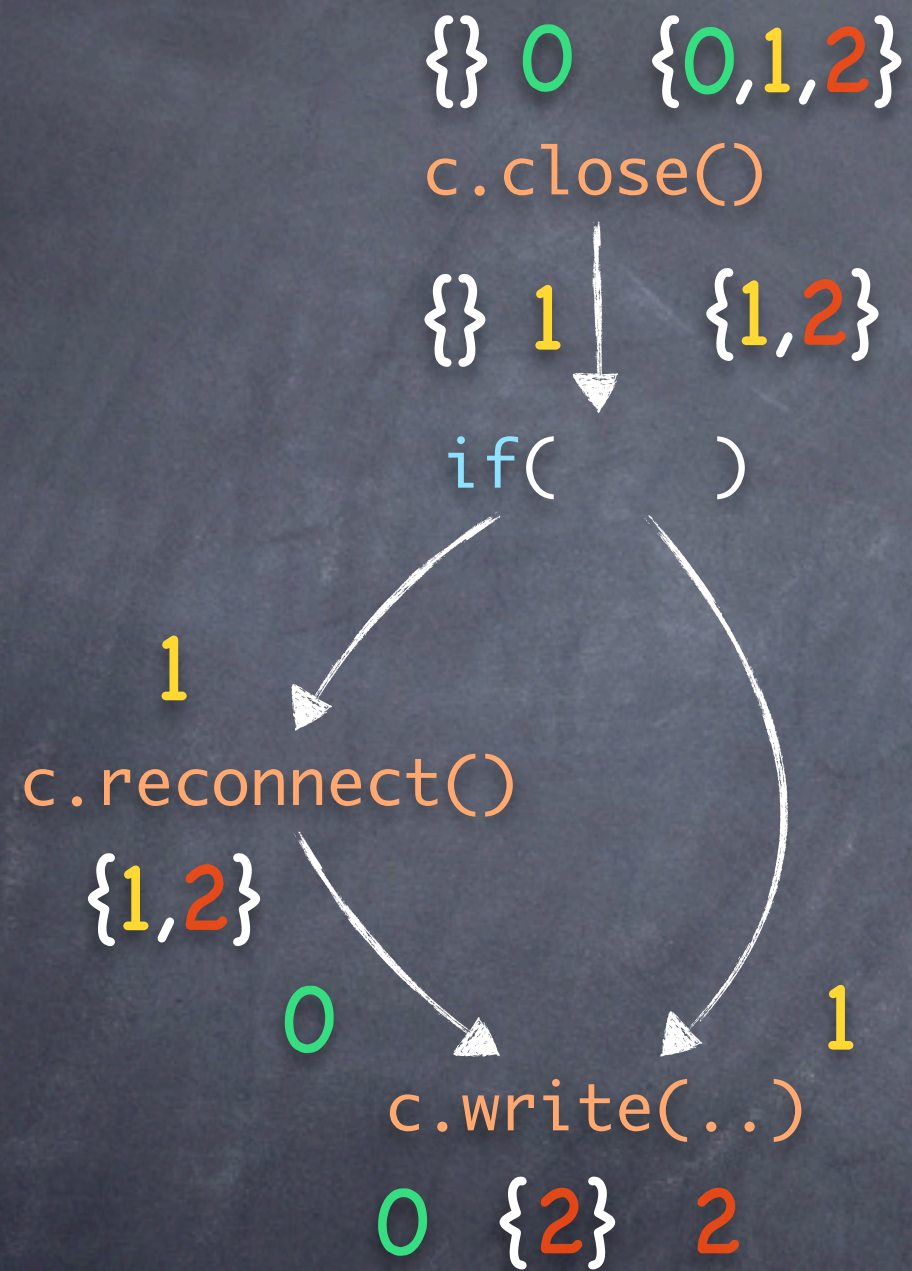
1

`c.reconnect()`

`{1,2}`

0





Tested Properties

ASyncContainsAll	FailSafeIterMap
ASyncIterC	HasNextElem
ASyncIterM	HasNext
FailSafeEnum	LeakingSync
FailSafeEnumHT	Reader
FailSafeIter	Writer

Benchmark programs

antlr	jython
bloat	luindex
chart	lusearch
fop	pmd
hsqldb	xalan

(whole 2006 DaCapo benchmark suite, except eclipse)

Overall success

	antlr	bloat	chart	fop	hsqldb	kython	luindex	lusearch	pmd	xalan
ASyncContainsAll		0/71	0/6			0/31	0/18	0/18	0/10	
ASyncIterC		0/1621	0/498	0/146	0/33	0/128	0/149	0/149	0/671	
ASyncIterM		0/1684	0/507	0/176	0/39	0/138	0/152	0/152	0/718	
FailSafeEnum	0/76	0/3	0/1	6/18	0/120	44/110	0/61	0/61	0/21	0/222
FailSafeEnumHT	26/133	0/102	0/44	0/205	3/114	61/153	0/37	0/37	0/100	0/319
FailSafeIter	0/23	830/1394	149/510	0/288	0/112	112/253	0/217	16/217	287/546	0/158
FailSafeIterMap	0/130	444/1180	49/374	OOME	0/252	133/250	0/136	0/136	204/583	0/540
HasNextElem	0/117	0/4		0/12	0/53	34/64	0/22	0/22	0/11	1/63
HasNext		452/849	48/248	0/72	0/16	24/63	0/74	0/74	184/346	
LeakingSync	0/170	0/1994	0/920	0/2347	0/528	0/1082	0/629	0/629	0/986	0/1005
Reader	0/50	0/7	0/65	0/102	3/1216	4/139	0/226	0/226	0/102	0/106
Writer	35/171	15/563	0/70	0/429	10/1378	0/462	0/146	0/146	0/62	0/751

Residual (hybrid) tpestate analysis



Matthew B. Dwyer and Rahul Purandare.

Residual dynamic tpestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. ASE 2007.

calculates summary transitions for regions of code,
very effective, but only single-object properties

Static analysis for multi-object properties

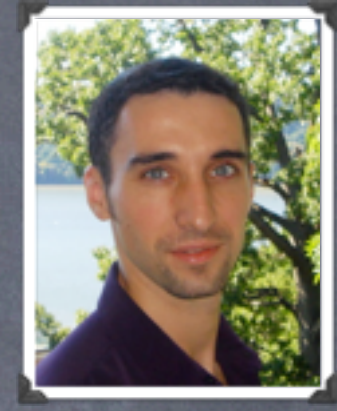


Nomair A. Naeem and Ondrej Lhotak.

Typestate-like analysis of multiple interacting objects. OOPSLA 2008.

supports multi-object properties,
very precise, but unsound as a hybrid analysis

Static typestate analysis



Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay.
Effective typestate verification in the presence of aliasing. ISSTA 2006.

purely static, very effective,
but only single-object properties

Type systems for tpestate



Robert DeLine and Manuel Fähndrich.
Typestates for objects. ECOOP 2004.



Kevin Bierhoff and Jonathan Aldrich.
Modular tpestate checking of aliased objects. OOPSLA '07.

In the paper

We merge condition (2), i.e., not merging state sets, by simply not merging configurations at any time. In particular, we do not merge configurations at control flow merge points: if a conditional expression leads to a configuration c_1 in one branch and to c_2 in another branch then we propagate both c_1 and c_2 after both branches have merged.

For efficiency, we designed the *Top-shadows* Analysis to compute flow-sensitive shadow information only on an intra-procedural level. In particular, our shadow-set analysis is only intra-procedural. One may think that such an analysis would have to be quite expensive. However, before we designed our analysis, we manually investigated the native mutation points that our representative instrumentation causes and found that, in most cases, intra-procedural analysis information was sufficient to determine top shadows. To take care of condition (2), i.e., mutation everywhere in the presence of intra-procedural control flow, we pass this precise intra-procedural information with coarse-grained inter-procedural summary information that can be computed relatively efficiently. The results that we present in Section 3 confirm that this solution is both precise and efficient.

In line with our example from Section 3, the *Top-shadows* Analysis computes for every shadow-bearing method both a forward and a backward-analysis pass. The forward and backward analysis are both instances of a general *worklist* algorithm that we show as Algorithm 1. In this algorithm, the relation $f(x \mapsto g)$ denotes the function that is equal to f in all values x , except for x , in which case it returns g .

$$f(x \mapsto g) = \begin{cases} g & \text{if } x = x \\ f(x) & \text{otherwise} \end{cases}$$

We will explain the internal workings of this algorithm in Section 4.2. First we will explain how we translate the algorithm's parameters.

Algorithm 1: *worklist(method, start_{in}, start_{out}, d)*

```

1: if  $d = \text{method}$ 
2:   return  $\text{start}_{\text{in}}$ 
3: while  $d \neq \text{start}_{\text{in}}$ 
4:    $\text{push\_job}(\text{start}_{\text{in}}, \text{start}_{\text{out}}, d)$ 
5:    $\text{start}_{\text{in}} \leftarrow \text{method}$ 
6:   while  $d \neq \text{start}_{\text{in}}$ 
7:      $\text{start}_{\text{out}} \leftarrow \text{method}$ 
8:      $\text{start}_{\text{in}} \leftarrow \text{method}$ 
9:      $\text{start}_{\text{out}} \leftarrow \text{method}$ 
10:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
11:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
12:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
13:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
14:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
15:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
16:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
17:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
18:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
19:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
20:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
21:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
22:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
23:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
24:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
25:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
26:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
27:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
28:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
29:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
30:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
31:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
32:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
33:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
34:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
35:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
36:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
37:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
38:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
39:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
40:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
41:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
42:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
43:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
44:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
45:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
46:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
47:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
48:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
49:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
50:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
51:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
52:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
53:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
54:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
55:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
56:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
57:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
58:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
59:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
60:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
61:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
62:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
63:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
64:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
65:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
66:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
67:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
68:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
69:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
70:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
71:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
72:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
73:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
74:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
75:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
76:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
77:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
78:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
79:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
80:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
81:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
82:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
83:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
84:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
85:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
86:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
87:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
88:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
89:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
90:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
91:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
92:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
93:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
94:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
95:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
96:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
97:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
98:     $\text{start}_{\text{in}} \leftarrow \text{method}$ 
99:     $\text{start}_{\text{out}} \leftarrow \text{method}$ 
100:    $\text{start}_{\text{in}} \leftarrow \text{method}$ 

```

4.1 Initialising the worklist algorithm

The initialization depends on whether we perform a forward or backward analysis. The first argument to the algorithm is a set initial "jobs". A job is a pair $(\text{start}, \text{end})$, denoting a statement start with a set of configurations, i.e., with a set of sets of activation states.

For the forward analysis we must initialize the algorithm with configurations that model all possible control flow that could have occurred before entering the current method m , but without having occurred already. Later in this section, we explain how we actually handle m -recursion of m as well, though a special recursive function rec_{in} .

If p is the initial state of M_{in} and $\text{shadow}(\text{shadow})$ is the set of shadows outside of m , then the set of initial jobs associated with m 's entry statement s of configurations that are reachable from the initial configuration (p) by executing any of the shadows shadow is:

$$\text{init}_{\text{in}} = \{ \langle s, \text{reaching}(p, \{ \langle s, \text{shadow}(\text{shadow}) \rangle \}) \}$$

Here we define for every set s of configurations and m of shadows $\text{reaching}(s, m)$ as the smallest set of configurations that leads to s .

- $s \in \text{reaching}(s, m)$, and
- $t \in \text{reaching}(s, m)$ if $t \in \text{reaching}(s, m)$.

Note that this third-party computation is flow-insensitive: we do not consider the order in which the shadows in m could occur. This allows us to compute reaching very efficiently. For the backward analysis, we generate initial configurations in a similar but not identical way. A visiting trace can only start at the beginning of the program, but it can end (ending a visitation) in the current method to itself, or in another method (either with or without a call stack or not). We generate initial jobs to cover these three cases. Due to space limitations we give a formal definition in the accompanying document [7, Section 3.1.1-1.4].

The second and third arguments to the algorithm, start_{in} and $\text{start}_{\text{out}}$, are recursive functions that model the point-to control flow within the current method m , respectively outside of m . Figure 3 visualizes both recursive functions. We show the current method m , here foo , as a box. The method contains two inner statements. The first statement resembles a potentially-recursive call (including mutually-recursive calls), the second one a possibly non-recursive call. The dashed arrows denote the recursive function rec_{in} , which is given by m 's control flow graph. This graph allows us to actually handle control flow caused by conditionals, loops and exceptions. Solid arrows show the second, intra-procedural recursive function, rec_{out} . During the execution of m , inside reaching within m our entry methods to be called. These calls either can or cannot transitively perform a recursive call back into m . When the call must be recursive, then configurations that we computed for the call site can reach m 's entry statement, see rec_{in} (1). Conversely, for configurations that we computed for any of m 's exit statements, we need to propagate these configurations back to any potential recursive call within m , see (2). At compile-time, we can usually only determine that a method call may be recursive, not that it must be. Hence, we also need to propagate configurations from the call site to also into rec_{out} , see (3). The calls that we possibly not recursive (see

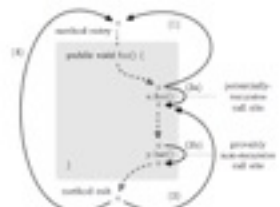


Figure 3: Function rec_{in} (dashed) & rec_{out} (solid)

determined by a call graph), it suffices to propagate configurations past the call site itself, see (3). Lastly, we need to take into account the case in which method m returns (either normally or by throwing an exception: rec_{out} handles both cases), and then to recursion. To model this case we propagate configurations from m 's exit(s) to its entry (2).

In line with Figure 3, we define rec_{in} as follows. Let $\text{head}(\text{rec})$ be the set of entry statements of m , and $\text{tail}(\text{rec})$ the set of m 's exit statements. Further, let $\text{rec}_{\text{in}}(\text{rec})$ be the set of potentially recursive inside statements of m , and $\text{rec}_{\text{out}}(\text{rec})$ the set of possibly non-recursive inside statements respectively. Then:

$$\text{rec}_{\text{in}} = \begin{cases} \text{head}(\text{rec}) & \text{if } x \in \text{rec}_{\text{in}}(\text{rec}) \\ \text{head}(\text{rec}) \cup \text{rec}_{\text{in}}(\text{rec}) & \text{if } x \in \text{rec}_{\text{out}}(\text{rec}) \\ \text{head}(\text{rec}) \cup \text{rec}_{\text{out}}(\text{rec}) & \text{if } x \in \text{tail}(\text{rec}) \\ \text{tail}(\text{rec}) & \text{otherwise} \end{cases}$$

For the backward analysis we invert both recursive functions: the rec_{in} and rec_{out} , i.e., we flip all their edges, before passing them to our worklist algorithm. This means the backward analysis to actually compute backwards.

The fourth and final parameter to our worklist algorithm is d , the automaton's transition function. For the forward analysis we use the transition function of M_{in} . For the backward analysis we invert this function, flipping all edges. Effectively, this yields a non-deterministic version of M_{in} . This is sufficient because Algorithm 1 determines reachability (see line 6). By not determining M_{in} ahead-of-time, we ensure that both the forward and the backward analysis act on the same state sets, just the transition function differs. This, in turn, makes it easier to determine top shadows.

4.2 Actual worklist algorithm

Algorithm 1 first initializes the worklist set with the set of initial jobs, as explained above. The algorithm further

initializes two mappings before and after that store the configurations that have been computed so far before, respectively after each statement. These sets allow us to perform a terminating fixed-point iteration.

Next, the algorithm iterates through its worklist. For every job $(\text{start}, \text{end})$, the algorithm first updates start 's before-set. Then, when a statement leads to shadows, we just leave the configurations unchanged (line 4 in Algorithm 1). Otherwise, we compute (line 7), for every new configuration $c \in \text{rec}_{\text{in}}$ and shadow s at start , successor configurations using the supplied transition function d . In CLASP, programmers describe events through *Aspects* [2] pointers. Although we don't use the case, pointers can describe, thereby causing one single statement to be associated with multiple shadows. To compute the transition, the algorithm accesses the shadow's unique label (label shadows). In our running example, this label could be "loop", "exception" or "return". To allow the analysis to later on compute state labels of M_{in} with the state set labels that M_{in} uses, we determine state machines locally. Line 8 computes the unique set of successor states.

The algorithm then updates start 's after-set and associates new jobs with two different kinds of successor statements. First, in lines 10-12, the algorithm adds new jobs containing the successor configurations rec_{out} , for any statement that is a successor of start in m 's control flow graph, an instruction by rec_{out} . Lines 17-18 use the second-recursive function rec_{out} to handle inter-procedural control flow.

When propagating configurations along a rec_{out} edge, it is not correct to just copy the configurations from the edge's source to its target. Note that between any two iterations of m , other methods may execute and cause transitions in the terminating state machine. To actually model these potential transitions by "other methods", Algorithm 1 associates in line 19 with any inter-procedural successor not just the set of new configurations rec_{out} , but instead the set of configurations $\text{reaching}(\text{rec}_{\text{out}}, \text{reaching}(\text{shadow}, \text{start}))$. We defined the function reaching already above. The function $\text{reaching}(\text{shadow}, \text{start})$ computes the set of all shadows reached to start . We define this set as follows. For any inner statement start (potentially recursive or not), the set $\text{reaching}(\text{shadow}, \text{start})$ contains all shadows in all methods transitively executed through start , except for the case in m itself. After all, these are all the shadows that one can reach before reaching m 's entry statement again. Otherwise, i.e., if start is a tail of m , then $\text{reaching}(\text{shadow}, \text{start})$ contains all shadows in the program, except for the case in m . (Our implementation further narrows down related shadows by computing each shadow's unique binding to the binding stored in the configuration rec_{out} .)

It is worthwhile noting that, because we compute the expression $\text{reaching}(\text{rec}_{\text{out}}, \text{reaching}(\text{shadow}, \text{start}))$ for each statement separately, we get a certain amount of commonality. While shadows inside a certain method m (with $m \neq d$) may be relevant to one statement of m they may be irrelevant to other statements in m , and by recomputing the above function we properly distinguish such cases.

4.3 Removing top shadows

In addition to the top-shadows Analysis, CLASP also contains implementation of a syntactic Quick Check and a heuristic, pointer-based Dynamic Shadow Analysis [7, 9]. Because these analyses do not take control flow into account,

benchmark set showed some instances where this additional information would have been helpful, but not many. It even holds that, although our analysis is intra-procedural only, there are some instances where our analysis is more precise than Naive and Liskov's. This is due to the highly context-sensitive points to sets that we compute. It is important to note that Naive and Liskov also use shadow lattices to determine relevant instrumentation points. Unfortunately, their analysis therefore suffers from the same monochromatic problem that we described above. Naive and Liskov had previous their own analysis aimed except for the use of shadow lattices [30].

Dreyer and Parnianpour use existing topological analysis to speedup runtime monitoring [34]. Their work identifies "safe regions" in the code using a static topological analysis. Safe regions can be methods, single statements or compound statements (e.g., *loops*). A region is safe if its deterministic transition function does not drive the representative automaton into a final state. A special case of a safe region would be a region that does not change the automaton's state at all – an "identity region". For regions that are safe but not identity regions, the authors recommend the effect of the region and change the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to update the representative with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not let programmers who wish to inspect their code manually. The fact that the author's transformation changes the program under test to

In the thesis

$x_1 \neq x_2$	$x = x_1$	$x \neq x_1$
$x = x_1$	ff	$x = x_1$
$x \neq x_1$	$x = x_1$	$x \neq x_1 \wedge x \neq x_2$

(a) Binding binding representative when x_1 and x_2 must-not-alias

$x_1 = x_2$	$x = x_1$	$x \neq x_1$
$x = x_1$	$x = x_1 \wedge x = x_2$	ff
$x \neq x_1$	ff	$x \neq x_1 \wedge x \neq x_2$

(b) Binding binding representative when x_1 and x_2 must-alias

$x_1 = x_2$	$x = x_1$	$x \neq x_1$
$x = x_1$	$x = x_1 \wedge x \neq x_2$	$x \neq x_1 \wedge x = x_2$
$x \neq x_1$	$x = x_1 \wedge x \neq x_2$	$x \neq x_1 \wedge x \neq x_2$

(c) Binding binding representative when x_1 and x_2 may-alias

Table 5.2: Possible simplifications of binding representatives using alias information

type $V \rightarrow \tilde{O}$. Hence, in the following we will often write $\text{compatible}(s, t)$ in place of $\text{compatible}(s, t)$.

In the following, we will denote the set of all binding representatives by \tilde{B} . Using the notion of compatibility, we can further define an inclusion relation on binding representatives. Let b_1 and b_2 be two binding representatives. We say that b_2 is “at least as permissive” as b_1 , or $b_1 \subseteq b_2$, if the following holds:

$$b_1 \subseteq b_2 \iff (\forall x. \text{compatible}(x, b_1) \implies \text{compatible}(x, b_2))$$

Informally, $b_1 \subseteq b_2$ means that for every variable v , every object a that can be bound to v according to the binding representative b_1 can also be bound to a according to the binding representative b_2 . We will write $b_1 \subset b_2$ if $b_1 \subseteq b_2$ but b_1 and b_2 are different.

In the following, we will denote the empty binding representative, in which both binding functions β^+ and β^- are undefined for all variables, by \top . Note that by the

failure group. In Chapter 7 we explain how CLARA ranks potential failure groups before it reports these groups to the user.

5.3.3 Runtime overhead after Nop-shadows Analysis

In Table 5.6 we show the reduction in runtime overhead that the Nop-shadows Analysis causes. We do not show *baseline* because for this benchmark already the Orphan-shadows Analysis removed all of the monitoring overhead. As our results show, the analysis was able to remove all overhead from *baseline*, which is not surprising because the Nop-shadows Analysis removed all shadows from this benchmark for all properties. The analysis was equally effective in eliminating the overhead for *antis* *hasNextElem*. For *blast-FailSafeIterMap*, the analysis reduced the overhead by large amounts. However, the remaining overhead is still very large, and large overheads also remain in many other cases. In the next chapter we will present an approach that can lower the runtime overhead further by performing partial instrumentation only.

5.3.4 Analysis time

As mentioned, the Nop-shadows Analysis runs after the Orphan-shadows Analysis and Quick Check have already been applied. Hence, the combined analysis time will always be at least as long as the time that we reported in the last chapter. However, the reader will find that the Nop-shadows Analysis often comes at a rather moderate cost. In all but two cases, the total compilation and analysis time including all three analysis stages was under ten minutes (in the last Chapter we reported nine minutes with only the first two analysis stages enabled). The combination *blast-FailSafeIterMap* took about 18 minutes in total, and *blast-FailSafeIter* took just about 25 minutes in total. The *blast* benchmark has some very large methods. Many of these methods use iterators on collections. This explains these extraordinarily high analysis times.

The Nop-shadows Analysis itself took under 50 seconds on average. This time includes all re-instrumentations of the Orphan-shadows Analysis and Nop-shadows Analysis

Appendix B

Proof of correctness of Nop-shadows Analysis

In Section 5.1.2, we defined the semantics of dependency state machines in terms of a predicate *necessaryShadow* that researchers can be chosen freely, as long as it adheres to a given soundness condition, Condition 5.1. In this appendix we will show that if the Nop-shadows Analysis disables a shadow, then the soundness condition will hold for all events that this shadow could notify the runtime monitor about when the program under test is executed. To state the soundness condition, for any sound implementation of *necessaryShadow* we demand:

$$\begin{aligned} \forall s \in \Sigma \quad \forall t = t_1 \dots t_n \dots t_n \in \Sigma^* \quad \forall i \in \mathbb{N} : \\ s = t_i \wedge \text{matches}(t_1 \dots t_n) \neq \text{matches}(t_1 \dots t_{i-1} t_{i+1} \dots t_n) \\ \implies \text{necessaryShadow}(s, t, i) \end{aligned}$$

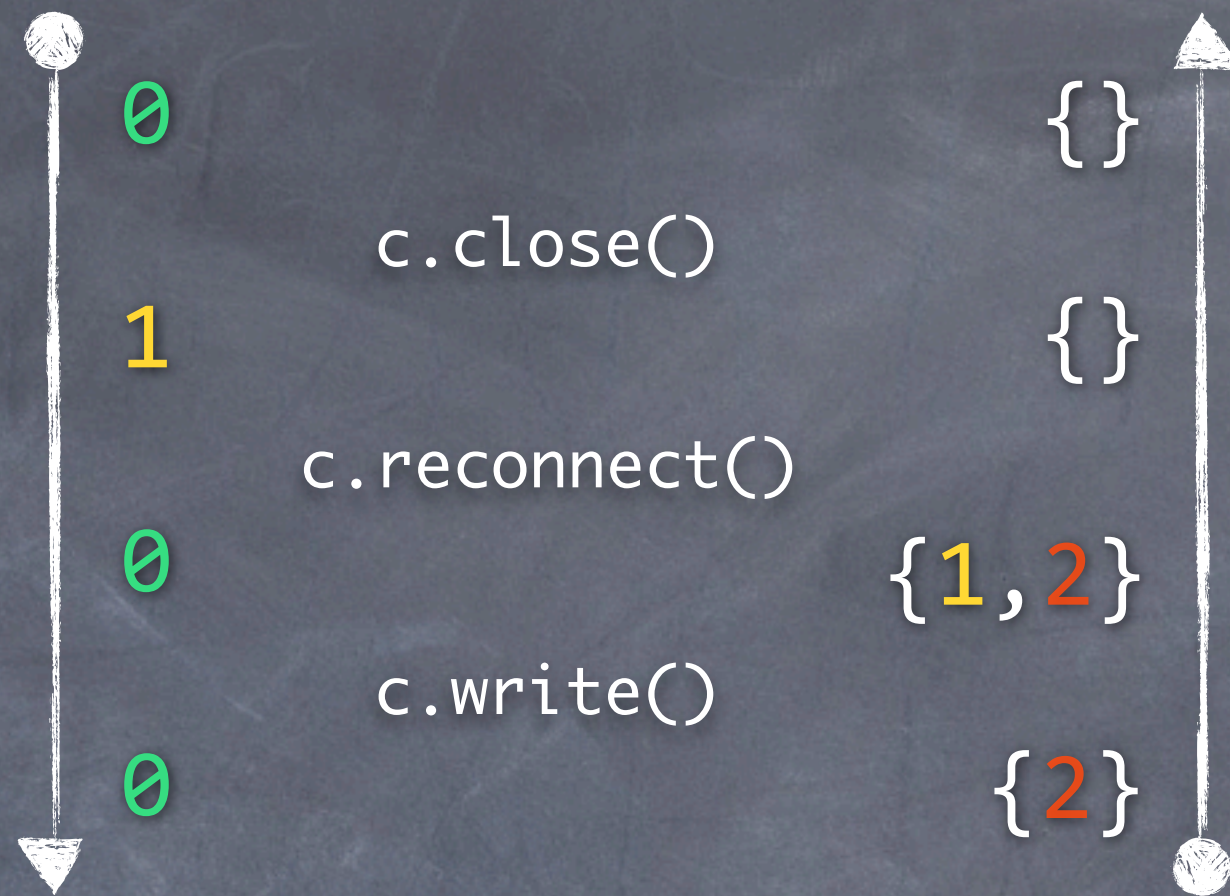
Helper definitions. In the following, we define for any transition function δ the function δ^* as the transitive closure of δ . Also, for any deterministic finite-state machine $\mathcal{M} = (Q, \Sigma, q, \delta, F)$, we define for any $q \in Q$ a state machine \mathcal{M}_q as $\mathcal{M}_q = (Q, \Sigma, q, \delta, F)$.

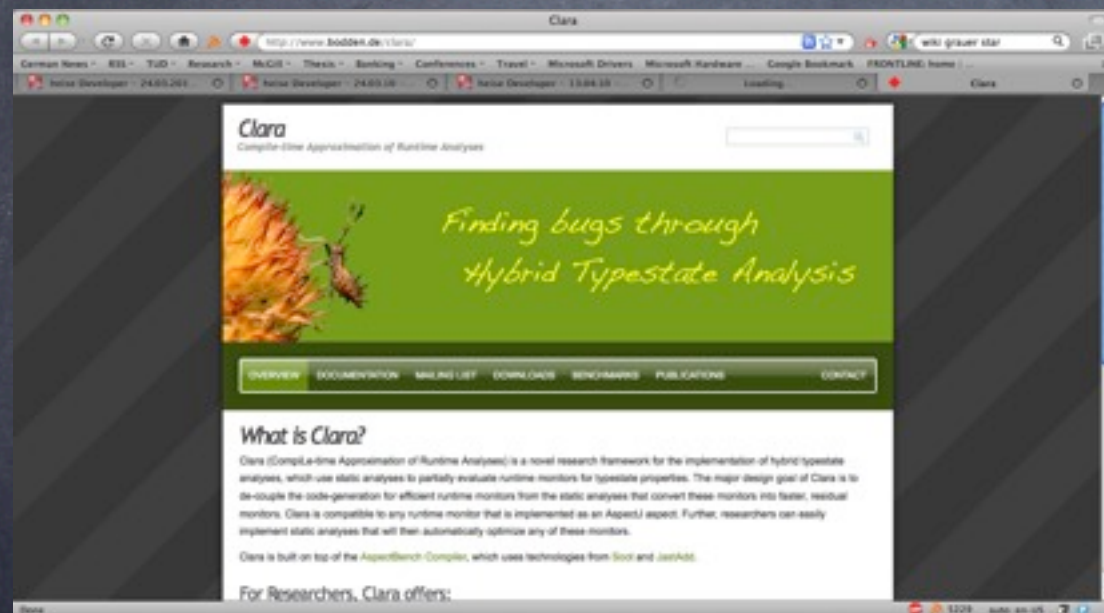
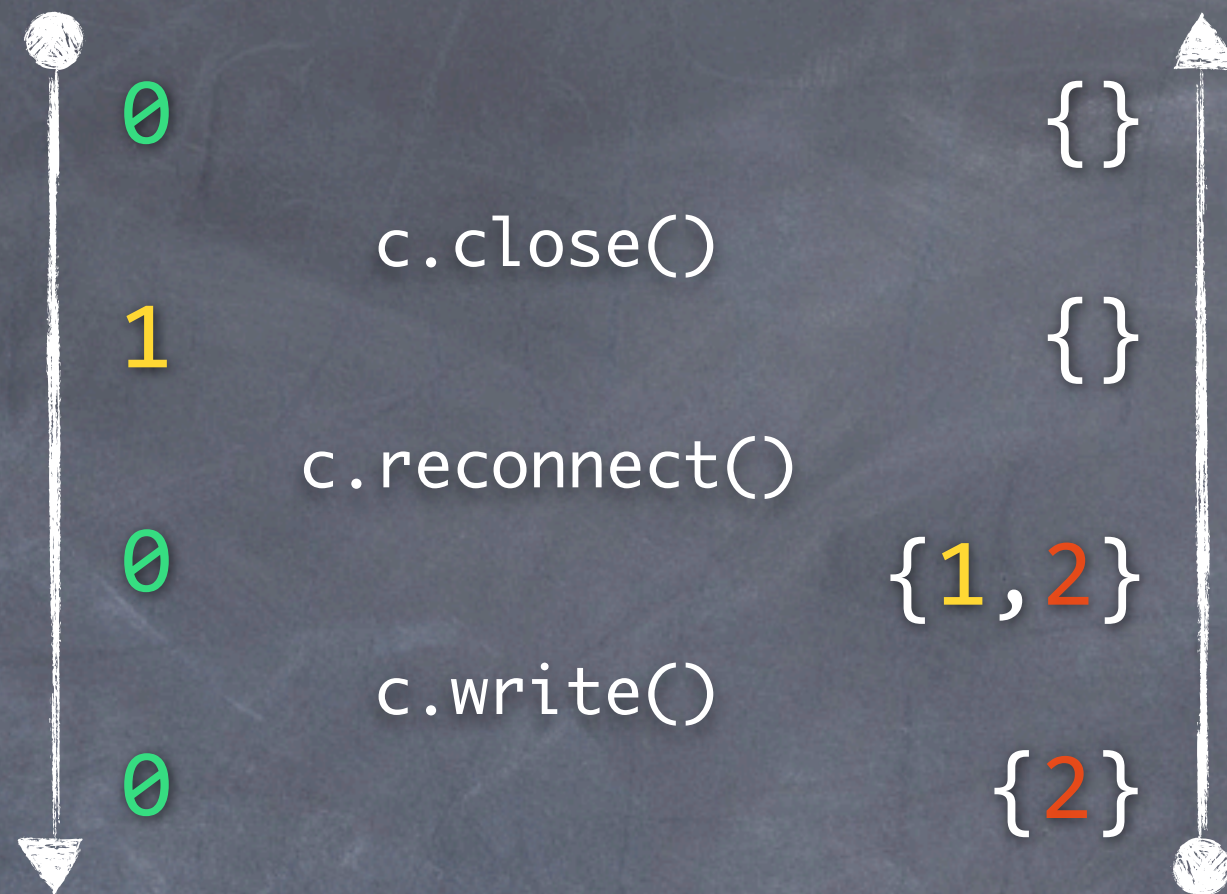
Correctness of condition for shadow removal. Assume that the Nop-shadows Analysis disables a shadow s with $\text{label}(s) = t_i$. In the following, we will prove

constraint system
to deal with
aliasing

results on
reduction of
runtime overhead

correctness proof

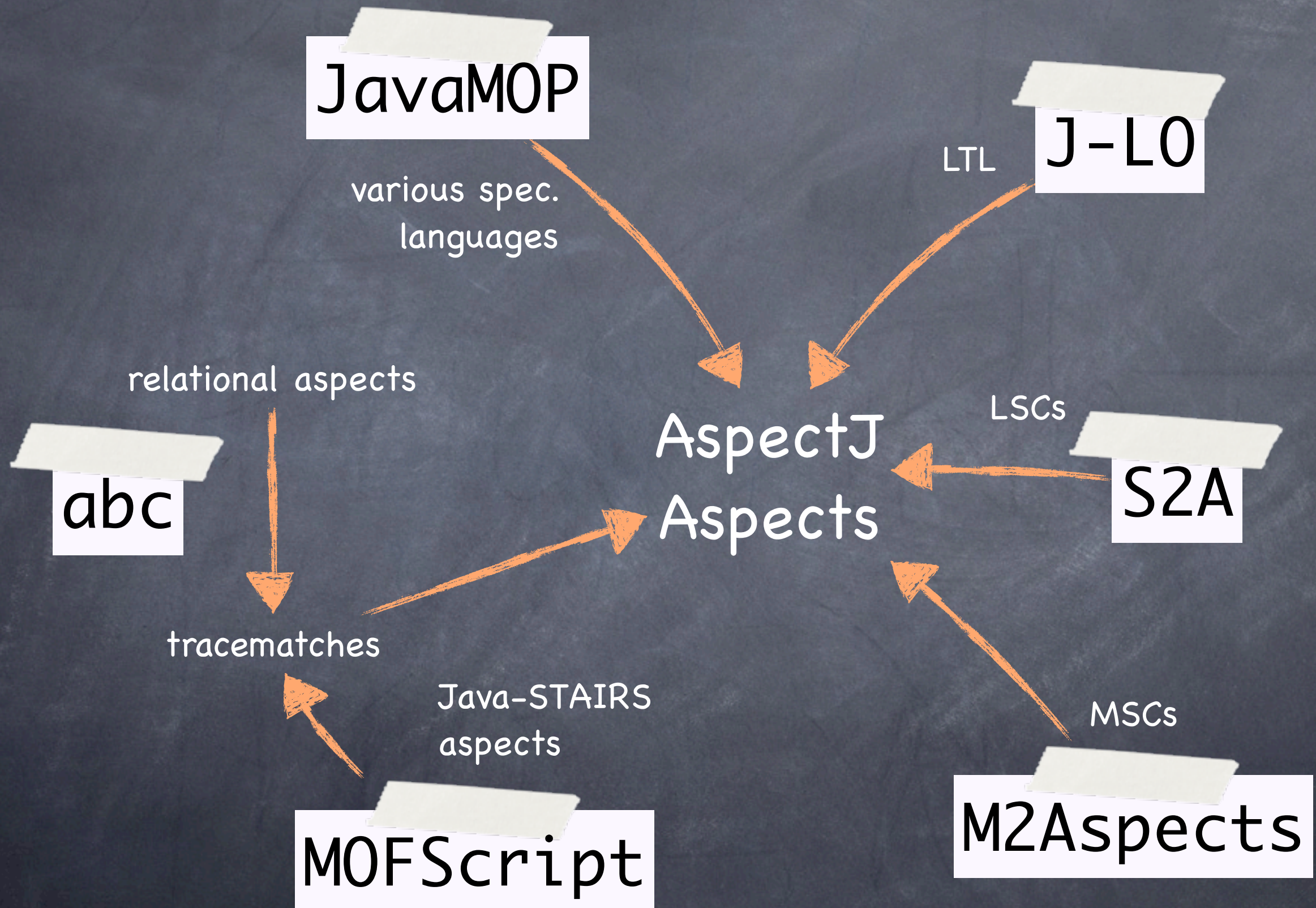




	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll		0/71	0/6			0/31	0/18	0/18	0/10	
ASyncIterC		0/1621	0/498	0/146	0/33	0/128	0/149	0/149	0/671	
ASyncIterM		0/1684	0/507	0/176	0/39	0/138	0/152	0/152	0/718	
FailSafeEnum	0/76	0/3	0/1	6/18	0/120	44/110	0/61	0/61	0/21	0/222
FailSafeEnumHT	26/133	0/102	0/44	0/205	3/114	61/153	0/37	0/37	0/100	0/319
FailSafeIter	0/23	830/1394	149/510	0/288	0/112	112/253	0/217	16/217	287/546	0/158
FailSafeIterMap	0/130	444/1180	49/374	OOME	0/252	133/250	0/136	0/136	204/583	0/540
HasNextElem	0/117	0/4		0/12	0/53	34/64	0/22	0/22	0/11	1/63
HasNext		452/849	48/248	0/72	0/16	24/63	0/74	0/74	184/346	
LeakingSync	0/170	0/1994	0/920	0/2347	0/528	0/1082	0/629	0/629	0/986	0/1005
Reader	0/50	0/7	0/65	0/102	3/1216	4/139	0/226	0/226	0/102	0/106
Writer	35/171	15/563	0/70	0/429	10/1378	0/462	0/146	0/146	0/62	0/751

www.bodden.de/clara/

Runtime-verifying finite-state properties



Runtime-verifying finite-state properties

```
Set closed = new WeakIdentityHashSet();
```

```
after(Connection c) returning:  
    call(* Connection.close()) && target(c) {  
        closed.add(c);  
    }
```

```
after(Connection c) returning:  
    call(* Connection.reconnect()) && target(c) {  
        closed.remove(c);  
    }
```

```
after(Connection c) returning:  
    call(* Connection.write(..)) && target(c) {  
        if(closed.contains(c))  
            error("May not write to "+c+", as it is closed!");  
    }
```



```
Set closed = new WeakIdentityHashSet();
```

```
dependent after close(Connection c) returning:  
    call(* Connection.close()) && target(c) {  
        closed.add(c);  
    }
```

```
dependent after reconnect(Connection c) returning:  
    call(* Connection.reconnect()) && target(c) {  
        closed.remove(c);  
    }
```

```
dependent after write(Connection c) returning:  
    call(* Connection.write(..)) && target(c) {  
        if(closed.contains(c))  
            error("May not write to "+c+", as it is closed!");  
    }
```

```
dependency {  
    close, write, reconnect;  
    initial    connected: close -> connected,  
                write -> connected,  
                reconnect -> connected,  
                close -> disconnected;  
    close: close -> disconnected,  
            write -> error;  
    final    error: write -> error;  
}
```



```
Set closed = new WeakIdentityHashSet();
```

```
dependent after close(Connection c) returning:  
    call(* Connection.close()) && target(c) {  
        closed.add(c);  
    }
```

```
dependent after reconnect(Connection c) returning:  
    call(* Connection.reconnect()) && target(c) {  
        closed.remove(c);  
    }
```

```
dependent after write(Connection c) returning:  
    call(* Connection.write(..)) && target(c) {  
        if(closed.contains(c))  
            error("May not write to "+c+", as it is closed!");  
    }
```

```
dependency {  
    close, write, reconnect;  
    initial    connected: close -> connected,  
                write -> connected,  
                reconnect -> connected,  
                close -> disconnected;  
    close: close -> disconnected,  
            write -> error;  
    final     error: write -> error;  
}
```



```
Set closed = new WeakIdentityHashSet();
```

```
dependent after close(Connection c) returning:  
    call(* Connection.close()) && target(c) {  
        closed.add(c);  
    }
```

```
dependent after reconnect(Connection c) returning:  
    call(* Connection.reconnect()) && target(c) {  
        closed.remove(c);  
    }
```

```
dependent after write(Connection c) returning:  
    call(* Connection.write(..)) && target(c) {  
        if(closed.contains(c))  
            error("May not write to "+c+", as it is closed!");  
    }
```

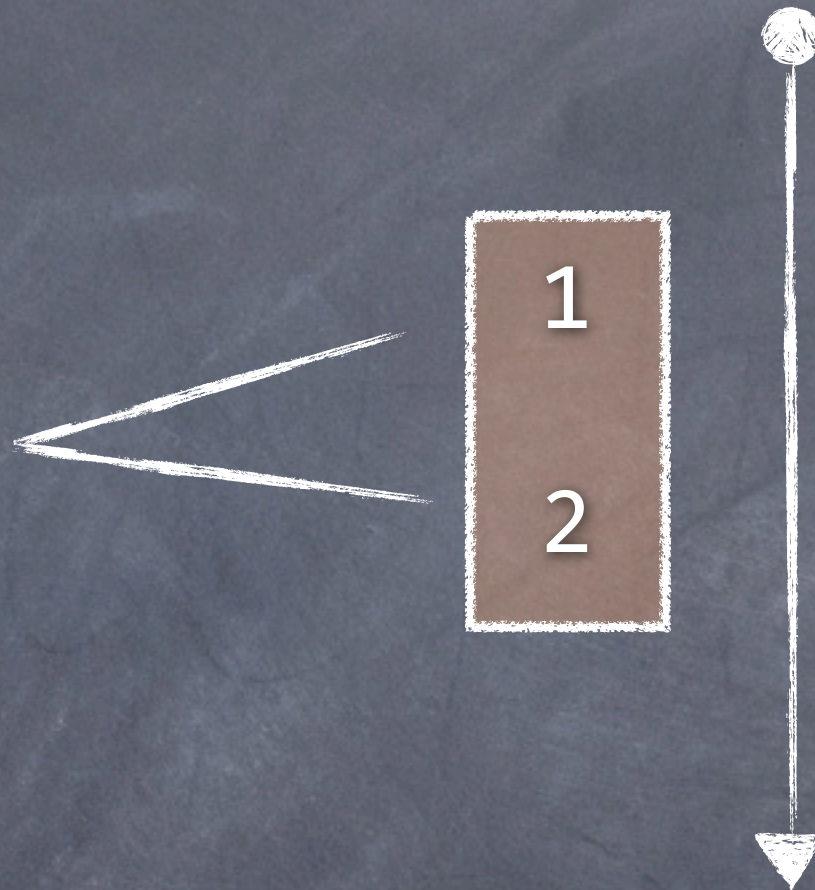
```
dependency {  
    close, write, reconnect;  
    initial    connected: close -> connected,  
                write -> connected,  
                reconnect -> connected,  
                close -> disconnected;  
    close: close -> disconnected,  
            write -> error;  
    final     error: write -> error;  
}
```

abstract

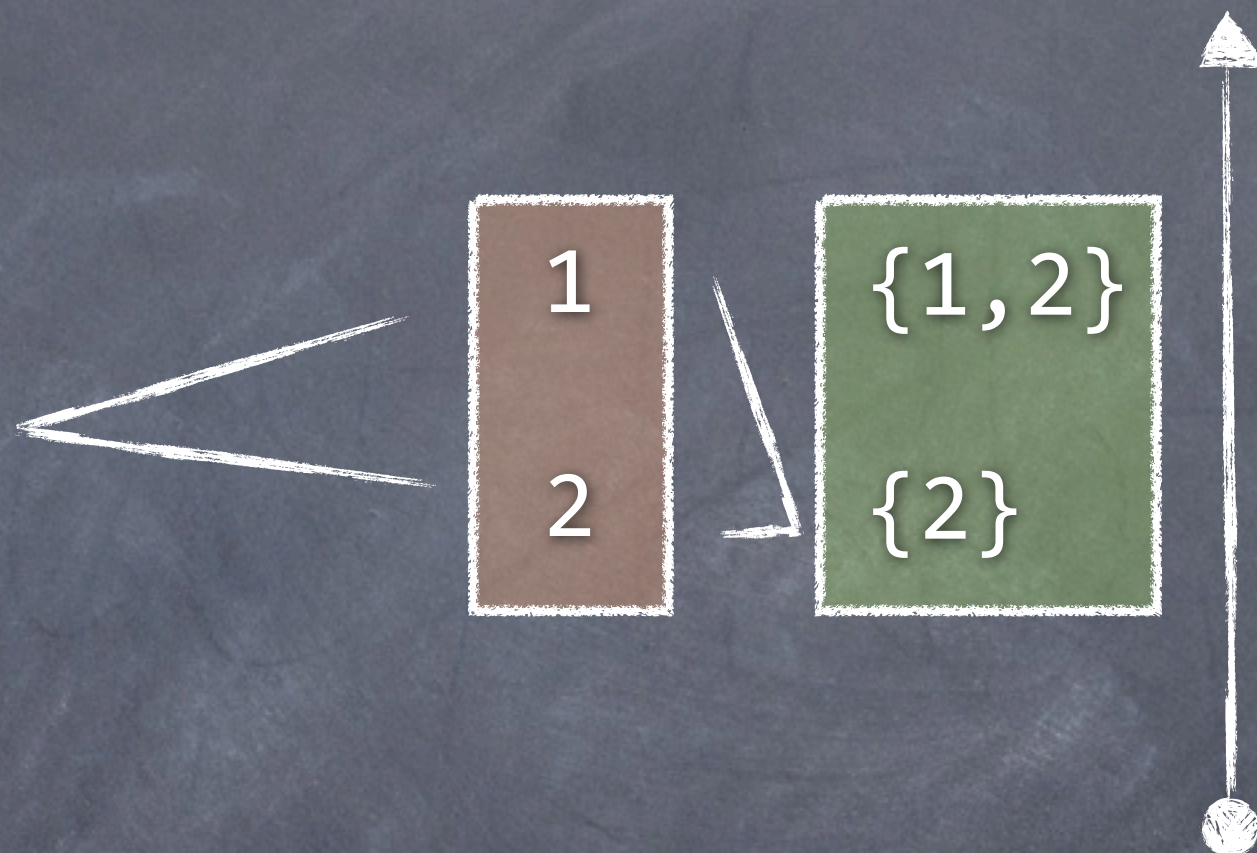
concrete

*finite-
state
property*

`c1.write(..);`



`c1.write(..);`

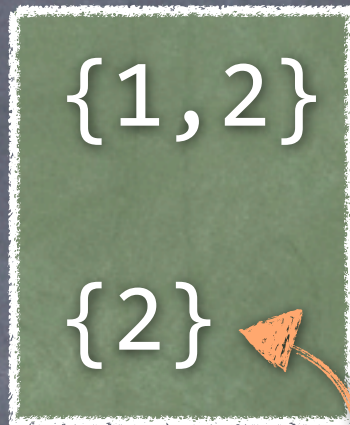


"source(*s*)"

s: c1.write(...);

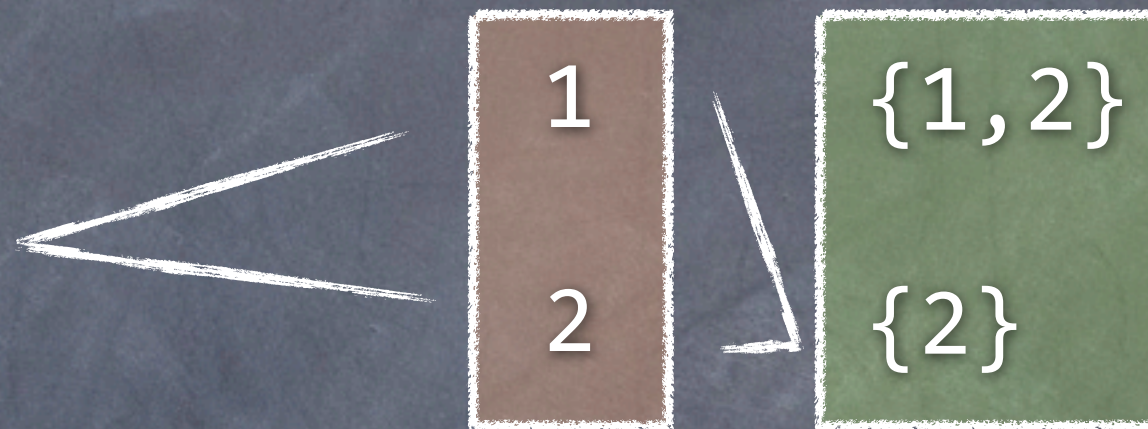
"target(*s*)"

"futures(*s*)"



“before **s**, can reach final state from states 1 or 2”

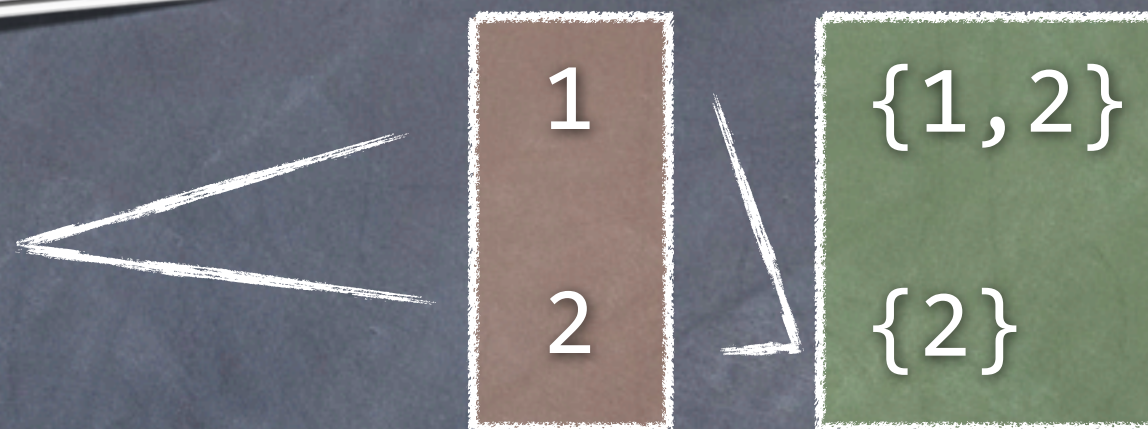
s: c1.write(...);



“before **s**, can reach final state from states 1 or 2”

“Continuation Equivalent”

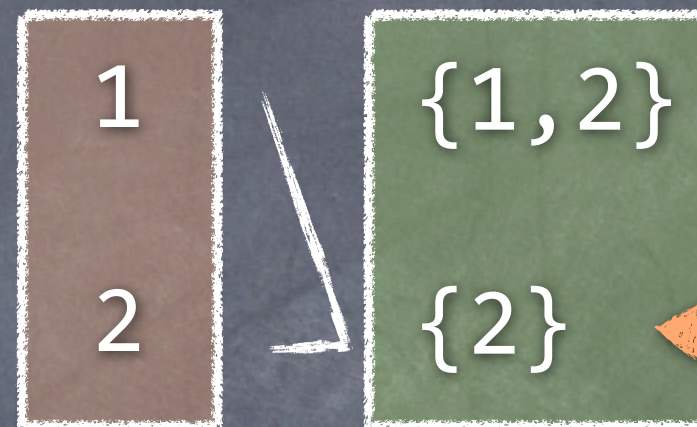
s: `c1.write(...)`



"before **s**, can reach final state from states 1 or 2"

"Continuation Equivalent"

s: `c1.write(...)`



"after **s**, can reach final state from state 2 only"



What
about
pointers?

For efficiency: **Focus** on
what's important



flow-sensitive
must-alias
may-alias

For efficiency: **Focus** on
what's important



flow-sensitive
must-alias
may-alias

For efficiency: **Focus** on
what's important



flow-insensitive
may-alias


```
public void foo() {
```

```
    x.foo();
```

```
    y.bar();
```

```
}
```



```
public void foo() {
```

```
    x.foo();
```

```
    y.bar();
```

```
}
```



```
public void foo() {
```

```
    x.foo();
```

```
    y.bar();
```

```
}
```



```
public void foo() {
```

```
    c1.close();
```

```
    x.foo();
```

```
    y.bar();
```

```
}
```

```
conn.write();
```



```
public void foo() {
```

```
<0,any>
```

```
c1.close();
```

```
x.foo();
```

```
y.bar();
```

```
}
```

```
conn.write();
```



```
public void foo() {
```

```
<0,any>
```

```
c1.close();
```

```
<1,c=o(c1)> <0,c#o(c1)>
```

```
x.foo();
```

```
y.bar();
```

```
}
```

```
conn.write();
```



```
public void foo() {
```

$\langle 0, any \rangle$

```
c1.close();
```

```
x.foo();
```

```
y.bar();
```

```
}
```

$\langle 1, c=o(c1) \rangle \langle 0, c \neq o(c1) \rangle$

```
conn.write();
```



```
public void foo() {
```

$\langle 0, \text{any} \rangle$

```
c1.close();
```

```
x.foo();
```

```
y.bar();
```

```
}
```

$\langle 1, c=o(c1) \rangle \langle 0, c \neq o(c1) \rangle$

```
conn.write();
```

$\langle 2, c=o(c1) \wedge c=o(conn) \rangle$

$\langle 0, c \neq o(c1) \vee c \neq o(conn) \rangle$


```
public void foo() {
```

$\langle 0, \text{any} \rangle$

```
c1.close();
```

```
x.foo();
```

$\langle 2, c=o(c1) \rangle \langle 0, c \neq o(c1) \rangle$

```
y.bar();
```

```
}
```

$\langle 1, c=o(c1) \rangle \langle 0, c \neq o(c1) \rangle$

```
conn.write();
```

$\langle 2, c=o(c1) \wedge c=o(conn) \rangle$
 $\langle 0, c \neq o(c1) \vee c \neq o(conn) \rangle$

