Efficient and Precise Typestate Analysis by Determining Continuation-Equivalent States

Eric Bodden

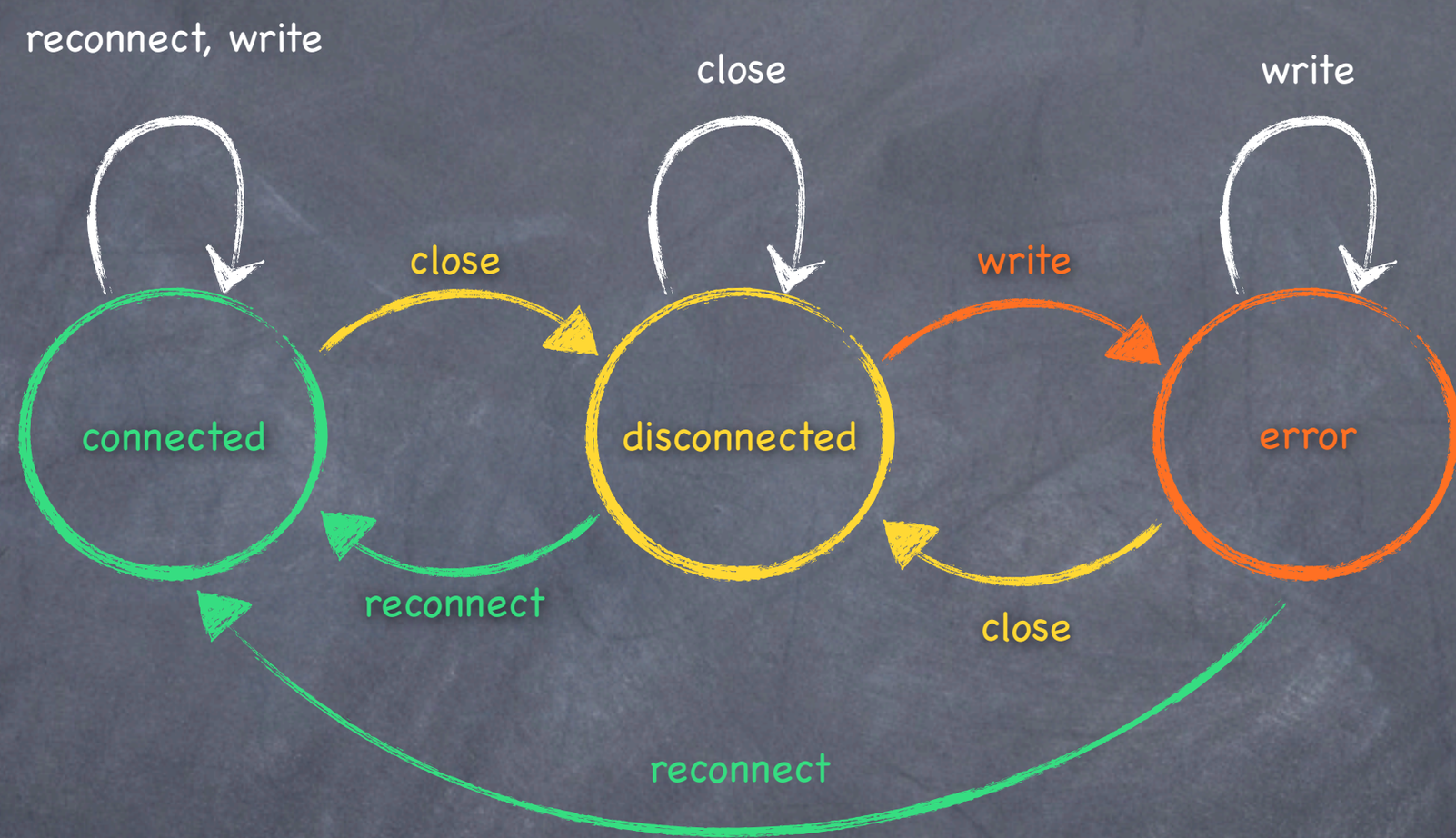
TECHNISCHE UNIVERSITÄT DARMSTADT

CASED

Finite-state properties

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

Finite-state properties

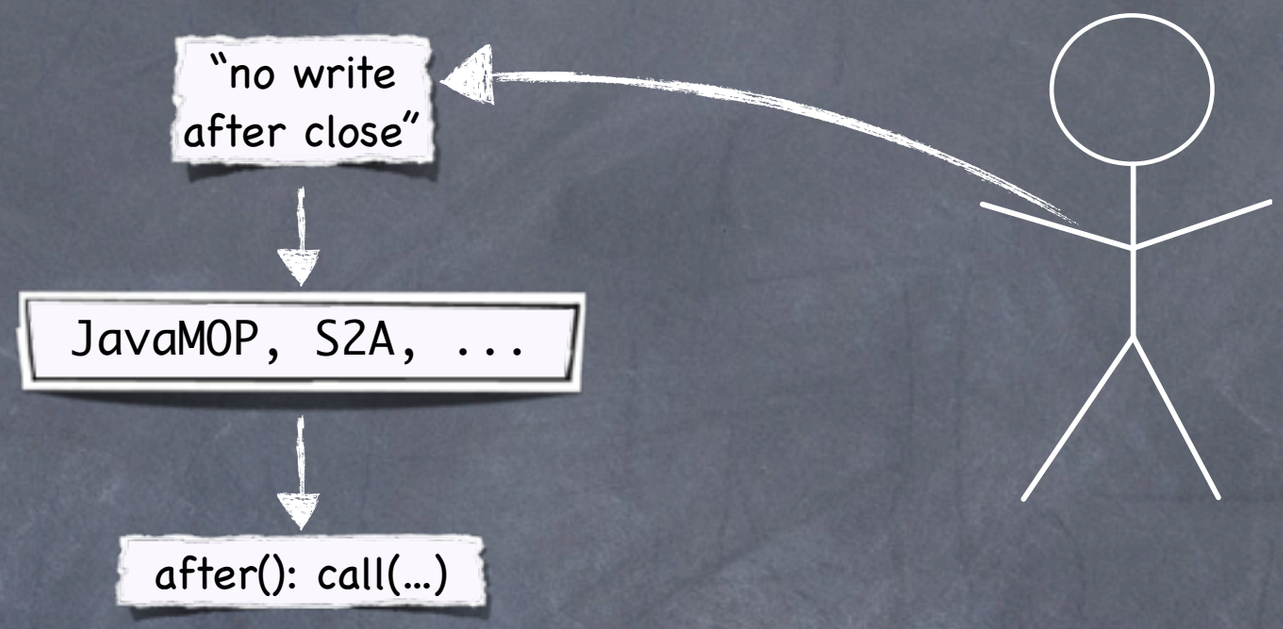


"After closing a connection c,
don't write to c until c is reconnected."

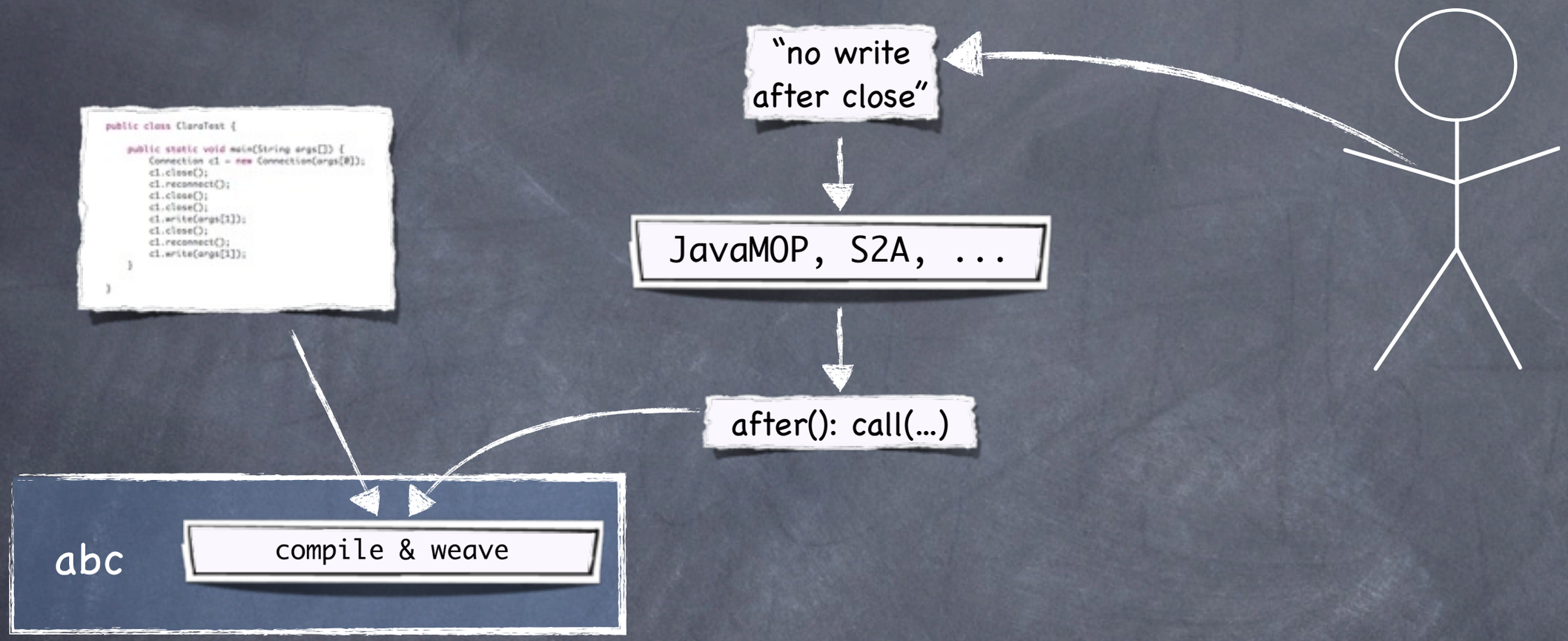
Runtime-verifying finite-state properties

Runtime-verifying finite-state properties

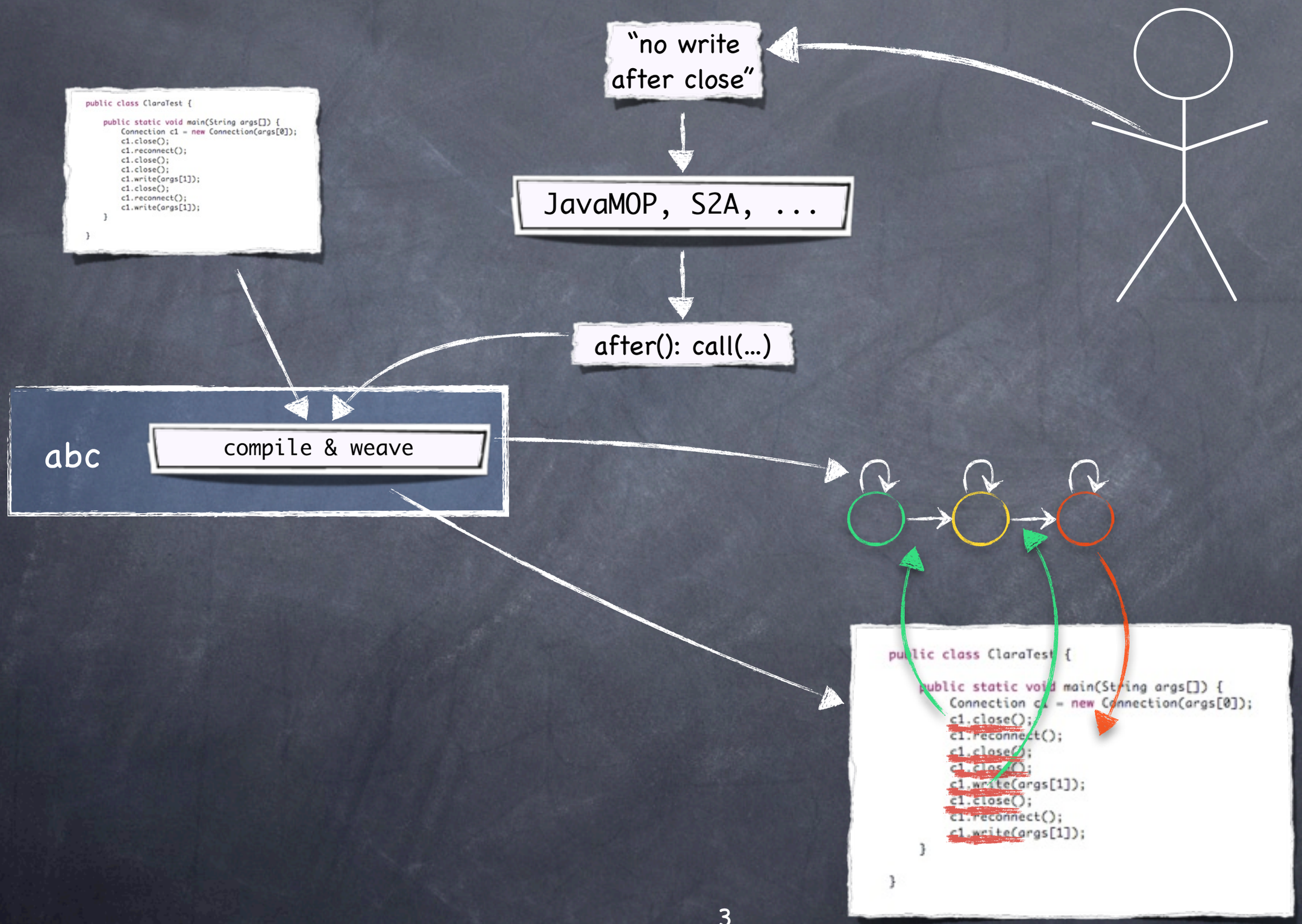
Runtime-verifying finite-state properties



Runtime-verifying finite-state properties



Runtime-verifying finite-state properties



Runtime-verifying finite-state properties

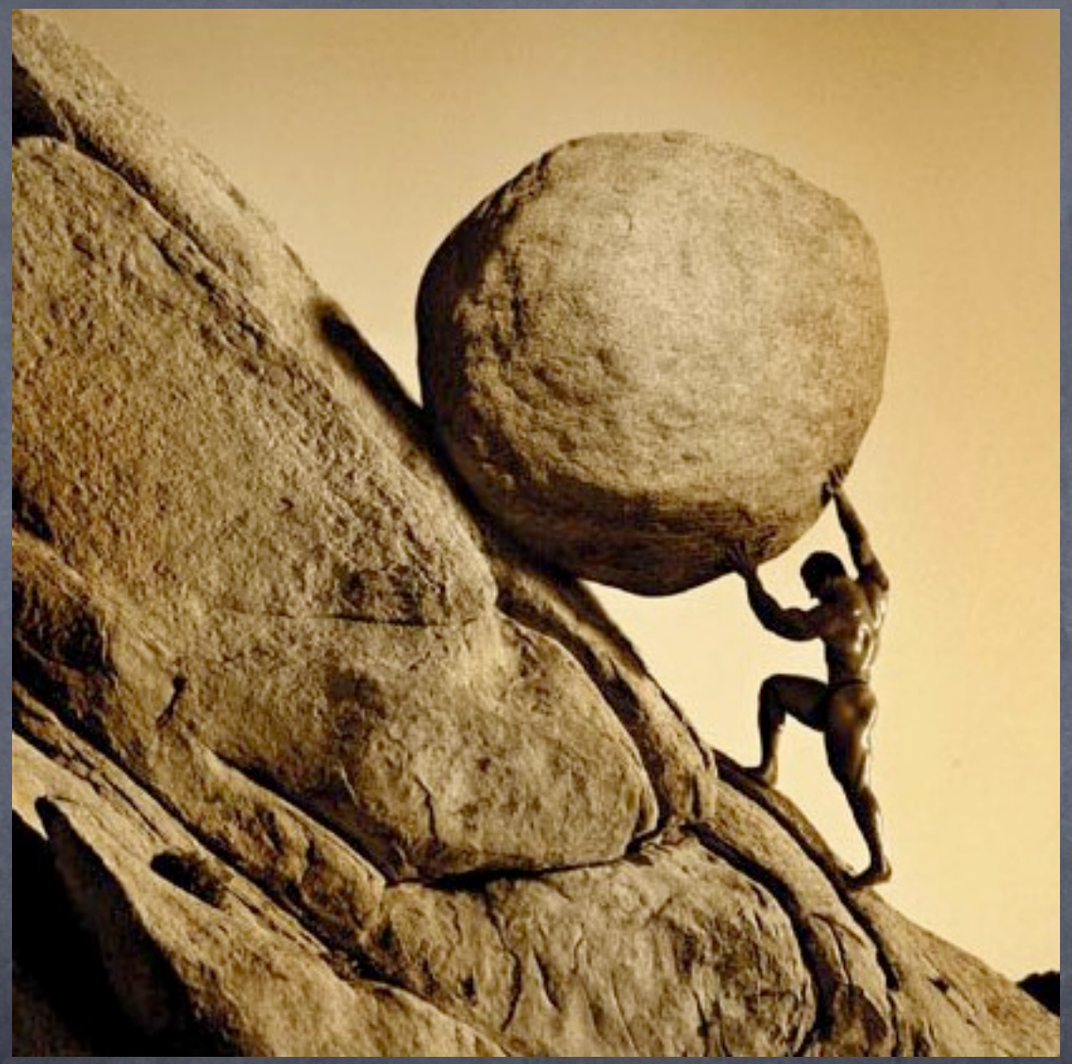
Runtime-verifying finite-state properties

No static guarantees

Runtime-verifying finite-state properties

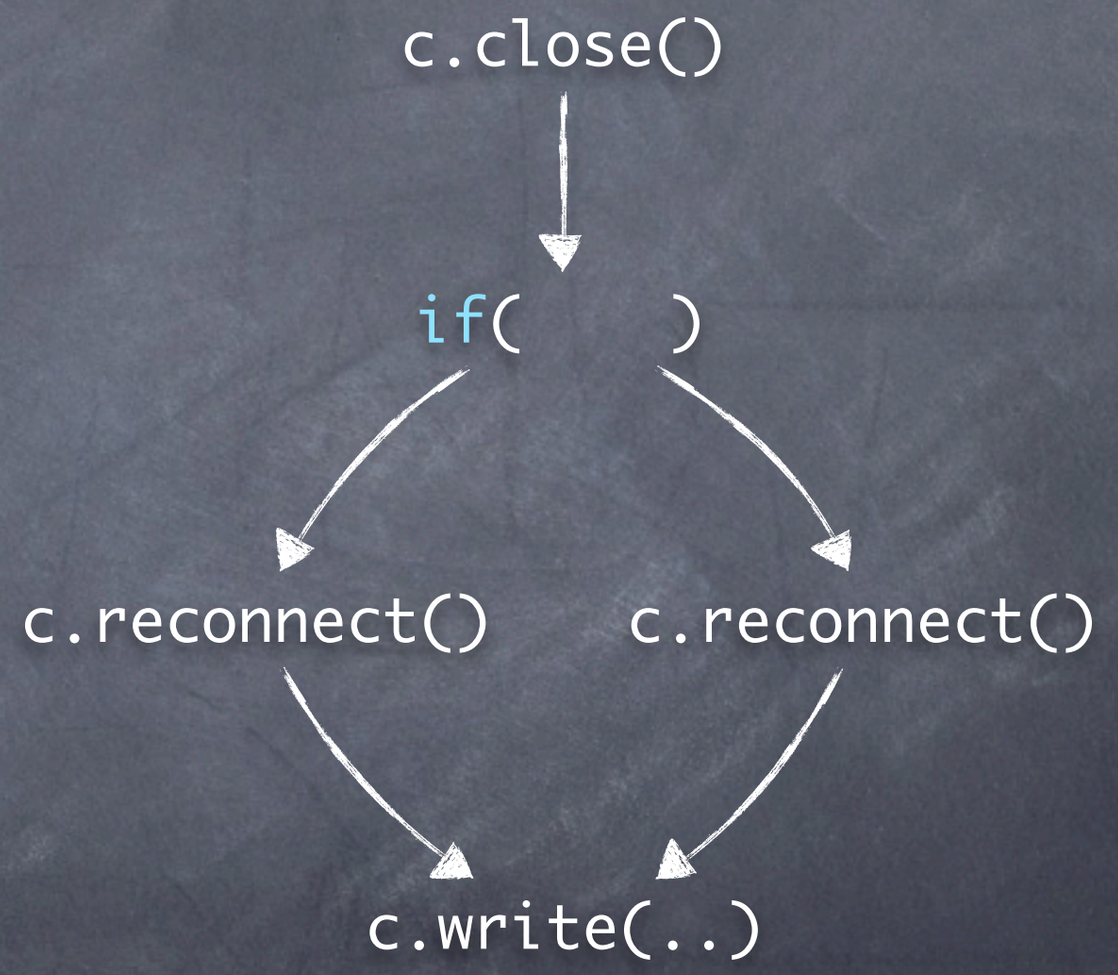
Potentially large runtime overhead

Runtime-verifying finite-state properties

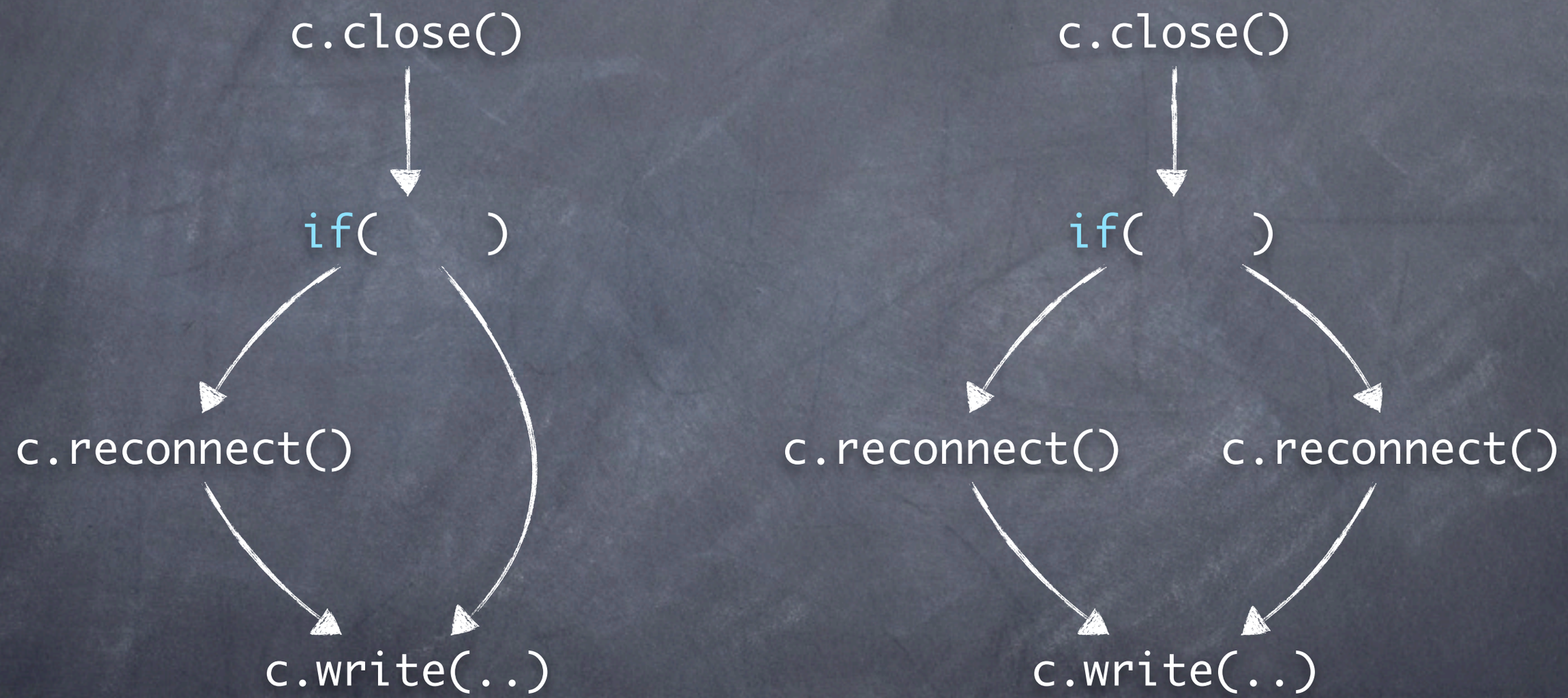


When to finish testing?

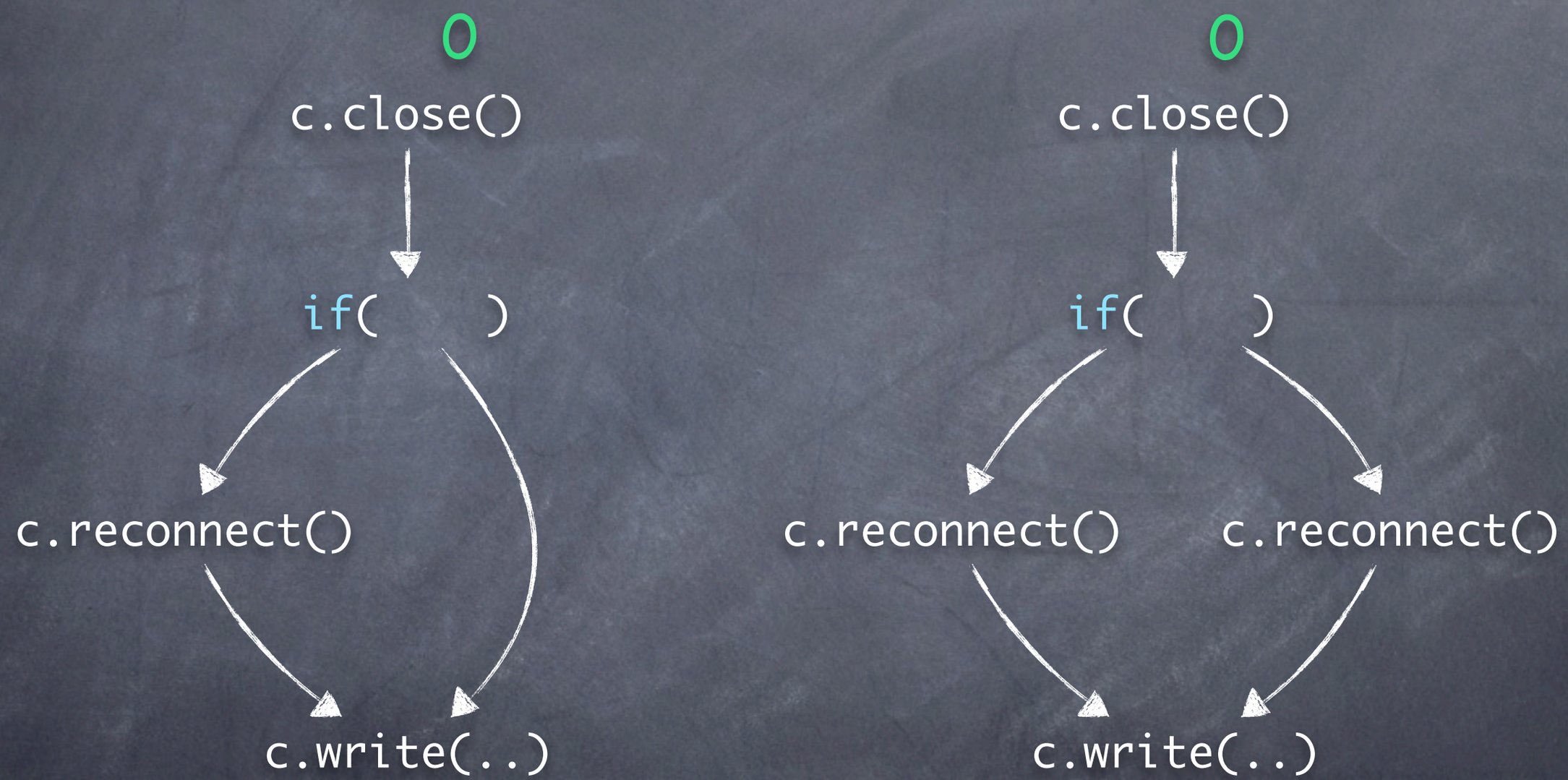
Pure static analysis



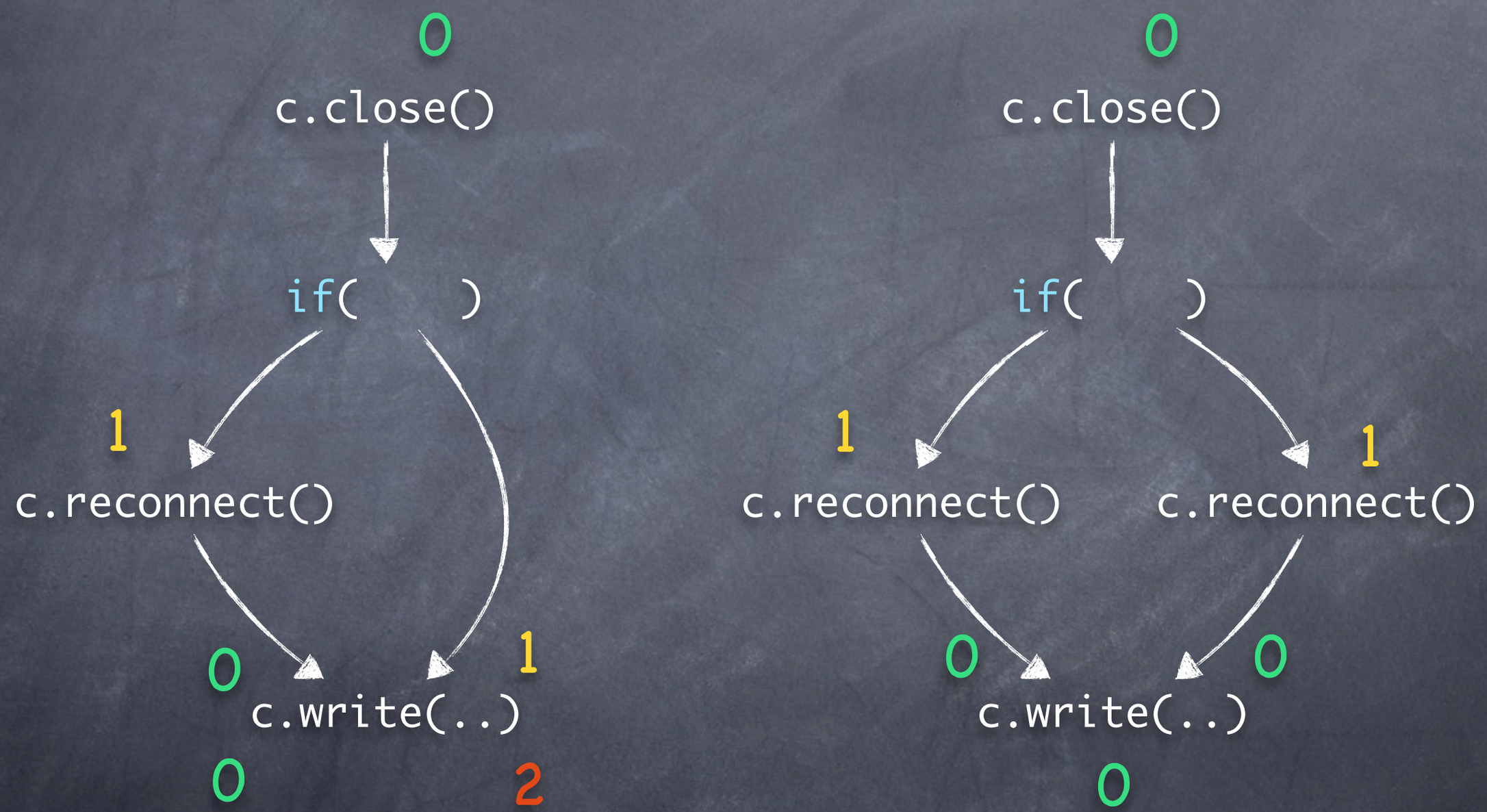
Pure static analysis



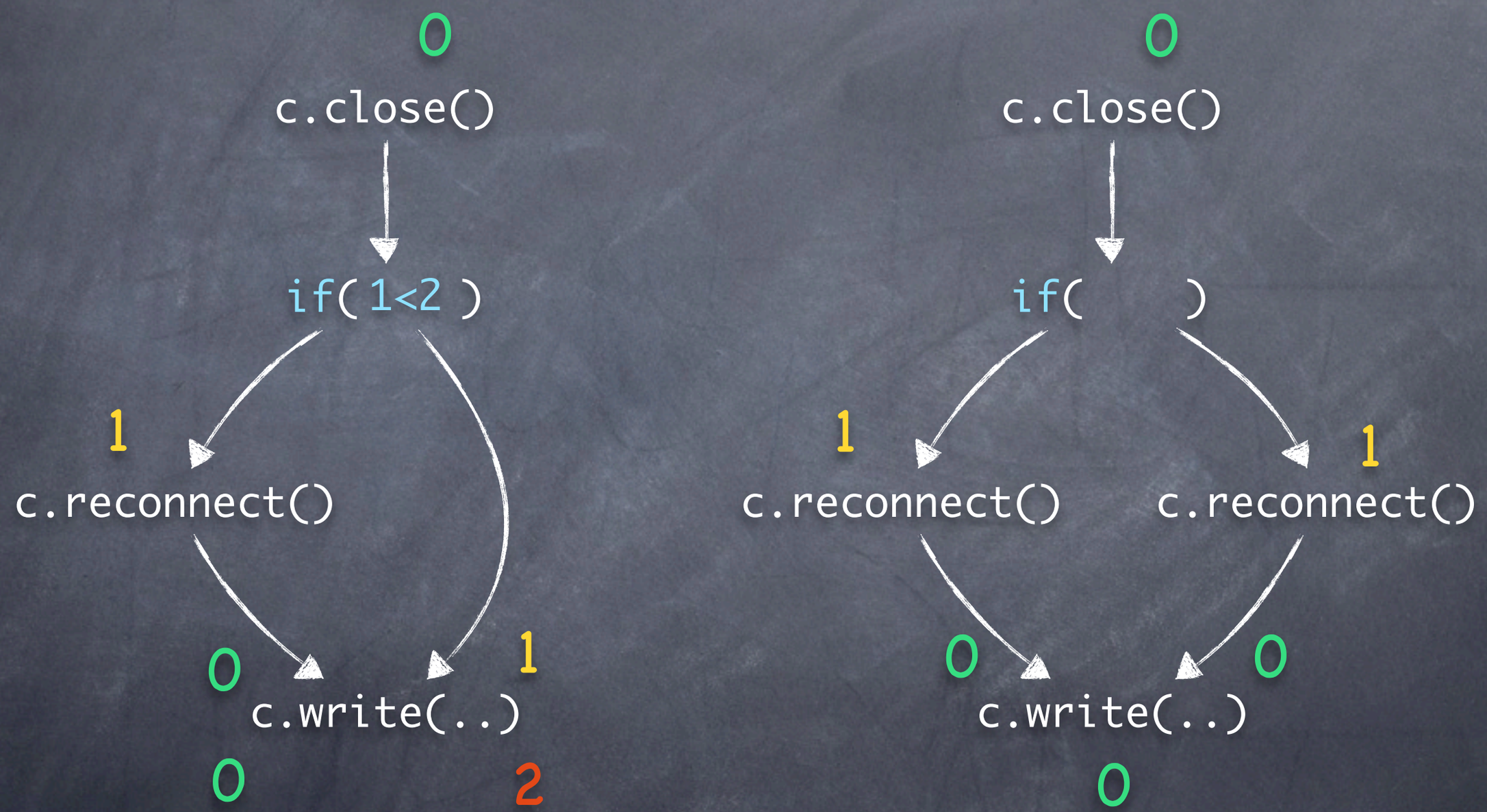
Pure static analysis



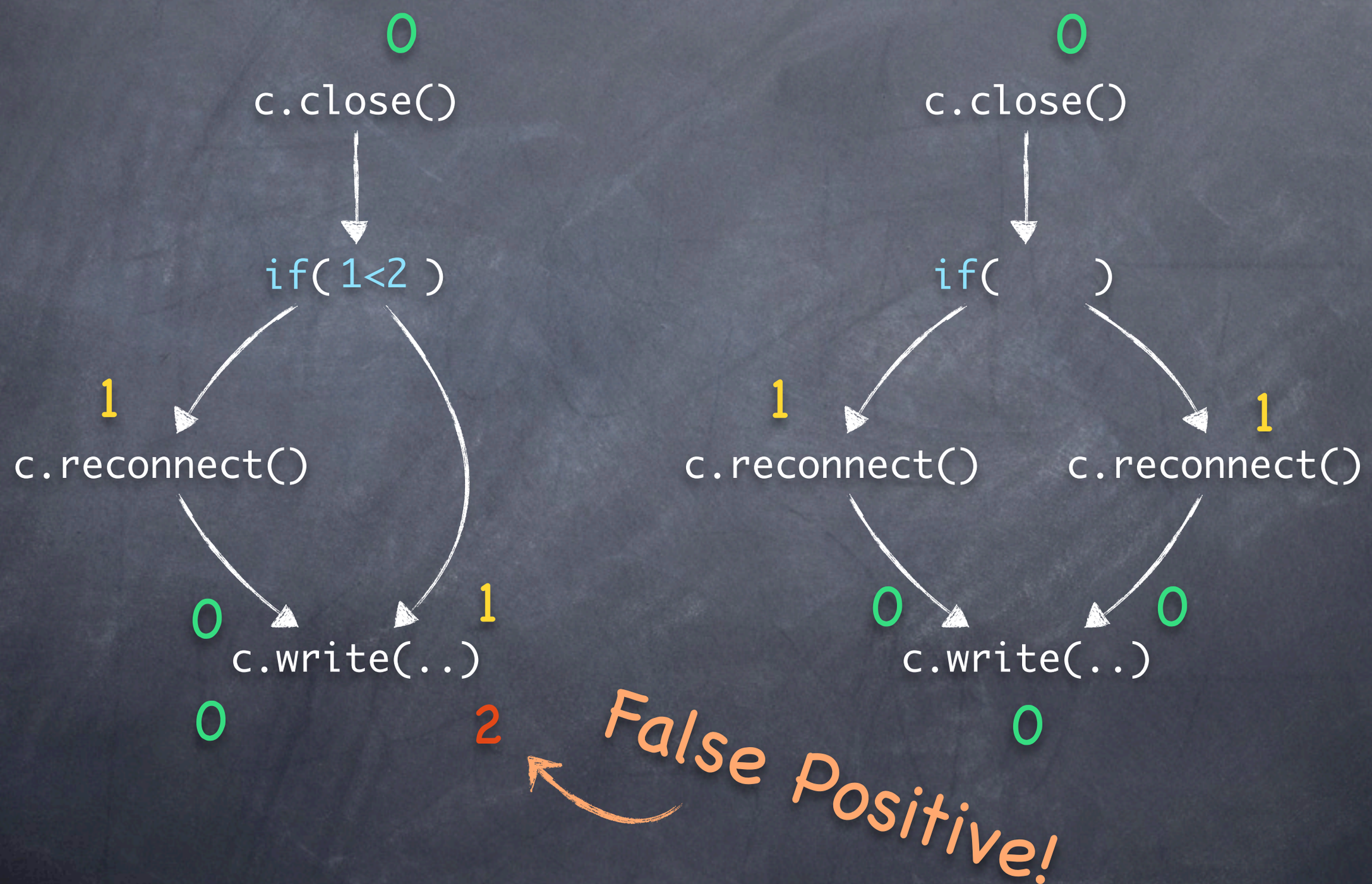
Pure static analysis



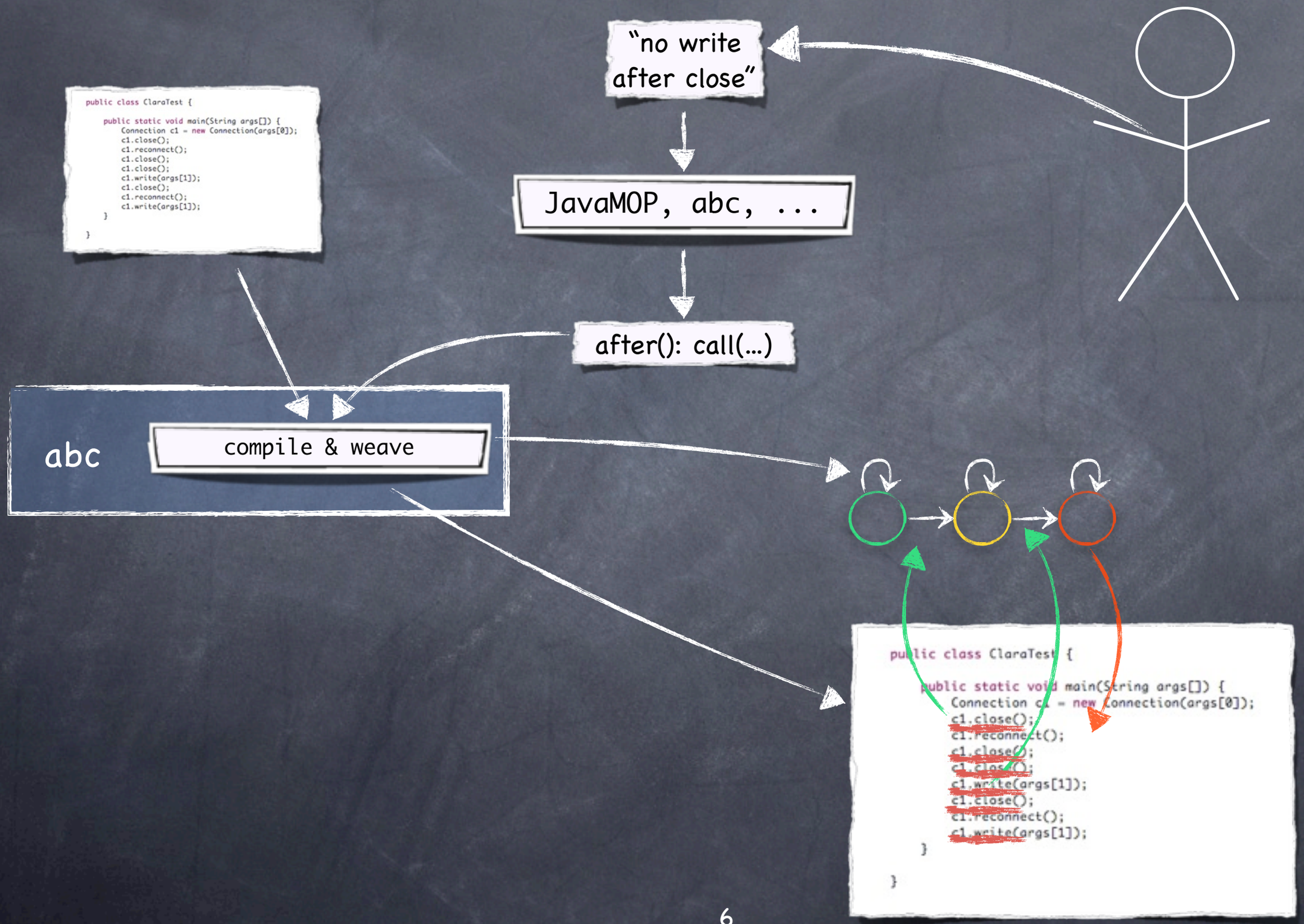
Pure static analysis



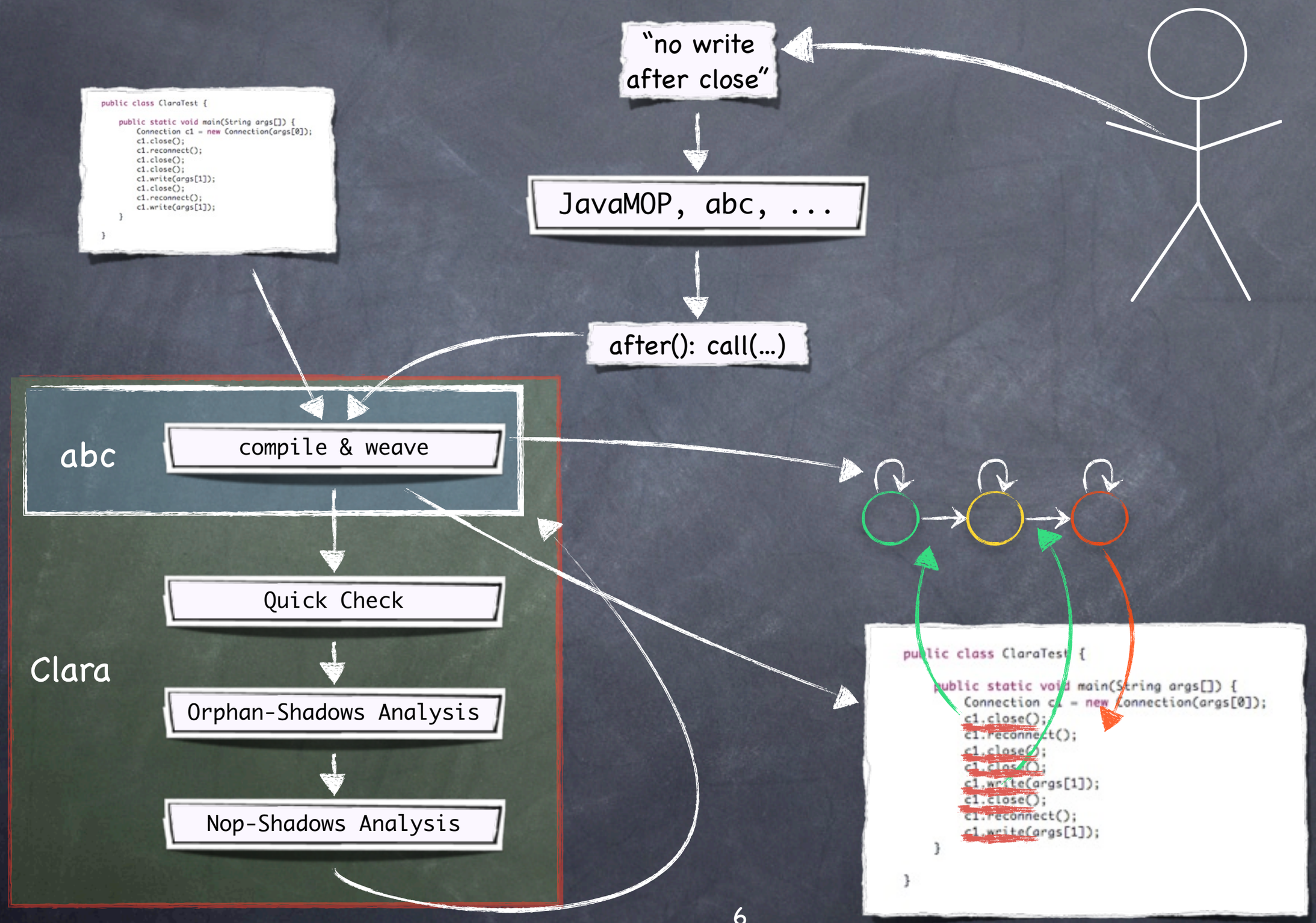
Pure static analysis



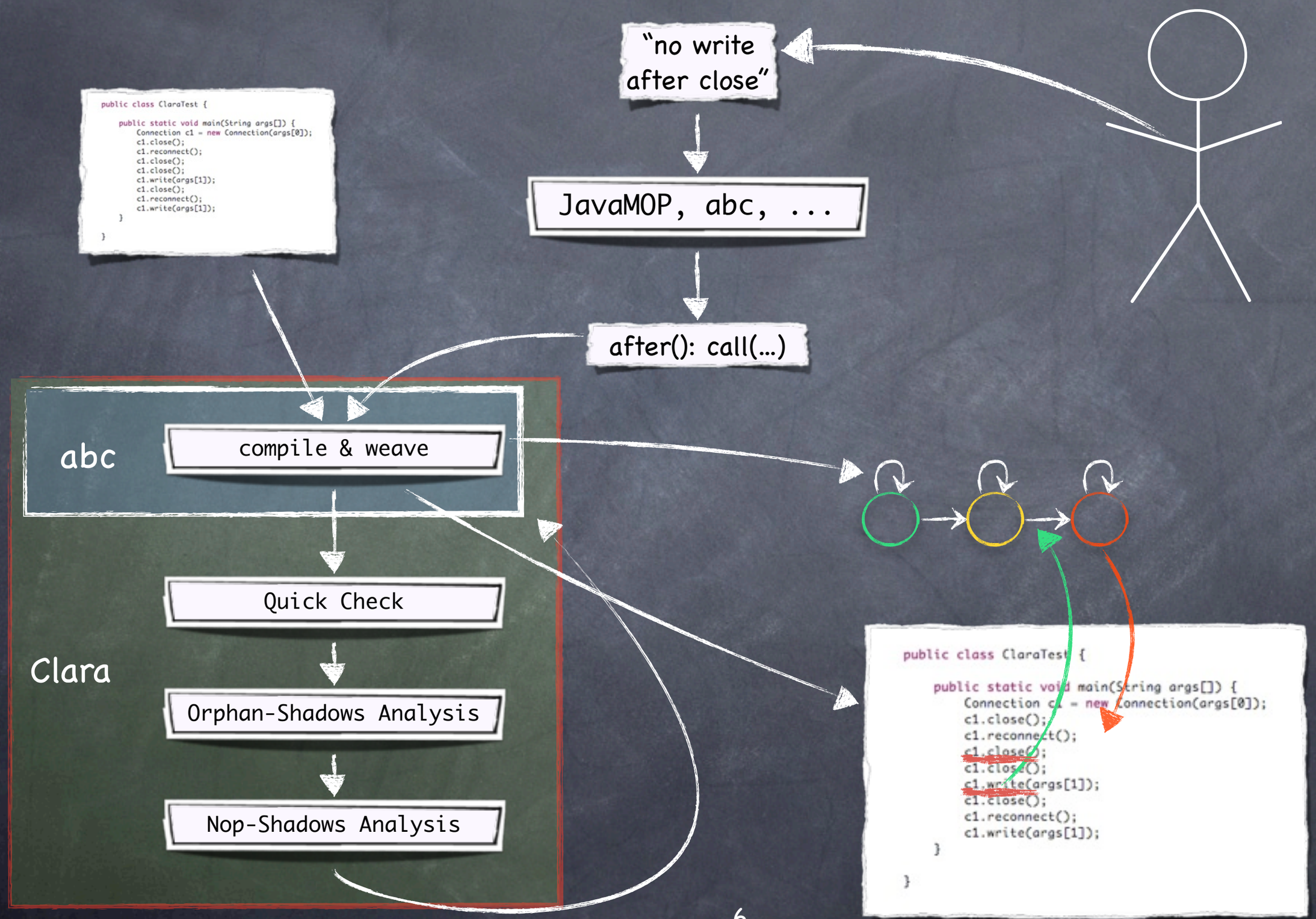
The Clara Framework

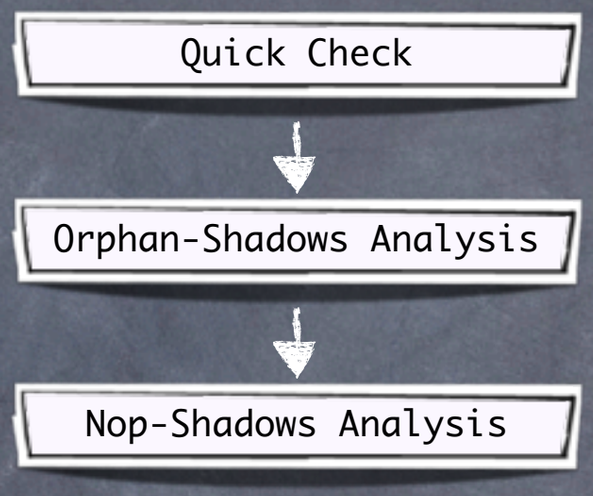


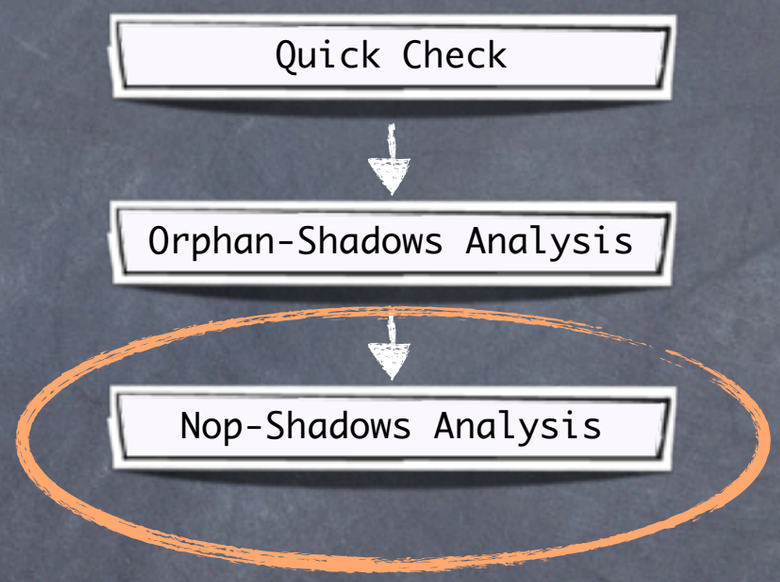
The Clara Framework

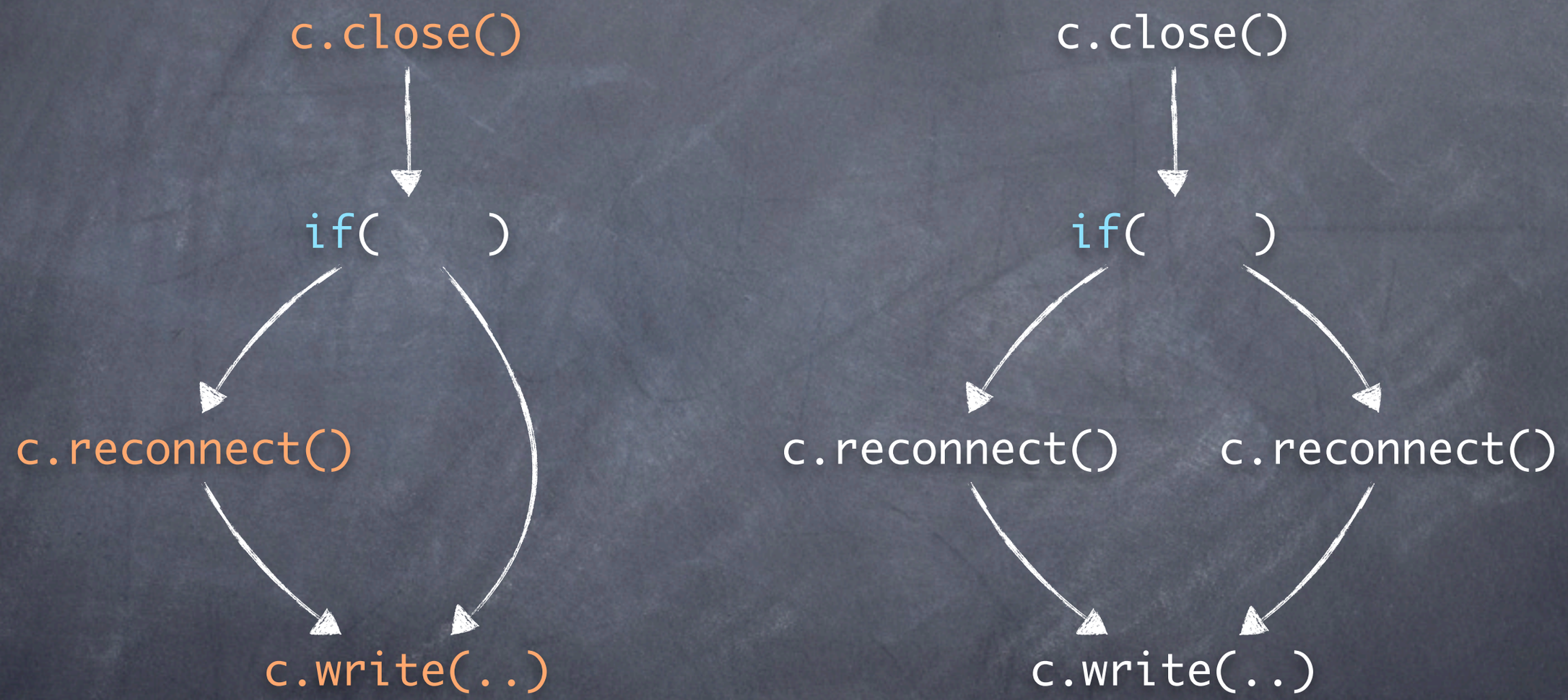


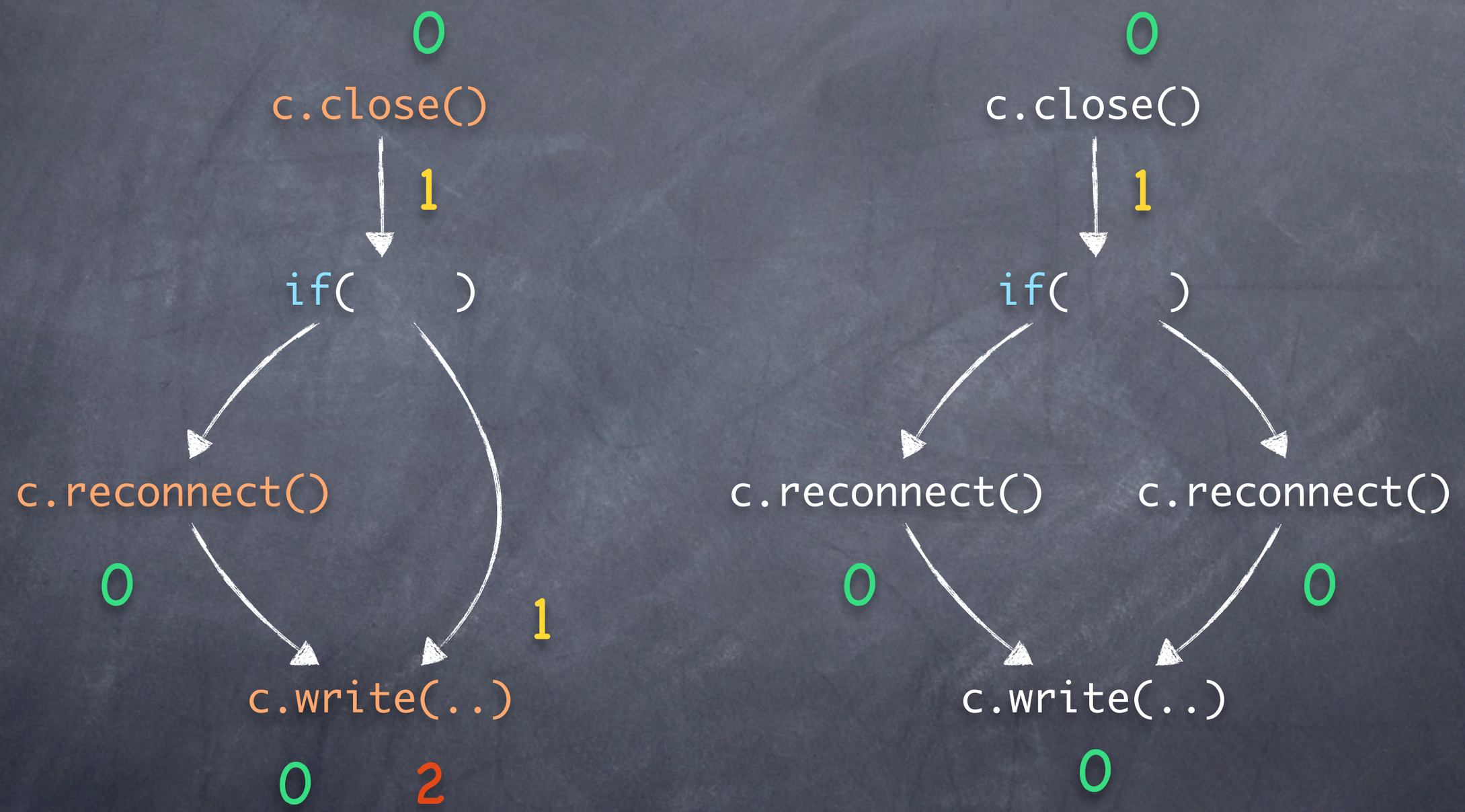
The Clara Framework

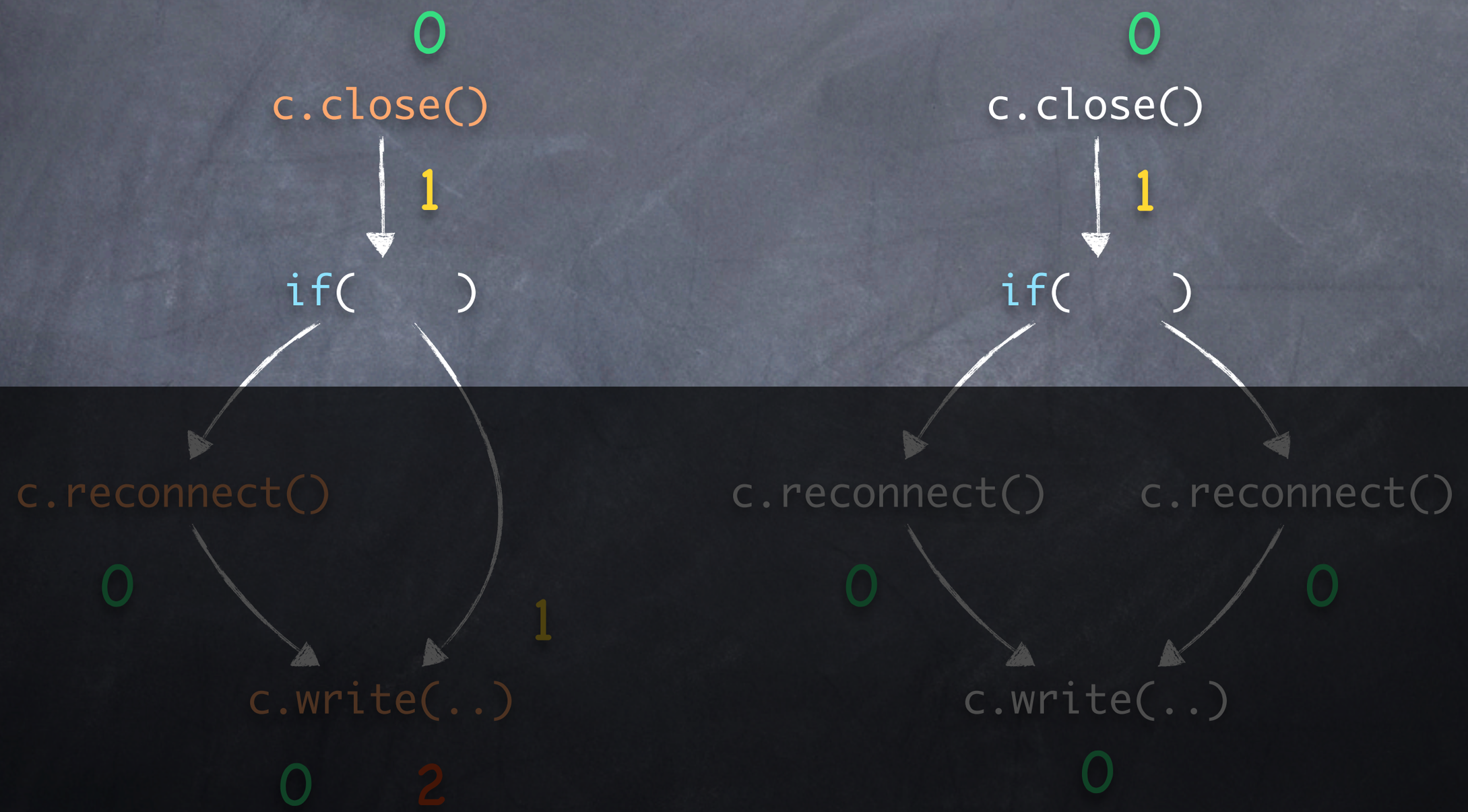



Quick Check
Orphan-Shadows Analysis
Nop-Shadows Analysis


Quick Check
Orphan-Shadows Analysis
Nop-Shadows Analysis


c.close()
if( )
c.reconnect()
c.write(..)
c.close()
if( )
c.reconnect()
c.reconnect()
c.write(..)


0
c.close()
1
if( )
c.reconnect()
0
1
c.write(..)
0
2
0
c.close()
1
if( )
c.reconnect()
c.reconnect()
0
0
c.write(..)
0


0
c.close()
1
if( )
c.reconnect()
0
1
c.write(..)
0
2
0
c.close()
1
if( )
c.reconnect()
c.reconnect()
0
0
c.write(..)
0

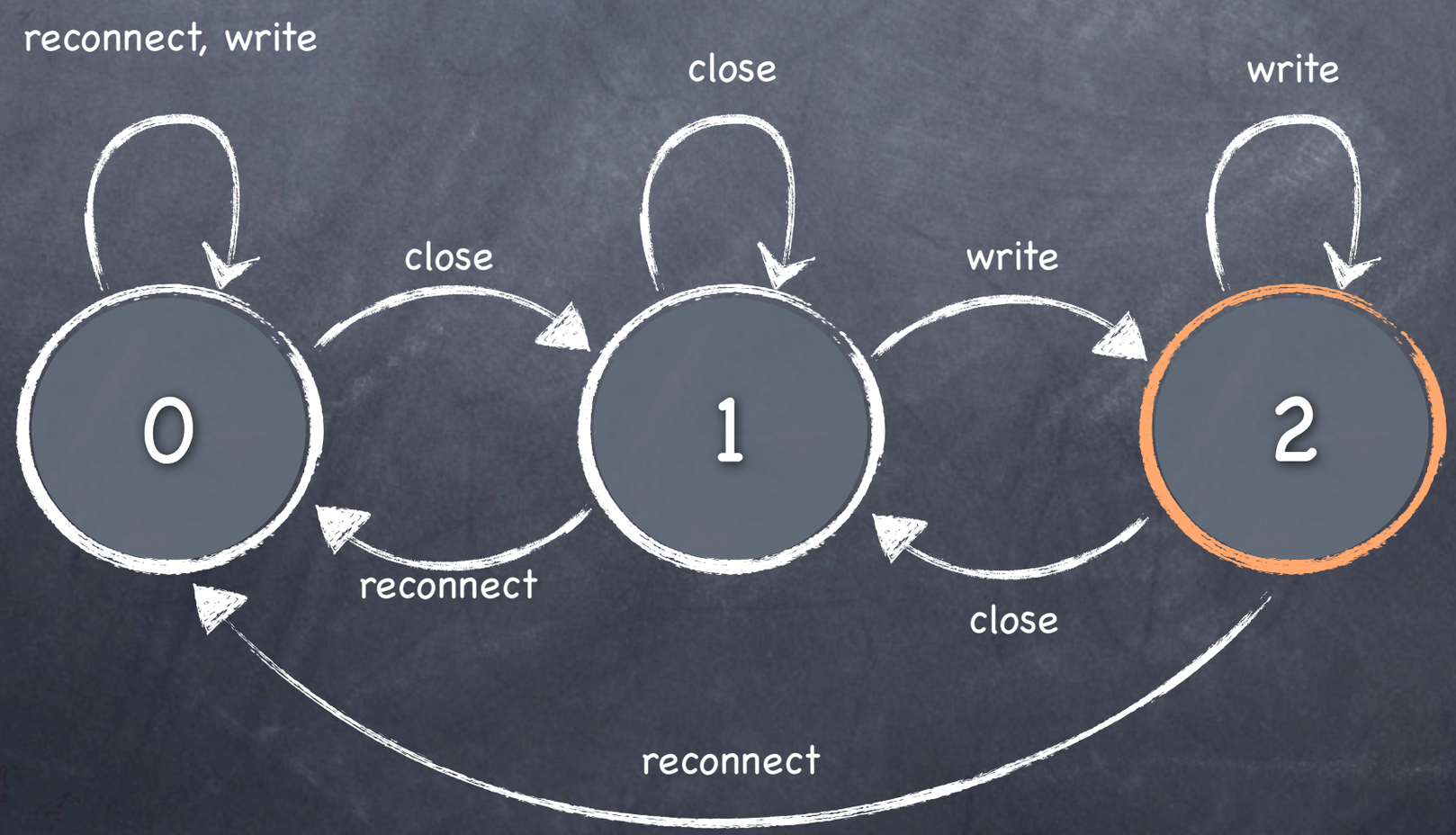
Nop-Shadows Analysis

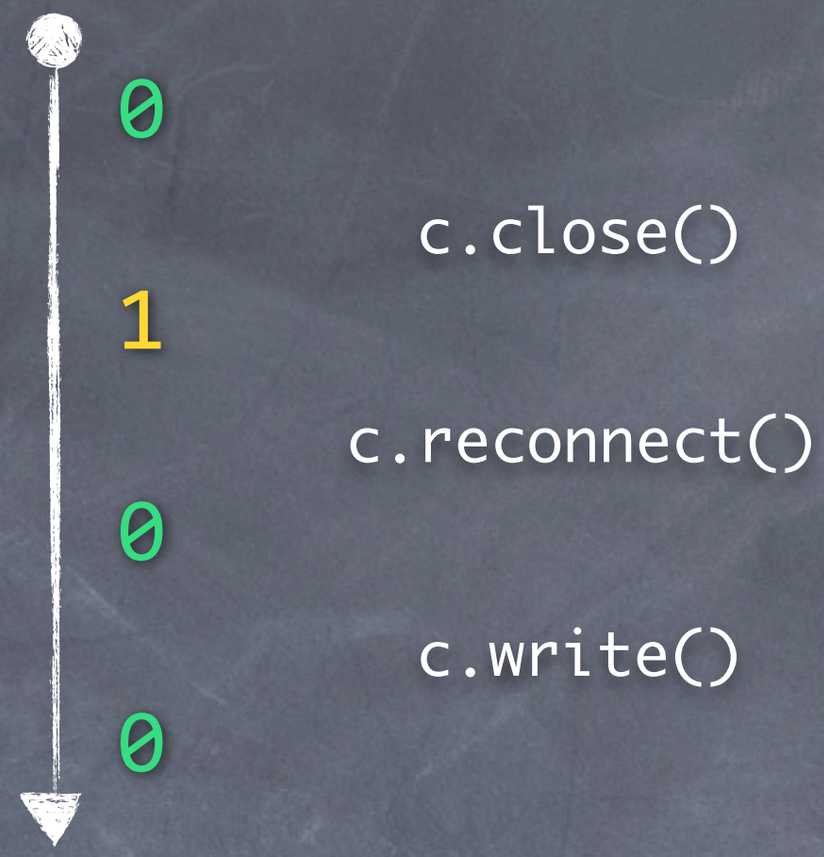
Idea:

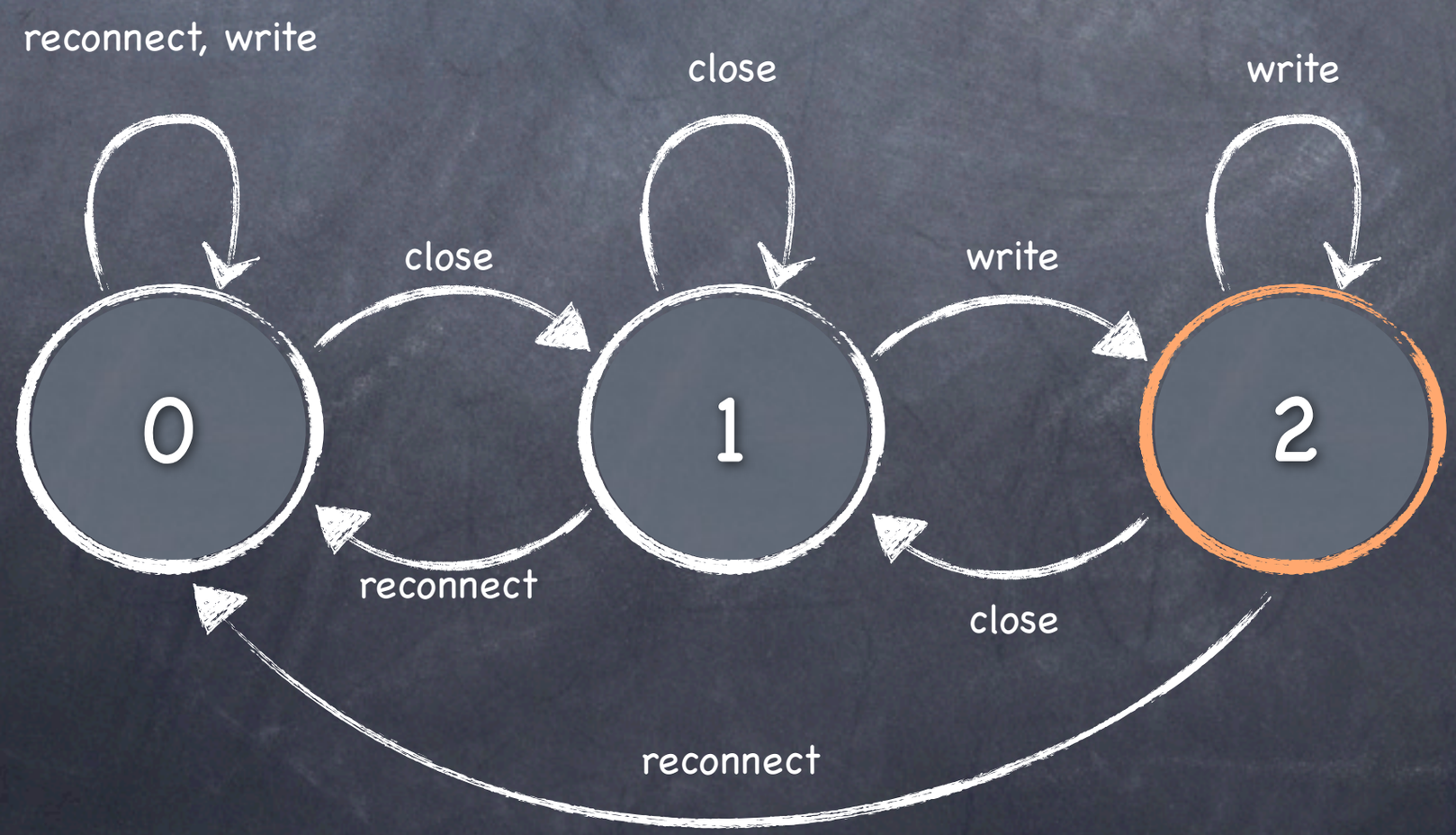
For every transitioning statement s:

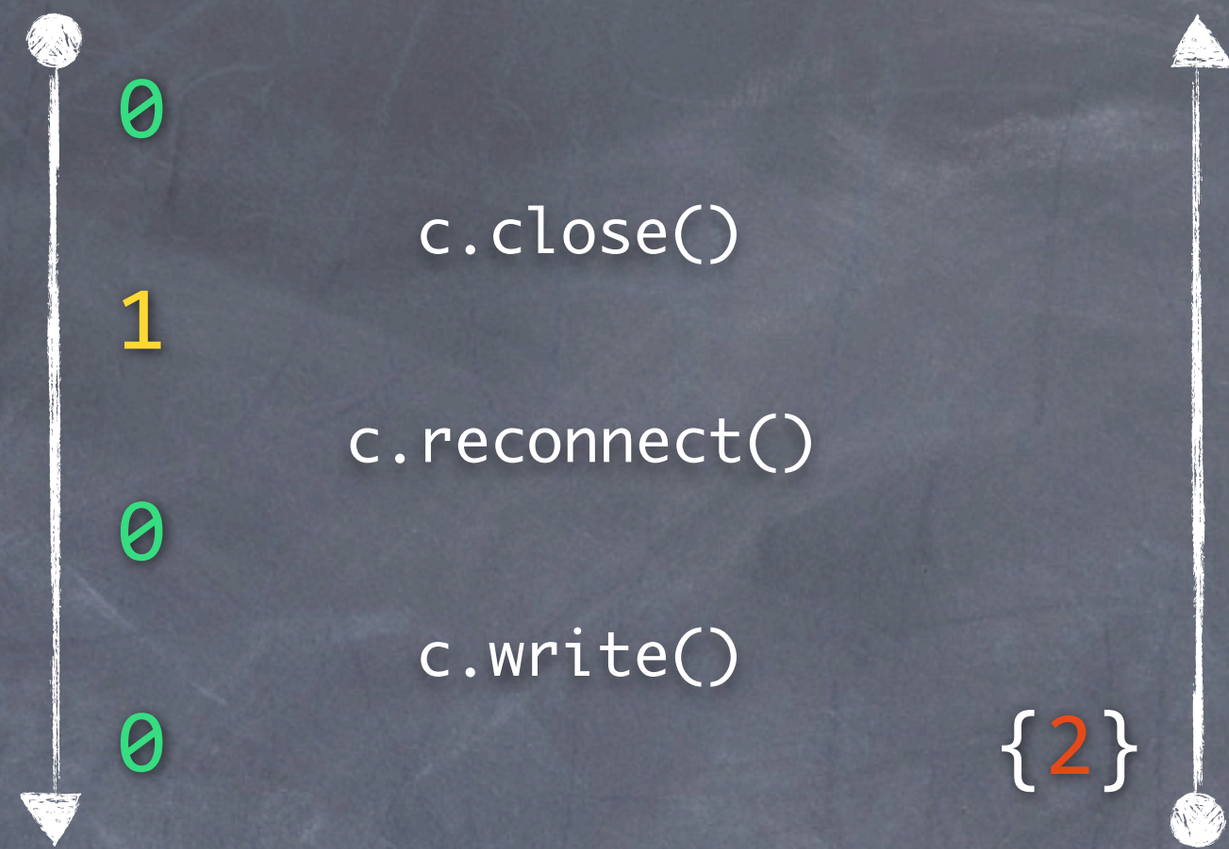
- Identify states that are equivalent at s.
- If s may transition only between equivalent states then disable transitions at s.

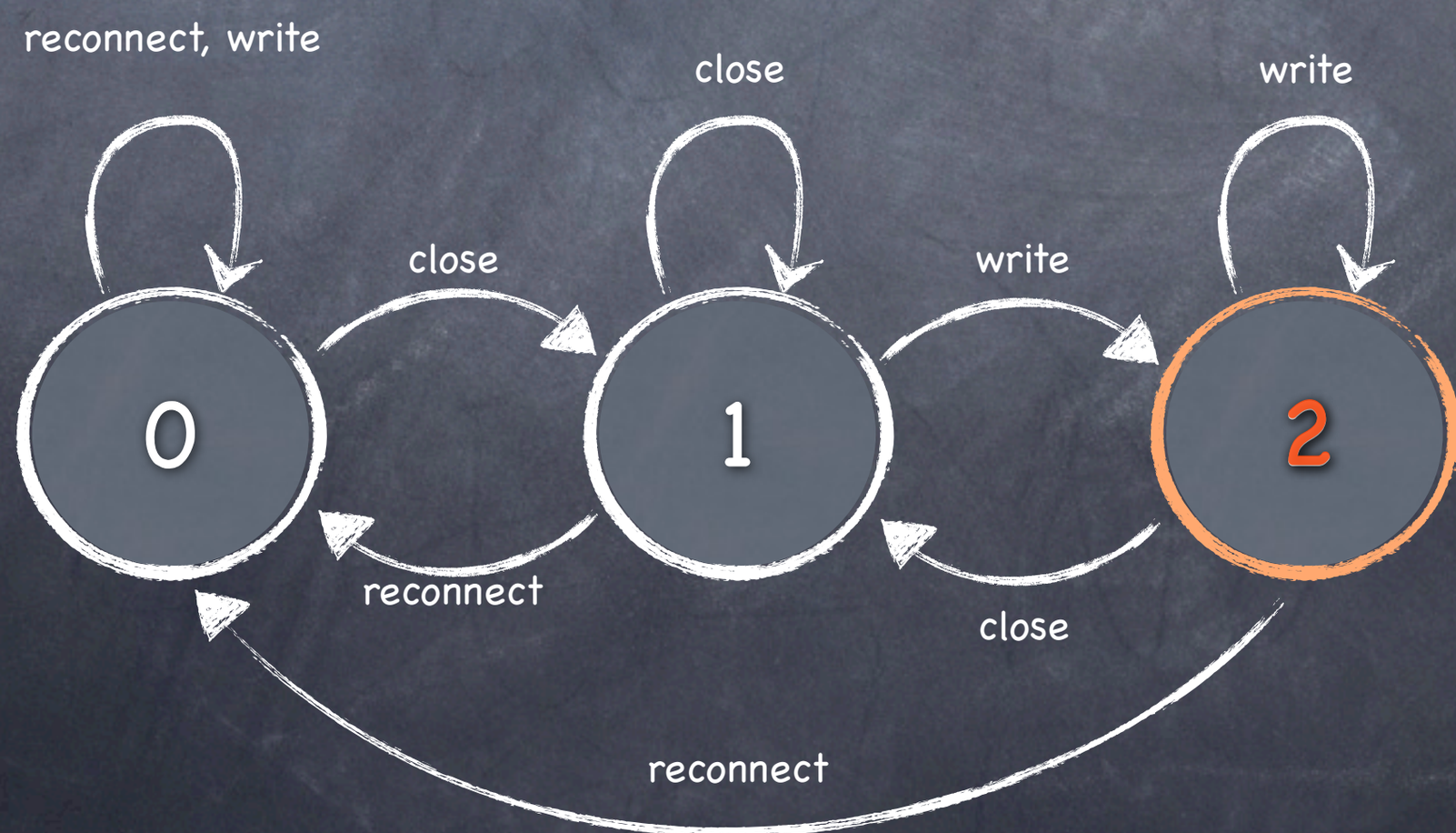
```
c.close()

c.reconnect()

c.write()
```

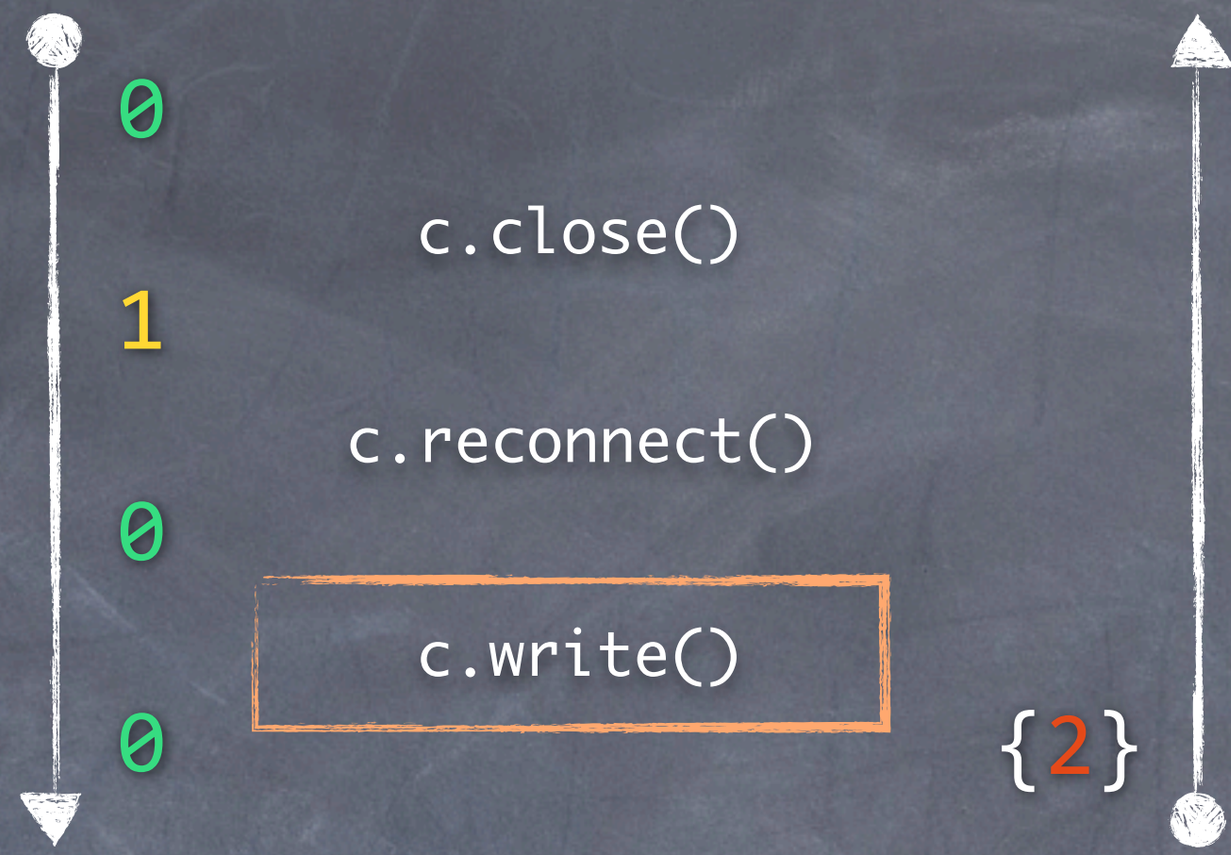


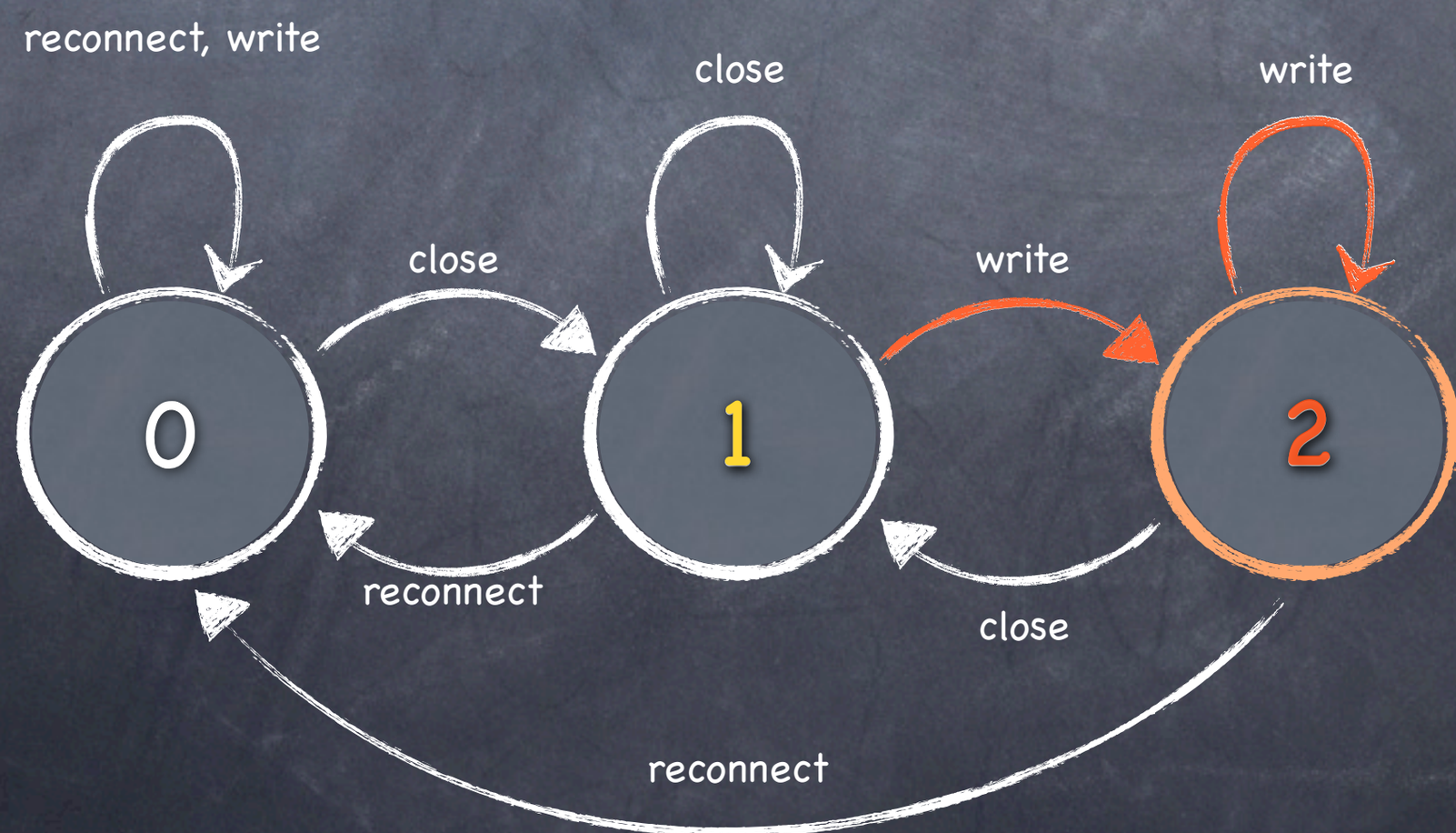

0
c.close()
1
c.reconnect()
0
c.write()
0

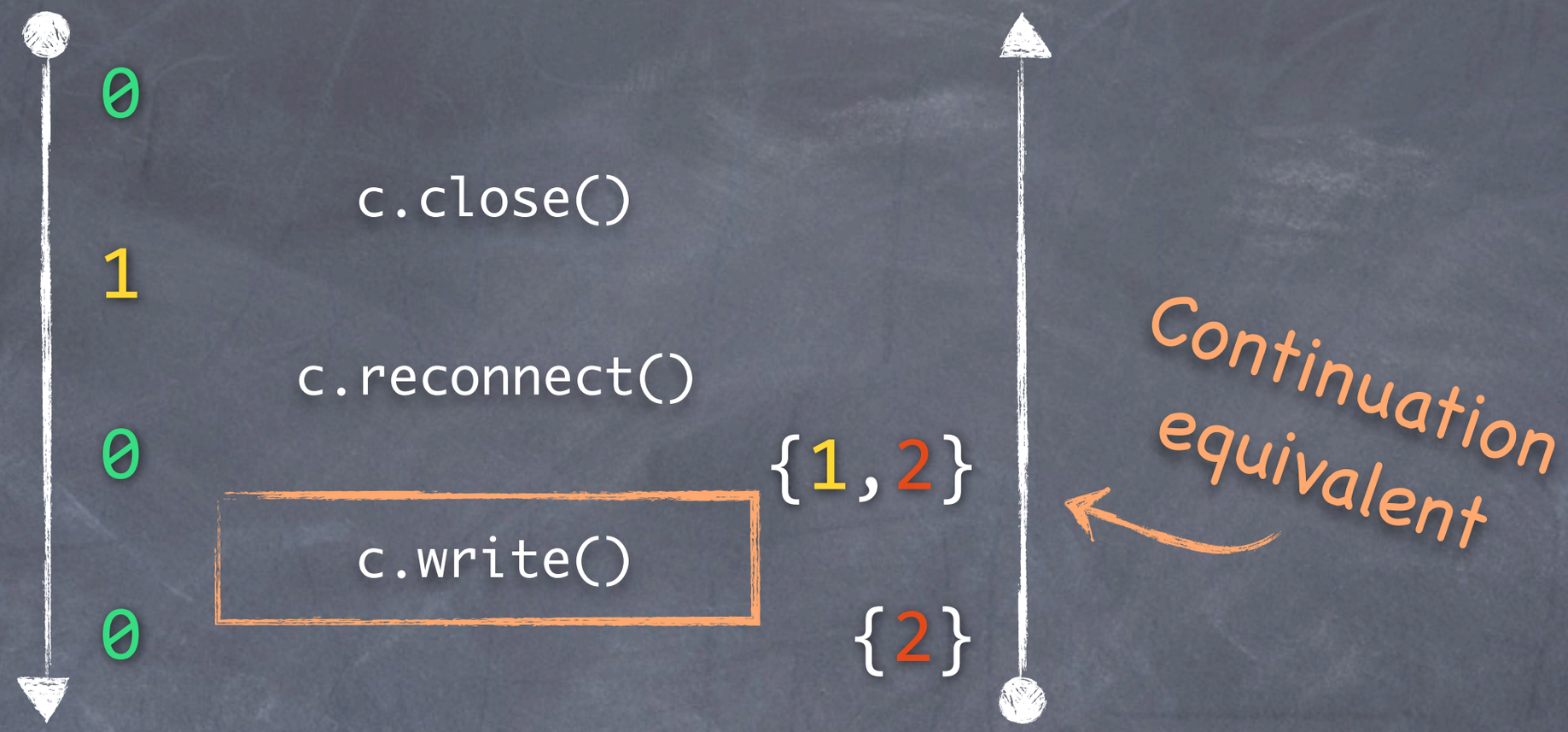

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

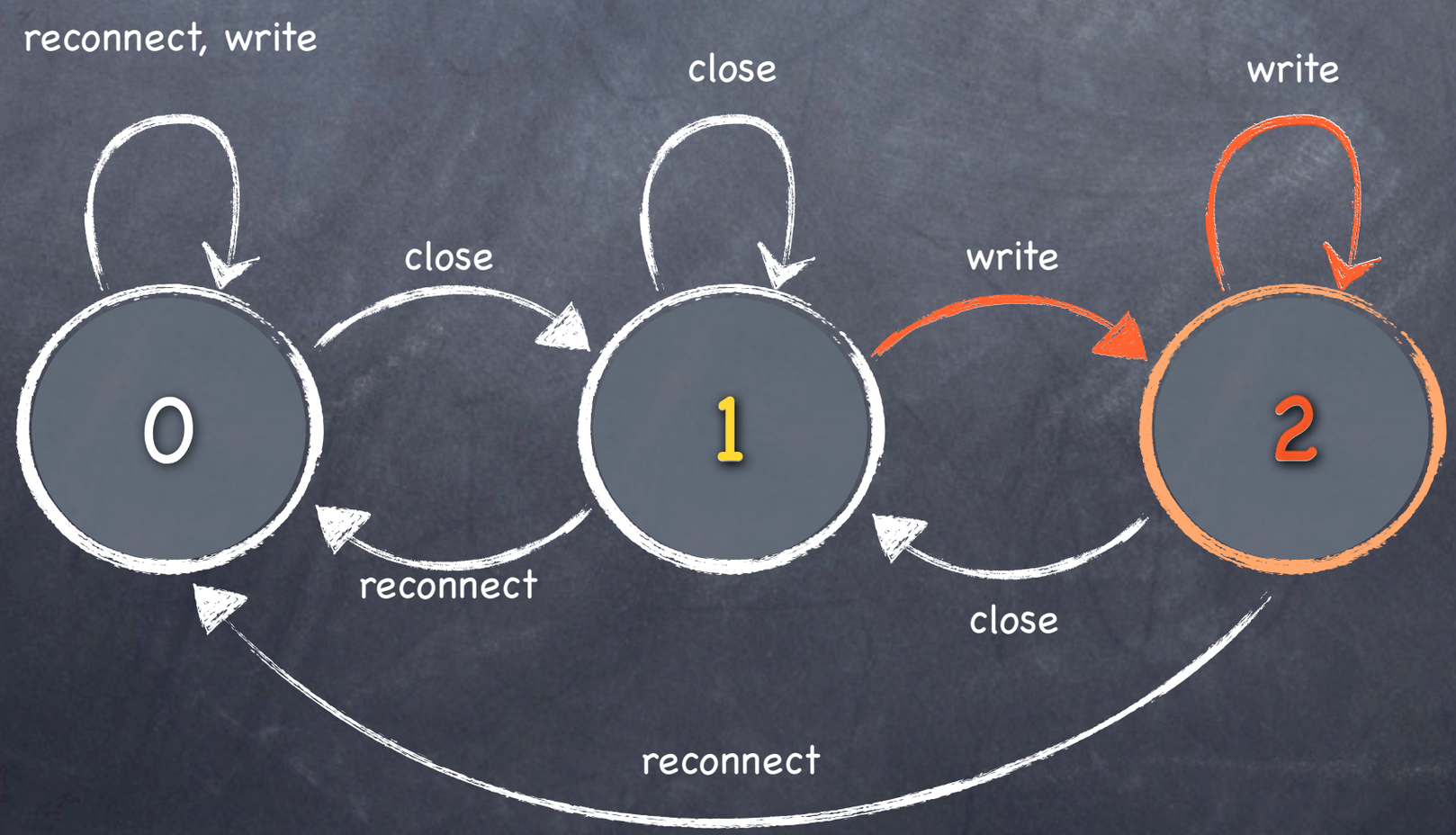

0
c.close()
1
c.reconnect()
0
c.write()
0
{2}

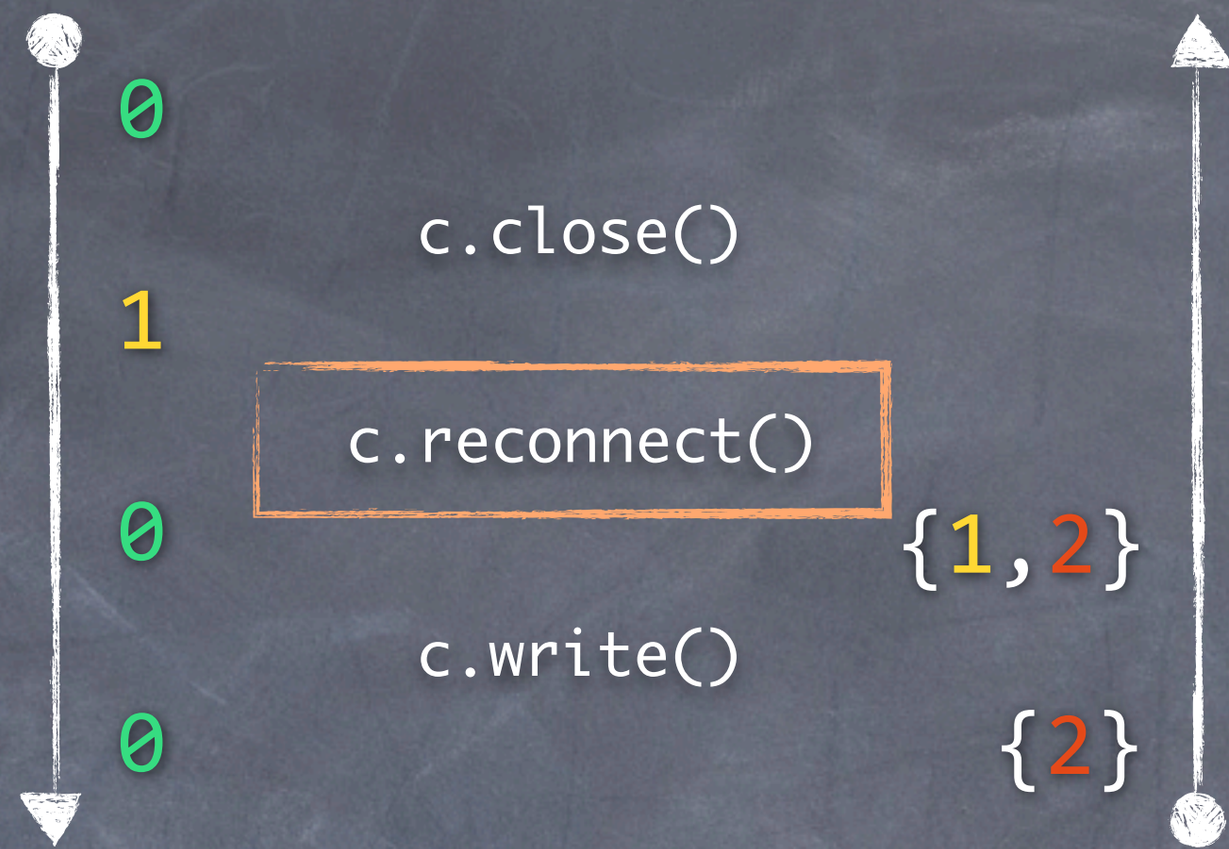

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

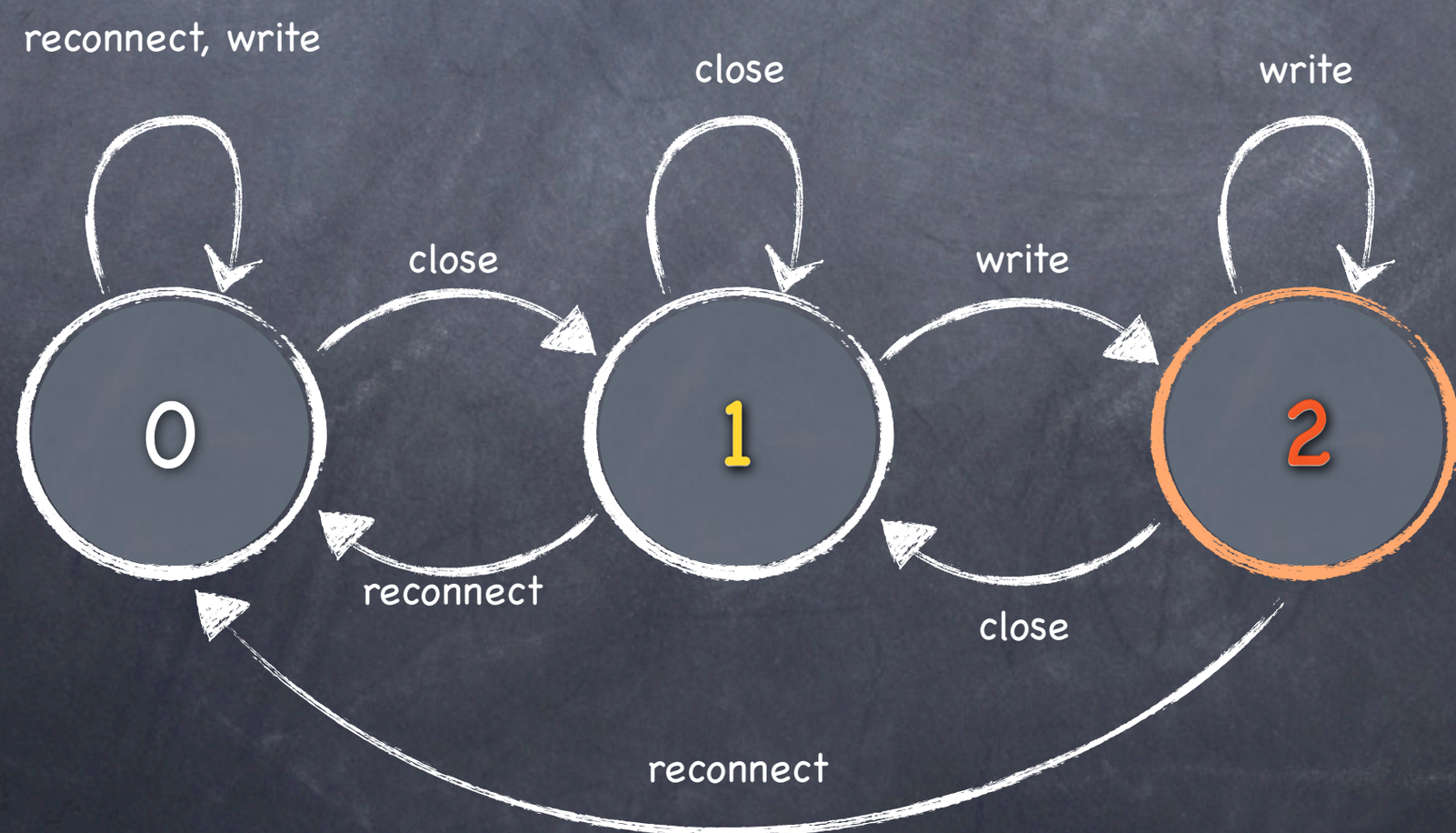

0
c.close()
1
c.reconnect()
0
c.write()
0
{2}

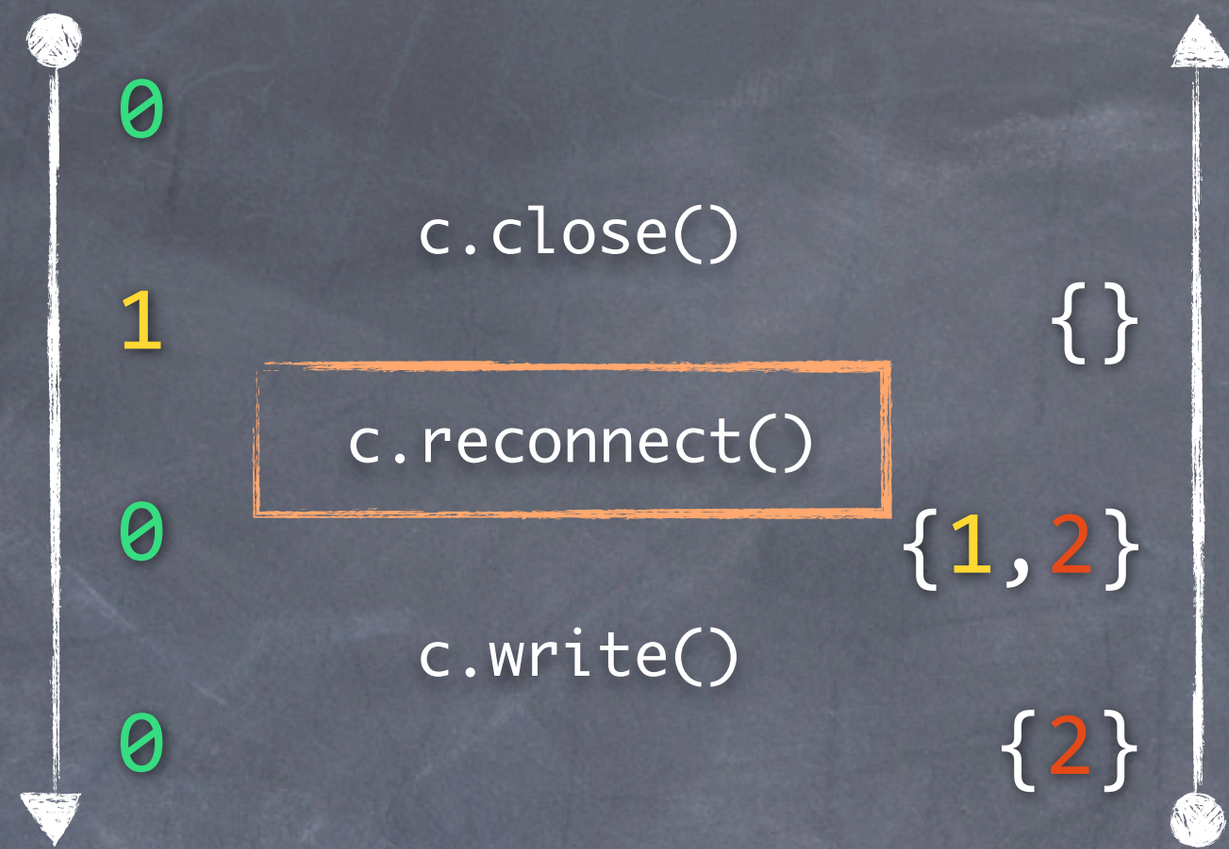

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

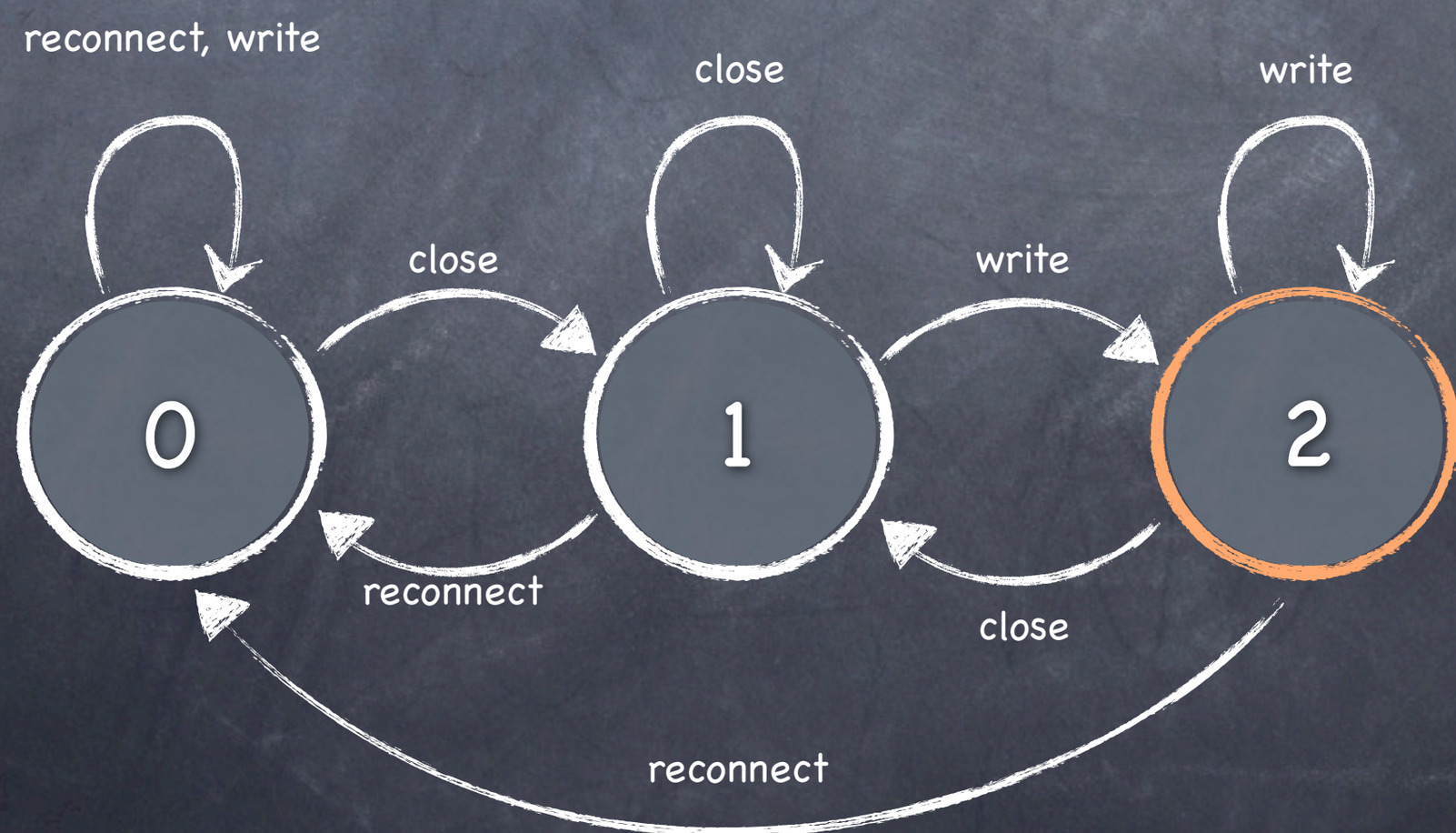

0
c.close()
1
c.reconnect()
0
{1,2}
c.write()
0
{2}
Continuation equivalent

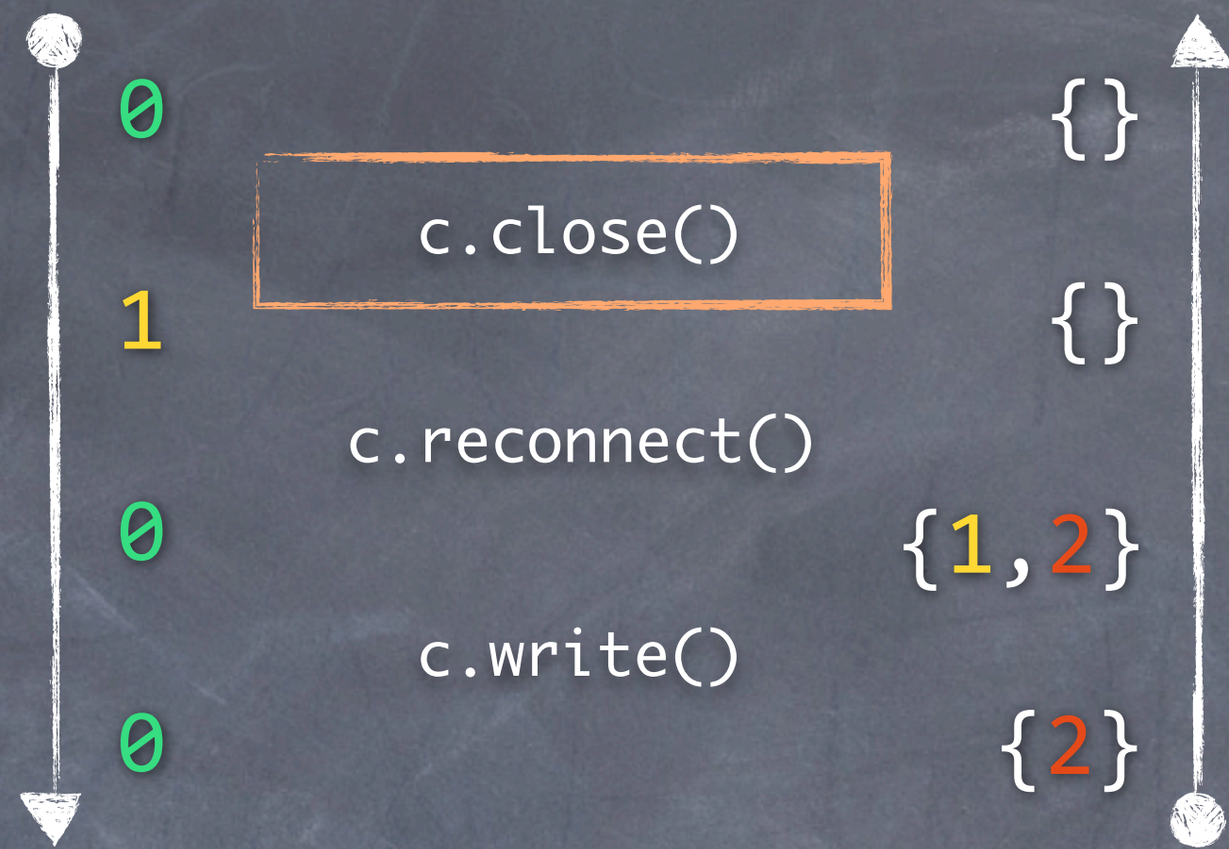

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

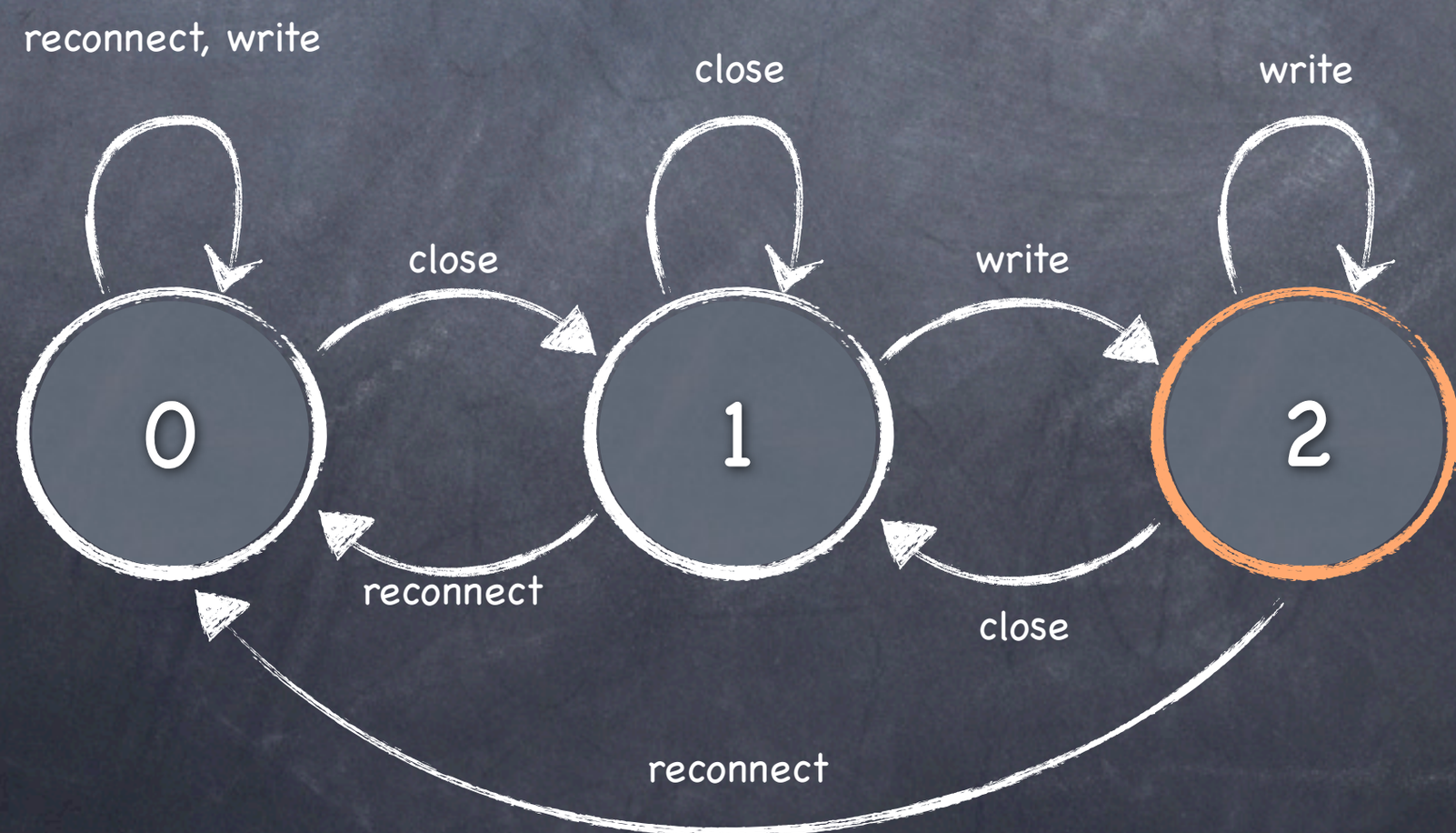

0
c.close()
1
c.reconnect()
0
{1,2}
c.write()
0
{2}

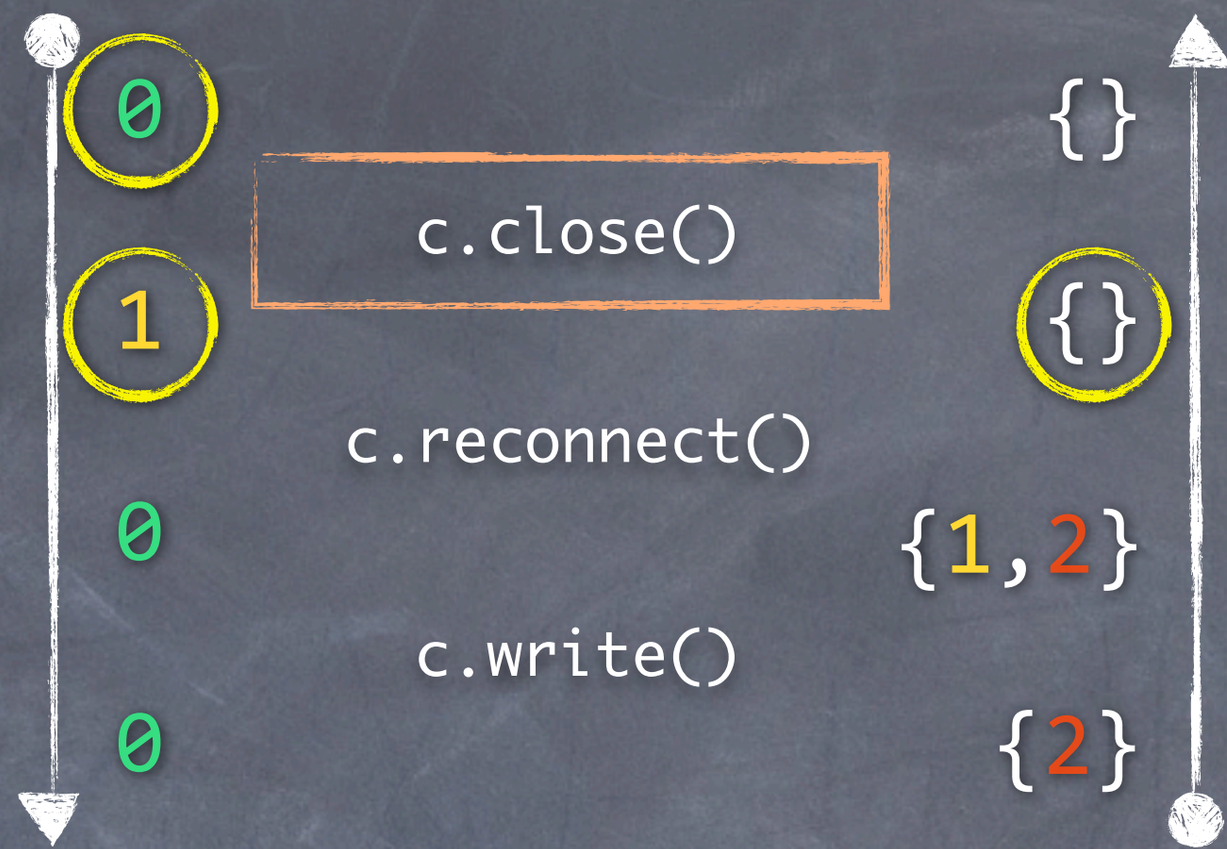

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

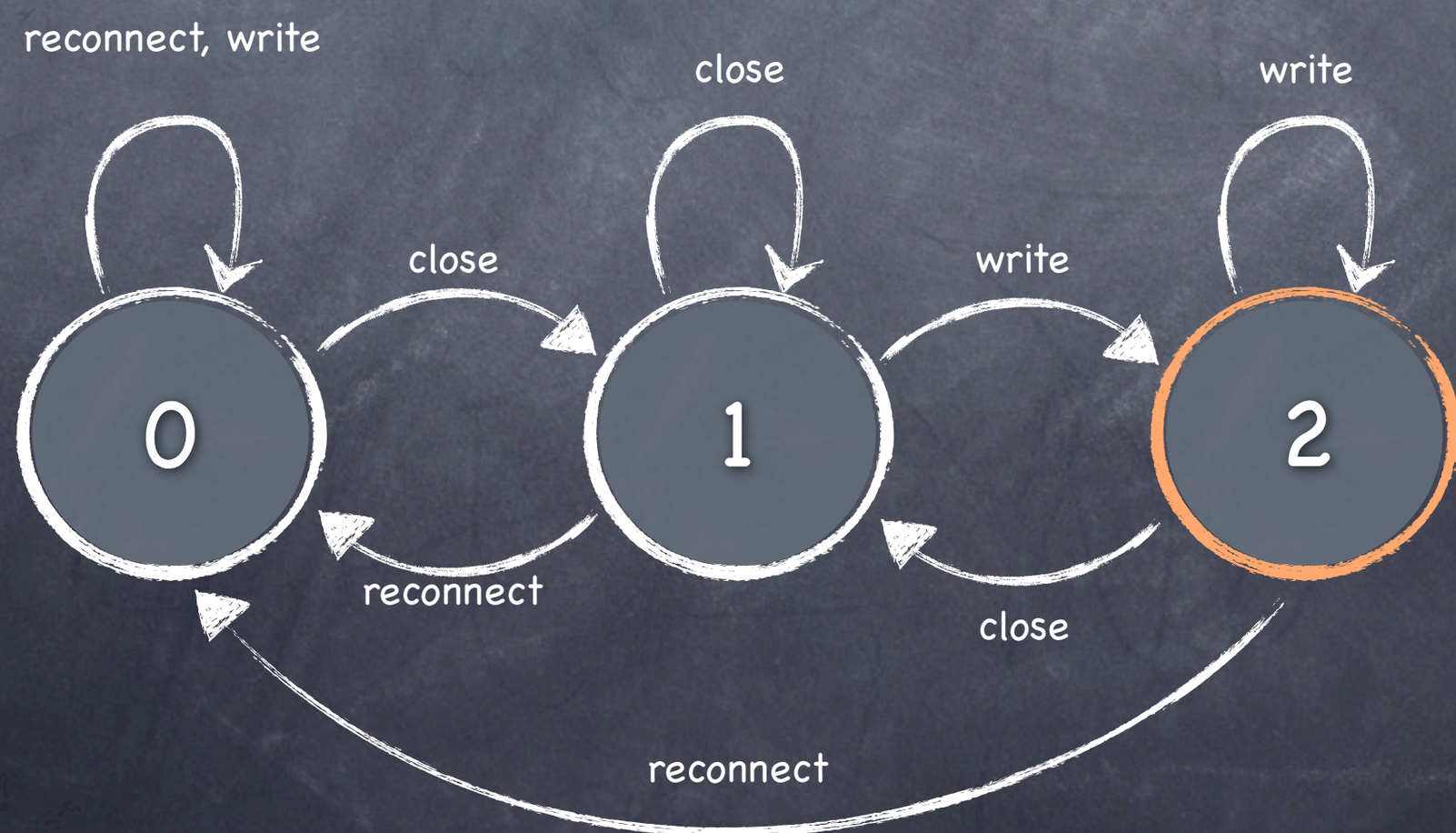

0
c.close()
1
{}
c.reconnect()
0
{1,2}
c.write()
0
{2}

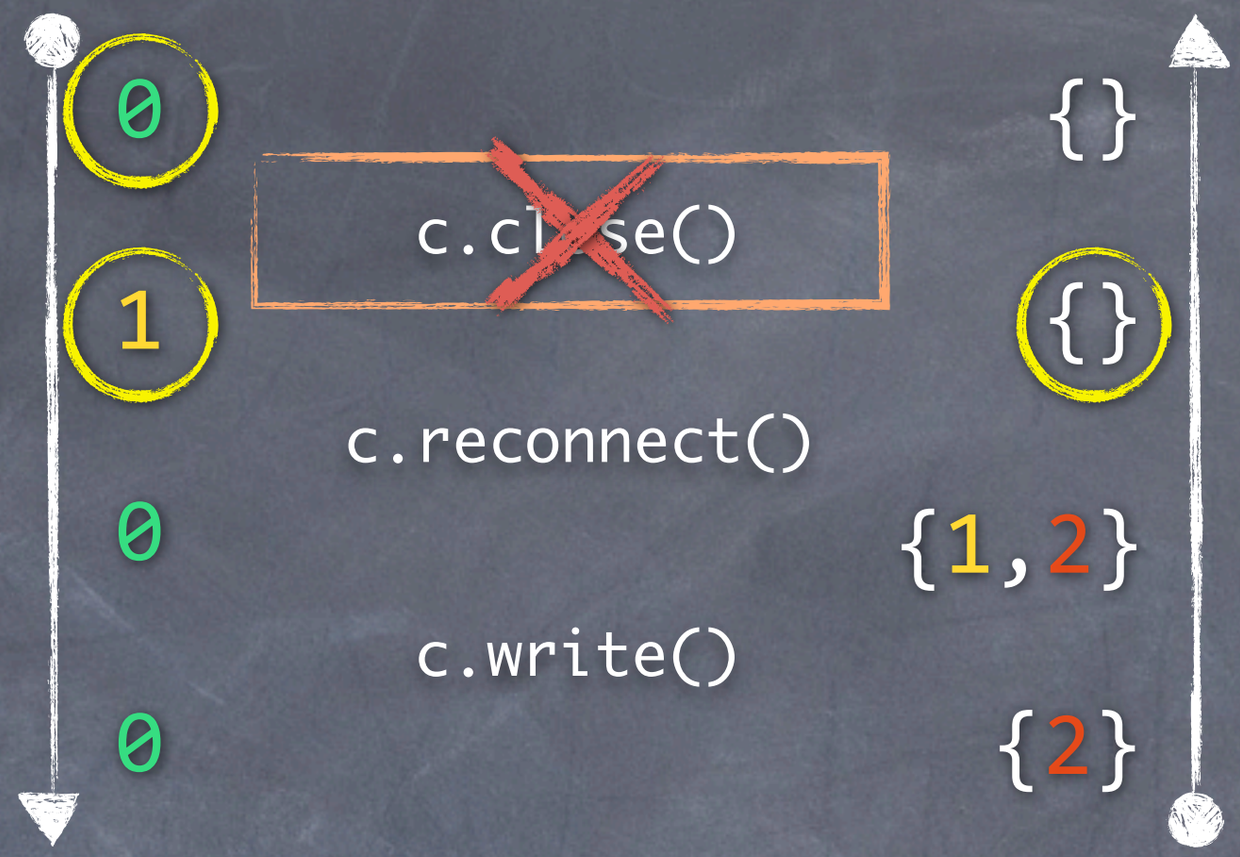

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

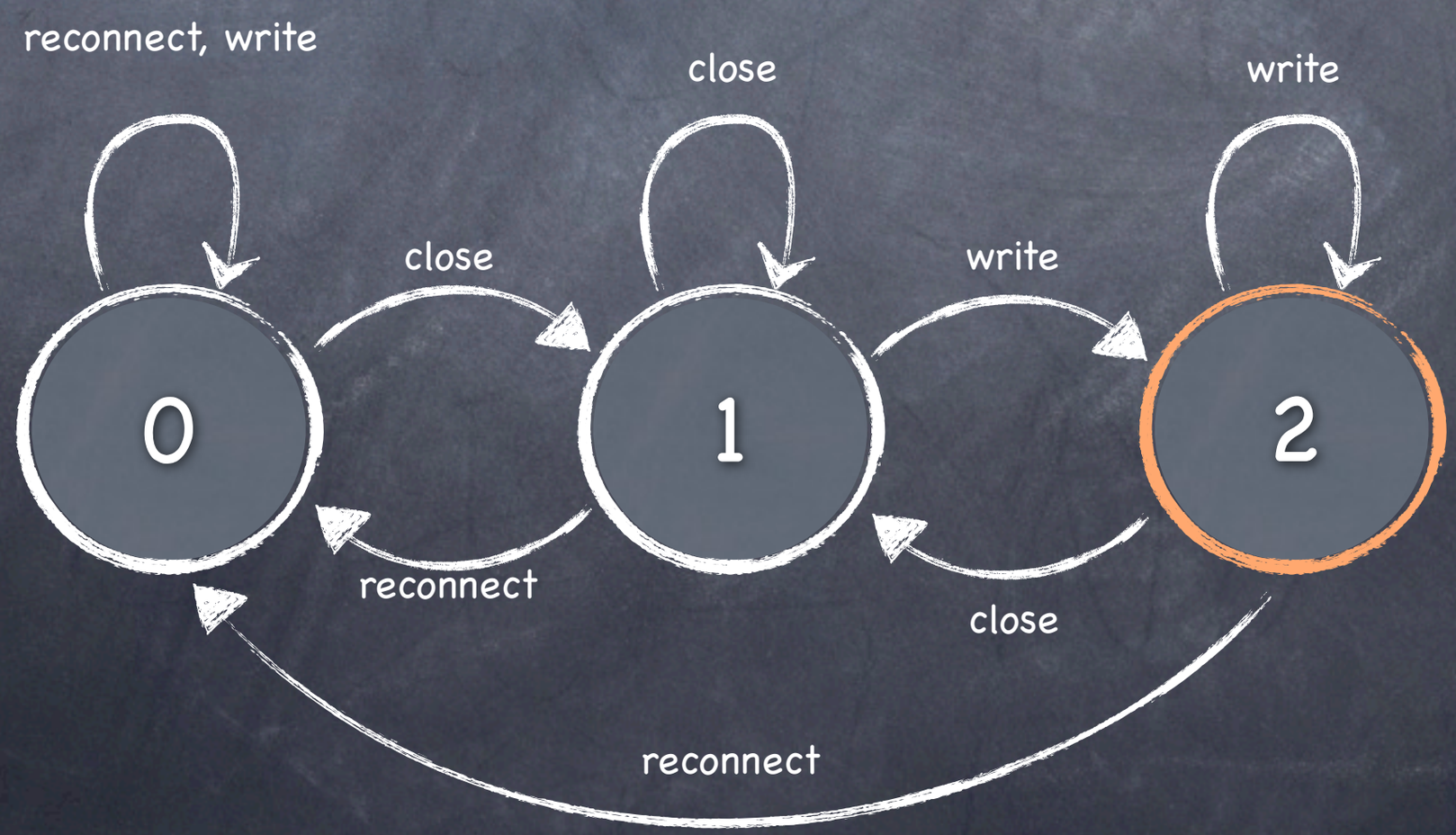

0
{}
c.close()
1
{}
c.reconnect()
0
{1,2}
c.write()
0
{2}

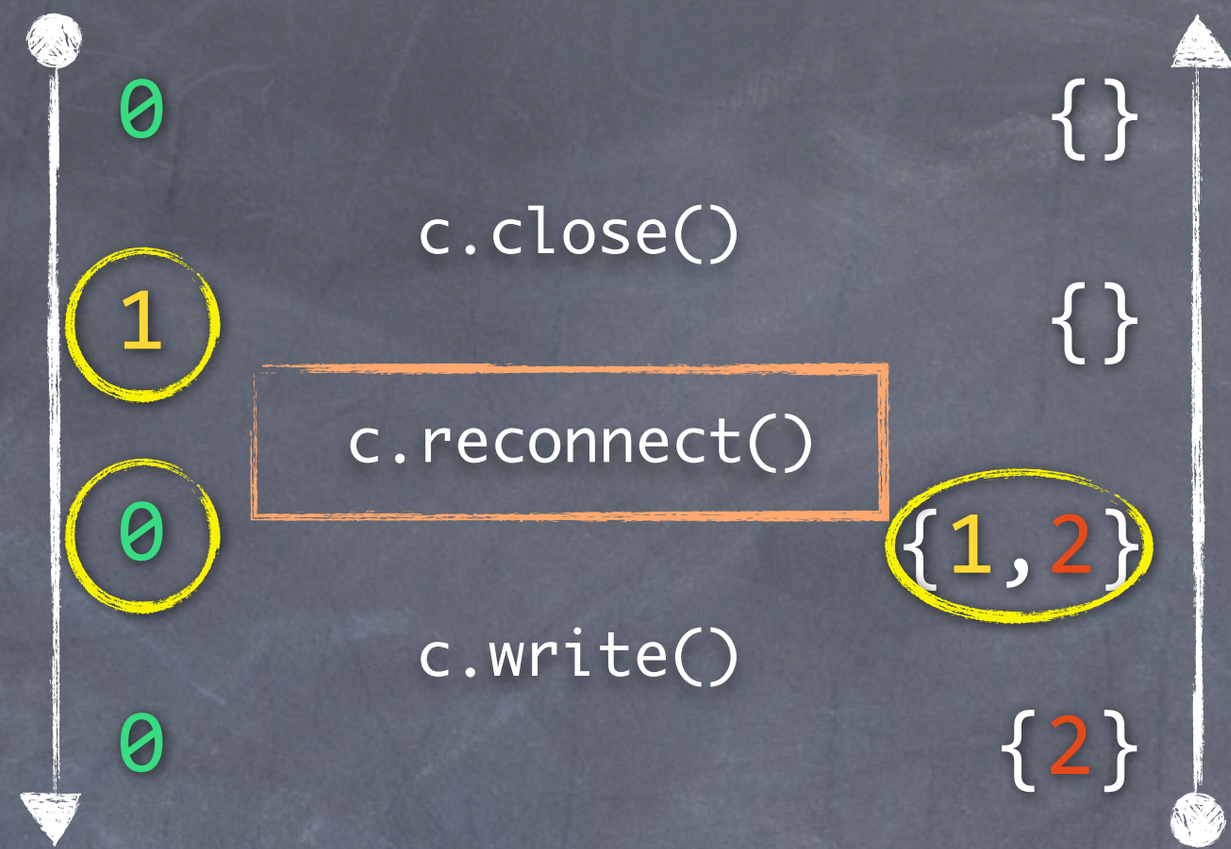

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

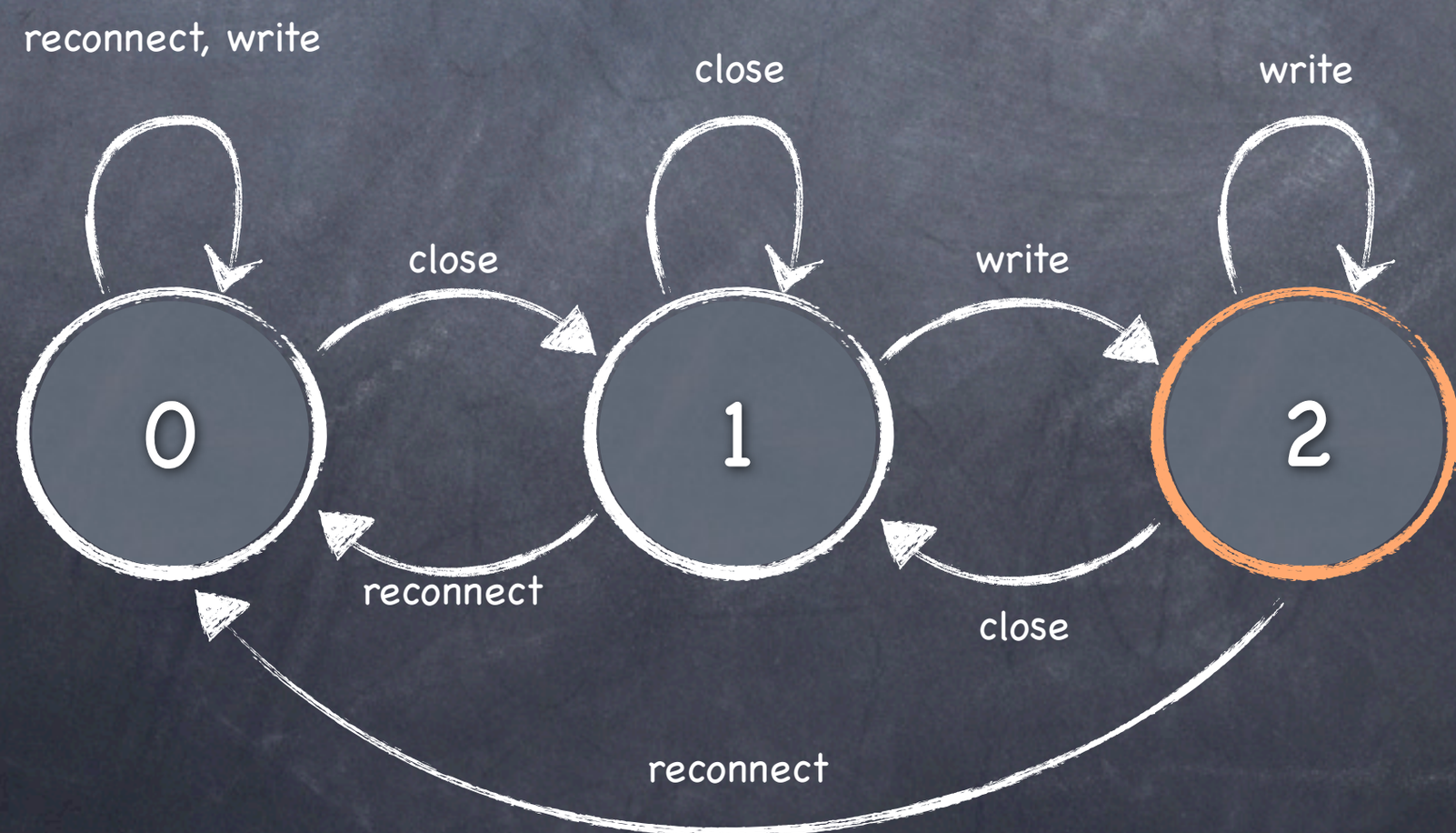

0
{}
c.close()
1
{}
c.reconnect()
0
{1,2}
c.write()
0
{2}

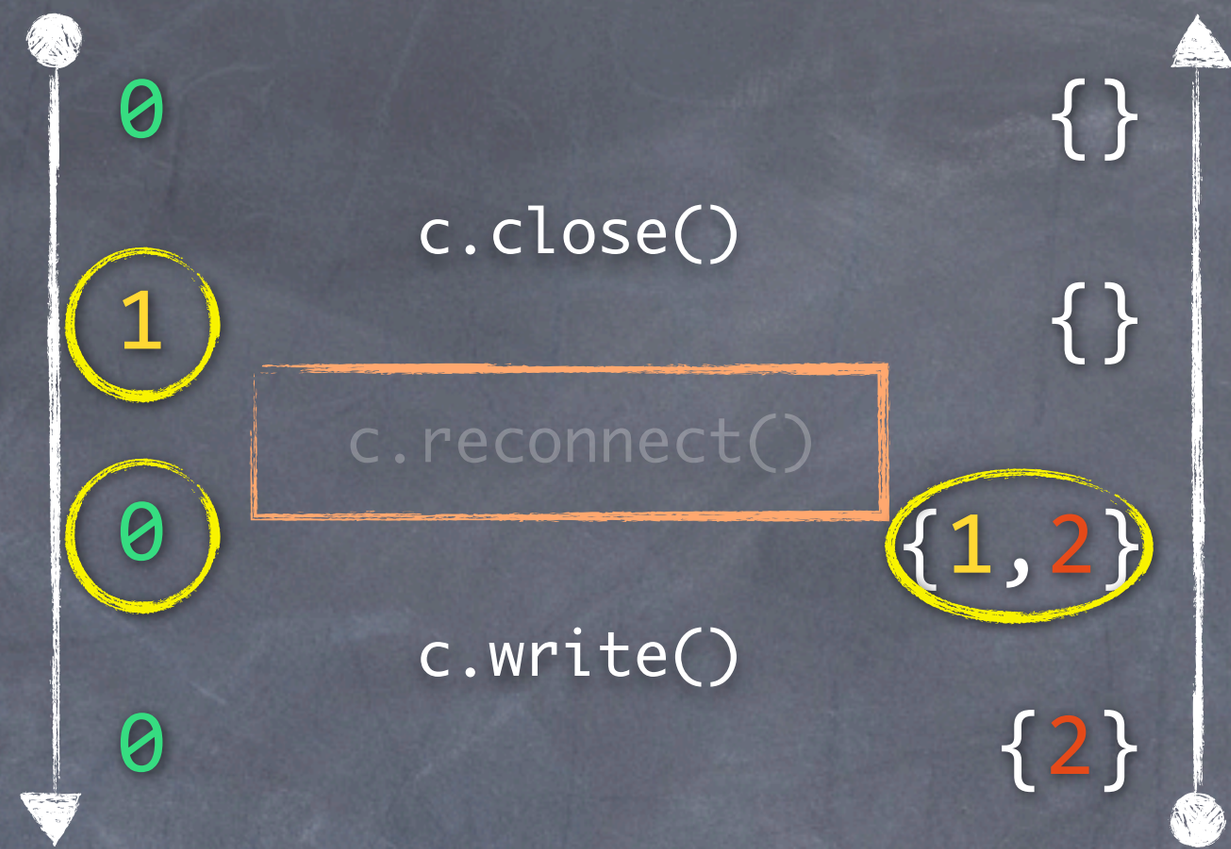

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

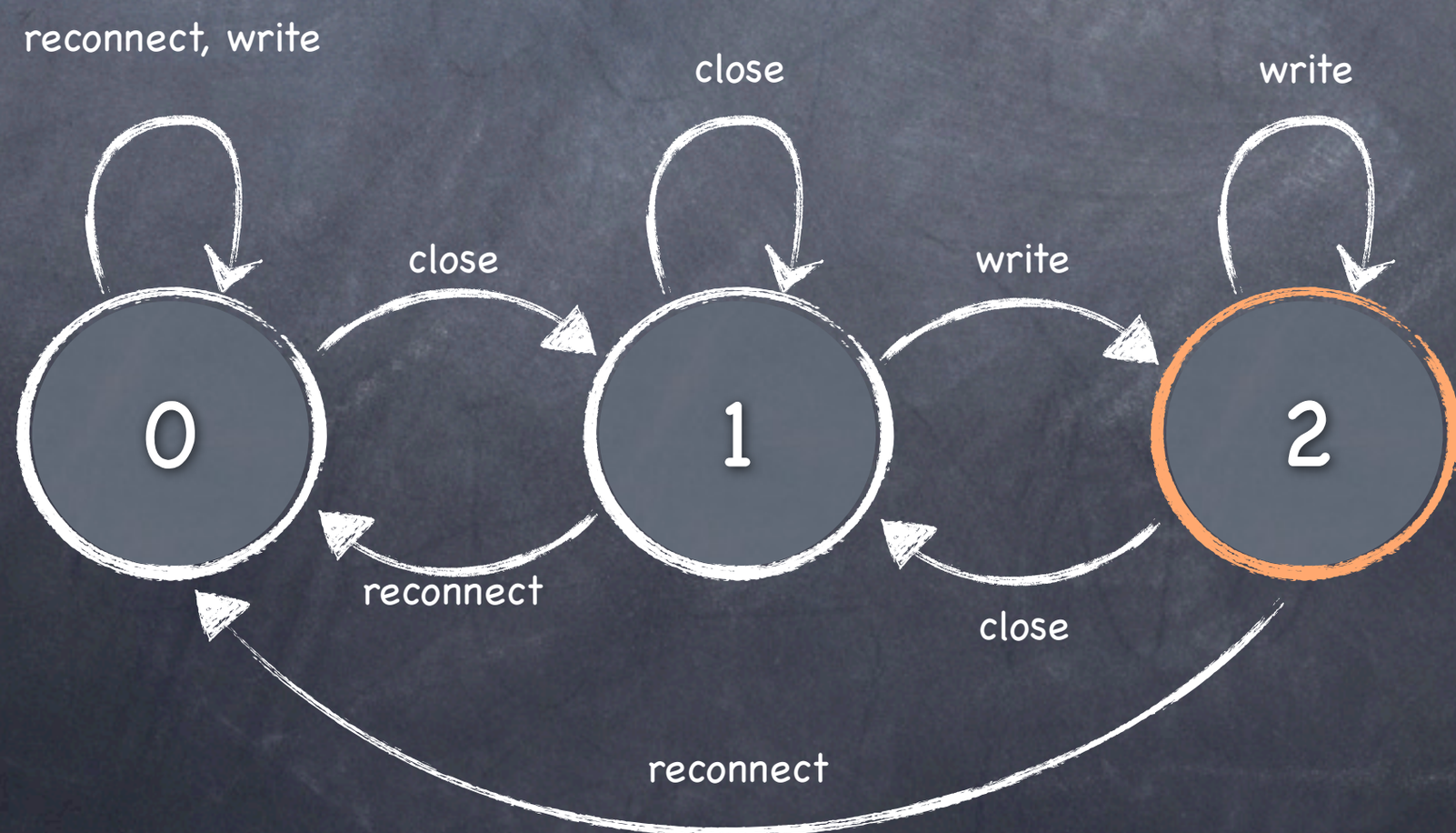

0
{}
c.close()
1
{}
c.reconnect()
0
{1,2}
c.write()
0
{2}

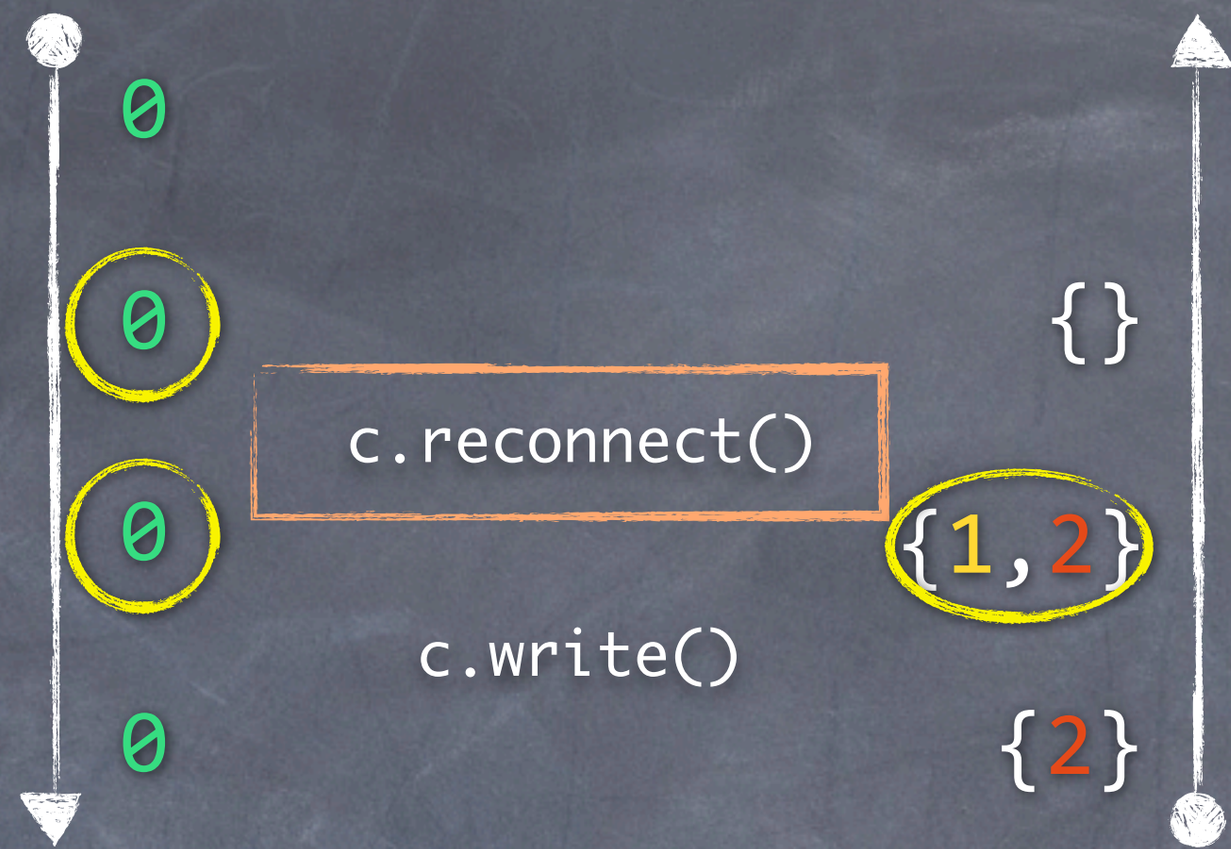

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

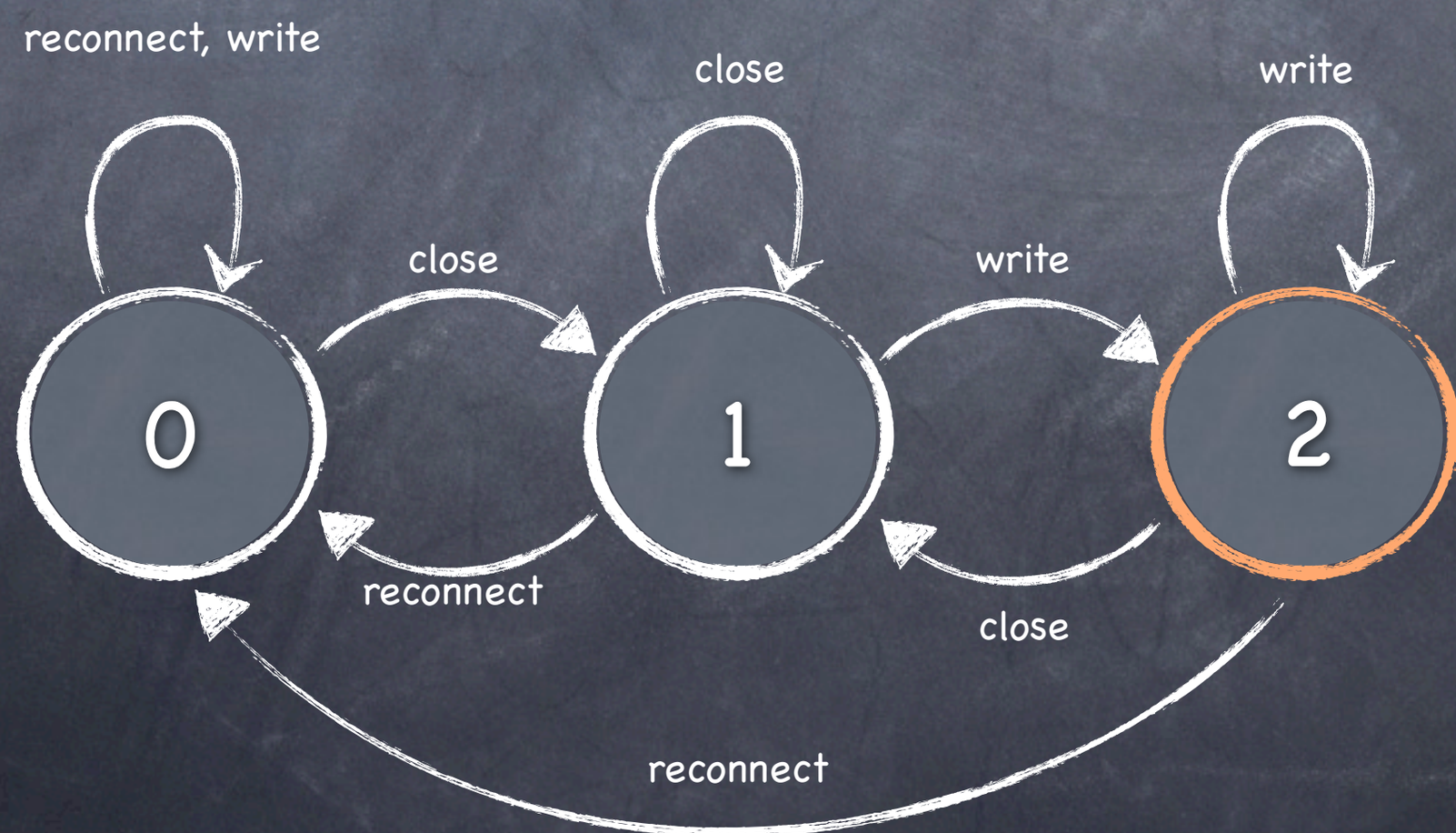

0
{}
c.close()
1
{}
c.reconnect()
0
{1,2}
c.write()
0
{2}

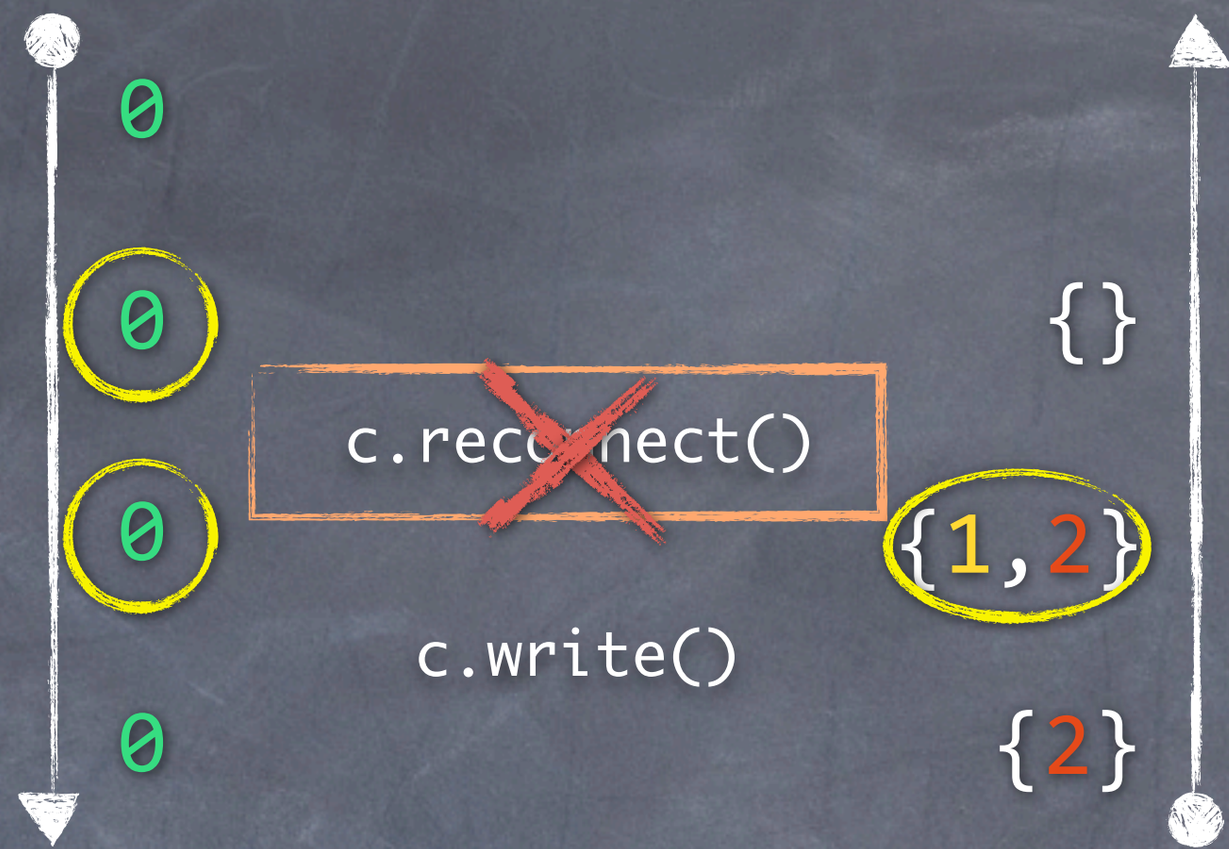

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

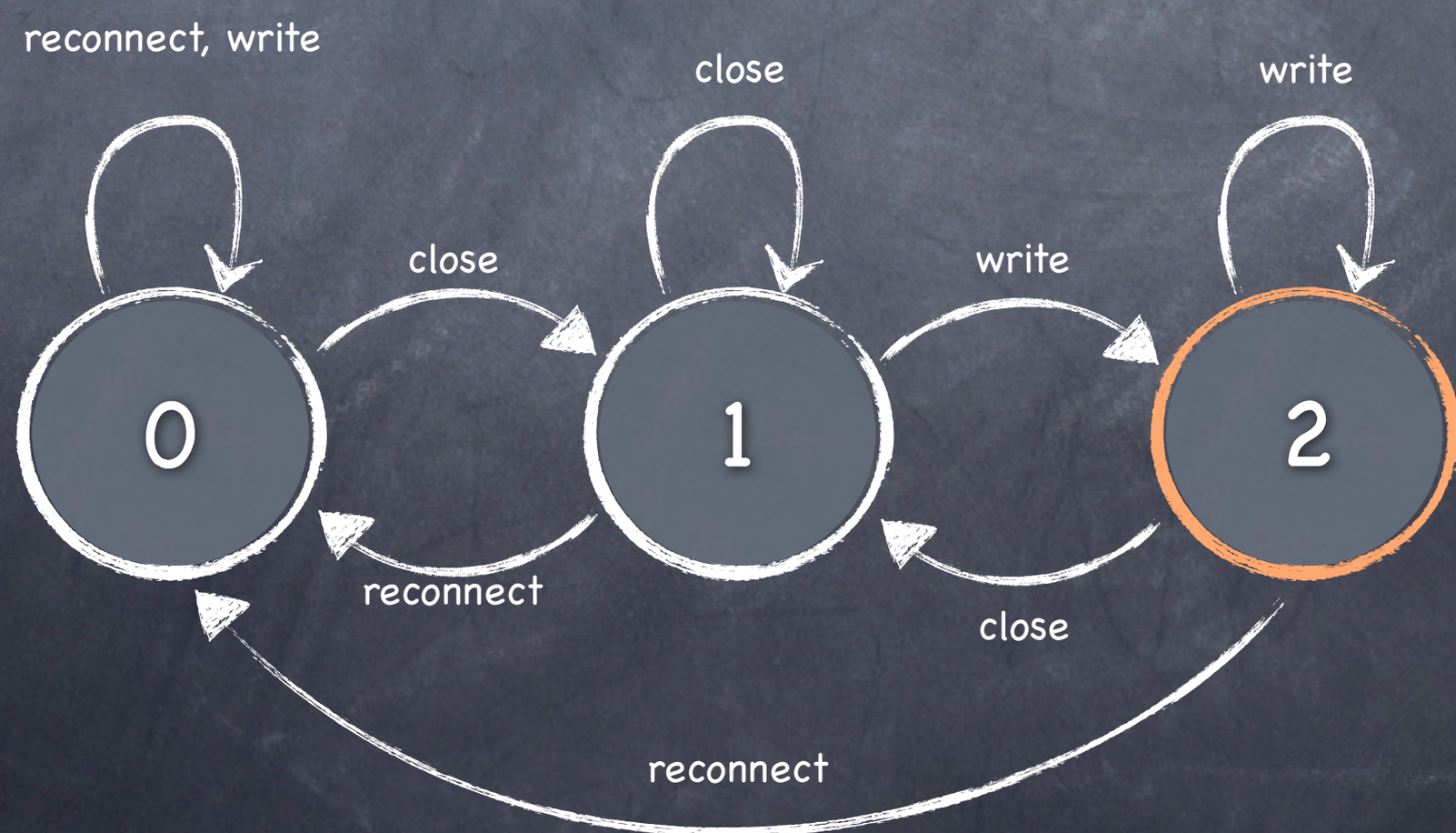

0
{}
c.close()
1
{}
c.reconnect()
0
{1,2}
c.write()
0
{2}

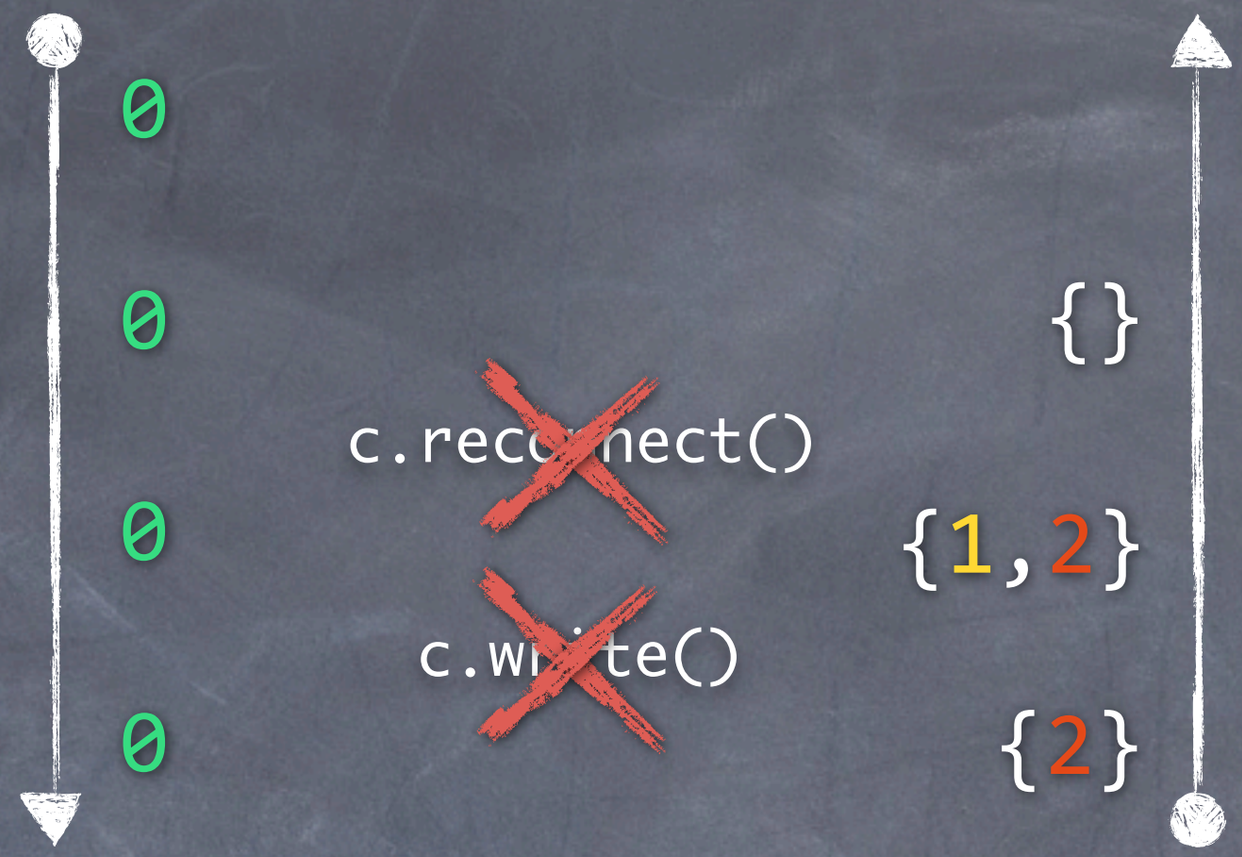

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

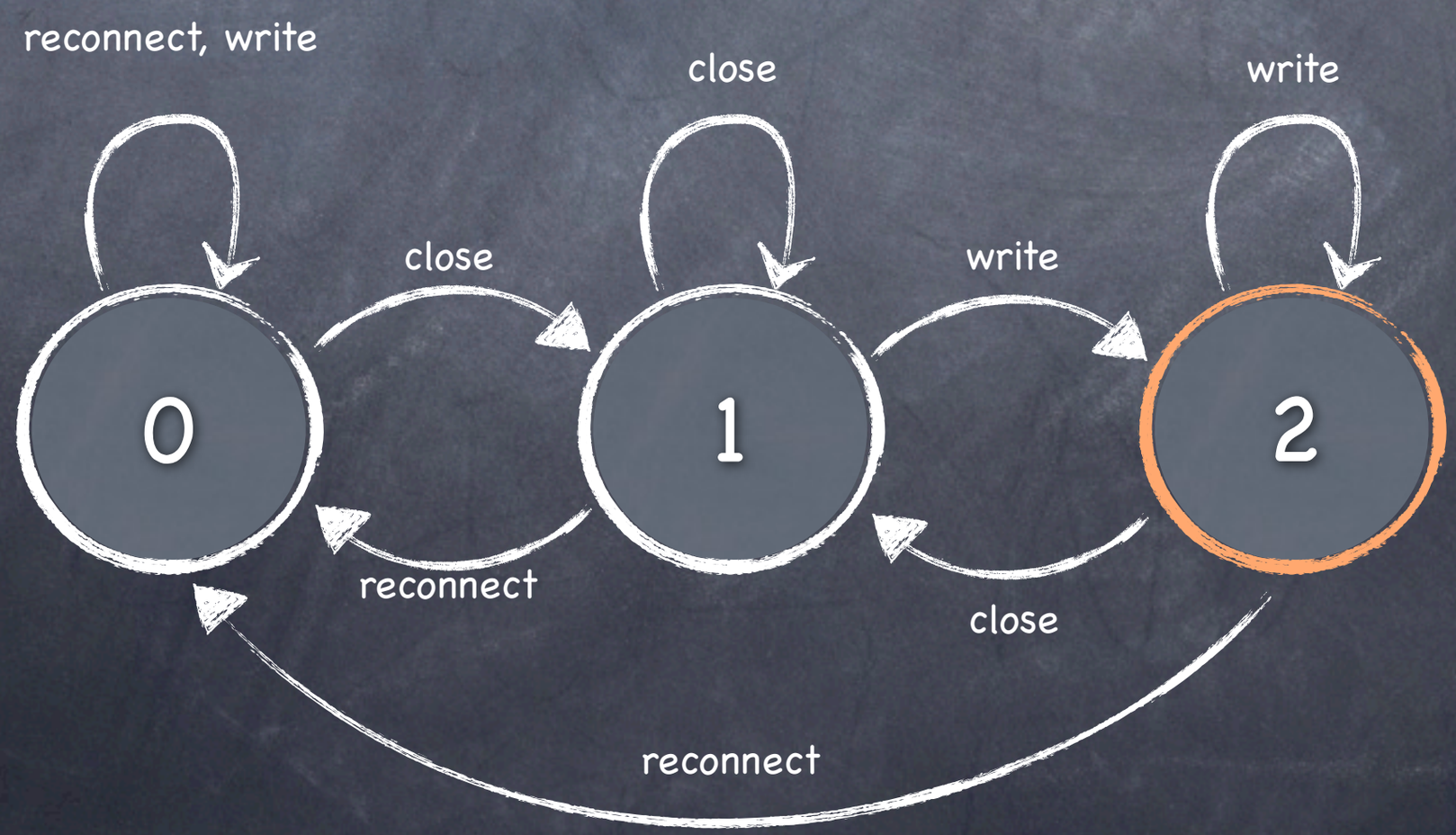

0
0
{}
c.reconnect()
0
{1,2}
c.write()
0
{2}

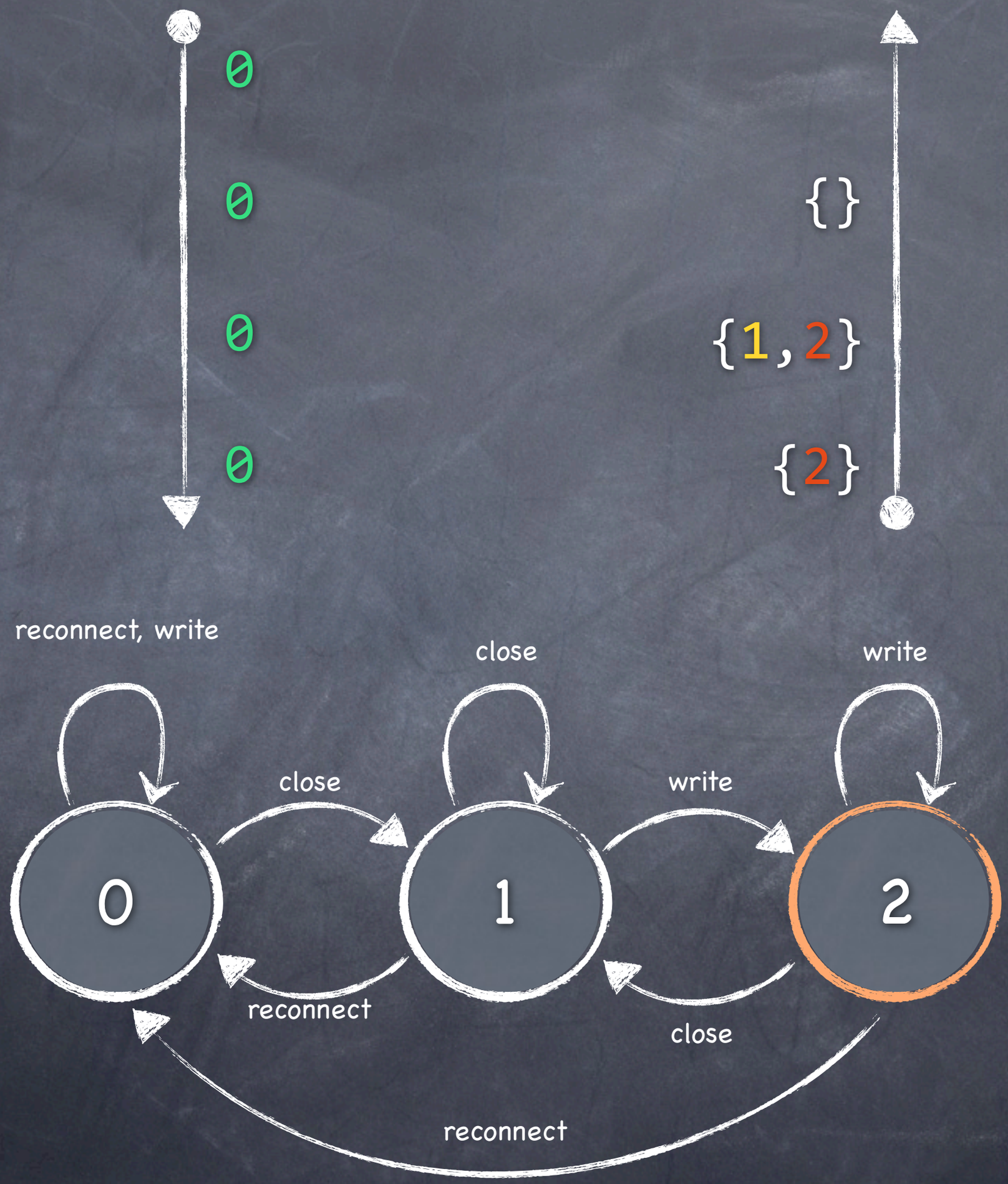

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect


0
0
0
0
c.reconnect()
c.write()
{}
{1,2}
{2}

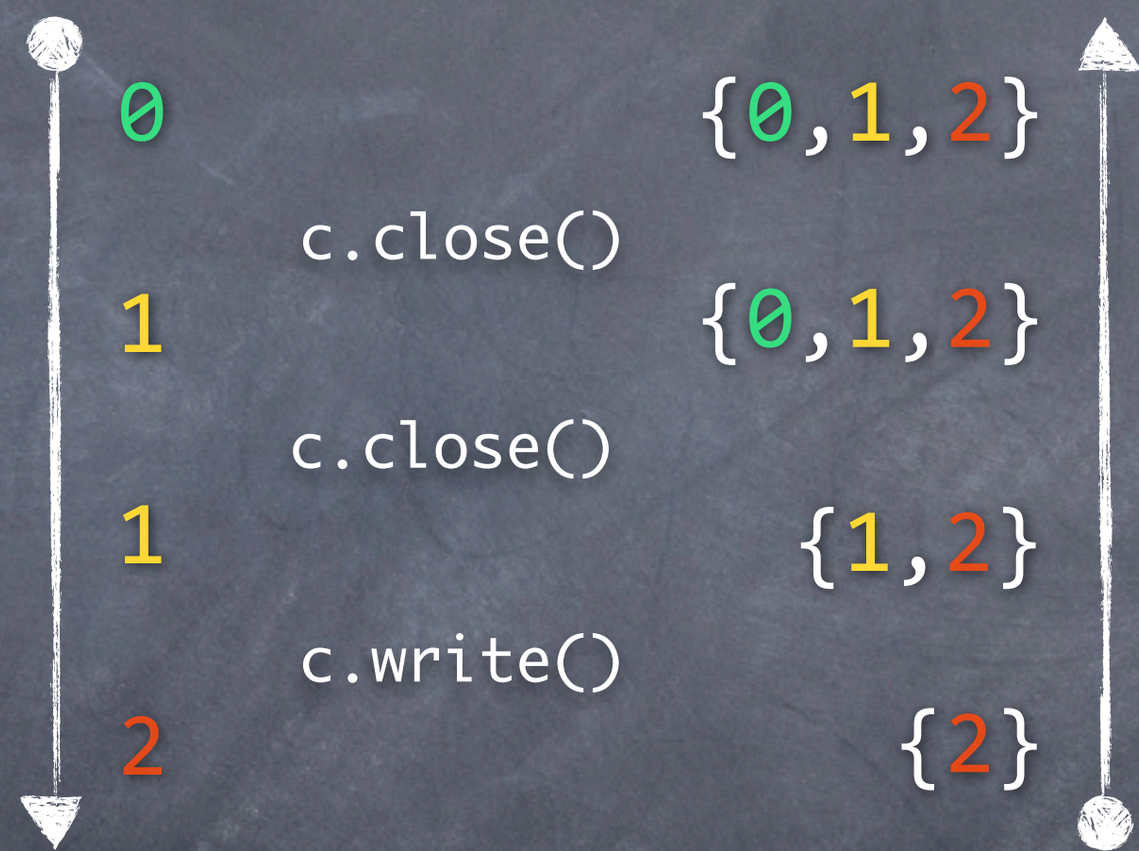

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect


0
0
0
0
c.reconnect()
c.write()
{}
{1,2}
{2}

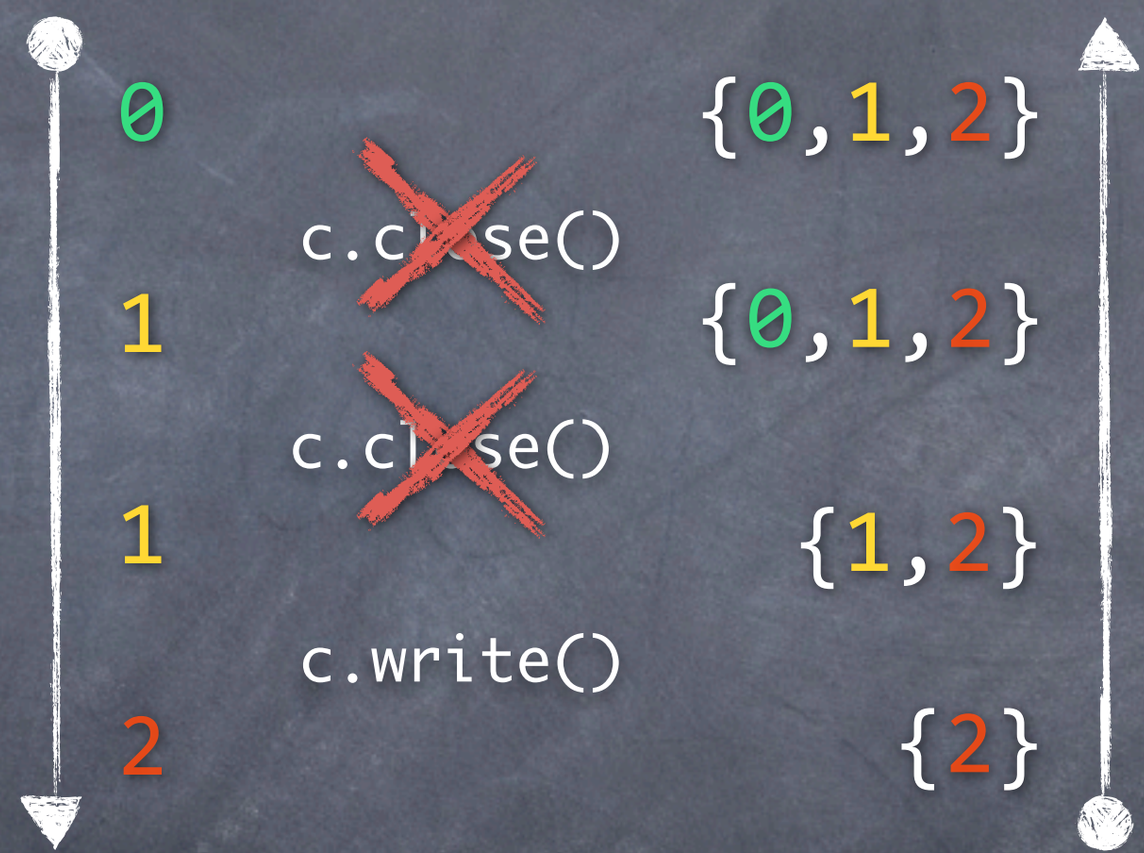

reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect


0
0
0
0
{}
{1,2}
{2}
reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect

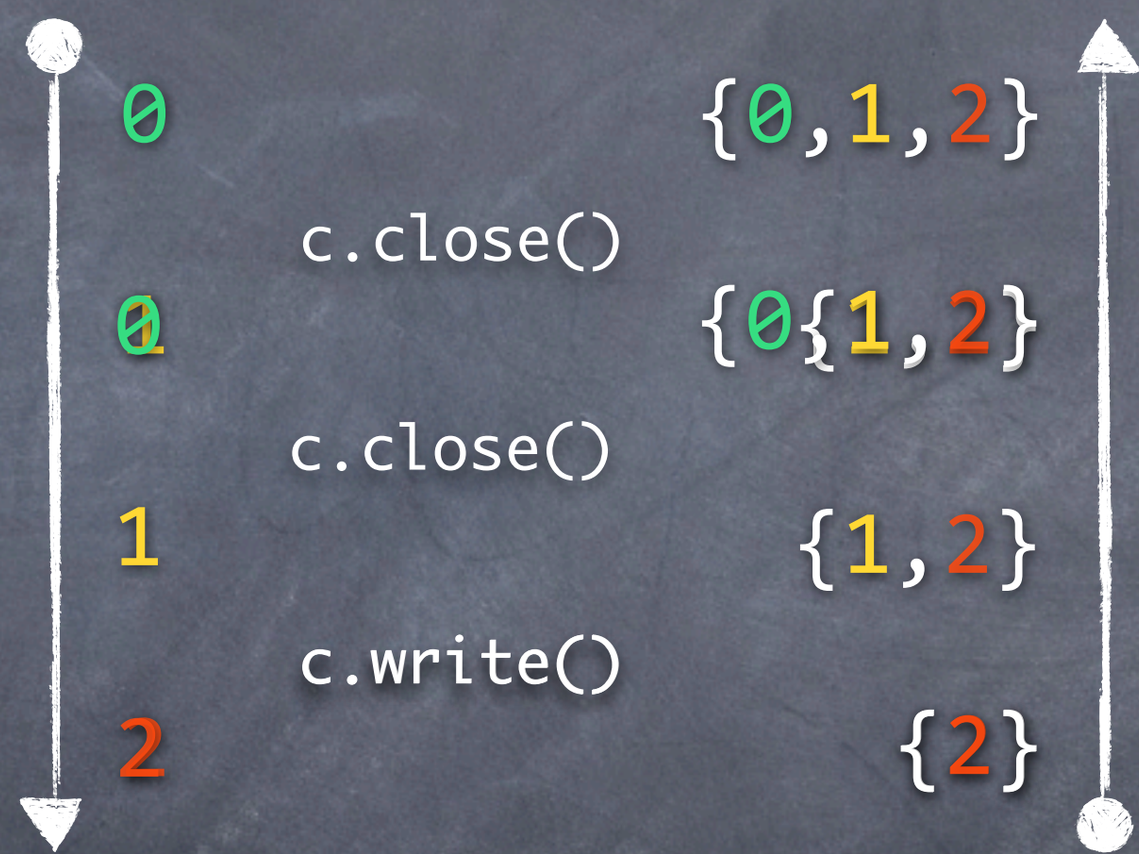
Algorithm is greedy



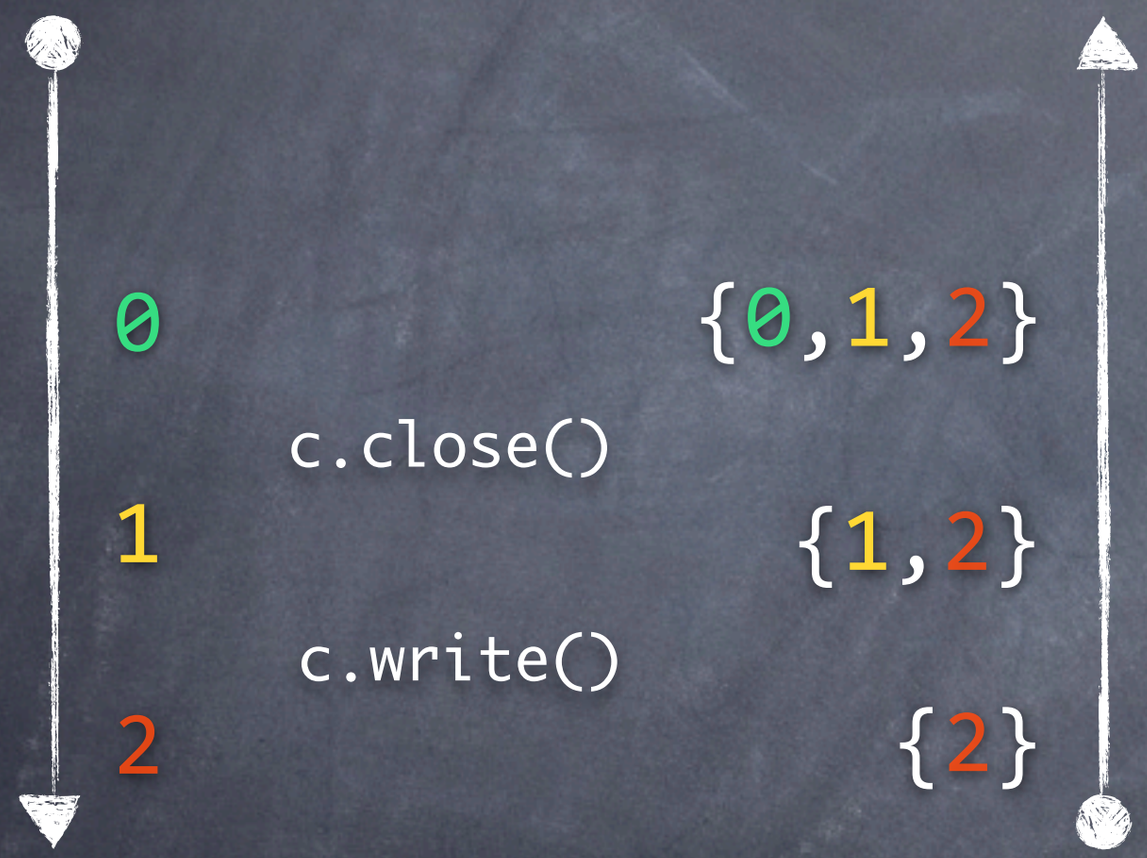
Algorithm is greedy

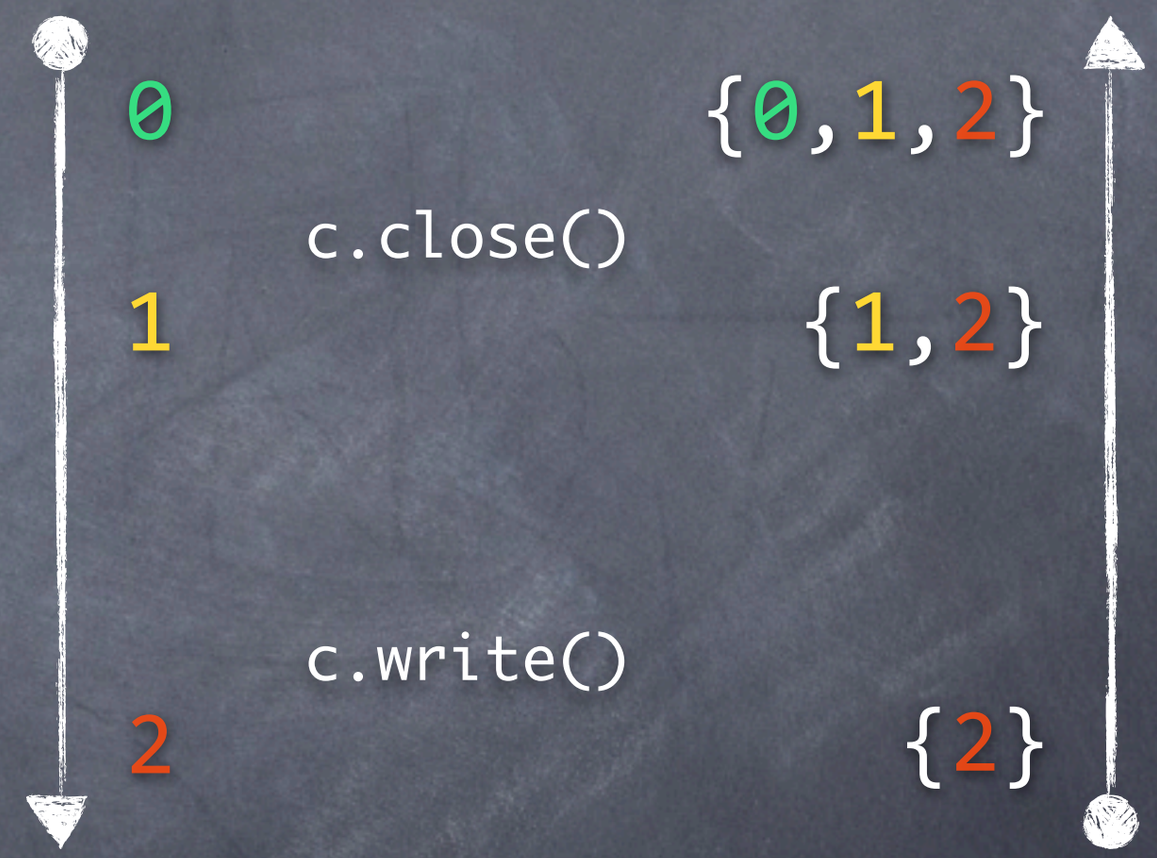


Algorithm is greedy



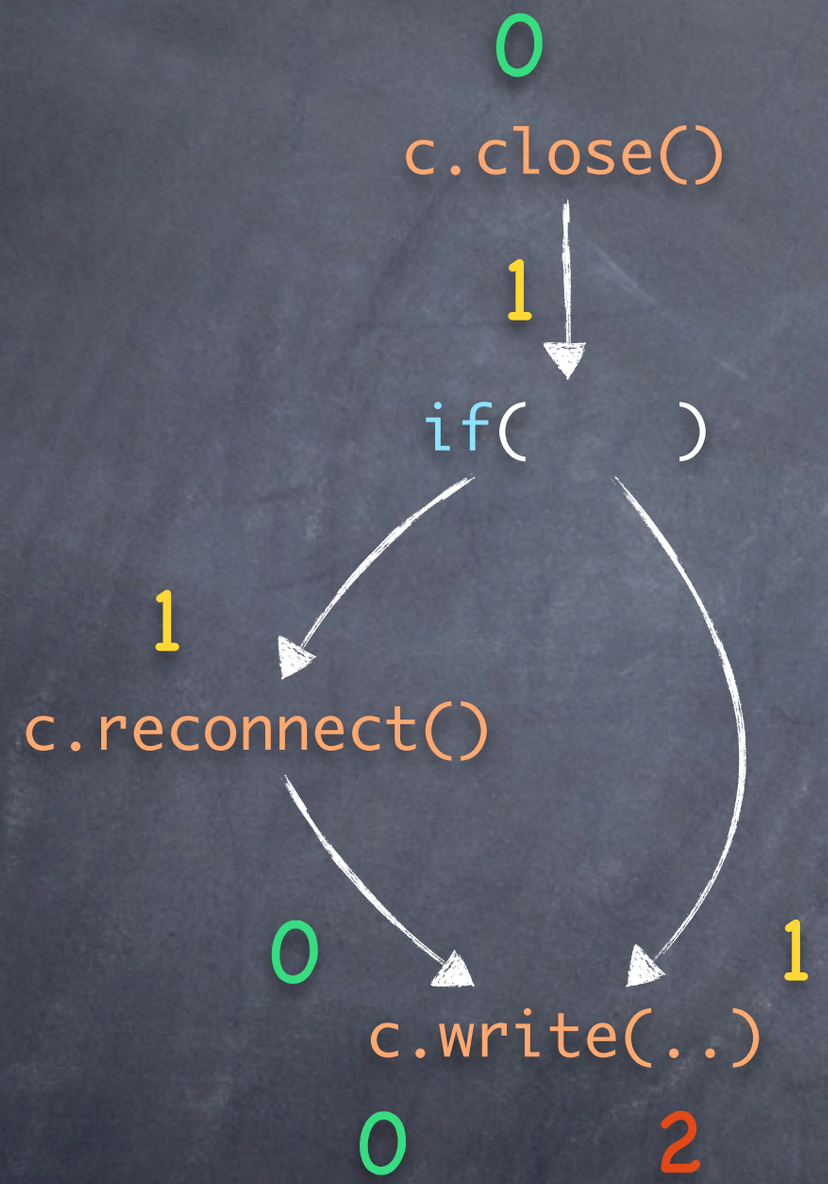
Algorithm is greedy

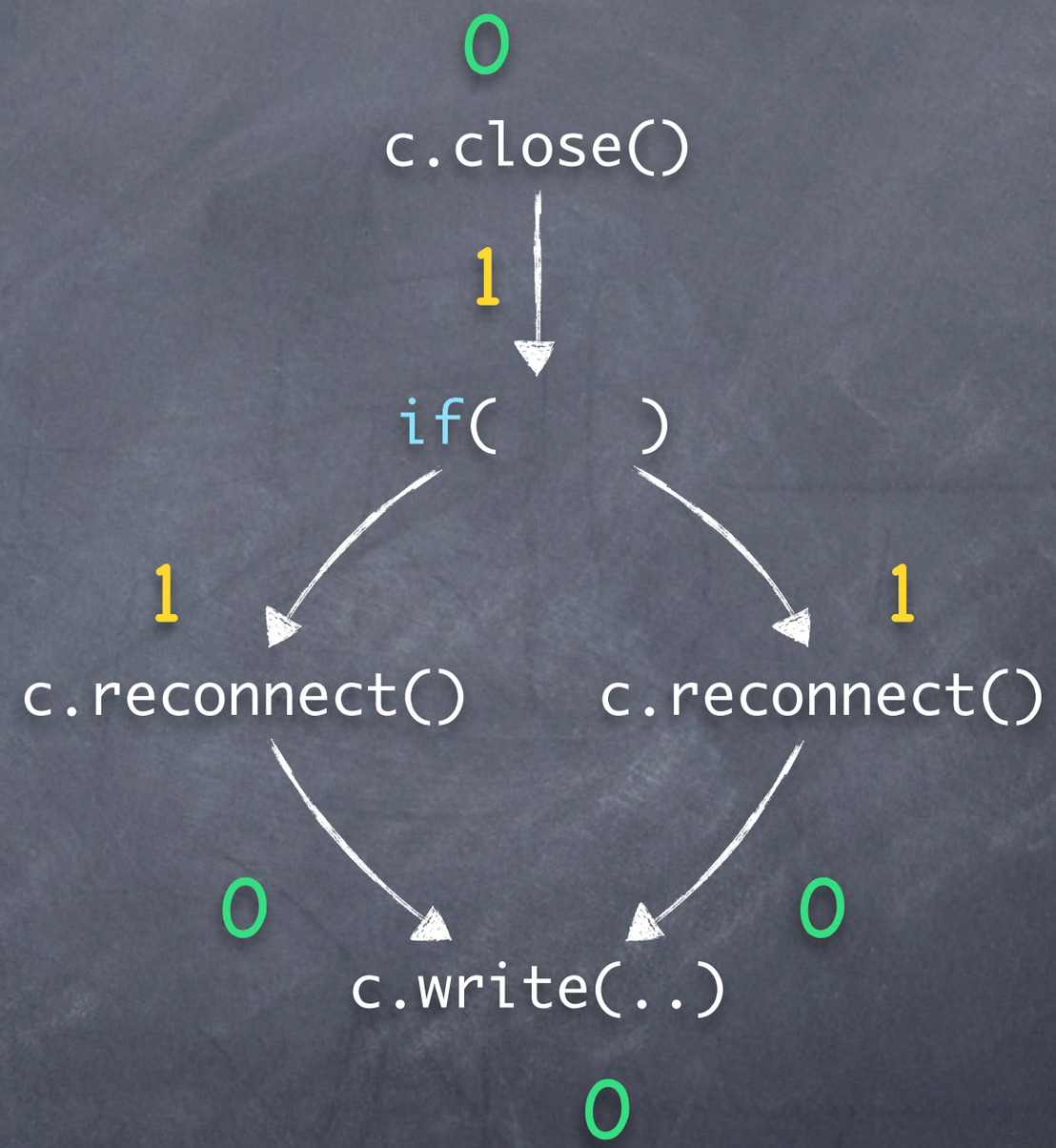


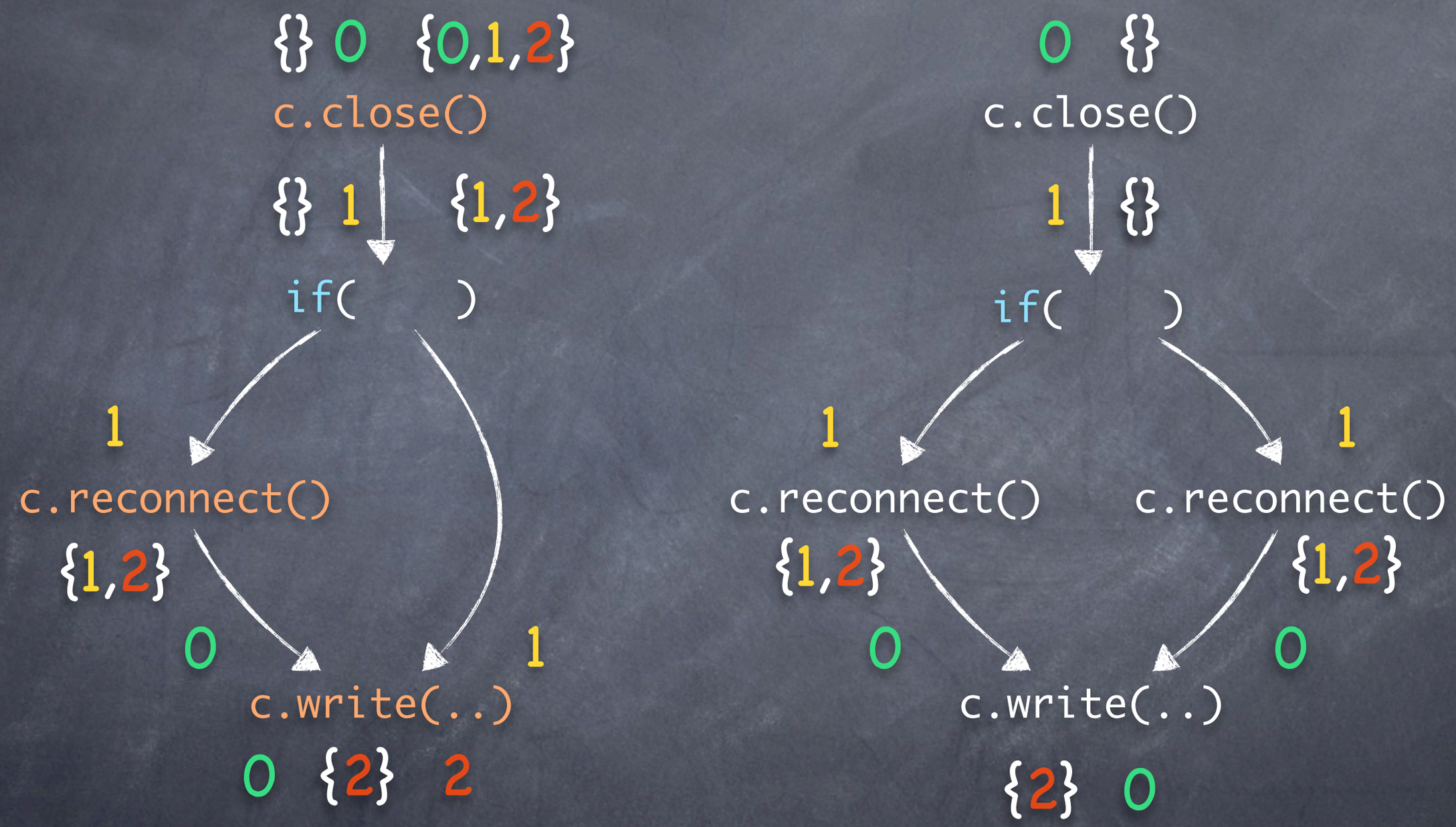


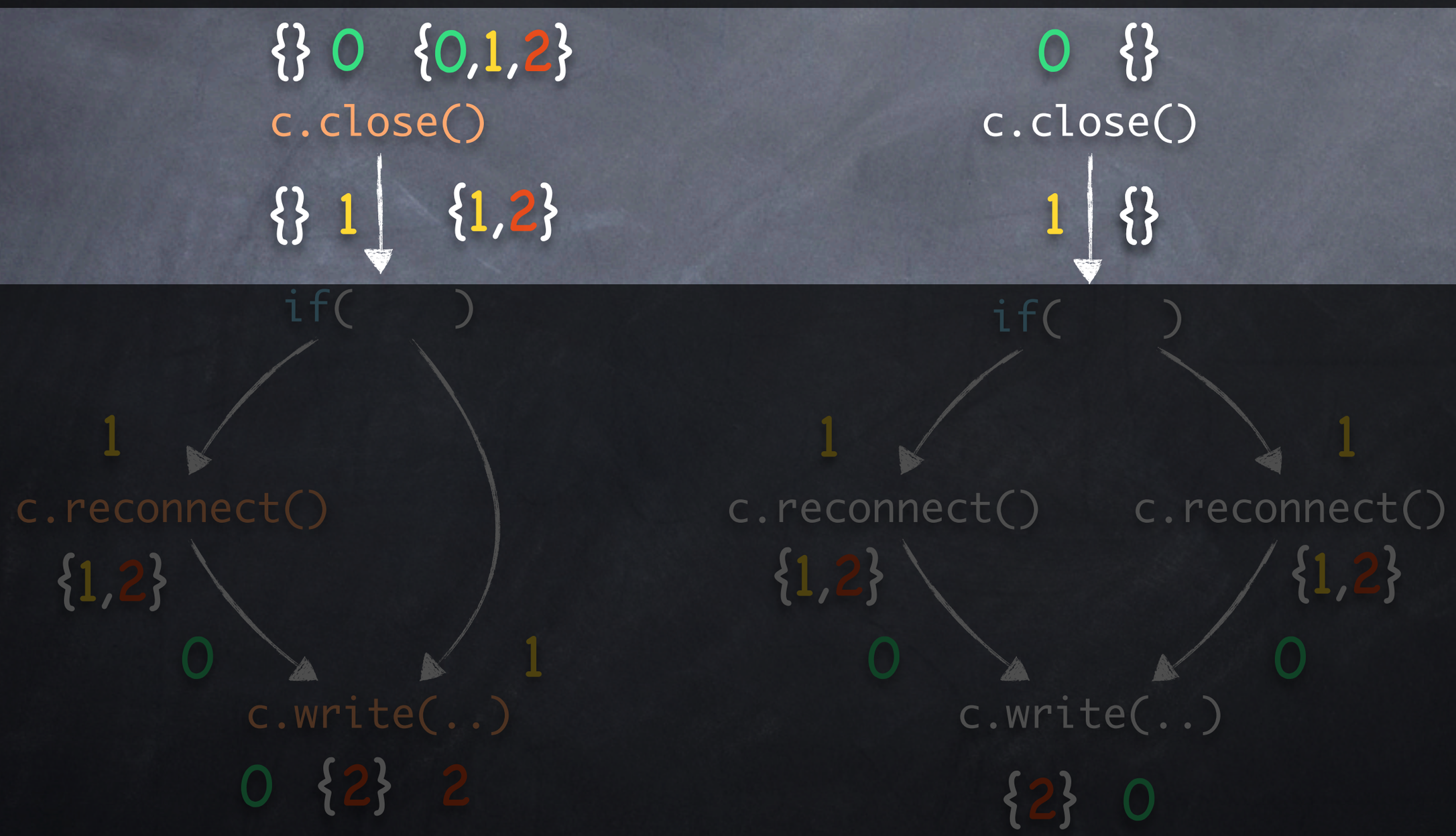
```
c1.close();

c1.reconnect();

c1.close();

c1.close();

c1.write(..);

c1.close();

c1.reconnect();

c1.write(..);
```

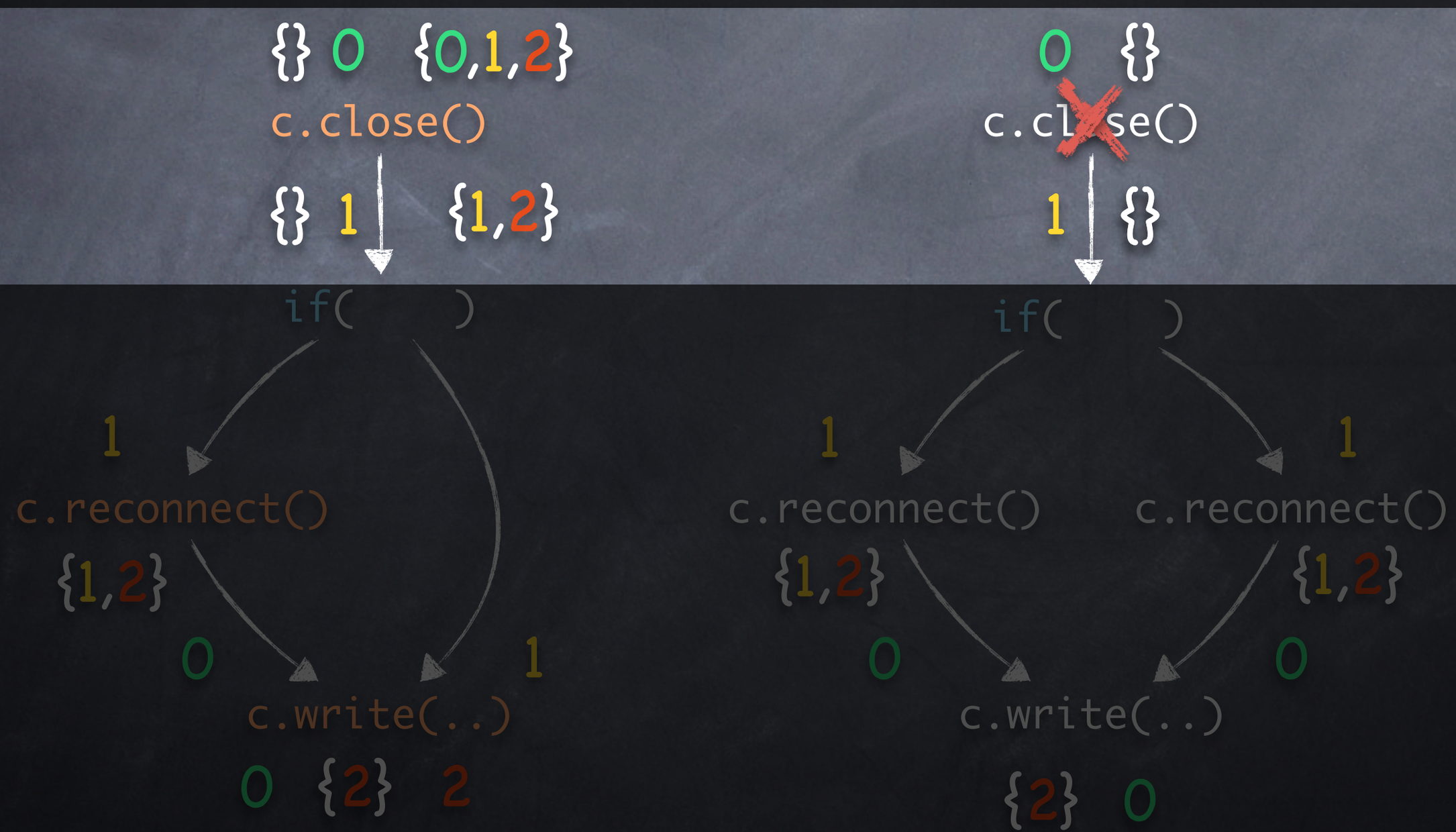
```
c1.close();

c1.write(..);
```

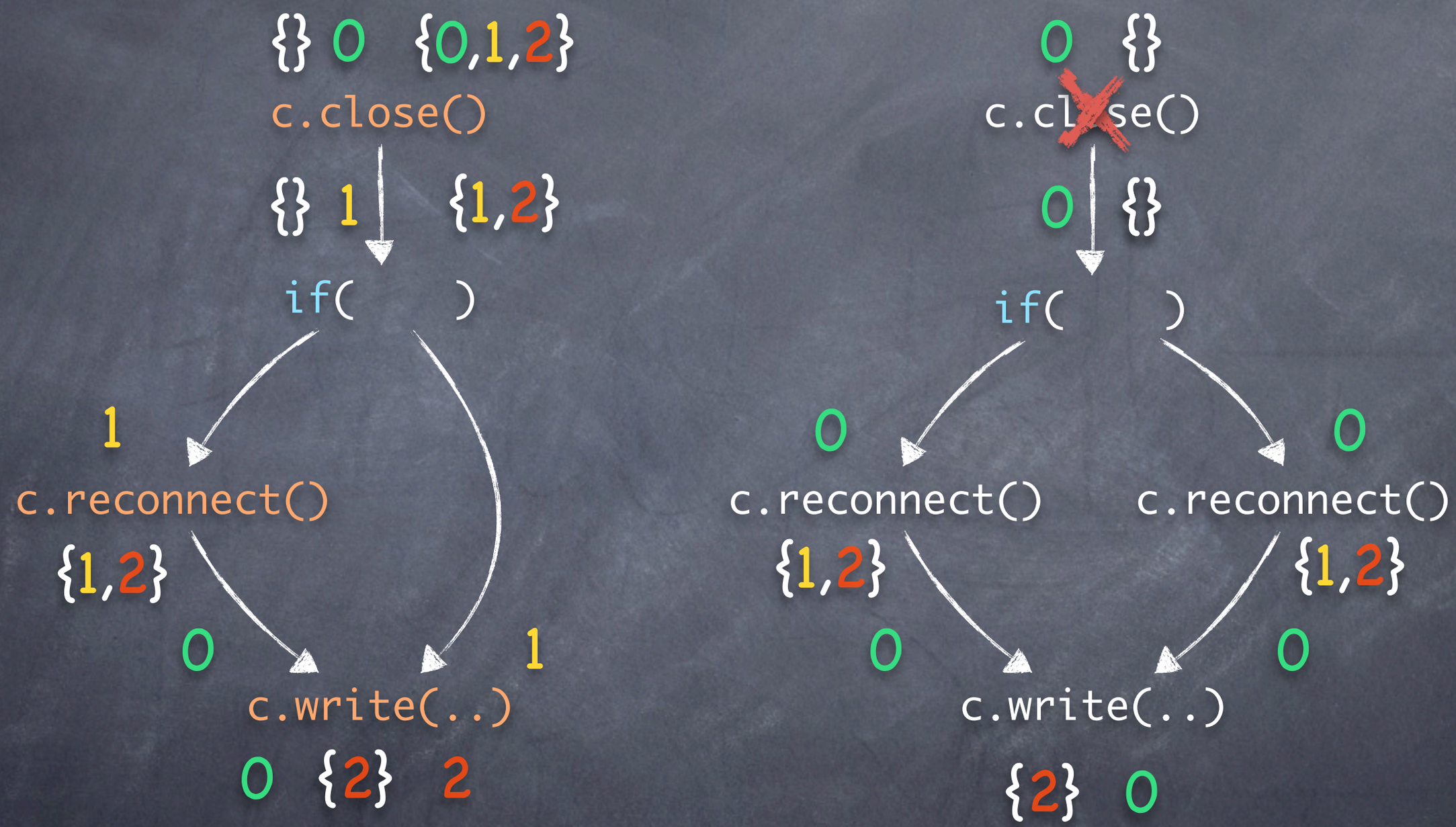

0
c.close()
1
if( )
1
c.reconnect()
0
1
c.write(..)
0
2

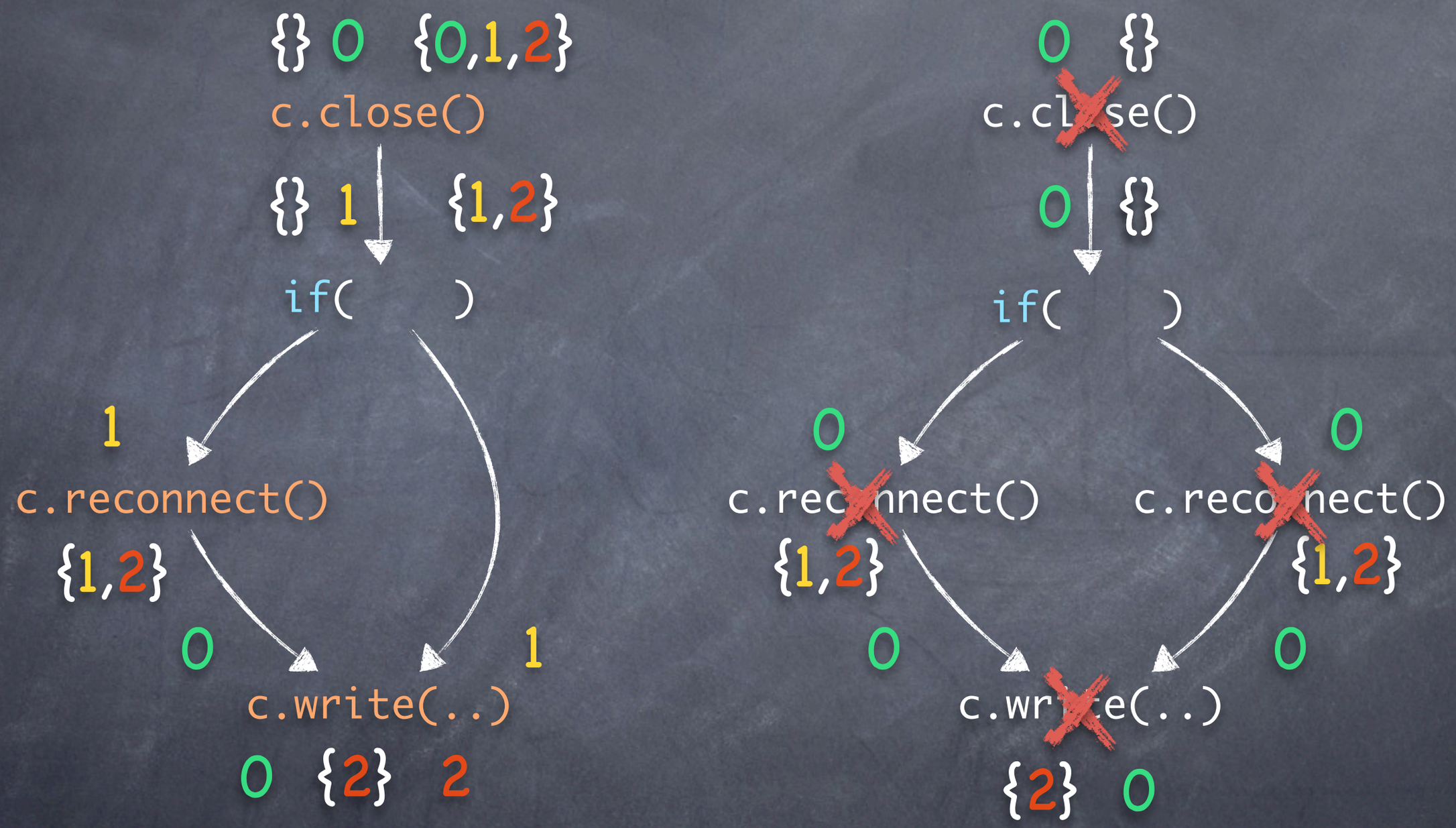

0
c.close()
1
if( )
1
1
c.reconnect()
c.reconnect()
0
0
c.write(..)
0


{} 0 {0,1,2}
c.close()
{} 1 {1,2}
if( )
1
c.reconnect()
{1,2}
0
1
c.write(..)
0 {2} 2
0 {}
c.close()
1 {}
if( )
1
1
c.reconnect()
c.reconnect()
{1,2}
{1,2}
0
0
c.write(..)
{2} 0


{} 0 {0,1,2}
c.close()
{} 1 {1,2}
if( )
1
c.reconnect()
{1,2}
0
1
c.write(..)
0 {2} 2
0 {}
c.close()
1 {}
if( )
1
c.reconnect()
{1,2}
0
1
c.reconnect()
{1,2}
0
c.write(..)
{2} 0


{} 0 {0,1,2}
c.close()
{} 1 {1,2}
if( )
1
c.reconnect()
{1,2}
0
1
c.write(..)
0 {2} 2
0 {}
c.close()
1 {}
if( )
1
c.reconnect()
{1,2}
0
1
c.reconnect()
{1,2}
0
c.write(..)
{2} 0


{} 0 {0,1,2}
c.close()
{} 1 {1,2}
if( )
1
c.reconnect()
{1,2}
0
1
c.write(..)
0 {2} 2
0 {}
c.close()
0 {}
if( )
0
0
c.reconnect()
c.reconnect()
{1,2}
{1,2}
0
0
c.write(..)
{2} 0


{} 0 {0,1,2}
c.close()
{} 1 {1,2}
if( )
1
c.reconnect()
{1,2}
0
1
c.write(..)
0 {2} 2
0 {}
c.close()
0 {}
if( )
0
0
c.reconnect()
c.reconnect()
{1,2}
{1,2}
0
0
c.write(..)
{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	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

Residual (hybrid) typestate analysis

Matthew B. Dwyer and Rahul Purandare.
Residual dynamic typestate 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

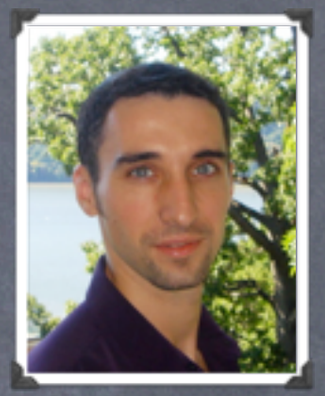
Nomair A. Naeem and Ondrej Lhotak.
Typestate-like analysis of multiple interacting objects. OOPSLA 2008.

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

Static typestate analysis







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

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

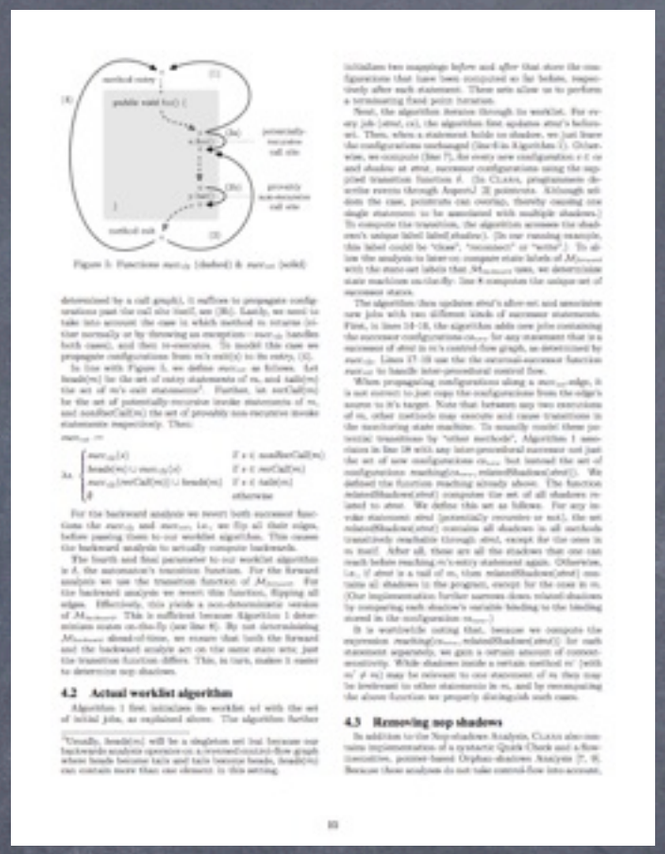
Type systems for typestate


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


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

In the paper

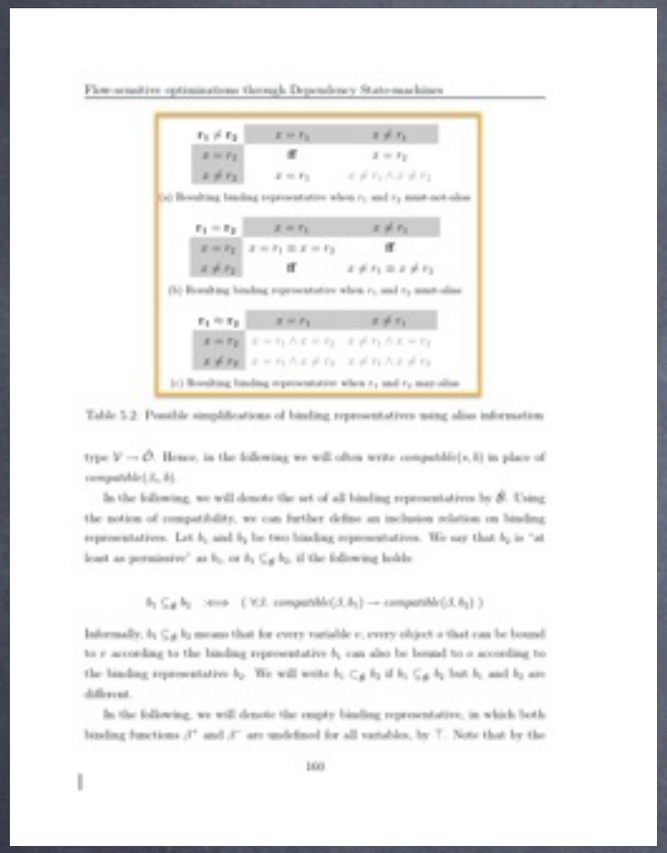
formal description of algorithms



loops, exceptions, method calls, recursion

reference to Ph.D. thesis

In the thesis



Flow-sensitive optimizations through Dependency State-machines

$r_1 \neq r_2$	$x = r_2$	$x \neq r_2$
$x = r_2$	ff	$x = r_2$
$x \neq r_2$	$x = r_2$	$x \neq r_2 \wedge x \neq r_2$

(a) Resulting binding representative when r_1 and r_2 must-not-alias

$r_1 = r_2$	$x = r_1$	$x \neq r_1$
$x = r_2$	$x = r_1 \equiv x = r_2$	ff
$x \neq r_2$	ff	$x \neq r_1 \equiv x \neq r_2$

(b) Resulting binding representative when r_1 and r_2 must-alias

$r_1 \approx r_2$	$x = r_1$	$x \neq r_1$
$x = r_2$	$x = r_1 \wedge x = r_2$	$x \neq r_1 \wedge x = r_2$
$x \neq r_2$	$x = r_1 \wedge x \neq r_2$	$x \neq r_1 \wedge x \neq r_2$

(c) Resulting binding representative when r_1 and r_2 may-alias

Table 5.2: Possible simplifications of binding representatives using alias information

type $V \to \hat{\mathcal{O}}$. Hence, in the following we will often write $\mathit{compatible}(v, b)$ in place of $\mathit{compatible}(\{\}, b)$.

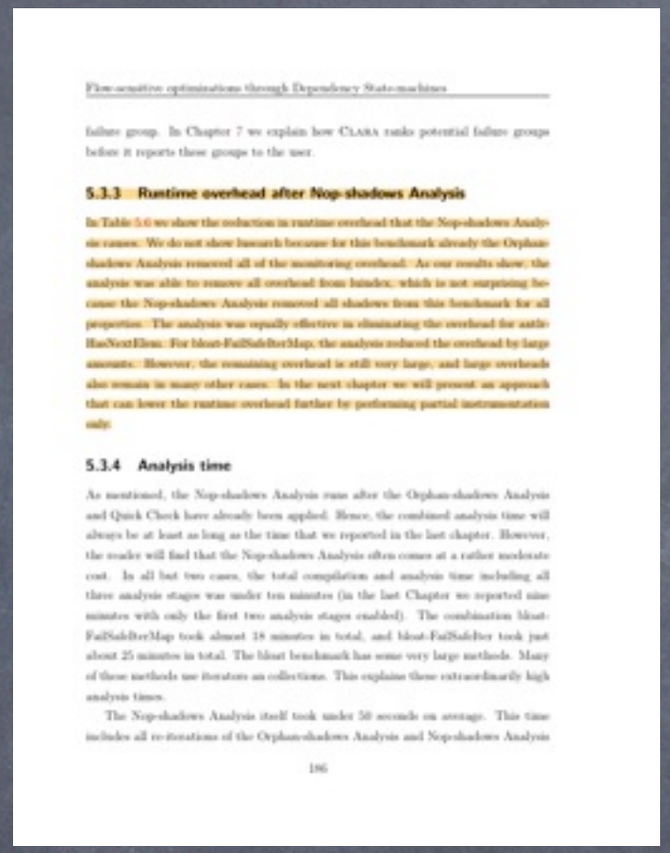
In the following, we will denote the set of all binding representatives by $\hat{\mathcal{B}}$. Using the notion of compatibility, we can further define an inclusion relation on binding representatives. Let b_1 and b_2 be two binding representatives. We say that b_2 is "at least as permissive" as b_1,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

$$b_1 \subseteq_{\mathcal{B}} b_2 \iff (\forall \mathcal{O}.\ \mathit{compatible}(\mathcal{O}, b_1) \to \mathit{compatible}(\mathcal{O}, b_2))$$

Informally, $b_1 \subseteq_{\mathcal{B}} b_2$ means that for every variable v, every object o that can be bound to v according to the binding representative b_1 can also be bound to o according to the binding representative b_2. We will write $b_1 \subset_{\mathcal{B}} b_2$ if $b_1 \subseteq_{\mathcal{B}} b_2$ 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

160



Flow-sensitive optimizations through Dependency State-machines

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

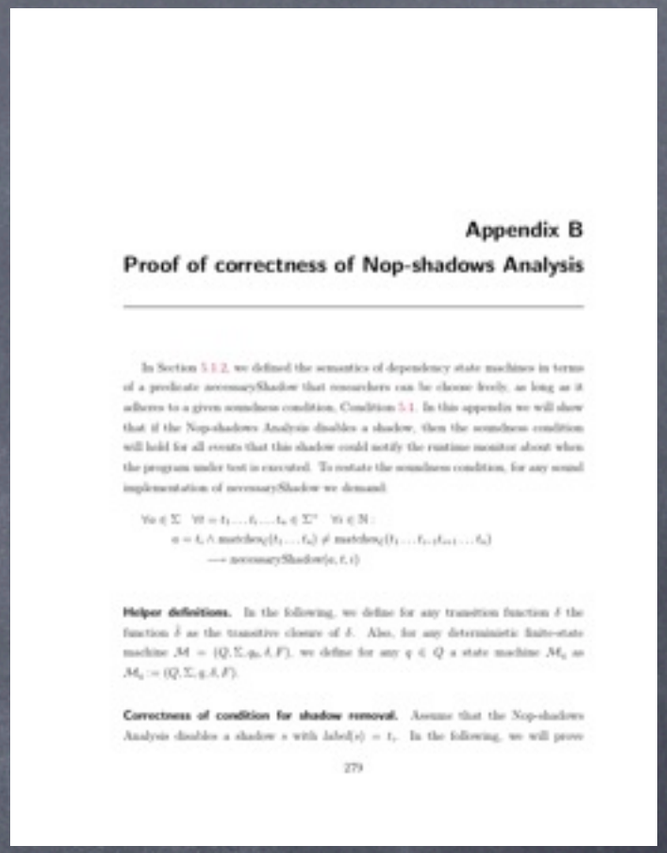
In Table 5.6 we show the reduction in runtime overhead that the Nop-shadows Analysis causes. We do not show bassarch because for this benchmark already the Orphan-shadows Analysis removed all of the monitoring overhead. As our results show, the analysis was able to remove all overhead from bindex, which 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 autoHasNextElem. For bloat-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 bloat-FailSafeIterMap took almost 18 minutes in total, and bloat-FailSafeIter took just about 25 minutes in total. The bloat 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-iterations of the Orphan-shadows Analysis and Nop-shadows Analysis

186



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 choose 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 restate the soundness condition, for any sound implementation of necessaryShadow we demand:

$$\forall o \in \Sigma\ \ \forall t = t_1, \ldots, t_i, \ldots, t_n \in \Sigma^*\ \ \forall i \in \mathbb{N}:$$
$$o = t_i \wedge \mathit{matchesq}(t_1 \ldots t_n) \neq \mathit{matchesq}(t_1 \ldots t_i, t_{i+1} \ldots t_n)$$
$$\longrightarrow \mathit{necessaryShadow}(o, t, i)$$

Helper definitions. In the following, we define for any transition function δ the function $\hat{\delta}$ as the transitive closure of δ. Also, for any deterministic finite-state machine $\mathcal{M} = (Q, \Sigma, q_0, \delta, F)$, we define for any $q \in Q$ a state machine $\mathcal{M}_q$ as $\mathcal{M}_q := (Q, \Sigma, q, \delta, F)$.

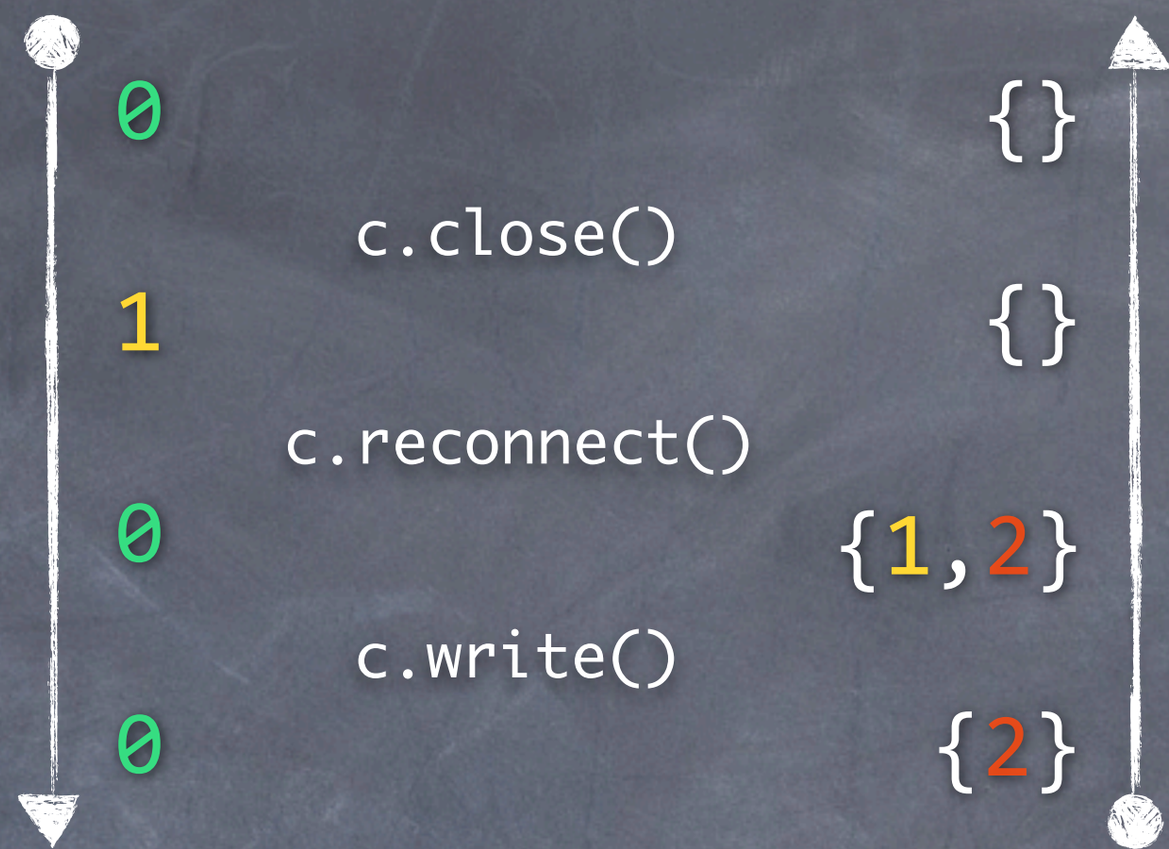
Correctness of condition for shadow removal. Assume that the Nop-shadows Analysis disables a shadow s with $\mathit{label}(s) = t_i$. In the following, we will prove

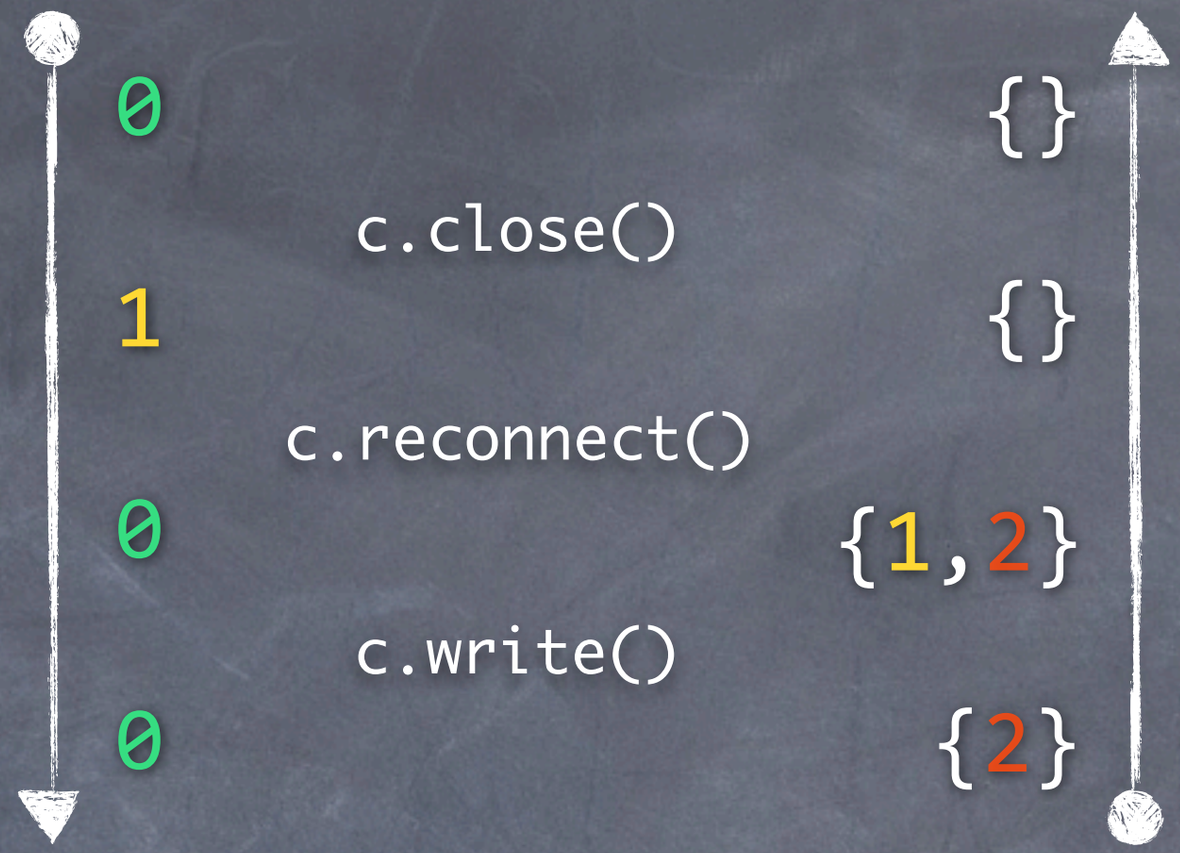
279

constraint system to deal with aliasing

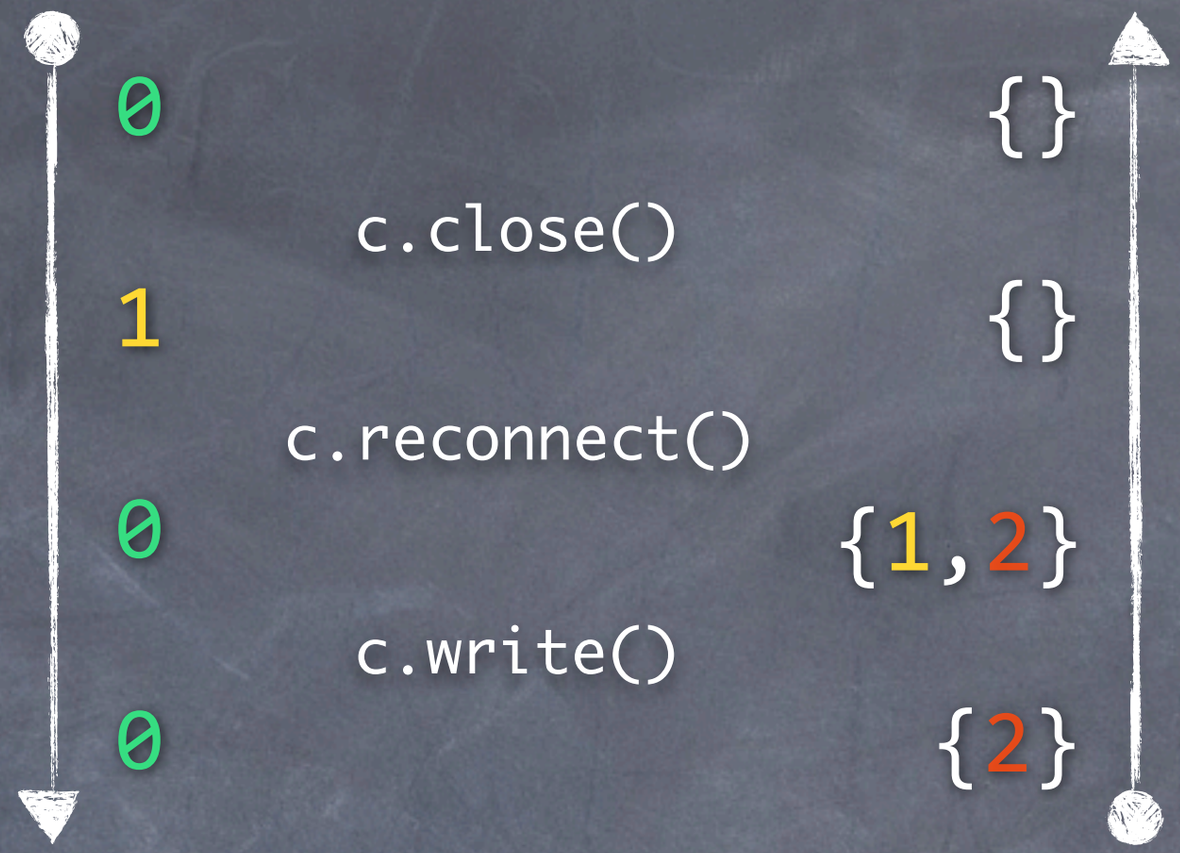
results on reduction of runtime overhead

correctness proof

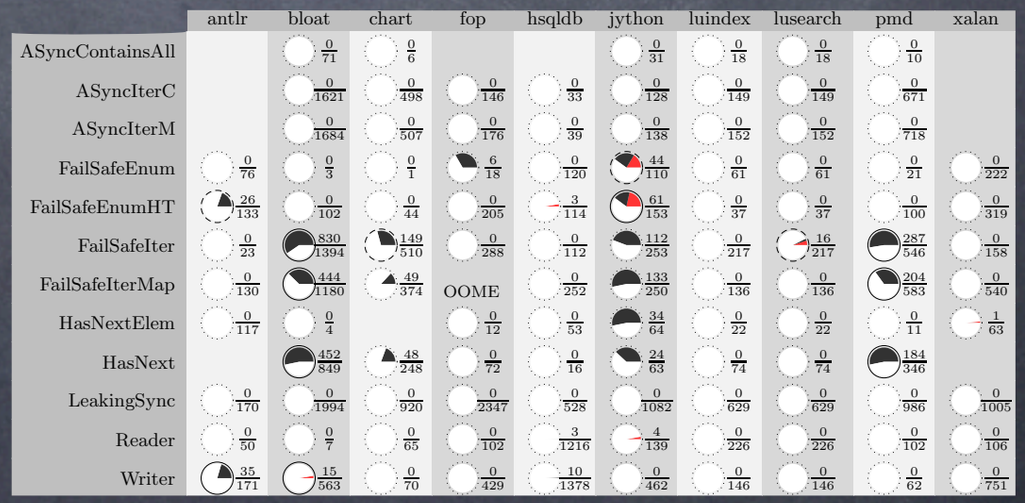

0
{}
c.close()
1
{}
c.reconnect()
0
{1,2}
c.write()
0
{2}



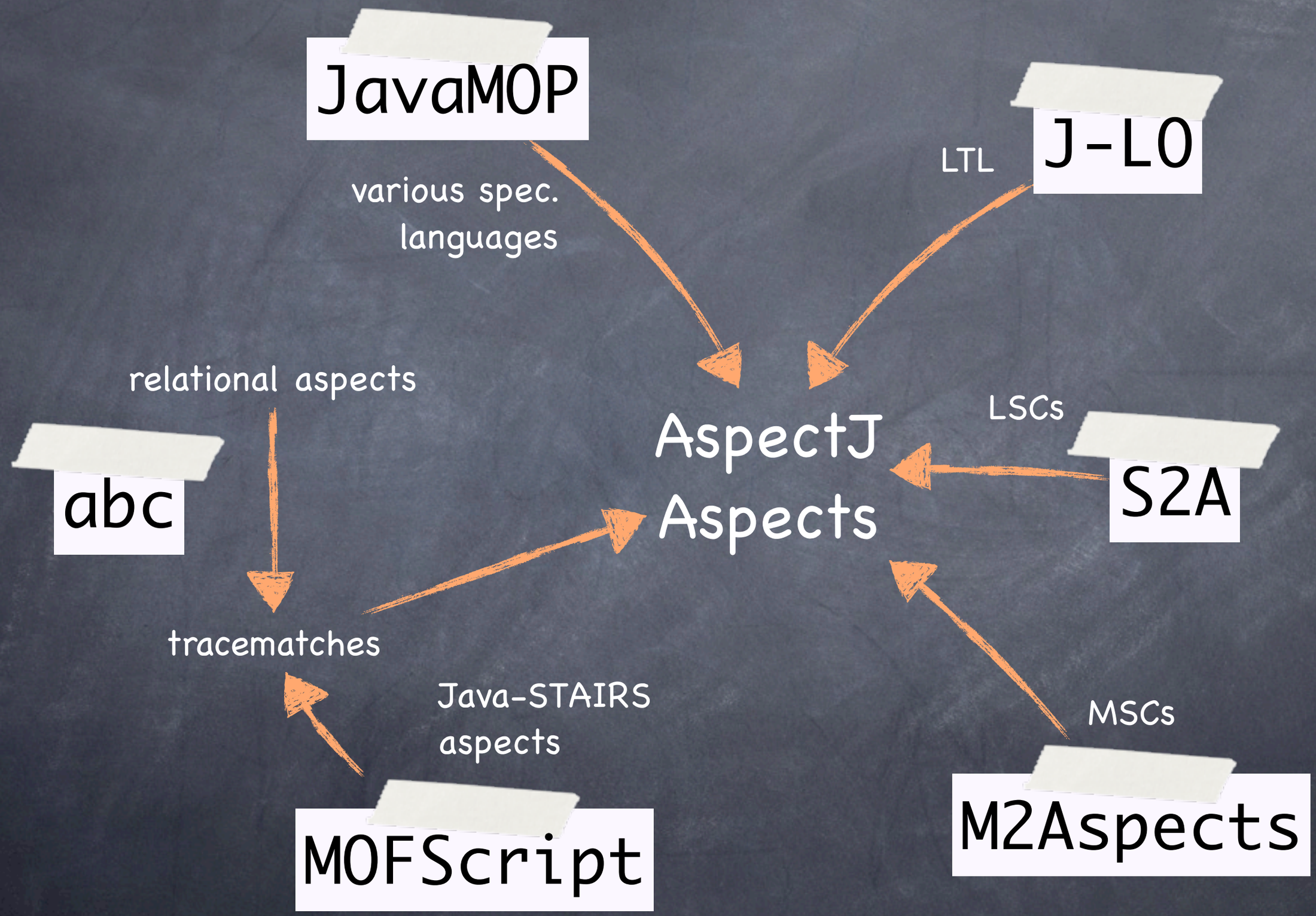
	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll		0/71	0/6			0/31	0/18	0/18	0/10	
ASyncIterC		0/1621	0/498	0/146	0/33	0/128	0/149	0/149	0/671	
ASyncIterM		0/1684	0/507	0/176	0/39	0/138	0/152	0/152	0/718	
FailSafeEnum	0/76	0/3	0/1	6/18	0/120	44/110	0/61	0/61	0/21	0/222
FailSafeEnumHT	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!");
}
```

abstract

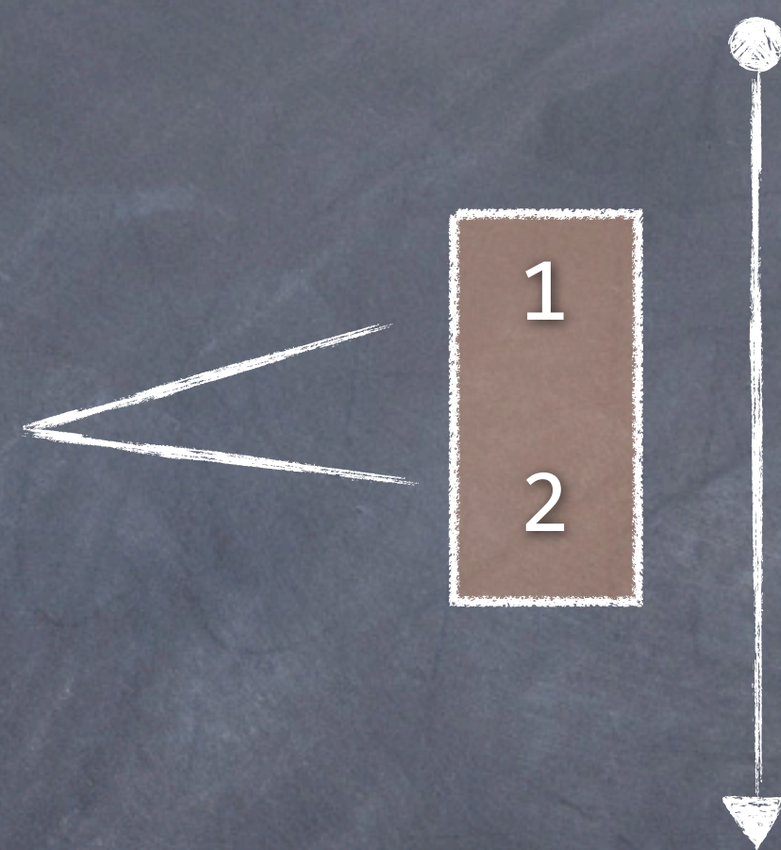
concrete

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

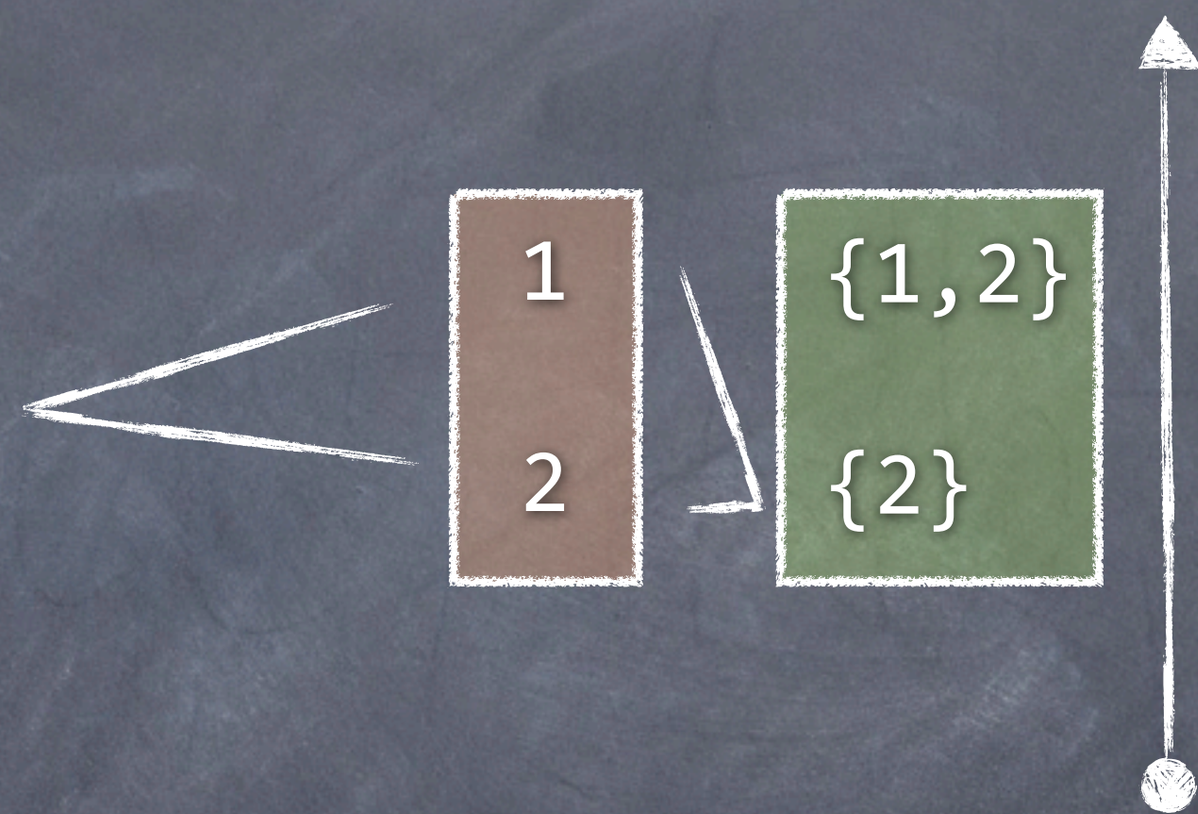
finite-state property

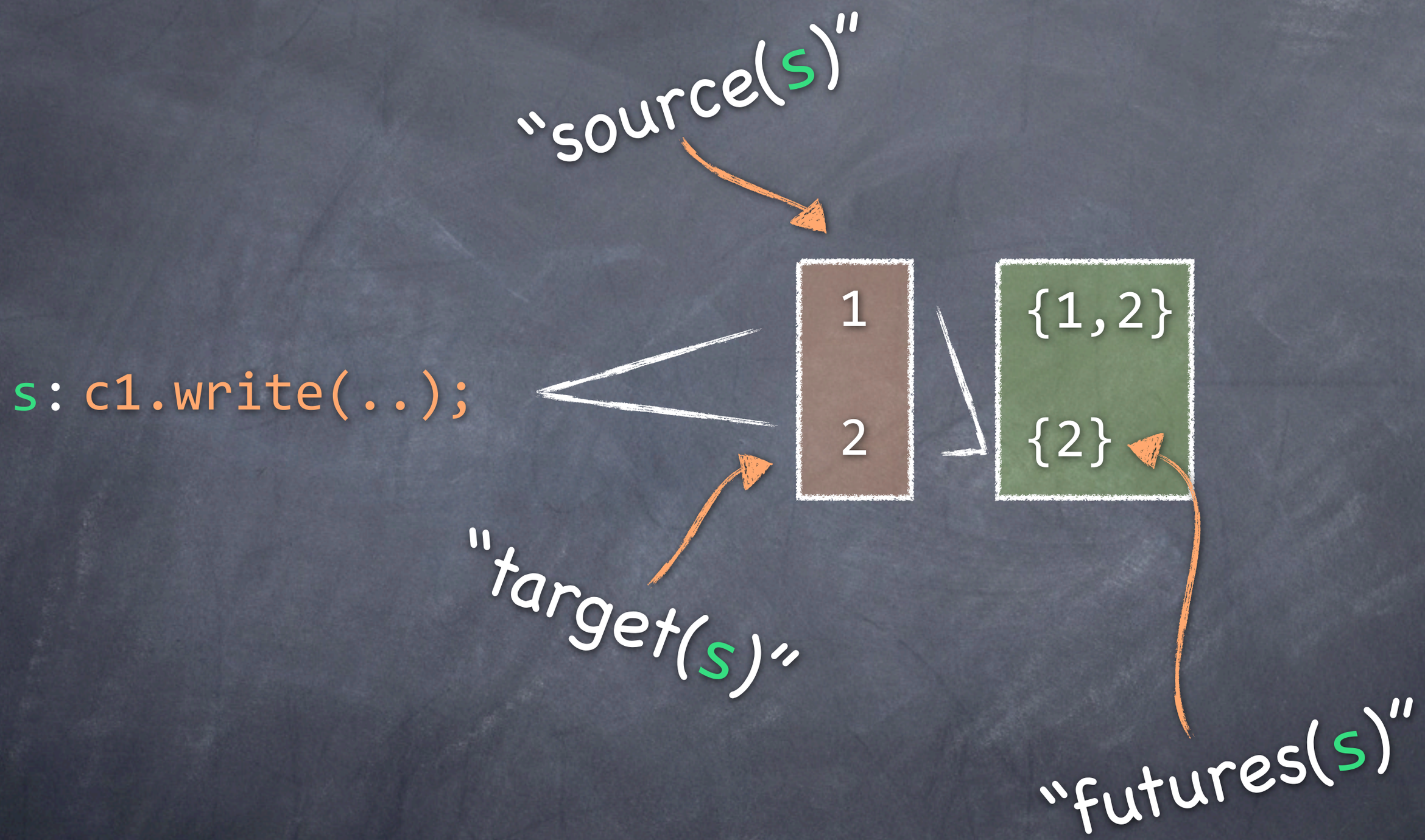
```
c1.write(..);
```

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



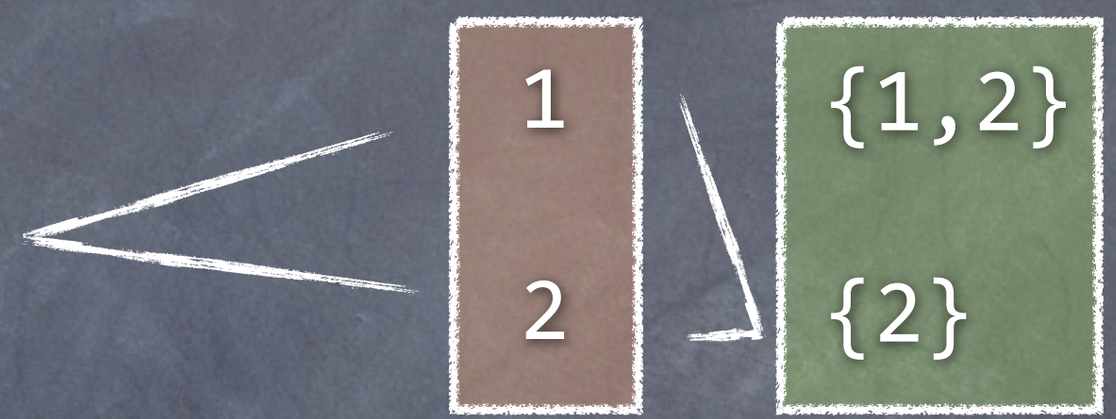
```
c1.write(..);
```




"source(s)"
s: c1.write(..);
1
2
{1,2}
{2}
"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"

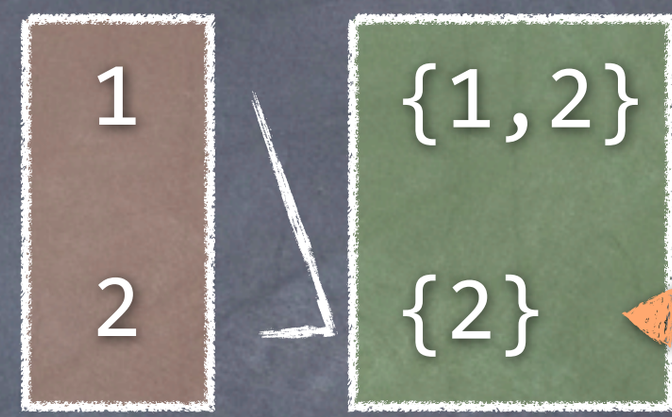
```
s: c1.write(..);
```

1	{1,2}
2	{2}

"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

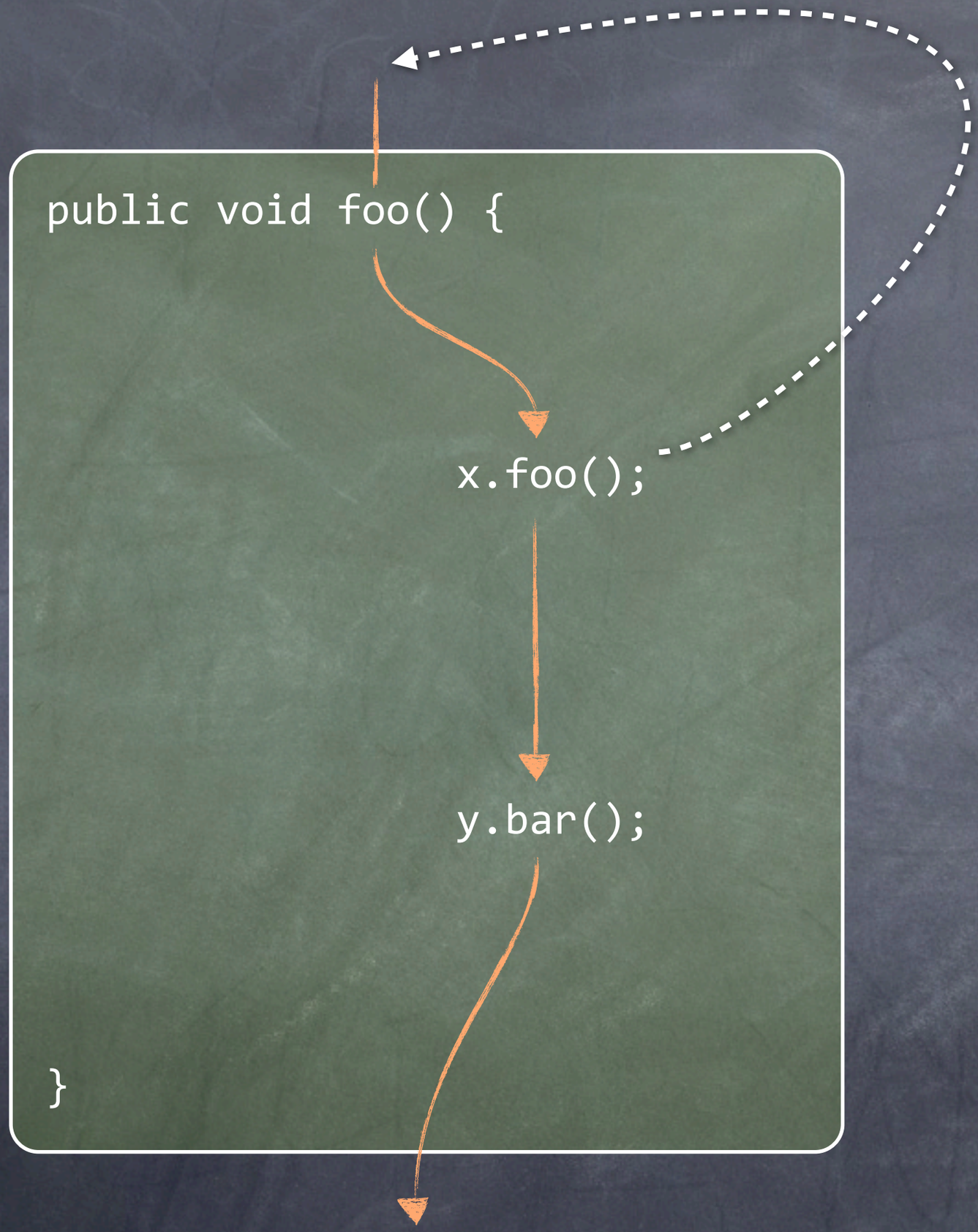
For efficiency: Focus on what's important

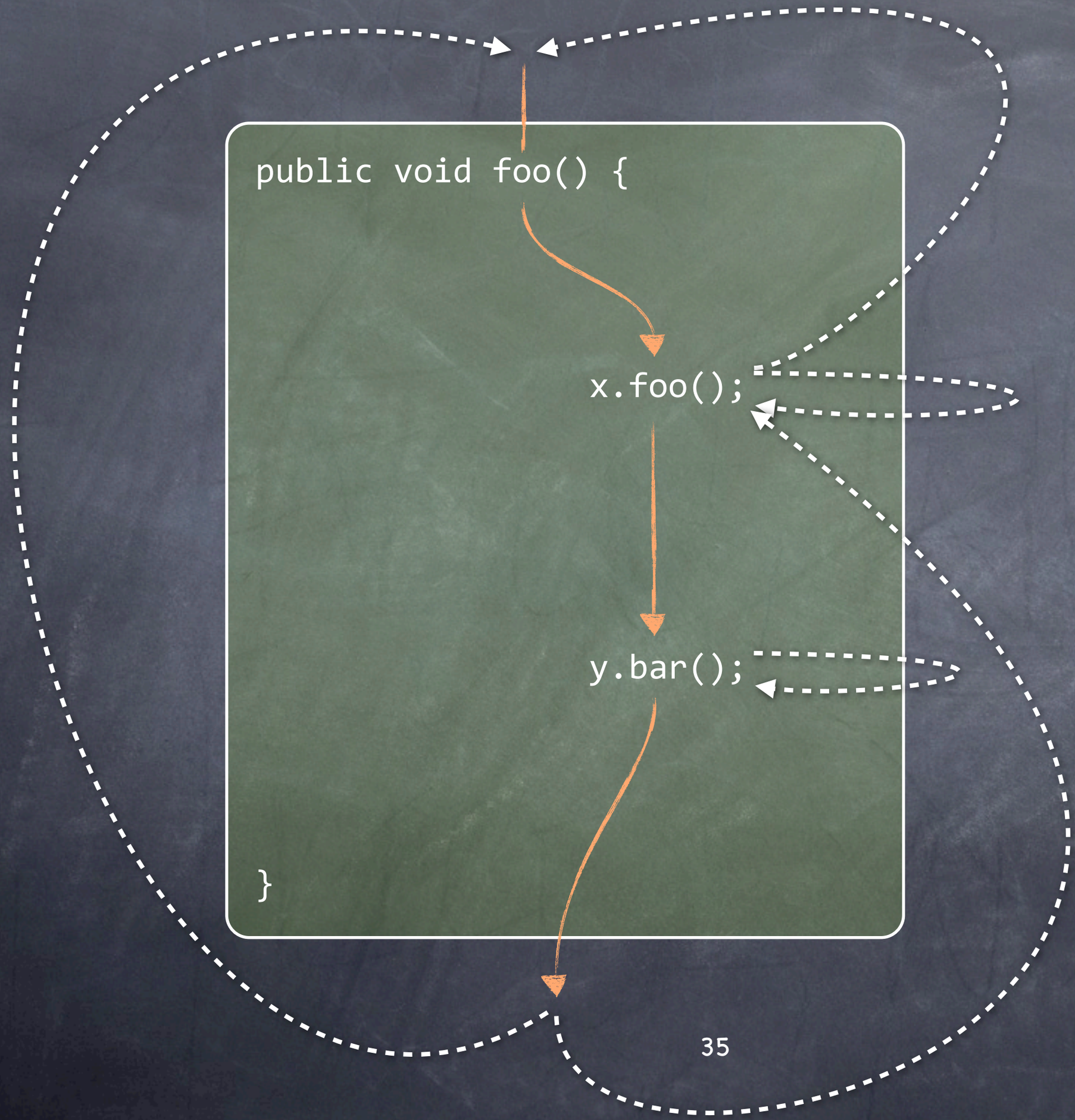
For efficiency: Focus on what's important

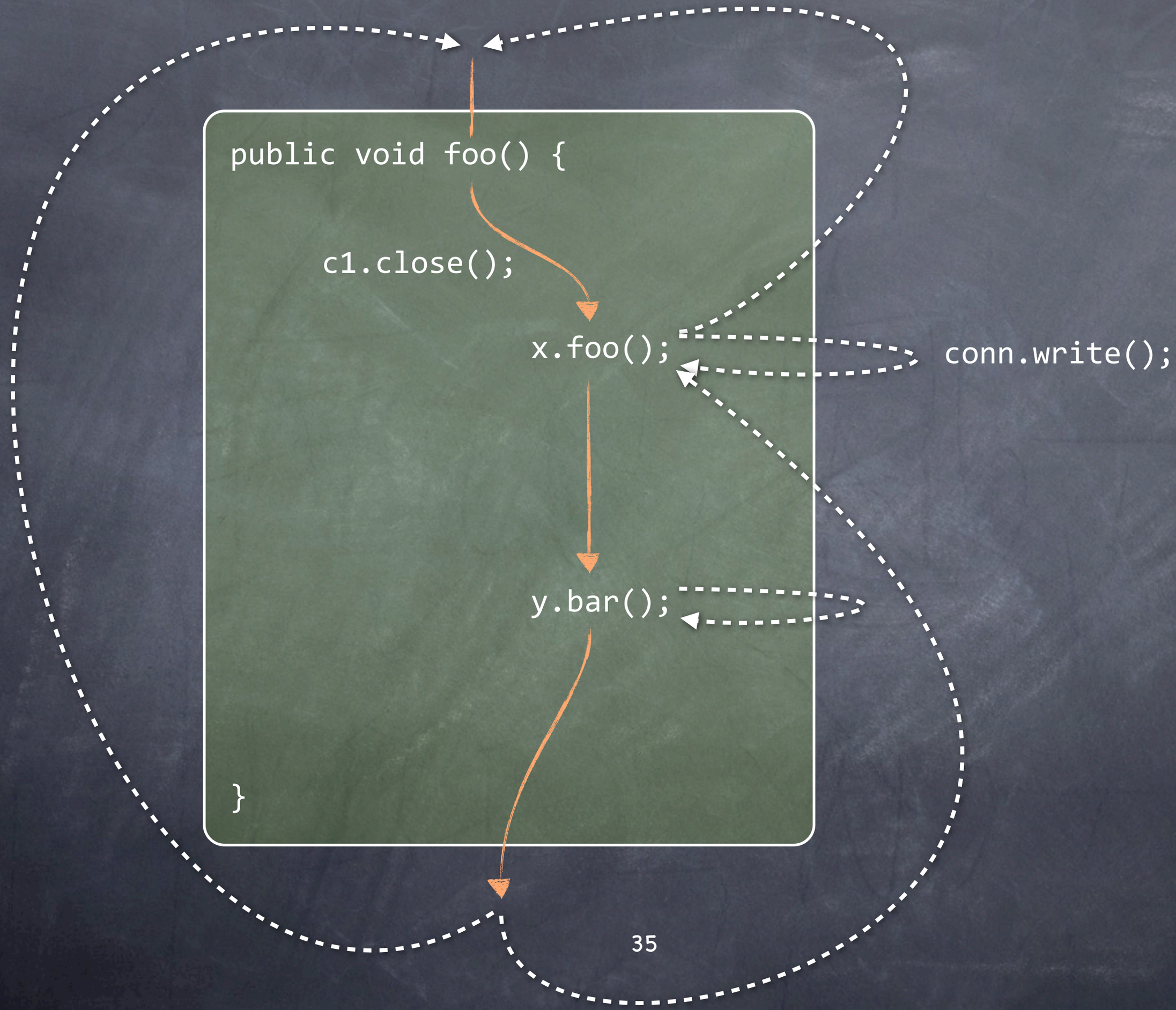
flow-sensitive
must-alias
may-alias

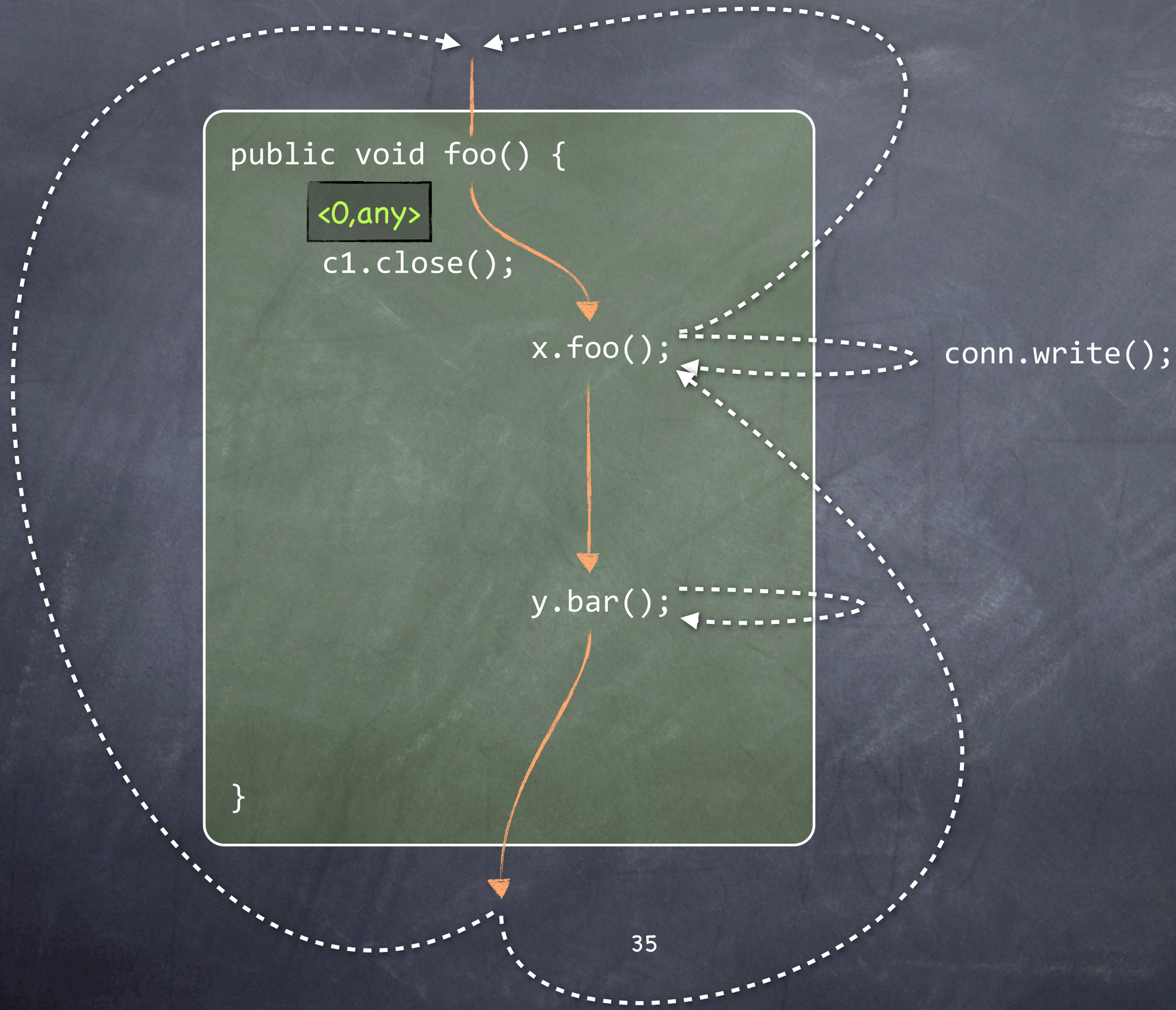
flow-insensitive
may-alias

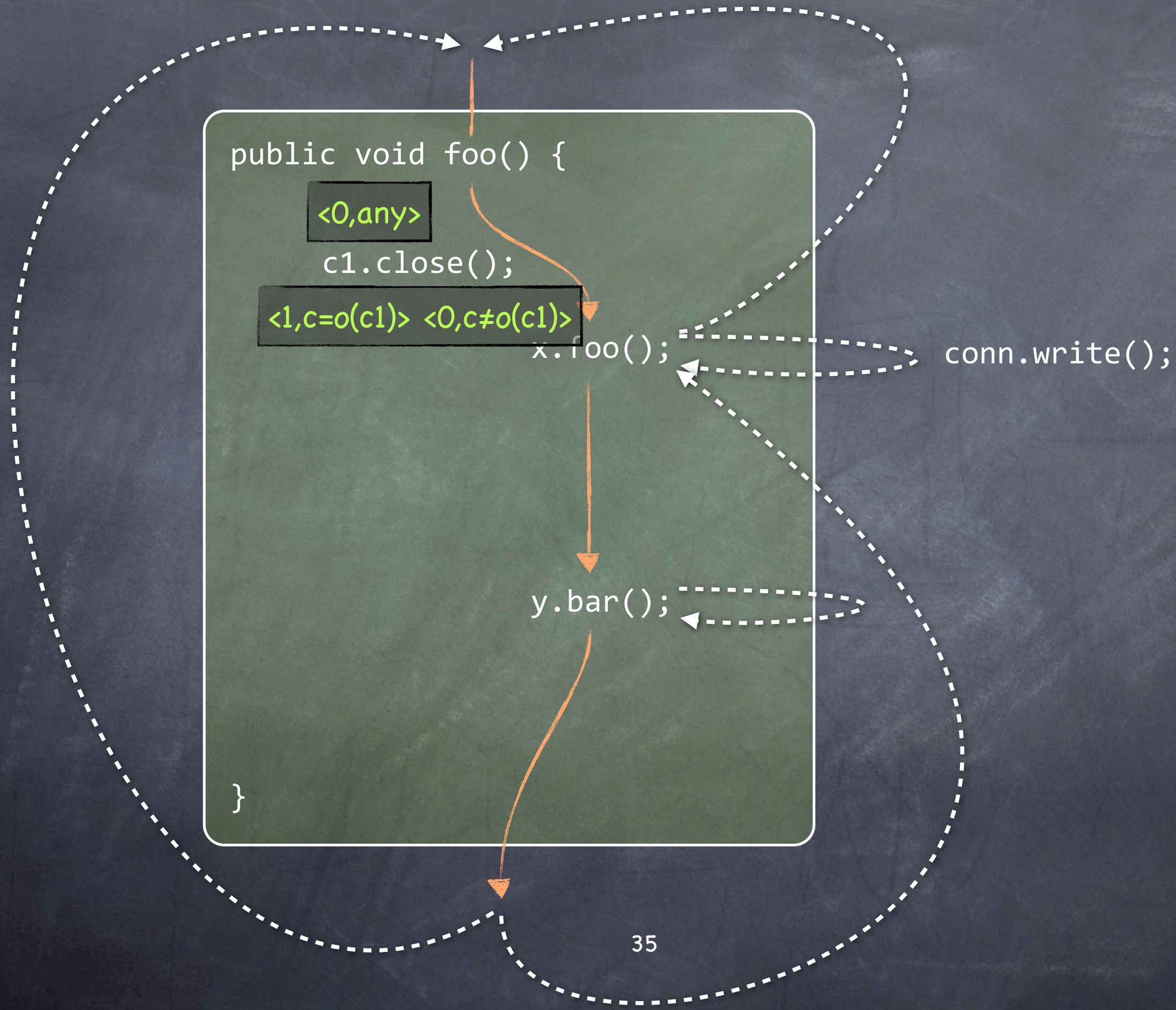
```
public void foo() {

    x.foo();

    y.bar();

}
```

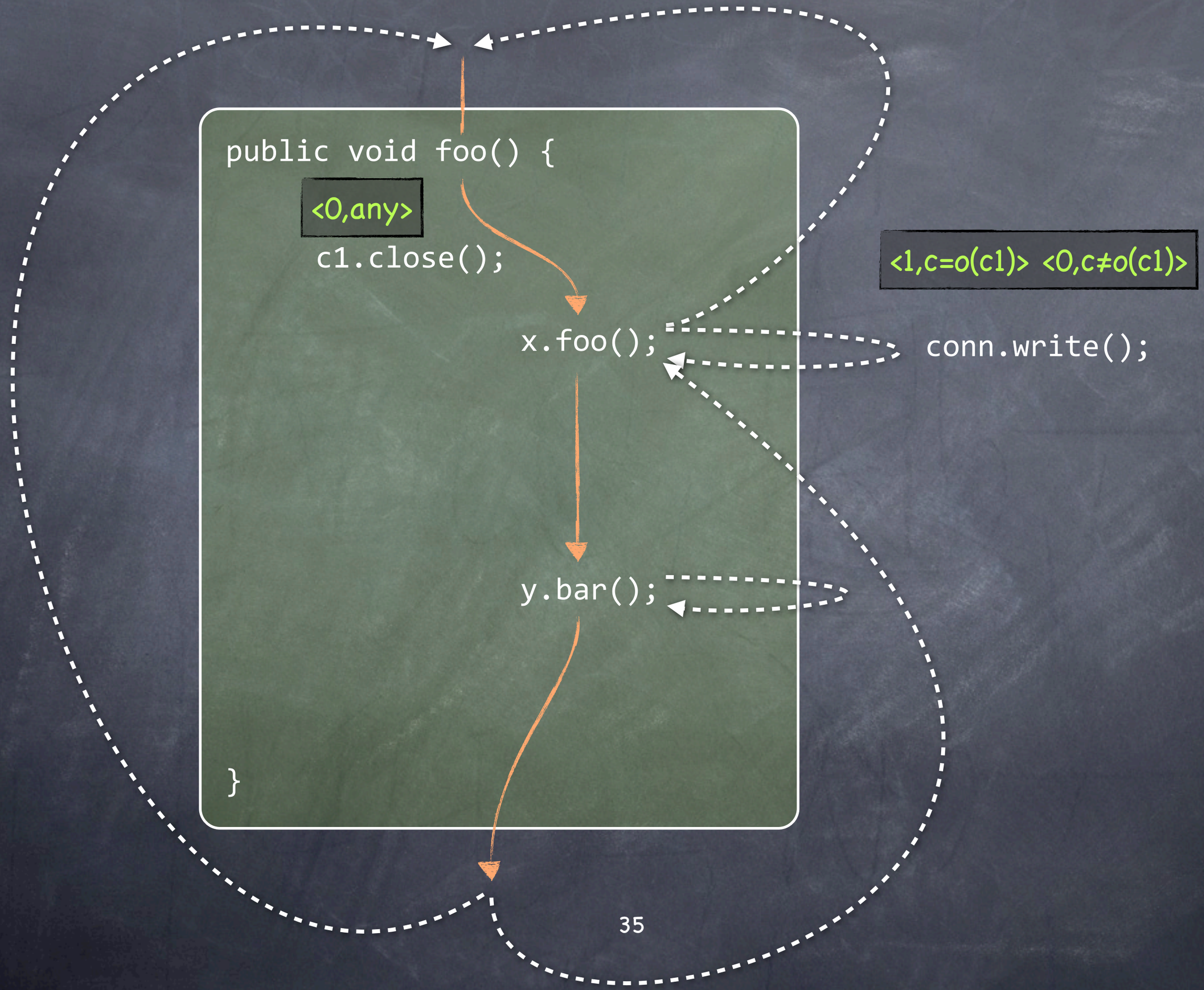

public void foo() {
x.foo();
y.bar();
}

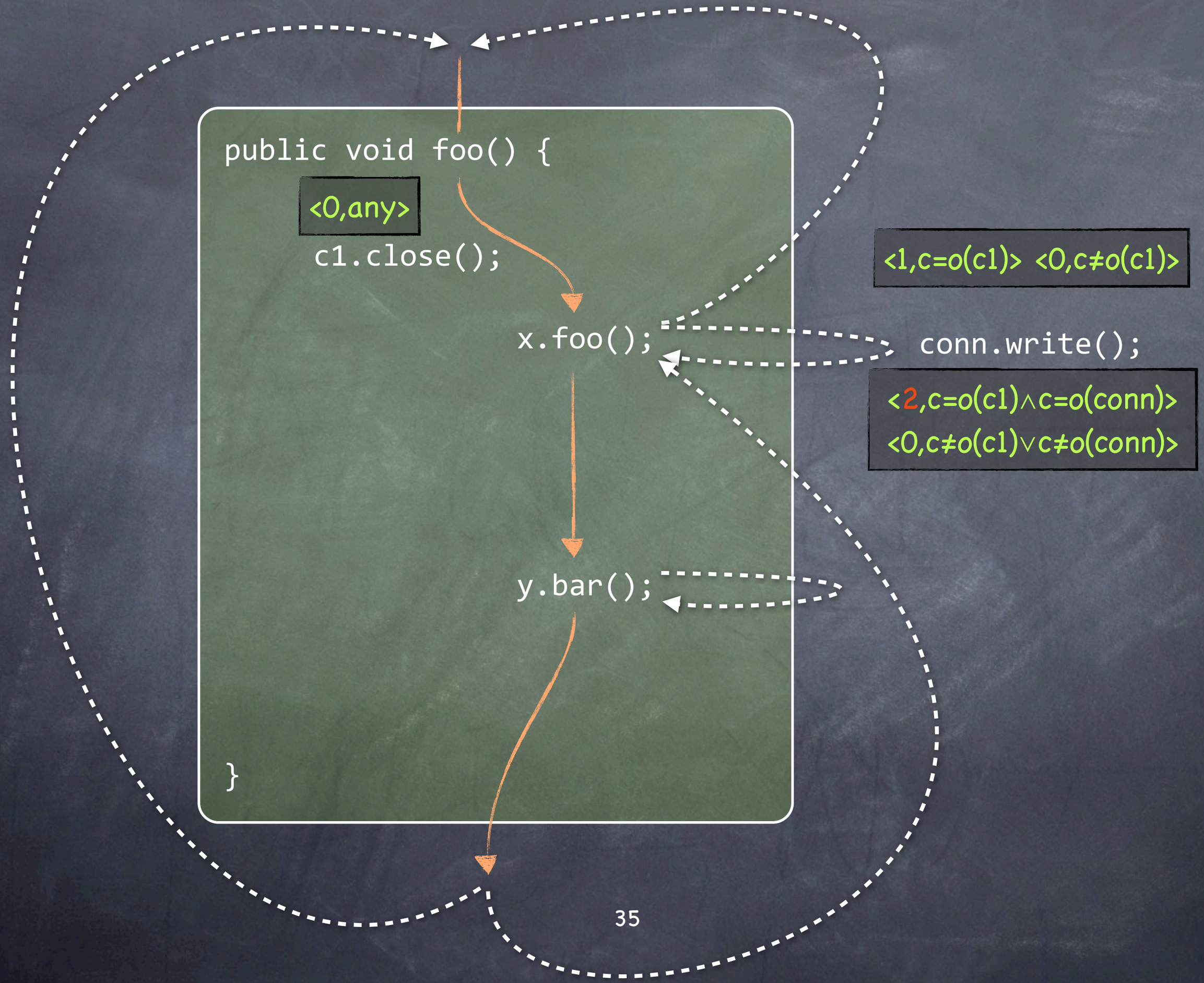

public void foo() {
x.foo();
y.bar();
}

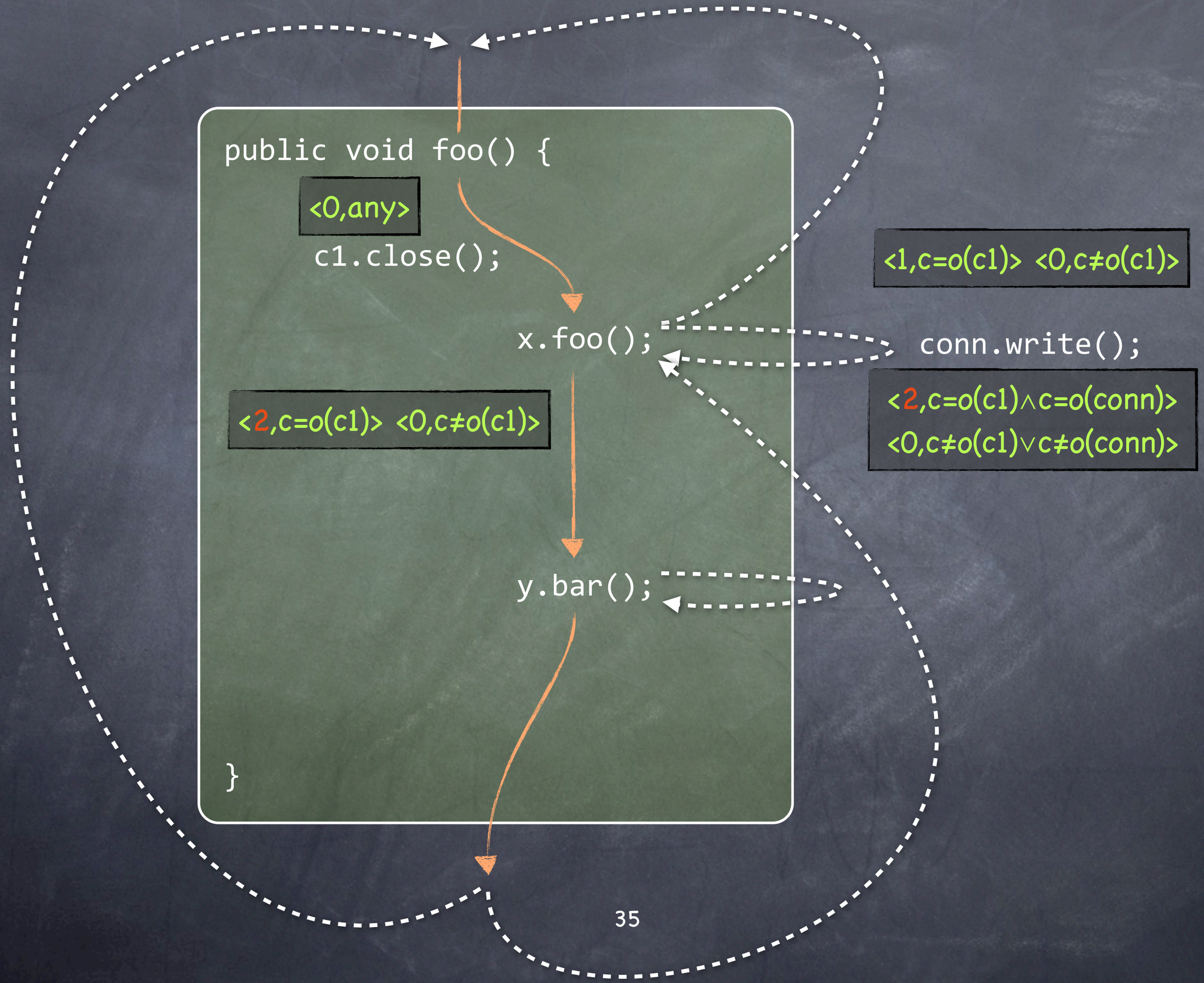

public void foo() {
c1.close();
x.foo();
conn.write();
y.bar();
}

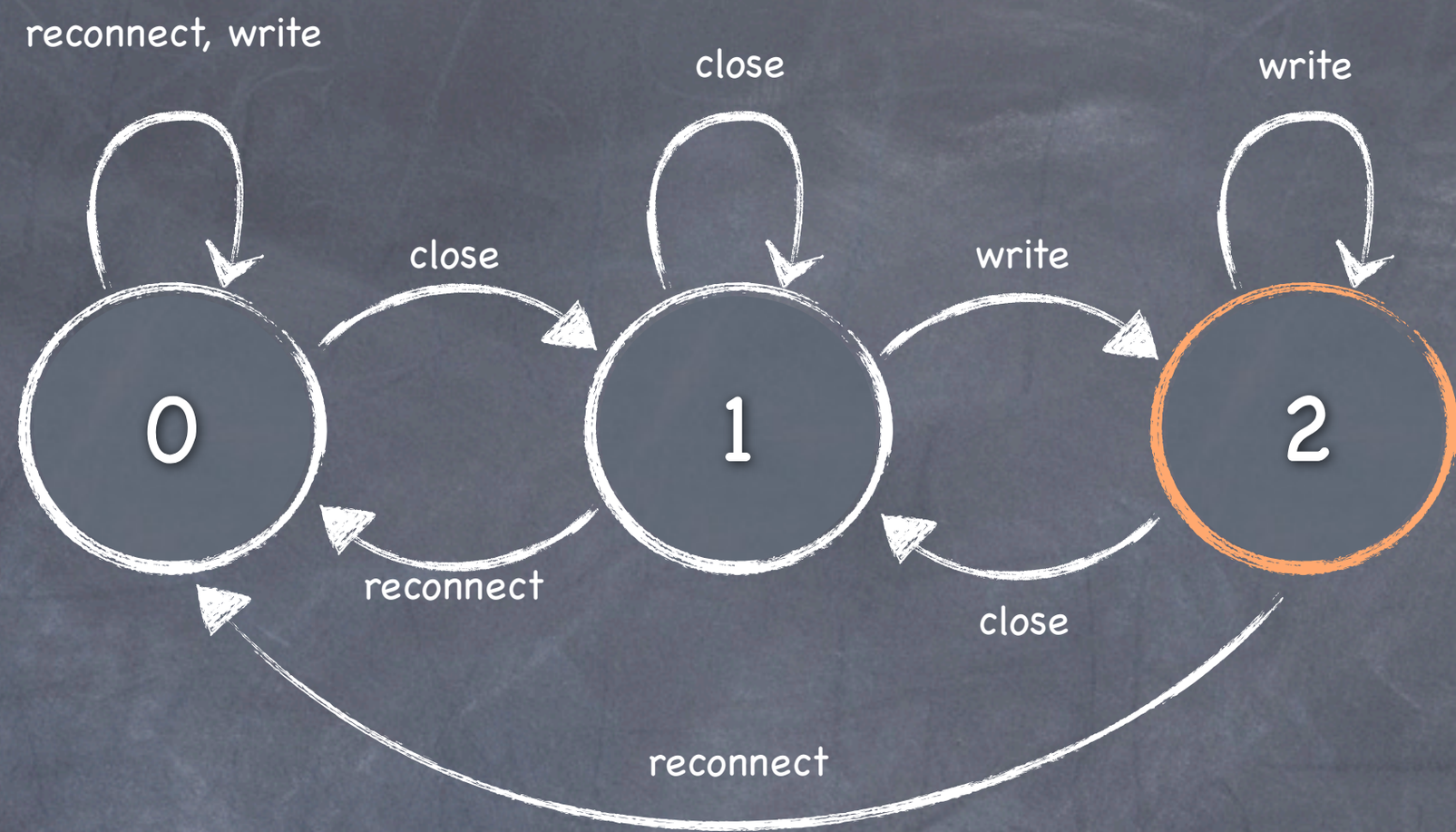

public void foo() {
<0,any>
c1.close();
x.foo();
conn.write();
y.bar();
}

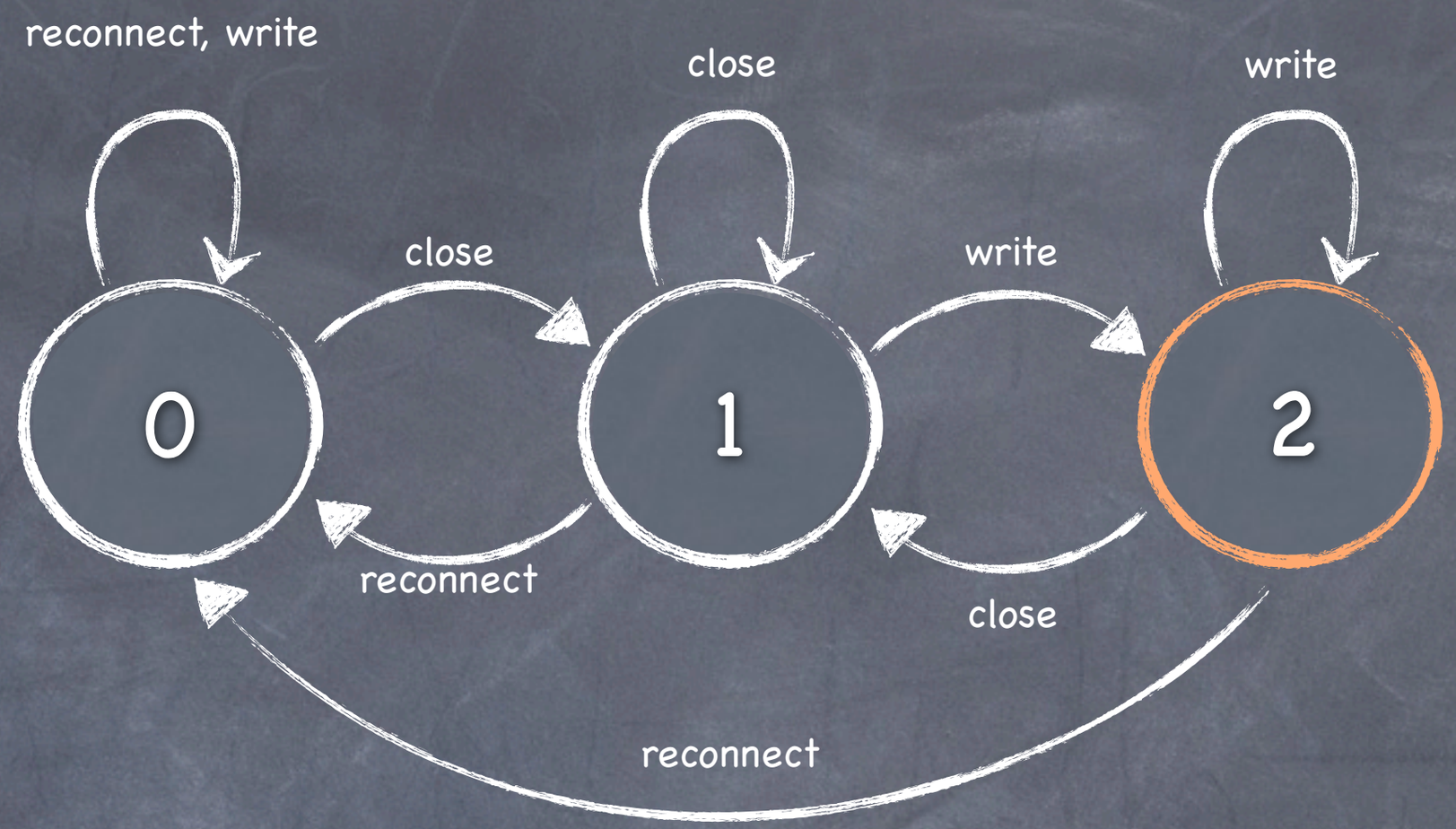

public void foo() {
<0,any>
c1.close();
<1,c=o(c1)> <0,c≠o(c1)>
x.foo();
conn.write();
y.bar();
}

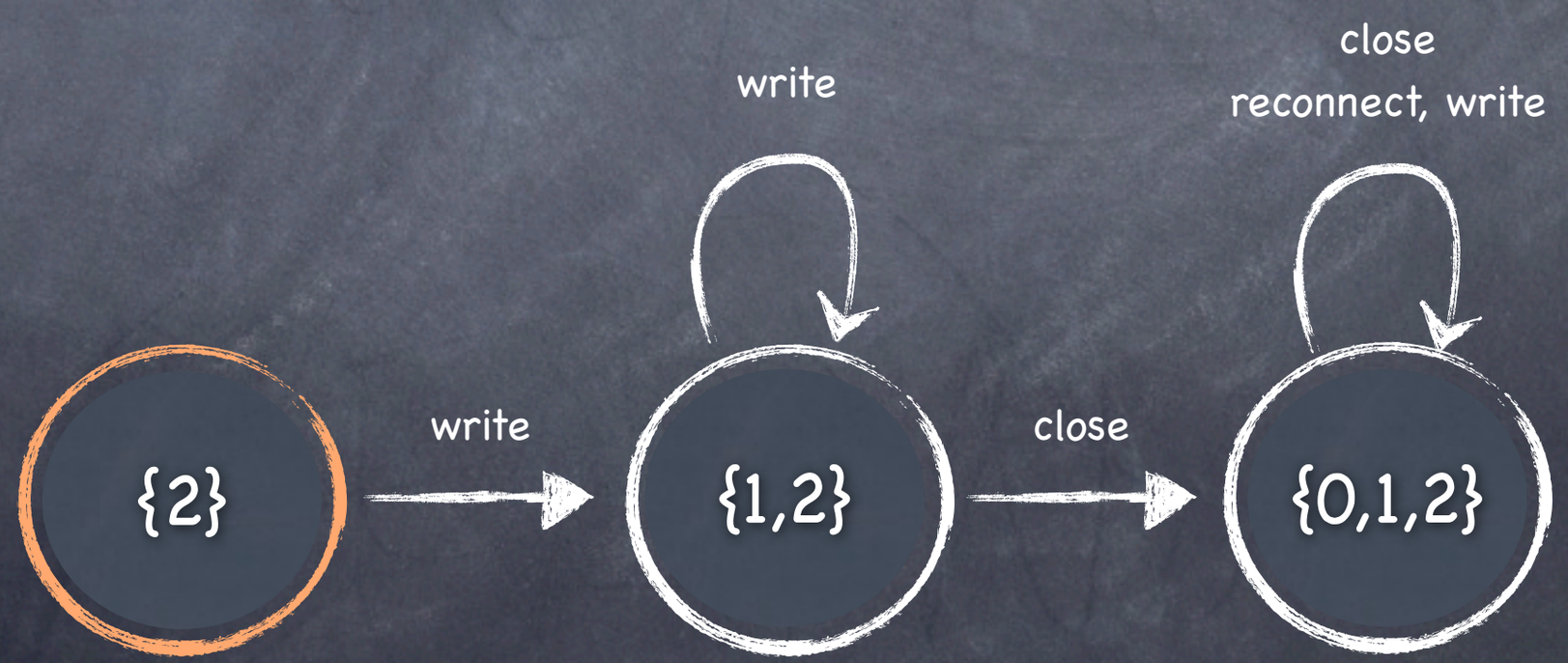

public void foo() {
<0,any>
c1.close();
x.foo();
y.bar();
}
<1,c=o(c1)> <0,c≠o(c1)>
conn.write();


public void foo() {
<0,any>
c1.close();
x.foo();
y.bar();
}
<1,c=o(c1)> <0,c≠o(c1)>
conn.write();
<2,c=o(c1)∧c=o(conn)>
<0,c≠o(c1)∨c≠o(conn)>


public void foo() {
<0,any>
c1.close();
x.foo();
<1,c=o(c1)> <0,c≠o(c1)>
conn.write();
<2,c=o(c1)∧c=o(conn)>
<0,c≠o(c1)∨c≠o(conn)>
<2,c=o(c1)> <0,c≠o(c1)>
y.bar();
}


reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect


reconnect, write
close
write
close
write
0
1
2
reconnect
close
reconnect


write
close
reconnect, write
write
close
{2}
{1,2}
{0,1,2}