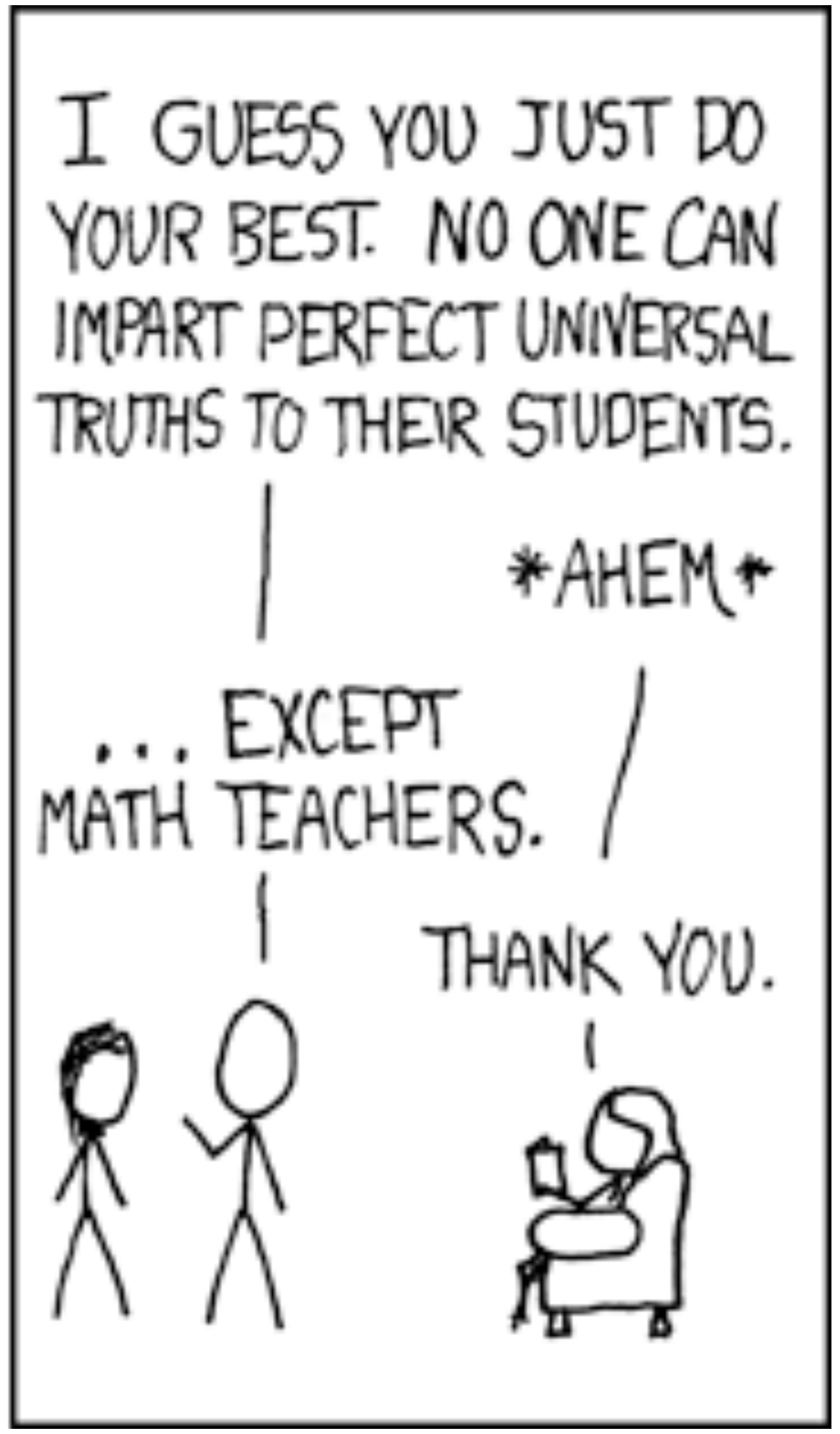
The Shape of a Program

Lisa Lippincott

ACCU, April 2018

Why don’t we routinely write down the reasoning behind our programs in a formal way, and have computers check it?

The mathematical tools we use for proofs present a poor user interface for procedural programming.



xkcd.com/263

Many people understand mathematics as describing timeless, universal truths.

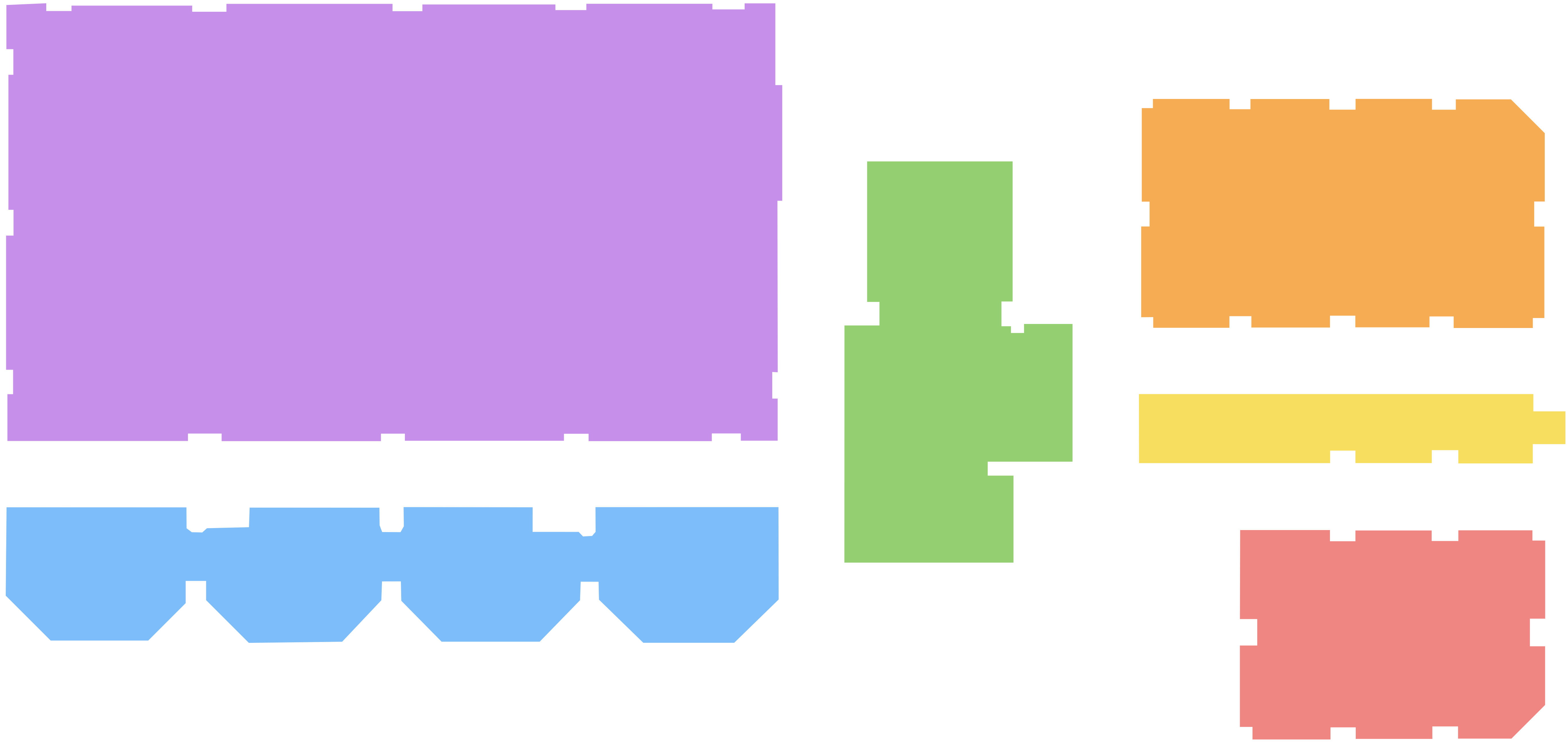
Local

I’m going to be talking about things you already know, perhaps with language you don’t know.

The code here is written in a fantasy C++, with extensions that make proofs fit into the code.

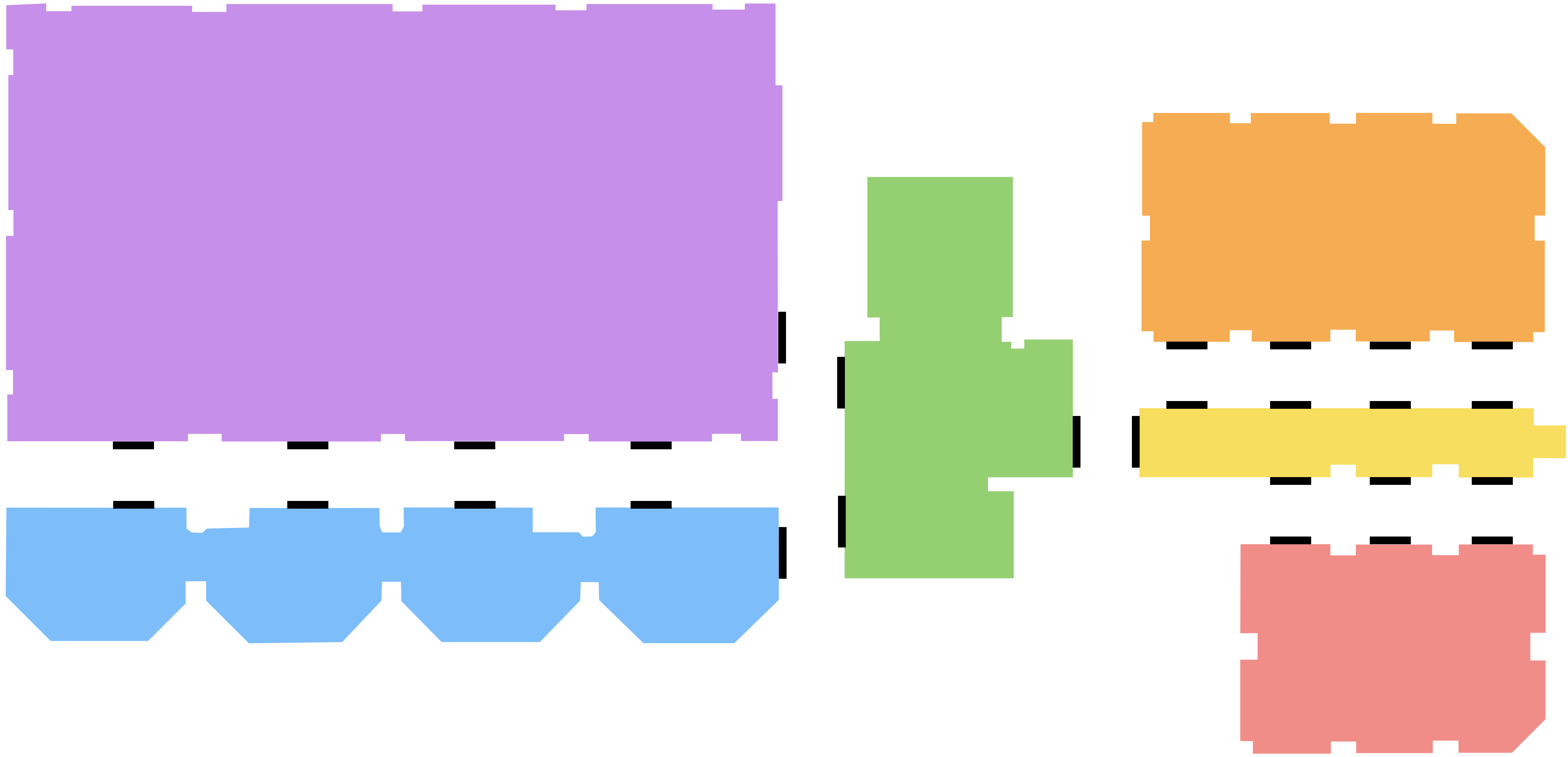
Topology is about places.

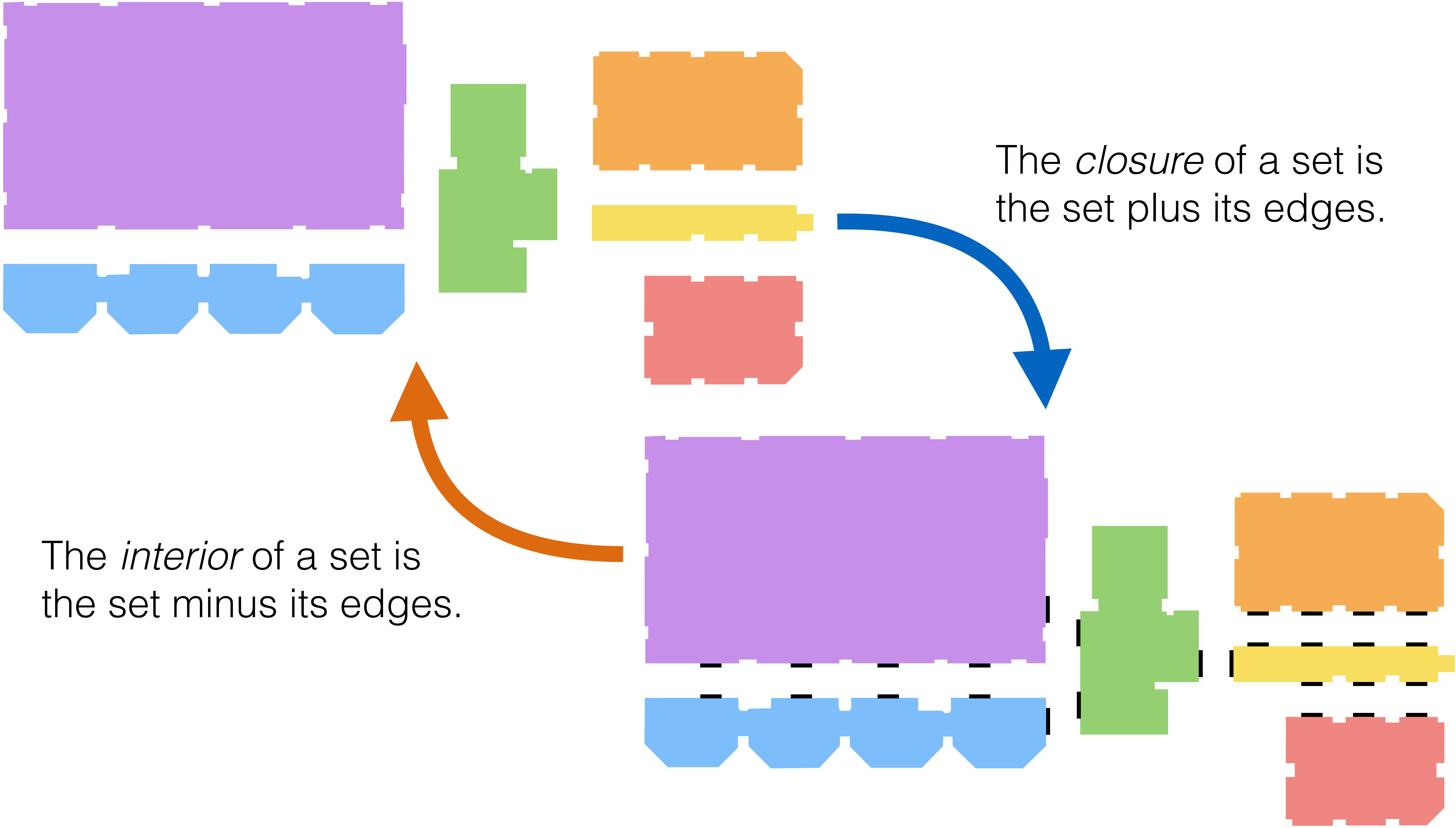
And especially about the thresholds in between places.



Open sets are meaningful areas, not including their edges.

Closed sets are meaningful areas, including their edges.





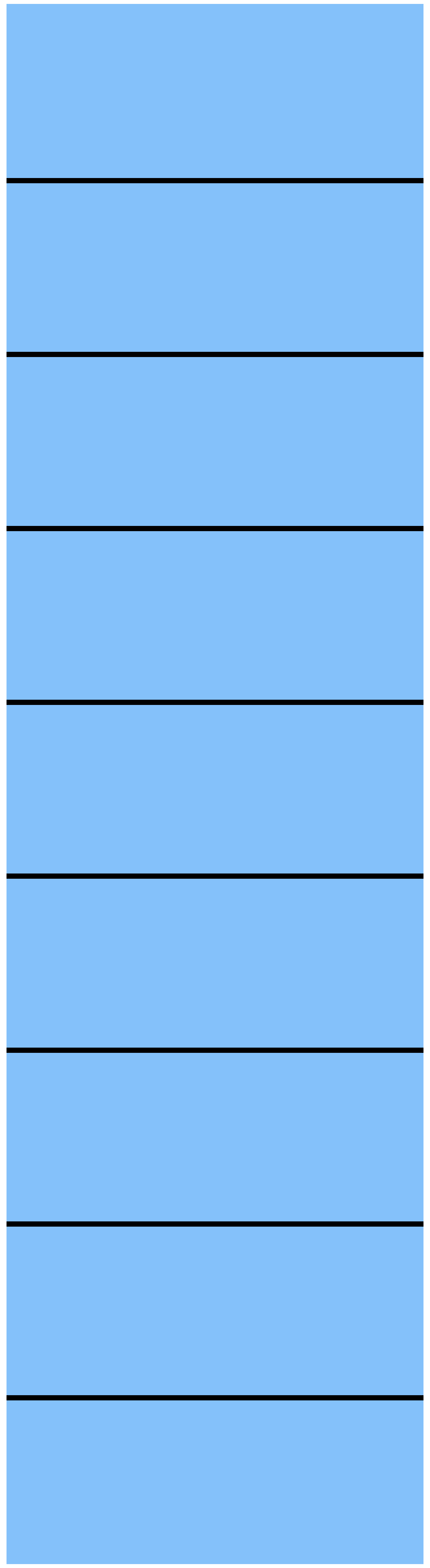
The *closure* of a set is
the set plus its edges.

The *interior* of a set is
the set minus its edges.

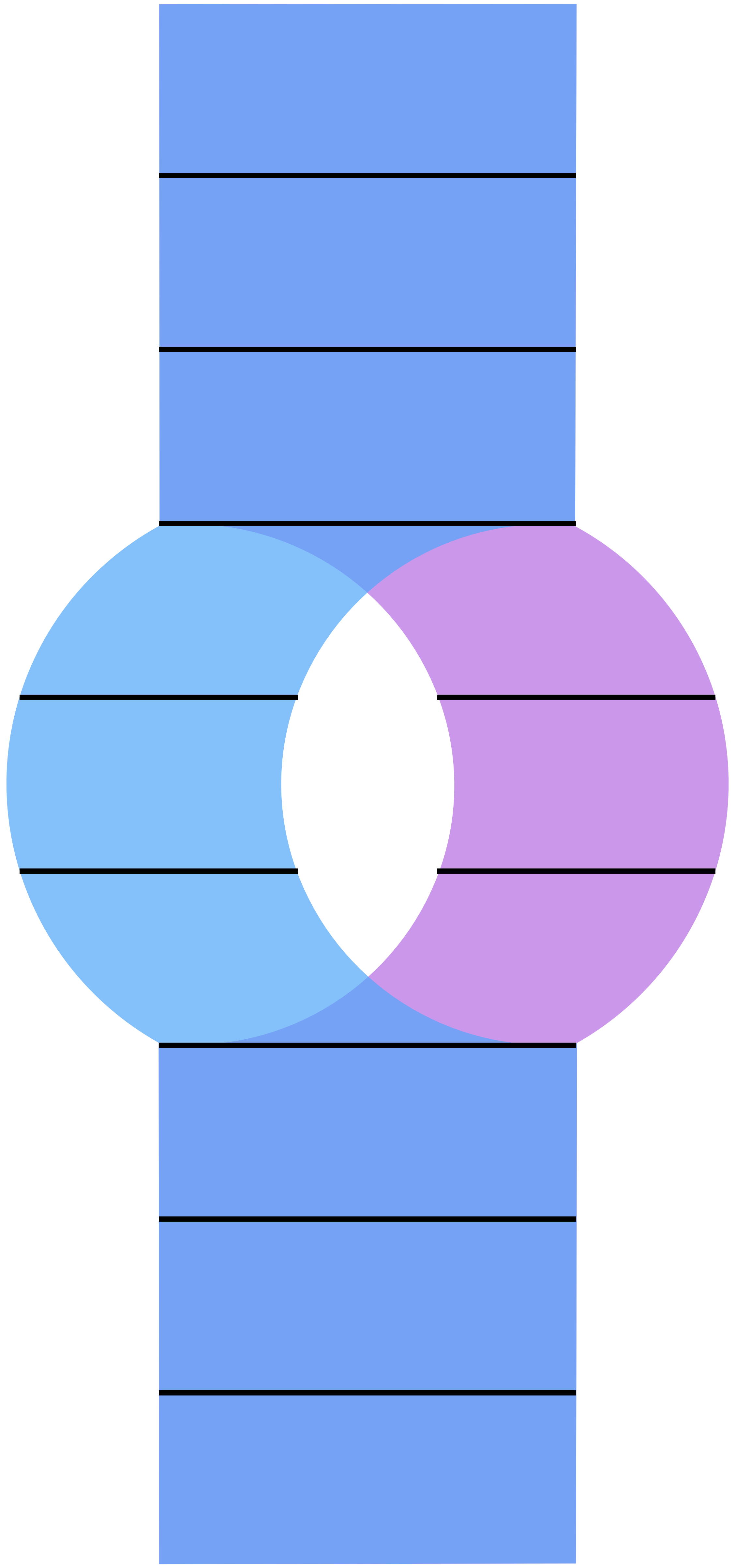
Areas are *adjacent* if they share an edge.

Chains of adjacency *connect* areas together.

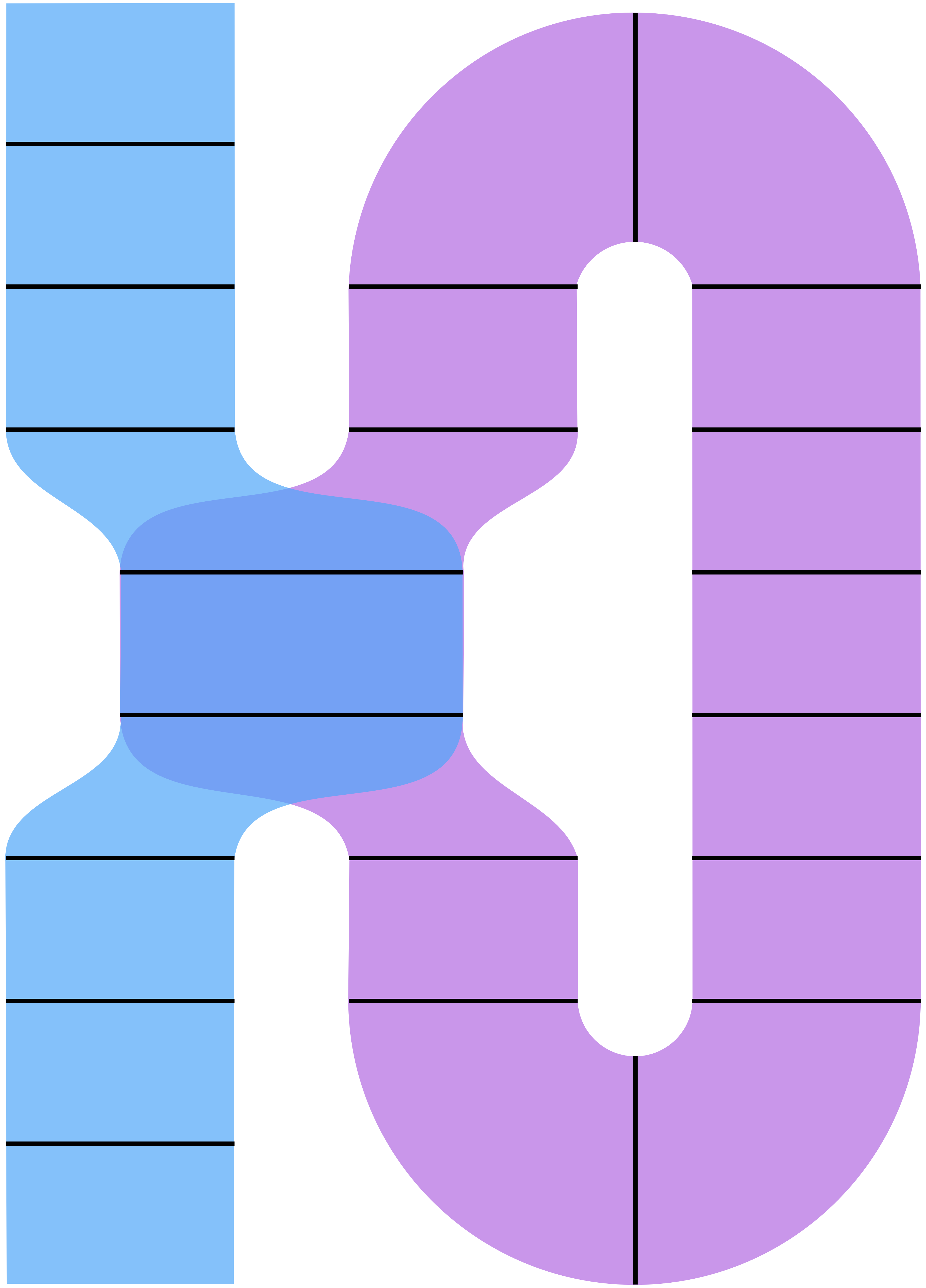
Sequence

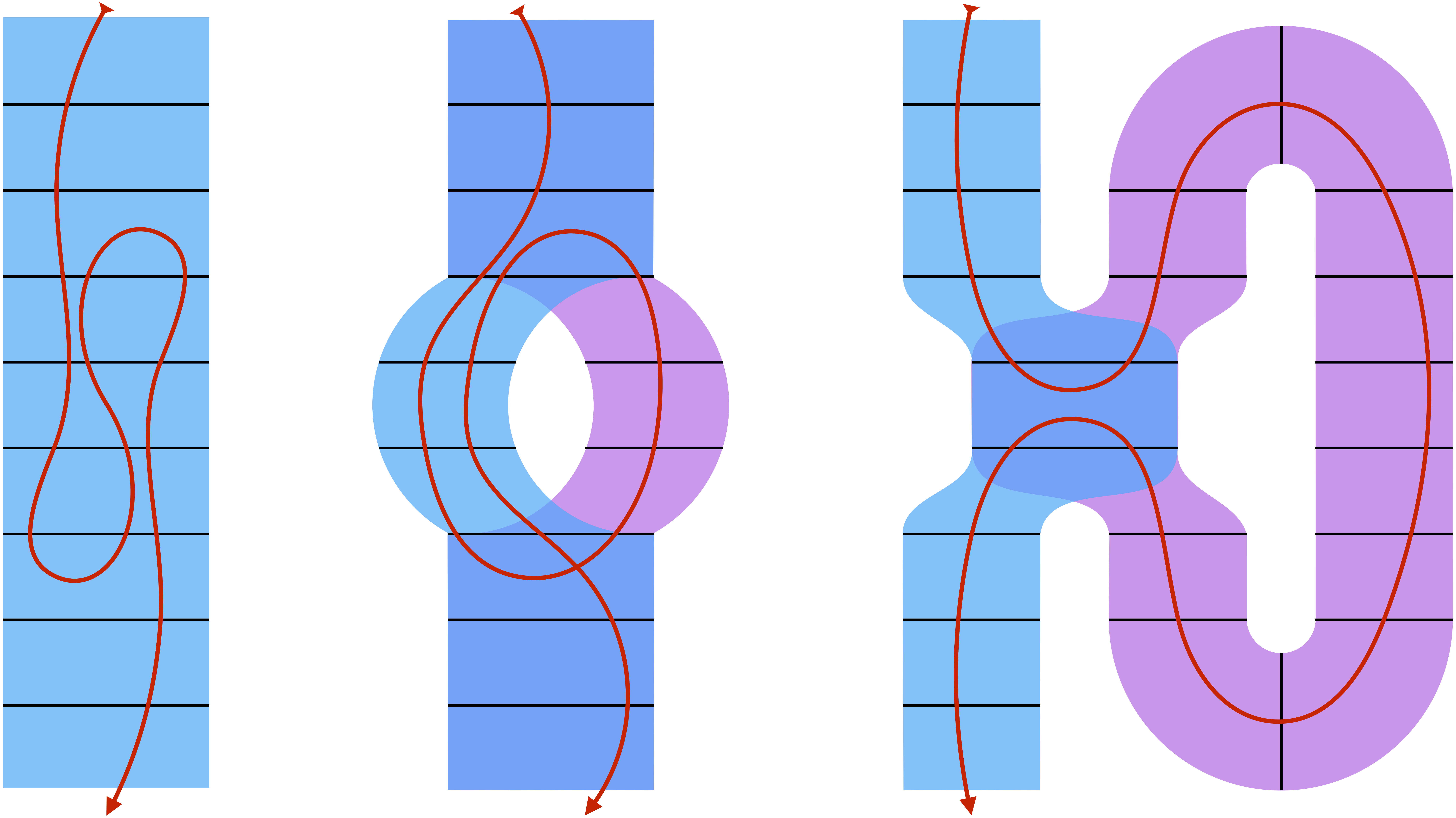


Branch



Loop

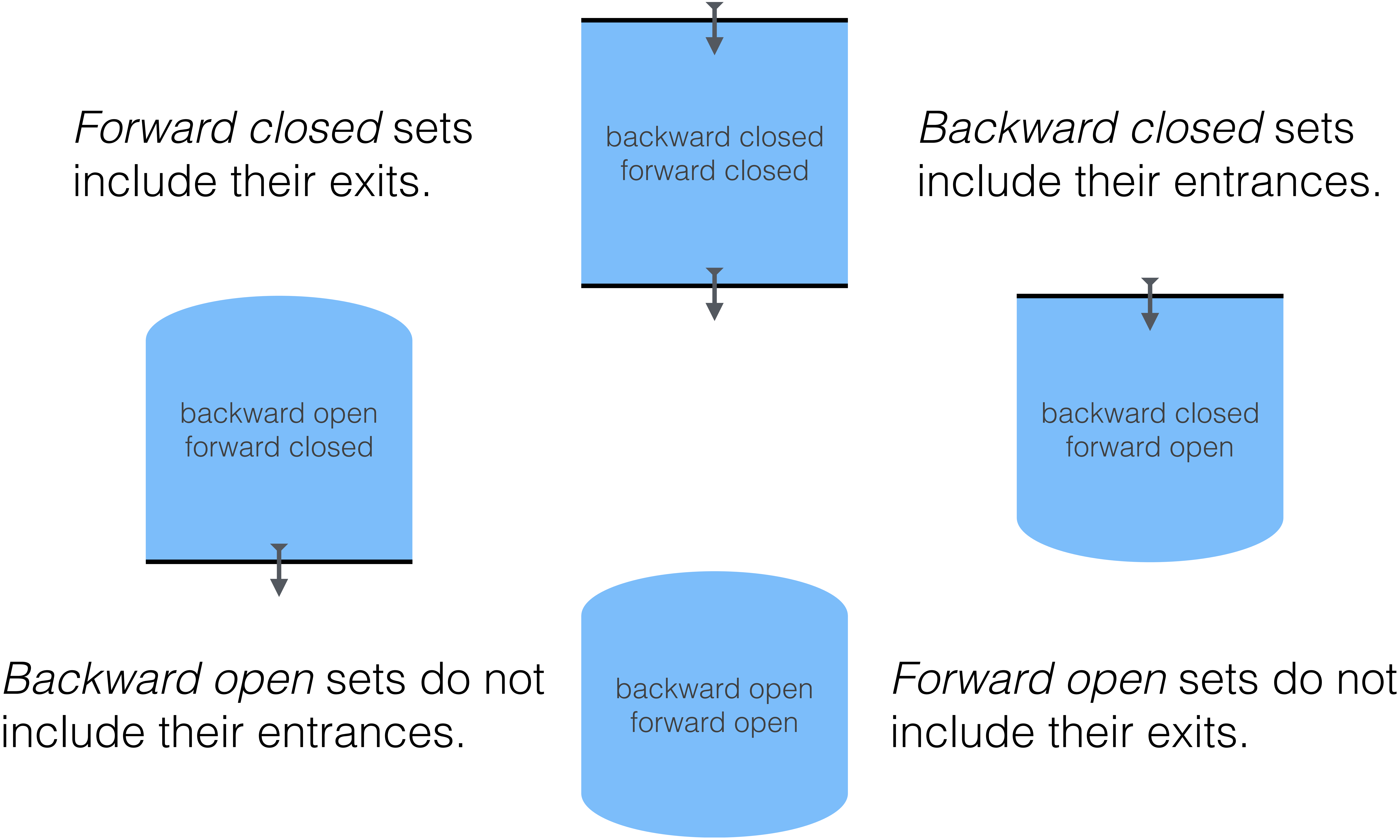




Sequence

Branch

Loop



Forward closed sets
include their exits.

backward closed
forward closed

Backward closed sets
include their entrances.

backward open
forward closed

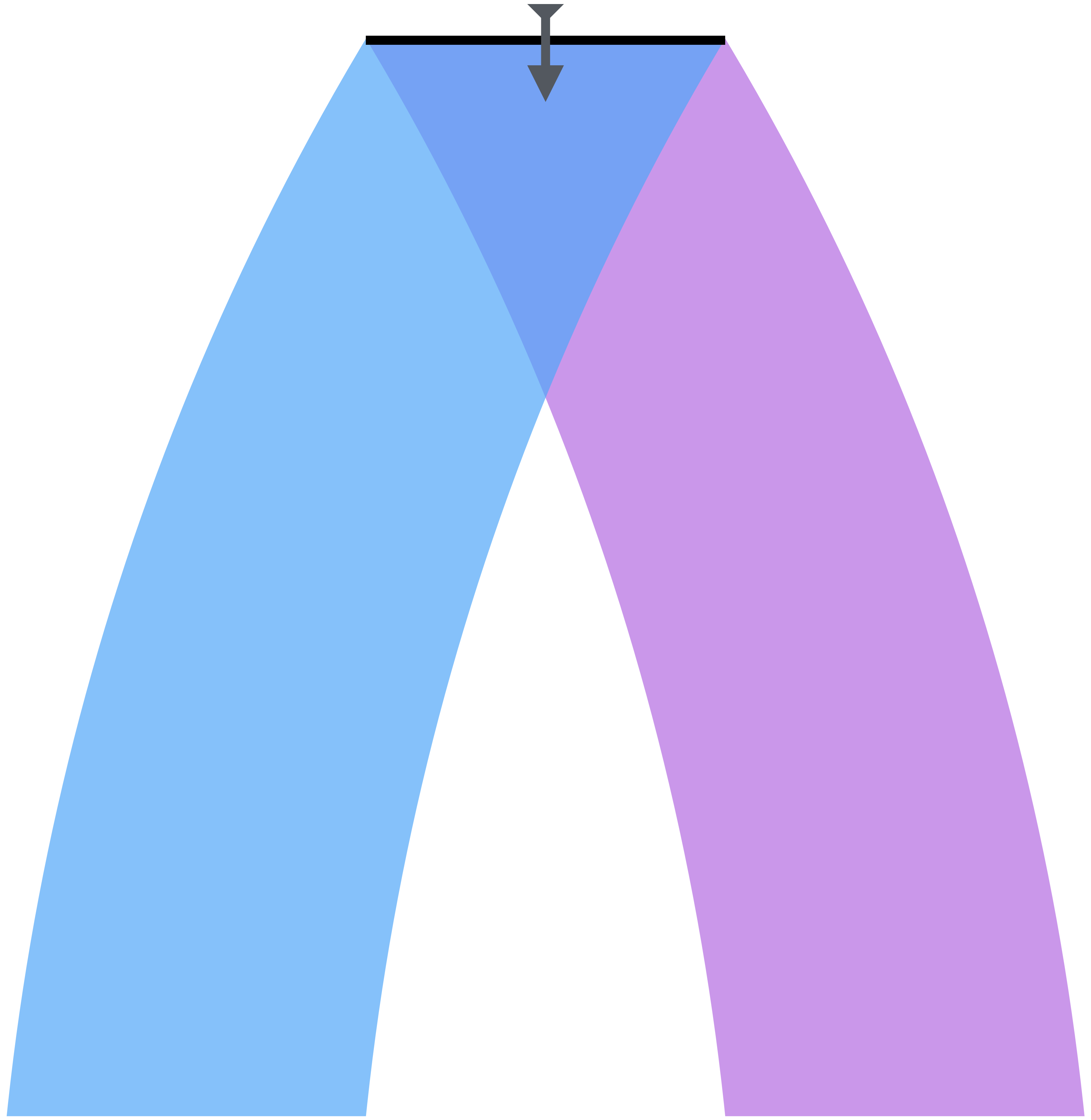
backward closed
forward open

Backward open sets do not
include their entrances.

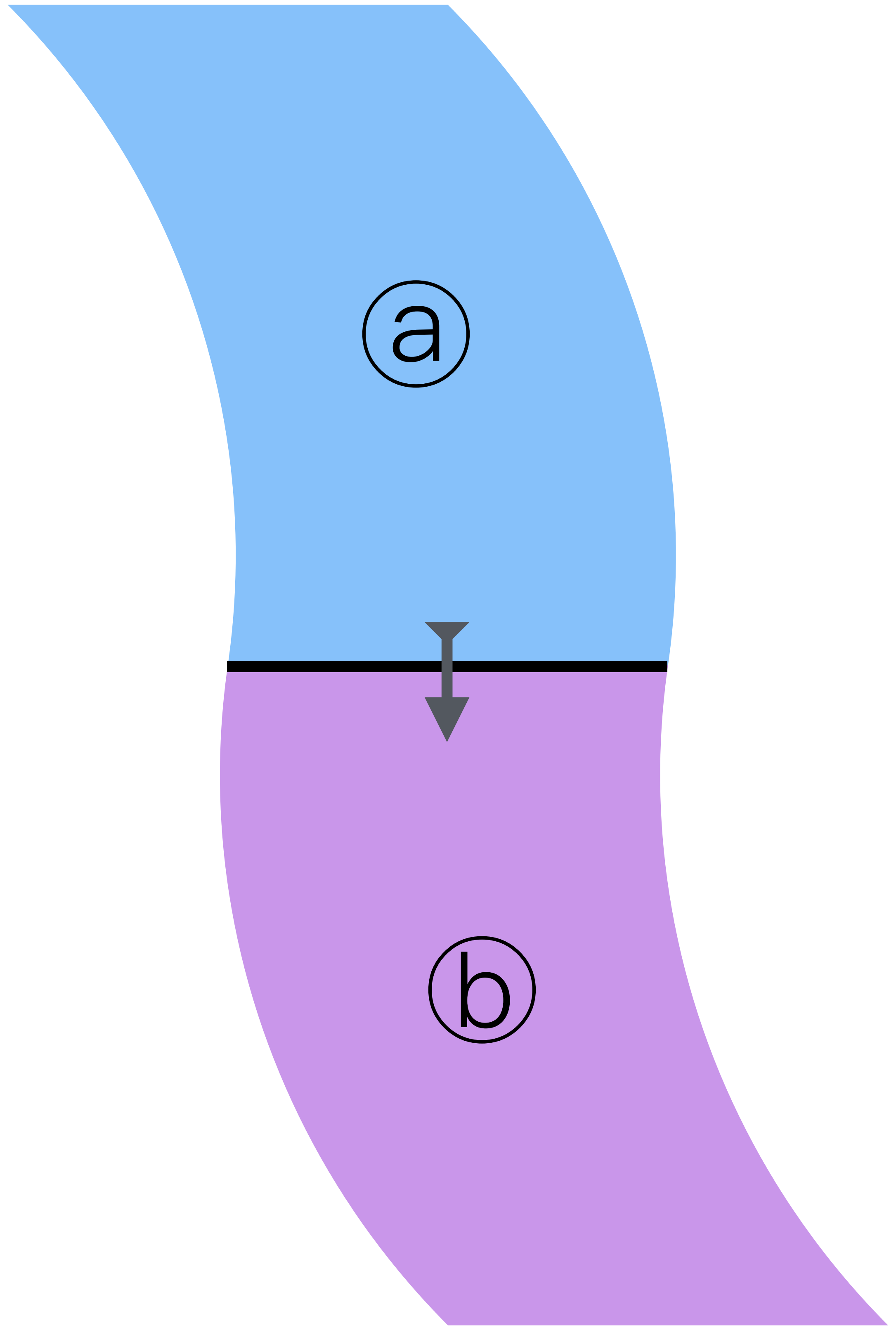
backward open
forward open

Forward open sets do not
include their exits.

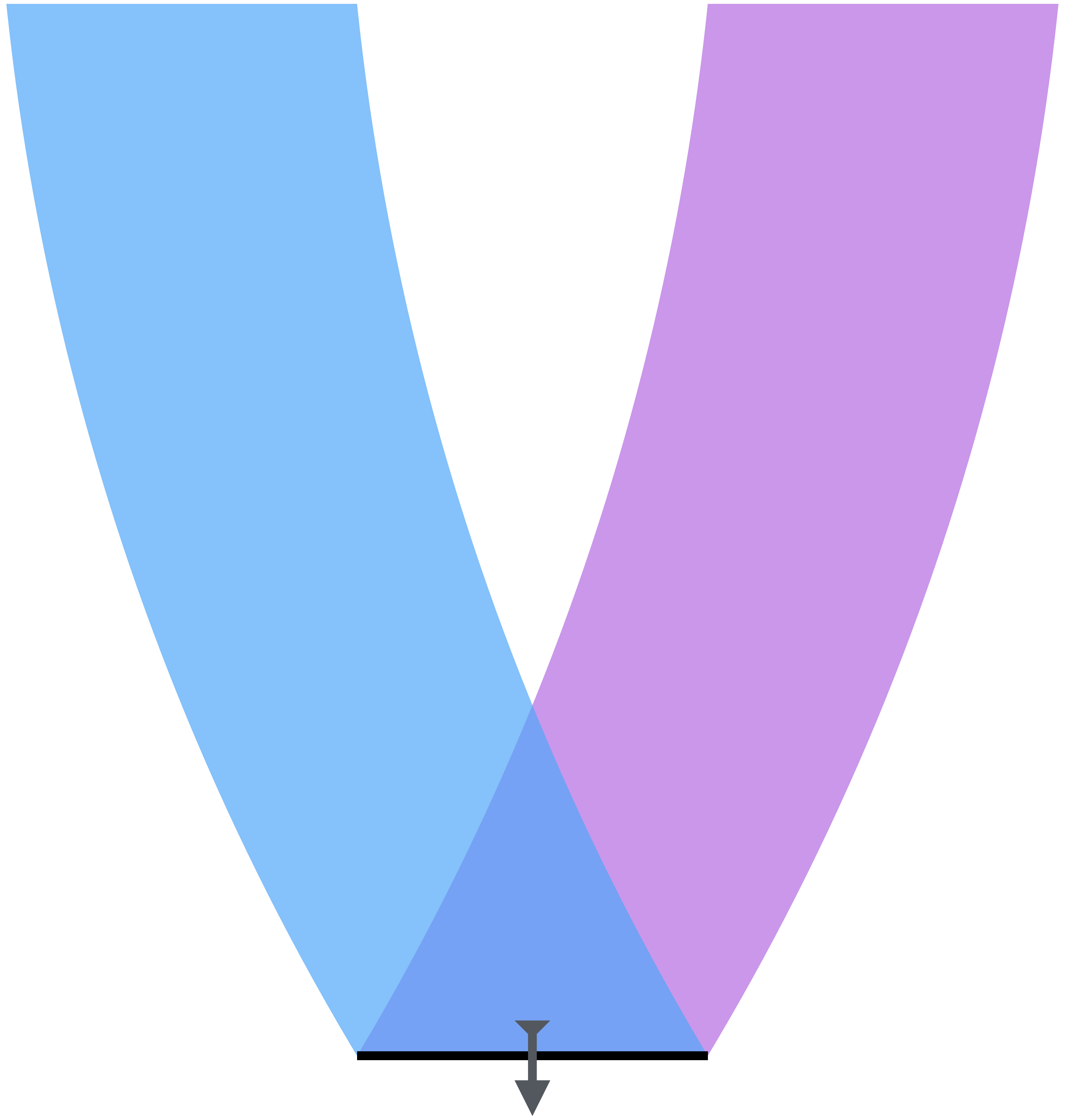
Not adjacent



Adjacent

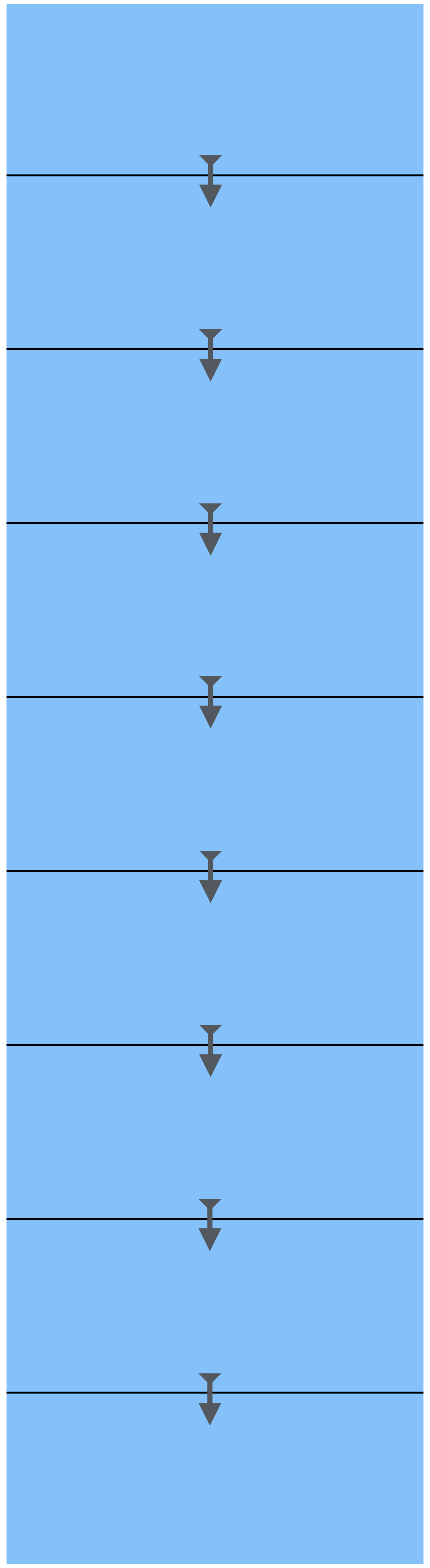


Not adjacent

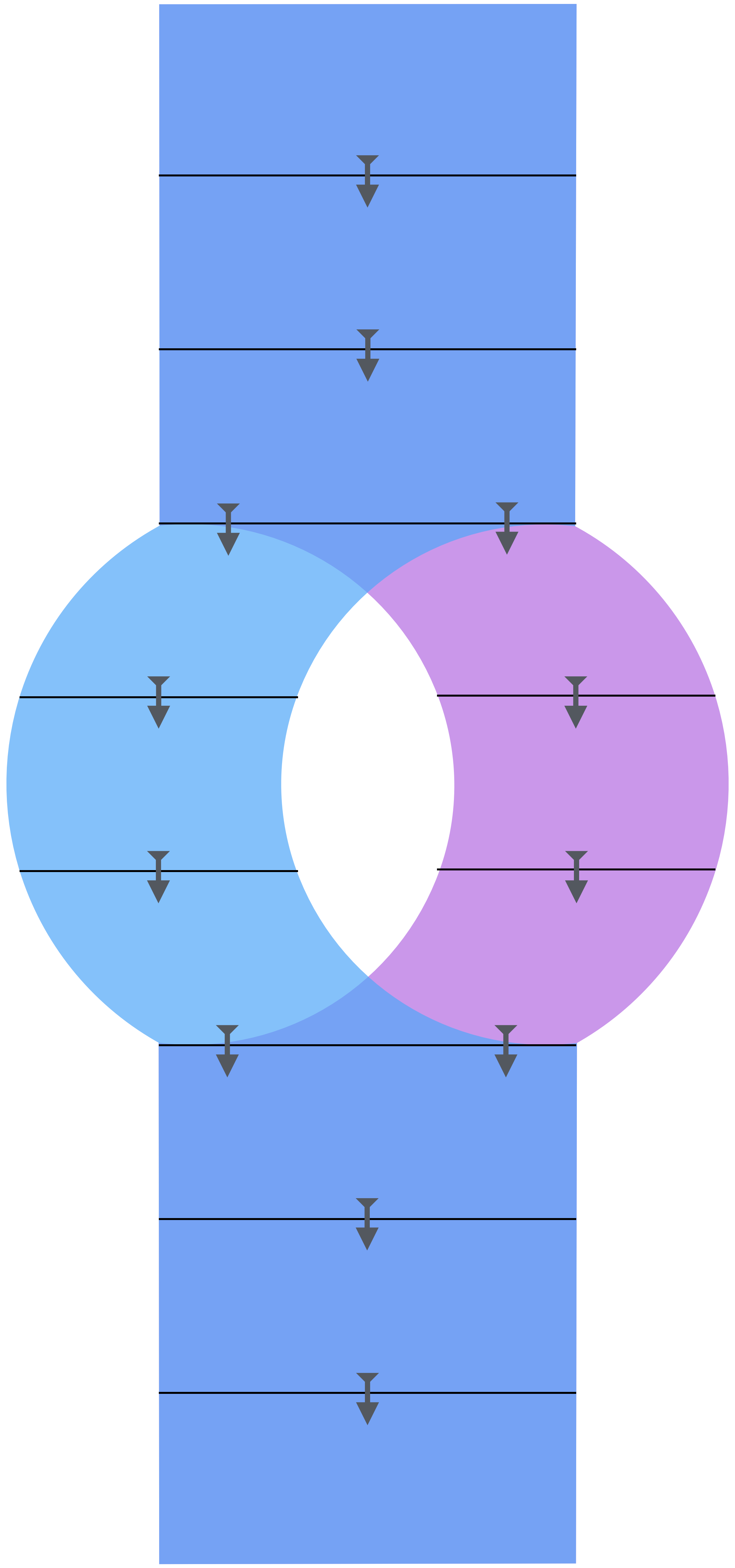


ⓐ immediately precedes ⓑ

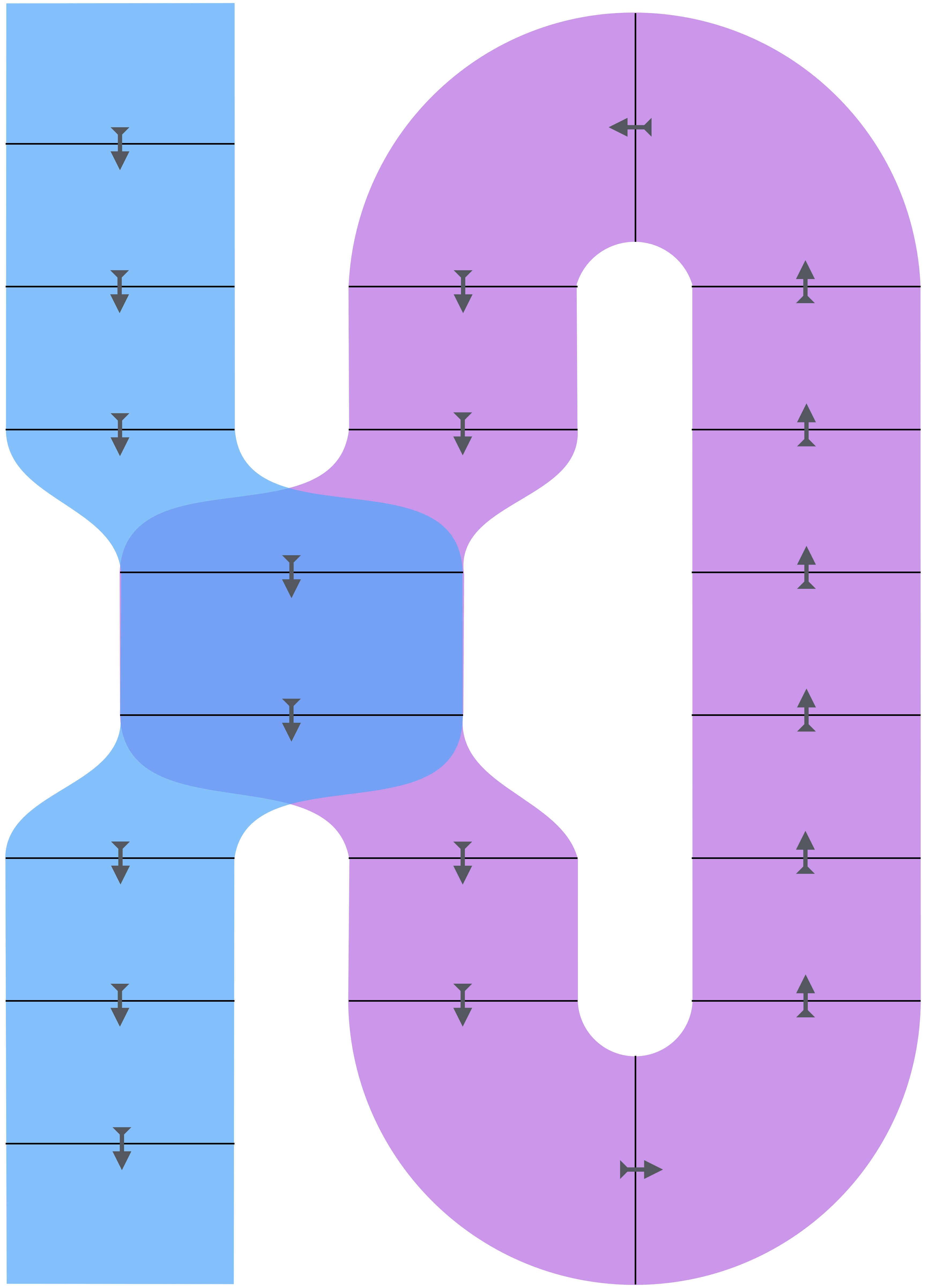
Sequence

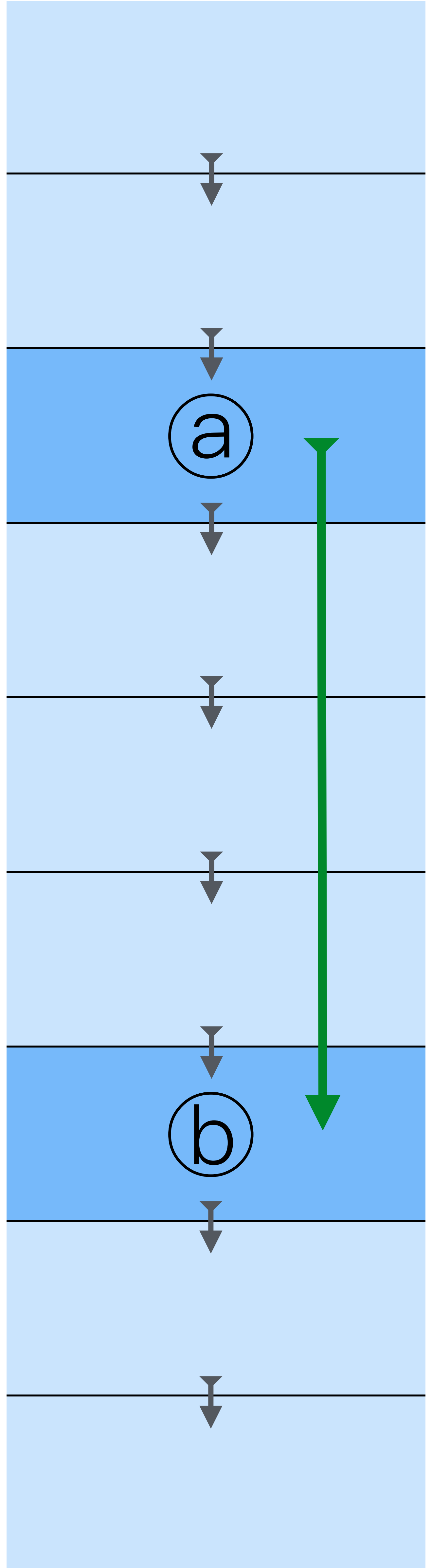


Branch



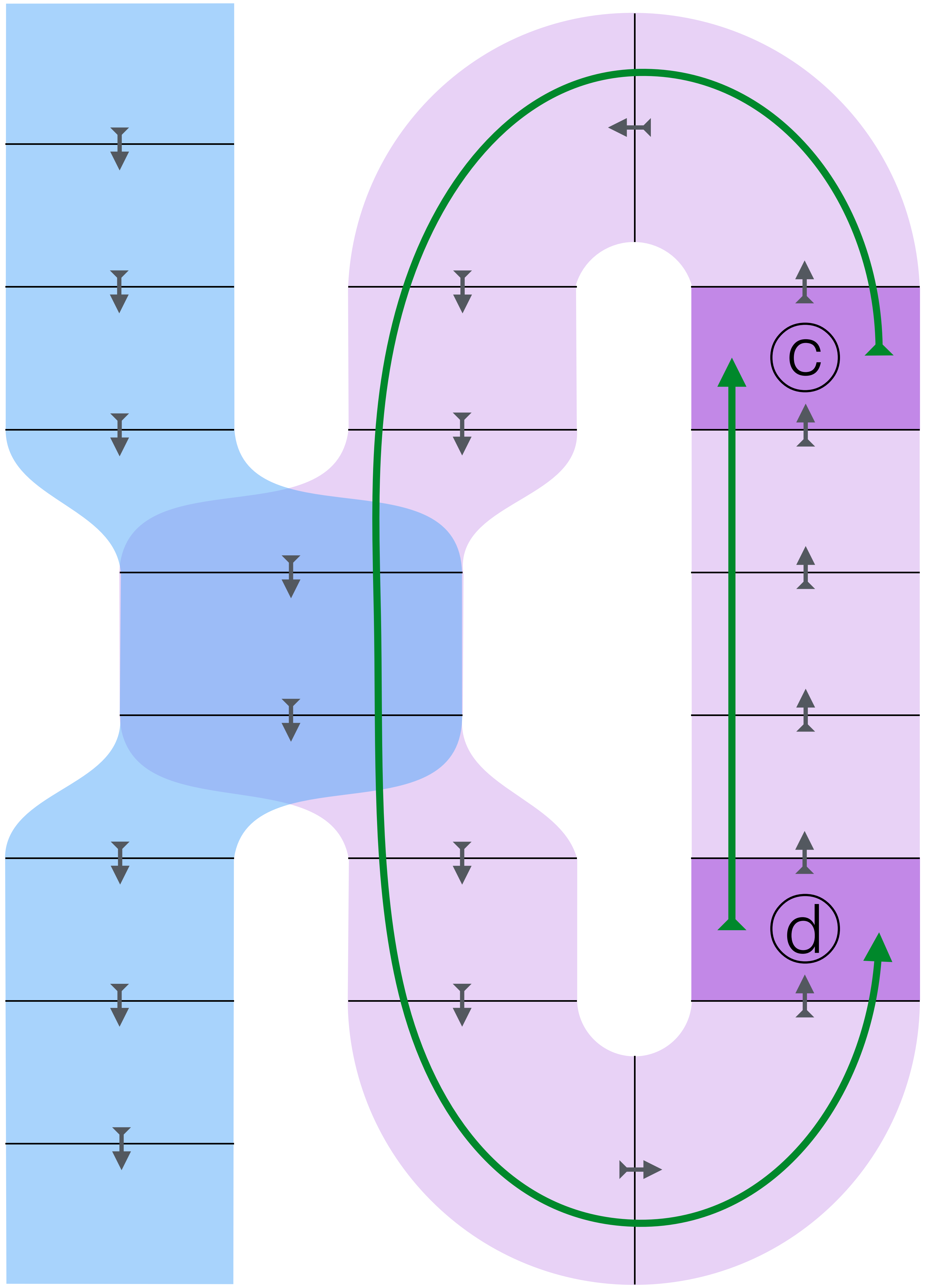
Loop



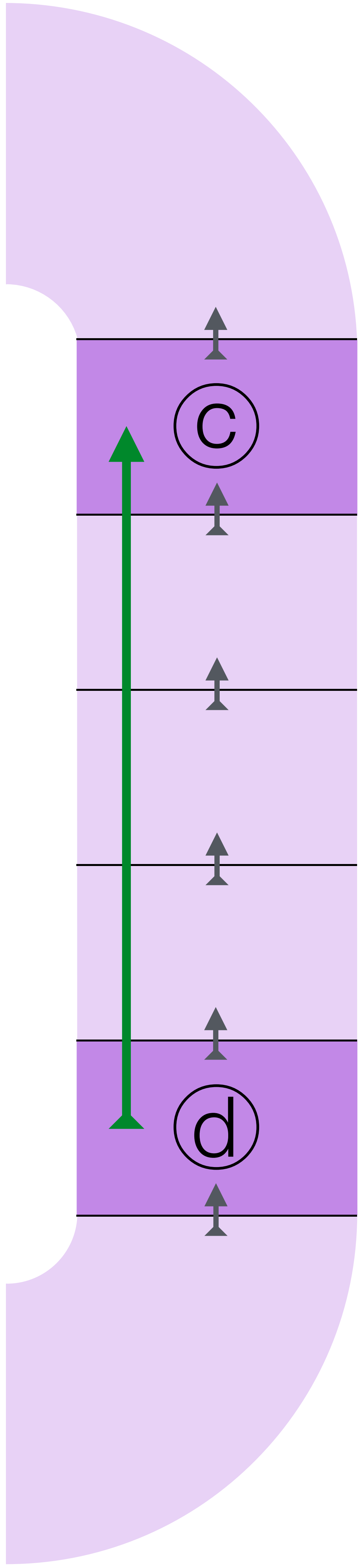


ⓐ is before ⓑ:
there is a connection from ⓐ to ⓑ, but
there is no connection from ⓑ to ⓐ.

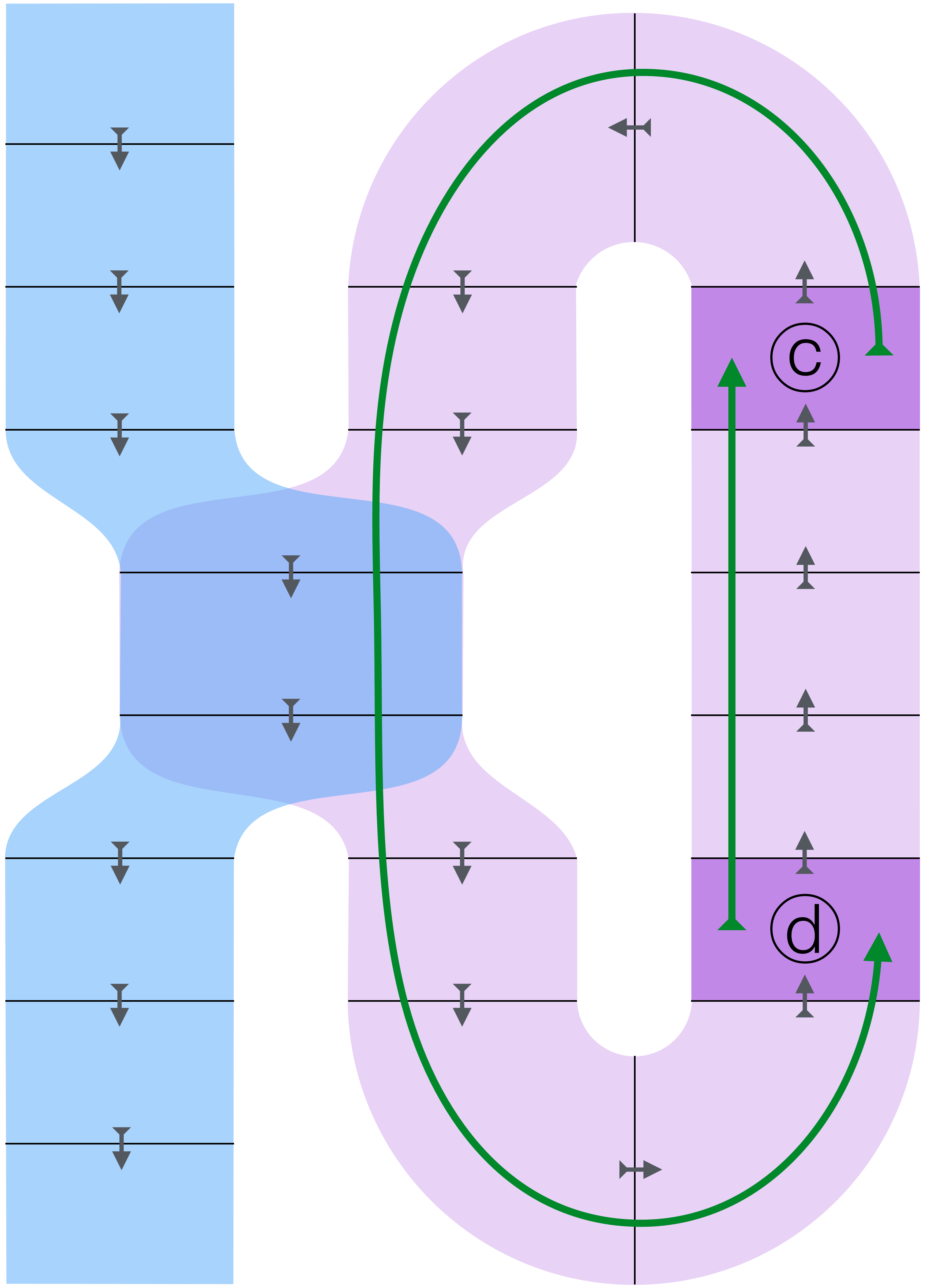
ⓒ is both before and after ⓓ:
there is a connection from ⓒ to ⓓ, and
there is also a connection from ⓓ to ⓒ.

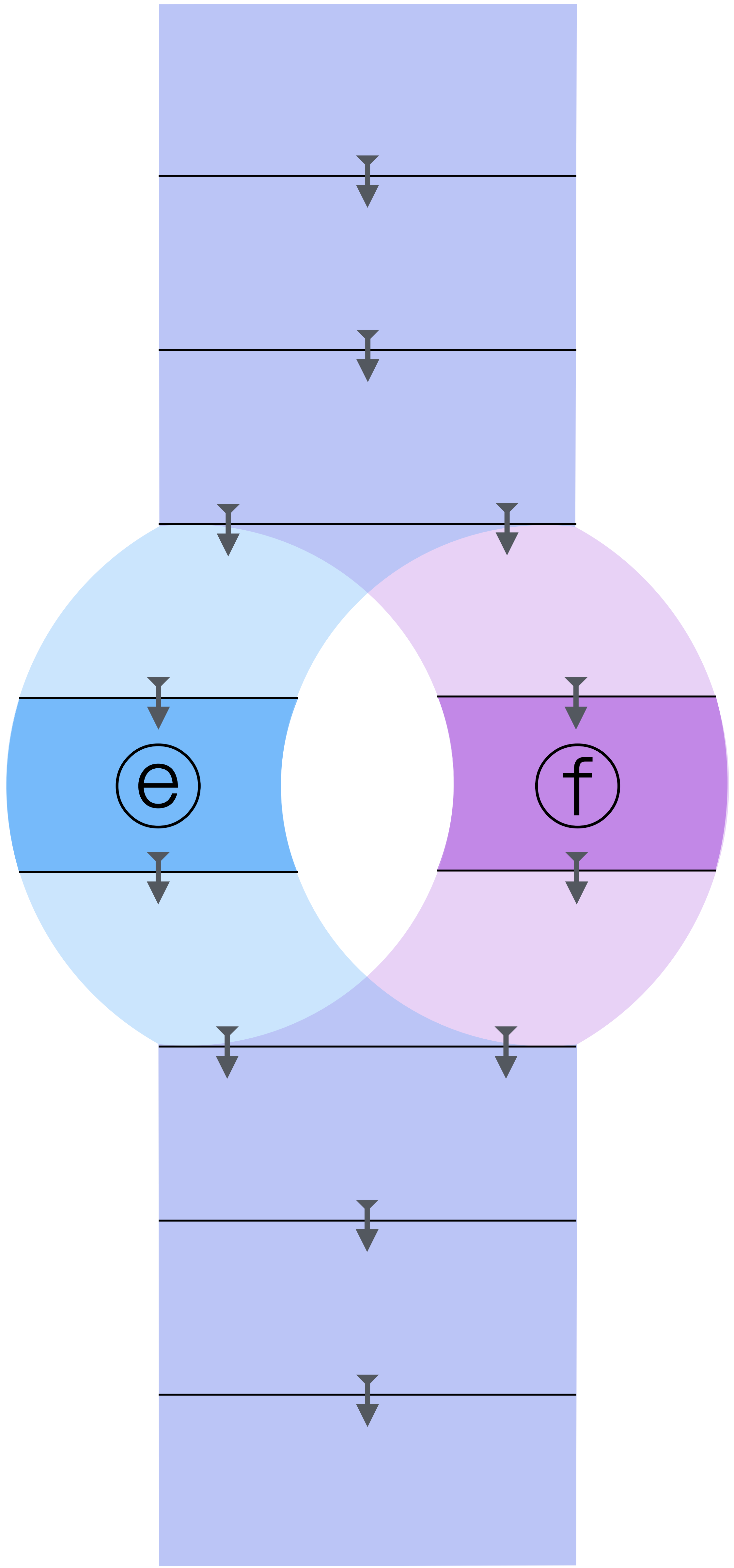


But in this smaller neighborhood, ⓒ is after ⓓ: there is no connection from ⓒ to ⓓ, but there is a connection from ⓓ to ⓒ.

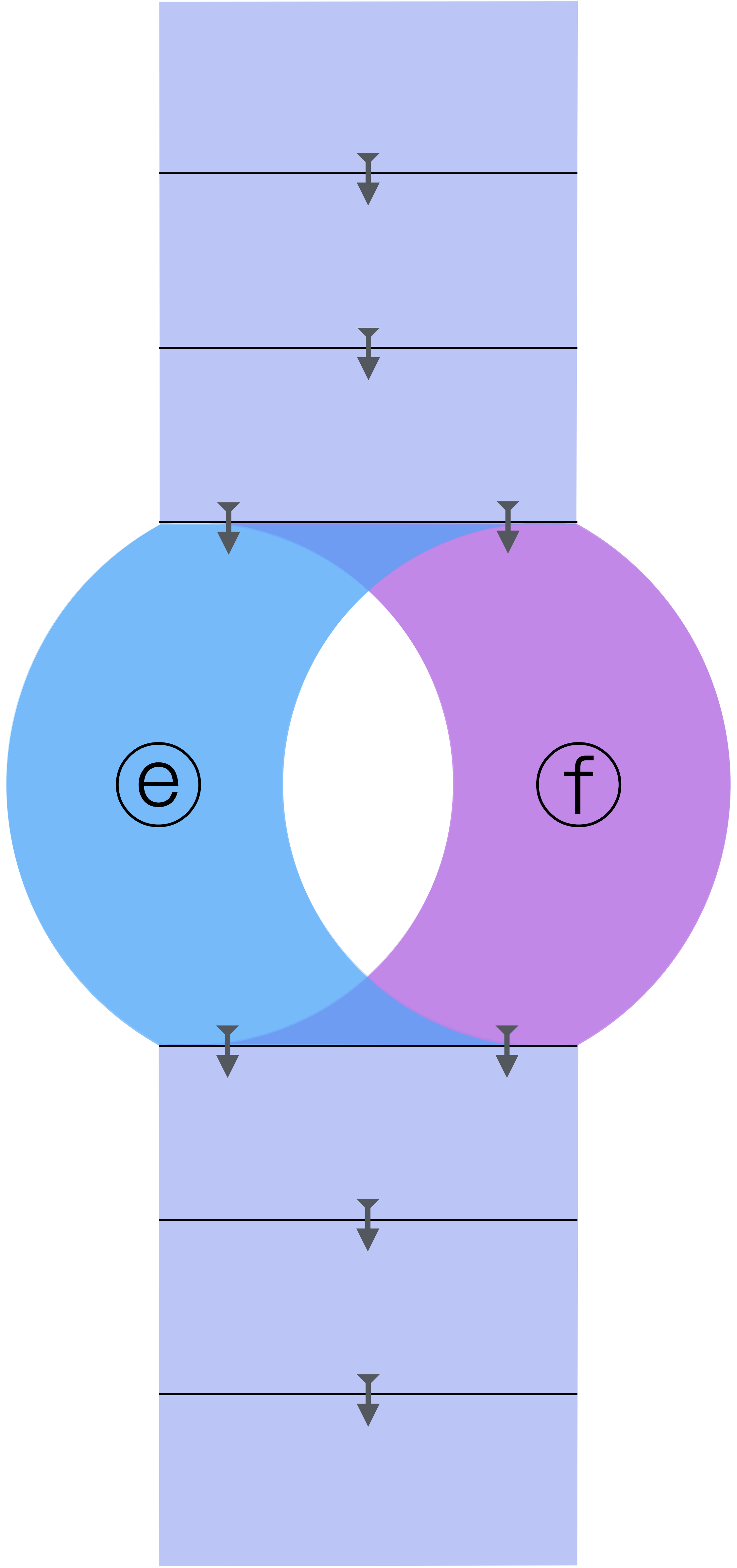


ⓒ is both before and after ⓓ:
there is a connection from ⓒ to ⓓ, and
there is also a connection from ⓓ to ⓒ.



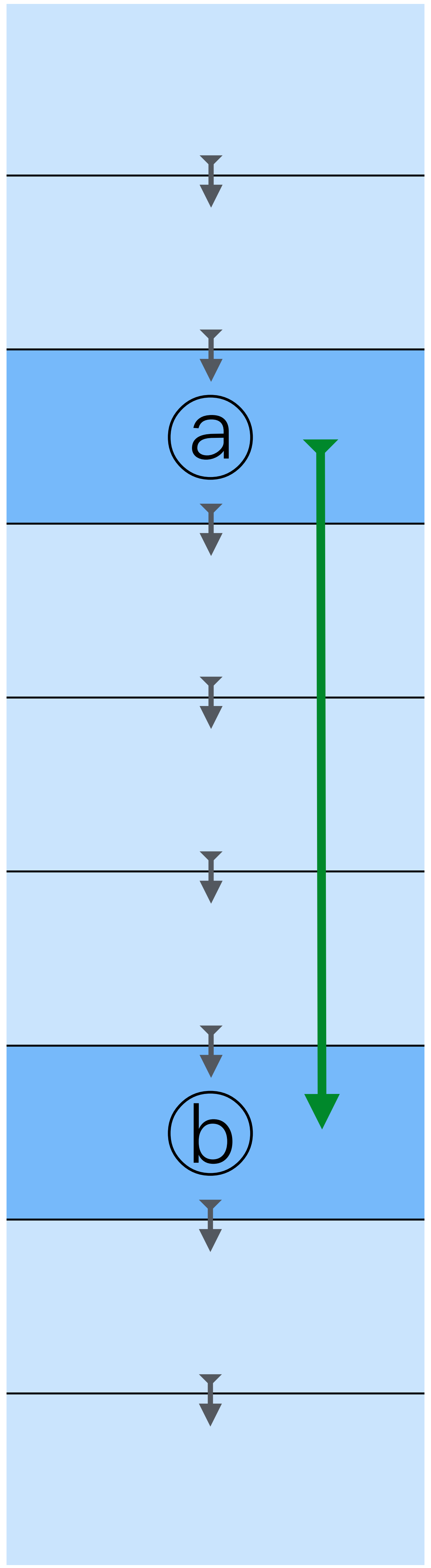


ⓔ and ⓕ are alternative possibilities:
there is a connection neither
from ⓔ to ⓕ nor from ⓕ to ⓔ.

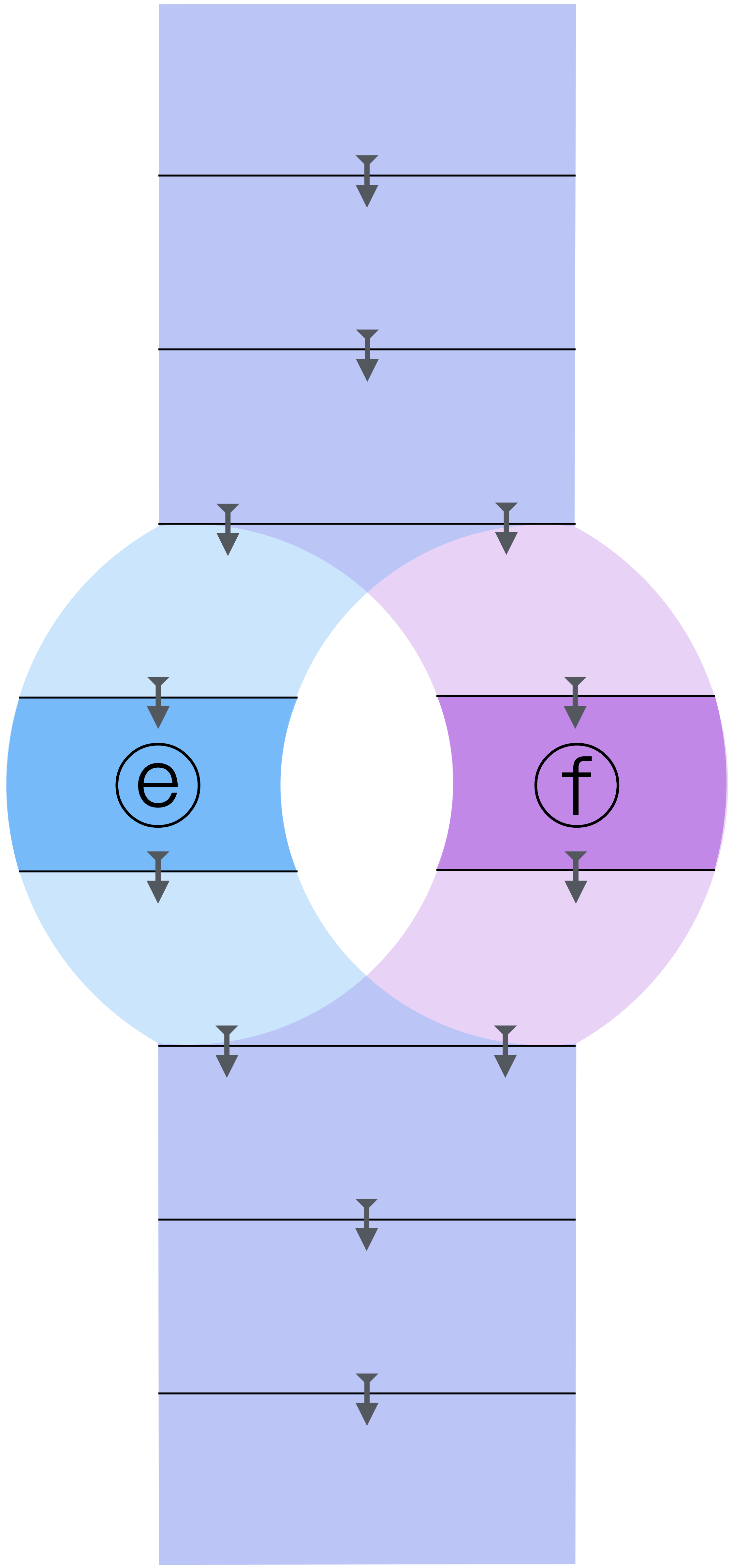


If we expand ⓔ and ⓕ to share entrances and exits, they remain alternative possibilities.

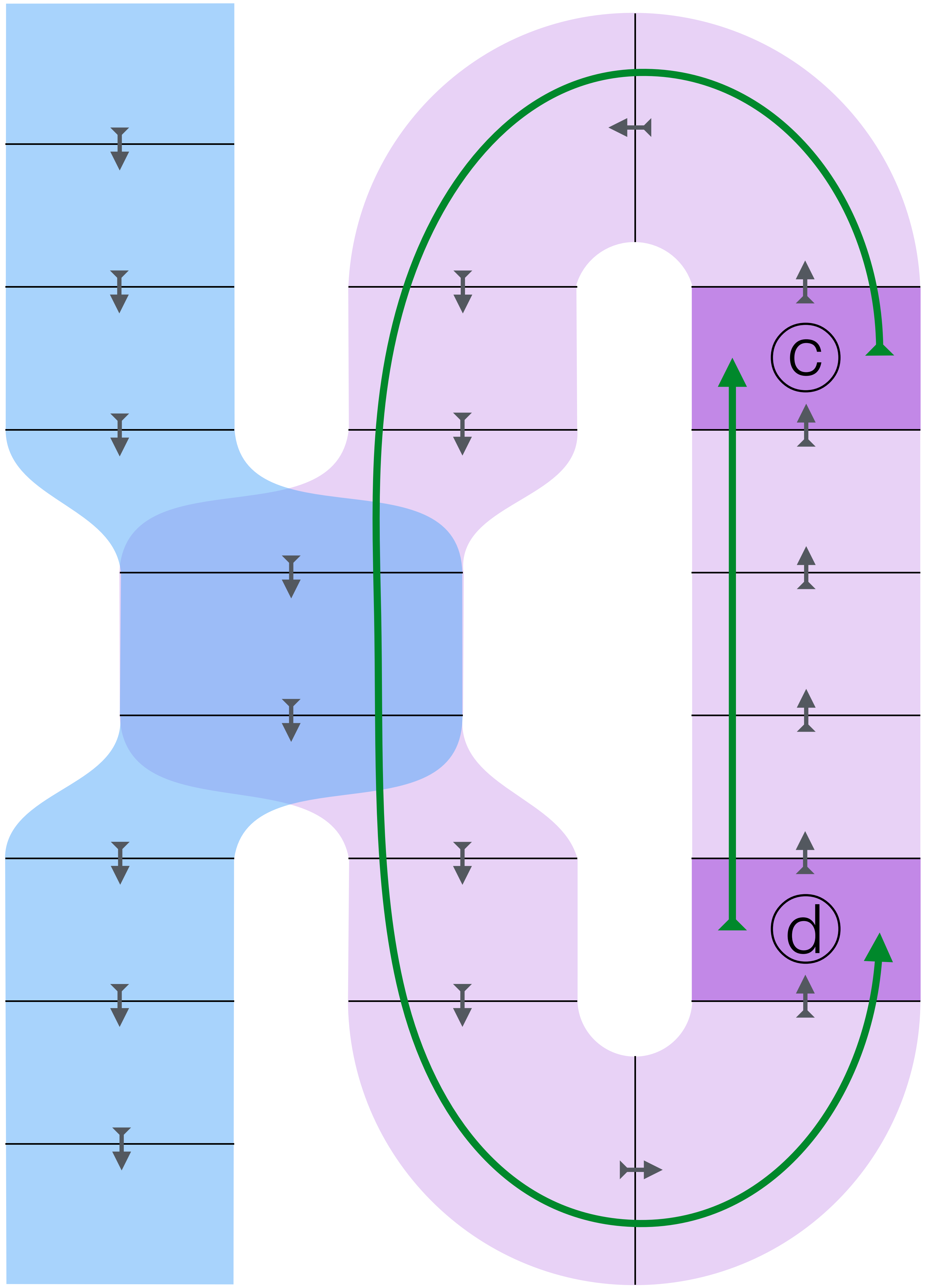
ⓐ is before ⓑ

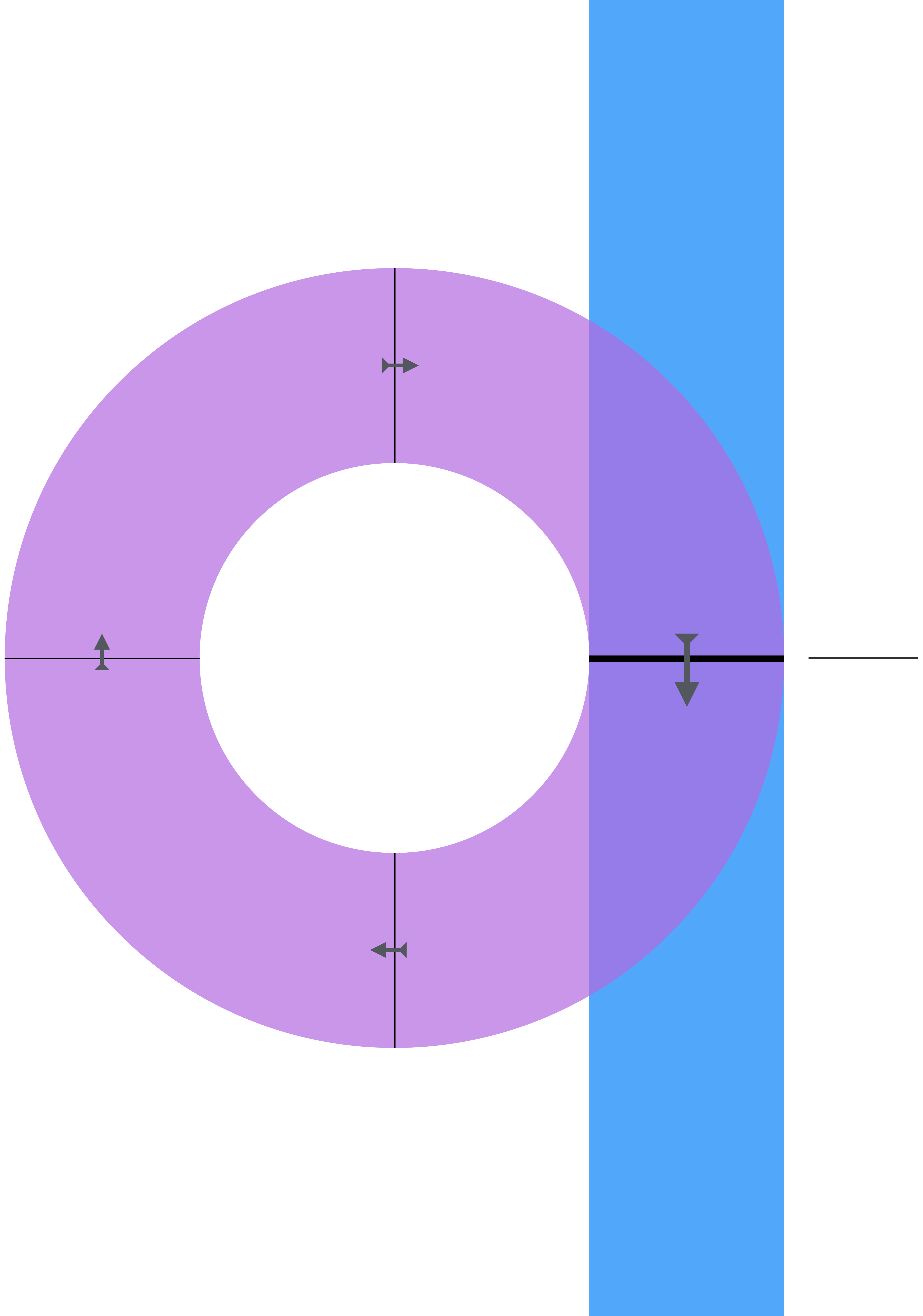


ⓔ is alternative to ⓕ



ⓒ is both before and after ⓓ





Assertions are experiments.

Successful assertions are repeatable, and have no meaningful effect.

Assertion edge

An assertion describes its edge, in dimensions of space and possibility.

Some things need to be asserted, but not govern branches:

readable(const T&)
writable(T&)

destructible(T&)

deallocatable(void *, size_t)
array_deallocatable(void *, size_t)

exception_is_rethrowable()

dynamic_type_identifiable(T&)

dereferencable(Iterator)
reachable(Iterator, Iterator)

resizable(vector<T>&)
reallocatable(vector<T>&)

fclosable(int)

in_the_past(time_point<steady_clock>)

proper(T&)

Capabilities can be asserted, but can't govern branches:

readable(const T&)
writable(T&)

destructible(T&)

deallocatable(void *, size_t)
array_deallocatable(void *, size_t)

exception_is_rethrowable()

dynamic_type_identifiable(T&)

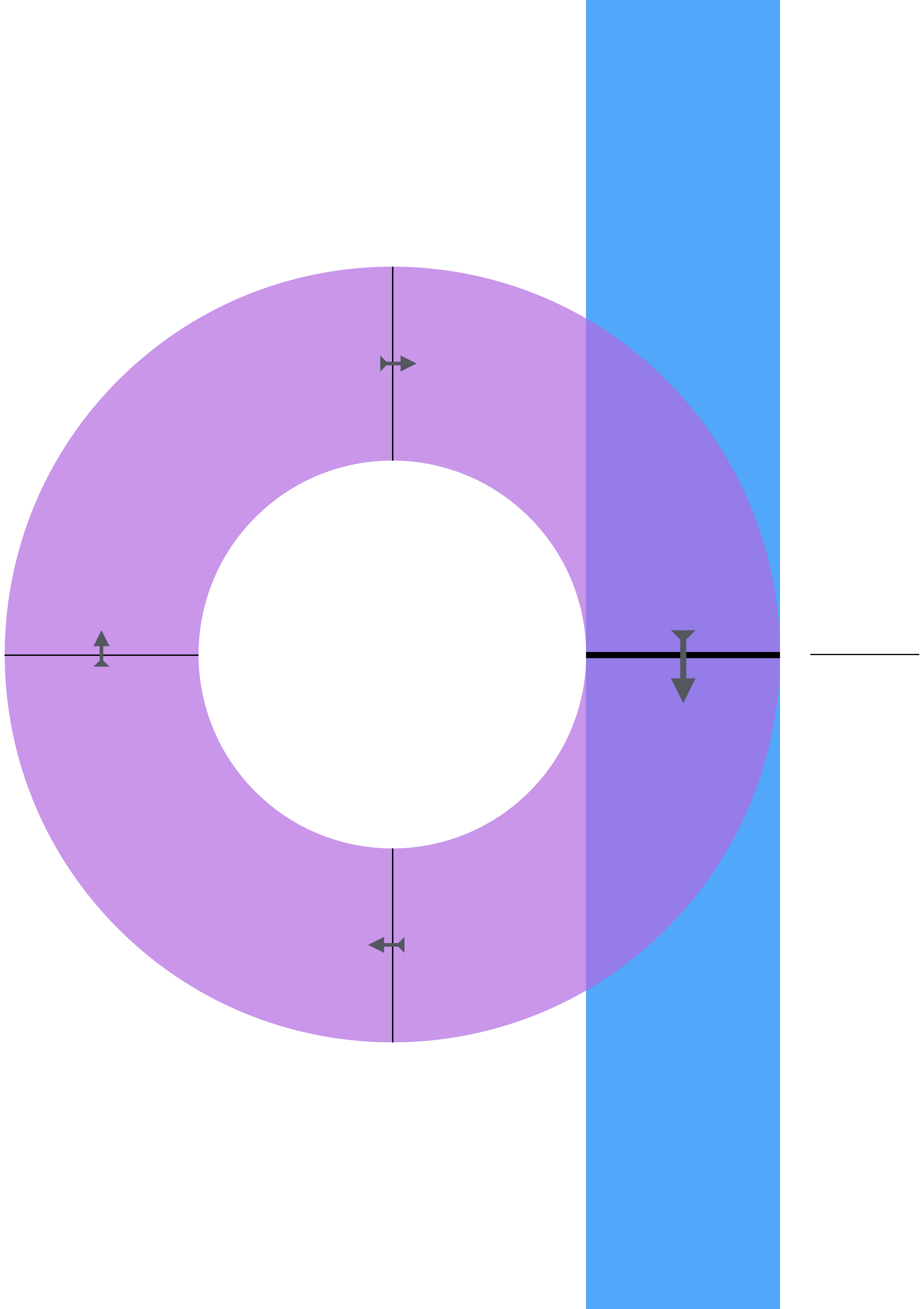
dereferencable(Iterator)
reachable(Iterator, Iterator)

resizable(vector<T>&)
reallocatable(vector<T>&)

fclosable(int)

~~in_the_past(time_point<steady_clock>)~~
memorable(time_point<steady_clock>)

~~proper(T&)~~
usable(T&)



Assertions are experiments.

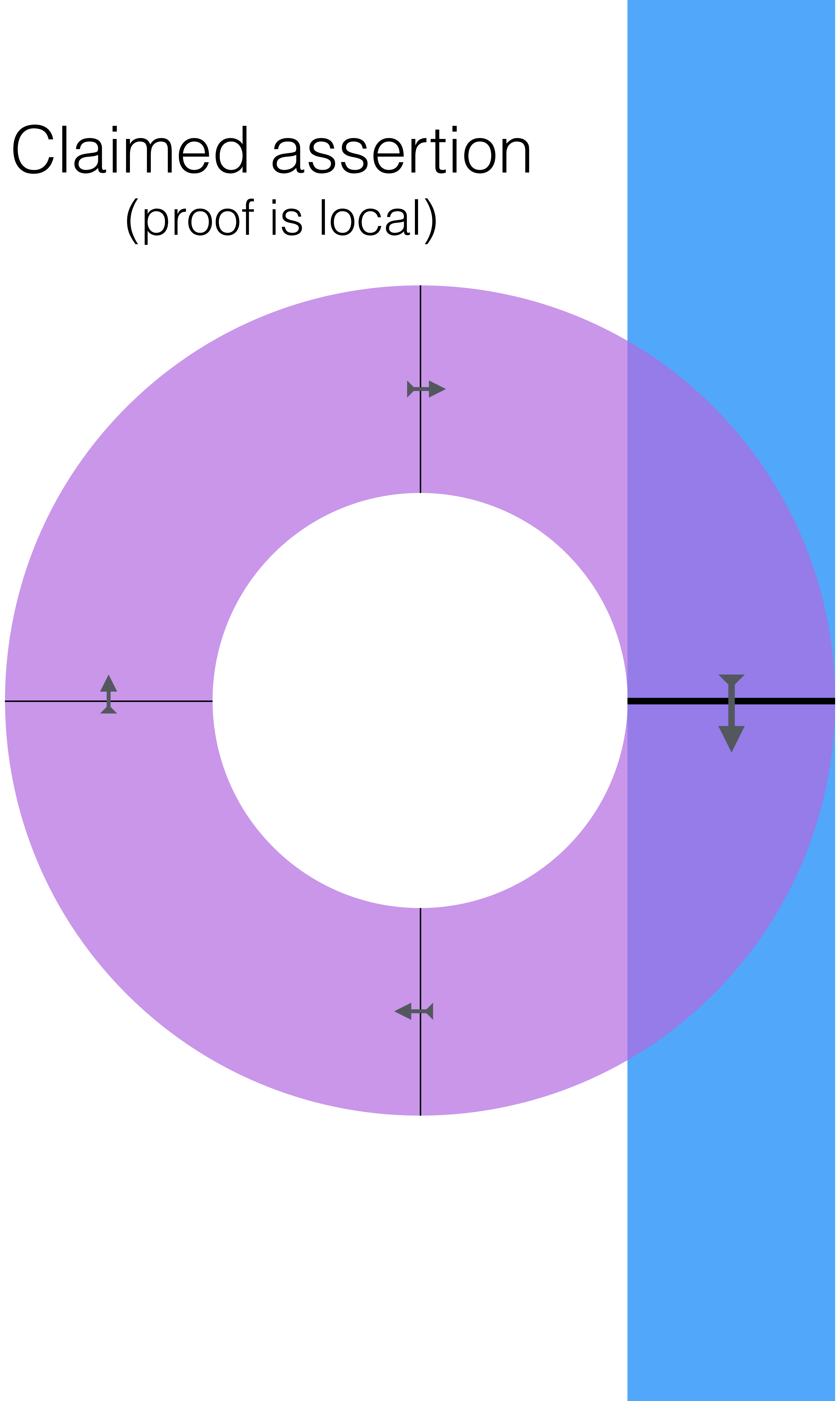
Successful assertions are repeatable, and have no meaningful effect.

Assertion edge

An assertion describes its edge, in dimensions of space and possibility.

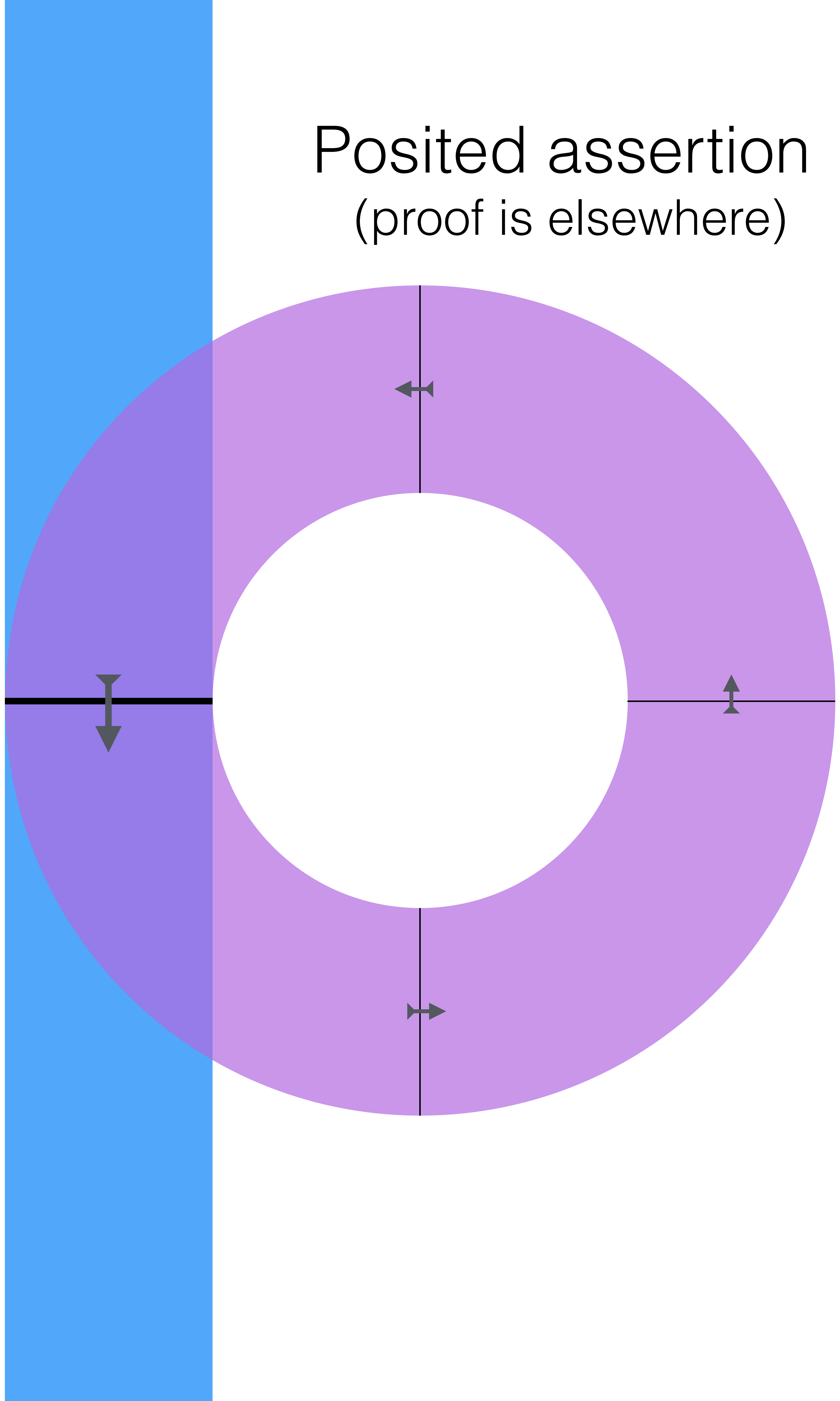
Claimed assertion

(proof is local)



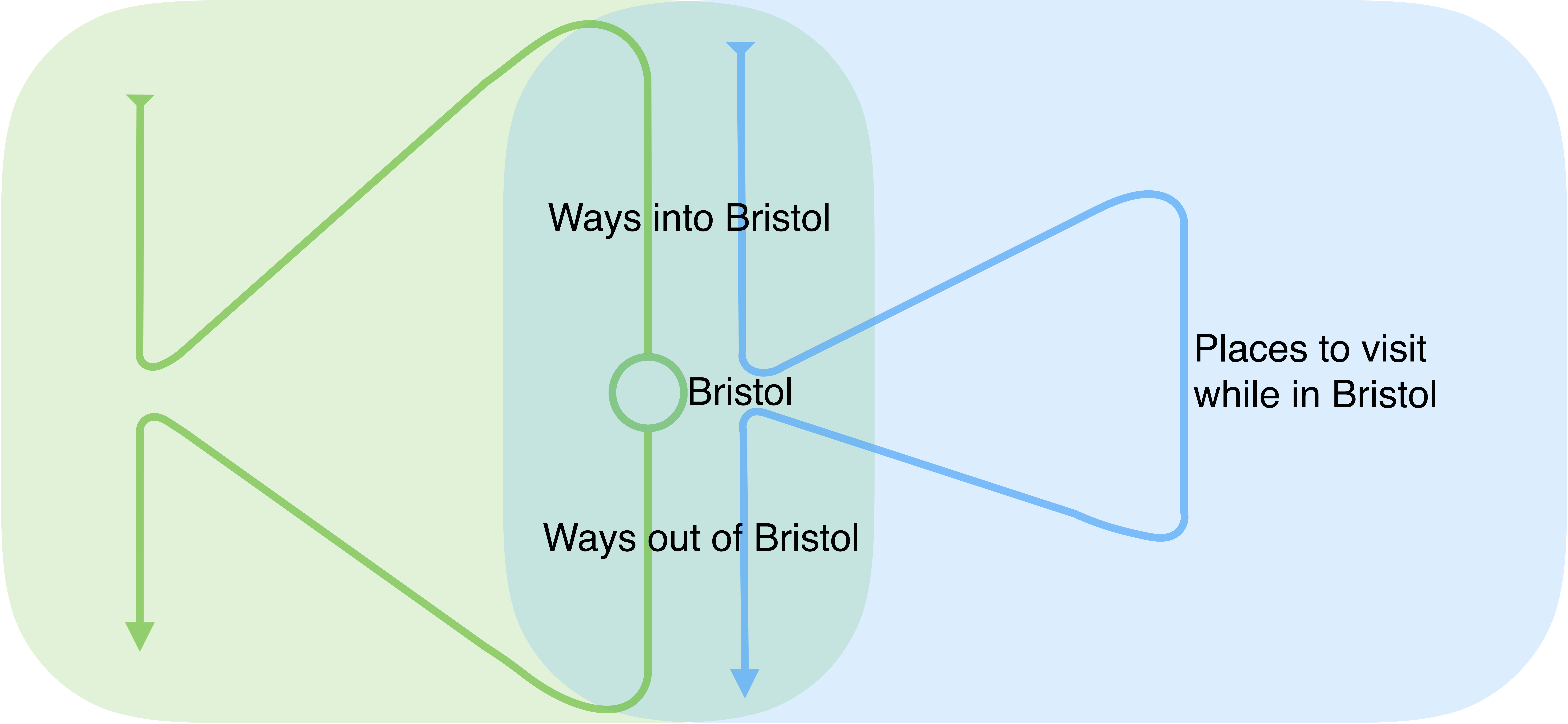
Posited assertion

(proof is elsewhere)



Map of the UK

Map of Bristol



Ways into Bristol

Bristol

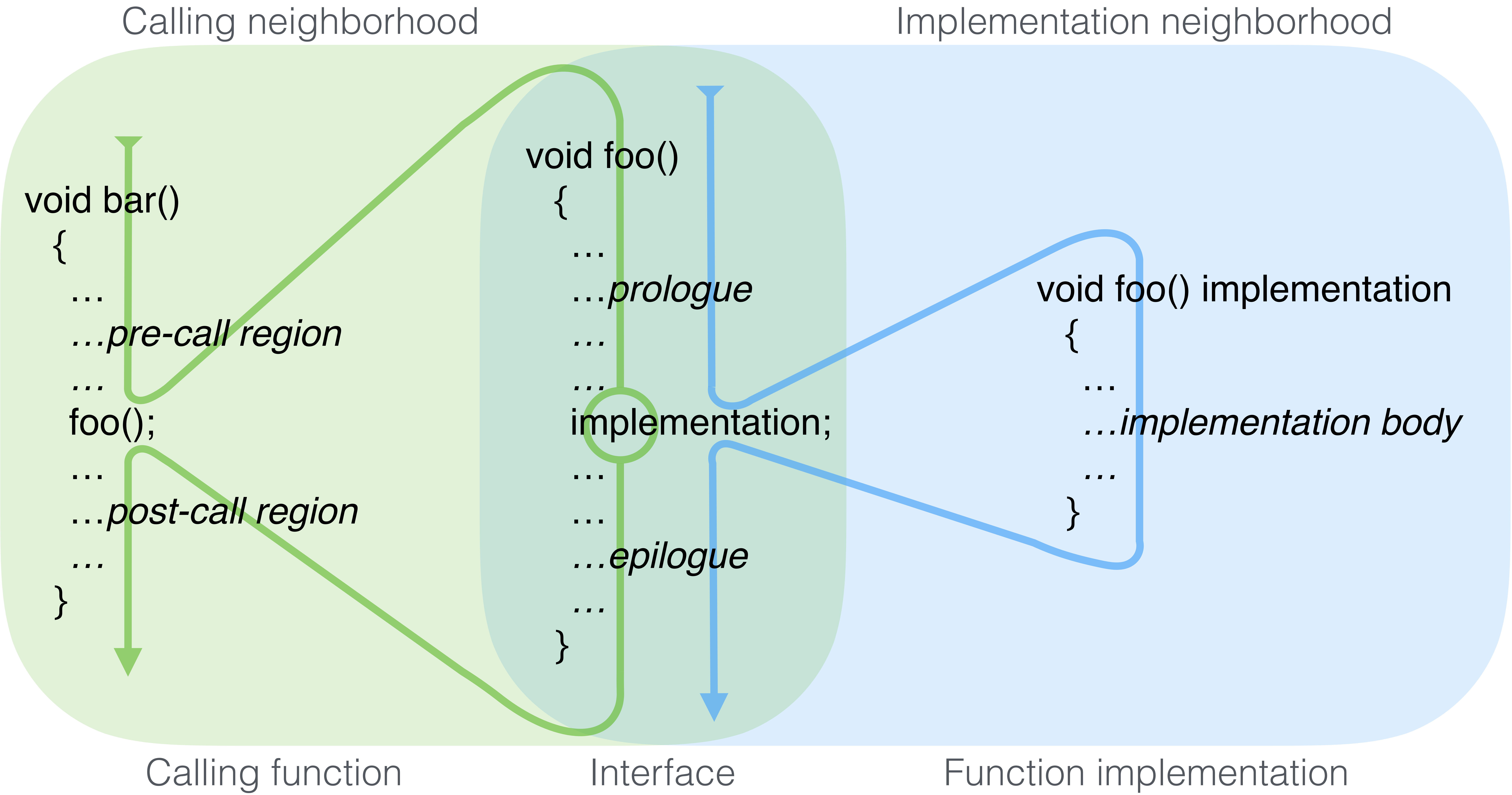
Places to visit
while in Bristol

Ways out of Bristol

Far from Bristol

Outskirts of Bristol

Inside Bristol



Calling neighborhood

Implementation neighborhood

void bar()
{
…
…*pre-call region*
…
foo();
…
…*post-call region*
…
}

void foo()
{
…
…*prologue*
…
…
implementation;
…
…
…*epilogue*
…
}

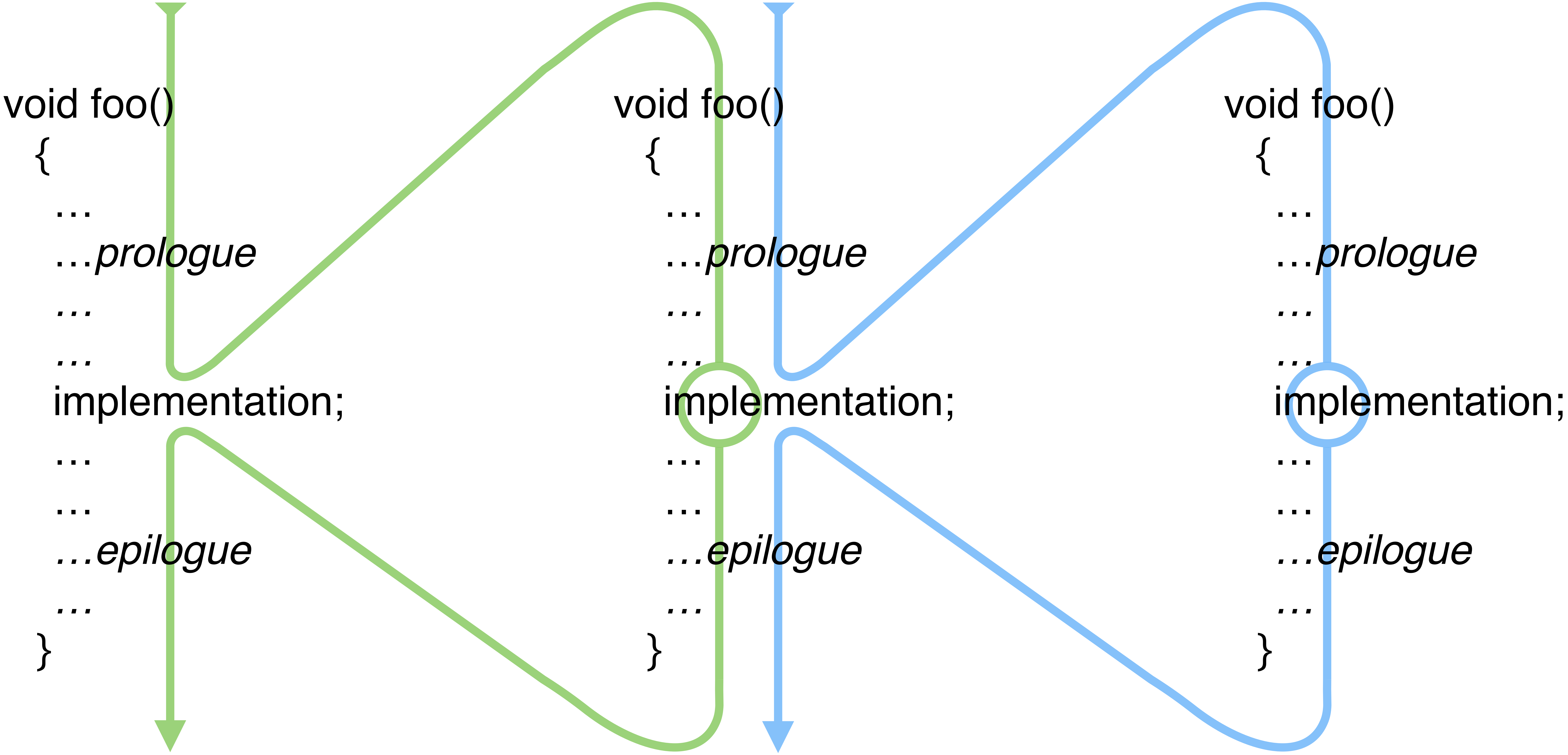
void foo() implementation
{
…
…*implementation body*
…
}

Calling function

Interface

Function implementation

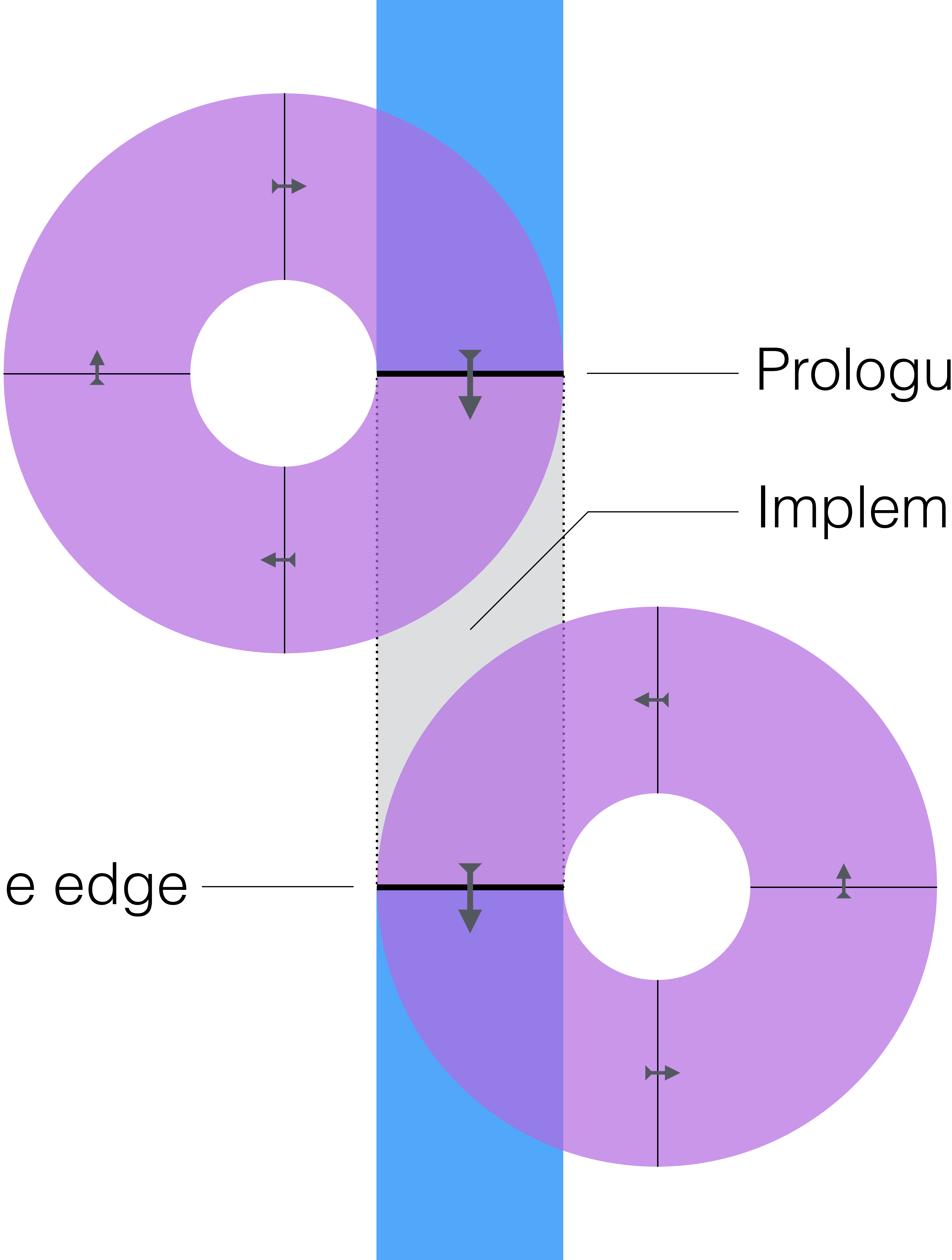
A function interface is an experiment in two parts, nestable within itself.



```
void foo()
 {
  …
  …prologue
  …
  …
  implementation;
  …
  …
  …epilogue
  …
 }
```

```
void foo()
 {
  …
  …prologue
  …
  …
  implementation;
  …
  …
  …epilogue
  …
 }
```

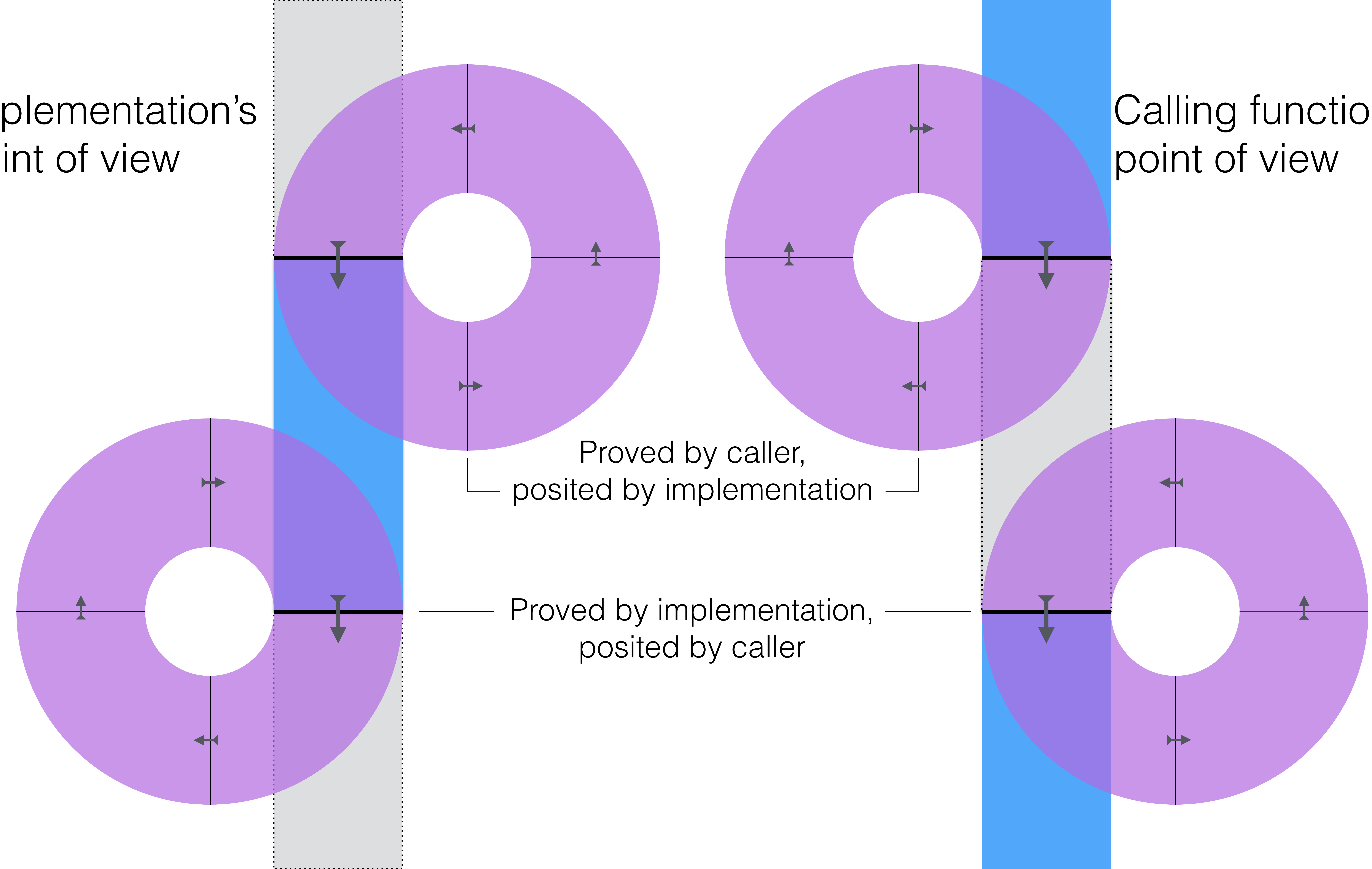
```
void foo()
 {
  …
  …prologue
  …
  …
  implementation;
  …
  …
  …epilogue
  …
 }
```



Prologue edge

Implementation gap

Epilogue edge



Implementation's point of view

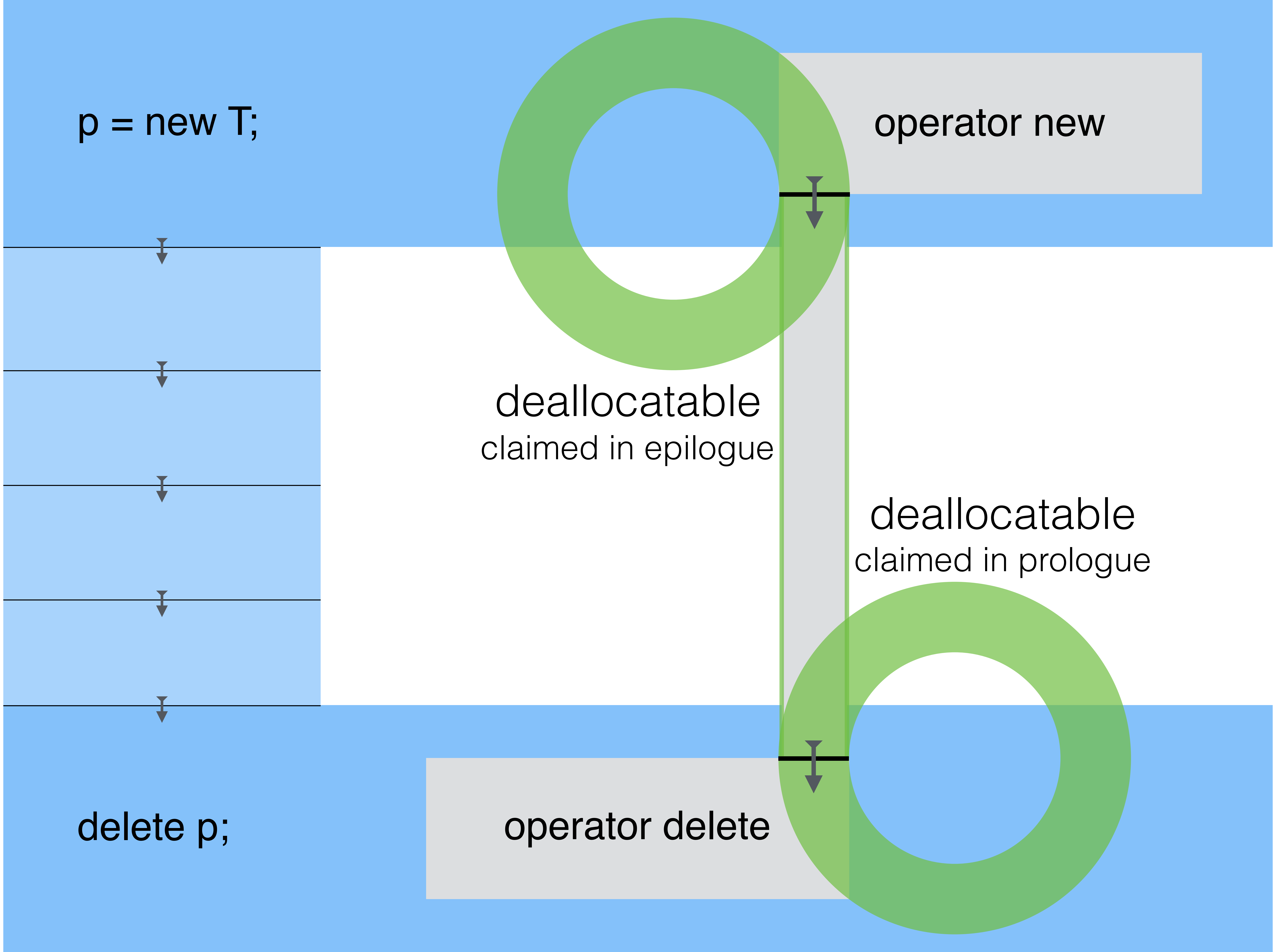
Calling function's point of view

Proved by caller, posited by implementation

Proved by implementation, posited by caller

```
void *operator new( size_t s )
  {
   …
   implementation;
   …
   claim deallocatable( result, s );
   …
  }
```

```
void operator delete( void *p, size_t s )
  {
   …
   claim deallocatable( p, s );
   …
   implementation;
   …
  }
```



p = new T;

operator new

deallocatable

claimed in epilogue

deallocatable

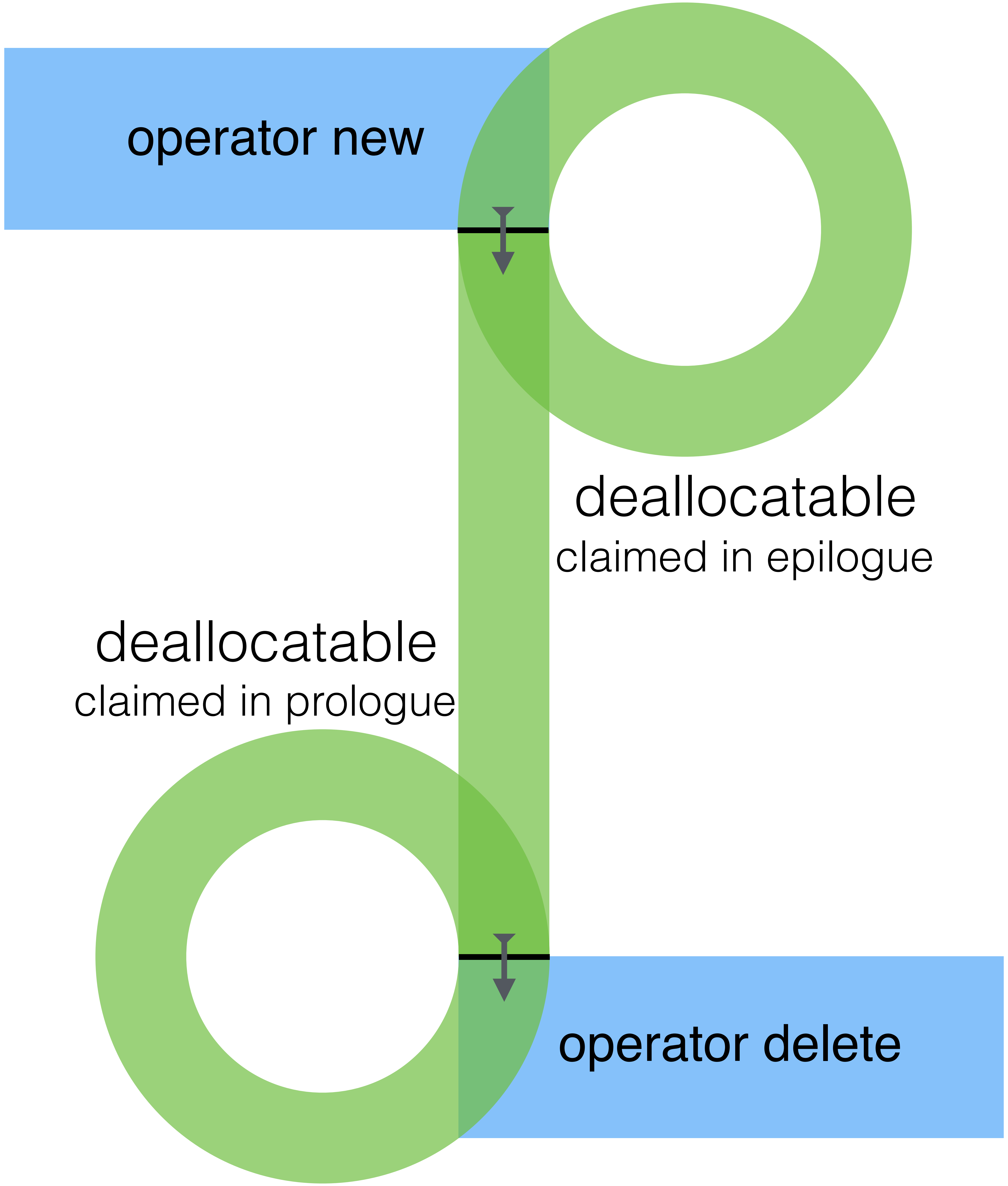
claimed in prologue

delete p;

operator delete

Heap implementation neighborhood

(partial)



operator new

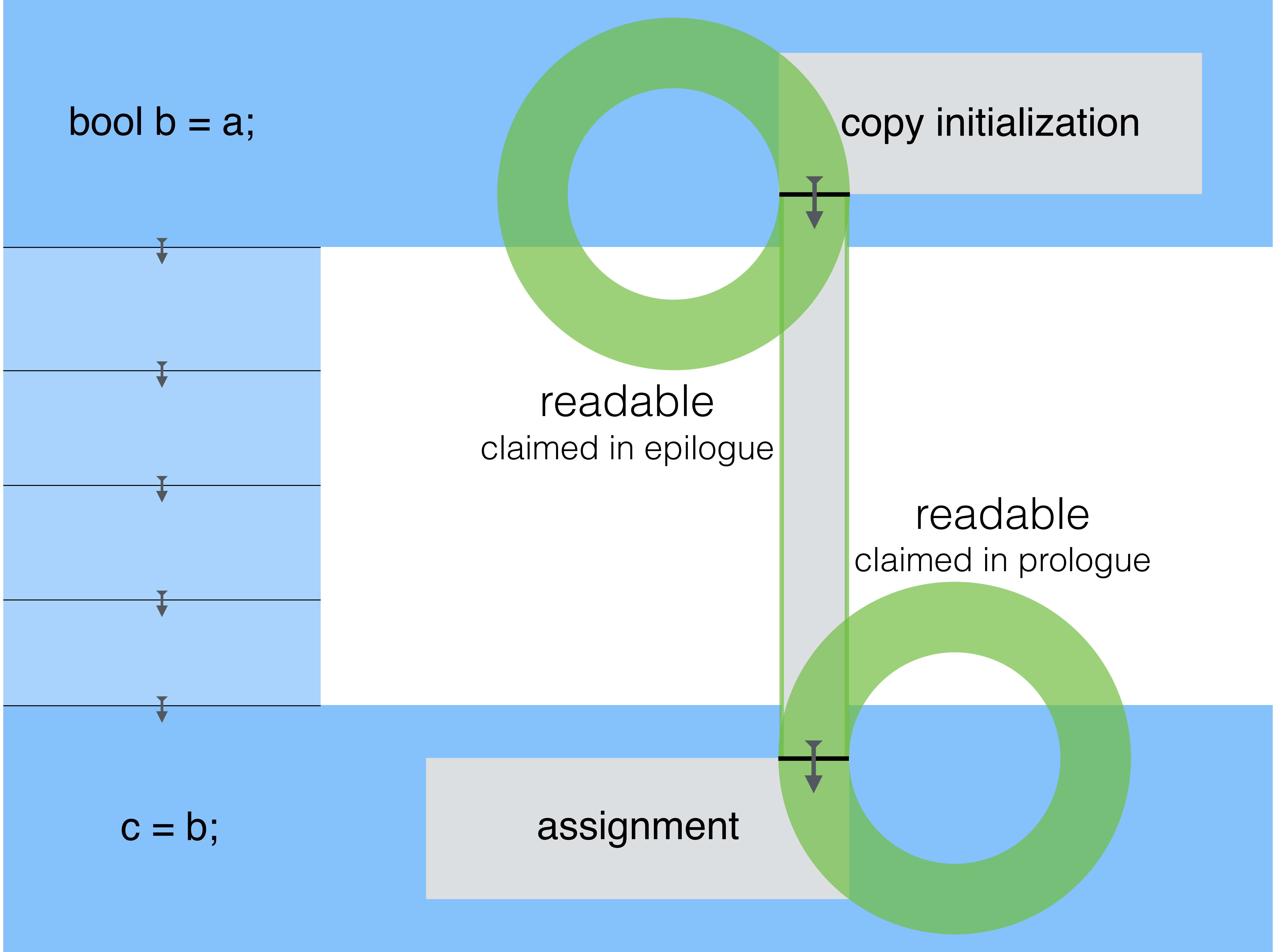
deallocatable

claimed in epilogue

deallocatable

claimed in prologue

operator delete



`bool b = a;`

copy initialization

readable

claimed in epilogue

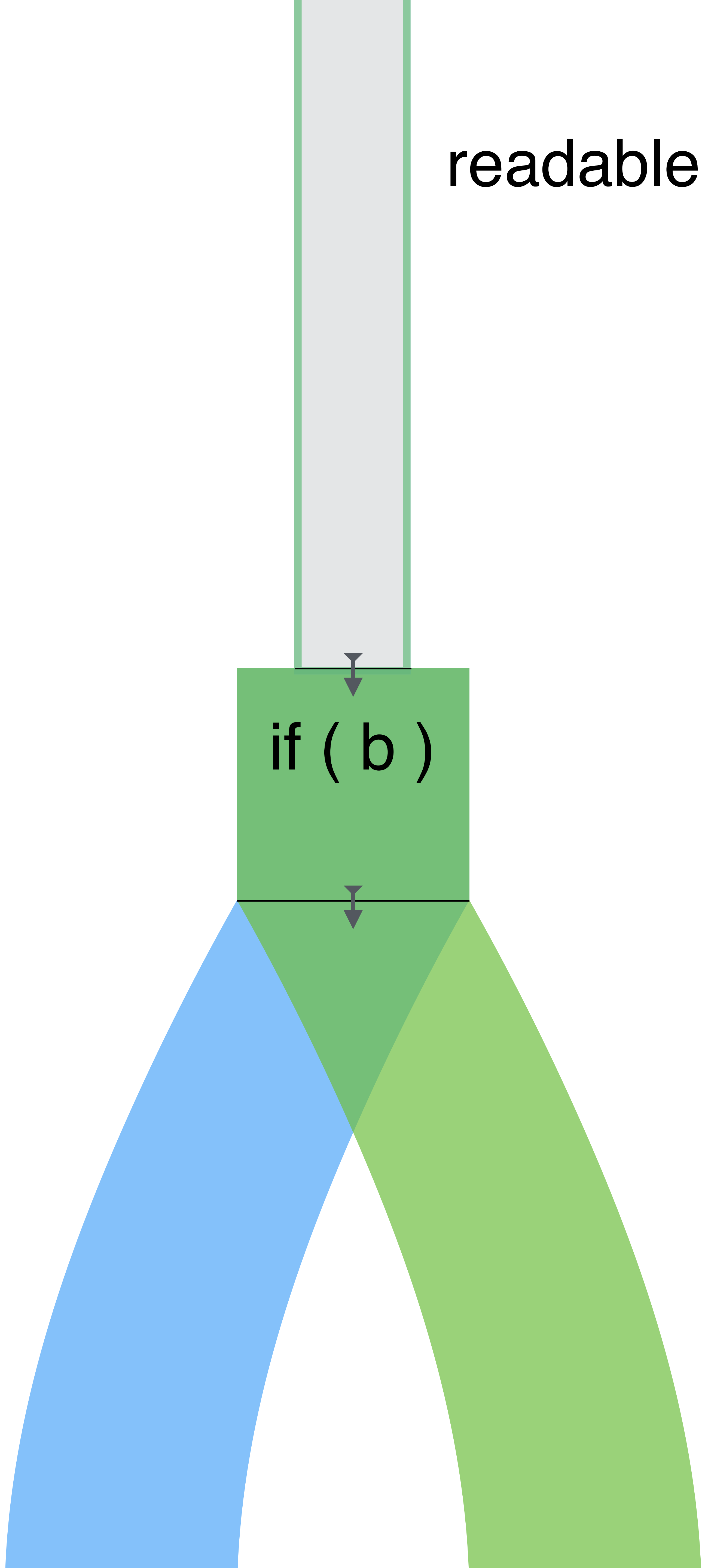
readable

claimed in prologue

`c = b;`

assignment

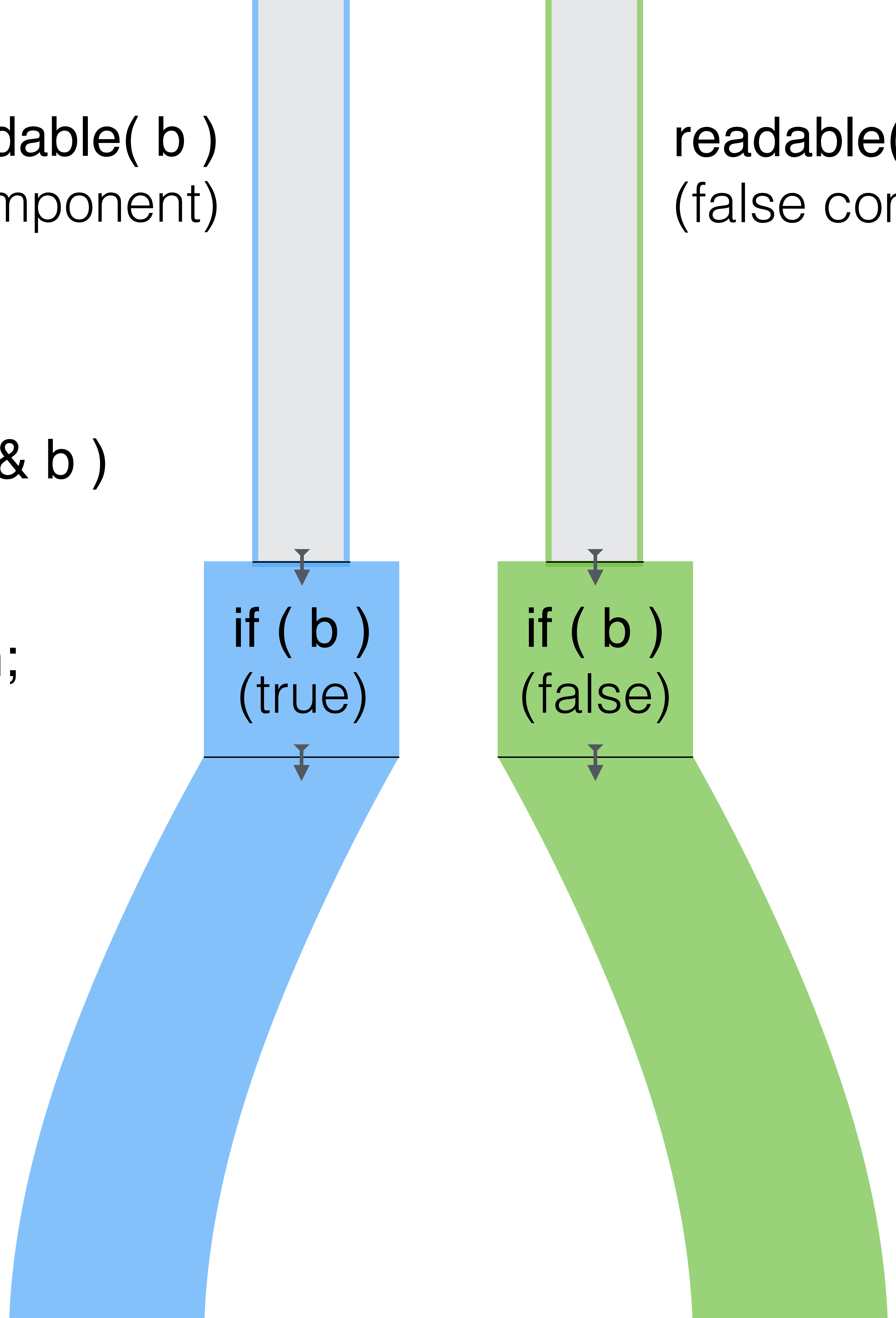
```
void readable( const bool& b )
  {
    claim addressable( b );
    require implementation;
  }
```



readable(b)

if (b)

```
void readable( const bool& b )
  {
   claim addressable( b );
   require implementation;
  }
```



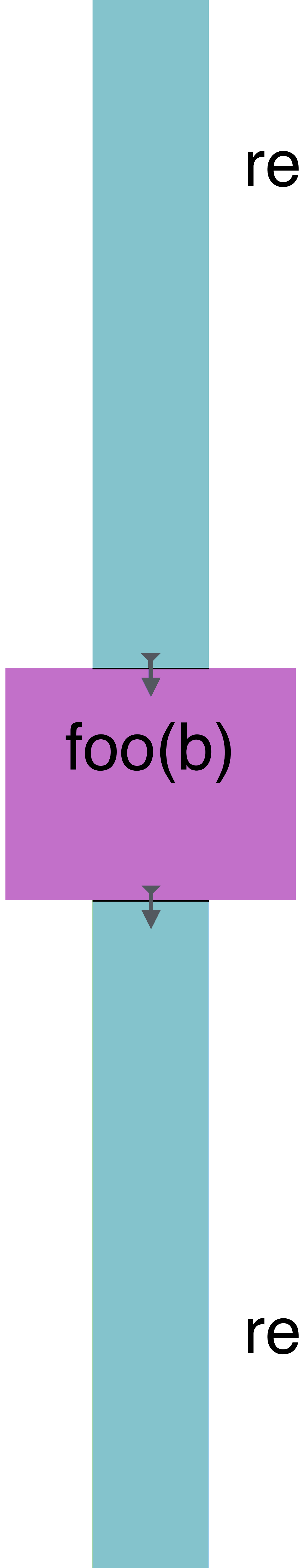
readable(b)
(true component)

readable(b)
(false component)

if (b)
(true)

if (b)
(false)

```
inline void usable( const bool& b )
  {
   require readable( b );
  }
```

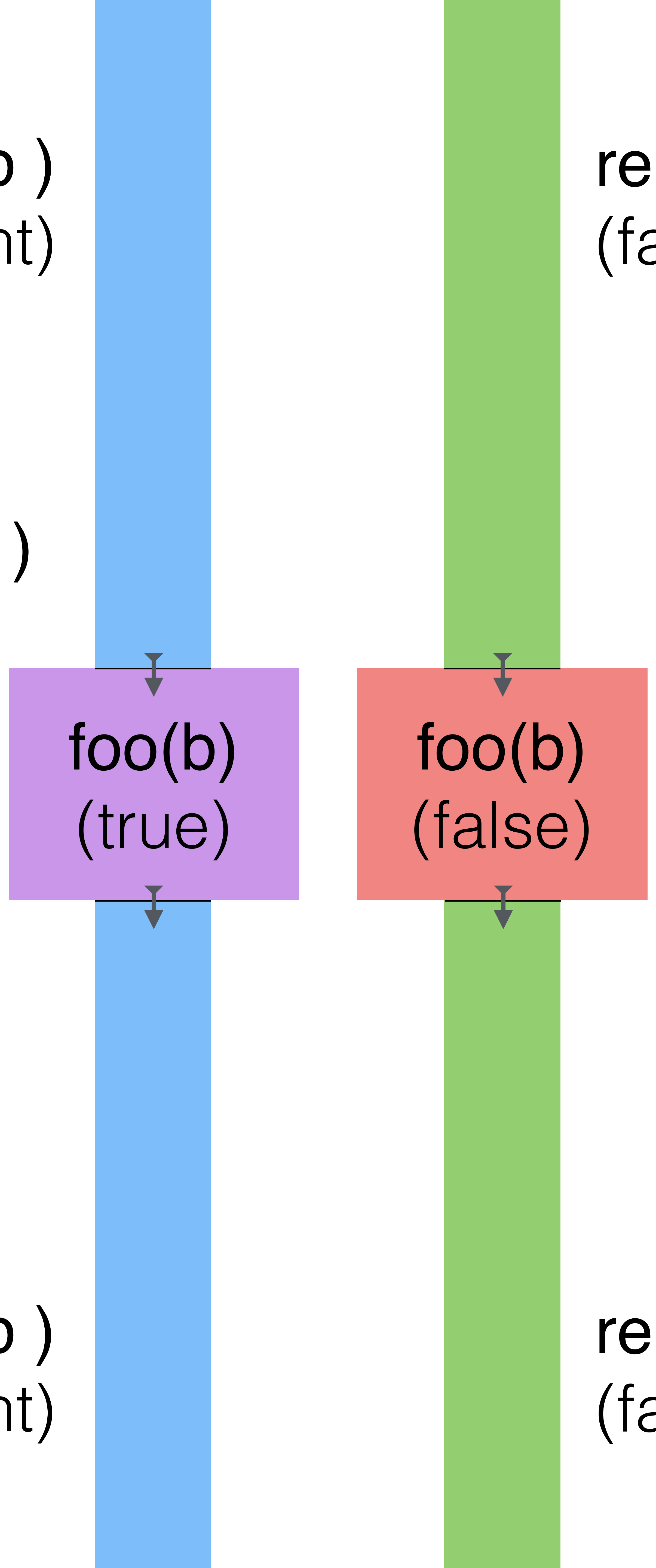


readable(b)

foo(b)

readable(b)

```
void foo( const bool& b )
  {
   claim usable( b );
   implementation;
   claim usable( b );
  }
```



readable(b)
(true component)

readable(b)
(false component)

```
inline void usable( const bool& b )
  {
   require readable( b );
  }
```

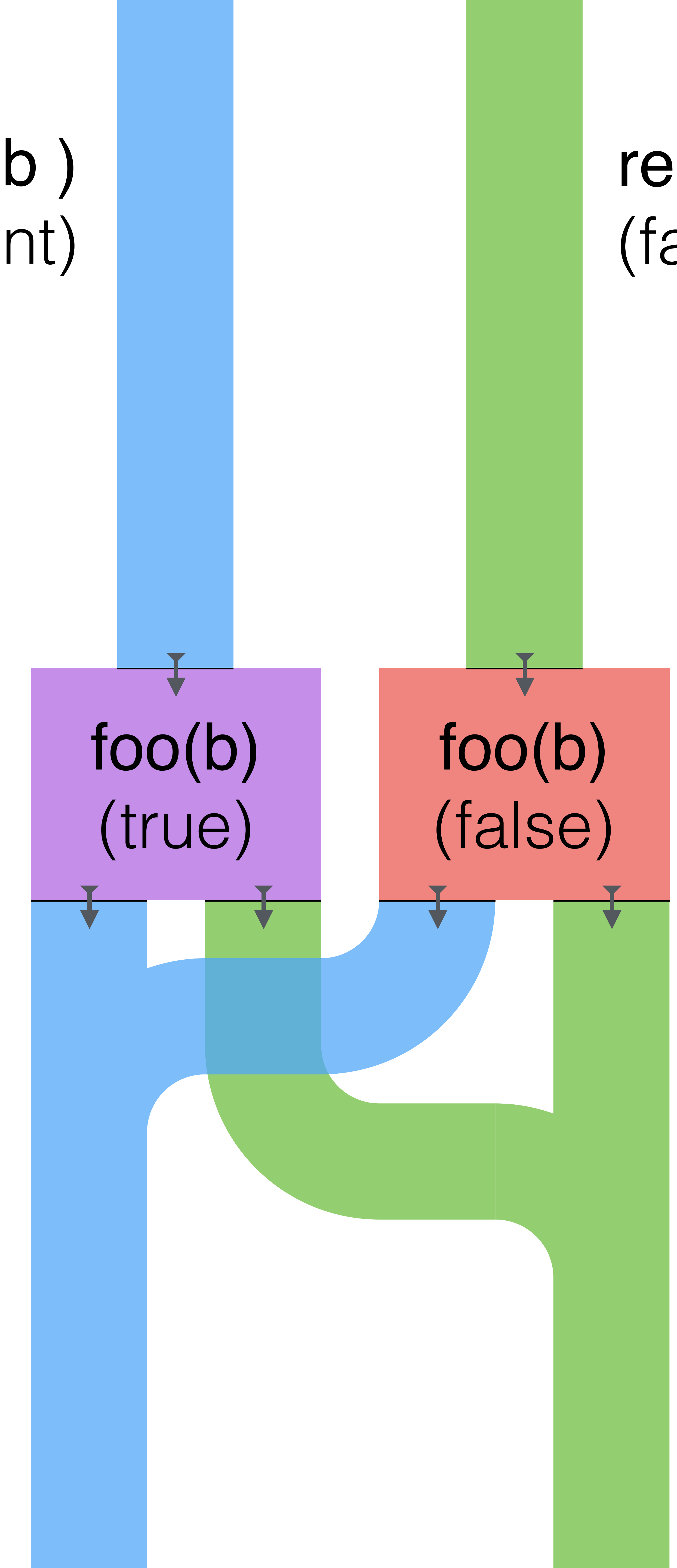
foo(b)
(true)

foo(b)
(false)

```
void foo( const bool& b )
  {
   claim usable( b );
   implementation;
   claim usable( b );
  }
```

readable(b)
(true component)

readable(b)
(false component)



readable(b)
(true component)

readable(b)
(false component)

```
inline void usable( bool& b )
   {
    require readable( b );
    require writable( b );
   }
```

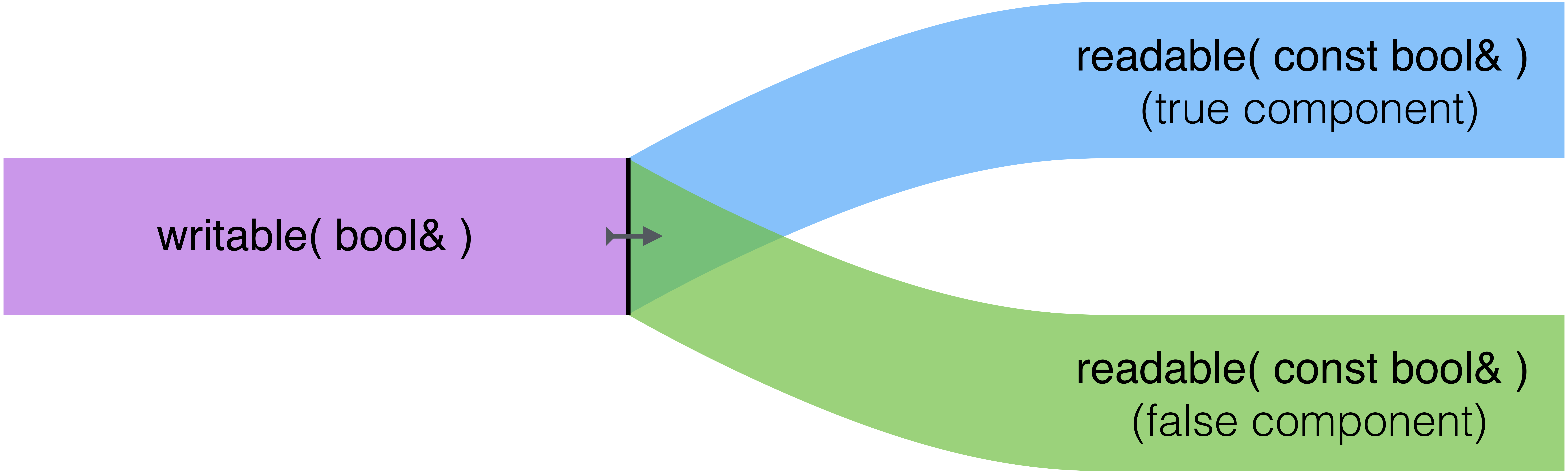
foo(b)
(true)

foo(b)
(false)

```
void foo( bool& b )
   {
    claim usable( b );
    require implementation;
    claim usable( b );
   }
```

readable(b)
(true component)

readable(b)
(false component)



writable(bool&)

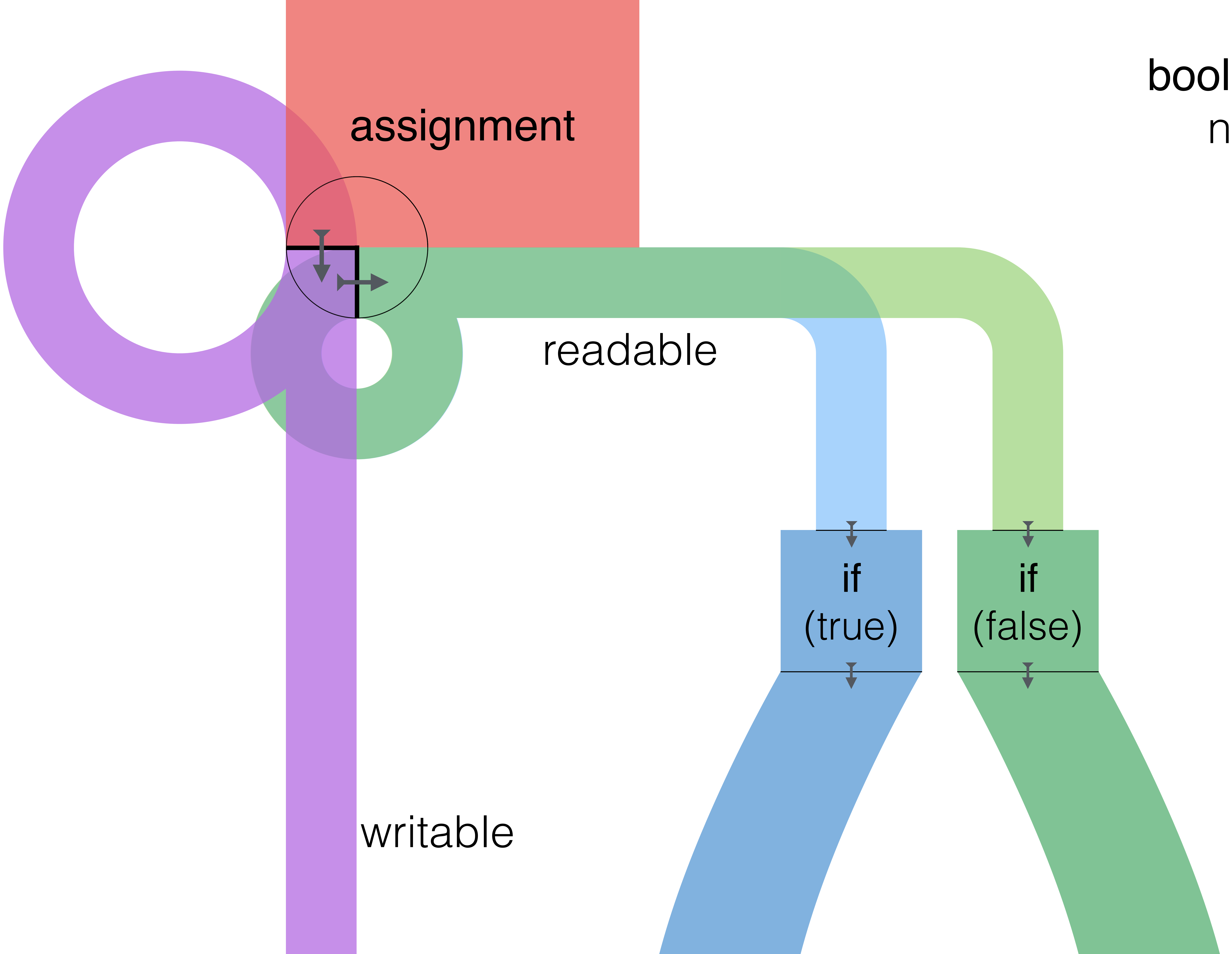
readable(const bool&)
(true component)

readable(const bool&)
(false component)

```
void writable( bool& b )
  {
   claim addressable( b );
   require implementation;
   require readable( b );
  }
```

```
void readable( const bool& b )
  {
   claim addressable( b );
   require implementation;
  }
```

bool implementation
neighborhood
(partial)



All of these neighborhoods are composed of *nothing but edges*.

It’s edges all the way down.

And it’s edges all the way up.

Questions?