Locally Atomic Capabilities

and How to Count Them

Lisa Lippincott

Why don’t we routinely have computers check that our reasoning about our programs is mathematically sound?

It’s not that our reasoning is usually unsound.

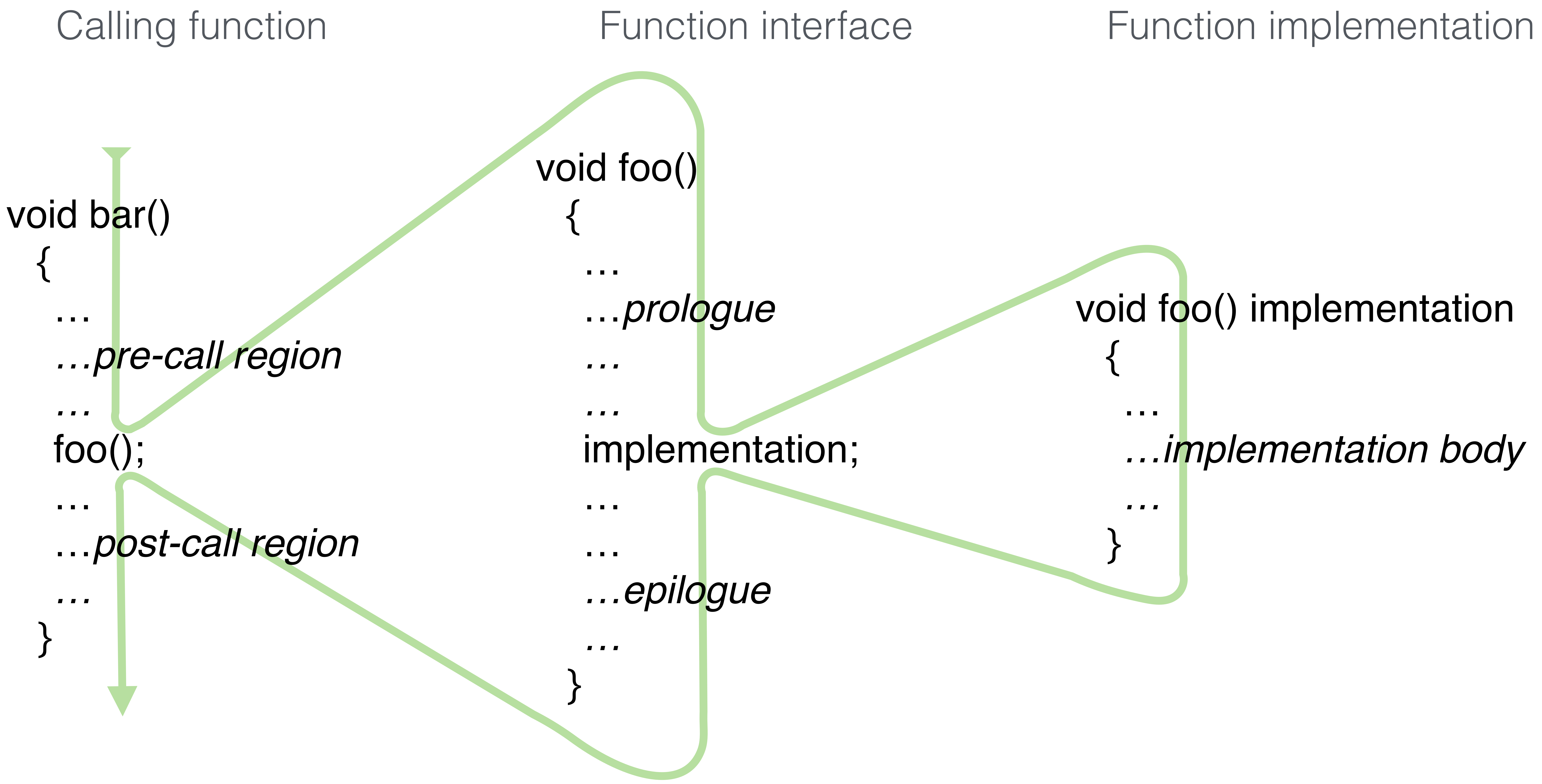
It’s not that we can’t write formally enough for a computer.

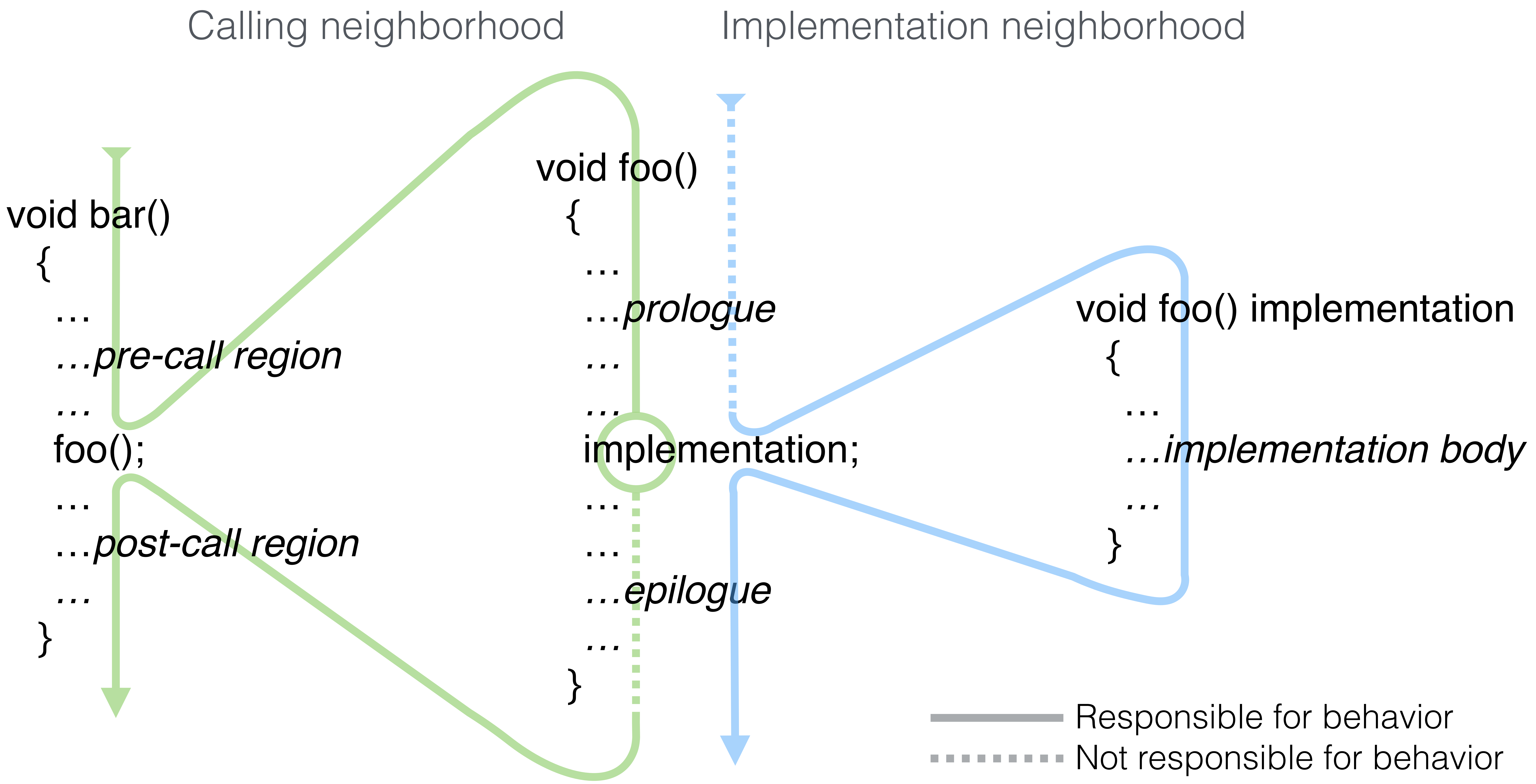
It’s that the way we write proofs is contrary to the way we write software, so it’s difficult to write both at once.

How can we make it easier to write programs with proofs?

Find a way to write programs more like we write mathematics?
(Other people are trying this.)

Or find a way to write mathematics more like we write programs?
(I’m trying this.)


Calling function
Function interface
Function implementation
void bar()
{
…
…pre-call region
…
foo();
…
…post-call region
…
}
void foo()
{
…
…prologue
…
…
implementation;
…
…
…epilogue
…
}
void foo() implementation
{
…
…implementation body
…
}


Calling neighborhood
Implementation neighborhood
void bar()
{
…
…pre-call region
…
foo();
…
…post-call region
…
}
void foo()
{
…
…prologue
…
…
implementation;
…
…
…epilogue
…
}
void foo() implementation
{
…
…implementation body
…
}
Responsible for behavior
Not responsible for behavior

✅

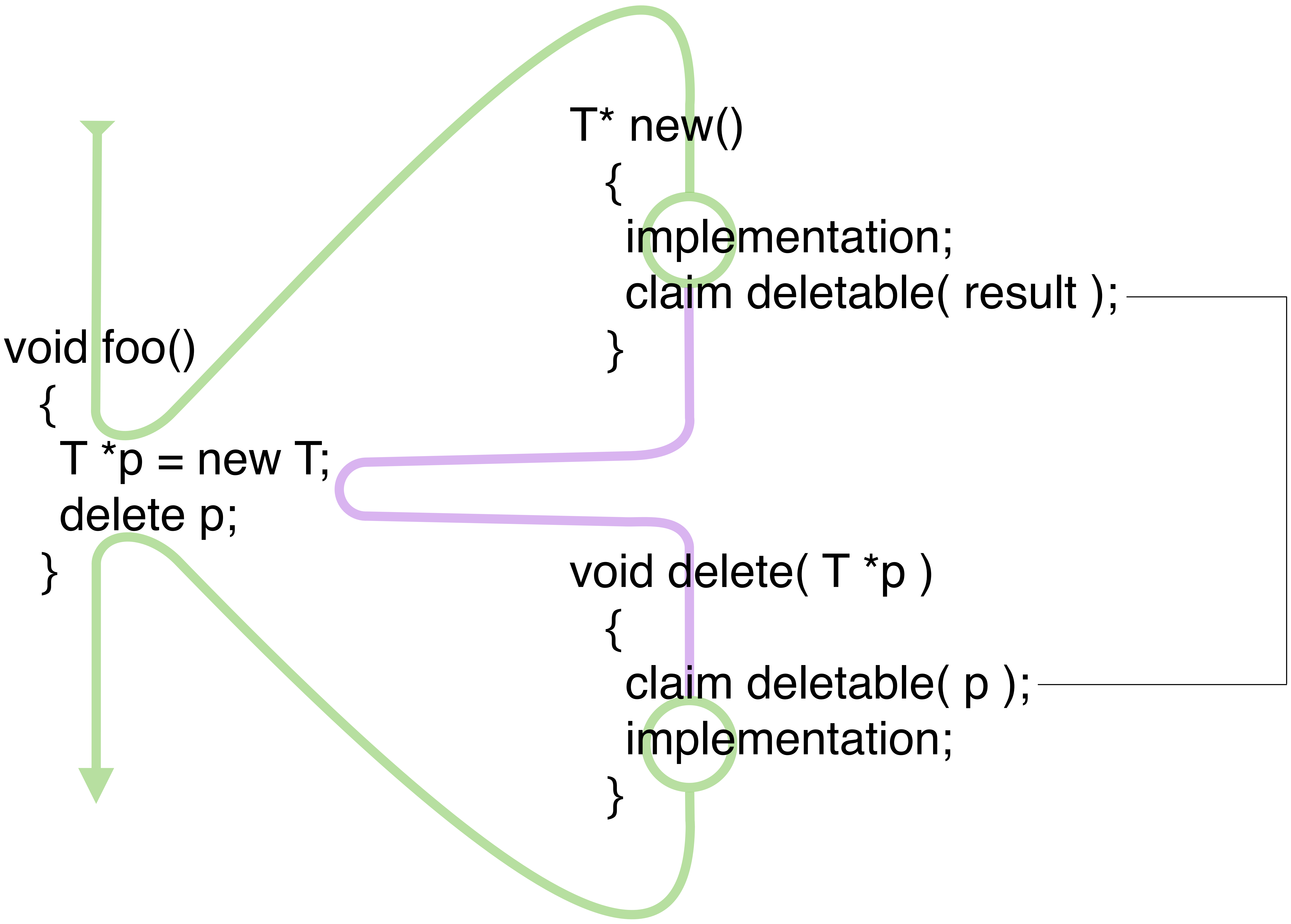
```
void foo()
  {
    T *p = new T;
    delete p;
  }
```

❌

```
void bar()
  {
    T t;
🔥  delete &t;
  }
```

❌

```
void baz()
  {
    T *p = new T;
    delete p;
🔥  delete p;
  }
```



deletable connects
the epilogue of **new** to
the prologue of **delete.**

```
void foo()
  {
   T *p = new T;
   delete p;
  }
```

```
inline T* new()
  {
   void *p = operator new( sizeof( T ) );
   return new ( p ) T;
  }
```

```
inline void delete( T *p )
  {
   p->~T();
   operator delete( p, sizeof(T) );
  }
```

deallocatability

destructibility

✢ Simplified by ignoring exceptions, polymorphic types, and null pointers.

```
inline void deletable( T *p )
  {
   require destructible( *p );
   require deallocatable( p, sizeof(T) );
  }
```

“**require**” introduces a partial assertion: an assertable capability without a *sense*.

“**claim deletable(t)**” claims both destructibility and deallocatability.

“**posit deletable(t)**” posits both destructibility and deallocatability.

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

```
void foo( size_t s )
  {
   void *p = operator new( s );
   operator delete( p, s );
  }
```

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

```
void deallocatable( void *p, size_t s )
  {
   require implementation;
  }
```

"**require implementation**" indicates a capability implemented elsewhere, in the function implementation of **deallocatable**.

To the caller, this is a *locally atomic capability*.

$$2\,H_2 + O_2 \longrightarrow 2\,H_2O$$

Atoms are conserved. Atoms are neither created nor destroyed.

Atoms may be rearranged, but they don’t change.

Atoms of the same kind are interchangeable.

$$^{232}_{90}\mathrm{Th} \longrightarrow {}^{208}_{82}\mathrm{Pb} + 6\,{}^{4}_{2}\mathrm{He} + 4\,\bar{\nu}_{\mathrm{e}}$$

Local atoms are locally conserved. Local atoms aren't created or destroyed locally.
(But local atoms may be created or destroyed in other neighborhoods.)

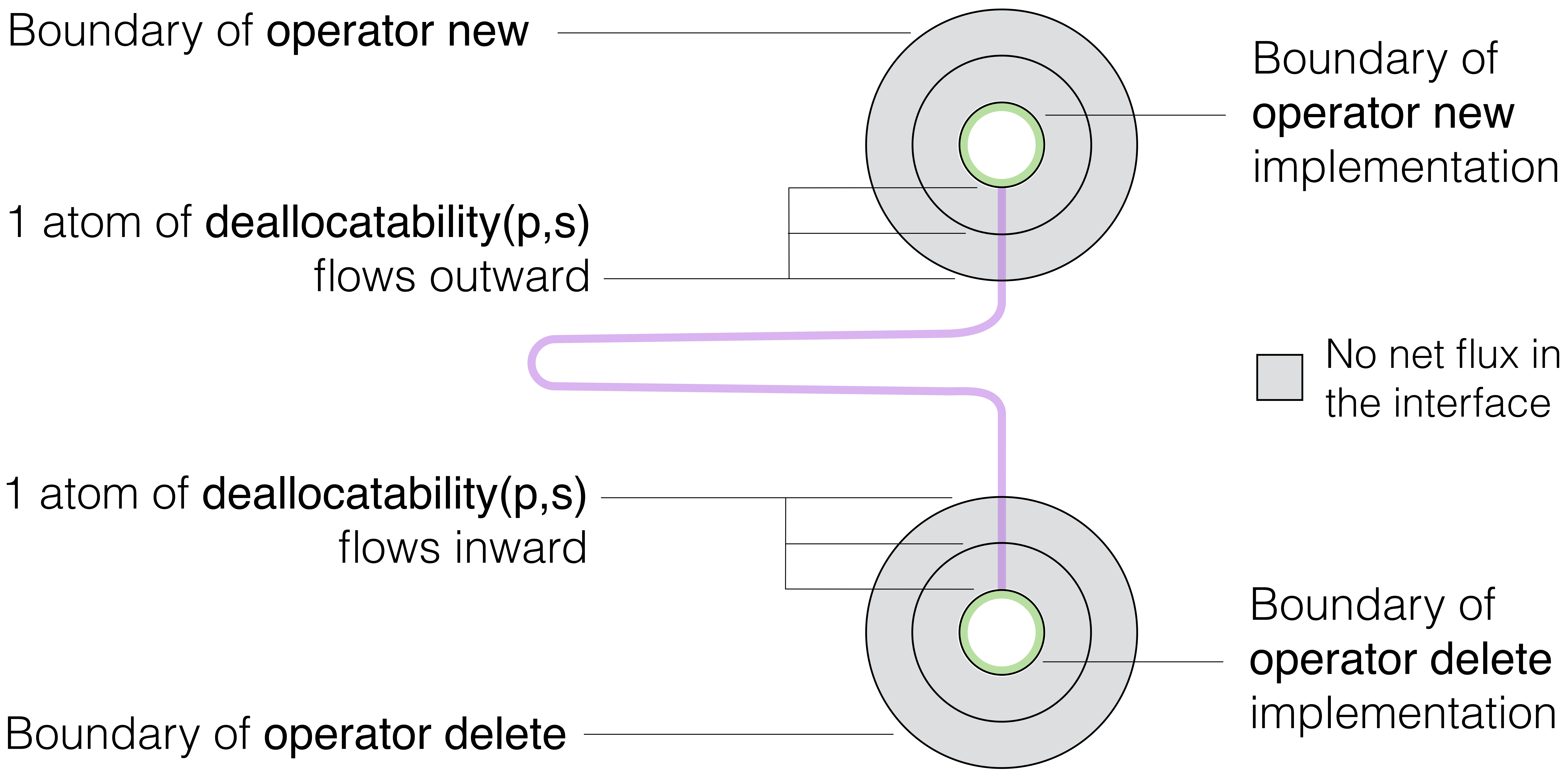
Local atoms don't change in the local neighborhood.
(But they may change in other neighborhoods.)

Local atoms of the same kind are interchangeable in the local neighborhood.
(But they may have differences that matter in other neighborhoods.)

```
void foo( size_t s )
  {
   void *p = operator new( s );
   operator delete( p, s );
  }
```

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

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



Boundary of **operator new**
Boundary of **operator new** implementation
1 atom of **deallocatability(p,s)** flows outward
No net flux in the interface
1 atom of **deallocatability(p,s)** flows inward
Boundary of **operator delete** implementation
Boundary of **operator delete**

✅

	Change in deallocatability	Total deallocatability
void foo()		
{		
T *p = new T;	+1	1
delete p;	-1	0
}		

❌

	Change in deallocatability	Total deallocatability
void bar()		
{		
T t;	0	0
🔥 delete &t;	-1	-1
}		

❌

	Change in deallocatability	Total deallocatability
void baz()		
{		
T *p = new T;	+1	1
delete p;	-1	0
🔥 delete p;	-1	-1
}		

❌

	Δ deallocatability	Total deallocatability
void foo()		
{		
T *p = new T;	+1	1
T *q = new T;	+1	2
delete q;	-1	1
🔥 delete q;	-1	0
}		

```
void foo()
  {
    T *p = new T;
    T *q = new T;
    delete q;
    delete q;
  }
```

	Δ deallocatable(p, sizeof(T))	Total deallocatable(p, sizeof(T))	Δ deallocatable(q, sizeof(T))	Total deallocatable(q, sizeof(T))
T *p = new T;	+1	1	0	0
T *q = new T;	0	1	+1	1
delete q;	0	1	-1	0
delete q;	0	1	-1	-1

```
void deallocatable( void *p, size_t s )
  {
   require implementation;
  }
```

```
void deallocatable( void *p, size_t s )
  {
  claim proper( p );
  claim proper( s );

  require implementation;

  claim proper( p );
  claim proper( s );
  }
```

```
inline void proper( void *& p )
  {
   require readable( p );
   require writable( p );
  }
```

```
inline void proper( size_t& s )
  {
   require readable( s );
   require writable( s );
  }
```

```
void deallocatable( void *const p, const size_t s )
  {
   claim proper( p );
   claim proper( s );

   require implementation;

   claim proper( p );
   claim proper( s );
  }
```

```
inline void proper( void *const& p )
  {
   require readable( p );
  }
```

```
inline void proper( const size_t& s )
  {
   require readable( s );
  }
```

```
void readable( [addressable] const volatile byte& vb )
  {
   require implementation;
  }
```

```
void readable( [addressable] const byte& b )
  {
   volatile auto& vb = b;
   require readable( vb );

   require implementation;
  }
```

```
void readable( [addressable] const volatile byte& vb )
  {
   claim addressable( vb );
   require implementation;
   claim addressable( vb );
  }
```

```
void readable( [addressable] const byte& b )
  {
   volatile auto& vb = b;
   require readable( vb );

   claim addressable( b );
   require implementation;

   claim addressable( b );
  }
```

```
void readable( [addressable] const volatile byte& vb )
  {
   require implementation;
  }
```

```
void writable( [addressable] volatile byte& vb )
  {
   require implementation;

   require readable( vb );
  }
```

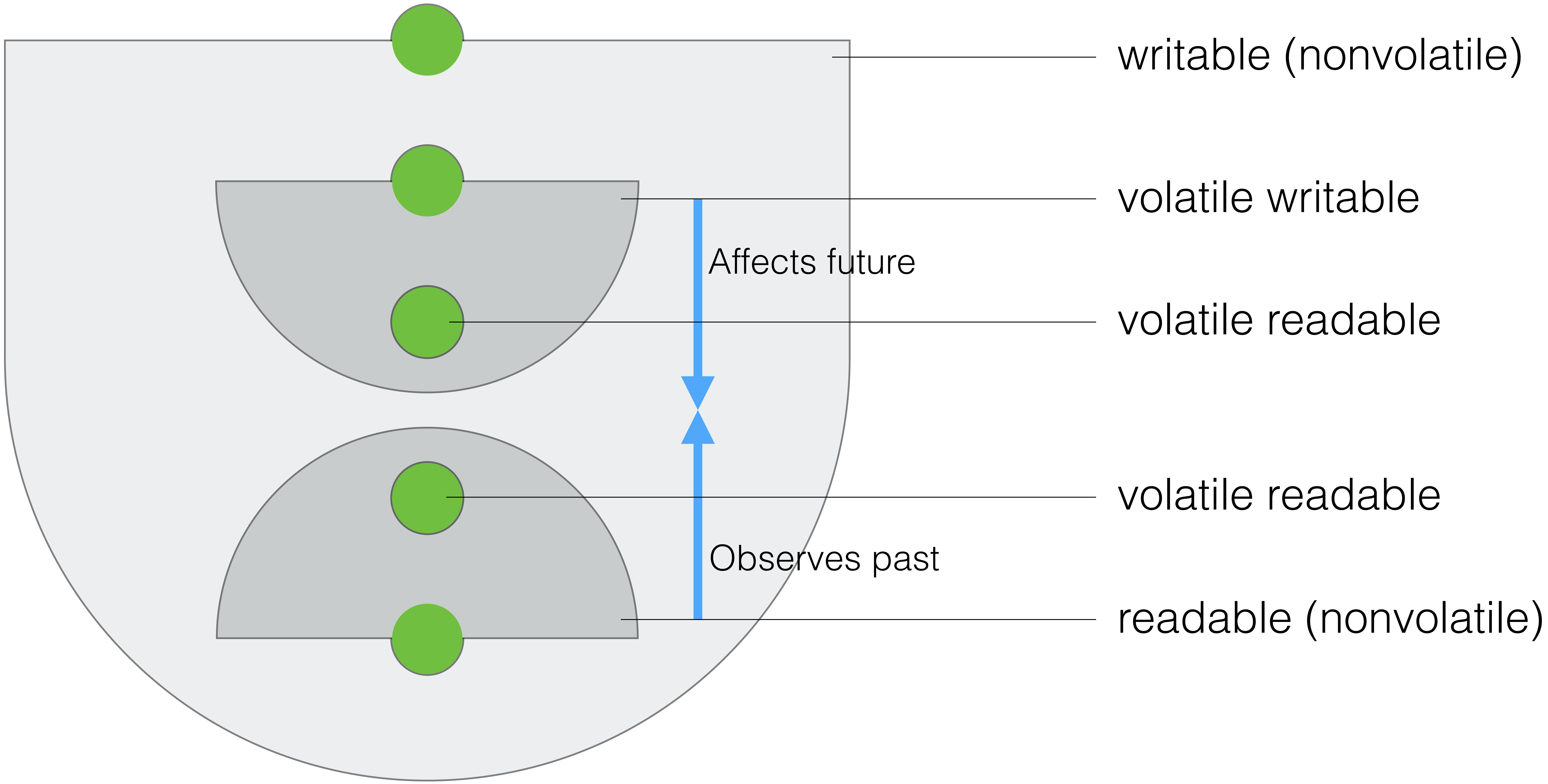
```
void readable( [addressable] const byte& b )
  {
   volatile auto& vb = b;
   require readable( vb );

   require implementation;
  }
```

```
void writable( [addressable] byte& b )
  {
   require implementation;

   volatile auto& vb = b;
   require writable( vb );

   require readable( b );
  }
```


writable (nonvolatile)
volatile writable
Affects future
volatile readable
volatile readable
Observes past
readable (nonvolatile)

```
byte::byte( const volatile byte& from ) [indiscernable]
  {
   implementation;
  }
```

```
byte::byte( const byte& from )
  {
   implementation;
   claim substitutable( *this, from );
  }
```

```
void write( [writable] volatile byte& to,
            const byte& from )
  {
   implementation;
   claim readable( to );
  }
```

```
void write( [writable] byte& to,
            const byte& from )
  {
   implementation;
   claim readable( to );
   claim substitutable( to, from );
  }
```

~~Substitutability is local knowledge of equality.~~

Equality is the omniscient expectation of substitutability in every neighborhood, if only everyone knew.

```
bool operator==( const byte& b0, const byte& b1 )
  {
   implementation;
   if ( result )
      claim substitutable( b0, b1 );
  }
```

```
void equality_is_reflexive( const byte& b )
  {
   claim implementation;
   claim b == b;
  }
```

When do separate assertions assert the same locally atomic capability?

When do separate function calls produce substitutable results?

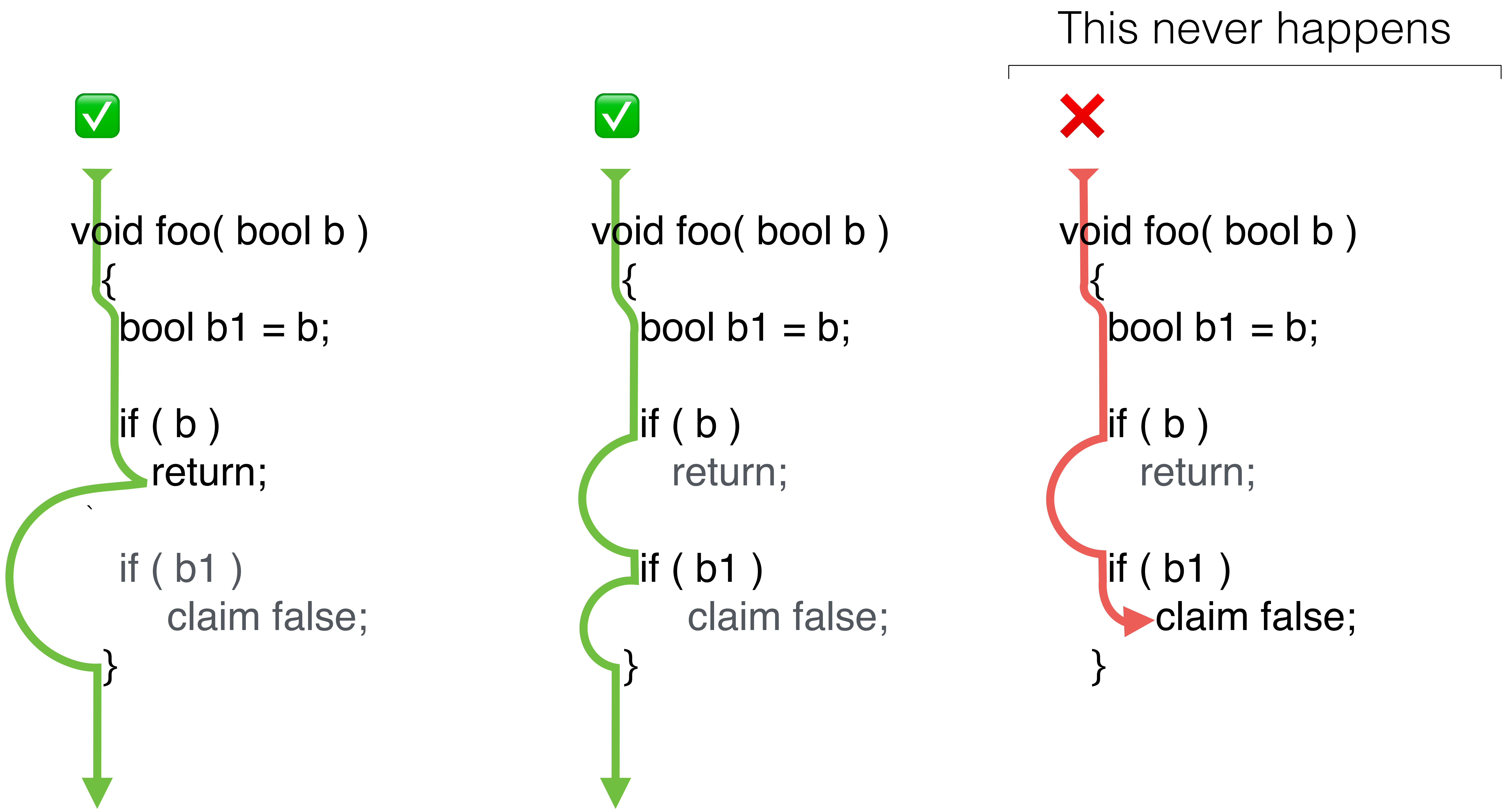
More generally, when are locally atomic events interchangeable?

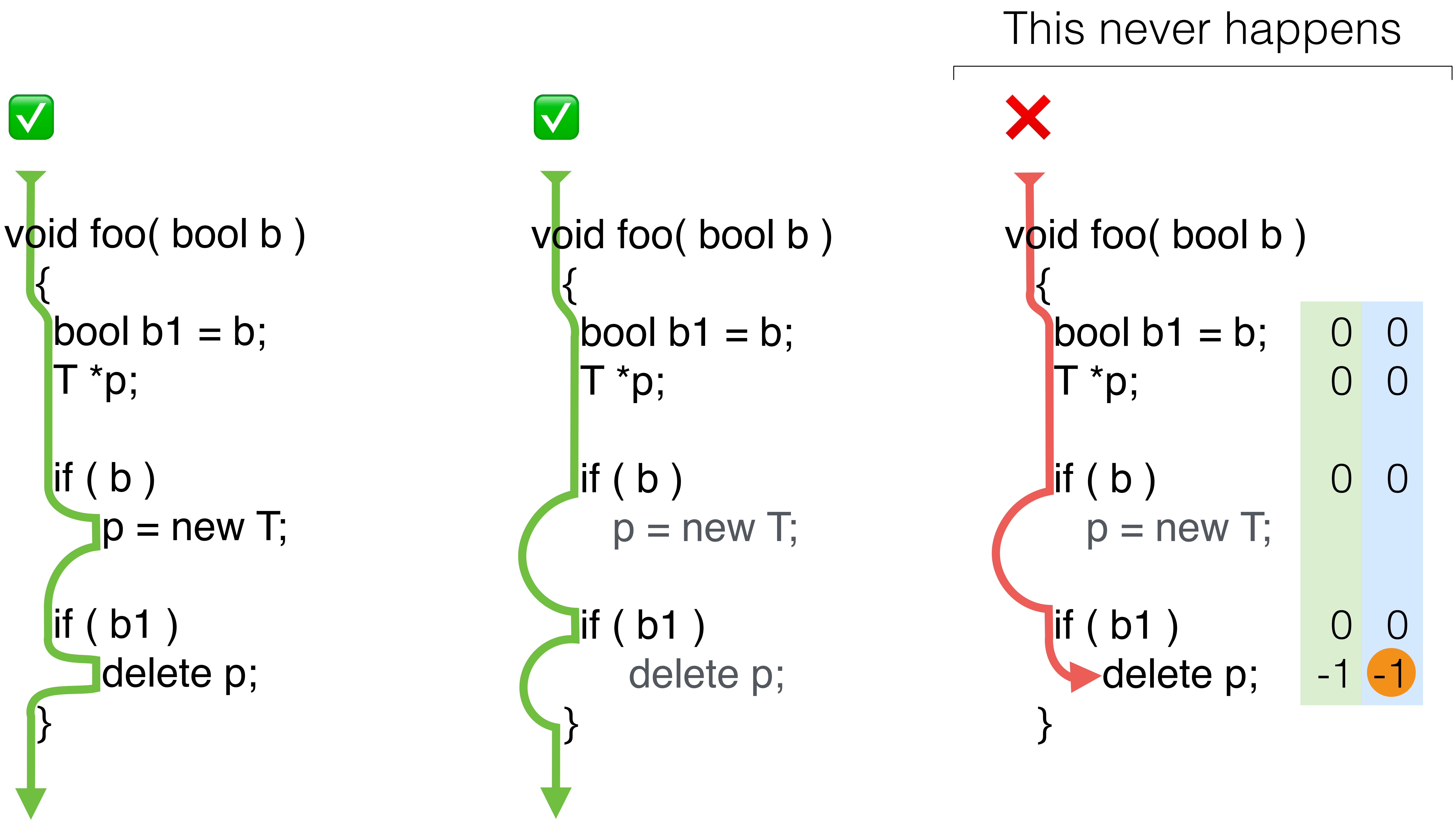
Locally atomic events are interchangeable when, in their prologues:

- the execution paths are identical,
- corresponding directly asserted capabilities are substitutable, and
- corresponding directly required capabilities are interchangeable.

If locally atomic events are interchangeable, then, in their epilogues:

- the execution paths will be identical,
- corresponding directly asserted capabilities will be substitutable, and
- corresponding directly required capabilities will be interchangeable.


This never happens
void foo(bool b)
{
bool b1 = b;
if (b)
return;
if (b1)
claim false;
}
void foo(bool b)
{
bool b1 = b;
if (b)
return;
if (b1)
claim false;
}
void foo(bool b)
{
bool b1 = b;
if (b)
return;
if (b1)
claim false;
}


This never happens
void foo(bool b)
{
bool b1 = b;
T *p;
if (b)
p = new T;
if (b1)
delete p;
}
void foo(bool b)
{
bool b1 = b;
T *p;
if (b)
p = new T;
if (b1)
delete p;
}
void foo(bool b)
{
bool b1 = b;
T *p;
if (b)
p = new T;
if (b1)
delete p;
}
0 0
0 0
0 0
0 0
-1 -1

Locally atomic events relate to each other through capabilities.

A causal nexus is formed when a prologue uses a capability provided by an earlier epilogue.

Locally atomic capabilities expose their relationships in their interfaces.

Requirements in the prologue expose dependency; requirements in the epilogue expose change.

The exposed relationships affect the neighborhood through substitutability.

Dependency allows the possibility of nonsubstitutability; change dispels substitutability.

Tracking capabilities and substitutability allows *in vitro* local testing.

Substitutability tells us which events, capabilities, and branches are interchangeable.

Questions?