What is the basic interface?

Lisa Lippincott

We routinely prove that our programs work.
(If we didn't, programs would hardly ever work.)

The proof behind a well-written function is mostly evident in its text.
(Because that's part of what we mean by "well-written.")

Sometimes, the intended proof is wrong.
(And that's a major source of bugs.)

So why aren't we having computers check our proofs?

We generally aren’t clear about what needs to be proved.

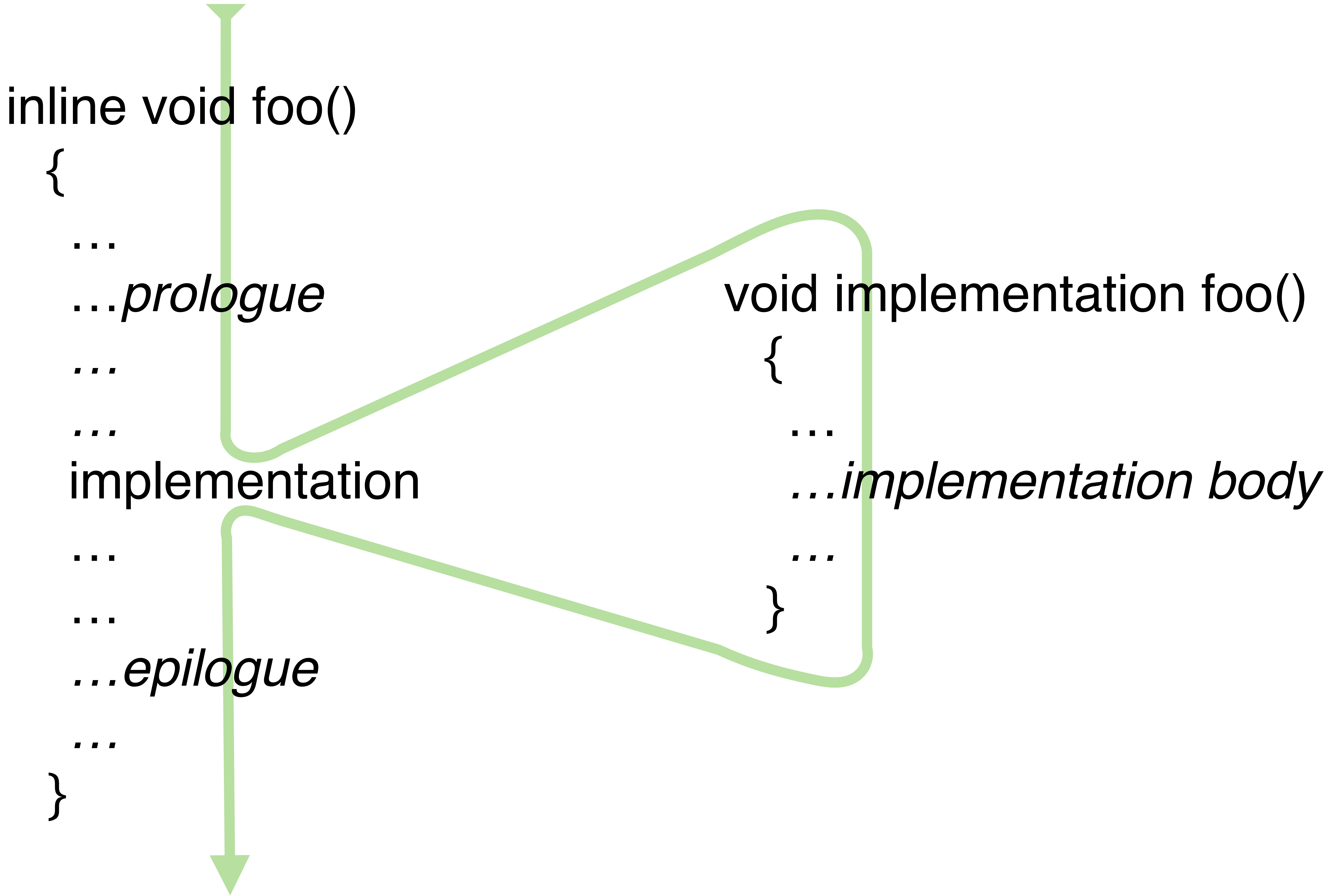
We don’t often write function interfaces formally.
(And formal mathematical notation is poorly suited to procedural programming.)

We don’t write down all the details of function interfaces.
(In full, they’re pretty complicated. It’s easier to let a lot go unsaid.)

We don’t even agree about what can go unsaid.

Function interface

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

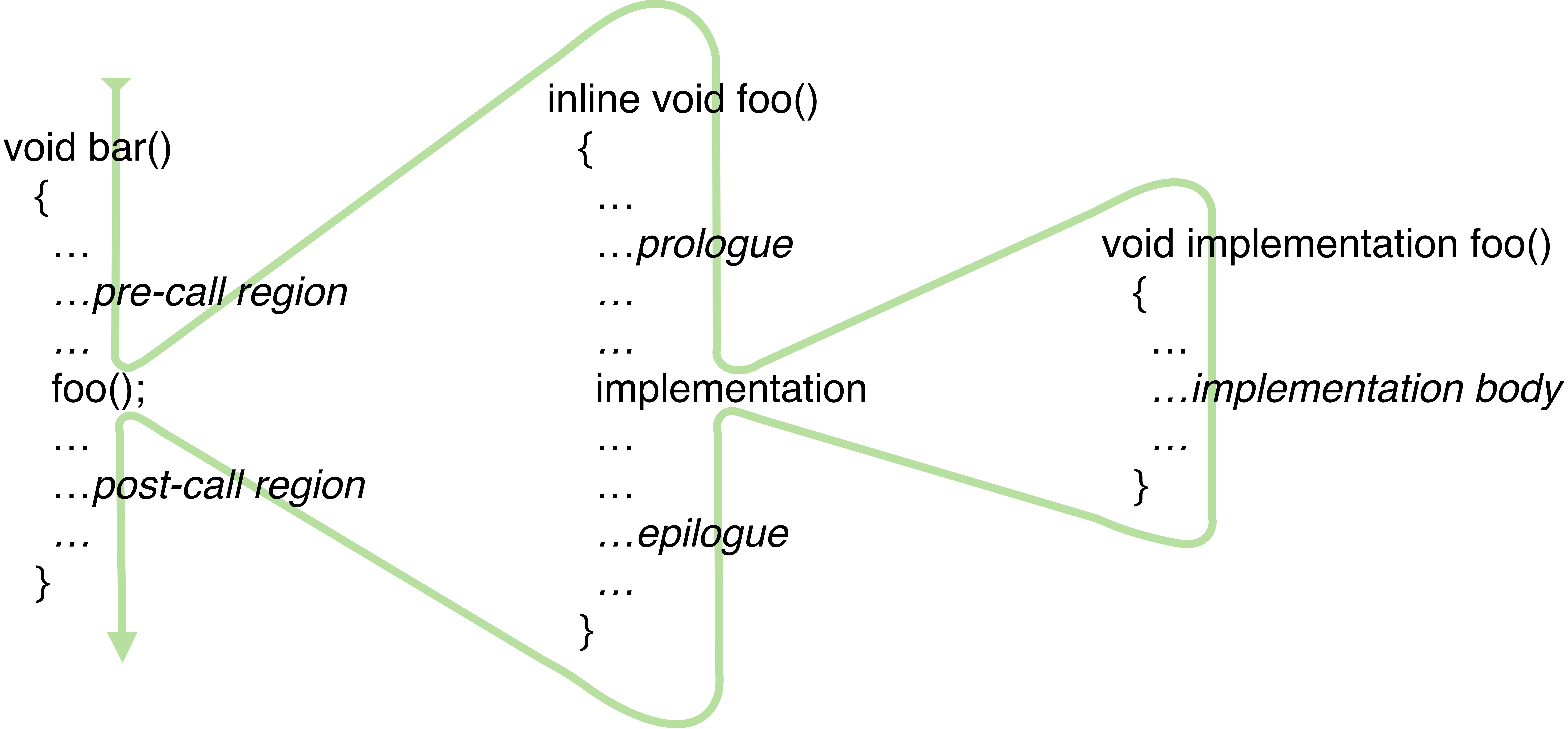


Function interface

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

Function implementation

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



Calling function

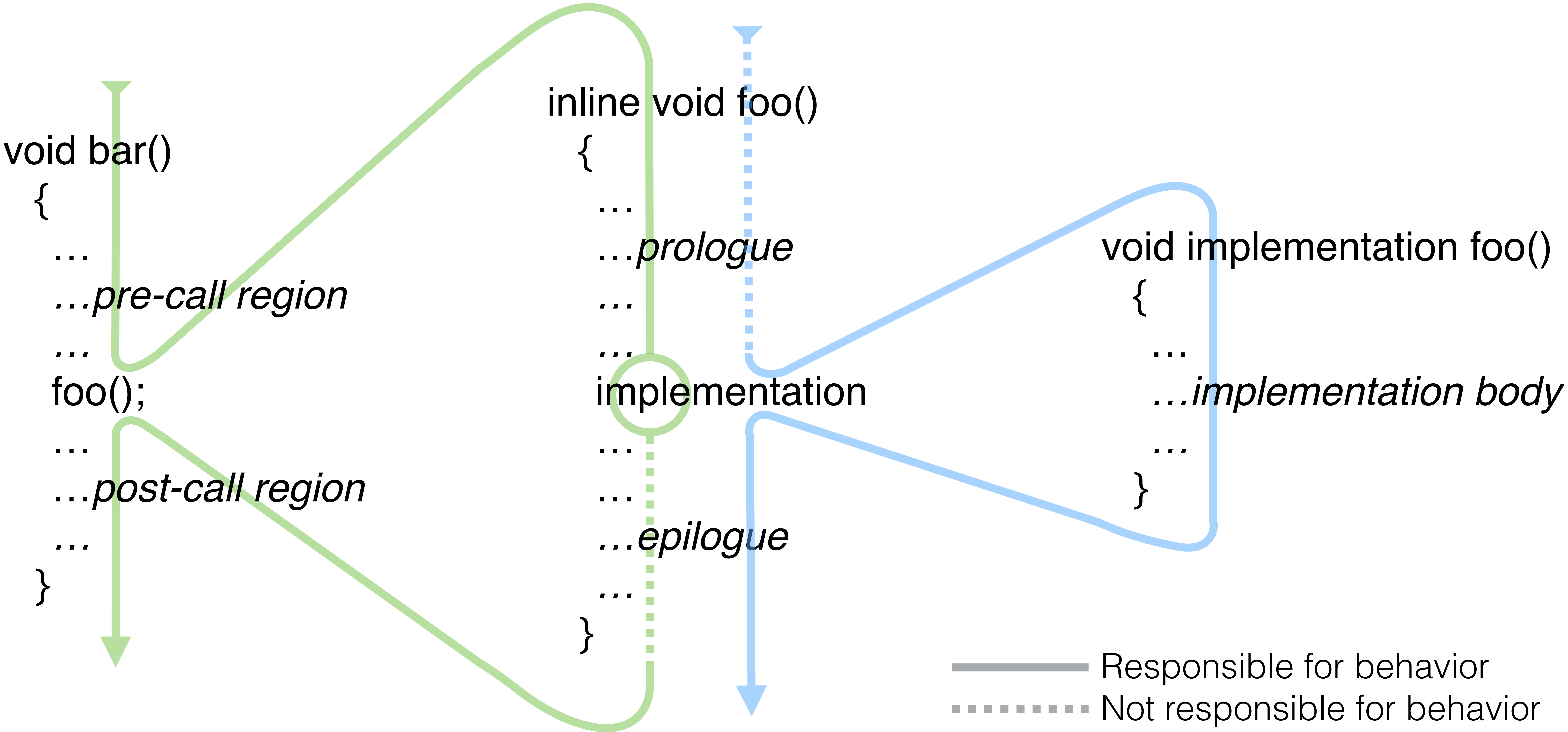
```
void bar()
  {
   …
   …pre-call region
   …
   foo();
   …
   …post-call region
   …
  }
```

Function interface

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

Function implementation

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



Calling neighborhood

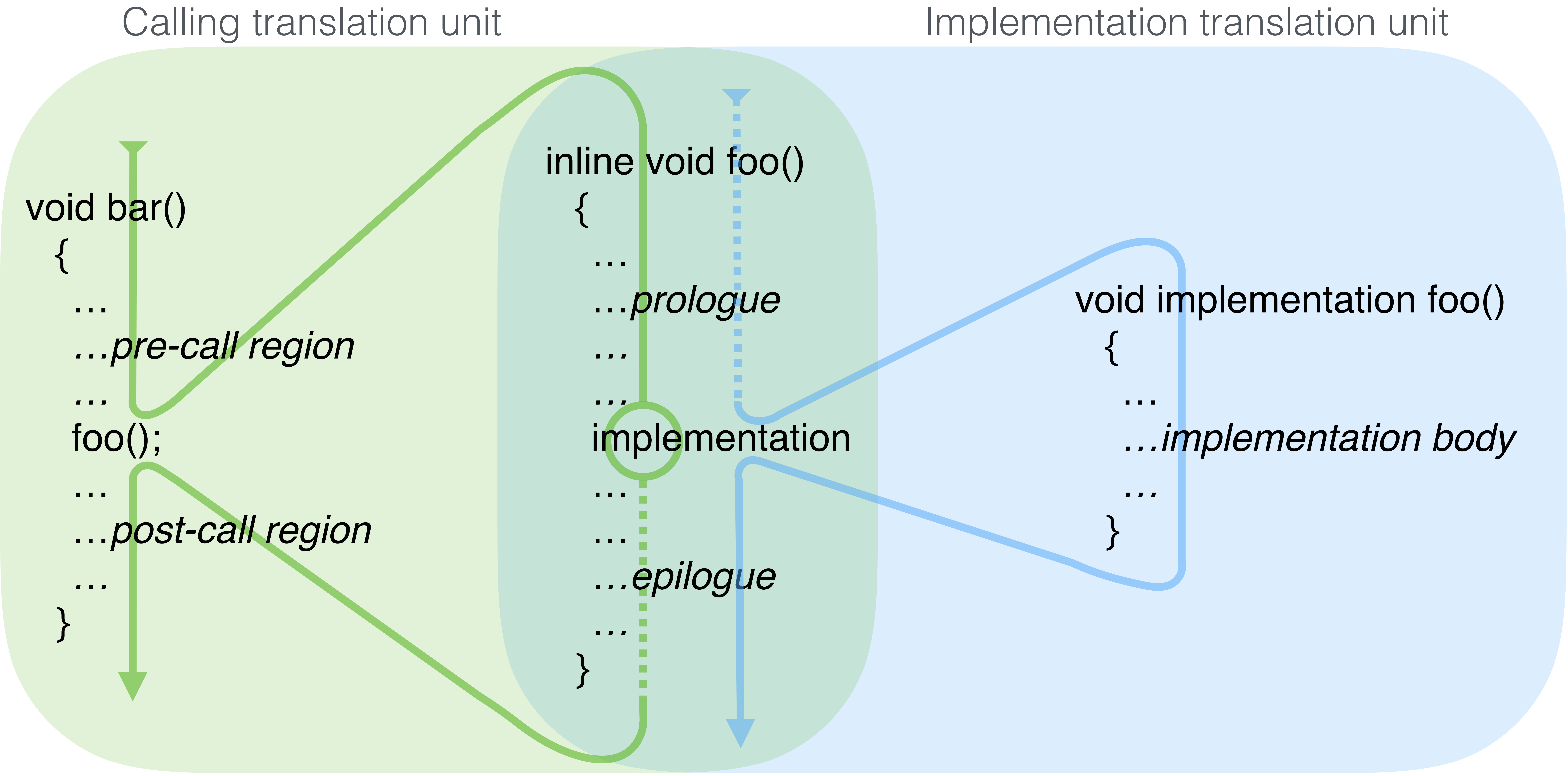
Implementation neighborhood

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

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

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

━━ Responsible for behavior
┅┅ Not responsible for behavior



Calling translation unit

Implementation translation unit

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

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

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

```
inline void foo()
  {


    implementation





  }
```

```
inline void foo()
  {

● // no undefined behavior so far!
    implementation


● // no undefined behavior so far!
  }
```

```
inline void foo()
  {

● // no undefined behavior so far!
    try implementation
      {

    ● // no undefined behavior so far!
    ● return;
      }
    catch ( ... )
      {

    ● // no undefined behavior so far!
    ● throw;
      }
  }
```

```
inline void foo( int& x )
  {


    implementation


  }
```

```
void
implementation foo( int& x )
  {
   x = 5;
  }
```

```
void bar1()
  {
   int z[2];
⚠ foo( *(z+2) );
  }
```

```
void bar2()
  {
   int *p = new int();
   delete p;
⚠ foo( *p );
  }
```

```
void bar3()
  {
   const int y = 2;
⚠ foo( const_cast<int&>(y) );
  }
```

```
inline void foo( int& x )
  {


    implementation


  }
```

```
void
implementation foo( int& x )
  {
🔥 x = 5;
  }
```

```
void bar1()
  {
   int z[2];
⚠️ foo( *(z+2) );
  }

void bar2()
  {
   int *p = new int();
   delete p;
⚠️ foo( *p );
  }

void bar3()
  {
   const int y = 2;
⚠️ foo( const_cast<int&>(y) );
  }
```

```
inline void foo( int& x )
  {

🔥 claim writable( x );
   implementation



  }
```

```
void
implementation foo( int& x )
  {
   x = 5;
  }
```

writable(t)   // t *is an lvalue of non-const POD type* T

Has no effect when assignment of an rvalue of type **T** to **t** has defined behavior.

Otherwise, the behavior is undefined. 🔥

```
inline void foo( int& x )
   {

● claim writable( x );
   implementation


   }
```

```
void bar()
  {
⚠ int x;
   foo( x );
  }
```

```
inline void foo( int& x )
  {

🔵 claim writable( x );
   implementation



  }
```

```
void
implementation foo( int& x )
  {
🔥 int y = x;
  }
```

```
void bar()
  {
⚠ int x;
   foo( x );
  }
```

```
inline void foo( int& x )
  {
🔥 claim readable( x );
🔵 claim writable( x );
   implementation


  }
```

```
void
implementation foo( int& x )
  {
   int y = x;
  }
```

readable(t) *//* **t** *is an lvalue of POD type* **T**

Has no effect when lvalue-to-rvalue conversion of **t** has defined behavior.

Otherwise, the behavior is undefined. 🔥

```
inline void foo( int& x )
  {
● claim readable( x );
● claim writable( x );
  implementation



  }
```

```
void
implementation foo( int& x )
  {
   int y = x;
  }
```

```
void bar()
  {
   int x = 1;
   foo( x );
🔥 x = 0;
  }
```

```
inline void foo( int& x )
  {
🔵 claim readable( x );
🔵 claim writable( x );
   implementation



  }
```

```
void
implementation foo( int& x )
  {
⚠️ x.~int();
   new(&x) const int( 2 );
  }
```

```
void bar()
  {
   int x = 1;
   foo( x );
🔥 int y = x;
  }
```

```
inline void foo( int& x )
  {
🔵 claim readable( x );
🔵 claim writable( x );
   implementation



  }
```

```
void
implementation foo( int& x )
  {
⚠️ x.~int();
   new(&x) int;
  }
```

```
void bar()
  {
   int x = 1;
   foo( x );
   int y = x;
  }
```

```
 inline void foo( int& x )
   {
 🔵 claim readable( x );
 🔵 claim writable( x );
    implementation

🔥🔵 claim readable( x );
 🔵 claim writable( x );
   }
```

```
void
implementation foo( int& x )
  {
⚠️ x.~int();
   new(&x) int;
  }
```

```
void bar()
  {
   int x = 1;
   try { foo( x ); }
   catch ( ... ) {}
   int y = x;
  }
```

```
inline void foo( int& x )
  {
🔵 claim readable( x );
🔵 claim writable( x );
   try implementation
     {
   🔵 claim readable( x );
   🔵 claim writable( x );
     }
   catch( ... )
     {
 🔥🔵 claim readable( x );
   🔵 claim writable( x );
     }
  }
```

```
void
implementation foo( int& x )
  {
⚠️ x.~int();
   new(&x) int;
   throw exception();
  }
```

```
inline void foo( int& x )
  {

● claim proper( x );
  try implementation
    {

   ● claim proper( x );
    }
  catch( ... )
    {

   ● claim proper( x );
    }
  }
```

```
inline void proper( int& i )
  {
   readable( i );
   writable( i );
  }
```

```
inline void foo( const int& x )
  {

● claim proper( x );
  try implementation
    {

   ● claim proper( x );
    }
  catch( ... )
    {

   ● claim proper( x );
    }
  }
```

```
inline void proper( int& i )
  {
   readable( i );
   writable( i );
  }
```

```
inline void proper( const int& i )
  {
   readable( i );
  }
```

proper(t)   *//* t *is an lvalue of a cv-qualified type* cvT

A sequence of assertions, specific to cvT, that are

- implicitly claimed 🔵
- as applied to parameters, function results, and thrown exceptions
  (including *this, except before construction, after constructor failure, or after destruction)
- immediately before responsibility is passed between neighborhoods.

```
void bar()
  {
⚠️ int x;
   foo( x );
   int y = x;
  }
```

```
inline void foo( int& x )
  {

🔥🔵 claim proper( x );
   try implementation
     {


     🔵 claim proper( x );
       }
   catch (...)
     {
     🔵 claim proper( x );
       }
  }
```

```
void
implementation foo( int& x )
  {
   x = 5;
  }
```

```
void bar()
  {
⚠ int x;
   foo( x );
   int y = x;
  }
```

```
inline void foo( [improper] int& x )
  {


   try implementation
     {



     }
   catch ( ... )
     {

     }
  }
```

```
void
implementation foo( int& x )
  {
❓ x = 5;
  }
```

```
void bar()
   {
⚠ int x;
   foo( x );
❓ int y = x;
   }
```

```
inline void foo( [improper] int& x )
   {

     claim writable( x );
     try implementation
       {



       }
     catch ( ... )
       {

       }
   }
```

```
void
implementation foo( int& x )
   {
     x = 5;
   }
```

```
void bar()
  {
⚠ int x;
   foo( x );
   int y = x;
  }
```

```
inline void foo( [improper] int& x )
  {

   claim writable( x );
   try implementation
     {
      claim proper( x );


     }
   catch ( ... )
     {

     }
  }
```

```
void
implementation foo( int& x )
  {
   x = 5;
  }
```

```
void bar()
  {
⚠  int x;
   foo( x );
   int y = x;
  }
```

```
inline void foo( [improper] int& x )
  {

   claim writable( x );
   try implementation
     {
      claim proper( x );


     }
   catch ( ... )
     {
      claim writable( x );
     }
  }
```

```
void
implementation foo( int& x )
  {
   x = 5;
  }
```

writable(t) // t *is a non-const lvalue of type* T
(local version)

Has no effect when **writable(t)** has previously been asserted
(including as a consequence of construction)

- in the responsible neighborhood
- more recently than **t** appears to have been destroyed.

Otherwise, the behavior is undefined. 🔥

readable(t) // t *is an lvalue of POD type* **T**

(local version)

Has no effect when **readable(t)** has previously been asserted
(including as a consequence of initialization or assignment)

- in the responsible neighborhood
- more recently than **t** appears to have been destroyed.

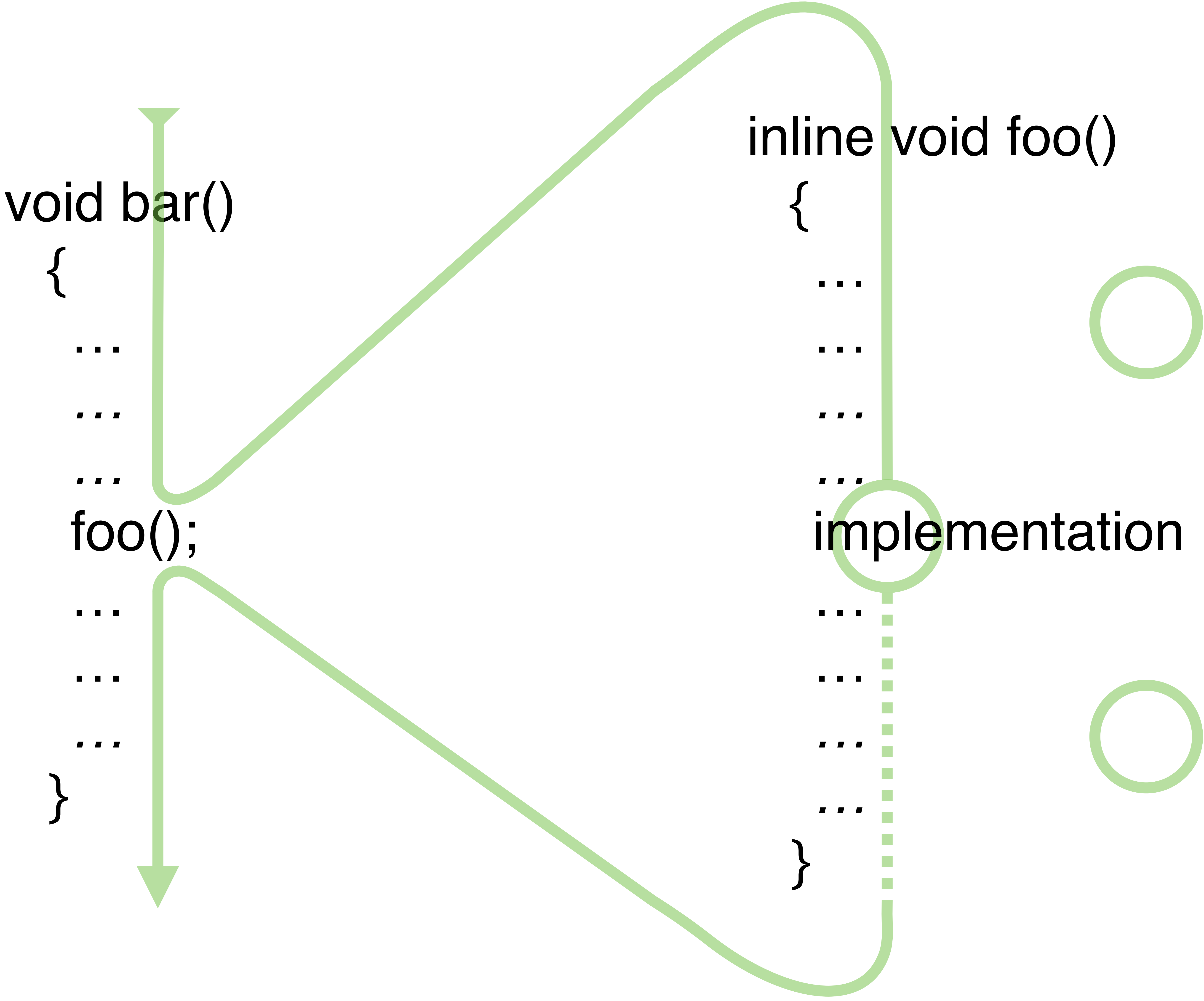
Otherwise, the behavior is undefined. 🔥

destructible(t) // t *is an lvalue of type* T
(local version)

Has no effect when **destructible(t)** has previously been asserted
(including as a consequence of construction)

- in the responsible neighborhood
- more recently than **t** appears to have been destroyed.

Otherwise, the behavior is undefined. 🔥



```
void bar()
  {
    …
    …
    …
    foo();
    …
    …
    …
  }
```

```
inline void foo()
  {
    …
    …
    …
    …
    implementation
    …
    …
    …
    …
  }
```

○ Anything locally readable may be read,
anything locally writable may be written, and
anything locally destructible may be destroyed.

○ Only locally readable objects are read,
only locally writable objects are written, and
only locally destructible objects are destroyed.

```
inline void foo( int& x )
  {
   claim destructible( x );
   try implementation
     {
      claim destructible( x );
     }
   catch ( ... )
     {
      claim destructible( x );
     }
  }
```

Should these claims
be part of the basic interface?
(Is **destructible** part of **proper**?)

Pro: Some functions need to destroy objects passed to them.

Con: Those are unusually dangerous functions.
Interfaces to unusually dangerous functions should spell out the dangers.

```
inline int foo( int& x )
  {

    try implementation
      {

    ● claim destructible( result );
      }
    catch ( ... )
      {

    ● claim destructible( exception );
      }
  }
```

Only for object types.

```
inline type::type()
  {


    try implementation
      {

   ● posit destructible( *this );
      }
    catch ( ... )
      {

   ● claim destructible( exception );

      }
  }
```

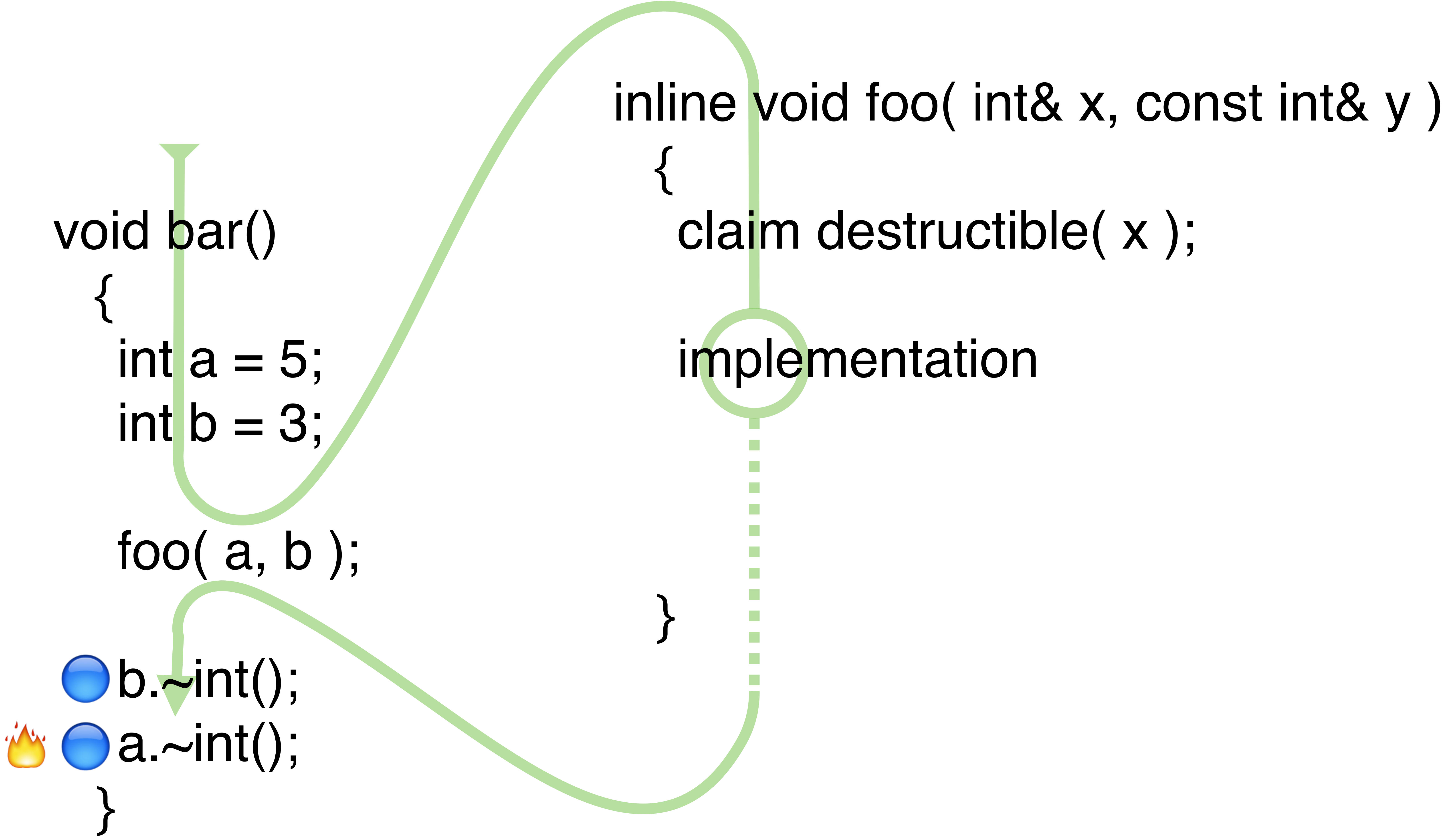
```
inline type::~type()
  {
 ● claim destructible( *this );

    try implementation
      {


      }
    catch ( ... )
      {

   ● claim destructible( exception );

      }
  }
```

```
inline type::type()
  {
 ●claim unoccupied_space( *this );

   try implementation
     {

     ●posit destructible( *this );
     }
   catch ( ... )
     {

     ●claim destructible( exception );
     ●claim unoccupied_space( *this );
     }
  }
```

```
inline type::~type()
  {
 ●claim destructible( *this );

   try implementation
     {

     ●claim unoccupied_space( *this );
     }
   catch ( ... )
     {

     ●claim destructible( exception );
     ●claim unoccupied_space( *this );
     }
  }
```

```
inline void foo( int& x, const int& y )
  {
   claim destructible( x );

   implementation




  }
```

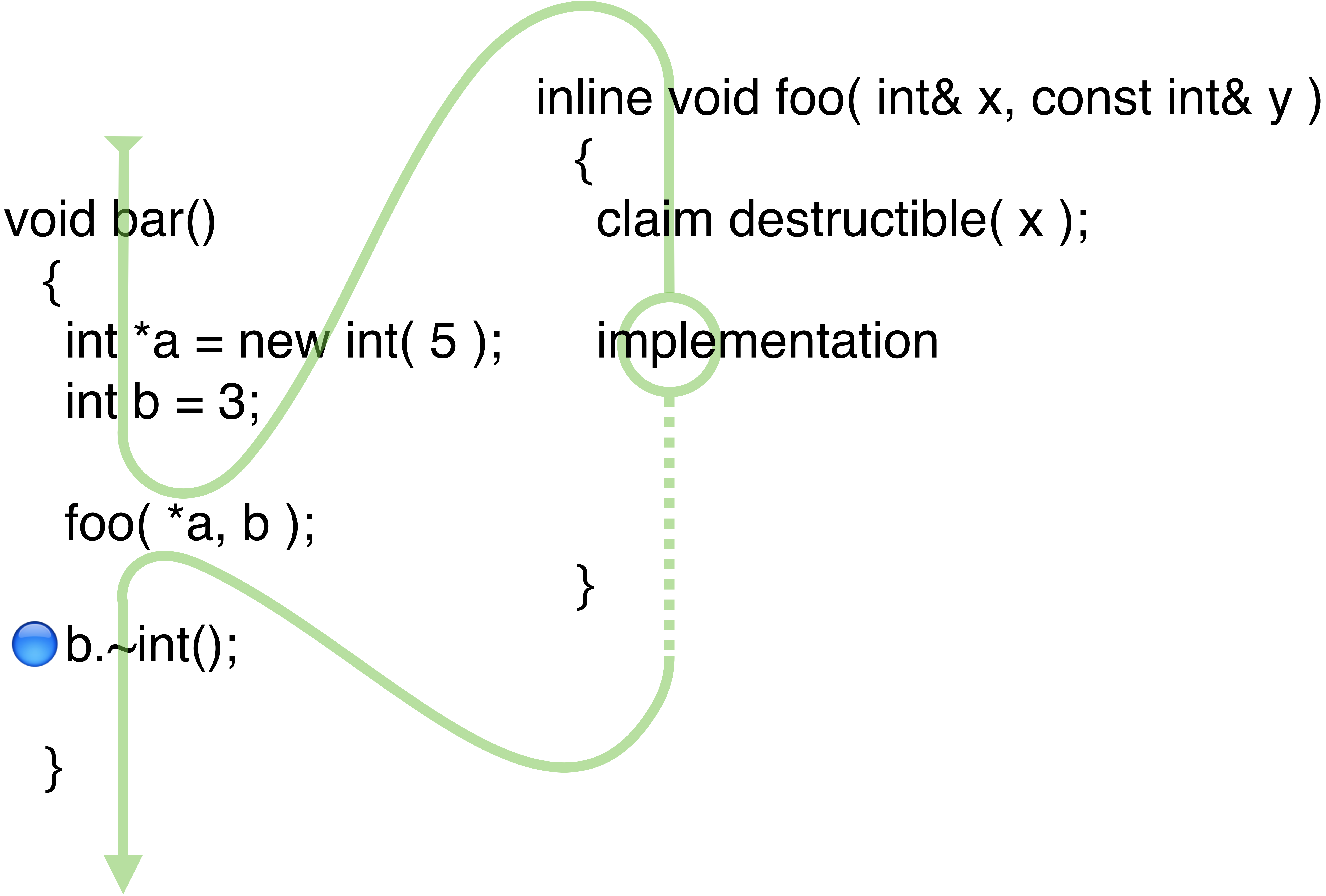
```
void
implementation foo( int& x,
                    const int& y )
  {
   x.~int();
   new( &x ) int( y );
  }
```



```
void bar()
  {
   int a = 5;
   int b = 3;

   foo( a, b );

🔵b.~int();
🔥🔵a.~int();
  }
```

```
inline void foo( int& x, const int& y )
  {
   claim destructible( x );

   implementation


  }
```

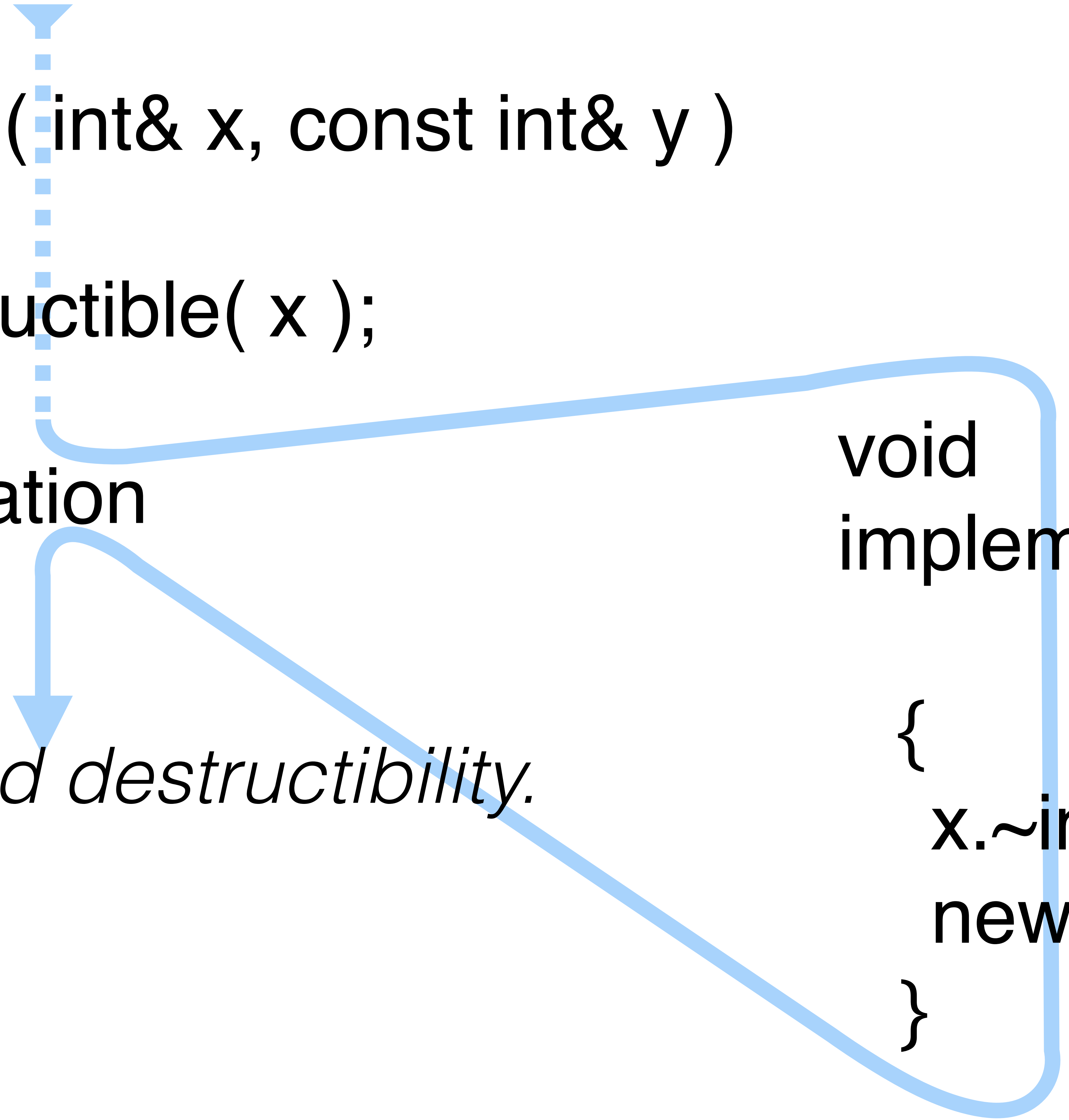
```
void
implementation foo( int& x,
                    const int& y )
  {
   x.~int();
   new( &x ) int( y );
  }
```



```
void bar()
  {
    int *a = new int( 5 );
    int b = 3;

    foo( *a, b );

    b.~int();

  }
```

```
inline void foo( int& x, const int& y )
  {
    claim destructible( x );

    implementation


  }
```

```
void
implementation foo( int& x,
                     const int& y )
  {
    x.~int();
    new( &x ) int( y );
  }
```

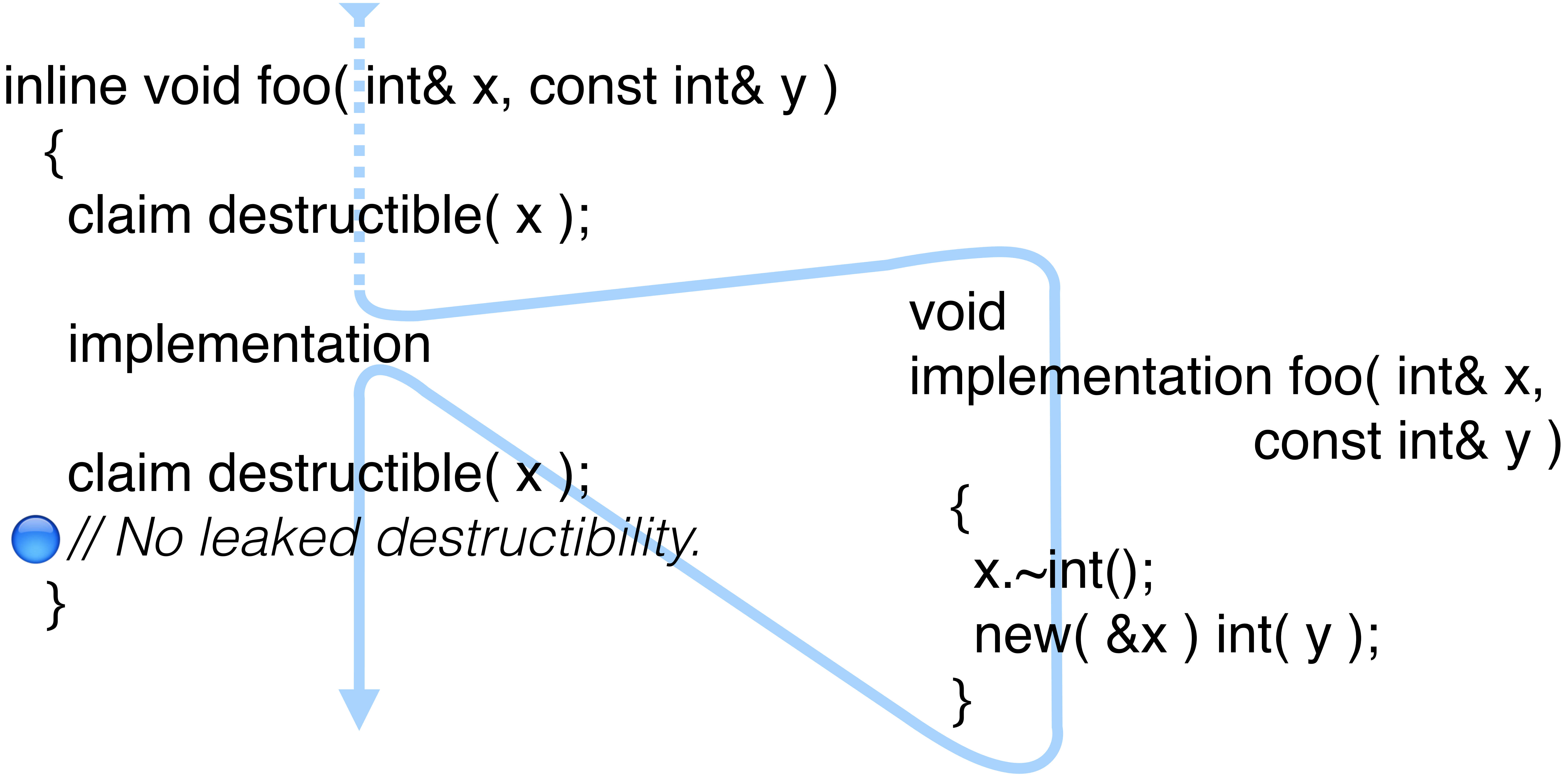


```
void bar()
  {
    int a = 5;
    int b = 3;

    foo( a, b );

    b.~int();
    a.~int();
  }
```

```
inline void foo( int& x, const int& y )
  {
    claim destructible( x );

    implementation

🔥🔵// No leaked destructibility.
  }
```

```
void
implementation foo( int& x,
                    const int& y )
  {
    x.~int();
    new( &x ) int( y );
  }
```

```
void bar()
  {
   int a = 5;
   int b = 3;

   foo( a, b );

   b.~int();
   a.~int();
  }
```



```
inline void foo( int& x, const int& y )
  {
   claim destructible( x );

   implementation

   claim destructible( x );
  // No leaked destructibility.
  }
```

```
void
implementation foo( int& x,
                     const int& y )
  {
   x.~int();
   new( &x ) int( y );
  }
```

```
void bar()
  {
   int a = 5;
   int b = 3;

   foo( a, b );

   b.~int();
   a.~int();
  }
```

```
inline void foo( int& x, const int& y )
  {
   claim destructible( x );

   try implementation
     {
      claim destructible( x );
   ● // No leaked destructibility.
     }
   catch ( ... )
     {
   ● // No leaked destructibility.
     }
  }
```

```
void
implementation foo( int& x,
                    const int& y )
  {
   x.~int();
   new( &x ) int( y );
  }
```

```
inline void foo( int& x )
  {
   try implementation
   // No leaked readability or writability.
  }
catch ( ... )
  {
   // No leaked readability or writability.
  }
```

Should these assertions be part of the basic interface?

(Can other capabilities be leaked?)

Con: Apart from destruction, no operations forget readability or writability.

Pro: Some operations need to hide or transfer these capabilities.

```
inline void foo( int& x, const int& y )
  {


    claim destructible( x );
●   claim proper( x );
●   claim proper( y );

    implementation

    claim destructible( x );
●   claim proper( x );
●   claim proper( y );
  }
```

```
void
implementation foo( int& x,
                    const int& y )
  {

    x.~int();
    new( &x ) int( y );
  }
```

```
void bar()
  {
   int a = 5;

⚠ foo( a, a );
  }
```

```
inline void foo( int& x, const int& y )
  {

   claim destructible( x );
🔵 claim proper( x );
🔵 claim proper( y );

   implementation

   claim destructible( x );
🔵 claim proper( x );
🔵 claim proper( y );
  }
```

```
void
implementation foo( int& x,
                    const int& y )
  {

   x.~int();
🔥 new( &x ) int( y );
  }
```

Two lvalues are *irresponsibly aliased* when aliasing is not communicated across an interface.

Specifically, **t** and **t’** are irresponsibly aliased when:

- the execution path through the responsible neighborhood clearly requires **t** and **t’** to have the same address, and
- the shared execution path through the interface does not clearly require **t** and **t’** to have the same address.

writable(t)   // t *is an lvalue of non-const POD type* T

(local non-aliasing version)

Has no effect when

- **writable(t)** has previously been asserted in the responsible neighborhood more recently than **t** appears to have been destroyed, and
- **t** is not irresponsibly aliased to any lvalue that may be readable, writable, or destructible in an irresponsible neighborhood.

Otherwise, the behavior is undefined. 🔥

readable(t) // **t** *is an lvalue of POD type* **T**

(local non-aliasing version)

Has no effect when

- **readable(t)** has previously been asserted in the responsible neighborhood more recently than **t** appears to have been destroyed, and
- **t** is not irresponsibly aliased to any lvalue that may be writable or destructible in an irresponsible neighborhood.

Otherwise, the behavior is undefined. 🔥

destructible(t) *// **t** is an lvalue of type* **T**

(local non-aliasing version)

Has no effect when

- **destructible(t)** has previously been asserted in the responsible neighborhood more recently than **t** appears to have been destroyed, and
- **t** is not irresponsibly aliased to any lvalue that may be readable, writable, or destructible in an irresponsible neighborhood.

Otherwise, the behavior is undefined. 🔥

```
void bar()
  {
   int a = 5;

⚠️ foo( a, a );
  }
```

```
inline void foo( int& x, const int& y )
  {

     claim destructible( x );
  🔵 claim proper( x );
🔥🔵 claim proper( y );

     implementation

     claim destructible( x );
  🔵 claim proper( x );
  🔵 claim proper( y );
  }
```

```
void
implementation foo( int& x,
                    const int& y )
  {

   x.~int();
   new( &x ) int( y );
  }
```

```
void bar()
  {
   int a = 5;

⚠️ foo( a, a );
  }
```

```
inline void foo( int& x, const int& y )
  {


     claim destructible( x );
  🔵 claim proper( x );
🔥🔵 claim proper( y );

     implementation

     claim destructible( x );
  🔵 claim proper( x );
  🔵 claim proper( y );
  }
```

```
void
implementation foo( int& x,
                    const int& y )
  {
   int t = y;
   x.~int();
   new( &x ) int( t );
  }
```

```
void bar()
  {
   int a = 5;

   foo( a, a );
  }
```

```
inline void foo( int& x, const int& y )
  {
   claim &x == &y;

   claim destructible( x );
 ● claim proper( x );
 ● claim proper( y );

   implementation

   claim destructible( x );
 ● claim proper( x );
 ● claim proper( y );
  }
```

```
void
implementation foo( int& x,
                    const int& y )
  {
   int t = y;
   x.~int();
   new( &x ) int( t );
  }
```

```
void bar()
  {
   int a = 5;
   int b = 3;
   foo( a, b );
  }
```

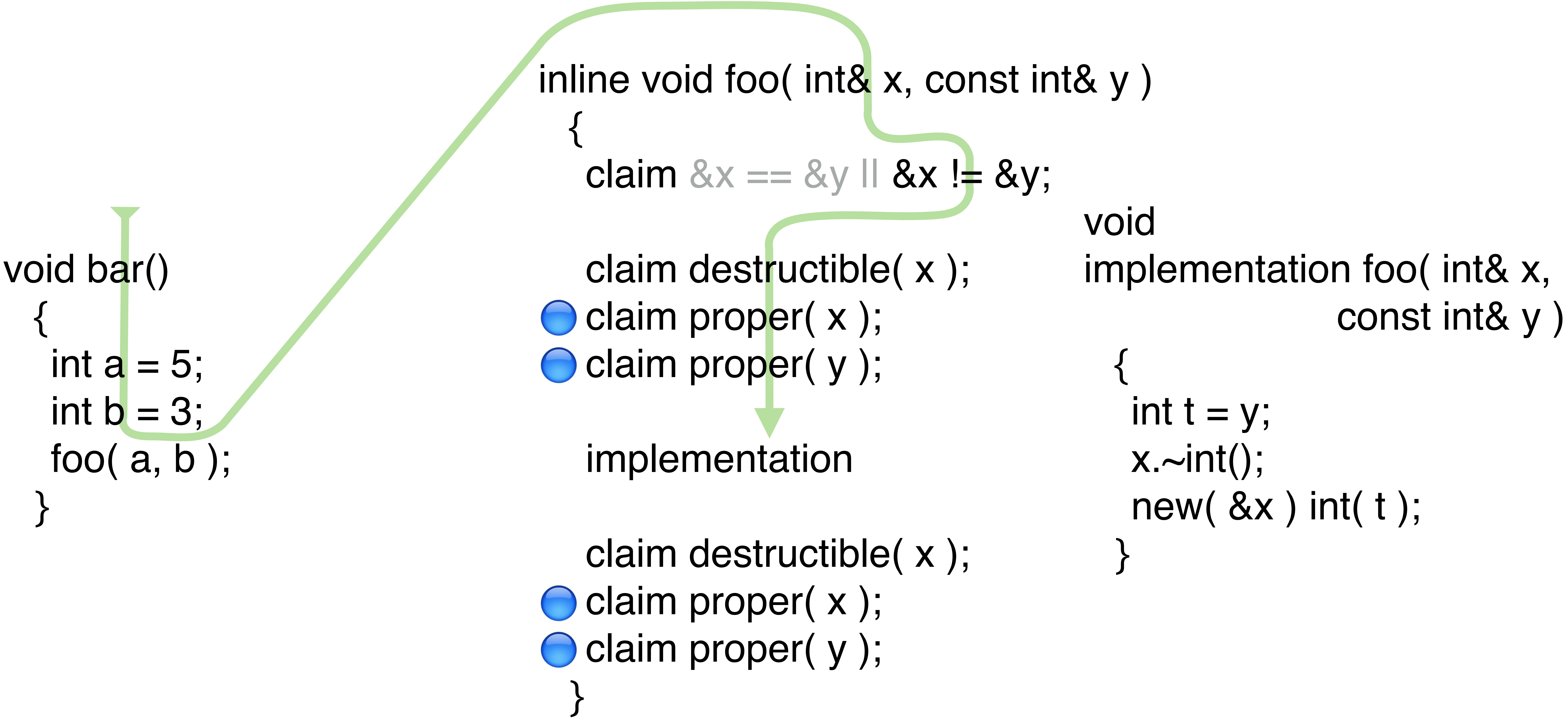
```
inline void foo( int& x, const int& y )
  {
🔥 claim &x == &y;

   claim destructible( x );
🔵 claim proper( x );
🔵 claim proper( y );

   implementation

   claim destructible( x );
🔵 claim proper( x );
🔵 claim proper( y );
  }
```

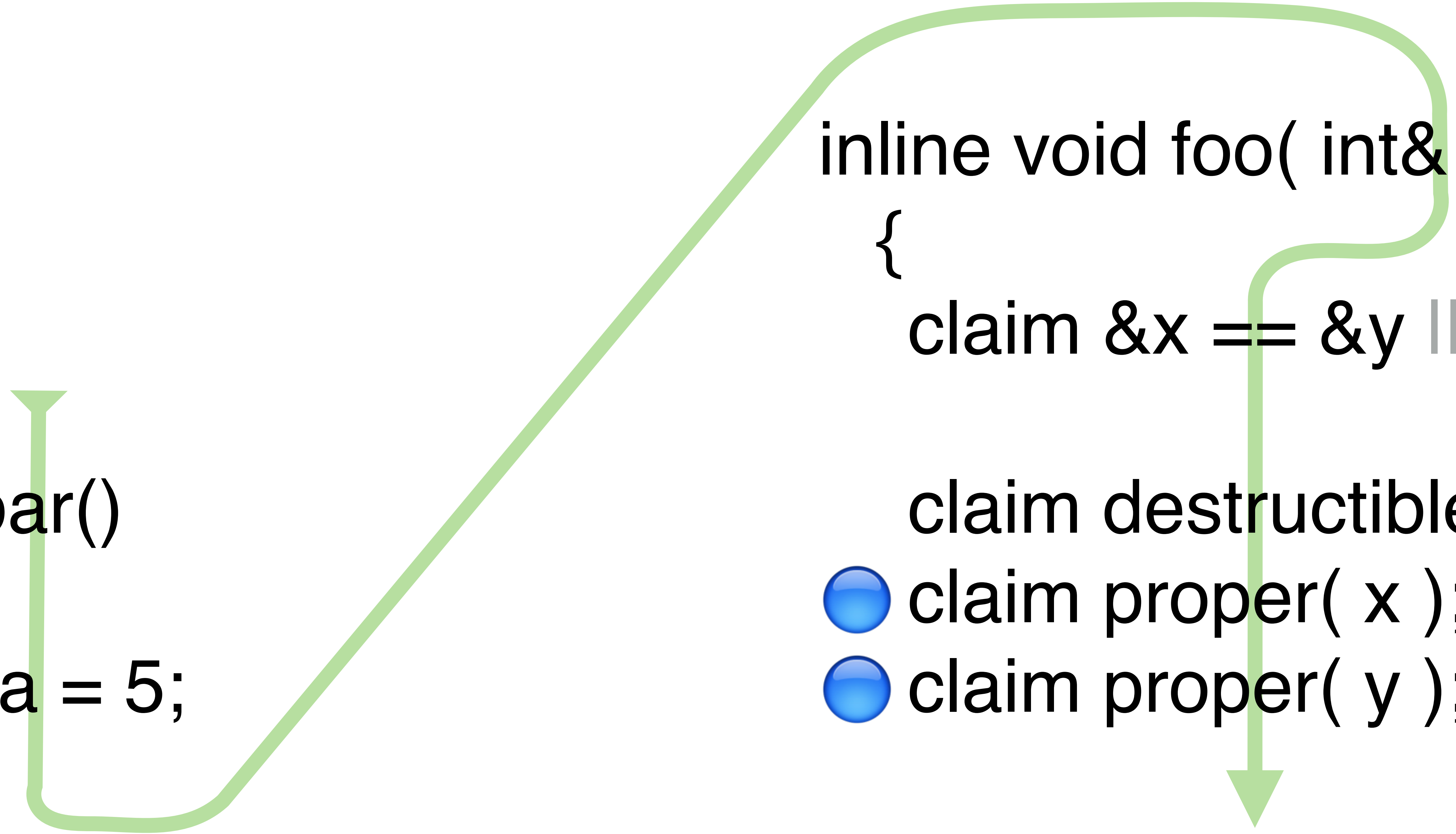
```
void
implementation foo( int& x,
                    const int& y )
  {
   int t = y;
   x.~int();
   new( &x ) int( t );
  }
```

```
void bar()
  {
    int a = 5;
    int b = 3;
    foo( a, b );
  }
```

```
inline void foo( int& x, const int& y )
  {
    claim &x == &y || &x != &y;

    claim destructible( x );
  ● claim proper( x );
  ● claim proper( y );

    implementation

    claim destructible( x );
  ● claim proper( x );
  ● claim proper( y );
  }
```

```
void
implementation foo( int& x,
                    const int& y )
  {
    int t = y;
    x.~int();
    new( &x ) int( t );
  }
```



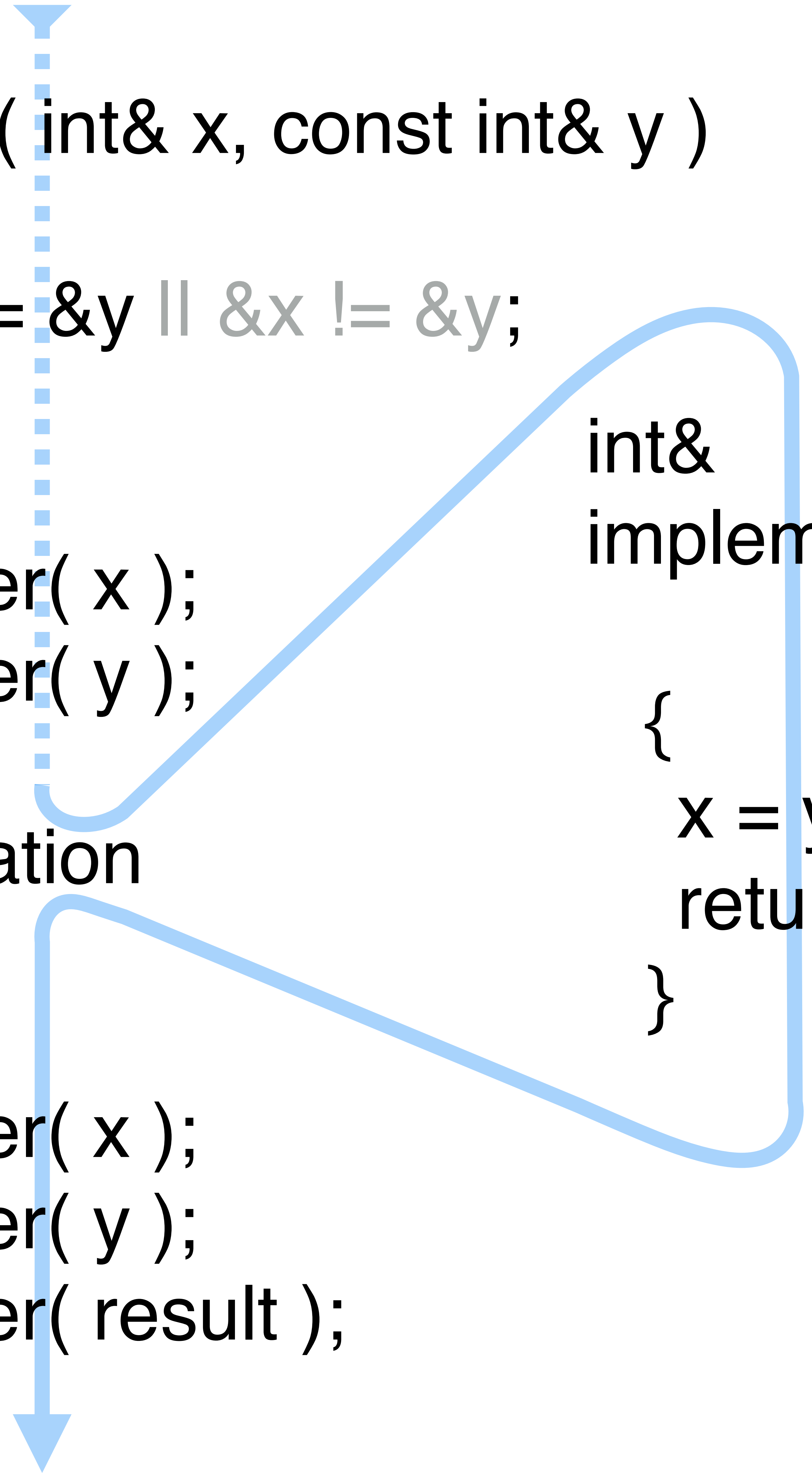


```
void bar()
  {
   int a = 5;

   foo( a, a );
  }
```

```
inline void foo( int& x, const int& y )
  {
   claim &x == &y || &x != &y;

   claim destructible( x );
●  claim proper( x );
●  claim proper( y );

   implementation

   claim destructible( x );
●  claim proper( x );
●  claim proper( y );
  }
```

```
void
implementation foo( int& x,
                    const int& y )
  {
   int t = y;
   x.~int();
   new( &x ) int( t );
  }
```

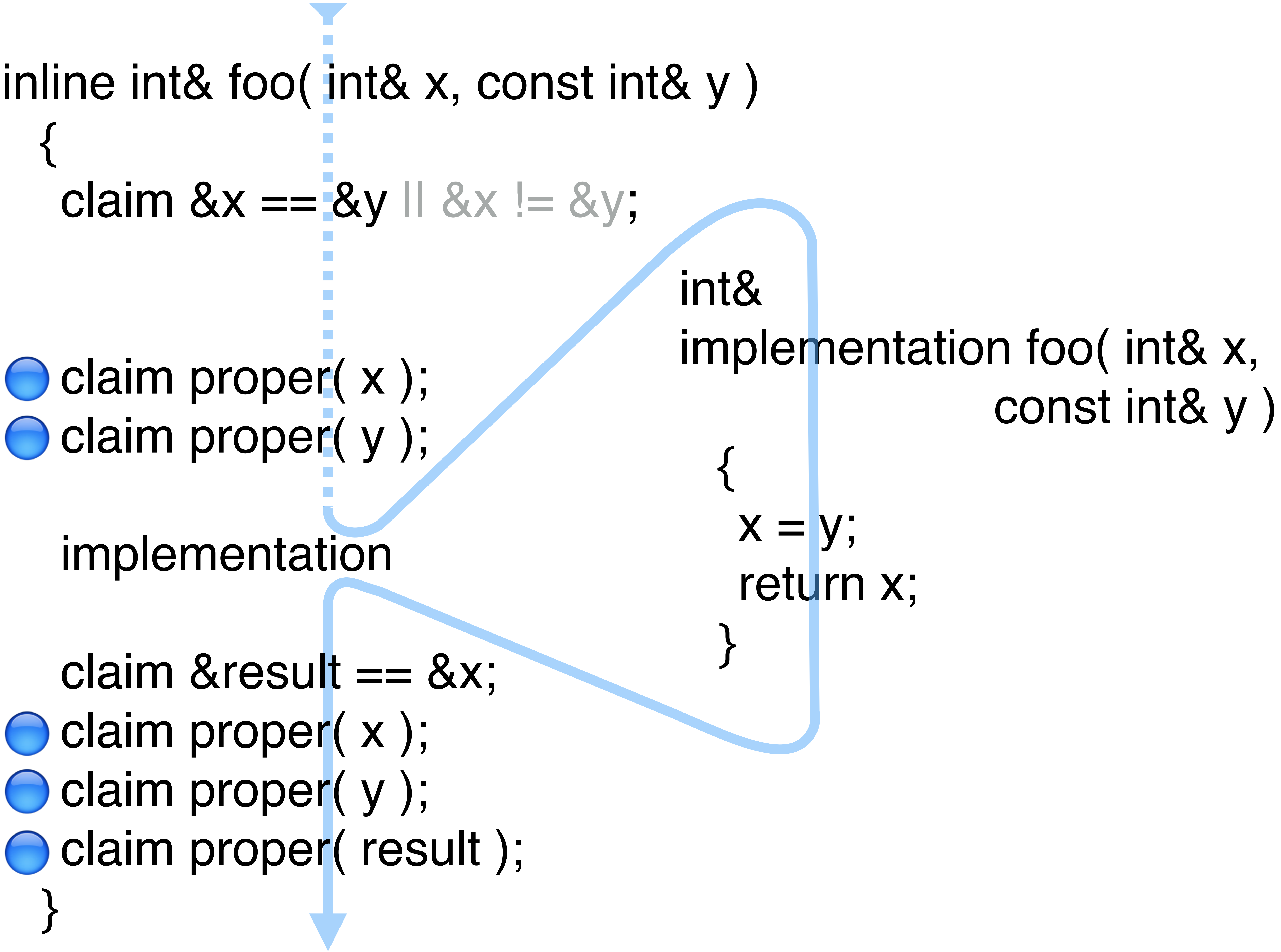
```
void bar()
  {
   int a = 5;

   foo( a, a ) = 0;
  }
```



```
inline int& foo( int& x, const int& y )
  {
   claim &x == &y || &x != &y;


● claim proper( x );
● claim proper( y );

   implementation


● claim proper( x );
● claim proper( y );
🔥● claim proper( result );
  }
```

```
int&
implementation foo( int& x,
                    const int& y )
  {
   x = y;
   return x;
  }
```

```
void bar()
  {
   int a = 5;

   foo( a, a ) = 0;
  }
```



```
inline int& foo( int& x, const int& y )
  {
   claim &x == &y || &x != &y;


● claim proper( x );
● claim proper( y );

   implementation

   claim &result == &x;
● claim proper( x );
● claim proper( y );
● claim proper( result );
  }
```

```
int&
implementation foo( int& x,
                    const int& y )
  {
   x = y;
   return x;
  }
```

```
inline int& foo( int& x, const int& y )
  {
   claim &x == &y || &x != &y;

   implementation

   claim &result == &x || &result != &x;
   claim &result == &y || &result != &y;
  }
```

Should these claims
be part of the basic interface?
(When the cv-unqualified types match.)

Pro: More common functions would fit the basic interface.
Including assignment, stream operators, and swap.

Con: Proofs have to consider many more execution paths.
Also, assignment, stream operators, and swap need more specific interfaces anyway.

```
inline const int *
foo( const int *x,
     const int *y )
  {



      implementation



  }
```

```
const int *implementation
foo( const int *x,
     const int *y )
  {
   std::less<const int *> less;
   bool q = less( x, y );

   return q ? x : y;
  }
```

```
inline int
foo( const int& x,
     const int& y )
  {




   implementation




  }
```

```
int implementation
foo( const int& x,
     const int& y )
  {
   std::less<const int *> less;
⚠  bool q = less( &x, &y );

   return q ? x : y;
  }
```

```
void bar( const int& a0,
          const int& b0 )
  {
   auto r0 = foo( a0, b0 );

   auto a1 = a0;
   auto b1 = b0;

   auto r1 = foo( a1, b1 );

🔥 claim r0 == r1;
  }
```

```
inline int
foo( const int& x,
     const int& y )
  {








   implementation



  }
```

```
int implementation
foo( const int& x,
     const int& y )
  {
   std::less<const int *> less;
⚠️ bool q = less( &x, &y );

   return q ? x : y;
  }
```

An rvalue expression is *indiscernible* to a function if the function is symmetric with respect to permutations of the possible values of the expression.

When an expression is indiscernible, all values of the expression are alike to the function.

```
void bar( const int& a0,
          const int& b0 )
  {
   auto r0 = foo( a0, b0 );

   auto a1 = a0;
   auto b1 = b0;

   auto r1 = foo( a1, b1 );

🔥 claim r0 == r1;
  }
```

```
inline int
foo( const int& x,
     const int& y )
  {





🔵 claim proper( x );
🔵 claim proper( y );
   implementation




  }
```

```
int implementation
foo( const int& x,
     const int& y )
  {
   std::less<const int *> less;
⚠️ bool q = less( &x, &y );

   return q ? x : y;
  }
```

These values are locally *indiscernible*:

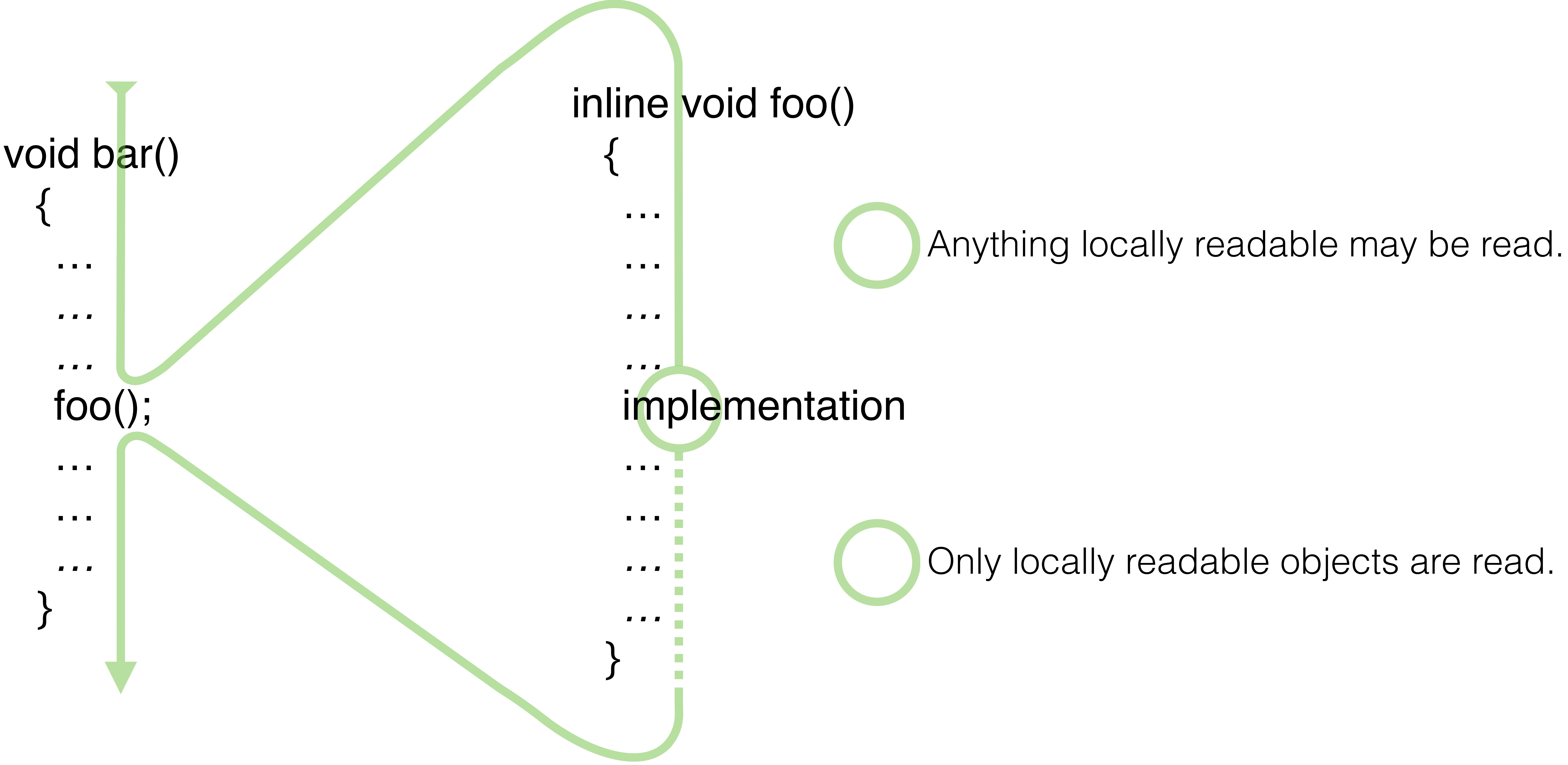
- addresses of readable, writable, destructible, or proper lvalues
  (The functions **readable**, **writable**, **destructible**, and **proper** make them indiscernible.)

- end iterators of readable, writable, destructible, or proper ranges
  (**readable_range**, **writable_range**, **destructible_range**, and **proper_range** make them indiscernible.)

- iterators that have been dereferenced
  (Interfaces to **operator*** and **operator->** need to make them indiscernible.)

- representation bytes of readable scalar objects
  (**readable** makes them indiscernible.)

- padding bytes of readable POD objects
  (**readable** makes them indiscernible.)



```
void bar()
  {
    …
    …
    …
    foo();
    …
    …
    …
  }
```

```
inline void foo()
  {
    …
    …
    …
    …
    implementation
    …
    …
    …
    …
  }
```

Anything locally readable may be read.

Only locally readable objects are read.

A function’s *gross input* consists of every rvalue mentioned in the prologue:

- (implicitly) the addresses of the parameters and local variables,
- the rvalues calculated in the prologue, and
- the values of lvalues asserted **readable** in the prologue.

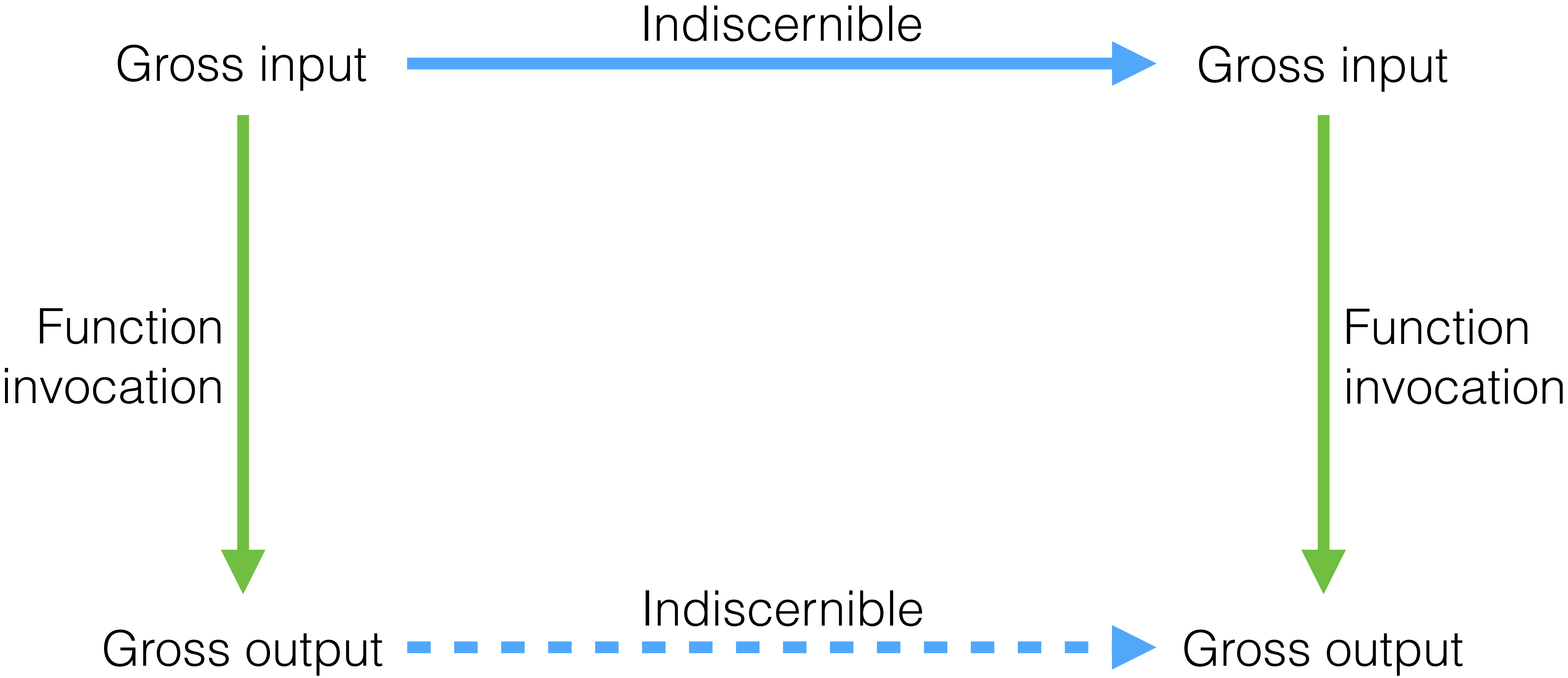
A function’s *gross output* consists of every rvalue mentioned in the epilogue:

- (implicitly) the addresses of the parameters, local variables, and result,
- the rvalues calculated in the epilogue, and
- the values of lvalues asserted **readable** in the epilogue.

It’s tempting to say this, because it’s *almost* right:

> A function’s ***net input*** is its gross input modulo indiscernible changes.
>
> A function’s ***net output*** is its gross output modulo indiscernible changes.
>
> Functions should repeatably map their net input to their net output.

But pretending symmetry is a kind of equality never quite works…



Indiscernible

Gross input

Gross input

Function invocation

Function invocation

Indiscernible

Gross output

Gross output

An rvalue is *irresponsibly indiscernible* when local indiscernibility is not communicated on return from implementation to caller.

Specifically, **v** is irresponsibly indiscernible when:

- the execution path through the responsible implementation neighborhood clearly requires **v** to be locally indiscernible, and
- the shared execution path through the interface does not clearly require **v** to be locally indiscernible.

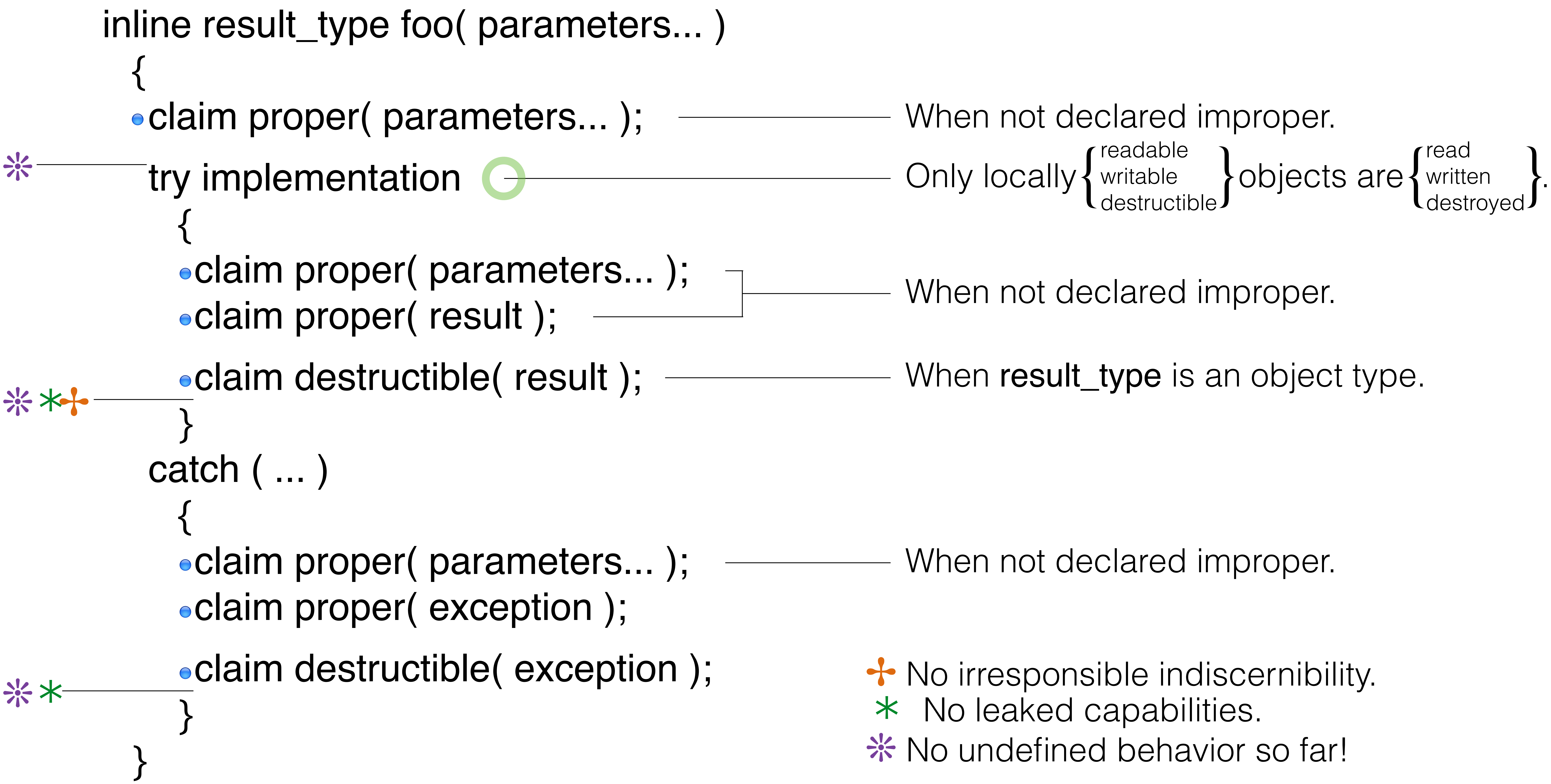
```
void bar( const int& a0,
          const int& b0 )
  {
   auto r0 = foo( a0, b0 );

   auto a1 = a0;
   auto b1 = b0;

   auto r1 = foo( a1, b1 );

   claim r0 == r1;
  }
```

```
inline int
foo( const int& x,
     const int& y )
  {




🔵 claim proper( x );
🔵 claim proper( y );
   implementation

🔵 claim proper( result );
🔥🔵 // No irresponsible indiscernibility.
  }
```

```
int implementation
foo( const int& x,
     const int& y )
  {
   std::less<const int *> less;
⚠️ bool q = less( &x, &y );

   return q ? x : y;
  }
```

```
inline int
foo( const int& x,
     const int& y )
  {
   std::less<const int *> less;
   bool q = less( &x, &y );
   // q is locally discernible!

● claim proper( x );
● claim proper( y );
   implementation

● claim proper( result );
● // No irresponsible indiscernibility.
  }
```

```
int implementation
foo( const int& x,
     const int& y )
  {
   std::less<const int *> less;
   bool q = less( &x, &y );

   return q ? x : y;
  }
```



```
       inline result_type foo( parameters... )
         {
         •claim proper( parameters... );
❊         try implementation ○
           {
           •claim proper( parameters... );
           •claim proper( result );

           •claim destructible( result );
❊✱✢        }
          catch ( ... )
           {
           •claim proper( parameters... );
           •claim proper( exception );

           •claim destructible( exception );
❊✱         }
         }
```

When not declared improper.

Only locally {readable, writable, destructible} objects are {read, written, destroyed}.

When not declared improper.

When **result_type** is an object type.

When not declared improper.

✢ No irresponsible indiscernibility.
✱ No leaked capabilities.
❊ No undefined behavior so far!

```
       inline result_type type::foo( parameters... )
         {
         •claim proper( *this );
         •claim proper( parameters... );
❊          try implementation ○
             {
             •claim proper( *this );
             •claim proper( parameters... );
             •claim proper( result );

❊✳✢          •claim destructible( result );
             }
           catch ( ... )
             {
             •claim proper( *this );
             •claim proper( parameters... );
             •claim proper( exception );

❊✳           •claim destructible( exception );
             }
         }
```

When not declared improper.

Only locally $\left\{\begin{array}{l}\text{readable}\\ \text{writable}\\ \text{destructible}\end{array}\right\}$ objects are $\left\{\begin{array}{l}\text{read}\\ \text{written}\\ \text{destroyed}\end{array}\right\}$.

When not declared improper.

When **result_type** is an object type.

When not declared improper.

✢ No irresponsible indiscernibility.
✳ No leaked capabilities.
❊ No undefined behavior so far!

```
      inline type::type( parameters... )
        {
        •claim unoccupied_space( *this );

        •claim proper( parameters... );
❊ ——————
          try implementation ○
            {
            •claim proper( *this );
            •claim proper( parameters... );

            •posit destructible( *this );
❊✳✣ ————————
            }
          catch ( ... )
            {
            •claim proper( parameters... );
            •claim proper( exception );

            •claim destructible( exception );
            •claim unoccupied_space( *this );
❊✳ ————————
            }
        }
```

When not declared improper.

Only locally $\left\{\begin{array}{l}\text{readable}\\ \text{writable}\\ \text{destructible}\end{array}\right\}$ objects are $\left\{\begin{array}{l}\text{read}\\ \text{written}\\ \text{destroyed}\end{array}\right\}$.

When not declared improper.

When not declared improper.

✣ No irresponsible indiscernibility.
✳ No leaked capabilities.
❊ No undefined behavior so far!

```
      inline type::~type()
        {
        •claim destructible( *this );

        •claim proper( *this );
❊        try implementation ○
            {
            •claim unoccupied_space( *this );
❊ ✱ ✤       }
         catch ( ... )
            {
            •claim proper( exception );

            •claim destructible( exception );
            •claim unoccupied_space( *this );
❊ ✱         }
        }
```

When not declared improper.

Only locally {readable, writable, destructible} objects are {read, written, destroyed}.

✤ No irresponsible indiscernibility.
✱ No leaked capabilities.
❊ No undefined behavior so far!

What does propriety mean for compound types?

How does bytewise copying work?
And what's the deal with padding bytes?

How does propriety work for incomplete types? Or polymorphic types?

How do virtual functions share interfaces?

What are the interfaces for **new** and **delete** expressions?

More generally, what are the interfaces for all the built-in operations?

When is a function call repeatable? Or eliminable?

What does repeating or eliminating a function call even mean?

Questions?