The Shape of a Program

Lisa Lippincott
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.
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 interior of a set is the set minus its edges.

The closure of a set is the set plus its edges.
Areas are *adjacent* if they share an edge.

Chains of adjacency *connect* areas together.
Sequence

Branch

Loop
**Forward closed** sets include their exits.

**Backward closed** sets include their entrances.

**Backward open** sets do not include their entrances.

**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 ℓ are alternative possibilities: there is a connection neither from ℓ to ℓ nor from ℓ to ℓ.
If we expand $\mathbb{e}$ and $\mathbb{f}$ to share entrances and exits, they remain alternative possibilities.
Assertions are experiments.

Successful assertions are repeatable, and have no meaningful effect.

An assertion describes its edge, in dimensions of space and possibility.
Some things need to be asserted, but not govern branches:

readable( const T& )
writeable( T& )
destructible( T& )
deallocatable( void *, size_t )
array_deallocatable( void *, size_t )
dynamic_type_identifiable( T& )
dereferencable( Iterator )
reachable( Iterator, Iterator )
reachable( Iterator, Iterator )
resizable( vector<T>& )
reallocatable( vector<T>& )
f closable( int )
in_the_past( time_point<steady_clock> )
proper( T& )
Capabilities can be asserted, but can’t govern branches:

readable( const T& )
writeable( 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>& )
f closable( 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.

An assertion describes its edge, in dimensions of space and possibility.
Claimed assertion
(proof is local)

Posited assertion
(proof is elsewhere)
void bar() {
    ...
    ...
    pre-call region
    ...
    foo();
    ...
    ...
    post-call region
    ...
}

void foo() {
    ...
    ...
    prologue
    ...
    implementation;
    ...
    ...
    epilogue
    ...
}

void foo() implementation {
    {
    ...
    ...
    implementation body
    ...
    
}
A function interface is an experiment in two parts, nestable within itself.
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 )
{
    ...
    implementation;
    ...
    claim deallocatable( p, s );
    ...
}
p = new T;

operator new

deallocatable claimed in epilogue

deallocatable claimed in prologue

operator delete

delete p;
bool b = a;
c = b;

readable claimed in prologue
readable claimed in epilogue
copy initialization
assignment
void readable(const bool& b) {
    claim addressable(b);
    require implementation;
}
void readable( const bool& b )
{
    claim addressable( b );
    require implementation;
}
inline void usable( const bool& b )
{
    require readable( b );
}

void foo( const bool& b )
{
    claim usable( b );
    implementation;
    claim usable( b );
}
inline void usable( const bool& b )
{
    require readable( b );
}

void foo( const bool& b )
{
    claim usable( b );
    implementation;
    claim usable( b );
}
inline void usable(const bool& b)
{
    require readable(b);
    require writable(b);
}

void foo(const bool& b)
{
    claim usable(b);
    require implementation;
    claim usable(b);
}
void writable( bool& b )
{
    claim addressable( b );
    require implementation;
    require readable( b );
}

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

bool implementation neighborhood (partial)

if (true)

if (false)

readable

writable
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?