AST Semantics and Use-After-Move Detection

Ivan Murashko

Contents

1 Leveraging AST semantics **1**
1.1 Detecting move operations 2
1.2 Smart pointer detection 4
1.3 Smart pointer special handling 5

2 Detecting use-after-move **8**
2.1 Marking values as moved-from 8
2.2 Use-after-move detection algorithm 10

3 Conclusion **12**

This article continues from Building the LifetimeCheck Pass. The previous part defined the state model from the lifetime safety profile [1]. Here we focus on the semantic information that makes this model useful for C++: AST attributes, smart pointer recognition, and move operations.

1 Leveraging AST semantics

ClangIR operations carry AST attributes that preserve semantic information. The LifetimeCheck pass uses these through the AST attribute interfaces defined in `ASTAttrInterfaces.td`. This is one of the most important parts of the example: semantic questions are wrapped in named interface methods instead of being scattered as raw AST operations throughout the pass.

To understand how this works, let's see what CIR looks like for a simple program:

C++ source:

```cpp
void test() {
  std::unique_ptr<int> ptr;
  ptr = std::make_unique<int>(42);
  auto ptr2 = std::move(ptr);
}
```

ClangIR (simplified):

```
cir.func @test() {
  %ptr = cir.alloca !cir.ptr<!ty_unique_ptr>
           loc("test.cpp":2:3)
  %val = cir.call @make_unique()
           loc("test.cpp":3:9)
  cir.store %val, %ptr
           loc("test.cpp":3:7)
  %ptr2 = cir.alloca !cir.ptr<!ty_unique_ptr>
           loc("test.cpp":4:8)
  %moved = cir.call @move(%ptr)
           loc("test.cpp":4:15)
  cir.store %moved, %ptr2
           loc("test.cpp":4:8)
  cir.return
}
```

Figure 1: C++ code and the corresponding ClangIR representation

Key observations:

- Each CIR operation has a source location attached to it.
- Types are preserved (`!ty_unique_ptr`).
- Operations are in SSA form (`%ptr`, `%val`, etc.).
- High-level operations (alloca, call, store) match C++ semantics.

1.1 Detecting move operations

To detect if a function parameter is an rvalue reference (T&&), indicating a potential move, the pass uses the following implementation. This demonstrates the key pattern of *type categorization* in move semantics. The pass handles Owner, Pointer, and Value types differently because they have distinct moved-from semantics:

```cpp
void LifetimeCheckPass::checkArgForRValueRef(
    CallOp callOp, unsigned argIdx,
    ASTFunctionDeclInterface funcDecl) {
  // Use interface method instead of direct AST access
  if (!funcDecl.isParamRValueReference(argIdx))
    return;

```

```
8     auto arg = callOp.getArgOperand(argIdx);
9
10    // LoadOp means by-value, not a move - check for moved-from value
11    if (auto loadOp = arg.getDefiningOp<cir::LoadOp>()) {
12      auto srcAddr = loadOp.getAddr();
13      if (isValueTypeMovedFrom(srcAddr)) {
14        checkPointerDeref(srcAddr, callOp.getLoc());
15      }
16      return;
17    }
18
19    // Handle move semantics for address arguments
20    mlir::Value addr = arg;
21    if (!addr.getDefiningOp<cir::AllocaOp>() ||
       ↪  !getPmap().count(addr))
22      return;
23
24    // Type categorization: Owner vs Pointer vs Value
25    // Each category has different moved-from semantics
26
27    if (owners.count(addr)) {
28      // Owner types carrying [[gsl::Owner]]
29      if (getPmap()[addr].count(State::getInvalid()) ||
30          getPmap()[addr].count(State::getNullPtr())) {
31        checkPointerDeref(addr, callOp.getLoc());
32        return;
33      }
34      if (!isSkippableTemporary(addr))
35        markOwnerAsMovedFrom(addr, callOp.getLoc());
36      return;
37    }
38
39    if (ptrs.count(addr)) {
40      // Pointer types (raw pointers, references, iterators)
41      if (getPmap()[addr].count(State::getInvalid())) {
42        checkPointerDeref(addr, callOp.getLoc());
43        return;
44      }
45      if (!isSkippableTemporary(addr))
46        markPointerOrValueTypeAsMovedFrom(addr, callOp.getLoc());
47      return;
48    }
49
```

```
50    // Value types (primitives, structs without pointer semantics)
51    if (getPmap()[addr].count(State::getInvalid())) {
52      checkPointerDeref(addr, callOp.getLoc());
53      return;
54    }
55    if (!isSkippableTemporary(addr))
56      markPointerOrValueTypeAsMovedFrom(addr, callOp.getLoc());
57  }
```

The Pointer branch describes a conservative implementation. It is not the same rule as ordinary raw pointer copy: copying a raw pointer preserves the points-to set, while this rvalue-reference path marks the tracked Pointer-category address moved-from after the invalid-state check.

The interface method `isParamRValueReference` is defined in the AST attribute interface TableGen file:

```
1  InterfaceMethod<"", "bool", "isParamRValueReference",
2                  (ins "unsigned":$paramIdx), [{}],
3    /*defaultImplementation=*/ [{
4      if (paramIdx >= $_attr.getAst()->getNumParams())
5        return false;
6      auto *param = $_attr.getAst()->getParamDecl(paramIdx);
7      return param->getType()->isRValueReferenceType();
8    }]
9  >
```

This approach is better than scattering direct AST access throughout the pass because:

- Interface methods provide a local API for the analysis.
- They handle null checks and edge cases.
- They are reusable across multiple passes.
- They clearly document what AST information is being accessed.

1.2 Smart pointer detection

Standard library smart pointers have special semantics: after a move, they are guaranteed to be null (unlike general owner types). The pass uses an AST interface from the implementation to detect them:

```
// In ASTAttrInterfaces.td
InterfaceMethod<"", "bool", "isSmartPointerOwner", (ins), [{}],
  /*defaultImplementation=*/ [{
    if (!$_attr.getAst()->getDeclContext()->isStdNamespace())
      return false;
    llvm::StringRef name = $_attr.getAst()->getName();
    return name == "unique_ptr" || name == "shared_ptr";
  }]
>
```

Usage in the pass:

```
bool isSmartPointer = false;
if (auto recordTy = mlir::dyn_cast<cir::RecordType>(type)) {
  if (auto astAttr = recordTy.getAst()) {
    isSmartPointer = astAttr.isSmartPointerOwner();
  }
}

if (isSmartPointer) {
  // Some smart pointer methods can be called even after move:
  // get/reset/operator bool for unique_ptr and shared_ptr,
  // release for unique_ptr only.
}
```

1.3 Smart pointer special handling

Smart pointers (`std::unique_ptr`, `std::shared_ptr`) have a well-defined empty state after move in the model used by the ClangIR tests. This allows certain methods to be called safely because they inspect or reset the object without dereferencing the stored pointer:

```
bool LifetimeCheckPass::isSmartPointerSafeMethod(
    llvm::StringRef methodName) {
  // These methods are safe to call on moved-from smart pointers
  // This helper is name-based; release is unique_ptr-only.
  return methodName == "get" ||
         methodName == "release" ||
         methodName == "reset" ||
         methodName == "operator bool";
}
```

It is worth noting that this helper is intentionally simple. It checks the method name, not the precise smart-pointer kind, so `release()` must be read as a `std::unique_ptr` case.[1]

The pass checks method calls on potentially invalid owners:

```
void LifetimeCheckPass::checkOperators(
    CallOp callOp, ASTCXXMethodDeclInterface m) {
  auto addr = getThisParamOwnerCategory(callOp);
  if (!addr)
    return;

  // Check if 'this' is in invalid state
  if (!getPmap()[addr].count(State::getInvalid()))
    return;

  // Special handling for smart pointers
  bool isSafeMethod = false;
  if (isSmartPointerType(addr.getType(), IsSmartPointerTyCache)) {
    std::string methodName = m.getDeclName().getAsString();
    isSafeMethod = isSmartPointerSafeMethod(methodName);
  }

  if (!isSafeMethod) {
    // Unsafe operation on moved-from object
    checkPointerDeref(addr, callOp.getLoc());
  }
}
```

This allows code like:

```
std::unique_ptr<int> ptr = std::make_unique<int>(42);
std::unique_ptr<int> ptr2 = std::move(ptr);
if (ptr) {  // OK: operator bool is safe
  // ...
}
ptr.reset();             // OK: reset is safe
int *p = ptr.get();      // OK: get is safe
int value = *ptr;        // ERROR: operator* requires non-null!
```

[1] The implementation comment for `isSmartPointerSafeMethod()` is broader than the modeled standard-library API: `release()` is a `std::unique_ptr` operation, not a `std::shared_ptr` operation. The helper can be made more precise by making safe-method classification aware of the smart-pointer kind, or by keeping the comment and tests explicitly `std::unique_ptr`-only for `release()`.

Reinitializing a moved-from smart pointer with `reset(new int(...))` is valid C++, but the tests for this pass do not model that reinitialization path. The example above uses `reset()` because it leaves the pointer empty, so the subsequent dereference is genuinely invalid.

The C++ standard specifies these requirements. For `operator*` and `operator->`:

> [unique.ptr.single.observers]
> [util.smartptr.shared.obs]
> "operator* and operator-> require that get() != nullptr"

Method	Applies to	OK?	Behavior on moved-from
Safe methods (allowed on moved-from smart pointers)			
`get()`	unique/shared	Yes	Returns the stored pointer, possibly null
`release()`	unique only	Yes	Releases ownership and returns the pointer
`reset()`	unique/shared	Yes	Clears or replaces the stored pointer
`operator bool`	unique/shared	Yes	Tests whether the stored pointer is non-null
Unsafe methods (require non-null pointer)			
`operator*`	unique/shared	No	Requires a non-null stored pointer
`operator->`	unique/shared	No	Requires a non-null stored pointer

Figure 2: Smart pointer methods: safe vs unsafe on moved-from instances

This distinction is crucial: the pass allows checking a moved-from smart pointer with `if (ptr)` but reports an error for `*ptr`.

2 Detecting use-after-move

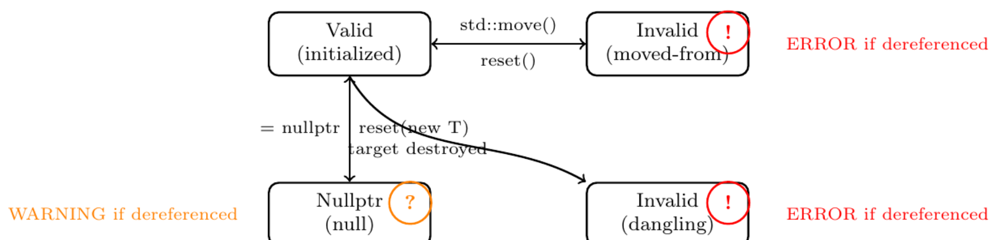

Figure 3: State transitions for tracked values in LifetimeCheck

One of the pass's key responsibilities is detecting use-after-move bugs. Let's trace through the algorithm.

2.1 Marking values as moved-from

When the pass encounters a move operation, it marks the source as invalid:

```
1  void LifetimeCheckPass::checkMovedFromValue(StoreOp storeOp) {
2    auto loadOp = storeOp.getValue().getDefiningOp<cir::LoadOp>();
3    if (!loadOp)
4      return;
5
6    auto srcAddr = loadOp.getAddr();
7
8    // 1. Check if source is already moved-from (use-after-move!)
9    if (isValueTypeMovedFrom(srcAddr)) {
10     checkPointerDeref(srcAddr, storeOp.getLoc());
11     return;
12   }
13
14   // 2. Track rvalue initialization (e.g., int b(std::move(a)))
15   auto destAddr = storeOp.getAddr();
16   auto allocaOp = destAddr.getDefiningOp<cir::AllocaOp>();
17   if (!allocaOp)
18     return;
19
20   // Note: Actual implementation also validates
     ↪  getPmap().count(srcAddr)
21   if (isValueType(destAddr) && isValueType(srcAddr)) {
```

```
22      if (!hasInvalidState(srcAddr) && !loadOp.getIsDeref()) {
23        // Mark source as moved-from
24        markPointerOrValueTypeAsMovedFrom(srcAddr, storeOp.getLoc());
25      }
26    }
27  }
```

Helper methods make the logic clear:

```
1  bool LifetimeCheckPass::isValueType(mlir::Value addr) {
2    return !owners.count(addr) && !ptrs.count(addr);
3  }
4
5  bool LifetimeCheckPass::hasInvalidState(mlir::Value addr) {
6    return getPmap().count(addr) &&
7           getPmap()[addr].count(State::getInvalid());
8  }
9
10 bool LifetimeCheckPass::isValueTypeMovedFrom(mlir::Value addr) {
11   return isValueType(addr) && hasInvalidState(addr);
12 }
```

Note the use of semantic helper methods with descriptive names. This pattern keeps conditionals readable and avoids complex inline expressions.

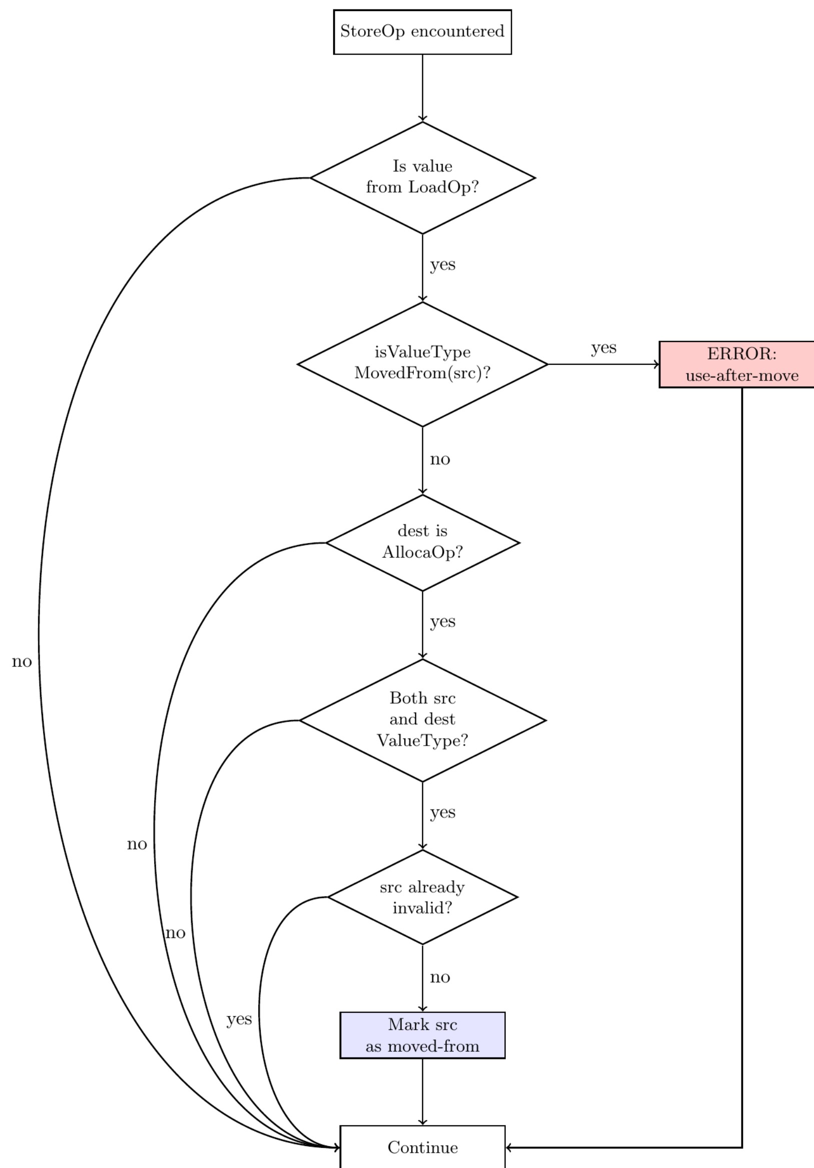

Figure 4: Algorithm flow for `checkMovedFromValue()`

The flowchart in Figure 4 shows two roles of `checkMovedFromValue()`. It detects use-after-move errors and tracks state changes by marking sources as moved-from during initialization.

2.2 Use-after-move detection algorithm

When loading from a moved-from value:

```
void LifetimeCheckPass::checkLoad(LoadOp loadOp) {
  auto addr = loadOp.getAddr();
  if (!getPmap().count(addr))
    return;

  // For pointer types, only check on dereference
  if (ptrs.count(addr)) {
    if (!loadOp.getIsDeref())
      return;
    checkPointerDeref(addr, loadOp.getLoc());
    return;
  }

  // For value types, check if moved-from
  if (isValueTypeMovedFrom(addr)) {
    checkPointerDeref(addr, loadOp.getLoc());
    return;
  }
}
```

The `checkPointerDeref` method emits a diagnostic:

```
void LifetimeCheckPass::checkPointerDeref(mlir::Value addr,
                                          mlir::Location loc,
                                          DerefStyle derefStyle) {
  // ... validation checks ...

  auto varName = getVarNameFromValue(addr);
  auto D = emitWarning(loc);

  bool isValueType = this->isValueType(addr);

  if (isValueType)
    D << "use of moved-from value '" << varName << "'";
  else
    D << "use of invalid pointer '" << varName << "'";

  // Emit history showing where it became invalid
  emitInvalidHistory(D, addr, loc, derefStyle);
}
```

Example diagnostic output:

```
warning: use of moved-from value 'x'
  int value = x;
```

```
3                   ^
4 note: became invalid here
5   auto y = std::move(x);
6              ^
```

Step	C++ code	LifetimeCheck actions
1	`int x = 42;`	**AllocaOp**: Create `x` **StoreOp**: Initialize `x` pmap[x] = {x}, category: Value
2	`int y = std::move(x);`	**CallOp**: Detect std::move call **LoadOp**: Load from `x` **StoreOp**: Store to `y` Action: Mark `x` as moved-from pmap[x] = {invalid}, pmap[y] = {y}
3	`int z = x;`	**LoadOp**: Attempt to load from `x` Check: isValueTypeMovedFrom(x)? YES **DIAGNOSTIC**: use of moved-from value Note: became invalid at step 2

Figure 5: Step-by-step use-after-move detection example

3 Conclusion

Use-after-move detection needs both sides of ClangIR. AST attributes identify the C++ operation, while the LifetimeCheck state model records which value has become invalid. The next article, Control Flow, Scope, and Advanced Lifetime Cases, extends the same model through memory operations, structured control flow, scopes, coroutines, and return values.

References

[1] Herb Sutter. Lifetime safety: Preventing common dangling. C++ Standards Committee Paper P1179R1, ISO/IEC JTC1/SC22/WG21, November 2019.