Building the LifetimeCheck Pass

Ivan Murashko

Contents

1 LifetimeCheck pass architecture **1**
- 1.1 Pass structure and operation visitation 1
- 1.2 Type categories and points-to sets 2
- 1.3 Type classification on allocation 4
- 1.4 Points-to set updates 9
- 1.5 Call operation dispatch 14

2 Conclusion **17**

This article continues the discussion from Static Analysis with ClangIR. We now move from motivation to the concrete pass structure. The goal is to see which state the LifetimeCheck pass keeps, how it classifies values, and how common CIR operations update the analysis.

1 LifetimeCheck pass architecture

1.1 Pass structure and operation visitation

The LifetimeCheck pass `LifetimeCheck.cpp` walks CIR operations and tracks the lifetime state of program variables. The pass is structured as an operation visitor:

```
1 struct LifetimeCheckPass : public
  ↪  LifetimeCheckBase<LifetimeCheckPass> {
2   void runOnOperation() override;
3
4   void checkOperation(Operation *op);
5   void checkStore(StoreOp op);
6   void checkLoad(LoadOp op);
7   void checkCall(CallOp op);
8   // ... other operation handlers
```

```
9
10 private:
11   // State tracking
12   llvm::DenseMap<mlir::Value, PSet> pmap;
13   llvm::DenseSet<mlir::Value> owners;
14   llvm::DenseSet<mlir::Value> ptrs;
15   // ... other state
16 };
```

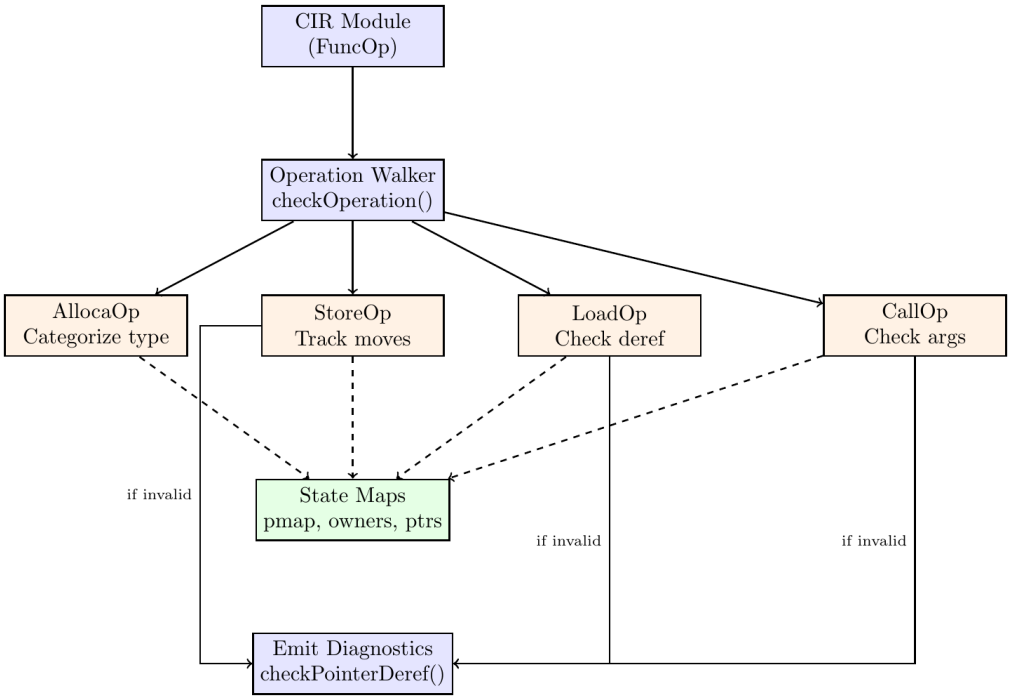

Figure 1: LifetimeCheck pass architecture and data flow

The pass processes each operation in the CIR module, updates state maps (`pmap`, `owners`, `ptrs`), and emits diagnostics when invalid operations are detected.

1.2 Type categories and points-to sets

The pass categorizes every tracked variable into one of three categories, following the lifetime safety profile definitions [1]:

1. **Owner**: Types that own resources and manage their lifetime (`std::unique_ptr`, `std::vector`, `std::string`). When an owner goes out of scope, it destroys what it owns.

2. **Pointer**: Types that reference memory without owning it (raw pointers, references, `std::string_view`, iterators). Pointers become dangling if their target is destroyed.

3. **Value**: Everything else—primitives (`int`, `float`), structs without pointer/owner semantics. Values themselves do not dangle, but they can be in a moved-from (invalid) state.

These are the conceptual categories from the profile. The implementation shown here recognizes owner and pointer record types through attributes such as `[[gsl::Owner(T)]]` and `[[gsl::Pointer(T)]]`. The test smart-pointer types are annotated this way; treating standard containers and smart pointers as if they were annotated is part of the intended direction, not a general rule implemented for every standard type.

Kind	**Examples**	**Lifetime issues**
Owner	`std::unique_ptr<T>`	Can be moved-from (becomes null)
Owner	`std::shared_ptr<T>`	Can be moved-from (becomes null)
Owner	`std::vector<T>`	Can be moved-from (unspecified state)
Owner	`std::string`	Can be moved-from (unspecified state)
Pointer	`T*`	Can dangle if target destroyed
Pointer	`T&`	Can dangle if target destroyed
Pointer	`std::string_view`	Can dangle if string destroyed
Pointer	vector iterator	Can be invalidated
Value	`int`, `float`	Can be moved-from
Value	`bool`, `char`	Can be moved-from
Value	Plain structs	Can be moved-from

Figure 2: Type categories in LifetimeCheck

1.3 Type classification on allocation

When an `AllocaOp` is encountered, the pass must categorize the variable into one of the three categories (Owner, Pointer, or Value) before it can track lifetime state. This classification happens in `classifyAndInitTypeCategories()`, which is called for every local variable declaration.

The classification determines how the variable's lifetime will be tracked throughout its scope. The algorithm follows a decision tree based on the variable's type:

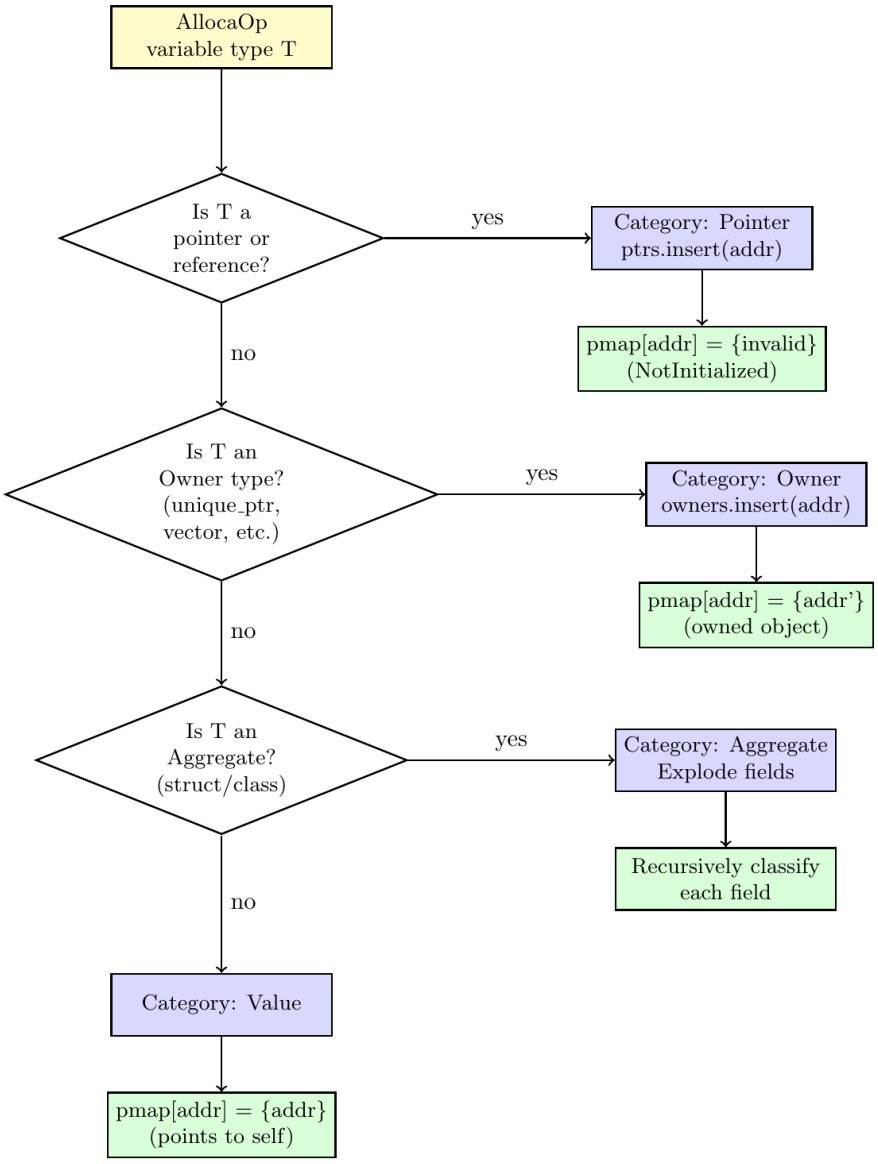

Figure 3: Conceptual type classification flow; this pass recognizes owner and pointer records through GSL-style attributes.

The classification process follows these steps in order:

1. **Check if type is a pointer or reference** — `isPointerType(t)` returns true for raw pointers (`T*`) and references (`T&`). These types reference memory without owning it. They are categorized as **Pointer**, added to the `ptrs` set, and initialized with `pmap[addr] = {invalid}`

to indicate they are uninitialized and must not be dereferenced until assigned.[1]

2. **Check if type is an Owner** — `isOwnerType(t)` returns true for record types carrying the `[[gsl::Owner]]` attribute. Annotated smart-pointer test types and custom owners are categorized as **Owner**, added to the `owners` set, and initialized with `pmap[addr] = {owned_object}` where `owned_object` is represented as `addr'` ("addr prime") indicating the owner manages a distinct resource.

3. **Check if type is an Aggregate** — `isAggregateType(t)` returns true for non-lambda record types that contain pointer-typed members. This is a deliberately narrower implementation rule than the full P1179 aggregate definition. The pass performs field explosion by tracking member addresses obtained from `GetMemberOp` operations. This allows tracking individual fields with pointer semantics within a larger struct. The pass limits explosion to one level deep to avoid excessive complexity.

4. **Default to Value** — All other types (primitives like `int`, `float`, plain structs without special semantics) are categorized as **Value**. They are initialized with `pmap[addr] = {addr}`, meaning the value points to itself.

This categorization is critical because it determines the subsequent tracking behavior:

- **Owners** can be moved-from (becoming invalid) or destroyed (killing all pointers to the owned resource).
- **Pointers** can become dangling when their target is destroyed, and must be checked on every dereference.
- **Values** can be moved-from but cannot dangle (they do not reference external memory).

Here is the implementation pattern from `LifetimeCheck.cpp`, simplified:

```
1 void LifetimeCheckPass::classifyAndInitTypeCategories(
2     mlir::Value addr, mlir::Type t, mlir::Location loc,
3     unsigned nestLevel) {
```

[1] The P1179 specification also categorizes iterators, `std::string_view`, and other Range types as Pointers. However, these are marked as TODO in the implementation and are not enforced by this lifetime checker.

```
    getPmap()[addr] = {};  // Initialize empty pset

    // Determine category based on type
    auto localStyle = [&]() {
      if (isPointerType(t))
        return TypeCategory::Pointer;
      if (isOwnerType(t))
        return TypeCategory::Owner;
      if (isAggregateType(this, t))
        return TypeCategory::Aggregate;
      return TypeCategory::Value;
    }();

    switch (localStyle) {
    case TypeCategory::Pointer:
      // Add to pointer set and mark as uninitialized
      ptrs.insert(addr);
      markPsetInvalid(addr, InvalidStyle::NotInitialized, loc);
      break;

    case TypeCategory::Owner:
      // Add to owner set and initialize with owned object
      addOwner(addr);
      getPmap()[addr].insert(State::getOwnedBy(addr));
      currScope->localValues.insert(addr);
      break;

    case TypeCategory::Aggregate: {
      // Only track first level of aggregate fields
      if (nestLevel > 1)
        break;

      auto members = mlir::cast<cir::RecordType>(t).getMembers();
      // Track fields accessed via GetMemberOp
      std::for_each(addr.use_begin(), addr.use_end(), [&](auto &use) {
        auto op = dyn_cast<cir::GetMemberOp>(use.getOwner());
        if (!op || op.getResult().use_empty())
          return;
        // Recursively classify each field
        auto eltAddr = op.getResult();
        auto eltTy = eltAddr.getType().getPointee();
        classifyAndInitTypeCategories(eltAddr, eltTy, loc,
        ↪  ++nestLevel);
```

```
    });

    // Fallthrough to also treat as Value for aggregate pointers
    LLVM_FALLTHROUGH;
  }
  case TypeCategory::Value:
    // Initialize to point to itself
    getPmap()[addr].insert(State::getLocalValue(addr));
    currScope->localValues.insert(addr);
    break;
  }
}
```

The key insight is that this single categorization decision made at allocation time drives all subsequent lifetime analysis for the variable. For example, once a variable is classified as an Owner, the pass knows to track move operations and invalidate dependent pointers when it goes out of scope.

For each address being tracked, the pass maintains a points-to set (pset) that records what that address points to at that program point:

```
// Maps from address (mlir::Value) to what it points to
llvm::DenseMap<mlir::Value, PSet> pmap;

// PSet can contain:
// - Concrete addresses (what this pointer points to)
// - nullptr (known null)
// - invalid (dangling or moved-from)
```

Example state tracking:

```
int x = 42;
int *p = &x;
int *q = p;

// After these operations:
// pmap[&x] = {x}      // x points to itself (it's a value)
// pmap[p] = {x}       // copying a raw pointer preserves p
// pmap[q] = {x}       // q points to x
```

Statement	Operation	State after execution

`int x = 42;`	AllocaOp + StoreOp	`pmap[x] = {x}`
		
		`owners: {}, ptrs: {}`

`int *p = &x;`	AllocaOp + StoreOp	`pmap[x] = {x}, pmap[p] = {x}` `owners: {}, ptrs: {p}`

`int *q = p;`	LoadOp + StoreOp	`pmap[x] = {x}`
		
		`pmap[p] = {x}`
		`pmap[q] = {x}`
		`owners: {}, ptrs: {p, q}`

`int val = *p;`	Dereference LoadOp	OK: `p` still points to `x`

Figure 4: Points-to set evolution through program execution

This table demonstrates how the pass tracks state changes. Copying the raw pointer preserves both `p` and `q` as pointers to `x`; a later dereference of `p` is therefore still valid. Owner and value moves are the cases where the source may become invalid. This statement is about the ordinary pointer copy shown in the table. The implementation shown here is more conservative for an address passed to an rvalue-reference parameter: after checking whether the Pointer-category value is already invalid, `checkArgForRValueRef()` marks it moved-from.[2]

1.4 Points-to set updates

The core of lifetime tracking is updating points-to sets when values are stored. The `updatePointsTo()` function handles multiple cases depending on the source of the data being stored. Understanding this function is crucial because nearly every assignment operation flows through it.

[2]This conservative Pointer handling can be improved by distinguishing non-owning pointer moves from Owner and Value moves before calling `markPointerOrValueType AsMovedFrom()`.

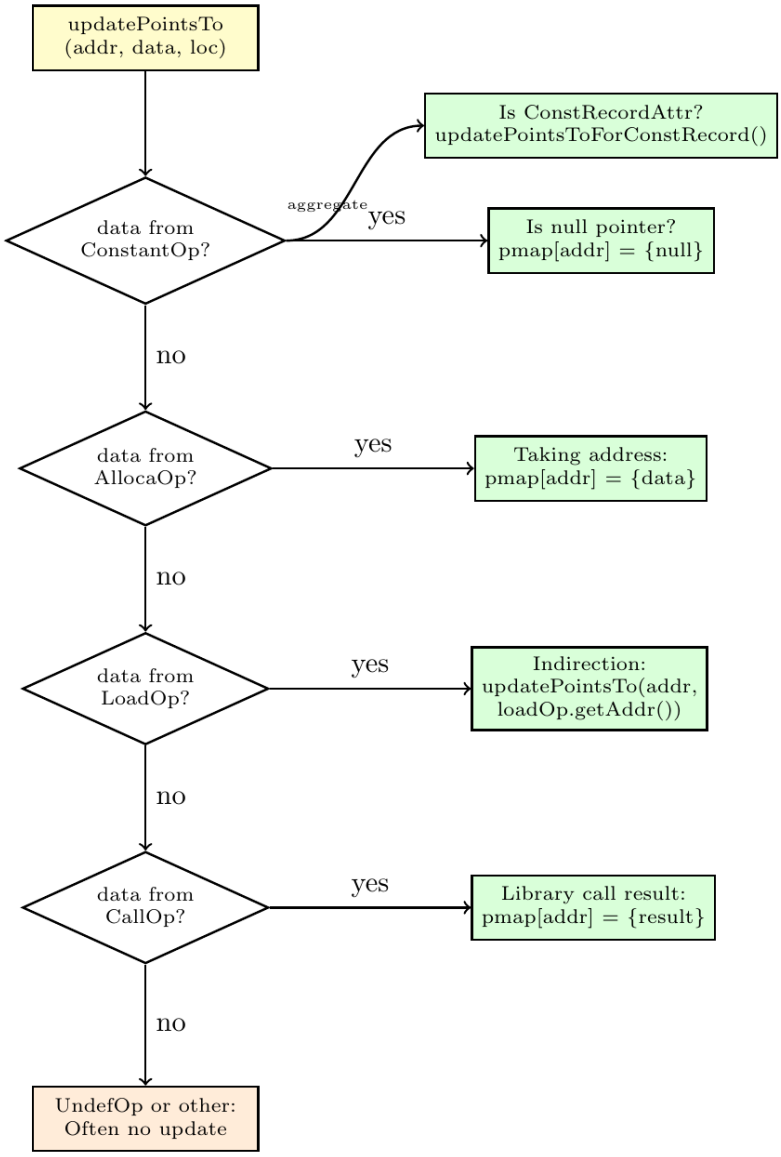

Figure 5: Points-to set update algorithm decision flow in `updatePointsTo()`

The update algorithm handles different data sources:

1. **ConstantOp source** — When storing a constant value:
   - If the constant is a null pointer (`cstOp.isNullPtr()`), set `pmap[addr] = {null}` using `markPsetNull()`.

- If the constant is an aggregate (`ConstRecordAttr`), call `updatePointsToForConstRecord()` to handle memberwise initialization of fields.
- Zero initialization (`ZeroAttr`) for records calls `updatePointsToForZeroRecord()` to set field psets appropriately.

2. **AllocaOp source** — When taking the address of a local variable (`p = &x;`), set `pmap[addr] = {x}`. The pointer now references the local variable.

3. **LoadOp source** — When the data comes from loading another address, recursively call `updatePointsTo(addr, loadOp.getAddr(), loc)`. This handles indirections like copying pointers or passing `this` through temporaries.

4. **CallOp source** — When the data is a function call result (e.g., `iter = vector<T>::begin()`), set `pmap[addr] = {callOp.getResult()}`. This tracks results from methods like container iterators.

5. **Other sources** — Operations like `PtrStrideOp` (array subscript), `GetElementOp` (field access), or undefined values may require special handling or can be safely ignored depending on context.

Each case preserves correctness according to the lifetime safety rules:

- **Owner stores** transfer ownership (e.g., `unique_ptr` move constructor transfers the owned object from source to destination).
- **Pointer stores** copy aliasing relationships (multiple pointers can point to the same object).
- **Value stores** update the value reference (values point to themselves).

Here is the implementation pattern from `LifetimeCheck.cpp`, simplified:

```
void LifetimeCheckPass::updatePointsTo(mlir::Value addr,
                                       mlir::Value data,
                                       mlir::Location loc) {
  auto dataSrcOp = data.getDefiningOp();

  // Handle function arguments (block arguments from entry block)
  if (!dataSrcOp) {
    auto blockArg = cast<BlockArgument>(data);
    if (!blockArg.getOwner()->isEntryBlock())
```

```
10        return;
11      getPmap()[addr].clear();
12      getPmap()[addr].insert(State::getLocalValue(data));
13      return;
14    }
15
16    // Ignore bitcasts and get actual source operation
17    dataSrcOp = ignoreBitcasts(dataSrcOp);
18
19    // Handle constant initialization
20    if (auto cstOp = dyn_cast<ConstantOp>(dataSrcOp)) {
21      // For aggregates, update fields individually
22      if (aggregates.count(addr)) {
23        if (auto constRecord =
24                mlir::dyn_cast<cir::ConstRecordAttr>(cstOp.getValue()
                ↪ ))
                ↪ {
25          updatePointsToForConstRecord(addr, constRecord, loc);
26          return;
27        }
28        if (auto zero =
          ↪ mlir::dyn_cast<cir::ZeroAttr>(cstOp.getValue())) {
29          if (auto zeroRecordTy =
            ↪ dyn_cast<RecordType>(zero.getType())) {
30            updatePointsToForZeroRecord(addr, zeroRecordTy, loc);
31            return;
32          }
33        }
34        return;
35      }
36
37      // Null pointer initialization
38      assert(cstOp.isNullPtr() && "other than null not implemented");
39      markPsetNull(addr, loc);
40      return;
41    }
42
43    // Taking address of local variable: p = &x;
44    if (auto allocaOp = dyn_cast<AllocaOp>(dataSrcOp)) {
45      getPmap()[addr].clear();
46      getPmap()[addr].insert(State::getLocalValue(allocaOp.getAddr())
        ↪ );
47      return;
```

```
48    }
49
50    // Array subscript: p = &a[0];
51    if (auto ptrStrideOp = dyn_cast<PtrStrideOp>(dataSrcOp)) {
52      auto array = getArrayFromSubscript(ptrStrideOp);
53      if (array) {
54        getPmap()[addr].clear();
55        getPmap()[addr].insert(State::getLocalValue(array));
56      }
57      return;
58    }
59
60    // Iterator/pointer from method calls: iter = vec.begin()
61    if (auto callOp = dyn_cast<CallOp>(dataSrcOp)) {
62      getPmap()[addr].clear();
63      getPmap()[addr].insert(State::getLocalValue(callOp.getResult())
      ↪  );
64    }
65
66    // Handle indirections through loads (e.g., temporaries copying
      ↪  'this')
67    if (auto loadOp = dyn_cast<LoadOp>(dataSrcOp)) {
68      updatePointsTo(addr, loadOp.getAddr(), loc);
69      return;
70    }
71  }
```

The key insight is that `updatePointsTo()` translates CIR operations into abstract points-to relationships. For example, when it sees `AllocaOp`, it knows this represents taking an address, so it creates a points-to relationship. When it sees `LoadOp`, it knows this is an indirection that should propagate the pset from the source address.

This abstraction layer allows the rest of the analysis to work with high-level points-to sets rather than low-level IR operations, greatly simplifying the checking logic.

CIR operation	Tracked by	Lifetime action
`AllocaOp`	alloca check	Categorize variable, initialize pset
`StoreOp`	store check	Update psets, check moved-from sources, track rvalue init
`LoadOp`	load check	Check loading from moved-from value or invalid pointer
`LoadOp(isDeref)`	load check	Check pointer dereference validity
`CallOp (move ctor)`	move ctor check	Mark source as moved-from
`CallOp (move assign)`	move assignment	Mark source as moved-from, invalidate dst pointers
`CallOp (rvalue ref)`	rvalue-ref args	Check source, classify category, then mark conservatively
`CallOp (method)`	method dispatch	Check `this`, handle smart pointer safe methods
`IfOp`	branch merge	Merge then/else states conservatively
`ReturnOp`	return check	Verify no dangling references returned

Figure 6: CIR operations and their lifetime checking actions

1.5 Call operation dispatch

The `CallOp` operation is the most complex to analyze because a single function call in C++ can have many different semantics depending on what is being called. A `CallOp` could represent:

- A move constructor (transferring ownership)
- A copy constructor (creating an alias)
- A move assignment operator (transfer + invalidation)
- An operator method on an owner/pointer (`operator*`, `operator->`)
- A regular function with rvalue reference parameters (indicating moves)
- A method call requiring validity checks on `this`
- A coroutine call requiring task tracking

The `checkCall()` method dispatches to specialized handlers based on what the call represents. Understanding this dispatch is crucial because lifetime bugs often occur through function calls.

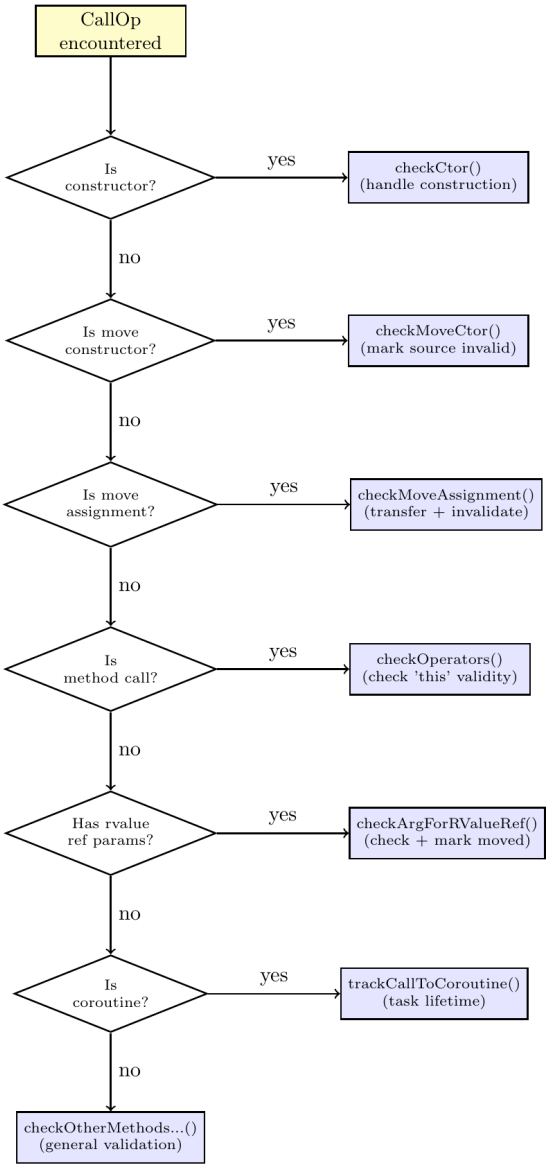

Figure 7: Main checks performed for a `CallOp`

The implementation uses the following dispatch sequence (simplified from `checkCall()`):

1. **Ignore calls without arguments** — calls with no arguments cannot

move or dereference a tracked local value through an argument, so the pass returns early.

2. **Coroutine tracking** — the pass first calls `trackCallToCoroutine()` to record temporary task values returned by coroutine calls.

3. **Rvalue reference parameter check** — the pass calls `checkMoveInCallArgs()`. This helper resolves the callee with `callOp.getDirectCallee(theModule)`, reads the AST function attribute, and then calls `checkArgForRValueRef()` for parameters whose type is `T&&`.

4. **General method/function checks** — if the call is not a method on an owner or pointer category, the pass calls `checkOtherMethodsAndFunctions()` to validate tracked arguments conservatively.

5. **Special member check** — for owner and pointer class methods, the pass resolves the direct callee with `callOp.getDirectCallee(theModule)`. A constructor calls `checkCtor()`, which in turn handles move construction through `checkMoveCtor()`. A move assignment calls `checkMoveAssignment()`, and a copy assignment calls `checkCopyAssignment()`.

6. **Operator and non-const method checks** — overloaded operators are checked with `checkOperators()`. Other non-const owner method calls invalidate the owner's old resource with `checkNonConstUseOfOwner()`.

Key insight: A single `CallOp` may trigger *multiple* checks. For example, a method call with rvalue reference parameters can first be processed by `checkMoveInCallArgs()` and later by owner or pointer method handling. A move constructor is dispatched through `checkCtor()`, and `checkCtor()` calls `checkMoveCtor()` when the constructor attribute says it is a move constructor.

Example code demonstrating multiple dispatch paths:

```
struct Owner {
  std::unique_ptr<int> data;

  // Move constructor: triggers checkCtor + checkMoveCtor
  Owner(Owner&& other) : data(std::move(other.data)) {}

  // operator* requiring 'this' validity: triggers checkOperators
  int& operator*() { return *data; }

```

```
10    // Method with rvalue ref param: triggers checkOperators +
      ↪  checkArgForRValueRef
11    void consume(std::unique_ptr<int>&& ptr) {
12      data = std::move(ptr);
13    }
14  };
15
16  void example() {
17    Owner o1;
18    Owner o2 = std::move(o1);      // Dispatch: checkCtor,
      ↪  checkMoveCtor
19    *o1;                           // Dispatch: checkOperators -> ERROR
20
21    std::unique_ptr<int> p = std::make_unique<int>(42);
22    o2.consume(std::move(p));      // Dispatch: checkOperators,
23                                   //           checkArgForRValueRef
24    *p;                            // ERROR: p moved-from
25  }
```

The dispatch mechanism demonstrates why ClangIR's AST attributes are useful: without them, distinguishing a move constructor from a regular constructor, or identifying rvalue reference parameters, would be extremely difficult at the IR level.

2 Conclusion

The pass architecture gives us the basic language of the analysis: owners, pointers, values, points-to sets, and operation handlers. The next article, AST Semantics and Use-After-Move Detection, shows how this state model becomes precise when we combine it with AST attributes and C++ move semantics.

References

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