Control Flow, Scope, and Advanced Lifetime Cases

Ivan Murashko

Contents

This article finishes the ClangIR lifetime analysis series. We will use the points-to model from Building the LifetimeCheck Pass and the moved-from state tracking from AST Semantics and Use-After-Move Detection. The remaining question is how the same analysis behaves when the program has stores, loads, branches, loops, scopes, coroutines, and return values.

1 Memory operations

Store and load operations are the workhorses of lifetime tracking. Nearly every assignment, initialization, and value use flows through `checkStore()` and `checkLoad()`. Understanding how these operations dispatch to specialized handlers is crucial for understanding the complete picture of lifetime analysis.

1.1 Store operation dispatch

When a `StoreOp` is encountered (representing any assignment or initialization in C++), the pass must determine what kind of store it is and handle it appropriately. Different store patterns have different lifetime semantics.

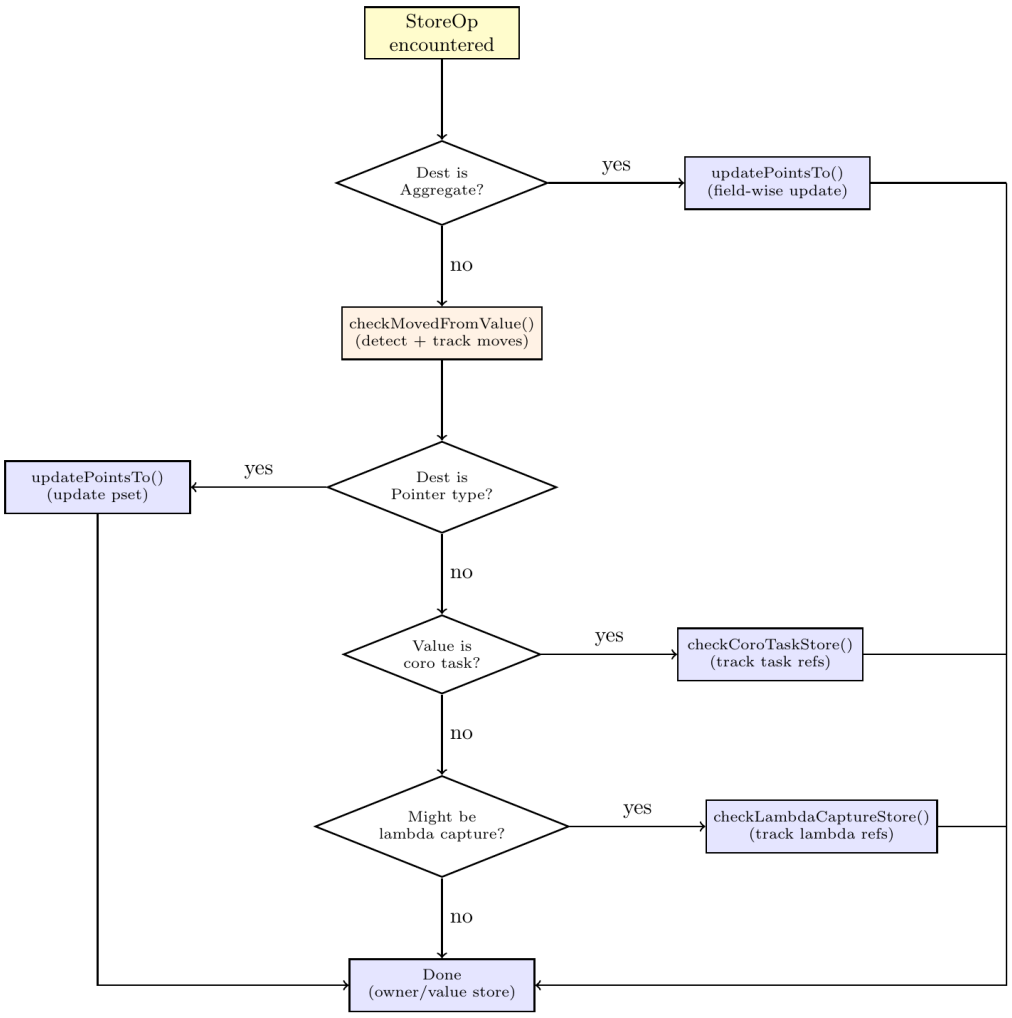

Figure 1: Store operation dispatch decision flow in `checkStore()`

The store dispatch logic follows this order:

1. **Aggregate stores** — If the destination is an aggregate type (struct/class tracked in the `aggregates` map), call `updatePointsTo()` to update individual field psets. This handles bulk initialization like `MyStruct s = {ptr1, ptr2};`.

2. **Moved-from value check** — *Always* call `checkMovedFromValue()` to detect if the source value is moved-from (error) and to track rvalue initialization (state update). This is the dual-purpose check explained in the previous article.

3. **Pointer stores** — If the destination is a pointer type (in the `ptrs` set), call `updatePointsTo()` to update its pset. This is the standard path for assignments like `ptr = &x;`.

4. **Coroutine task stores** — If the stored value is a coroutine task temporary (tracked in `currScope->localTempTasks`), call `checkCoroTaskStore()` to bind local values used in task initialization to the task's pset.

5. **Lambda capture stores** — Otherwise, call `checkLambdaCaptureStore()` to handle potential lambda captures by reference, which can create dangling references if the captured local goes out of scope.

6. **Owner/Value stores** — If none of the above apply, the store is to an owner or value type and requires no special handling (the value is copied/moved as normal).

Implementation pattern from `LifetimeCheck.cpp`, simplified:

```
void LifetimeCheckPass::checkStore(StoreOp storeOp) {
  auto addr = storeOp.getAddr();

  // Handle aggregate stores (field-wise update)
  if (aggregates.count(addr)) {
    auto data = storeOp.getValue();
    if (data.getDefiningOp<cir::ConstantOp>()) {
      updatePointsTo(addr, data, data.getLoc());
    }
    return;
  }

  // Always check for moved-from values (dual purpose: detect + track)
  checkMovedFromValue(storeOp);

  // Special handling for non-pointer types
  if (!ptrs.count(addr)) {
    if (currScope->localTempTasks.count(storeOp.getValue()))
      checkCoroTaskStore(storeOp);
    else
      checkLambdaCaptureStore(storeOp);
    return;
  }

```

```
25    // Standard pointer store: update points-to set
26    updatePointsTo(addr, storeOp.getValue(),
      ↪  storeOp.getValue().getLoc());
27  }
```

Key insight: The *order* of checks matters. Aggregate stores are decomposed first and may return early. For the remaining store cases, the pass checks for a moved-from value before it handles pointer updates, coroutine task stores, or lambda captures, so use-after-move is detected before the store is interpreted as a normal assignment.

1.2 Coroutine task stores

Coroutine tasks are special because they can capture references to local variables that may outlive the coroutine frame. When a task is initialized, the pass tracks which local values are passed as arguments, and binds them to the task's pset.

```
1   void LifetimeCheckPass::checkCoroTaskStore(StoreOp storeOp) {
2     auto taskTmp = storeOp.getValue();
3     auto taskAddr = storeOp.getAddr();
4
5     // Pattern: %tmp_task = cir.call @coroutine_call(%arg0, %arg1,
      ↪  ...)
6     //          cir.store %tmp_task, %task
7     //
8     // Bind local values used as arguments to pset(task)
9     if (auto call = taskTmp.getDefiningOp<cir::CallOp>()) {
10      bool potentialTaintedTask = false;
11      for (auto arg : call.getArgOperands()) {
12        auto alloca = arg.getDefiningOp<cir::AllocaOp>();
13        if (alloca && currScope->localValues.count(alloca)) {
14          // Task now depends on this local value
15          getPmap()[taskAddr].insert(State::getLocalValue(alloca));
16          potentialTaintedTask = true;
17        }
18      }
19    }
20  }
```

This ensures that when a local variable goes out of scope (triggering KILL), any tasks that reference it are also invalidated. Example:

```
Task<int> createTask() {
  int local = 42;
  auto task = asyncCompute(&local);  // task captures local by ref
  // End of scope: KILL(local) also invalidates task
  return task;  // WARNING: task references destroyed local
}
```

1.3 Lambda capture stores

Lambda captures by reference can similarly create dangling references. When a local value is stored into a lambda capture field, the pass binds that local to the lambda's pset:

```
void LifetimeCheckPass::checkLambdaCaptureStore(StoreOp storeOp) {
  auto localByRefAddr = storeOp.getValue();
  auto lambdaCaptureAddr = storeOp.getAddr();

  if (!localByRefAddr.getDefiningOp<cir::AllocaOp>())
    return;
  auto lambdaAddr = getLambdaFromMemberAccess(lambdaCaptureAddr);
  if (!lambdaAddr)
    return;

  // Bind captured local to lambda's pset
  if (currScope->localValues.count(localByRefAddr))
    getPmap()[lambdaAddr].insert(State::getLocalValue(localByRefAdd
    ↪  r));
}
```

Example:

```
auto makeLambda() {
  int local = 42;
  auto lambda = [&local]() { return local + 1; };
  // lambda captures local by reference
  // End of scope: KILL(local) invalidates lambda
  return lambda;  // WARNING: lambda references destroyed local
}
```

1.4 Load operation and dereference

Load operations are simpler than stores—they primarily check validity when dereferencing pointers. The key insight is the `isDeref` flag, which distin-

guishes between loading a pointer value (safe) and dereferencing it (requires validity check).

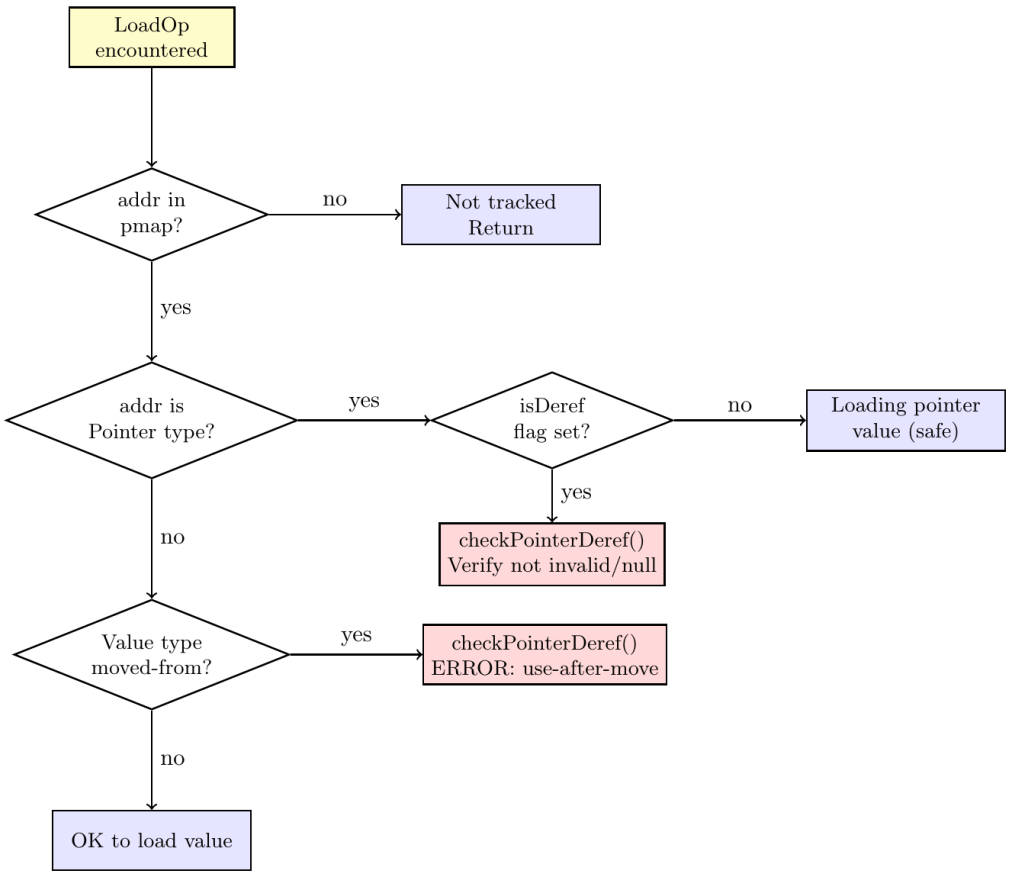

Figure 2: Load operation decision flow in `checkLoad()`

The load logic follows this shape:

1. **Check if tracked** — If the address is not in the pmap, it is not being tracked for lifetime purposes (e.g., it might be a global). Return early.
2. **Pointer type handling** — If the address is a pointer type:
   - If `isDeref` is `false`, this is loading the pointer *value* itself (e.g., copying a pointer), which is safe. Return.
   - If `isDeref` is `true`, this is *dereferencing* the pointer (e.g., `*ptr`). Call `checkPointerDeref()` to verify it is not invalid or null.

3. **Value type handling** — If the address is a value type, check if it is moved-from using `isValueTypeMovedFrom()`. If so, emit a diagnostic for use-after-move.

Implementation pattern from `LifetimeCheck.cpp`, simplified:

```
void LifetimeCheckPass::checkLoad(LoadOp loadOp) {
  auto addr = loadOp.getAddr();
  if (!getPmap().count(addr))
    return;  // Not tracked

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

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

The `isDeref` flag is set by CIRGen based on the C++ operation:

```
int *ptr = ...;
int *ptr2 = ptr;    // LoadOp(ptr, isDeref=false) - loading pointer
↪  value
int value = *ptr;   // LoadOp(ptr, isDeref=true) - dereferencing
↪  pointer
```

This distinction allows the pass to avoid false positives: copying a null pointer is legal, but dereferencing it is not.

2 Control flow analysis

ClangIR provides high-level control flow operations that make branch analysis easier than with raw LLVM IR. While MLIR offers a built-in dataflow analysis framework (`mlir::dataflow::DataFlowSolver`), the

LifetimeCheck pass implements custom control flow analysis for fine-grained control over state tracking and error reporting.

The pass builds on MLIR's base infrastructure, which provides the fundamental Region/Block/Operation hierarchy that structures the IR, along with traversal iterators such as `region.getBlocks()` and `block.getOperations()` for walking through the program structure. MLIR also provides operation type dispatch mechanisms (`isa<>()`, `cast<>()`) and region accessors (`getThenRegion()`, `getElseRegion()`) that enable navigation through control flow constructs. On top of this infrastructure, LifetimeCheckPass implements its own custom logic: it maintains state tracking through the pointer map (pmap) and scope management, implements state merging logic via `joinPmaps()` to handle control flow merge points, and defines the actual lifetime checking algorithms that detect use-after-move and dangling pointer errors.

The pass handles branches by merging state:

```
void LifetimeCheckPass::checkIf(IfOp ifOp) {
  // Collect pmaps from all branches for joining
  llvm::SmallVector<PMapType, 2> pmapOps;

  {
    PMapType localThenPmap = getPmap();
    PmapGuard pmapGuard{*this, &localThenPmap};
    checkRegionWithScope(ifOp.getThenRegion());
    pmapOps.push_back(localThenPmap);
  }

  // In case there's no 'else' branch, use the incoming pmap
  if (!ifOp.getElseRegion().empty()) {
    PMapType localElsePmap = getPmap();
    PmapGuard pmapGuard{*this, &localElsePmap};
    checkRegionWithScope(ifOp.getElseRegion());
    pmapOps.push_back(localElsePmap);
  } else {
    pmapOps.push_back(getPmap());
  }

  joinPmaps(pmapOps);
}
```

The merge operation conservatively handles both paths:

```
// If a pointer is invalid in any branch, it's invalid after the if
if (psetThen.count(State::getInvalid()) ||
```

```
    psetElse.count(State::getInvalid())) {
  mergedPset.insert(State::getInvalid());
}
```

Example:

```
std::unique_ptr<int> ptr = std::make_unique<int>(42);
if (condition) {
  auto ptr2 = std::move(ptr);  // ptr invalid in 'then' branch
} else {
  // ptr still valid in 'else' branch
}
// After merge: ptr is potentially invalid
int value = *ptr;  // WARNING: might be invalid
```

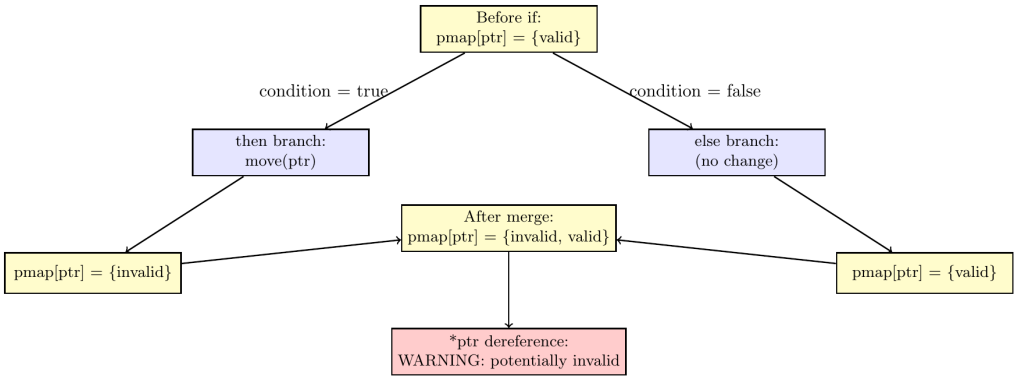

Figure 3: Control flow state merging in if/else branches

The conservative merge strategy ensures that if a variable becomes invalid in *any* branch, it is treated as potentially invalid after the merge. This prevents false negatives but may cause false positives that require user annotations to suppress.

2.1 Loop analysis

Loops present a challenge for lifetime analysis because they may execute zero or more times, and the number of iterations affects what values are valid. The LifetimeCheck pass uses a loop unrolling model that analyzes loops as if they were the first two iterations unrolled into conditional statements.

This approach, specified in P1179 §2.4.9, treats a loop:

```
for (/*init*/; /*cond*/; /*incr*/) {
  /*body*/
}
```

as if it were:

```
if (/*init*/; /*cond*/) {
  /*body*/; /*incr*/
}
if (/*cond*/) {
  /*body*/
}
```

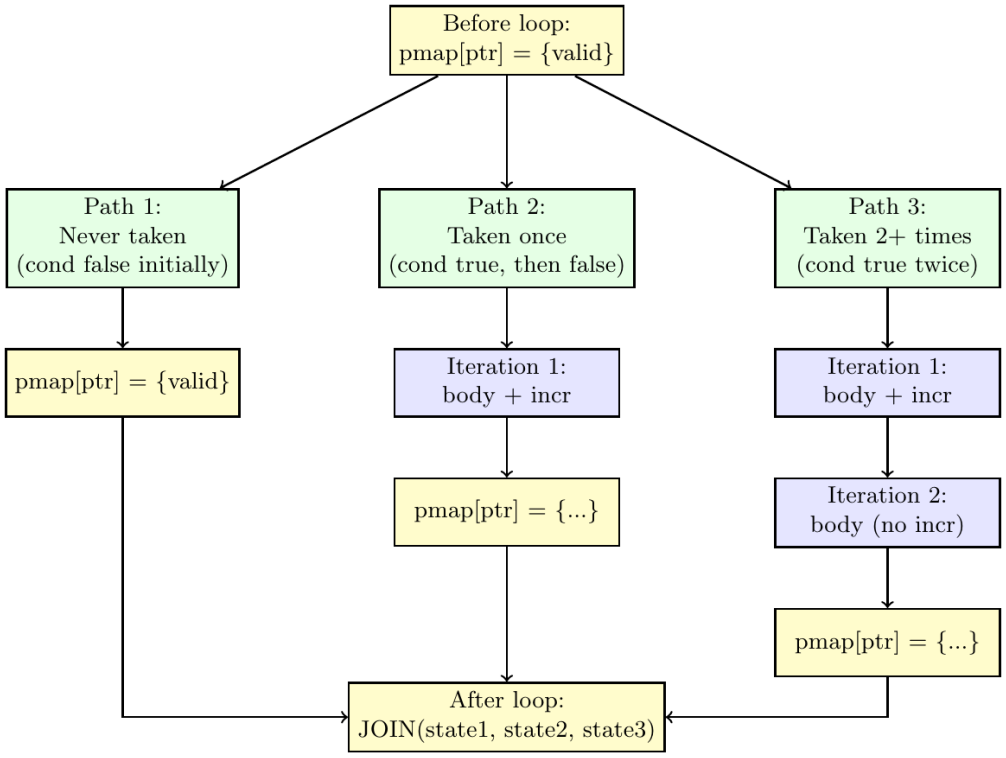

Figure 4: Loop unrolling model: three paths covering 0, 1, and 2+ iterations

The three paths represent:

1. **Never taken** — The loop condition is false from the start, so the body never executes. The pmap remains unchanged from before the loop.

2. **Taken once** — The condition is true initially, so the loop body executes once. In the implementation below, the first-taken path walks the regions in execution order but drops the step region when the loop has one. This is the shape of the pass as implemented here.

3. **Taken twice or more** — The condition is true at least twice. The pass starts from the first-taken exit pmap and checks the execution-order regions again, keeping the step region on this subsequent path.

After analyzing all three paths, the pass joins the resulting pmaps using the JOIN operation (explained below). This conservative approach ensures that:

- If a pointer becomes invalid *in any iteration*, it is treated as potentially invalid after the loop.
- If a loop may execute zero times, the analysis preserves the pre-loop state as one possibility.

Implementation pattern from `LifetimeCheck.cpp`, simplified:

```
void LifetimeCheckPass::checkLoop(LoopOpInterface loopOp) {
  // Treat loop as first two iterations unrolled with if statements
  llvm::SmallVector<PMapType, 4> pmapOps;
  llvm::SmallVector<Region *, 4> regionsToCheck;

  auto setupLoopRegionsToCheck = [&](bool isSubsequentTaken =
  ↪  false) {
    regionsToCheck = loopOp.getRegionsInExecutionOrder();
    if (loopOp.maybeGetStep() && !isSubsequentTaken)
      regionsToCheck.pop_back();
  };

  // Path 1: Never taken
  pmapOps.push_back(getPmap());

  // Path 2: Taken once (condition true, then false)
  PMapType loopExitPmap;
  {
    loopExitPmap = getPmap();
    PmapGuard pmapGuard{*this, &loopExitPmap};
    setupLoopRegionsToCheck();
    for (auto *region : regionsToCheck)
```

```
      checkRegion(*region);
    pmapOps.push_back(loopExitPmap);
  }

  // Path 3: Taken 2+ times (condition true at least twice)
  if (getPmap() != loopExitPmap) {
    PMapType otherTakenPmap = loopExitPmap;
    PmapGuard pmapGuard{*this, &otherTakenPmap};
    setupLoopRegionsToCheck(/*isSubsequentTaken=*/true);
    for (auto *region : regionsToCheck)
      checkRegion(*region);
    pmapOps.push_back(otherTakenPmap);
  }

  // Conservatively merge all three paths
  joinPmaps(pmapOps);
}
```

The exact region choice is an implementation detail of this pass. The important analysis property is that the never-taken pmap, the first-taken pmap, and the subsequent-taken pmap are all joined at the loop exit.

Example demonstrating loop lifetime tracking:

```
std::unique_ptr<int> ptr = std::make_unique<int>(42);
for (int i = 0; i < n; ++i) {
  if (i == 0) {
    auto ptr2 = std::move(ptr);  // ptr invalidated on first
    ↪  iteration
  }
  // After loop merge: ptr potentially invalid
}
*ptr;  // WARNING: pointer might be invalid
```

The analysis correctly identifies that `ptr` may be invalid after the loop because it could be moved on the first iteration (if `n > 0`).

2.2 Switch statement analysis

Switch statements are analyzed similarly to loops: they are transformed into an equivalent series of if-else statements. The pass handles switch statements in simple form—cases with single regions and explicit break or fallthrough behavior.

Per P1179 §2.4.7, a switch:

```
switch (a) {
  case 1: /*1*/
  case 2: /*2*/ break;
  default: /*3*/
}
```

is treated as:

```
if (auto& a=a; a==1) { /*1*/ }
else if (a==1 || a==2) { /*2*/ }
else { /*3*/ }
```

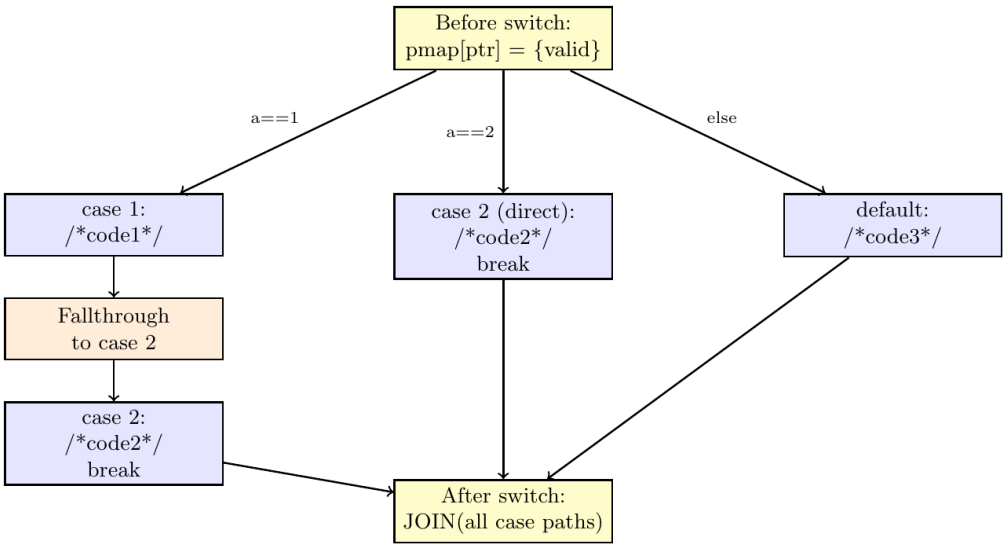

Figure 5: Switch statement analysis with fallthrough handling

Key aspects of switch analysis:

- **Fallthrough semantics** — When a case does not end with `break`, execution falls through to the next case. The pass models this by sequentially checking both case regions and merging the final state.
- **Break semantics** — When a case ends with `break`, execution jumps to after the switch. Each such path contributes independently to the final joined pmap.
- **Default case** — If present, the default case is checked as another independent path. If absent, the incoming pmap (representing "no case matched") is included in the join.

Implementation pattern from `LifetimeCheck.cpp`, simplified:

```
void LifetimeCheckPass::checkSwitch(SwitchOp switchOp) {
  llvm::SmallVector<PMapType, 2> pmapOps;

  // Only handle switch in simple form
  llvm::SmallVector<CaseOp> cases;
  if (!switchOp.isSimpleForm(cases))
    return;

  auto isCaseFallthroughTerminated = [&](Region &r) -> bool {
    Block &block = r.back();
    auto yieldOp = dyn_cast<YieldOp>(block.back());
    return !!yieldOp;
  };

  // Analyze each case
  for (size_t i = 0; i < cases.size(); ++i) {
    auto &caseOp = cases[i];
    PMapType localPmap = getPmap();
    PmapGuard pmapGuard{*this, &localPmap};

    // Check case region
    for (auto &region : caseOp.getRegions())
      checkRegion(region);

    // Handle fallthrough to next case
    if (isCaseFallthroughTerminated(caseOp.getRegion(0))) {
      if (i + 1 < cases.size()) {
        auto &nextCase = cases[i + 1];
        for (auto &region : nextCase.getRegions())
          checkRegion(region);
      }
    }

    pmapOps.push_back(localPmap);
  }

  // Merge all case paths
  joinPmaps(pmapOps);
}
```

Example demonstrating switch lifetime tracking:

```cpp
std::unique_ptr<int> ptr = std::make_unique<int>(42);
switch (value) {
  case 1:
    auto ptr2 = std::move(ptr);  // ptr invalidated in case 1
    break;
  case 2:
    // ptr still valid in case 2
    break;
  default:
    // ptr still valid in default
    break;
}
// After switch: ptr potentially invalid (could have taken case 1)
*ptr;  // WARNING: pointer might be invalid
```

2.3 The JOIN operation

The JOIN operation is the fundamental mechanism for merging program state from multiple control flow paths. It implements a conservative union strategy: if a variable has different states in different branches, the merged state must be conservative enough to cover all possibilities.

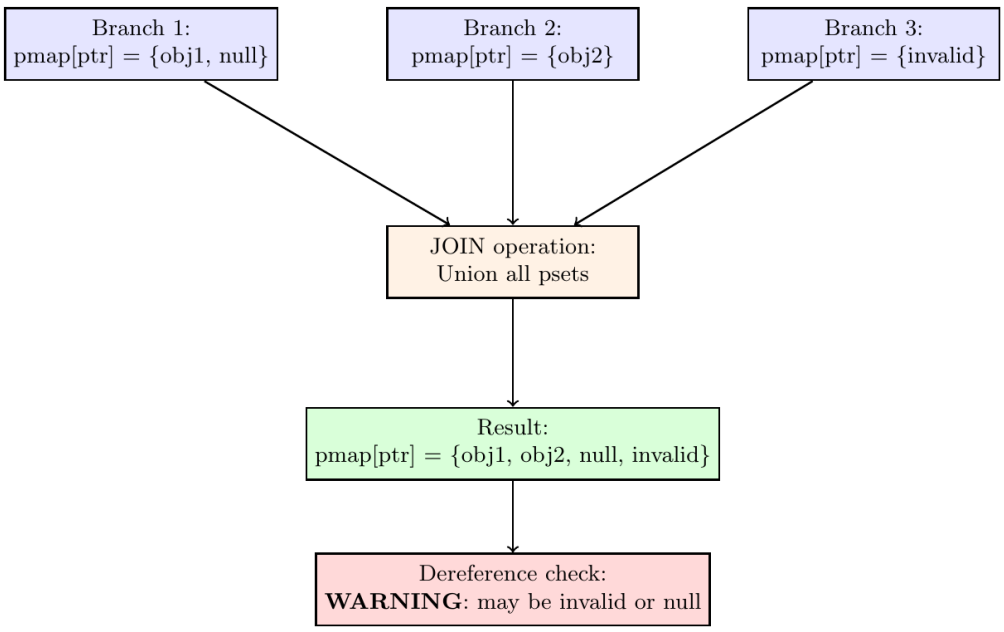

Figure 6: JOIN operation: conservative union of points-to sets from multiple branches

The JOIN algorithm (P1179 §2.3) [2] is simple but powerful:

1. For each tracked address `addr` in the pmap, collect its pset from each branch being merged.
2. Compute the *set union* of all these psets.
3. Replace `pmap[addr]` with the union result.

Implementation pattern from `LifetimeCheck.cpp`:

```
void LifetimeCheckPass::joinPmaps(SmallVectorImpl<PMapType> &pmaps)
↪  {
  for (auto &mapEntry : getPmap()) {
    auto &val = mapEntry.first;

    // Collect pset from each branch
    PSetType joinPset;
    for (auto &pmapOp : pmaps)
      llvm::set_union(joinPset, pmapOp[val]);

    // Update with union of all psets
    getPmap()[val] = joinPset;
  }
}
```

Why this is conservative:

- If a pointer is `{invalid}` in *any* branch, the merged pset contains `invalid`, so any subsequent dereference triggers a warning.
- If a pointer is `{null}` in some branches and `{valid}` in others, the merged pset is `{null, valid}`, indicating potential null.
- If a pointer points to different objects in different branches, the merged pset contains all possibilities.

Example demonstrating JOIN with three branches:

```
int *ptr;
switch (value) {
  case 1: ptr = nullptr; break;          // pmap[ptr] = {null}
  case 2: ptr = &x; break;               // pmap[ptr] = {x}
  case 3: ptr = invalid_ptr; break;      // pmap[ptr] = {invalid}
}
// After JOIN: pmap[ptr] = {null, x, invalid}
*ptr;  // WARNING: may be invalid or null
```

The conservative nature of JOIN is essential for soundness: the analysis would rather emit false positives (warnings for safe code) than false negatives (missing real bugs). This aligns with the C++ Core Guidelines [1] philosophy that lifetime safety analysis should be conservative but suppressible.

3 Scope and lifetime management

One of the fundamental operations in the C++ Core Guidelines lifetime safety profile (P1179) is the KILL operation. Understanding KILL is essential to understanding how the LifetimeCheck pass prevents use-after-free bugs.

3.1 Lexical scope tracking

The pass tracks lexical scopes using a `LexicalScopeContext` stack. Each scope maintains a set of local values declared within it. When a scope ends (e.g., closing brace of a function or block), all local values in that scope must be destroyed, and any pointers referencing them must be invalidated.

This is managed using the RAII pattern with `LexicalScopeGuard`:

```
void LifetimeCheckPass::checkRegionWithScope(Region &region) {
  // Create new scope
  LexicalScopeContext lexScope{currFunc};
  LexicalScopeGuard scopeGuard{*this, &lexScope};

  // Check operations in this scope
  for (auto &block : region)
    checkBlock(block);

  // Scope guard destructor runs here, calling kill() for all locals
}
```

When the `LexicalScopeGuard` destructor runs (at scope exit), it calls `kill()` for each local value that is going out of scope.

3.2 The KILL operation

When a local Owner goes out of scope, the lifetime analysis must invalidate all pointers that reference it or the resource it owns. This is the KILL operation from the C++ Core Guidelines lifetime safety profile [2]:

> KILL(x) means to replace all occurrences of x and x' and x" (etc.) in the pmap with invalid.

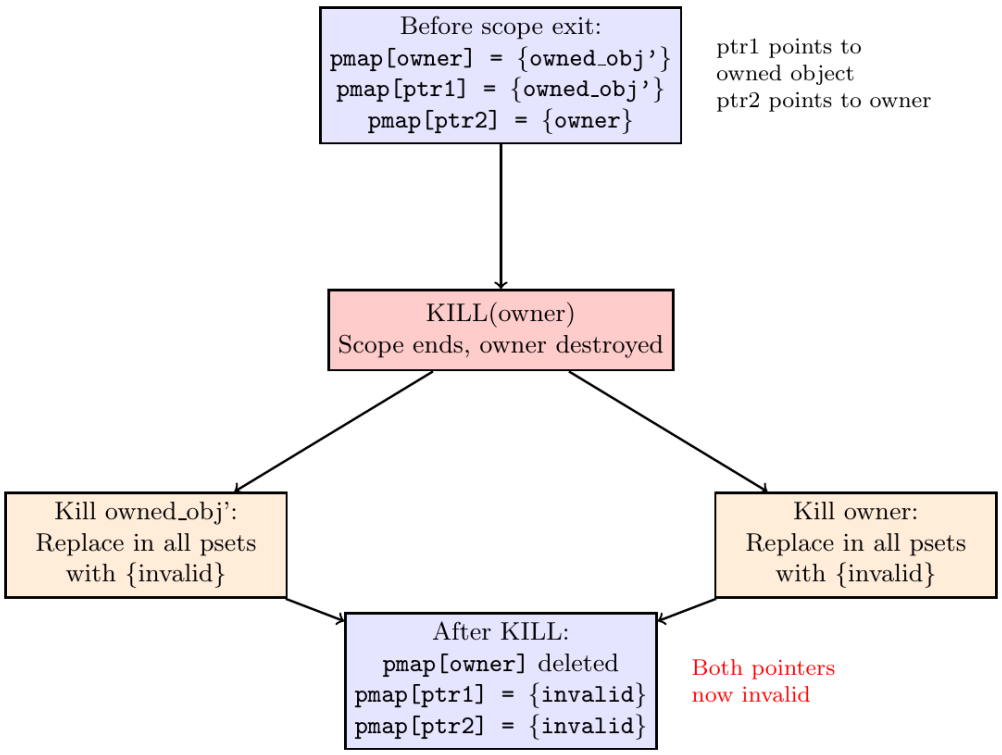

Figure 7: KILL operation cascading invalidation when owner goes out of scope

The KILL operation has cascading semantics:

- When `owner` is killed, both `owner` itself and the owned object `owner'` are invalidated.
- All pointers in the pmap that reference `owner` or `owner'` have their psets updated to `{invalid}`.
- The `owner` is removed from the category tracking sets when its scope ends. Other pmap entries that mention `owner` or `owner'` are rewritten to `{invalid}`.

Implementation pattern from `LifetimeCheck.cpp`, simplified:

```
void LifetimeCheckPass::killInPset(mlir::Value ptrKey,
                                   const State &s,
                                   InvalidStyle invalidStyle,
                                   mlir::Location loc) {
```

```
  auto &pset = getPmap()[ptrKey];
  if (pset.contains(s)) {
    pset.erase(s);
    markPsetInvalid(ptrKey, invalidStyle, loc);
  }
}

// KILL(x): replace all occurrences of x, x', x'' in pmap with
↪  invalid
void LifetimeCheckPass::kill(const State &s,
                             InvalidStyle invalidStyle,
                             mlir::Location loc) {
  assert(s.hasValue() && "does not know how to kill other types");
  mlir::Value v = s.getData();

  for (auto &mapEntry : getPmap()) {
    auto ptr = mapEntry.first;

    // Skip the entry being deleted
    if (v == ptr)
      continue;

    // Replace all occurrences of x' (owned object)
    if (s.isLocalValue() && owners.count(v))
      killInPset(ptr, State::getOwnedBy(v), invalidStyle, loc);

    // Replace all occurrences of x (the value itself)
    killInPset(ptr, s, invalidStyle, loc);
  }

  // Remove scoped local from category tracking sets
  if (invalidStyle == InvalidStyle::EndOfScope) {
    owners.erase(v);
    ptrs.erase(v);
    tasks.erase(v);
  }
}

// LexicalScopeGuard destructor triggers KILL for all locals
void LifetimeCheckPass::LexicalScopeGuard::cleanup() {
  auto *localScope = Pass.currScope;
  for (auto pointee : localScope->localValues)
    Pass.kill(State::getLocalValue(pointee),
```

```
                InvalidStyle::EndOfScope,
                getEndLocForHist(*localScope));
}
```

Example demonstrating KILL in action:

```
void example() {
  std::unique_ptr<int> owner = std::make_unique<int>(42);
  int *ptr1 = owner.get();    // ptr1 -> owned object
  auto *ptr2 = &owner;        // ptr2 -> owner itself

  {
    int *ptr3 = owner.get(); // ptr3 -> owned object
    // End of inner scope: KILL(ptr3)
    // pmap[ptr3] deleted
  }
  // ptr3 no longer exists

  *ptr1;  // Still OK here - owner still alive

  // End of function scope: KILL(owner)
  // This invalidates both ptr1 and ptr2
  // because one points to owner' and one points to owner
}

void dangling_reference() {
  int *dangling;
  {
    std::vector<int> vec = {1, 2, 3};
    dangling = vec.data();
    // End of scope: KILL(vec)
    // pmap[dangling] = {invalid} (points to destroyed resource)
  }
  *dangling; // ERROR: use of invalid pointer
}
```

The KILL operation is fundamental to preventing use-after-free bugs because it ensures that when an object is destroyed, all pointers to it are marked invalid. Any subsequent dereference of these pointers will trigger a diagnostic from `checkPointerDeref()`.

The relationship between KILL and the three type categories is important:

- **Owners** are killed when they go out of scope, which invalidates all pointers to the owned resource.

- **Pointers** are killed when they go out of scope, but this only affects pointers-to-pointers (rare).
- **Values** are killed when they go out of scope, invalidating any pointers directly to the value's address (e.g., `&x` where `x` is an int).

This is why the pass must carefully track which category each variable belongs to—the semantics of KILL depend on it.

4 Coroutines and async code

Coroutines introduce unique lifetime challenges because they can suspend execution and resume later, potentially after local variables have gone out of scope. The LifetimeCheck pass has specialized handling for coroutine operations to detect these bugs.

4.1 Await operation handling

The `co_await` operator suspends the active coroutine and may execute several regions of code: the awaiter's `await_ready()`, `await_suspend()`, and `await_resume()` methods. The pass conservatively analyzes all regions sequentially and joins their resulting states.

Implementation pattern from `LifetimeCheck.cpp`:

```
void LifetimeCheckPass::checkAwait(AwaitOp awaitOp) {
  // Conservative: assume all regions execute sequentially
  llvm::SmallVector<PMapType, 4> pmapOps;

  for (auto r : awaitOp.getRegions()) {
    PMapType regionPmap = getPmap();
    PmapGuard pmapGuard{*this, &regionPmap};
    checkRegion(*r);
    pmapOps.push_back(regionPmap);
  }

  // Join states from all awaiter methods
  joinPmaps(pmapOps);
}
```

This conservative approach ensures that if any awaiter method invalidates a pointer, the merged state reflects that possibility. Example:

```
Task<void> example(std::unique_ptr<int> ptr) {
  co_await suspendPoint();
  // After await: ptr might still be valid (conservative)
  *ptr;  // OK if suspendPoint doesn't move ptr
}
```

4.2 Coroutine task lifetime tracking

Coroutine tasks (the objects returned by coroutine functions) require special lifetime tracking because they can capture references to local variables passed as arguments. When a task is created, the pass tracks which locals are bound to it, ensuring that when those locals are destroyed, the task is invalidated.

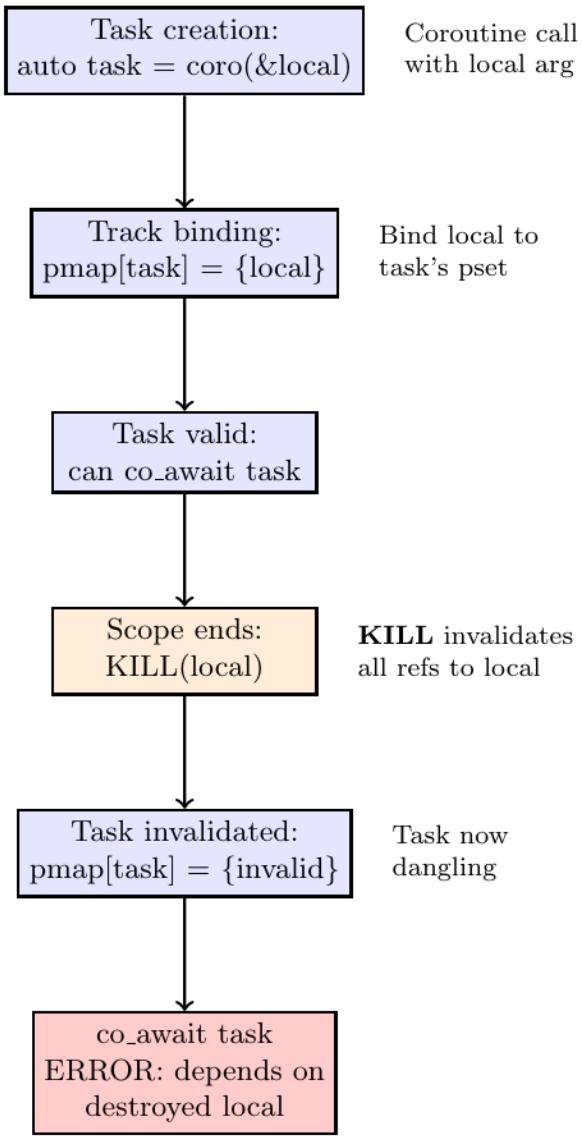

Figure 8: Coroutine task lifetime: tracking captured local references

The task tracking logic is implemented in two parts:

1. Identifying coroutine calls:

```
void LifetimeCheckPass::trackCallToCoroutine(CallOp callOp) {
  if (auto calleeFuncOp = callOp.getDirectCallee(theModule)) {
    if (calleeFuncOp.getCoroutine() ||
        (calleeFuncOp.isDeclaration() && callOp->getNumResults() >
        ↪ 0 &&
```

```
          isTaskType(callOp->getResult(0)))) {
        currScope->localTempTasks.insert(callOp->getResult(0));
      }
      return;
    }
    // Handle indirect calls to coroutines, for instance when
    // lambda coroutines are involved with invokers.
    if (callOp->getNumResults() > 0 &&
    ↪  isTaskType(callOp->getResult(0))) {
      currScope->localTempTasks.insert(callOp->getResult(0));
    }
  }
```

2. Binding locals to task pset (covered earlier in `checkCoroTaskStore()`):

When the task is stored to a variable, any local values passed as arguments are added to the task's pset. When those locals are destroyed (KILL operation), the task becomes invalid.

Common coroutine lifetime bug example:

```
Task<int> dangling_task_example() {
  int local = 42;
  auto task = async_compute(&local);  // task captures &local
  // End of scope: KILL(local)
  // pmap[task] contains local, so task is invalidated too
  return task;  // WARNING: task depends on destroyed local
}

Task<void> use_after_await() {
  Task<int> task = compute();
  co_await std::move(task);
  // task is now moved-from
  int result = co_await task;  // ERROR: use-after-move
}
```

4.3 Temporary vs persistent tasks

Most temporaries in C++ can be ignored for move tracking because they are destroyed at the end of the full expression. However, coroutine task temporaries are an exception because they may be captured by coroutine frames and outlive their expression.

```
bool LifetimeCheckPass::isSkippableTemporary(mlir::Value v) {
  auto allocaOp = v.getDefiningOp<cir::AllocaOp>();
```

```
  if (!allocaOp)
    return false;

  // Temporaries have "ref.tmp" prefix
  auto name = allocaOp.getName();
  if (!name.starts_with("ref.tmp"))
    return false;

  // IMPORTANT: Do not skip coroutine tasks
  // They need lifetime tracking even as temporaries
  if (isTaskType(v))
    return false;

  return true;  // Other temporaries can be skipped
}
```

This distinction allows:

```
// Regular temporary - can skip move tracking
int x = std::move(SomeClass());  // temporary SomeClass() can be
↪  skipped

// Task temporary - must track
auto result = co_await createTask();  // task temporary is tracked
```

4.4 Lambda captures by reference

Lambdas that capture local variables by reference create similar lifetime issues to coroutine tasks. The `checkLambdaCaptureStore()` function (covered earlier) binds captured locals to the lambda's pset.

Common lambda capture bug:

```
std::function<int()> create_lambda() {
  int local = 42;
  auto lambda = [&local]() { return local + 1; };
  // lambda captures local by reference
  // End of scope: KILL(local) invalidates lambda
  return lambda;  // WARNING: lambda captures destroyed local
}

void lambda_use_after_move() {
  std::vector<int> vec = {1, 2, 3};
  auto lambda = [vec = std::move(vec)]() { return vec.size(); };
```

```
  // vec is moved-from
  auto size = vec.size();  // ERROR: use-after-move
}
```

The pass detects these patterns by tracking stores to lambda capture fields and binding the captured locals to the lambda's pset. When the locals go out of scope, the KILL operation propagates through the pset and invalidates the lambda.

5 Return value safety

Returning references or pointers to local variables is a classic C++ bug. The LifetimeCheck pass handles one important form of this problem: returning lambdas that may capture local references. Plain reference and pointer returns are a natural extension point.

Implementation pattern from `LifetimeCheck.cpp`:

```
void LifetimeCheckPass::checkReturn(ReturnOp retOp) {
  // Invalidate all local values on return
  if (retOp.getNumOperands() == 0)
    return;

  auto retTy = retOp.getOperand(0).getType();
  // Currently only handles lambda return types
  if (!isLambdaType(retTy))
    return;

  // The return value is loaded from the return slot
  auto loadOp = retOp.getOperand(0).getDefiningOp<cir::LoadOp>();
  assert(loadOp && "expected cir.load");
  if (!loadOp.getAddr().getDefiningOp<cir::AllocaOp>())
    return;

  // Track lambda for later checking
  // Actual check happens at scope exit (LexicalScopeGuard)
  currScope->localRetLambdas.insert(
      std::make_pair(loadOp.getAddr(), loadOp.getLoc()));
}
```

The clever aspect is that the pass does not check the lambda immediately. Instead, it defers the check until the scope ends (in `LexicalScopeGuard::cl` `eanup()`). This allows it to determine which locals the lambda captured and whether any are being destroyed at scope exit.

From `LexicalScopeGuard::cleanup()`:

```
void LifetimeCheckPass::LexicalScopeGuard::cleanup() {
  auto *localScope = Pass.currScope;
  // Kill all local values going out of scope
  for (auto pointee : localScope->localValues)
    Pass.kill(State::getLocalValue(pointee),
    ↪  InvalidStyle::EndOfScope,
              getEndLocForHist(*localScope));

  // Check returned lambdas for dangling references
  for (auto l : localScope->localRetLambdas)
    Pass.checkPointerDeref(l.first, l.second,
    ↪  DerefStyle::RetLambda);
}
```

Example demonstrating return safety checking:

```
auto return_dangling_lambda() {
  int local = 42;
  auto lambda = [&local]() { return local; };
  return lambda;  // WARNING: lambda captures local by reference
  // At scope exit:
  //   1. KILL(local) invalidates all refs to local
  //   2. checkPointerDeref on lambda detects it's invalid
}

int& return_reference_to_local() {
  int local = 42;
  return local;  // BUG: returning reference to local
  // Future work for this pass: diagnose plain reference returns
  ↪  too.
}
```

The implementation focuses on lambda returns because they are common in modern C++ (especially with std::function and callbacks). One natural extension would be to detect all dangling reference returns, not just lambdas.

6 Implementation patterns and best practices

During implementation, several patterns emerged that improve code quality and maintainability.

6.1 Semantic helper methods

Instead of complex inline conditions, extract logic into methods with semantic names:

```
// Bad: hard to understand at a glance
if (getPmap().count(srcAddr) &&
    getPmap()[srcAddr].count(State::getInvalid()) &&
    !owners.count(srcAddr) &&
    !ptrs.count(srcAddr)) {
  // ...
}

// Good: clear intent
if (isValueTypeMovedFrom(srcAddr)) {
  // ...
}
```

6.2 Composable building blocks

Build complex checks from simple, reusable pieces:

```
// Atomic checks
bool isValueType(mlir::Value addr);
bool hasInvalidState(mlir::Value addr);

// Composed checks
bool isValueTypeMovedFrom(mlir::Value addr) {
  return isValueType(addr) && hasInvalidState(addr);
}
```

6.3 API symmetry

Maintain parallel naming for related operations:

```
// For owner types
void markOwnerAsMovedFrom(mlir::Value addr, mlir::Location loc);

// For pointer/value types (added for symmetry)
void markPointerOrValueTypeAsMovedFrom(mlir::Value addr,
                                       mlir::Location loc);
```

6.4 Early-exit pattern

Reduce nesting with early returns:

```
// Before: deeply nested
if (auto allocaOp = getDefiningOp<AllocaOp>()) {
  if (isValueType(addr)) {
    if (!hasInvalidState(addr)) {
      // actual logic buried here
    }
  }
}

// After: early exits
auto allocaOp = getDefiningOp<AllocaOp>();
if (!allocaOp)
  return;
if (!isValueType(addr))
  return;
if (hasInvalidState(addr))
  return;
// actual logic at top level
```

Refactoring pattern	**Before**	**After**	**Benefit**
Semantic helper methods	4-line condition	1-line call	Readability
Duplicate code elimination	8+ locations	1 method	Maintainability
Early-exit pattern	3-level nesting	Flat structure	Clarity
API symmetry	Inconsistent names	Parallel naming	Discoverability
Composable helpers	Monolithic checks	Building blocks	Reusability

Figure 9: Refactoring patterns and their impact

These patterns are small, but they matter for an analysis pass. Most checks are not complicated individually; the difficulty is keeping many small cases consistent. Semantic helper methods and symmetric APIs make it easier to review the pass and to extend it later.

7 Limitations and future work

The LifetimeCheck pass has known limitations:

- **Interprocedural analysis**: The pass analyzes each function independently. It does not track lifetime across function boundaries, which can lead to false negatives when a dangling pointer is returned from a function.
- **Temporary detection**: The pass uses a string prefix (`ref.tmp`) to identify temporaries. A more robust approach would be adding an `is_temporary` attribute to AllocaOp during CIRGen.
- **Field-sensitive analysis**: The pass tracks first-level aggregate fields that are actually reached through `GetMemberOp` uses. Nested aggregate fields and unused fields are not fully modeled.
- **Custom smart pointers**: The special smart-pointer path recognizes only `unique_ptr` and `shared_ptr` names. User-defined smart pointers following the same patterns are not handled specially.

Future improvements could address these by:

- Adding summary information to function signatures.
- Enhancing CIRGen to mark temporaries explicitly.
- Implementing field-sensitive tracking using CIR's GetMemberOp.
- Allowing user annotations for custom smart pointer types.

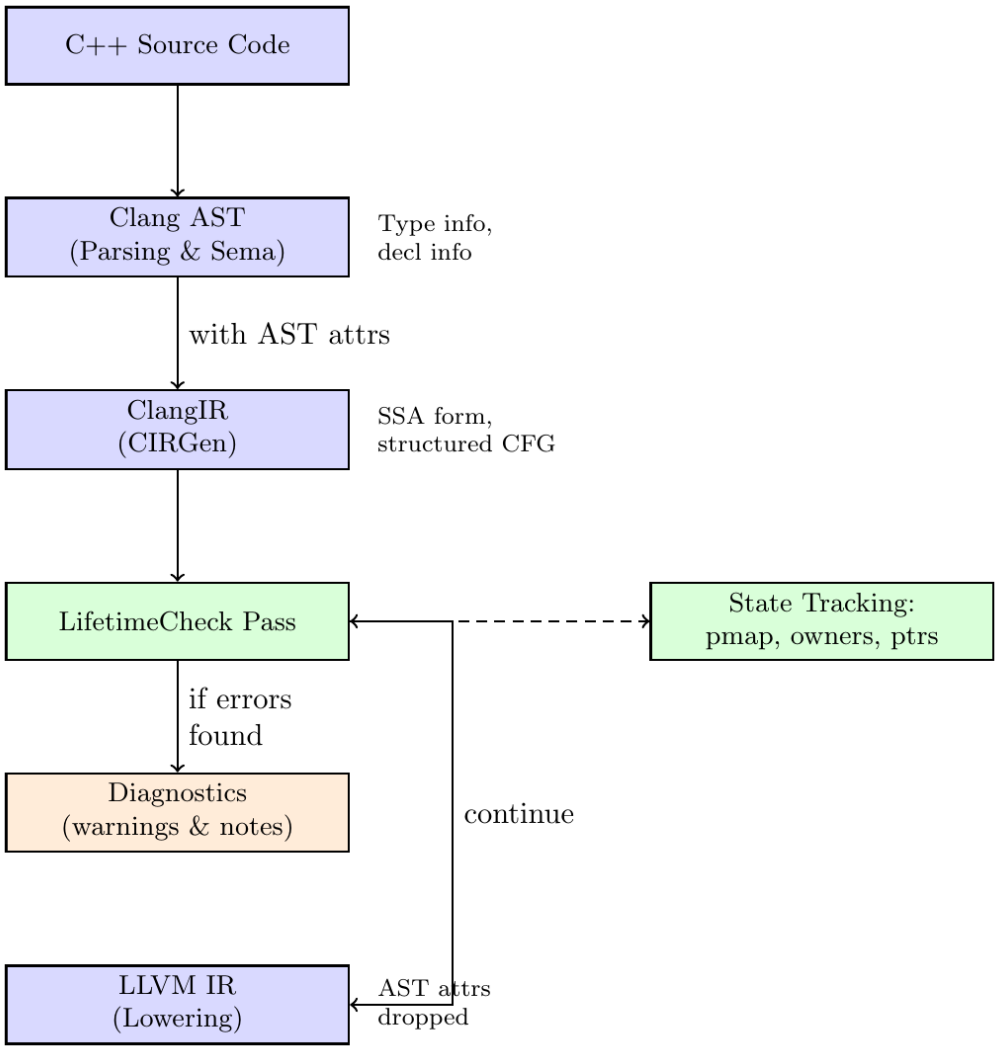

Figure 10: Workflow from C++ source to lifetime diagnostics

The LifetimeCheck pass operates on ClangIR after CIRGen and before lowering to LLVM IR. This is the useful point in the pipeline: the pass still has access to AST semantic information, but it can also use IR structure for data-flow reasoning.

8 Lessons learned

Studying the LifetimeCheck pass demonstrates several advantages of ClangIR for compiler analysis:

1. **AST attributes preserve semantics**: Access to AST information through interfaces makes it easy to answer questions like "Is this parameter an rvalue reference?" or "Is this type a std::unique_ptr?" that would be difficult or impossible with LLVM IR.

2. **High-level operations clarify intent**: CIR's StoreOp, LoadOp, and CallOp carry more semantic meaning than LLVM's generic instructions, making the analysis code more readable and maintainable.

3. **SSA simplifies dataflow**: Unlike AST where variables can be reassigned, CIR's SSA form makes it easy to track value flow. Each `mlir::Value` has exactly one definition.

4. **Structured control flow**: CIR's IfOp, SwitchOp, and LoopOp are easier to analyze than LLVM's unstructured basic block CFG. Branch merging logic is straightforward.

5. **Location tracking**: MLIR requires every operation to have a source location, making diagnostic emission trivial compared to LLVM IR where location metadata is optional.

The LifetimeCheck pass serves as a practical template for building CIR analysis passes. The patterns demonstrated—AST attribute access, state tracking, operation visitation, and diagnostic emission—apply broadly to many kinds of static analysis tasks.

References

[1] Bjarne Stroustrup and Herb Sutter. C++ Core Guidelines. https://isocpp.org/guidelines, 2025. Lifetime Safety Profile: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-lifetime.

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