Library optimizations

Unlocking High Performance in Mojo through User-Defined Dialects

Mathieu Fehr, Jeff Niu, Tobias Grosser

How does clang optimize uses of the stdlib?

```cpp
void foo() {
    std::string a = "A long long long long string";
}
```

```cpp
int bar(std::vector<int>& v) {
  v.push_back(42);
  int res = v.back();
  v.pop_back();
  return res;
}
```

How does clang optimize uses of the stdlib?

```
void foo() {
    std::string a = "A long long long long string";
}
```

How does clang optimize uses of the stdlib?

```
void foo() {
    std::string a = "A long long long long string";
}
```

```
void foo() {
}
```

How does clang optimize uses of the stdlib?

```cpp
void foo() {
    std::string a = "A long long long long string";
}
```

```cpp
void foo() {
}
```

```asm
foo():
        push    rbx
        sub     rsp, 48
        lea     rbx, [rsp + 32]
        mov     qword ptr [rsp + 16], rbx
        mov     qword ptr [rsp + 8], 28
        lea     rdi, [rsp + 16]
        lea     rsi, [rsp + 8]
        xor     edx, edx
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>
                    ::_M_create(unsigned long&, unsigned long)@PLT
        mov     qword ptr [rsp + 16], rax
        mov     rcx, qword ptr [rsp + 8]
        mov     qword ptr [rsp + 32], rcx
        movups  xmm0, xmmword ptr [rip + .L.str+12]
        movups  xmmword ptr [rax + 12], xmm0
        movups  xmm0, xmmword ptr [rip + .L.str]
        movups  xmmword ptr [rax], xmm0
        mov     qword ptr [rsp + 24], rcx
        mov     rax, qword ptr [rsp + 16]
        mov     byte ptr [rax + rcx], 0
        mov     rdi, qword ptr [rsp + 16]
        cmp     rdi, rbx
        je      .LBB0_2
        mov     rsi, qword ptr [rsp + 32]
        inc     rsi
        call    operator delete(void*, unsigned long)@PLT
.LBB0_2:
        add     rsp, 48
        pop     rbx
        ret

.L.str:
        .asciz  "A long long long long string"
```

How does clang optimize uses of the stdlib?

```
void foo() {
    std::string a = "A long long long long string";
}
```

↓

```
void foo() {
}
```

This call could do anything

COMPILER EXPLORER

```
foo():
        push    rbx
        sub     rsp, 48
        lea     rbx, [rsp + 32]
        mov     qword ptr [rsp + 16], rbx
        mov     qword ptr [rsp + 8], 28
        lea     rdi, [rsp + 16]
        lea     rsi, [rsp + 8]
        xor     edx, edx
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>
                   ::_M_create(unsigned long&, unsigned long)@PLT
        mov     qword ptr [rsp + 16], rax
        mov     rcx, qword ptr [rsp + 8]
        mov     qword ptr [rsp + 32], rcx
        movups  xmm0, xmmword ptr [rip + .L.str+12]
        movups  xmmword ptr [rax + 12], xmm0
        movups  xmm0, xmmword ptr [rip + .L.str]
        movups  xmmword ptr [rax], xmm0
        mov     qword ptr [rsp + 24], rcx
        mov     rax, qword ptr [rsp + 16]
        mov     byte ptr [rax + rcx], 0
        mov     rdi, qword ptr [rsp + 16]
        cmp     rdi, rbx
        je      .LBB0_2
        mov     rsi, qword ptr [rsp + 32]
        inc     rsi
        call    operator delete(void*, unsigned long)@PLT
.LBB0_2:
        add     rsp, 48
        pop     rbx
        ret

.L.str:
        .asciz  "A long long long long string"
```

How does clang optimize uses of the stdlib?

```
foo():
        ret
```

```
void foo() {
    std::string a = "A long long long long string";
}
```

```
void foo() {
}
```

How does clang optimize uses of the stdlib?

```
int bar(std::vector<int>& v) {
  v.push_back(42);
  int res = v.back();
  v.pop_back();
  return res;
}
```

How does clang optimize uses of the stdlib?

```
int bar(std::vector<int>& v) {
  v.push_back(42);
  int res = v.back();
  v.pop_back();
  return res;
}
```

⬇

```
int bar(std::vector<int>& v) {
  v.push_back(42);
  v.pop_back();
  return 42;
}
```

How does clang optimize uses of the stdlib?

```cpp
int bar(std::vector<int>& v) {
  v.push_back(42);
  int res = v.back();
  v.pop_back();
  return res;
}
```

⬇

```cpp
int bar(std::vector<int>& v) {
  return 42;
}
```

How does clang optimize uses of the stdlib?

```cpp
int bar(std::vector<int>& v) {
  v.push_back(42);
  int res = v.back();
  v.pop_back();
  return res;
}
```

```cpp
int bar(std::vector<int>& v) {
  return 42;
}
```

```asm
bar(std::vector<int, std::allocator<int>>&):
        push    r15
        push    r14
        push    r13
        push    r12
        push    rbx
        mov     rbx, qword ptr [rdi + 8]
        cmp     rbx, qword ptr [rdi + 16]
        je      .LBB0_2
        mov     dword ptr [rbx], 42
        jmp     .LBB0_8
.LBB0_2:
        mov     r14, qword ptr [rdi]
        sub     rbx, r14
        movabs  rax, 9223372036854775804
        cmp     rbx, rax
        je      .LBB0_9
        mov     r12, rdi
        mov     rax, rbx
        sar     rax, 2
        cmp     rax, 1
        mov     rcx, rax
        adc     rcx, 0
        lea     r13, [rcx + rax]
        movabs  rdx, 2305843009213693951
        cmp     r13, rdx
        cmovae  r13, rdx
        add     rcx, rax
        cmovb   r13, rdx
        lea     rdi, [4*r13]
        call    operator new(unsigned long)@PLT
        mov     r15, rax
        mov     dword ptr [rax + rbx], 42
        test    rbx, rbx
        jle     .LBB0_5
        mov     rdi, r15
        mov     rsi, r14
        mov     rdx, rbx
        call    memcpy@PLT
.LBB0_5:
        test    r14, r14
        je      .LBB0_7
        mov     rdi, r14
        mov     rsi, rbx
        call    operator delete(void*, unsigned long)@PLT
.LBB0_7:
        add     rbx, r15
        mov     rdi, r12
        mov     qword ptr [r12], r15
        lea     rax, [r15 + 4*r13]
        mov     qword ptr [r12 + 16], rax
.LBB0_8:
        mov     eax, dword ptr [rbx]
        mov     qword ptr [rdi + 8], rbx
        pop     rbx
        pop     r12
        pop     r13
        pop     r14
        pop     r15
        ret
.LBB0_9:
        lea     rdi, [rip + .L.str]
        call    std::__throw_length_error(char const*)@PLT

.L.str:
        .asciz  "vector::_M_realloc_append"
```

How does clang optimize uses of the stdlib?

```cpp
int bar(std::vector<int>& v) {
  v.push_back(42);
  int res = v.back();
  v.pop_back();
  return res;
}
```

```cpp
int bar(std::vector<int>& v) {
  return 42;
}
```

```cpp
int bar(std::vector<int>& v) {
  if (v.capacity() == v.size())
    v.double_capacity();
  return res;
}
```

How does clang optimize uses of the stdlib?

```
int bar(std::vector<int>& v) {
  v.push_back(42);
  int res = v.back();
  v.pop_back();
  return res;
}
```

```
int bar(std::vector<int>& v) {
  return 42;
}
```

```
int bar(std::vector<int>& v) {
  if (v.capacity() == v.size())
    v.double_capacity();
  return res;
}
```

Cannot be optimized away even when inlined

How does clang optimize uses of the stdlib?

```
int bar(std::vector<int>& v) {
  v.push_back(42);
  int res = v.back();
  v.pop_back();
  return res;
}
```

```
int bar(std::vector<int>& v) {
  return 42;
}
```

ILLEGAL

```
int bar(std::vector<int>& v) {
  if (v.capacity() == v.size())
    v.double_capacity();
  return res;
}
```

Cannot be optimized away even when inlined

Many potentially illegal optimization opportunities

```
if (!hm.contains(x))
  hm[x] = 42;
```

Many potentially illegal optimization opportunities

Many potentially illegal optimization opportunities

```
hm.try_emplace(x, 42)
```

```
x * y + z
```

Many potentially illegal optimization opportunities

```
hm.try_emplace(x, 42)
```

```
fma(x, y, z)
```

Many potentially illegal optimization opportunities

```
hm.try_emplace(x, 42)
```

```
fma(x, y, z)
```

```
sort(v.begin(), v.end());
auto it = find(v.begin(), v.end(), x);
bool x = it != v.end();
```

Many potentially illegal optimization opportunities

```
hm.try_emplace(x, 42)
```

```
fma(x, y, z)
```

```
sort(v.begin(), v.end());
bool x = binary_search(v.begin(), v.end(), x);
```

Many potentially illegal optimization opportunities

```
hm.try_emplace(x, 42)
```

```
fma(x, y, z)
```

```
sort(v.begin(), v.end());
bool x = binary_search(v.begin(), v.end(), x);
```

```
for (int i = 0; i < N; i++)
    v.push_back(i)
```

Many potentially illegal optimization opportunities

```
hm.try_emplace(x, 42)
```

```
fma(x, y, z)
```

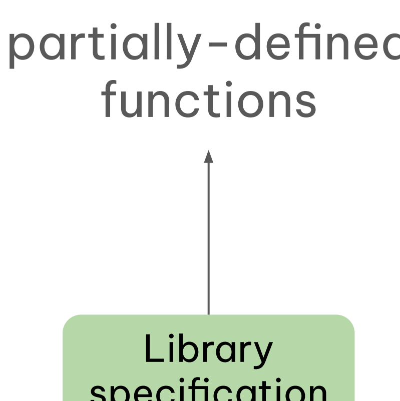
```
sort(v.begin(), v.end());
bool x = binary_search(v.begin(), v.end(), x);
```

```
v.reserve(v.size() + N);
for (int i = 0; i < N; i++)
   v.push_back(i)
```

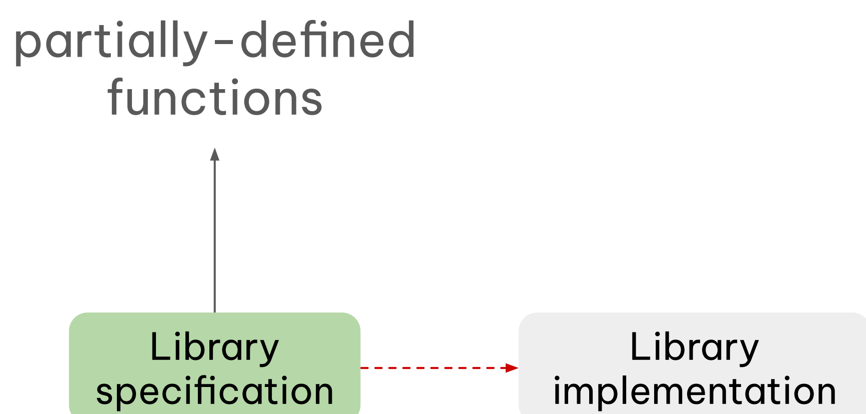
Information is lost too early

Library specification

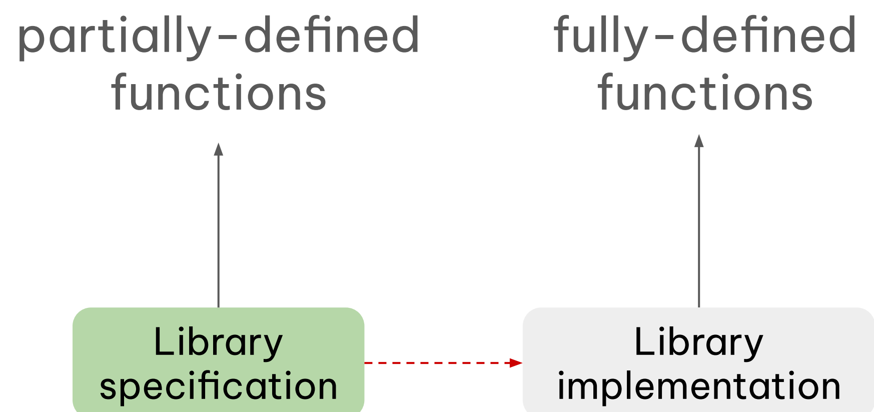
Information is lost too early



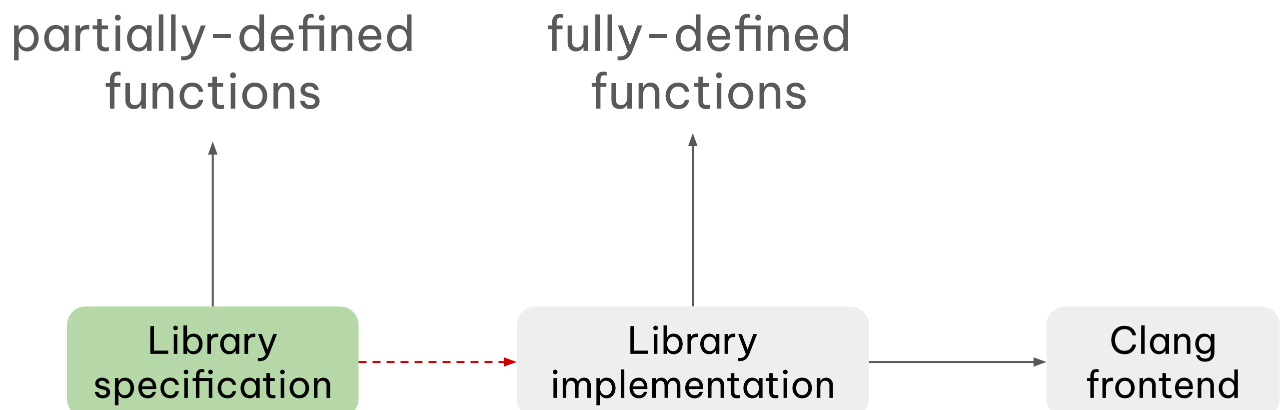
Information is lost too early



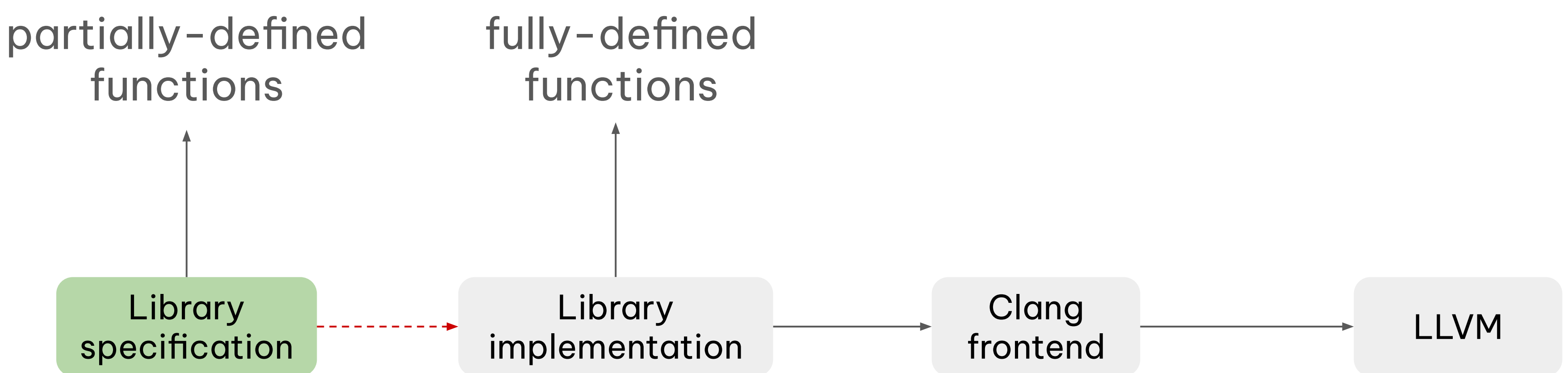
Information is lost too early



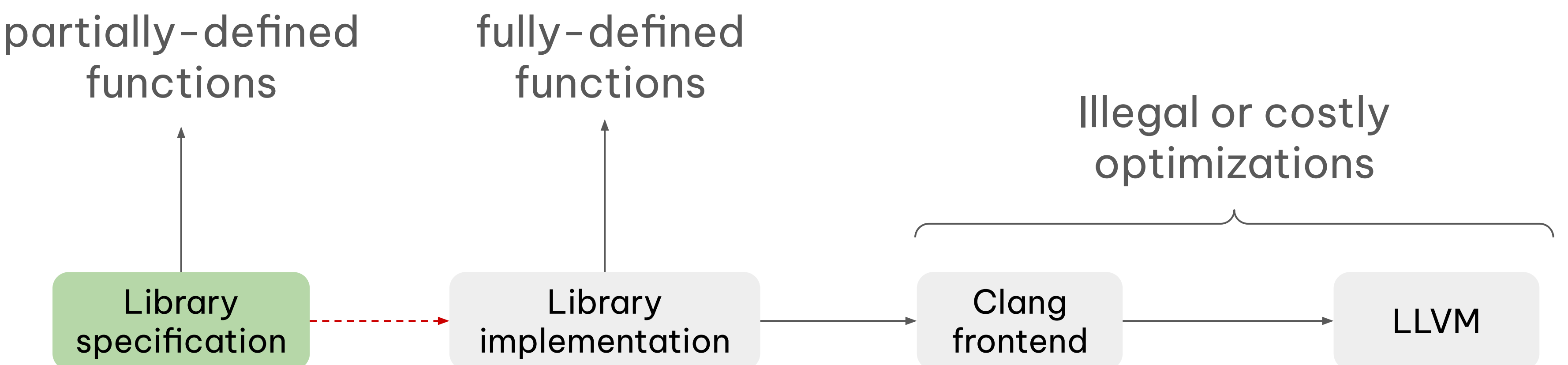
Information is lost too early



Information is lost too early

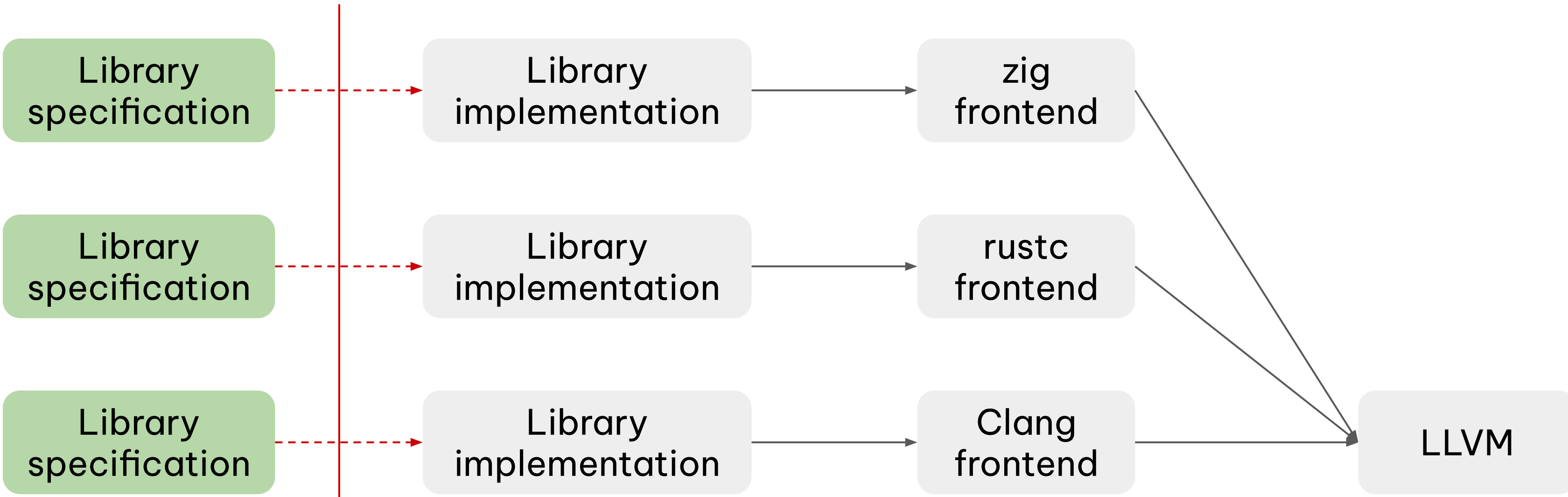


Information is lost too early

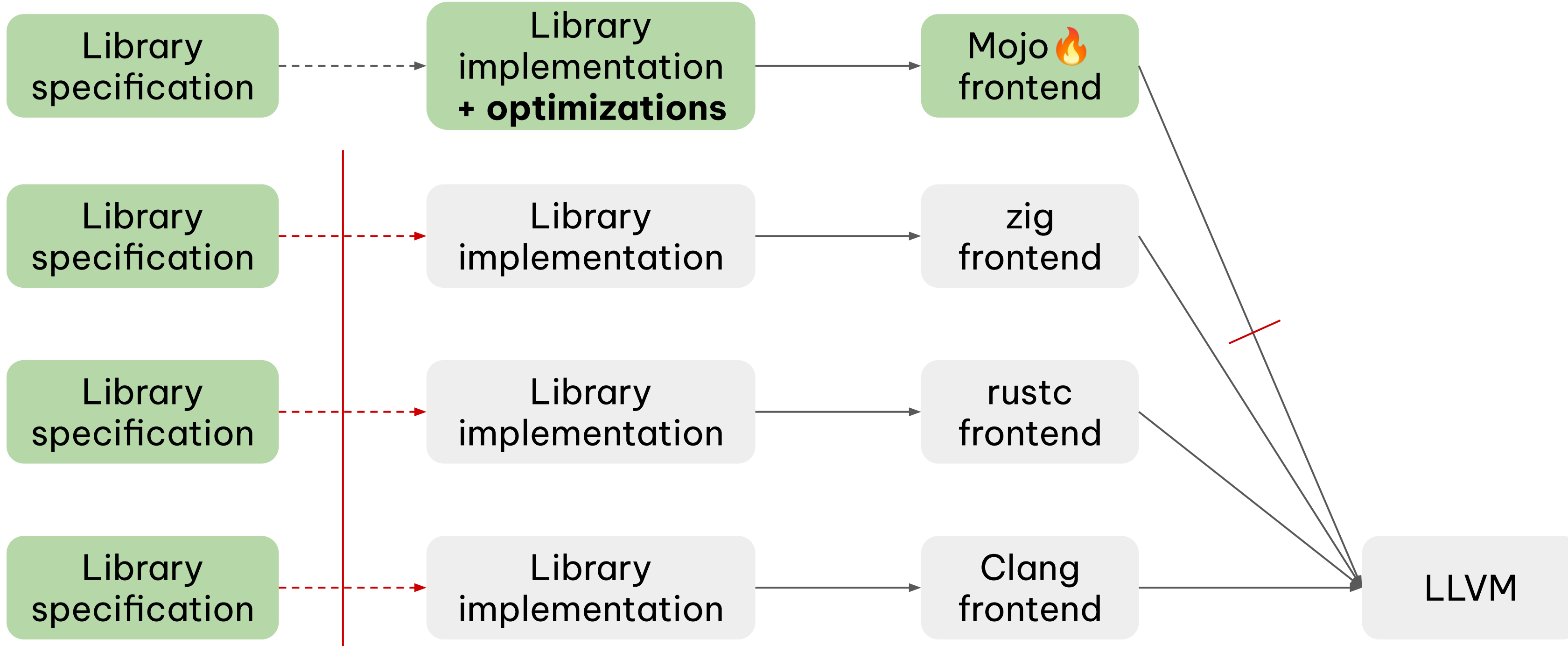


Information is lost too early

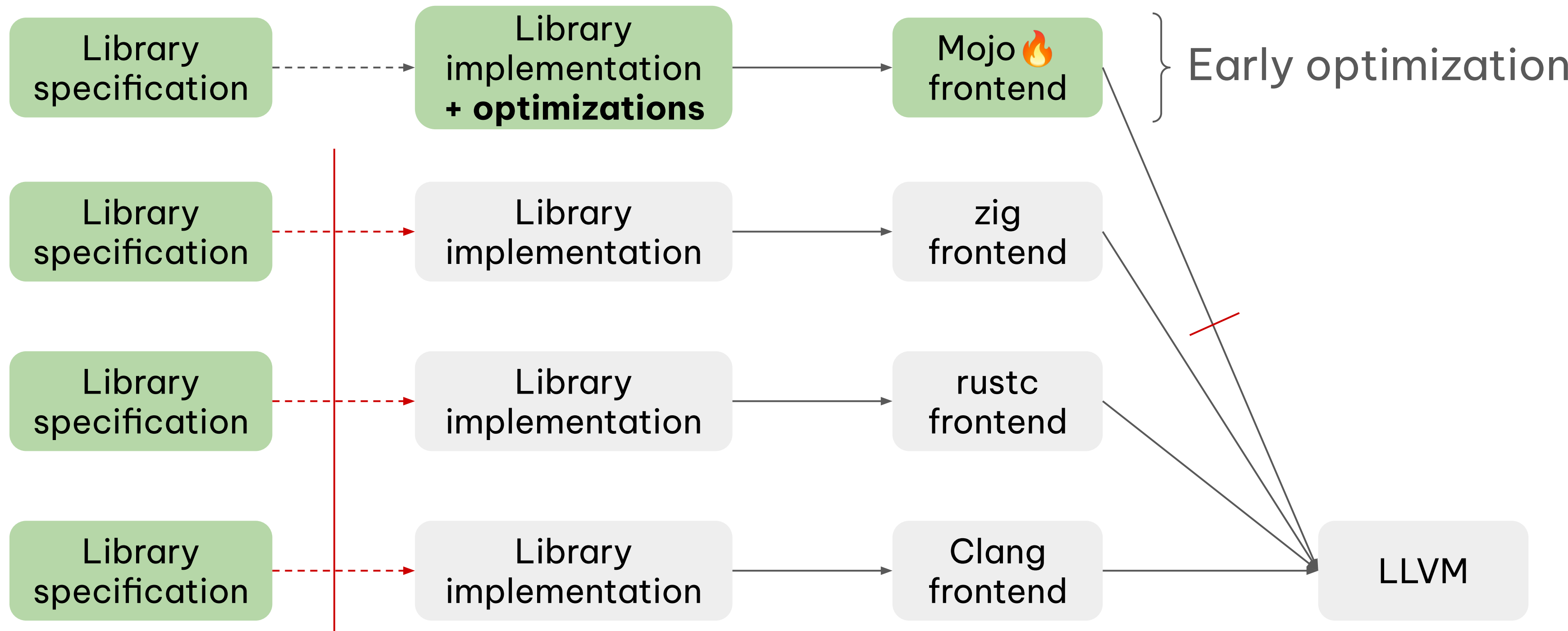
Information is lost too early



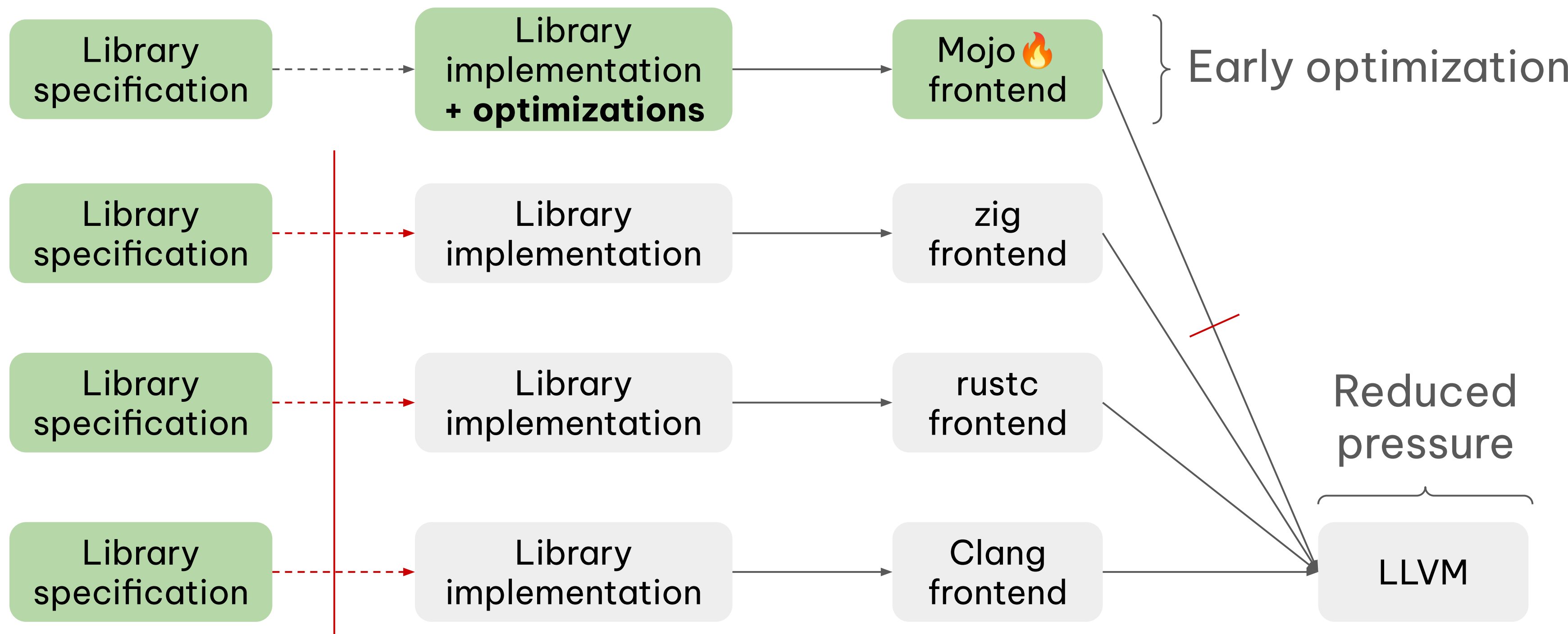
Information is lost too early



Information is lost too early

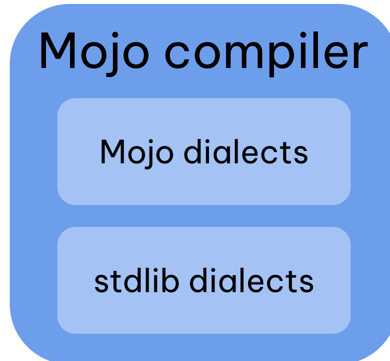


Information is lost too early



How can we solve this problem in Mojo🔥?

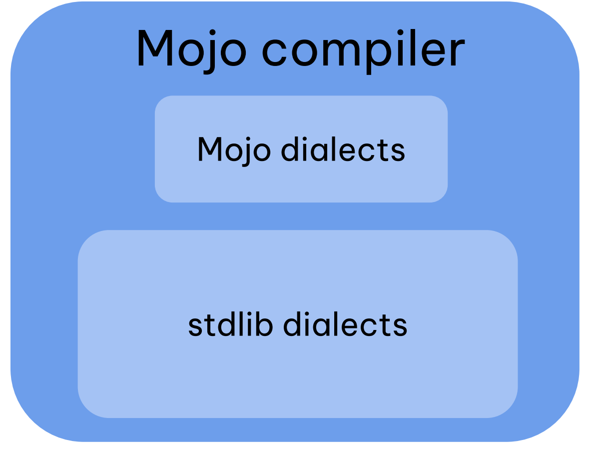
How can we solve this problem in Mojo ?



How can we solve this problem in Mojo ?

Doesn't scale well

How can we solve this problem in Mojo ?



Doesn't scale well

Complexifies the compiler

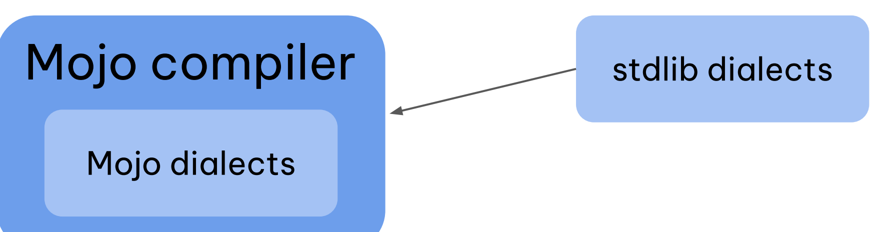
How can we solve this problem in Mojo ?

Doesn't scale well

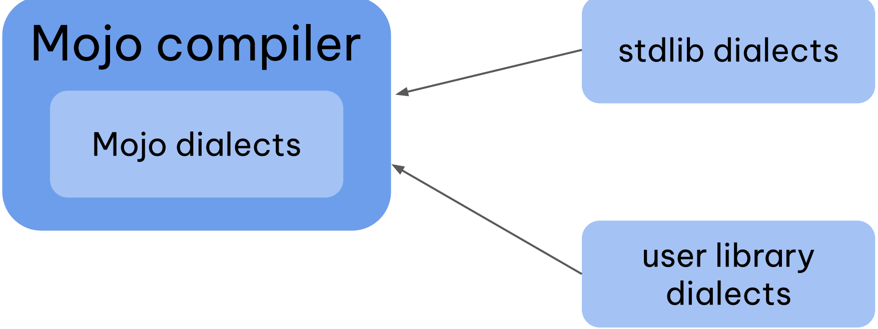
Complexifies the compiler

Doesn't generalize to user libraries

How can we solve this problem in Mojo🔥?



How can we solve this problem in Mojo ?



Anatomy of a custom op in Mojo🔥

```
fn mul_two(x: Int32) -> Int32:
    return x * 2
```

Anatomy of a custom op in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2
```

Anatomy of a custom op in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2
```

- Familiar for Mojo users

Anatomy of a custom op in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2
```

- Familiar for Mojo users
- No need to learn C++/ODS/TableGen

Anatomy of a custom op in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2
```

- Familiar for Mojo users
- No need to learn C++/ODS/TableGen
- Sufficient for all the cases we presented

Anatomy of a custom op in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2
```

- Familiar for Mojo users
- No need to learn C++/ODS/TableGen
- Sufficient for all the cases we presented
- Very few changes required in the compiler

Anatomy of a custom op in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2
```

Functions encode:
- A verifier

Anatomy of a custom op in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2
```

Functions encode:
- A verifier
- A lowering

Anatomy of a custom op in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2
```

Functions encode:

- A verifier
- A lowering
- An interpreter

Anatomy of a custom op in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2
```

Functions encode:

- A verifier
- A lowering
- An interpreter
- A few interfaces (side effects)

Anatomy of a custom optimization in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2


@custom_op(add_mul_two)
fn add(x: Int32, y: Int32) -> Int32:
    return x + y
```

Anatomy of a custom optimization in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2


@custom_op(add_mul_two)
fn add(x: Int32, y: Int32) -> Int32:
    return x + y
```

```
fn add_mul_two(inout op: Operation,
               inout b: Rewriter) -> Bool:
```

"Canonicalization patterns"

Anatomy of a custom optimization in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2


@custom_op(add_mul_two)
fn add(x: Int32, y: Int32) -> Int32:
    return x + y
```

```
fn add_mul_two(inout op: Operation,
               inout b: Rewriter) -> Bool:
    var loc = op.location()
    if op.operand(0) != op.operand(1):
        return True

    var new_op = Op[mul_two](
        loc,
        operands=List[Value](op.operand(0))
        results=List[Type](op.result(0).type()),
        params=op.get_attr("params"),
    )
    _ = b.insert(new_op)
    b.replace_op_with(op, new_op)
    return True
```

Anatomy of a custom optimization in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2


@custom_op(add_mul_two)
fn add(x: Int32, y: Int32) -> Int32:
    return x + y
```

```
fn add_mul_two(inout op: Operation,
               inout b: Rewriter) -> Bool:
    var loc = op.location()
    if op.operand(0) != op.operand(1):
        return True

    var new_op = Op[mul_two](
        loc,
        operands=List[Value](op.operand(0))
        results=List[Type](op.result(0).type()),
        params=op.get_attr("params"),
    )
    _ = b.insert(new_op)
    b.replace_op_with(op, new_op)
    return True
```

MLIR Mojo API

Anatomy of a custom optimization in Mojo🔥

```
@custom_op
fn mul_two(x: Int32) -> Int32:
    return x * 2


@custom_op(add_mul_two)
fn add(x: Int32, y: Int32) -> Int32:
    return x + y
```

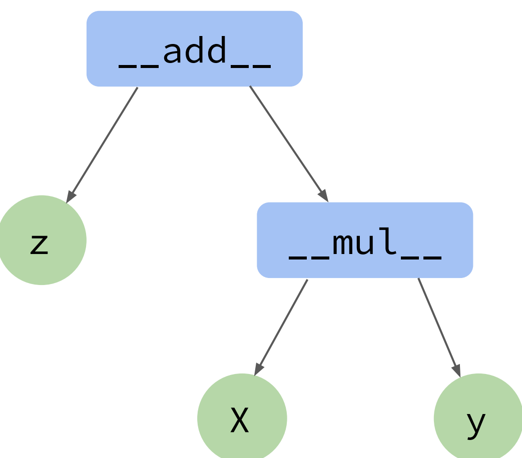
```
fn add_mul_two(inout op: Operation,
               inout b: Rewriter) -> Bool:
    var loc = op.location()
    if op.operand(0) != op.operand(1):
        return True

    var new_op = Op[mul_two](
        loc,
        operands=List[Value](op.operand(0))
        results=List[Type](op.result(0).type()),
        params=op.get_attr("params"),
    )
    _ = b.insert(new_op)
    b.replace_op_with(op, new_op)
    return True
```

We reference the functions directly

Handling memory-based optimizations

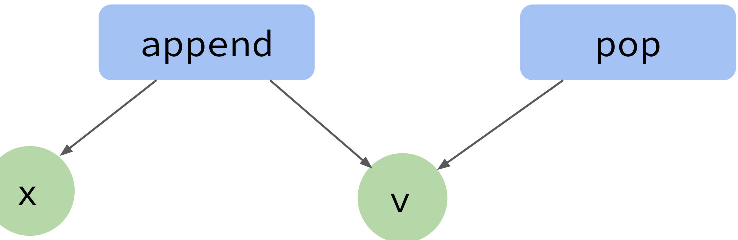
`z + x * y`



Handling memory-based optimizations

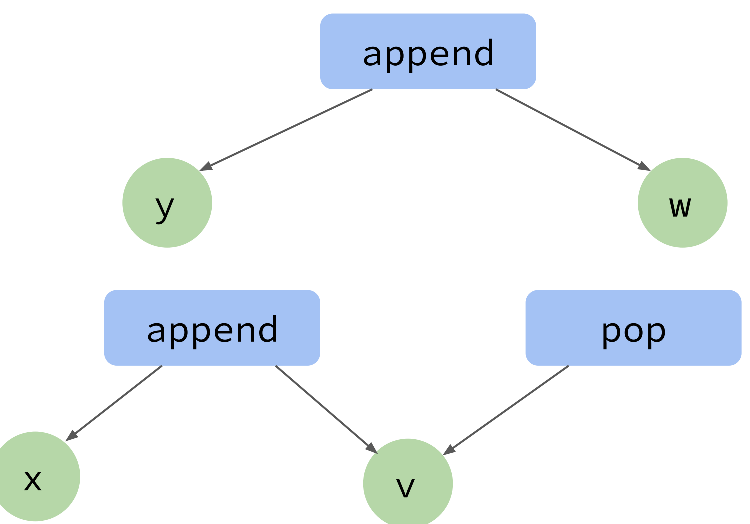
```
v.append(x)

v.pop()
```



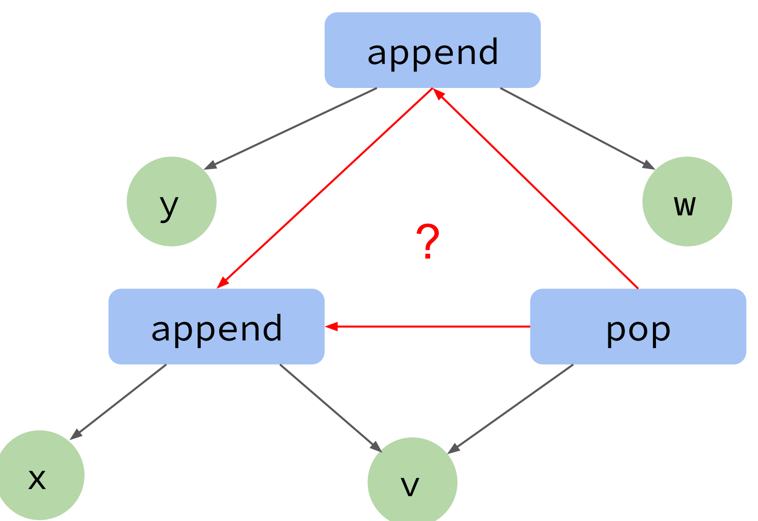
Handling memory-based optimizations

```
v.append(x)
w.append(y)
v.pop()
```



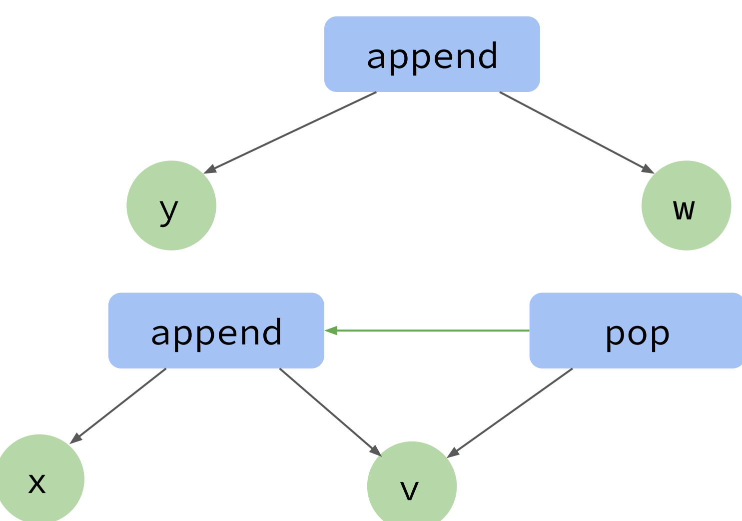
Handling memory-based optimizations

```
v.append(x)
w.append(y)
v.pop()
```



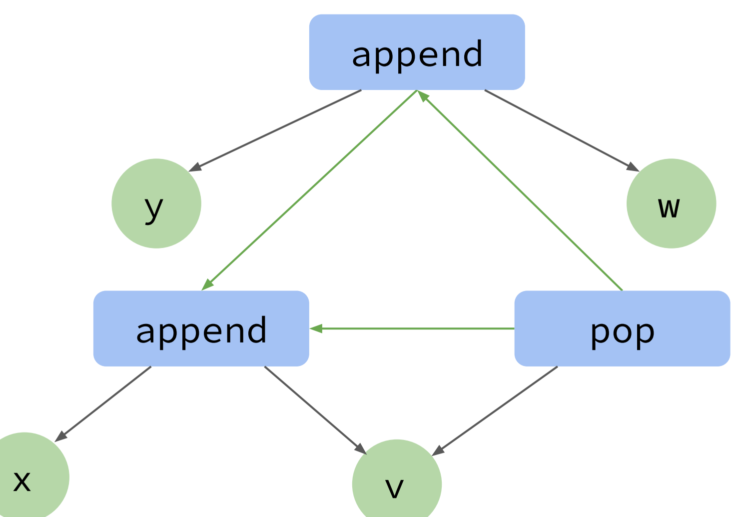
Handling memory-based optimizations with memorySSA

```
v.append(x)
w.append(y) # no alias
v.pop()
```

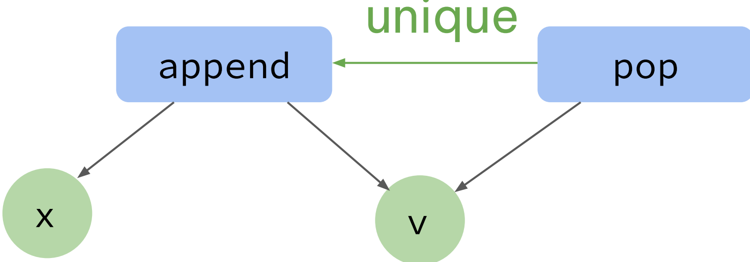


Handling memory-based optimizations with memorySSA

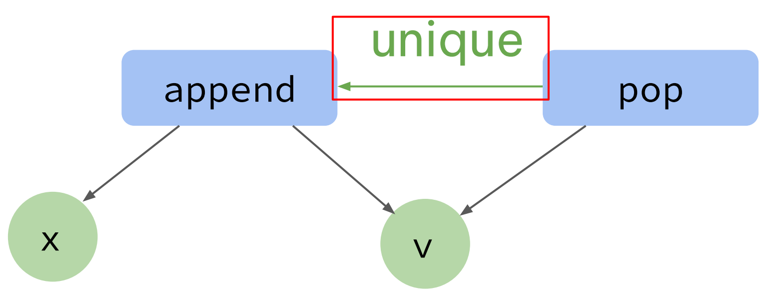
```
v.append(x)
w.append(y) # may alias
v.pop()
```



Optimizing the append/pop example



Optimizing the append/pop example

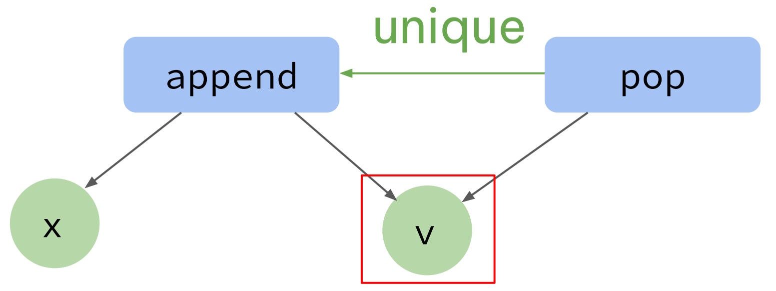



```
fn append_pop(inout op: Operation,
              inout r: Rewriter) -> Bool:
  var memoryssa = r.get_memoryssa()
  var append =
    memoryssa.get_unique_predecessor[Op[List.append]](op)
  if not append:
    return True

  if append.operands[0] != op.operands[0]:
    return True

  rewriter.erase_op(append)
  rewriter.replace_op(op, op.operands[1])
  return True
```

Optimizing the append/pop example

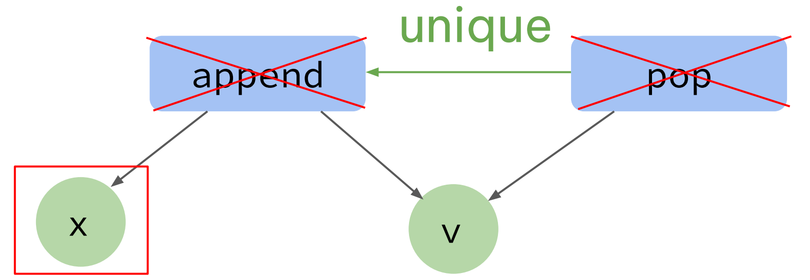



```
fn append_pop(inout op: Operation,
              inout r: Rewriter) -> Bool:
  var memoryssa = r.get_memoryssa()
  var append =
    memoryssa.get_unique_predecessor[Op[List.append]](op)
  if not append:
    return True

  if append.operands[0] != op.operands[0]:
    return True

  rewriter.erase_op(append)
  rewriter.replace_op(op, op.operands[1])
  return True
```

Optimizing the append/pop example




```
fn append_pop(inout op: Operation,
              inout r: Rewriter) -> Bool:
  var memoryssa = r.get_memoryssa()
  var append =
    memoryssa.get_unique_predecessor[Op[List.append]](op)
  if not append:
    return True

  if append.operands[0] != op.operands[0]:
    return True

  rewriter.erase_op(append)
  rewriter.replace_op(op, op.operands[1])
  return True
```

Optimizing the string example

Optimizing the string example

__init__ ← unique — __del__

```
fn string_init(inout op: Operation,
               inout r: Rewriter) -> Bool:
  var memoryssa = r.get_memoryssa()
  var del =
    memoryssa.get_unique_successor[Op[List.__del__]](op)

  if not del:
    return True
  if del.operands(0) != op.operands(0):
    return True

  rewriter.erase_op(op)
  rewriter.erase_op(del)
  return True
```

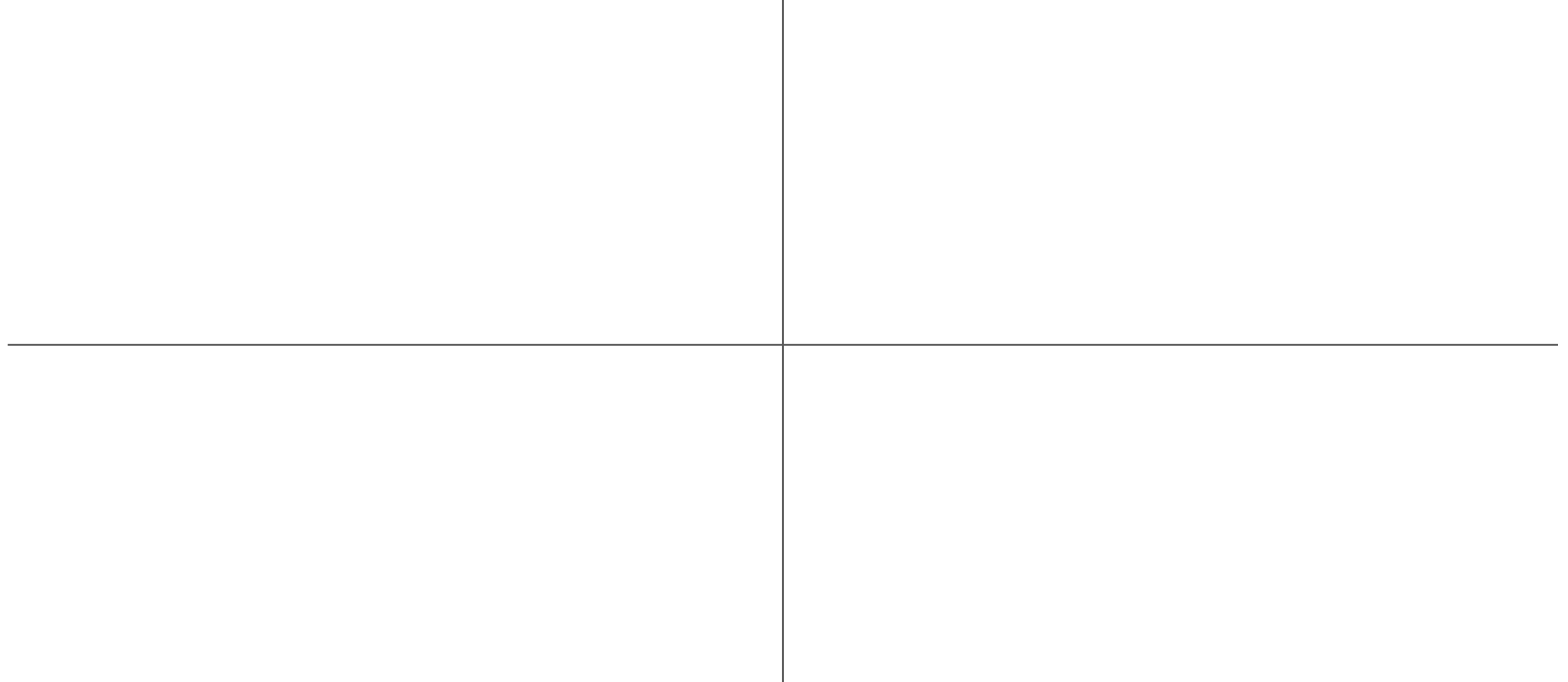
Optimizing the string example

```
fn string_init(inout op: Operation,
               inout r: Rewriter) -> Bool:
  var memoryssa = r.get_memoryssa()
  var del =
    memoryssa.get_unique_successor[Op[List.__del__]](op)

  if not del:
    return True
  if del.operands(0) != op.operands(0):
    return True

  rewriter.erase_op(op)
  rewriter.erase_op(del)
  return True
```

Conclusion

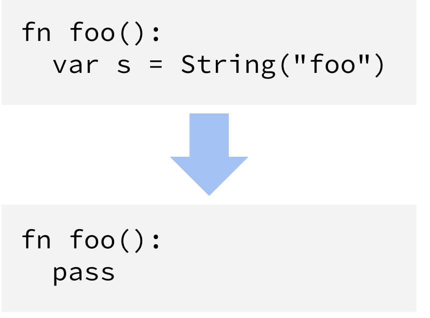


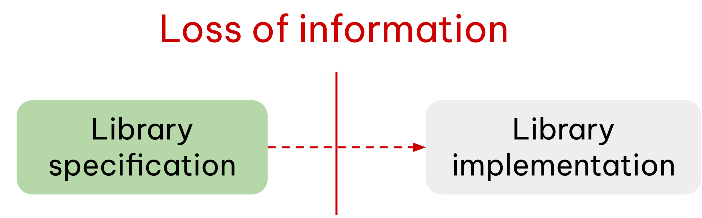
Conclusion

```
fn foo():
  var s = String("foo")
```

```
fn foo():
  pass
```

Conclusion

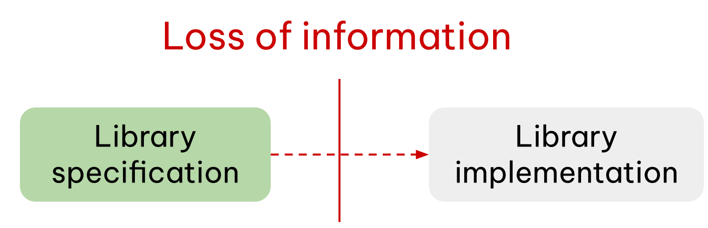




Conclusion

```
fn foo():
  var s = String("foo")
```

```
fn foo():
  pass
```

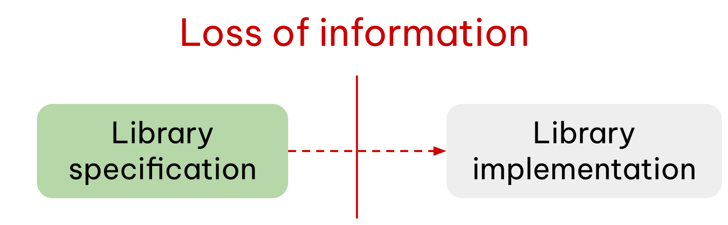



```
class String:
  @custom_op(string_init)
  fn __init__(inout self, x: StringLiteral):
    ...
```

Conclusion

```
fn foo():
  var s = String("foo")
```

```
fn foo():
  pass
```




```
class String:
  @custom_op(string_init)
  fn __init__(inout self, x: StringLiteral):
    ...
```