A Deep Dive into the Interprocedural Optimization Infrastructure
Outline

- What is IPO? Why is it?
- Introduction of IPO passes in LLVM
- Inlining
- Attributor
What is IPO?
What is IPO?

- **Pass Kind in LLVM**
  - Immutable pass
  - Loop pass
  - Function pass
  - Call graph SCC pass
  - Module pass

**Intra**procedural

**Inter**procedural

IPO considers more than one function at a time
Call Graph

- **Node**: functions
- **Edge**: from caller to callee

```c
void A() {
    B();
    C();
}
void B() {
    C();
}
void C() {
    ... 
}
```
SCC stands for “Strongly Connected Component”
Call Graph SCC

- SCC stands for “Strongly Connected Component”
Passes In LLVM
IPO passes in LLVM

- Where
  - Almost all IPO passes are under llvm/lib/Transforms/IP0
Categorization of IPO passes

- **Inliner**
  - AlwaysInliner, Inliner, InlineAdvisor, ...

- **Propagation between caller and callee**
  - Attributor, IP-SCCP, InferFunctionAttrs, ArgumentPromotion, DeadArgumentElimination, ...

- **Linkage and Globals**
  - GlobalDCE, GlobalOpt, GlobalSplit, ConstantMerge, ...

- **Others**
  - MergeFunction, OpenMPOpt, HotColdSplitting, Devirtualization...
Why is IPO?

- **Inliner**
  - Specialize the function with call site arguments
  - Expose local optimization opportunities
  - Save jumps, register stores/loads (calling convention)
  - Improve instruction locality

- **Propagation between caller and callee**
  - Other passes would benefit from the propagated information

- **Linkage and Globals related**
  - Exploit the fact all uses of internal values are known
  - Remove unused internal globals
  - Cooperates with LTO
Pass Kind

- Module Pass\(^1\)
  - Take a module as a “unit”
  - The most coarse-grained pass kind
Pass Kind

- **Call Graph SCC Pass**[^1]
  - Take a SCC of call graph as a “unit”
  - Applied in post order of call graph
    - bottom-up
- **Allowed**
  - Modify the current SCC
  - Add or remove globals
- **Disallowed**
  - Modify any SCCs other than the current one
  - Add or remove SCC

[^1]: Reference link
Common IPO Pitfalls

- Scalability
- Complicated linkages
- Optimization pipeline, phase ordering
- Function pointer, different “kinds” of call sites, non-call site uses, …
- Variadic functions, complicated attributes (naked, byval, inreg, …)
- Keeping call graphs updated (for new and old pass managers)
  - CallGraph … old PM
  - LazyCallGraph … new PM
Existing IPO passes
Simple inliner -inline

- Bottom-up Inlining
  - CGSCC pass
- Example

```c
void foo(int cond) {
    if (cond) {
        /* hot */
        ...
    } else {
        /* cold */
        ...
    }
}

void use_foo() {
    foo(x);
}
```
Partial inliner -partial-inliner

- Inlining hot region only
- Example

```c
void foo(int cond) {
    if (cond) {
        /* hot */
        ...
    } else {
        /* cold */
        ...
    }
}

void use_foo() {
    foo(x);
}

void foo.cold() {
    /* cold */
    ...
}

void use_foo() {
    if (x) {
        /* hot */
        ...
    } else {
        foo.cold();
    }
}
```
Always inliner -always-inline

- Try to inline functions marked “alwaysinline”
- Runs even in -O0 or with llvm passes disabled!
- Basically overrides the inliner heuristic.
- Example

```
> cat test.ll

define i32 @inner() alwaysinline {
  entry:
    ret i32 1
}

define i32 @outer() {
  entry:
    %ret = call i32 @inner()
    ret i32 %ret
}

> opt -always-inline test.ll -S

define i32 @inner() alwaysinline {
  entry:
    ret i32 1
}

define i32 @outer() {
  entry:
    ret i32 1
}```
IPSCCP - ipsccp

- Interprocedural Sparse Conditional Constant Propagation
- Blocks and instructions are assumed dead until proven otherwise.
- Traverses the IR to see which Instructions/Blocks/Functions are alive and which values are constant.
define internal i32 \@recursive(i32 %0) {
    \%2 = icmp eq i32 %0, 0
    br i1 \%2, label \%3, label \%4
    \%3 = add nsw i32 %0, 1
    \%4 = call i32 \@recursive(i32 %5)
    br label \%7

7: 
    %.0 = phi i32 [\%0, \%3], [\%6, \%4]
    ret i32 %.0
}

define i32 \@callsite() {
    \%1 = call i32 \@recursive(i32 0)
    \%2 = call i32 \@recursive(i32 \%1)
    ret i32 \%2
}

define internal i32 \@recursive(i32 \%0) {
    br label \%2
}

2:
    br label \%3

3: 
    ret i32 undef
}

define i32 \@callsite() {
    \%1 = call i32 \@recursive(i32 0)
    \%2 = call i32 \@recursive(i32 0)
    ret i32 0
}
Argument Promotion -argpromotion

- Promote “by pointer” arguments to be “by value” arguments
  - If the argument is only “loaded”
  - Handle both load and GEP instructions
  - Pass the loaded value to the function, instead of the pointer

- Flow
  - Save information about loads of viable arguments
  - Create new function
  - Insert such load instructions to the caller

- This is (partially) subsumed by the Attributor
Argument Promotion: Example

> cat test.ll

```assembly
%T = type { i32, i32 }
@G = constant %T { i32 17, i32 0 }
define internal i32 @test(%T* %p) {
  entry:
  %a.gep = getelementptr %T, %T* %p, i64 0, i32 0
  %a = load i32, i32* %a.gep
  %v = add i32 %a, 1
  ret i32 %v
}
define i32 @caller() {
  entry:
  %v = call i32 @test(%T* @G)
  ret i32 %v
}
```

> opt -S -argpromotion test.ll

```assembly
%T = type { i32, i32 }
@G = constant %T { i32 17, i32 0 }
define internal i32 @test(i32 %p.0.0.val) {
  entry:
  %v = add i32 %p.0.0.val, 1
  ret i32 %v
}
define i32 @caller() {
  entry:
  %G.idx = getelementptr %T, %T* @G, i64 0, i32 0
  %G.idx.val = load i32, i32* %G.idx
  %v = call i32 @test(i32 %G.idx.val)
  ret i32 %v
}
```
**InferFunctionAttrs** - inferattrs

- Annotate function attrs on known library functions.
- Example

```
> cat test.ll

define i8* @foo() {
  %1 = call i8* @malloc(i64 1)
  ret i8* %1
}

declare i8* @malloc(i64)

> opt -inferattrs test.ll -S

define i8* @foo() {
  %1 = call i8* @malloc(i64 1)
  ret i8* %1
}

; Function Attrs: nofree nounwind
declare noalias i8* @malloc(i64) #0

attributes #0 = { nofree nounwind }
```
DeadArgumentElimination -deadargelim

- Remove dead arguments from internal functions
- How:
  - Delete arglist (...) if no va_start is called
  - Assume all arguments dead unless proven otherwise
- Example

```assembly
; Dead arg only used by dead retval
define internal i32 @test(i32 %DEADARG) {
    ret i32 %DEADARG
}
define i32 @test2(i32 %A) {
    %DEAD = call i32 @test(i32 %A) ; 0 uses
    ret i32 123
}
define internal void @test() {
    ret void ; Argument was eliminated
}
define i32 @test2(i32 %A) {
    call void @test()
    ret i32 123
}
```
CalledValuePropagation
-called-value-propagation

- Add metadata to indirect call sites indicating potential callees
- Example

```assembly
define void @test_select_entry(i1 %flag) {
  entry:
    call void @test_select(i1 %flag)
    ret void
}

define internal void @test_select(i1 %f) {
  entry:
    %tmp = select i1 %f, void ()* @foo_1, void ()* @foo_2
    call void %tmp()
    ret void
}

declare void @foo_1() norecurse
declare void @foo_2() norecurse
```

```assembly
define void @test_select_entry(i1 %flag) {
  entry:
    call void @test_select(i1 %flag)
    ret void
}

define internal void @test_select(i1 %f) {
  entry:
    %tmp = select i1 %f, void ()* @foo_1, void ()* @foo_2
    call void %tmp0(), !callees !0
    ret void
}
declare void @foo_1() norecurse
declare void @foo_2() norecurse
[0 = !{void ()* @foo_1, void ()* @foo_2}]```
**FunctionAttrs**

- *function-attrs*
- *rpo-function-attrs*

- Deduce and propagate attributes
- Two versions
  - Bottom-up
  - Top-bottom (reverse post order)
- This is subsumed by the Attributor
- Example

```assembly
declare nonnull i8* @foo()

define i8* @bar(i1 %c, i8* %ptr) {
  br i1 %c, label %true, label %false
  true:
    %q = getelementptr inbounds i8, i8* %ptr, i32 1
    ret i8* %q

  false:
    %ret = call i8* @foo()
    ret i8* %ret
}
```
PruneEH -prune-eh

- Remove unused exception handling code
  - Turn `invoke` into `call` when the callee is proven not to throw an exception
- Example

```c
define void @foo() nounwind {
  ...
  ret void
}

define i32 @caller() personality i32 (...)* @eh_function {
  invoke void @foo() to label %Normal unwind label %Except
}

Normal:
  ret i32 0

Except:
  landingpad { i8*, i32 } catch i8* null
  ret i32 1
}
```

```c
define void @foo() nounwind {
  ...
  ret void
}

define i32 @caller() #0 personality i32 (...)* @eh_function {
  call void @foo(); Note there's no invoke
  br label %Normal; and the %Except block was removed.
}

Normal:
  ret i32 0
}
```

https://llvm.org/docs/Passes.html#prune-eh-remove-unused-exception-handling-info
GlobalDCE -globaldce

- Eliminate unreachable internal globals
- An aggressive algorithm
  - Initially assume all globals are dead
- Example

@A = global i32 0
@D = internal alias i32, i32* @A
@L1 = alias i32, i32* @A
@L2 = internal alias i32, i32* @L1
@L3 = alias i32, i32* @L2

@A = global i32 0
@L1 = alias i32, i32* @A
@L2 = internal alias i32, i32* @L1
@L3 = alias i32, i32* @L2
GlobalOpt -globalopt

- Optimize global values
  - Evaluate static constructors (llvm.global_ctors)
  - Optimize non-address-taken globals
    - Constant Propagation
    - Dead global elimination
GlobalOpt : Example

```assembly
@foo = internal global i32 4
define i32 @load_foo() {
  %four = load i32, i32* @foo
  ret i32 %four
}

@bar = global i32 5
define i32 @load_bar() {
  %may_not_five = load i32, i32* @bar
  ret i32 %may_not_five
}

%0 = type { i32, void (*)*, i8* }
@llvm.global_ctors = appending global ... @baz_constructor ...
@baz = global i32

define void @baz_constructor() {
  store i32 5, i32* @baz
  ret void
}
```

Constant Propagation

Dead global elimination

Define i32 @load_foo() {
  ret i32 4
}

Define i32 @load_bar() {
  ret i32 %may_not_five
}

Evaluate static constructor
Constant Merge -constmerge

- Merge duplicate global constants together into a shared one
  - Construct a map from constants to globals

Example

```assembly
@foo = constant i32 6
@bar = internal unnamed_addr constant i32 6
@baz = constant i32 6

define i32 @use_bar(i32 %arg) {
  %six = load i32, i32* @bar
  %ret = add i32 %arg, %six
  ret i32 %ret
}
```

```assembly
@foo = constant i32 6
@baz = constant i32 6

define i32 @use_bar(i32 %arg) {
  %six = load i32, i32* @foo, align 4
  %ret = add i32 %arg, %six
  ret i32 %ret
}
```
MergeFunctions -mergefunc

- Find equivalent functions and merge them
  - Introduce a “total order” among functions
  - Use binary search to find an equivalent function

```c
define internal i64 @foo(i32* %P, i32* %Q) {
  store i32 4, i32* %P
  store i32 6, i32* %Q
  ret i64 0
}

define internal i64* @bar(i32* %P, i32* %Q) {
  store i32 4, i32* %P, align 4
  store i32 6, i32* %Q, align 4
  ret i64* null
}

define i64 @use_foo(i32* %P, i32* %Q) {
  %ret = call i64 @foo(i32* %P, i32* %Q)
  ret i64 %ret
}

define i64* @use_bar(i32* %P, i32* %Q) {
  %ret = call i64 @bar(i32* %P, i32* %Q)
  ret i64* %ret
}

define i64* @use_foo(i32* %P, i32* %Q) {
  %ret = call i64 @foo(i32* %P, i32* %Q)
  ret i64 %ret
}

define i64* @use_bar(i32* %P, i32* %Q) {
  %ret = call i64 @bar(i32* %P, i32* %Q)
  ret i64* %ret
  https://llvm.org/docs/MergeFunctions.html
```
**OpenMPOpt -openmp-opt**

- **Various OpenMP specific optimization**
  - Runtime call deduplication
  - runtime call replacement
  - parallel region merging
  - GPU code optimization, ...

- **Example**

```
; Runtime call deduplication

define void @test() {
  %nthreds1 = call i32 @omp_get_num_threads()
  call void @use(%nthreds1)
  %nthreds2 = call i32 @omp_get_num_threads()
  call void @use(%nthreds2)
  ret void
}
```

```
define void @test() {
  %nthreds1 = call i32 @omp_get_num_threads()
  call void @use(%nthreds1)
  %nthreds1 = call i32 @omp_get_num_threads()
  call void @use(%nthreds1)
  ret void
}
```
HotColdSplitting -hotcoldsplit

- Split hot regions and cold regions
  - Extract cold regions to improve locality
- Example

```c
extern void bar(int);
extern void __attribute__((cold)) sink();
void foo(int cond) {
    if (cond) {
        if (cond > 10)
            bar(0);
        else
            bar(1);
        sink();
    }
    bar(2);
}

test

extern void bar(int);
extern void __attribute__((cold)) sink();
void foo(int cond) {
    if (cond) {
        if (cond > 10)
            bar(0);
        else
            bar(1);
        sink();
    }
    bar(2);
}
```

Hot Cold Splitting Optimization Pass In LLVM, A. Kumar, LLVM Developers' Meeting 2019
-attributor
-attributor-cgscce
-attributor-enable={all,module,cgscce} -O{1,2,3,...}

- Fixpoint iteration framework
  - Deduce various (>20 now) “attributes” aggressively and simultaneously

- Two versions
  - CGSCC pass and Module pass

- Example

```c
define i32 @f(i32 * %ptr, i32 %x) {    %load = load i32, i32 * %ptr    %res = add i32 %load, %x    ret i32 %res }
```

```c
define i32 @f(i32* nocapture nofreenonnull readonly align 4 dereferenceable(4) %ptr,    i32 %x) #0 {    %load = load i32, i32 * %ptr, align 4    %res = add i32 %load, %x    ret i32 %res }
```

```
attributes #0 = { argmemonly nofree nosync nounwind readonly willreturn }
```

---

The Attributor: A Versatile Inter-procedural Fixpoint Iteration Framework, J. Doerfert, LLVM Developers' Meeting 2019
Inlining (in LLVM)

Stefanos Baziotis
NEC Corporation and University of Athens
users.uoa.gr/~sdi1600105/
stefanos.baziotis@gmail.com
Inlining

- Replaces a function call (site) with the body of the called function.
Inlining

- Replaces a function call (site) with the body of the called function.
- Inlining is a relatively simple transformation. It’s the decision of whether (and how much) to inline or not that is difficult.
Inlining

- Replaces a function call (site) with the body of the called function.

- Inlining is a relatively simple transformation. It’s the decision of whether (and how much) to inline or not that is difficult.

- Actually, it has been shown to be at least as hard as the Knapsack problem, so, NP-complete\(^1\).

Inlining

- Replaces a function call (site) with the body of the called function.

- Inlining is a relatively simple transformation. It’s the decision of whether (and how much) to inline or not that is difficult.

- Actually, it has been shown to be at least as hard as the Knapsack problem, so, NP-complete\(^1\).

- For that reason, people have been using hand-written heuristics that “empirically work”. Lately, Machine Learning is being used.

---

Inlining - Can We Always Inline? No!

Usually, because we don’t have the function code:

- Other Modules / Compilation Units (LTO can help there)
- Shared Libraries
- Calls through function pointers (so, also virtual calls)
  - In reality, the compiler may inline some of the candidates in place\(^2\)^\(^3\).

\(^2\) Compiler Confidential, Eric Brumer, GoingNative 2013
\(^3\) Devirtualization in LLVM, P. Padleweksi, LLVM Developers’ Meeting 2016
Inlining - Can We Always Inline? No!

But also because of weird code structure:

- Recursive functions
  - Although tail recursion can be inlined.
  - Also, if at some point we can turn recursion into loops.
Inlining - Benefits

- Removes branching because of call.
Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```c
void foo() {
    for (...) { // hot loop
        bar();
        ...
    }
}
```
Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```c
150 void foo() {
151     for (...) { // hot loop
152         bar();
153         ...
154     }
155 }
```

```c
1 void bar() {
2     ...
3 }
```
Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```c
150 void foo() {
151     for (...) { // hot loop
152         bar();
153         ...
154     }
155 }
```

```c
1 void bar() {
2     ...
3 }
```
Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```c
150 void foo() {
151    for (...) { // hot loop
152        bar();
153        ...
154    }
155 }
```

```c
1 void bar() {
2    ...
3 }
```
Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```c
150  void foo() {
151      for (...) { // hot loop
152          bar();
153          ...
154      }
155  }
```

```c
1  void bar() {
2      ...
3  }
```
Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```c
150 void foo() {
151     for (...) {  // hot loop
152         bar();
153     }
154 }
```

```c
1 void bar() {
2     ...
3 }
```
Inlining - Benefits

- Removes branching because of call.
  - May help in (instruction cache) locality, for example if we inline a function in a loop.

```c
150 void foo() {
151     for (...) { // hot loop
152         bar();
153     }
154 }
155 }
1  void bar() {
  2      ...
  3  }
```
Inlining - Benefits

● Removes branching because of call.
  ○ May help in (instruction cache) locality, for example if we inline a function in a loop.

● Removes save / restore of registers, function prologue / epilogue etc.
  ○ Common heuristic: If the (actual) function code is less than two times the Call Instruction Sequence, inline it.
Inlining - Benefits

But *most importantly: It is an enabling transformation!*

```
define internal i32 @callee(i32 %A, i32 %B) {
    %C = sdiv i32 %A, %B
    ret i32 %C
}

define i32 @caller() {
    %X = call i32 @callee(i32 10, i32 3)
    ret i32 %X
}
```

```
define i32 @caller() {
    ret i32 3
}
```
Inlining - Drawbacks

- Code Duplication
  - Analyze same code multiple times
Inlining - Drawbacks

- **Code Duplication**
  - Analyze same code multiple times

- **Code Size Explosion**
  - Executable Size Grows
  - Impacts the Instruction Cache

Godbolt Snippet

```
void bar() {
    // big function
}

void foo() {
    for (...) {  // hot loop
        if (COND1)  // COND1 is rarely true
            bar();
    ...
}
```
Inlining - Drawbacks

- Code Duplication
  - Analyze same code multiple times

- Code Size Explosion
  - Executable Size Grows
  - Impacts the Instruction Cache

If this is latency-sensitive code, that may be a good decision!
Inlining - Drawbacks

- **Code Duplication**
  - Analyze same code multiple times

- **Code Size Explosion**
  - Executable Size Grows
  - Impacts the Instruction Cache

- **Increased Register Allocator Pressure**
  - There’s no register save / restore
    - Live ranges of registers are extended
  - More loop invariants may be discovered
    - More registers to keep them
Inlining in LLVM - Place in the Pipeline

Because it is the most important enabling transformation, inlining happens early in the pipeline. And it is the main focus of it.
Inlining in LLVM - Pass Manager

- Inlining is a Call-Graph SCC pass, which means it visits inlining candidates in a bottom-up SCC order.
  - First callees, then callers
Inlining in LLVM - Pass Manager

- Inlining is a Call-Graph SCC pass, which means it visits inlining candidates in a bottom-up SCC order.
  - First callees, then callers
- The Pass Manager interlaces function passes between the visits of the inliner to the functions.
Inlining in LLVM - Example of Pass Ordering

```c
1 int B() { return 2; }
2 int C() { return 1; }
3
4 void A() {
5     return B() + C();
6 }
```
Inlining in LLVM - Example of Pass Ordering

opt -inline -mem2reg
Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```

- Run inliner on `B()`
Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```

- Run inliner on `B()`
- Run `mem2reg` on `B()`
Inlining in LLVM - Example of Pass Ordering

opt -inline -mem2reg

- Run inliner on C()
Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```
Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```

- Run inliner on \( A() \)
Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```

- Run inliner on `A()`
  - We inline `B()`, `C()`
Inlining in LLVM - Example of Pass Ordering

```
opt -inline -mem2reg
```

- Run inliner on $A()$
  - We inline $B()$, $C()$
- Run mem2reg on $A()$
Further Reading (in chronological order)

Further Reading (in chronological order)

- R. Muth, S. Debray, “Partial Inlining”. Technical Summary
- Arnold, Matthew and Fink, Stephen and Sarkar, Vivek and Sweeney, Peter F. “A Comparative Study of Static and Profile-Based Heuristics for Inlining”, DYNAMO '00: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, January 2000, Pages 52–64
Further Reading (in chronological order)


Attributor
Attributor Overview

- **Fixpoint iteration framework**
  - Deduce *various* (>20 now) “attributes” aggressively and simultaneously

- **Update states till fixpoint is reached**
  - Dependencies between states are automatically caught by Attributor

- **There are Module/CGSCC pass for both the old and new pass manager**
Why is it powerful?

- Attributor provides easy way to add new fixpoint analyses
- We can connect analyses with each other during fixpoint iteration
- Many existing IPO passes can be replaced by Attributor

- IPSCCP
- Argument Promotion
- Dead Argument Elimination
- Infer Function Attrs
- Prune EH
LLVM-IR Positions

- A class to specify positions in LLVM-IR

```
define i32* @f(i32* %argument) #0 {
  %call-site-returned = call i32* @g(i32* %argument) #1
  %flt = getelementptr inbounds i32, i32* %call-site-returned, i64 1
  ret i32* %flt
}
```
Abstract state

Known = Assumed

indicate pessimistic fixpoint

Assumed = Known

indicate optimistic fixpoint

Known = Assumed

fixpoint state
Abstract attribute

● What we call “attribute” here
  ○ Any stuff that describe properties of an IR position
  ○ Not only LLVM-IR attribute! (e.g. nonnull, nocapture, nofree, ...)

● They are called “abstract attribute” in the code
  ○ AbstractAttribute class
  ○ Often abbreviated as AA
Abstract attribute: Example

- AAs that correspond to LLVM-IR attributes
  - AANonNull ... nonnull
  - AANoCapture ... nocapture
  - AAAlign ... align

- AAs that related to LLVM-IR attributes
  - AAMemoryBehavior ... readnone, readonly, writeonly
  - AAMemoryLocation ... readnone, argmemonly, inaccessiblememory ...

- AAs that unrelated to any LLVM-IR attributes
  - AAIsDead ... Liveness Analysis
  - AAValueSimplify ... Value Simplification
Abstract attribute: Core methods

- **AbstractAttribute::initialize**
  - Initialize the state

- **AbstractAttribute::updateImpl**
  - Update the state
  - We can query states of some other AAs by `Attributor::getAAFor`

- **AbstractAttribute::manifest**
  - Manifest the changes to the IR.
ChangeStatus AANonNullReturned::updateImpl(Attributor &A) {
    Function *F = getAnchorScope();
    auto Before = getState();
    auto& S = getState();

    for (Value *RetVal : /* Iterate all returned values of F in some way */)
        S &= A.getAAFor<AANonNull>(*this, IRPosition::value(RetVal));

    if (S == Before)
        return ChangeStatus::UNCHANGED;
    return ChangeStatus::CHANGED;
}

Update Function: Example

Clamp states for all returned values
Phase of Attributor

- **Seeding**: Determine which kind of deduction or analysis we try to do
- **Update**: Update states till fixpoint is reached
- **Manifest**: Transform IR according to the results
Attributor Feature

- **Performance related**
  - Dependency type

- **Utility for users**
  - Helper classes for generic deduction
  - Helper functions for traversing assumed live uses, instructions, basicblocks...
  - Provides a uniform analysis pass query API
  - Selective seeding
  - Time traces
Attributor Feature

- Provides helper classes for generic deduction
  - All alive returned values → Function returned
  - All call sites → Function
  - All call site arguments → Function argument

- Example
  - AAReturnedFromReturnedValues

```cpp
struct AANonNullReturned
    : AAReturnedFromReturnedValues<AANonNull, AANonNull> {
    /* We do not have to implement updateImpl */
};
```
Attributor Feature

- Provides abstract states for common situations
- Example
  - IncIntegerState
  - DecIntegerState
  - BitIntegerState
  - BooleanState
**Attributor: Selective Seeding**

- **attributor-seed-allow-list**
  Comma separated list of attribute names that are allowed to be seeded.
  
  ```
  --attributor-seed-allow-list=AANonNull
  ```

- **attributor-function-seed-allow-list**
  Comma separated list of function names that are allowed to be seeded.
  
  ```
  --attributor-seed-allow-list=foo
  ```
Attributor: Time Trace
Recap

Existing IPO passes

Common IPO Pitfalls

- Scalability
- Complicated linkages
- Optimization pipeline, phase ordering
- Function pointer, different “kinds” of call sites, non-call site uses, ...
- Variadic functions, complicated attributes (naked, byval, inreg, …)
- Keeping call graphs updated (for new and old pass managers)
  - CallGraph ... old PM
  - LazyCallGraph ... new PM
Recap - Attributor

Dependency Graph

Phase of Attributor

- **Seeding**: Determine which kind of deduction or analysis we try to do
- **Update**: Update states till fixpoint is reached
- **Manifest**: Transform IR according to the results
Recap

- Attributor technical talk & tutorial @ LLVM-Dev’19
- IPO panel @ LLVM-Dev’19
- IPO technical talk @ LLVM-Dev’20

Contact us if you are interested in any of this!
A Deep Dive into the Interprocedural Optimization Infrastructure