

Code analysis  
*and*  
Transformation

# Alias Analysis



Simone Campanoni  
simone.campanoni@northwestern.edu



# Memory alias analysis: the problem

- Does  $j$  depend on  $i$  ?

```
i: (*p) = varA + 1  
j: varB = (*q) * 2
```

```
i: obj1.f = varA + 1  
j: varB = obj2.f * 2
```

- Do  $p$  and  $q$  point to the same memory location?
  - Does  $q$  alias  $p$ ?

# Outline

- Enhance CAT with alias analysis
- Simple alias analysis
- Alias analysis in LLVM

# Exploiting alias analysis in CATs

- Easiest: extending the transformation

- Midway: extending the analysis

*This is what homework H5  
is about!*

- Hardest: writing a CAT-specific alias analysis

*This is what homework H6  
is about!*

Let's start looking at the interaction between

**memory alias analysis**

and

a code transformation you are familiar with:

**constant propagation**

... but first, let's recall a term

# Escape variables

```
int x, y;  
int *p;  
p = &x;  
myF(p);  
...
```

```
void myF (int *q){  
    ...  
}
```

# Constant propagation revisited

```
int x, y;
```

```
int *p;
```

```
...
```

```
x = 5;
```

```
y = x + 1;
```

Is x constant here?

- Yes, only one value of x reaches this last statement

# Constant propagation revisited

```
int x, y;
```

```
int *p;
```

```
...
```

```
x = 5;
```

```
*p = 42;
```

```
y = x + 1;
```

Is x constant here?

- Yes, because x doesn't "escape" and therefore only one value of x reaches this last statement



# Constant propagation revisited

```
int x, y;  
int *p;  
... = &x;
```

...

```
x = 5;
```

```
*p = 42;
```

```
y = x + 1;
```

Goal of memory  
alias analysis: understanding

We need to know which variables escape

(your H4)

Is x constant here?

- If p **does not point** to x, then x = 5
- If p **definitely points** to x, then x = 42
- If p **might point** to x, then we have two reaching definitions that reach this last statement, so x is not constant

To exploit **memory alias analysis** in a code transformation

typically you extend the related code analyses

to use the information about pointer aliases

# Let's exploit alias analysis for making liveness analysis more powerful

- A variable  $v$  is live at a given point of a program  $p$  if
  - Exist a directed path from  $p$  to an use of  $v$  and
  - that path does not contain any definition of  $v$
- **What is the most conservative output of the analysis?  
(the bottom of the lattice)**

GEN[ $i$ ] = ?

KILL[ $i$ ] = ?

IN[ $i$ ] = GEN[ $i$ ]  $\cup$  (OUT[ $i$ ] - KILL[ $i$ ])

OUT[ $i$ ] =  $\bigcup_{s \text{ a successor of } i}$  IN[ $s$ ]

# Liveness analysis revisited

```
int x, y;
```

```
int *p;
```

```
... = &x;
```

```
x = 5;
```

```
...(no uses/definitions of x)
```

```
*p = 42;
```

```
y = x + 1;
```

What is the most conservative output of the analysis?  
(the bottom of the lattice)

How can we modify liveness analysis?

Is x alive here?

- Yes, **does not point to elsewhere** and the use of the value of x stored there will be used later
- If p **definitely points** to x, then no
- If p **might point** to x, then yes

# Liveness analysis revisited

mayPointTo : variable -> set<variable>

mustPointTo: variable -> set<variable>

How can we modify conventional liveness analysis?

$GEN[i] = \{v \mid \text{variable } v \text{ is used by } i\}$

$KILL[i] = \{v' \mid \text{variable } v' \text{ is defined by } i\}$

$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$

$OUT[i] = \bigcup_{s \text{ a successor of } i} IN[s]$

# Liveness analysis revisited

mayPointTo : variable -> set<variable>

mustPointTo: variable -> set<variable>

$GEN[i] = \{ \text{mayPointTo}(v) \cup \text{mustPointTo}(v) \mid$   
what is referenced by variable  $v$  is read by  $i$  }

$KILL[i] = \{ \text{mustPointTo}(v) \mid$   
what is referenced by variable  $v$  is defined by  $i$  }

$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$

$OUT[i] = \bigcup_{s \text{ a successor of } i} IN[s]$

# Trivial analysis: no code analysis

```
int x, y;
```

```
int *p;
```

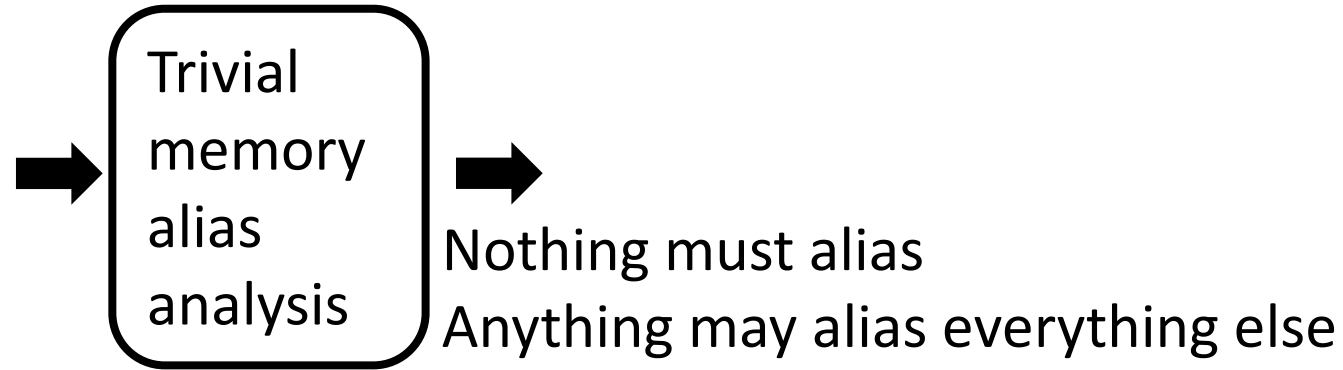
```
... = &x;
```

```
x = 5;
```

```
...(no uses/definitions of x)
```

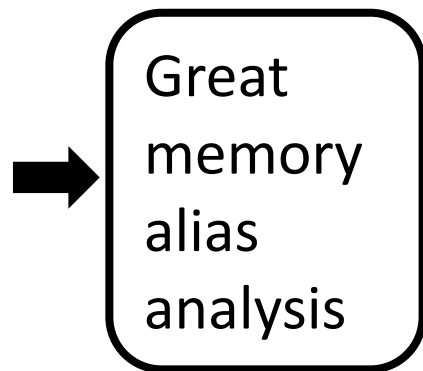
```
*p = 42;
```

```
y = x + 1;
```


$$\text{GEN}[i] = \{\text{mayPointTo}(v) \cup \text{mustPointTo}(v) \mid v \text{ is "used" by } i\}$$
$$\text{KILL}[i] = \{\text{mustPointTo}(v) \mid v \text{ is "defined" by } i\}$$
$$\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])$$
$$\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]$$

# Great alias analysis impact

```
int x, y;  
int *p;  
... = &x;
```



No aliases

Some compilers expose only data dependences.  
How can we compute aliases for them?

~~x = 5;~~

...(no uses/definitions of x)

\*p = 42;

y = ~~x~~<sub>5</sub> + 1;

$GEN[i] = \{ \text{mayPointTo}(v) \cup \text{mustPointTo}(v) \mid v \text{ is "used" by } i \}$

$KILL[i] = \{ \text{mustPointTo}(v) \mid v \text{ is "defined" by } i \}$

$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$

$OUT[i] = \bigcup_{s \text{ a successor of } i} IN[s]$



# Outline

- Enhance CAT with alias analysis
- Simple alias analysis
- Alias analysis in LLVM

# Memory alias analysis

- **Assumption:**

no dynamic memory, pointers can point only to variables

- **Goal:**

at each program point, compute set of (p->x) pairs  
if p points to variable x

- **Approach:**

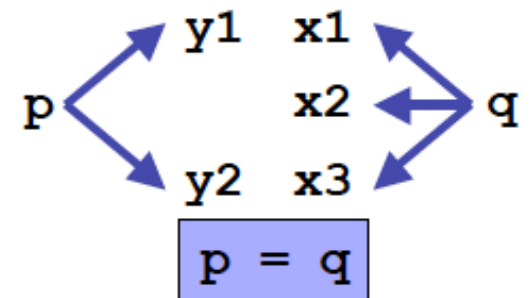
- Based on data-flow analysis
- May information

```
1: p = &x ;  
2: q = &y;  
3: if (...){  
4:   z = &v;  
   }  
5: x++;  
6: p = q;  
7: print *p
```

# May points-to analysis

Which variable does p point to?  $\longrightarrow$  print \*p

- Data flow values:
  - $\{(v, x) \mid v \text{ is a pointer variable and } x \text{ is a variable}\}$
- Direction: forward
- i:  $p = \&x$ 
  - $GEN[i] = \{(p, x)\}$        $KILL[i] = \{(p, v) \mid v \text{ escapes}\}$
  - $OUT[i] = GEN[i] \cup (IN[i] - KILL[i])$
- $IN[i] = \bigcup_{p \text{ is a predecessor of } i} OUT[p]$       Why?
- Different  $OUT[i]$  equation for different instructions
- i:  $p = q$ 
  - $GEN[i] = \{\}$        $KILL[i] = \{(p, x) \mid x \text{ escapes}\}$
  - $OUT[i] = \{(p, z) \mid (q, z) \in IN[i]\} \cup (IN[i] - KILL[i])$



# Code example

1: p = &x ;

2: q = &y;

3: if (...){

4: z = &v;

}

5: x++;

6: p = q;

GEN[1] = {(p, x)}

GEN[2] = {(q, y)}

GEN[3] = { }

GEN[4] = {(z, v)}

GEN[5] = { }

GEN[6] = { }

KILL[1] = {(p, x), (p, y), (p,v)}

KILL[2] = {(q, x), (q, y), (q,v)}

KILL[3] = { }

KILL[4] = {(z, x), (z, y), (z, v)}

KILL[5] = { }

KILL[6] = {(p, x), (p, y), (p, v)}

IN[1] = { }

IN[2] = {(p,x)}

IN[3] = {(q,y),(p,x)}

IN[4] = {(q,y),(p,x)}

IN[5] = {(z,v),(q,y),(p,x)}

IN[6] = {(z,v),(q,y),(p,x)}

OUT[1] = {(p,x)}

OUT[2] = {(q,y),(p,x)}

OUT[3] = {(q,y),(p,x)}

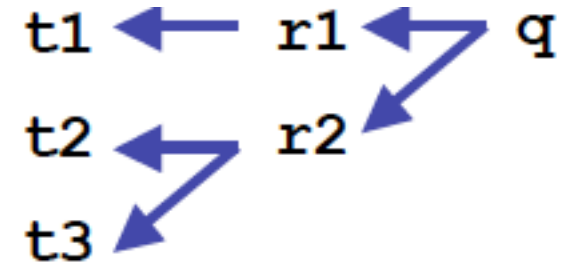
OUT[4] = {(z,v),(q,y),(p,x)}

OUT[5] = {(z,v),(q,y),(p,x)}

OUT[6] = {(z,v),(q,y),(p,y)}

# May points-to analysis

$p = *q$



- $IN[i] = \bigcup_{p \text{ is a predecessor of } i} OUT[p]$
- $i: p = \&x$ 
  - $GEN[i] = \{(p,x)\}$        $KILL[i] = \{(p,v) \mid v \text{ "escapes"}\}$
  - $OUT[i] = GEN[i] \cup (IN[i] - KILL[i])$
- $i: p = q$ 
  - $GEN[i] = \{\}$        $KILL[i] = \{(p,x) \mid x \text{ escapes}\}$
  - $OUT[i] = \{(p,z) \mid (q,z) \in IN[i]\} \cup (IN[i] - KILL[i])$
- $i: p = *q$ 
  - $GEN[i] = \{\}$        $KILL[i] = \{(p,x) \mid x \text{ escapes}\}$
  - $OUT[i] = \{(p,t) \mid (q,r) \in IN[i] \ \& \ (r,t) \in IN[i]\} \cup (IN[i] - KILL[i])$
- $i: *q = p$       **?? (1 point)**

- This was a reasonable alias analysis for understanding pointers that could point only to variables
- How about pointers that could point to memory locations?  
(stack and heap)
  - Challenge: memory locations don't have pre-defined symbols like variables

# Memory alias analysis: dealing with dynamically allocated memory

- Each invocation of a memory allocator creates a new piece of memory

```
p = new T();           p = malloc(10);
```

- Simple solution: generate a new “variable” for every DFA iteration to stand for new memory

```
for (i=0; i < 10; i++){  
    v[i] = new malloc(100);  
}
```

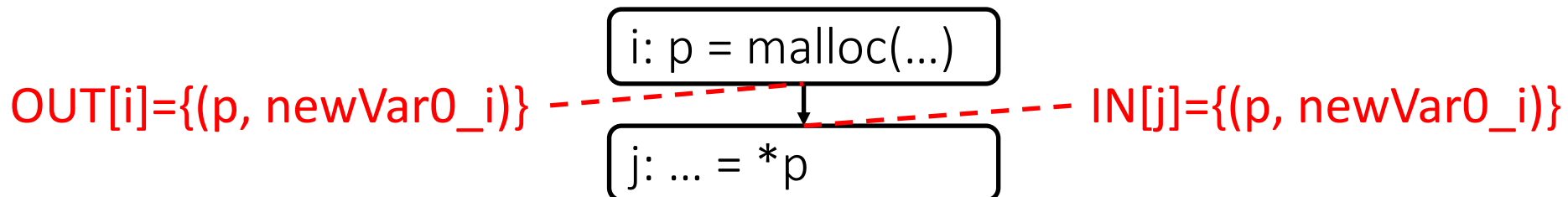
# Memory alias analysis: dealing with dynamically allocated memory

- Each invocation of a memory allocator creates a new piece of memory

`p = new T();`      `p = malloc(10);`

- Simple solution: generate a new “variable” for every DFA iteration to stand for new memory
- Extending our data-flow analysis

$$\text{OUT}[i] = \{(p, \text{newVar})\} \cup (\text{IN}[i] - \{(p, x) \text{ for all } x\})$$





# Memory alias analysis: dealing with dynamically allocated memory

- Each invocation of a memory allocator creates a new piece of memory

`p = new T();`      `p = malloc(10);`

- Simple solution: generate a new “variable” for every DFA iteration to stand for new memory
- Extending our data-flow analysis

$OUT[i] = \{(p, newVar)\} \cup (IN[i] - \{(p,x) \text{ for all } x\})$

$IN[j] = \{(p, newVar0\_i), (q, newVar0\_k), (w, newVar0\_i), (w, newVar0\_k)\}$

`i: p = malloc(...)`

`k: q = malloc(...)`

`z: w = phi([p,left],[q,right])`  
`j: ... = *w`

$IN[z] = \{(p, newVar0\_i), (q, newVar0\_k)\}$

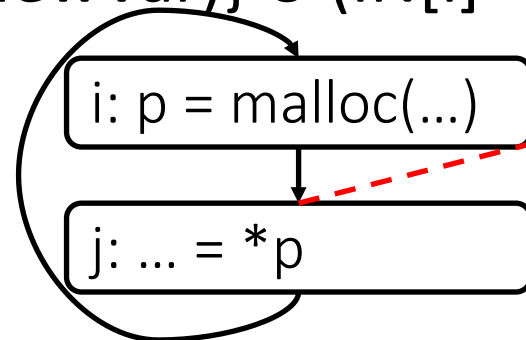
# Memory alias analysis: dealing with dynamically allocated memory

- Each invocation of a memory allocator creates a new piece of memory

`p = new T();`      `p = malloc(10);`

- Simple solution: generate a new “variable” for every DFA iteration to stand for new memory
- Extending our data-flow analysis

$$\text{OUT}[i] = \{(p, \text{newVar})\} \cup (\text{IN}[i] - \{(p, x) \text{ for all } x\})$$



$\text{IN}[j] = \{(p, \text{newVar0}_i),$   
 $(p, \text{newVar1}_i),$   
 $(p, \text{newVar2}_i),$   
 $\dots$

# Memory alias analysis: dealing with dynamically allocated memory

- Each invocation of a memory allocator creates a new piece of memory

`p = new T();`      `p = malloc(10);`

- Simple solution: generate a new “variable” for every DFA iteration to stand for new memory
- Extending our data-flow analysis

$$\text{OUT}[i] = \{(p, \text{newVar})\} \cup (\text{IN}[i] - \{(p, x) \text{ for all } x\})$$

- Problem:
  - Domain is unbounded
  - Iterative data-flow analysis may not converge

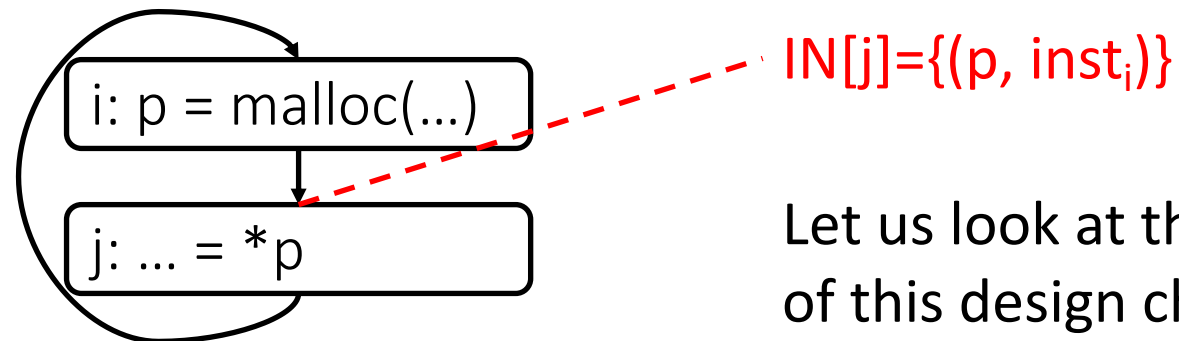
# Memory alias analysis: dealing with dynamically allocated memory

## Simple solution

- Create a summary “variable” for each allocation statement
  - Domain is now bounded
- Data-flow equation

i: p = new T

$OUT[i] = \{(p, inst_i)\} \cup (IN[i] - \{(p, x) \text{ for all } x\})$



$IN[j] = \{(p, inst_i)\}$

Let us look at the implication  
of this design choice

# Memory alias analysis: dealing with dynamically allocated memory

## Simple solution

- Create a summary “variable” for each allocation statement
  - Domain is now bounded
- Data-flow equation

i: p = new T

$OUT[i] = \{(p, inst_i)\} \cup (IN[i] - \{(p, x) \text{ for all } x\})$

for (i=0; i < 10; i++) v[i] = new malloc(100);

\*(v[0]) = ...

\*(v[1]) = ...

**Alias analysis result:**

v[i] and v[j] alias

**Dependence analysis result:**

These 2 instructions depend on each other

# Memory alias analysis: dealing with dynamically allocated memory

## Simple solution

- Create a summary “variable” for each allocation statement
  - Domain is now bounded
- Data-flow equation

i: p = new T

$$\text{OUT}[i] = \{(p, \text{inst}_i)\} \cup (\text{IN}[i] - \{(p, x) \text{ for all } x\})$$

## Alternatives

- Summary variable for odd iterations, summary variable for even iterations
- Summary variable for entire heap
- Summary node for each object type

**Analysis time/precision tradeoff**

# Alias analysis common tradeoffs

- Field sensitivity

  - obj->field1

  - obj->field2

- Flow sensitivity

- Context sensitivity

# Representations of aliasing

## **Alias pairs**

- Pairs that refer to the same memory
- High memory requirements

## **Equivalence sets**

- All memory references in the same set are aliases

## **Points-to pairs**

- Pairs where the first member points to the second



# How hard is the memory alias analysis problem?

- Undecidable
  - Landi 1992
  - Ramalingan 1994
- All solutions are conservative approximations
  - But all correct
- Is this problem solved?
  - Numerous papers in this area
  - Haven't we solved this problem yet? [Hind 2001]

# Alias analyses challenges

- So far we saw only one challenge: dynamic memory allocations

Let's see the other challenges

# Limits of intra-procedural analysis

```
foo() {  
  int x, y, a;  
  int *p;  
  x = 5;  
  p = foo(&x);  
  ...  
}
```

```
foo(int *p){  
  return p;  
}
```

**Does the function call modify x? where does p point to?**

- With our intra-procedural analysis, we don't know
- Make worst case assumptions
  - Assume that any reachable pointer may be changed
  - Pointers can be “reached” via globals and parameters
  - Pointers can be passed through objects in the heap
  - p may point to anything that might escape foo

The most accurate analyses are inter-procedural

# Quality of memory alias analysis

- Quality decreases
  - Across functions
  - When indirect access pointers are used
  - When dynamically allocated memory is used
  - When pointer arithmetic is used
  - When pointer to/from integer casting is used
- Partial solutions to mitigate them
  - Inter-procedural analysis
  - Shape analysis

# Outline

- Enhance CAT with alias analysis
- Simple alias analysis
- **Alias analysis in LLVM**

# What is available in LLVM

- LLVM includes several alias analyses
- Each one is specialized to understand a different code pattern
- Each one with its tradeoff between accuracy and analysis time

# Using dependence analysis in LLVM

Nothing must alias

Anything may alias everything else

```
int x, y;
```

```
int *p;
```

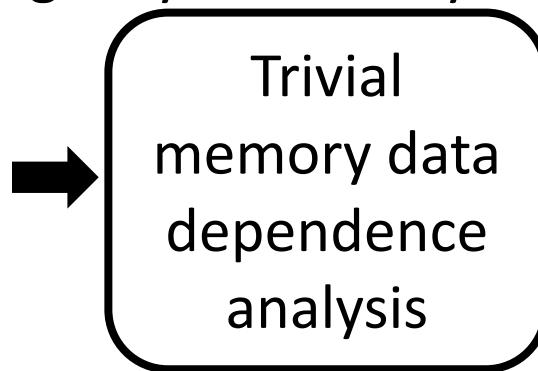
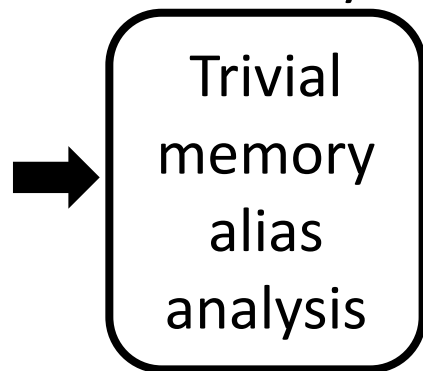
```
... = &x;
```

```
x = 5;
```

```
...(no uses/definitions of x)
```

```
*p = 42;
```

```
y = x + 1;
```



Every memory instruction depends on every instruction that might access memory

```
opt -no-aa -CAT bitcode.bc -o optimized_bitcode.bc
```

# LLVM alias analysis: basic

- Distinct globals, stack allocations, and heap allocations can never alias
  - `p = &g1 ; q = &g2;`
  - `p = alloca(...); q = alloca(...);`
  - `p = malloc(...); q = malloc(...);`
- They also never alias `nullptr`
- Different fields of a structure do not alias
- Baked in information about common standard C library functions
- ... a few more ...



# Using basicaa

```
int x, y;
```

```
int *p;
```

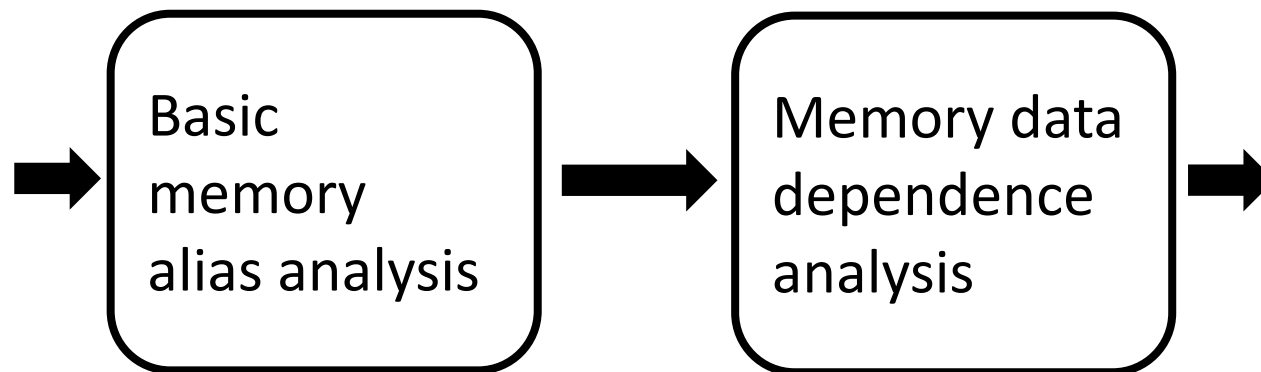
```
... = &x;
```

```
x = 5;
```

```
...(no uses/definitions of x)
```

```
*p = 42;
```

```
y = x + 1;
```



```
opt -no-aa -CAT bitcode.bc -o optimized_bitcode.bc
```

```
opt -basicaa -CAT bitcode.bc -o optimized_bitcode.bc
```

# LLVM alias analysis: globals-aa

- Specialized for understanding reads/writes of globals
  - Analyze only globals that don't have their address taken
- Context-sensitive
- Provide information for call instructions
  - e.g., does call i read/write global g1?

```
int g1;  
int g2;  
void f (void *p1){  
    ... = &g2;  
    g(p1);  
    ...  
}
```

# Using globals-aa

```
int x, y;
```

```
int *p;
```

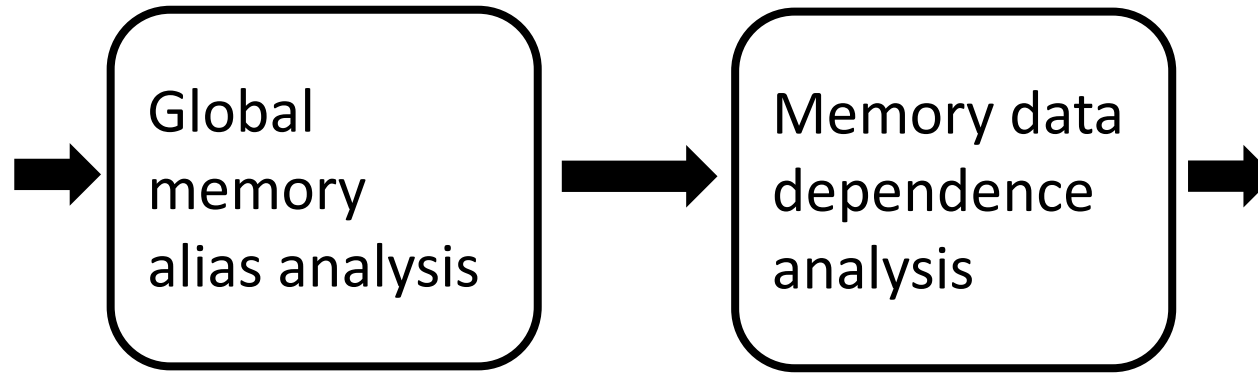
```
... = &x;
```

```
x = 5;
```

```
...(no uses/definitions of x)
```

```
*p = 42;
```

```
y = x + 1;
```



```
opt -globals-aa -CAT bitcode.bc -o optimized_bitcode.bc
```

- basicaa, globals-aa have their strengths and weaknesses
- We would like to use both of them!
- LLVM can chain alias analyses 😊
  - Best of N

# Using basicaa and globals-aa

```
int x, y;  
int *p;  
... = &x;  
x = 5;  
...(no uses/definitions of x)  
*p = 42;  
y = x + 1;
```



```
opt -basicaa -globals-aa -CAT bitcode.bc -o optimized_bitcode.bc
```

# Other LLVM alias analyses

- tbaa
  - cfl-steens-aa
  - scev-aa
  - cfl-anders-aa
- 
- + others not included in the official LLVM codebase

# Alias analyses used

- How can we find out what AA is used in O0/O1/O2/O3?
  - `opt -O3 -disable-output -debug-pass=Arguments bitcode.bc`
- -O0:
- -O1: `-basicaa -globals-aa -tbaa`
- -O2: `-basicaa -globals-aa -tbaa`
- -O3: `-basicaa -globals-aa -tbaa`
- You can always extend O3 adding other AA

- We have seen how to invoke alias analyses
- How can we access alias information and/or dependences in a pass?
- What does "alias" mean in LLVM exactly?  
What is the memory model adopted by LLVM?



- We have seen how to invoke alias analyses
- How can we access alias information and/or dependences in a pass?
- What does "alias" mean in LLVM exactly?  
What is the memory model adopted by LLVM?

# Asking LLVM to run an AA before our pass

```
void getAnalysisUsage(AnalysisUsage &AU) const override {  
    AU.addRequired< AAResultsWrapperPass >();  
    return;  
}
```

Which AA will run?

```
opt -basicaa -CAT bitcode.bc -o optimized_bitcode.bc
```

```
opt -globals-aa -CAT bitcode.bc -o optimized_bitcode.bc
```

```
opt -basicaa -globals-aa -CAT bitcode.bc -o optimized_bitcode.bc
```

# AliasAnalysis LLVM class

- Interface between passes that use the information about pointer aliases and passes that compute them (i.e., alias analyses)
- To access the result of alias analyses:

```
bool runOnFunction (Function &F) override {  
    AliasAnalysis &aliasAnalysis = getAnalysis< AAResultsWrapperPass >().getAAResults();
```

- AliasAnalysis provides information about pointers used by F
- You cannot use the AA results to check aliases of other functions

# AliasAnalysis LLVM class: queries

You can ask to AliasAnalysis the following common queries:

- *Do these two memory pointers alias?* `alias(...)`

<pre>(*p1) = ... ...     = *p2</pre>
--

- Can this instruction read/write a given memory location? `getModRefInfo(...)`
  - Can this function call read/write a given memory location?
- Does this function reads/modifies memory at all?
- Does this function call read/write memory at all?

# AliasAnalysis LLVM class: the memory location

- Memory location representation:
  - Starting address (Value \*)
  - Static size (e.g., 10 bytes)

```
p1 = malloc(sizeof(T1));
```

- From instruction/pointer to the memory location accessed
  - MemoryLocation::get(memInst)

# AliasAnalysis LLVM class: the alias method

- Query: the alias method  
aliasAnalysis.alias(...)

Input: 2 memory locations

```
1 ; ModuleID = 'program.bc'  
2 source_filename = "program.c"  
3 target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
```

```
aliasAnalysis.alias(pointer, memoryLocationSize, pointer2, memoryLocationSize2);
```

- The size can be platform dependent: ... = malloc(sizeof(long int))

```
if (auto pointerType = dyn_cast<PointerType>(pointer->getType())){  
    auto elementPointedType = pointerType->getElementType();  
    if (elementPointedType->isSized()){  
        size = currM->getDataLayout().getTypeStoreSize(elementPointedType);  
    }  
}
```

# AliasAnalysis LLVM class: the alias method

- Query: the alias method

aliasAnalysis.alias(...)

Input: 2 memory locations

```
aliasAnalysis.alias(pointer, memoryLocationSize, pointer2, memoryLocationSize2);
```

- What if you don't know the size of the memory location?

```
if (auto pointerType = dyn_cast<PointerType>(pointer->getType())){  
    auto elementPointedType = pointerType->getElementType();  
    if (elementPointedType->isSized()){  
        size = currM->getDataLayout().getTypeStoreSize(elementPointedType);  
    }  
}
```

# AliasAnalysis LLVM class: the alias method

- Query: the alias method

aliasAnalysis.alias(...)

Input: 2 memory locations

```
aliasAnalysis.alias(pointer, memoryLocationSize, pointer2, memoryLocationSize2);
```

```
aliasAnalysis.alias(pointer, pointer2);
```

Constraint:

Value(s) used in the APIs that are not constant must have been defined in the same function

Output: AliasResult (this is an enum)



# AliasResult

*Two pointers **may** refer to the same memory location*

MayAlias

PartialAlias

*Two pointers **always** refer to the same memory location*

MustAlias

*Two pointers **always** refer to the same memory location and they have the same start address*

NoAlias

*Two pointers **cannot** refer to the same memory location*

# Alias query example

```
switch (aliasAnalysis.alias(pointer, sizePointer, pointer2, sizePointer2)){
  case AliasResult::MayAlias:
    errs() << "  May alias :(\n";
    break ;
  case AliasResult::PartialAlias:
    errs() << "  Partial alias :)\n";
    break ;
  case AliasResult::MustAlias:
    errs() << "  Must alias :)\n";
    break ;
  case AliasResult::NoAlias:
    errs() << "  No alias :)\n";
    break ;
  default:
    abort();
}
```

# Memory instructions

- What if we want to use memory instructions directly?
  - e.g., can this load access the same memory object of this store?

```
switch (aliasAnalysis.alias(MemoryLocation::get(memInst), MemoryLocation::get(memInst2))) {  
  case AliasResult::MayAlias:  
    errs() << "  May alias :(\n";  
    break ;  
  case AliasResult::PartialAlias:  
    errs() << "  Partial alias :)\n";  
    break ;  
  case AliasResult::MustAlias:  
    errs() << "  Must alias :)\n";  
    break ;  
  case AliasResult::NoAlias:  
    errs() << "  No alias :)\n";  
    break ;  
  default:  
    abort();  
}
```

# Mod/ref queries

- Information about whether the execution of an instruction can modify (mod) or read (ref) a memory location
- It is always conservative (like alias queries)
- API: `getModRefInfo`
- This API is often used to understand dependences between function calls or between a memory instruction and a function call

# Mod/ref query example

Input:

- An instruction
- A memory location

... call inst, fence inst, ...

```
aliasAnalysis.getModRefInfo(memInst, pointer, sizePointer);
```

MemoryLocation

Output:

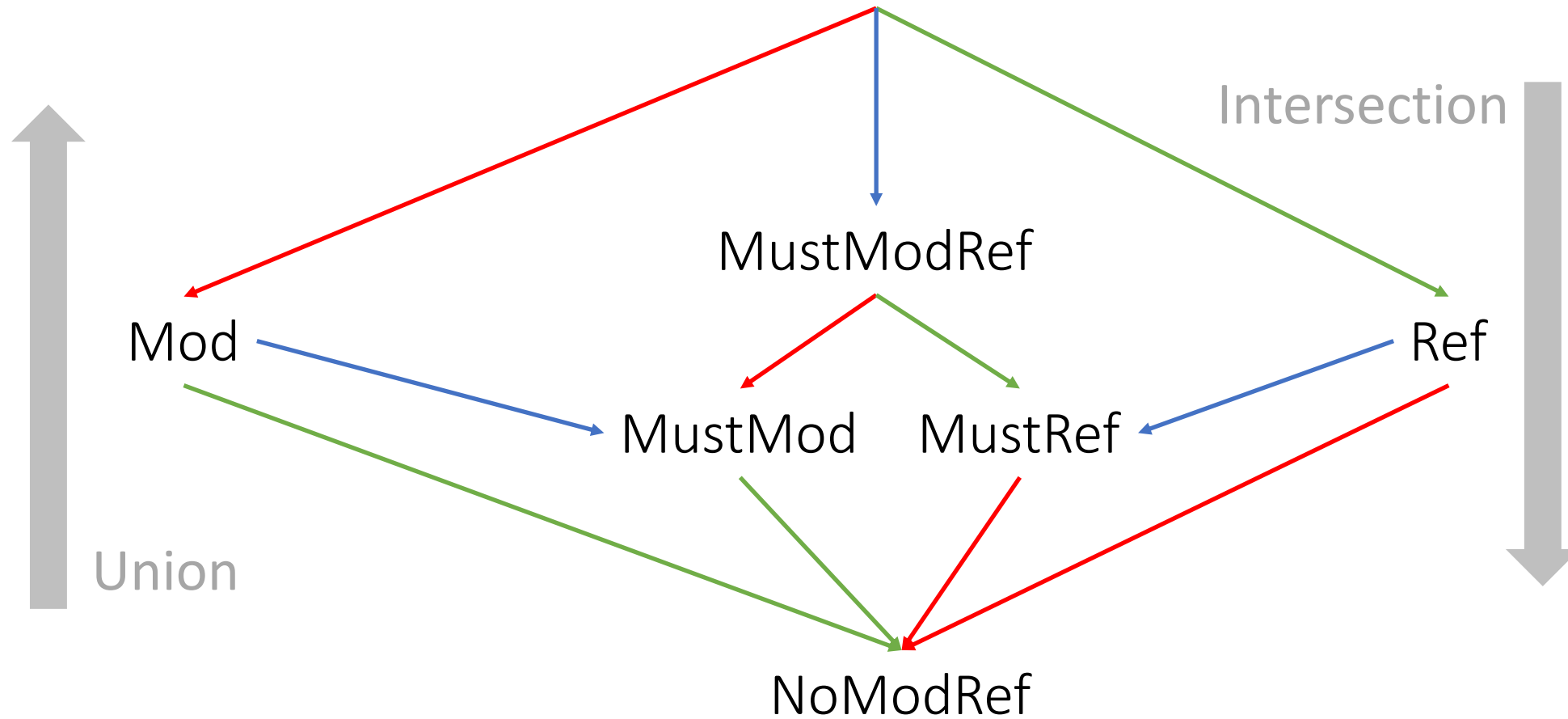
- Whether the memory location **may** be modified and/or **may** be read (the negation of may means cannot)
- ModRefInfo (this is an enum)

```
switch (aliasAnalysis.getModRefInfo(memInst, pointer, pointerSize)){  
  case ModRefInfo::ModRef:  
    break ;  
  case ModRefInfo::Mod:  
    break ;  
  case ModRefInfo::Ref:  
    break ;  
  case ModRefInfo::MustRef:  
    break ;  
  case ModRefInfo::MustMod:  
    break ;  
  case ModRefInfo::MustModRef:  
    break ;  
  case ModRefInfo::NoModRef:  
    break ;  
  default:  
    abort();  
}
```

# ModRefInfo

*(Most conservative output)* ModRef

- Found no ref
- Found no mod
- Found must alias



# Other alias queries

The AliasAnalysis and ModRef API includes other functions

- `canBasicBlockModify(const BasicBlock &BB,  
                          const MemoryLocation &Loc)`
- `pointsToConstantMemory(const MemoryLocation &Loc, ...)`
- `getModRefBehavior`
- ...



# getModRefBehavior

```
FunctionModRefBehavior calleeModRef = aliasAnalysis.getModRefBehavior(calleeFunction);
```

FMRB\_DoesNotAccessMemory

FMRB\_OnlyReadsMemory

FMRB\_OnlyWritesMemory

FMRB\_OnlyReadsArgumentPointees

FMRB\_OnlyAccessesArgumentPointees

FMRB\_OnlyWritesArgumentPointees

FMRB\_OnlyReadsInaccessibleOrArgMem

FMRB\_OnlyReadsInaccessibleMem

FMRB\_OnlyWritesInaccessibleOrArgMem

FMRB\_OnlyWritesInaccessibleMem

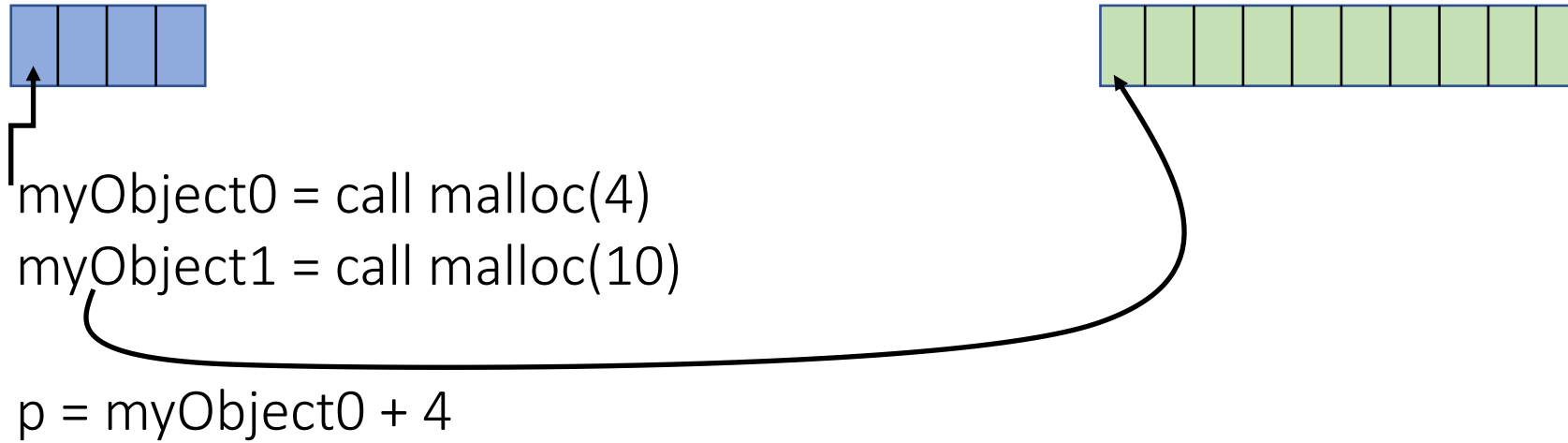
FMRB\_OnlyAccessesInaccessibleMem

FMRB\_OnlyAccessesInaccessibleOrArgMem

FMRB\_UnknownModRefBehavior

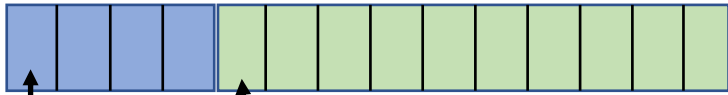
- We have seen how to invoke alias analyses
- How can we access alias information and/or dependences in a pass?
- What does "alias" mean in LLVM exactly?  
What is the memory model adopted by LLVM?

# The LLVM memory model



Can `p` alias `myObject1`?

# The LLVM memory model



myObject0 = call malloc(4)

myObject1 = call malloc(10)

p = myObject0 + 4

Can p alias myObject1?

Always have faith in your ability

Success will come your way eventually

**Best of luck!**