Pointers, Alias & ModRef Analyses

Alina Sbirlea (Google), Nuno Lopes (Microsoft Research)

Joint work with: Juneyoung Lee, Gil Hur (SNU), Ralf Jung (MPI-SWS), Zhengyang Liu, John Regehr (U. Utah)

72239989 Clang++ Compilation Optimization Bug

```
#include <string.h>

int kOne = 1;

__attribute__((weak))
void check(int x) {
  if (x != 9) {
    printf("ERROR: x = %d\n", x);
  }
  exit(0);
}

__attribute__((weak))
void buggy(void) {
  unsigned char dst[1] = {42};
  unsigned char src[1] = {9};
  unsigned char *dp = dst;
  unsigned char *sp = src;

  while (1) {
    if (kOne) {
      dp[0] = 9;
    } else {
      memcpy(dp, sp, 16);
      sp += 16;
      dp += 16;
    }
    check(dst[0]);
  }
```

PR36228: miscompiles Android: API usage mismatch between AA and AliasSetTracker

PR34548: incorrect Instcombine fold of inttoptr/ptrtoint

```
Example of an end-to-end miscompilation by clang

$ cat c.c
#include <stdio.h>

void f(int*, int*);

int main()
{
  int a=0, y[1], x = 0;
  uintptr_t pi = (uintptr_t) &x;
  uintptr_t yi = (uintptr_t) (y+1);
  uintptr_t n = pi != yi;

  if (n) {
    a = 100;
    pi = yi;
  }

  if (n) {
    a = 100;
    pi = (uintptr_t) y;
  }

  *(int *)pi = 15;

  printf("a=%d x=%d\n", a, x);

  f(&x,y);

  return 0;
}
```

```
pub fn test(gp1: &mut usize, gp2: &mut usize, b1:
bool, b2: bool) -> (i32, i32) {
  let mut g = 0;
  let mut c = 0;
  let y = 0;
  let mut x = 7777;
  let mut p = &mut g as *const _;

  {
    let mut q = &mut g;
    let mut r = &mut 8888;

    if b1 {
      p = (&y as *const _).wrapping_offset(1);
    }

    if b2 {
      q = &mut x;
    }

    *gp1 = p as usize + 1234;
    if q as *const _ == p {
      c = 1;
      *gp2 = (q as *const _) as usize + 1234;
      r = q;
    }
    *r = 42;
  }
  return (c, x);
}
```

Safe Rust program miscompiled by GVN

Pointers ≠ Integers

What’s a Memory Model?

```
char *p = malloc(4);
char *q = malloc(4);

q[2] = 0;

p[6] = 1;

print(q[2]);
```

UB?

0 or 1?

1) When is a memory operation UB?
2) What’s the value of a load operation?

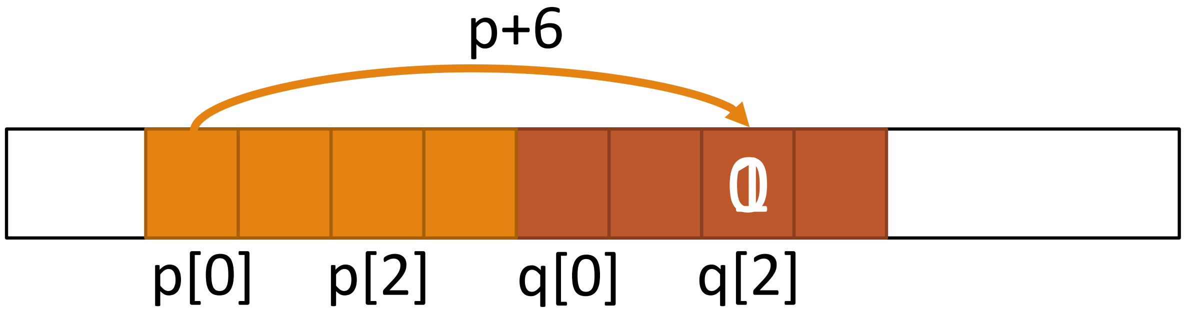
Flat memory model

```
char *p = malloc(4);
char *q = malloc(4);

q[2] = 0;

p[6] = 1;
```

Not UB

```
print(q[2]);
```

print(1)



Simple, but inhibits optimizations!

Two Pointer Types

- Logical Pointers, which originate from allocation functions (malloc, alloca, …):

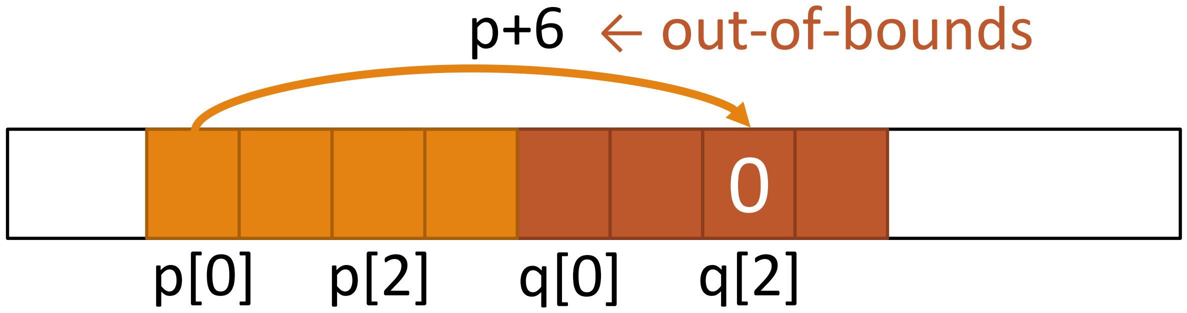
```
char *p = malloc(4);
char *q = p + 2;
char *r = q - 1;
```

- Physical Pointers, which originate from inttoptr casts:

```
int x = ...;
char *p = (char*)x;
char *q = p + 2;
```

Logical Pointers: data-flow provenance

```
char *p = malloc(4);
char *q = malloc(4);
char *q2 = q + 2;
char *p6 = p + 6;

*q2 = 0;
*p6 = 1;  UB

print(*q2);  print(0)
```



Pointer must be inbounds of object found in use-def chain!

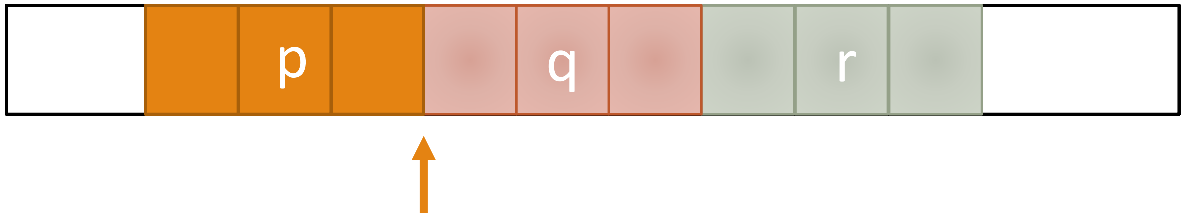
Logical Pointers: simple NoAlias detection

```
char *p = malloc(4);
char *q = malloc(4);

char *p2 = p + ...;
char *q2 = q + ...;
```

Don't alias

If 2 pointers are derived from different objects, they don't alias!

Physical Pointers: control-flow provenance




```
char *p = malloc(3);
char *q = malloc(3);
char *r = malloc(3);
int x = (int)p + 3;
int y = (int)q;

if (x == y) {
   *(char*)x = 1; // OK
}


*(char*)x = 1; // UB
```

Observed address of p (data-flow)

Observed p+n == q (control-flow)

Can’t access r, only p and q

Only p observed; p[3] is out-of-bounds

Physical Pointers: p ≠ (int*)(int)p

```
char *p = malloc(4);
char *q = malloc(4);
int x = (int)p + 4;
int y = (int)q;

*q = 0;

if (x == y)
  *(char*)y = 1;

print(*q); // 0 or 1
```

Ok to replace with q

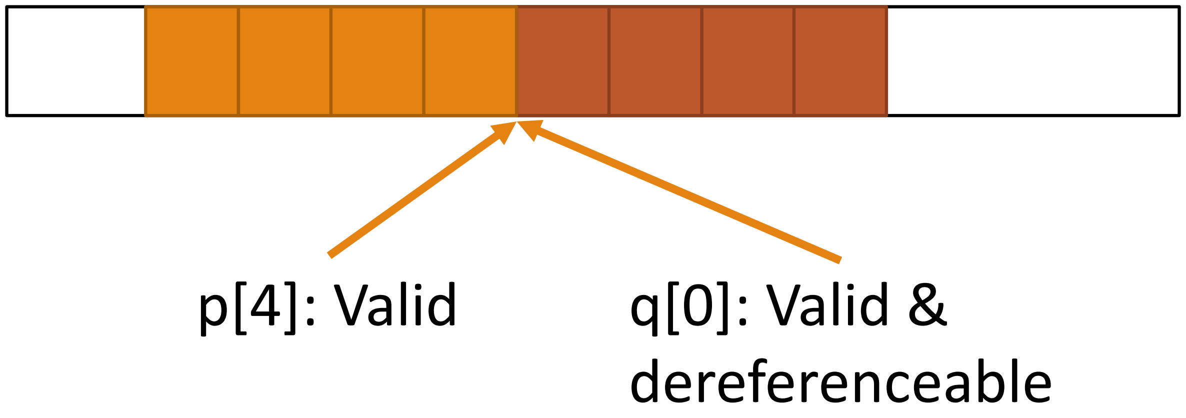
GVN →

```
char *p = malloc(4);
char *q = malloc(4);
int x = (int)p + 4;
int y = (int)q;

*q = 0;

if (x == y)
  *(char*)x = 1;

print(*q); // 0 or 1
```

Not ok to replace with ‘p + 4’

Physical Pointers: p+n and q

```
int x = (int)q; // or p+4

*(char*)x = 0; // q[0]
*(((char*)x)+1) = 0; // q[1]
*(((char*)x)-1) = 0; // p[3]
```



At inttoptr time we don't know which objects the pointer may refer to (1 or 2 objects).

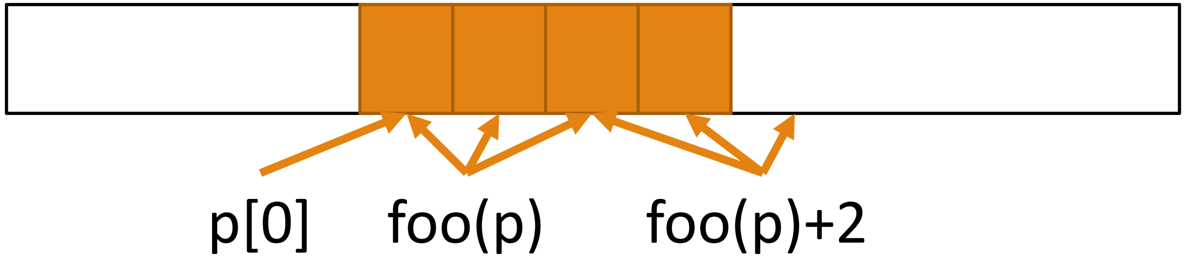
GEP Inbounds

```
%q = getelementptr inbounds %p, 4
```

Both %p and %q must be inbounds of the same object

```
char *p = malloc(4);
char *q = p +inbounds 5;

*q = 0; // UB
```

```
char *p = malloc(4);
char *q = foo(p);
char *r = q +inbounds 2;
p[0] = 0;
*r = 1;
```



Delayed ‘GEP inbounds’ Checking

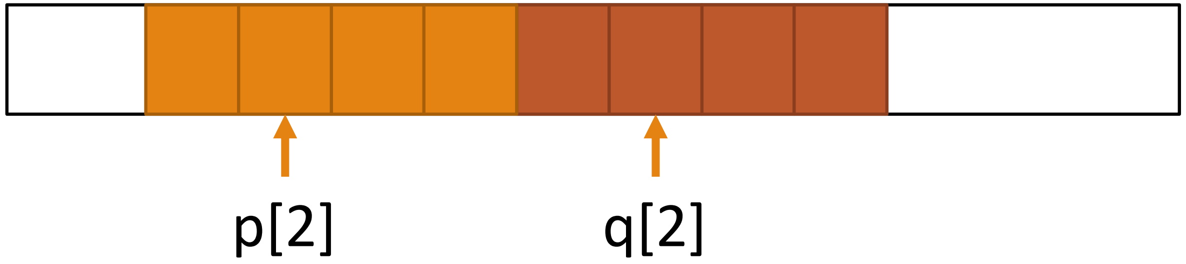
```
char *p = malloc(4);
char *q = p +inbounds 5;  // poison

*q = 0; // UB
```

- Logical pointers: there’s a use-def chain to alloc site, so immediate inbounds check is OK

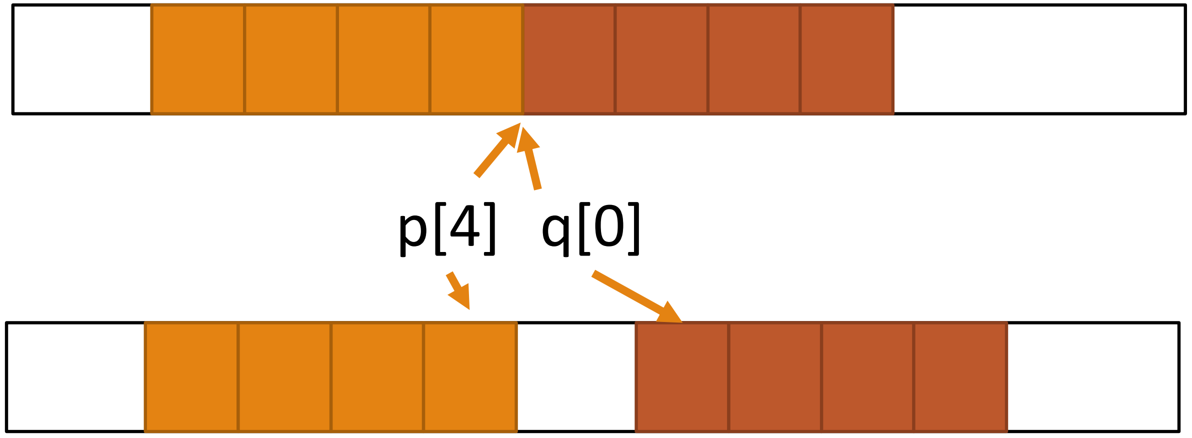
```
char *r = (char*)(int)p;
char *s = r +inbounds 5; // OK

*s = 0; // UB
        // OOB of all observed objects
```

- Physical pointers: there might be no path to alloc; delaying ensures gep doesn’t depend on memory state

No Layout Guessing



Dereferenceable pointers:
p+2 == q+2 is always false



Valid, but not dereferenceable pointers:
p+n == q is undef

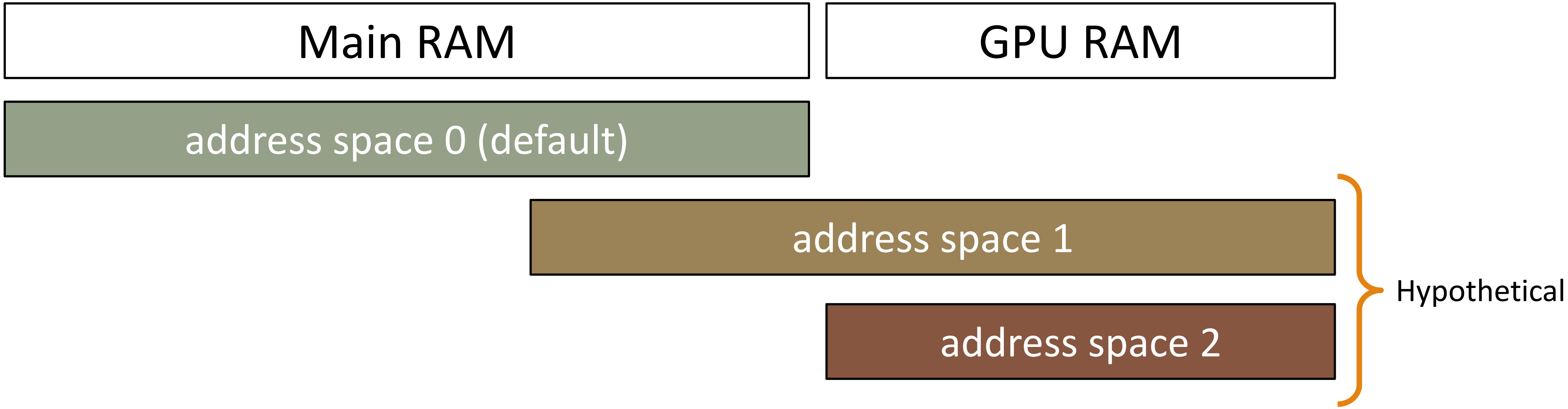
Consequences of Undef Ptr Comparison

```
char *p = ...;
char *q = ...;

if (p == q) {
   // p and q equal or
   // p+n == q (undef)
}
```

- GVN for pointers: not safe to replace p with q unless:
  - q is nullptr (~50% of the cases)
  - q is inttoptr
  - Both p and q are logical and are dereferenceable
  - …

Address Spaces

- Virtual view of the memory(ies)
- Arbitrary overlap between spaces
- (int*)0 not dereferenceable in address space 0



Pointer Subtraction

- Implemented as (int)p – (int)q
- Correct, but loses information vs p – q (only defined for p,q in same object)
- Analyses don't recognize this idiom yet

Malloc and ICmp Movement

- ICmp moves freely
- It's only valid to compare pointers with overlapping liveness ranges
- Potentially illegal to trim liveness ranges

```
char *p = malloc(4);
char *q = malloc(4);

// valid
if (p == q) { ... }

free(p);
```

invalid

```
char *p = malloc(4);
free(p);

char *q = malloc(4);

// poison
if (p == q) { ... }
```

Summary: so far

- Two pointer types:
  - Logical (malloc/alloca): data-flow provenance
  - Physical (inttoptr): control-flow provenance

- p ≠ (int*)(int)p
- There’s no “free” GVN for pointers

Alias Analysis

Alias Analysis queries

- alias()
- getModRefInfo()

AA Query

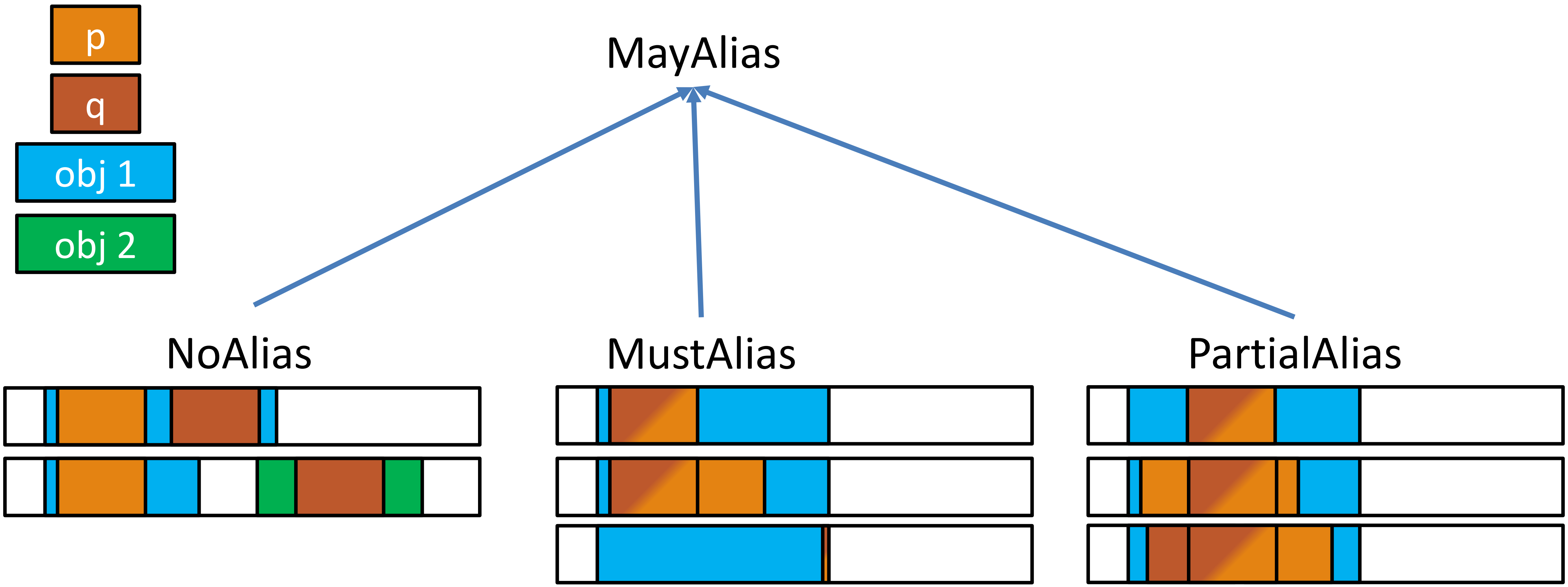
```
char *p = ...;
int  *q = ...;

*p = 0;
*q = 1;
print(*p); // 0 or 1?
```

alias(p, 1 , q, 4) = ?

alias(p, sz_p, q, sz_q)

what's the aliasing between pointers p, q and resp. access sizes sz_p, sz_q

AA Results

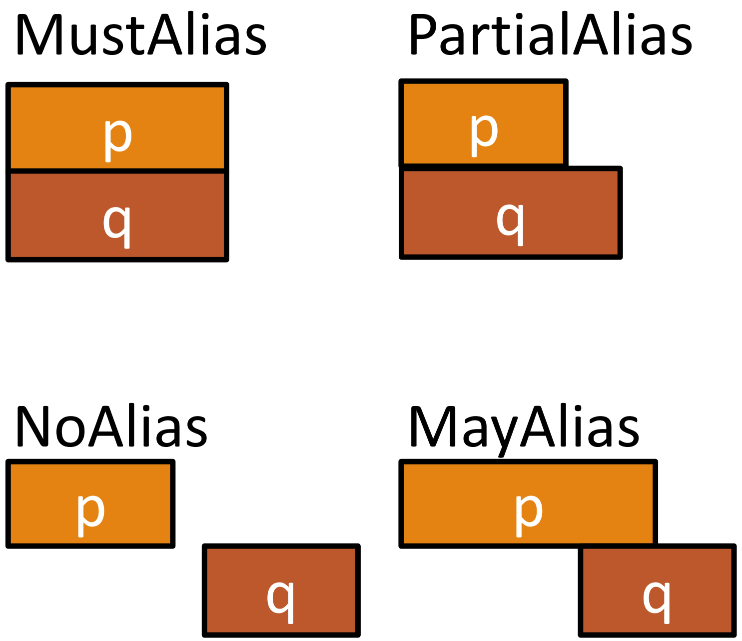


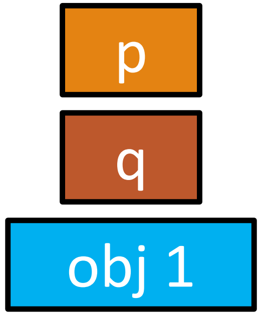
AA caveats

"Obvious" relationships between aliasing queries often don't hold

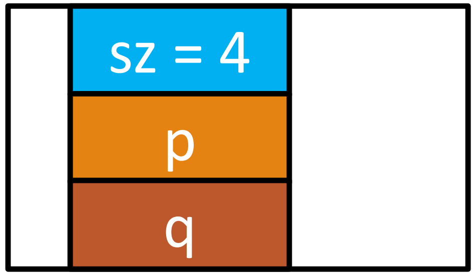
E.g. alias(p, sp,   q, sq)   == MustAlias doesn't imply
        alias(p, sp2, q, sq2) == MustAlias

And: alias(p, sp,   q, sq)   == NoAlias doesn't imply
        alias(p, sp2, q, sq2) == NoAlias





AA results

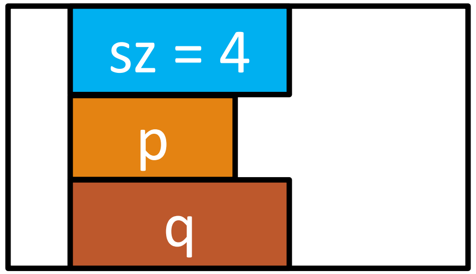


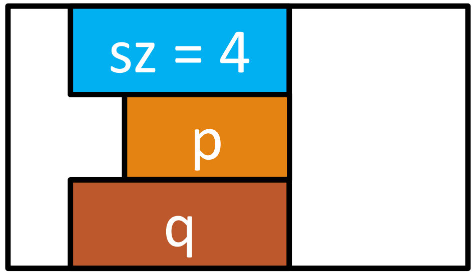

```
char *p = obj + x;
char *q = obj + y;
```

alias(p, 4, q, 4) = MustAlias

access size == object size implies idx == 0

AA results assume no UB.





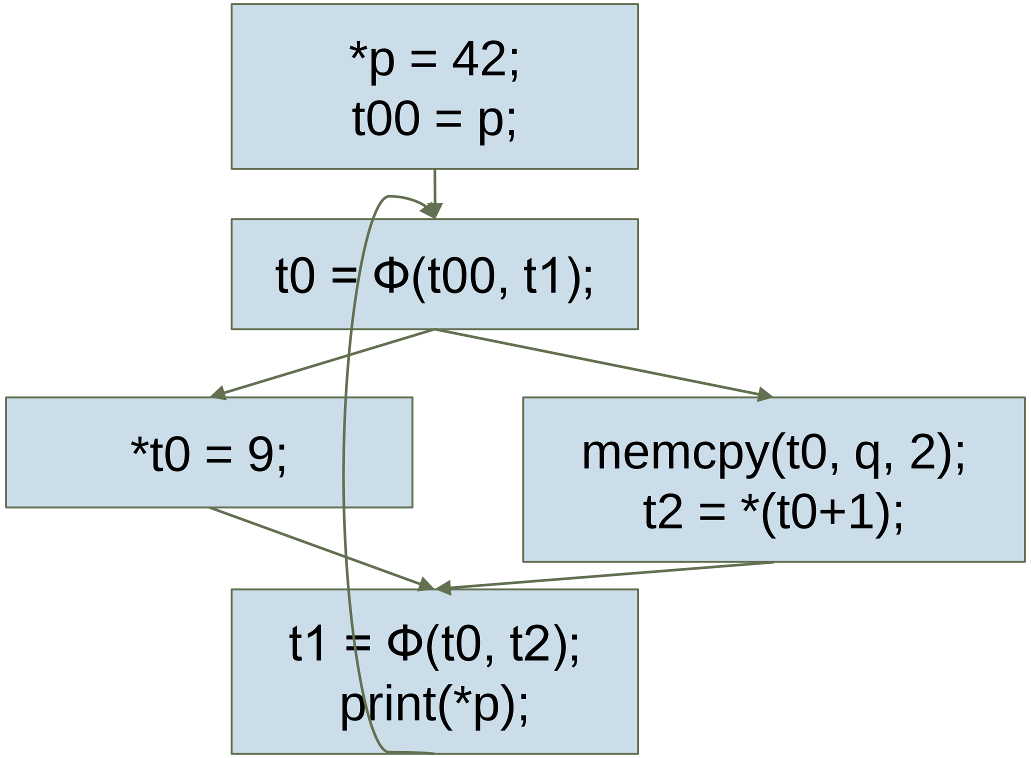
alias(p, 3, q, 4) = PartialAlias

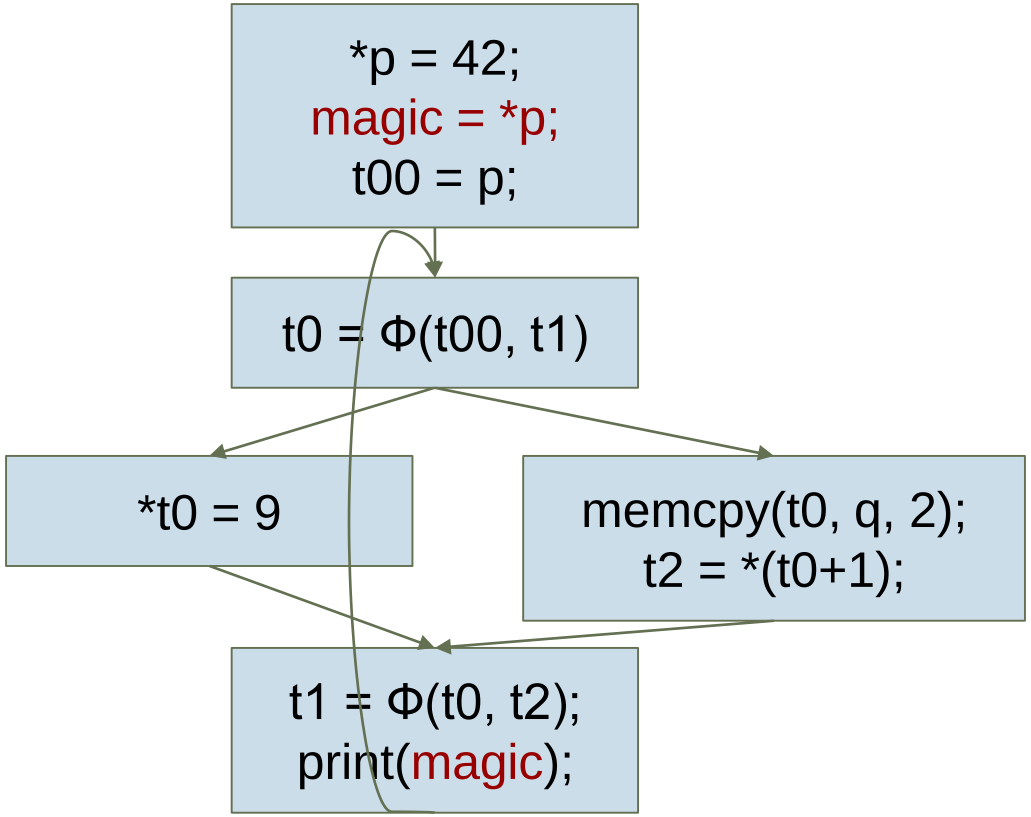
MustAlias requires further information (e.g. know p = q)

AA results are sometimes unexpected and can be overly conservative.

AA must consider UB (PR36228)

```
i8* p = alloca (2);
i8* q = alloca (1);
```





New in AA: precise access size

- Recent API changes introduced two access size types:
  - Precise: when the exact size is known
  - Upper bound: maximum size, but no minimum size guaranteed (can be 0)

- See D45581, D44748

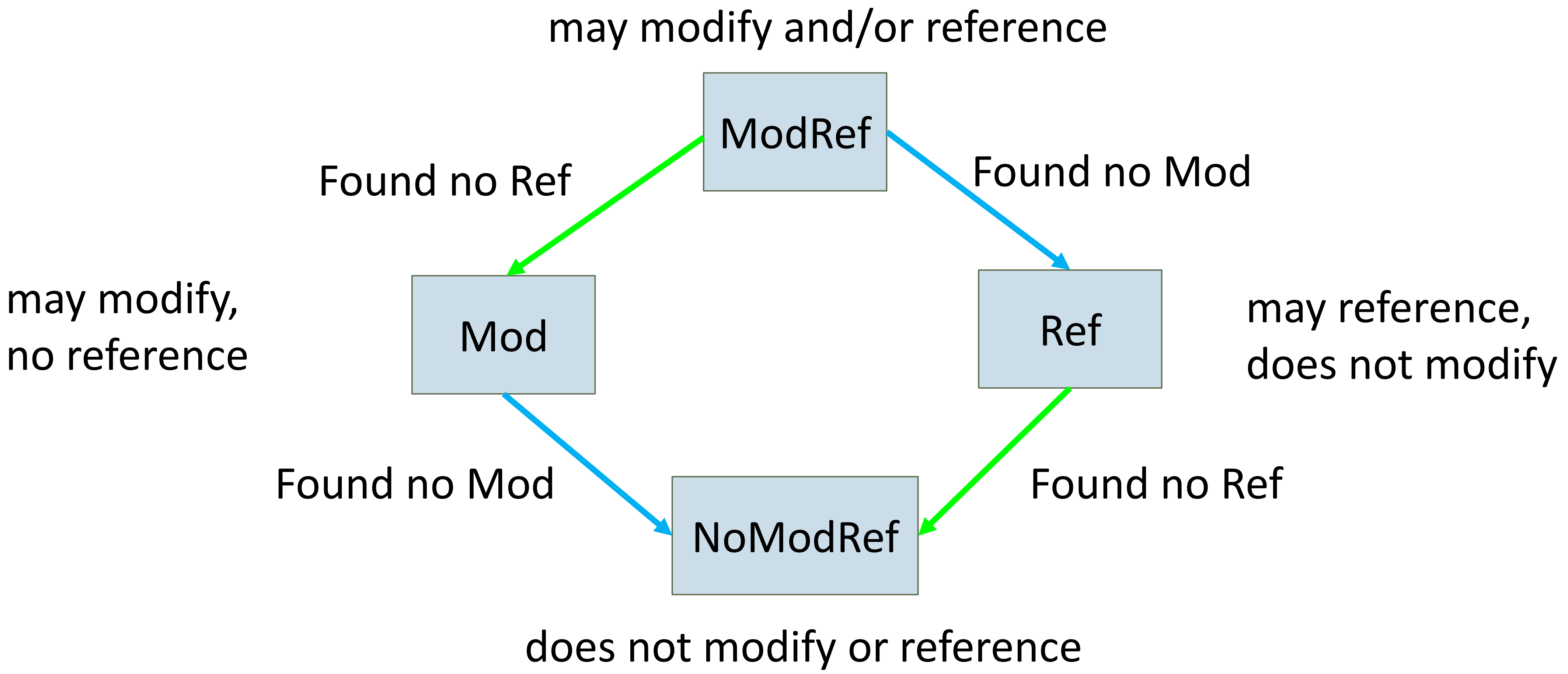
27

ModRef Analysis

ModRefInfo

- How instructions affect memory instructions:
  - Mod = modifies / writes
  - Ref = accesses / reads

ModRefInfo Overview



ModRef Example

```
declare i32 @g(i8*)
declare i32 @h(i8*) argmemonly
```

```
define void @f(i8* %p) {
  %1 = call i32 @g(i8* %p)                ; ModRef %p
  store i8 0, i8* %p                      ; Mod %p (no Ref %p)
  %2 = load i8, i8* %p                    ; Ref %p (no Mod %p)

  %3 = call i32 @g(i8* readonly %p)       ; ModRef %p  (%p may be a global)
  %4 = call i32 @h(i8* readonly %p)       ; Ref %p     (h only accesses args)

  %a = alloca i8
  %5 = call i32 @g(i8* readonly %a)       ; ModRef %a (tough %a doesn't escape)
```

New ModRefInfo API

- Checks:
  - isNoModRef
  - isModOrRefSet
  - isModAndRefSet
  - isModSet
  - isRefSet

- Retrieve ModRefInfo from FunctionModRefBehavior
  - createModRefInfo

- New value generators:
  - setMod
  - setRef
  - setModAndRef
  - clearMod
  - clearRef

  - unionModRef
  - intersectModRef

Using the New ModRef API

```
Result == MRI_NoModRef
```

→

```
isNoModRef(Result)
```

```
if (onlyReadsMemory(MRB))
  Result = ModRefInfo(Result & MRI_Ref);
else if (doesNotReadMemory(MRB))
  Result = ModRefInfo(Result & MRI_Mod);
```

→

```
if (onlyReadsMemory(MRB))
  Result = clearMod(Result);
else if (doesNotReadMemory(MRB))
  Result = clearRef(Result);
```

```
Result = ModRefInfo(Result & ...);
```

→

```
Result = intersectModRef(Result, ...);
```

Using the New ModRef API

```
ModRefInfo ArgMask = getArgModRefInfo(CS1, CS1ArgIdx);
ModRefInfo ArgR = getModRefInfo(CS2, CS1ArgLoc);

if (((ArgMask & MRI_Mod) != MRI_NoModRef &&
     (ArgR & MRI_ModRef) != MRI_NoModRef) ||
     ((ArgMask & MRI_Ref) != MRI_NoModRef &&
     (ArgR & MRI_Mod) != MRI_NoModRef)) {
      ...
}
```

```
ModRefInfo ArgModRefCS1 = getArgModRefInfo(CS1, CS1ArgIdx);
ModRefInfo ModRefCS2 = getModRefInfo(CS2, CS1ArgLoc);

if ((isModSet(ArgModRefCS1) && isModOrRefSet(ModRefCS2)) ||
    (isRefSet(ArgModRefCS1) && isModSet(ModRefCS2))) {
     …
}
```

Why have MustAlias in ModRefInfo?

- AliasAnalysis calls are expensive!
- Avoid double AA calls when ModRef + alias() info is needed.

- Currently used in MemorySSA

Example: promoting call arguments

- Call `foo` is `argmemonly` a
- `isMustSet(getModRefInfo(foo, a))`
- `getModRefInfo(foo, a)` can have both Mod and Ref set.

```
char *a, *b;


for {
  foo (a);
  b = *a + 5;
  *a ++;
}
```

```
char *a, *b, tmp;
// promote to scalar
tmp = *a;
for {
  foo (&tmp);
  b = tmp + 5;
  tmp ++;
}
*a = tmp;
```

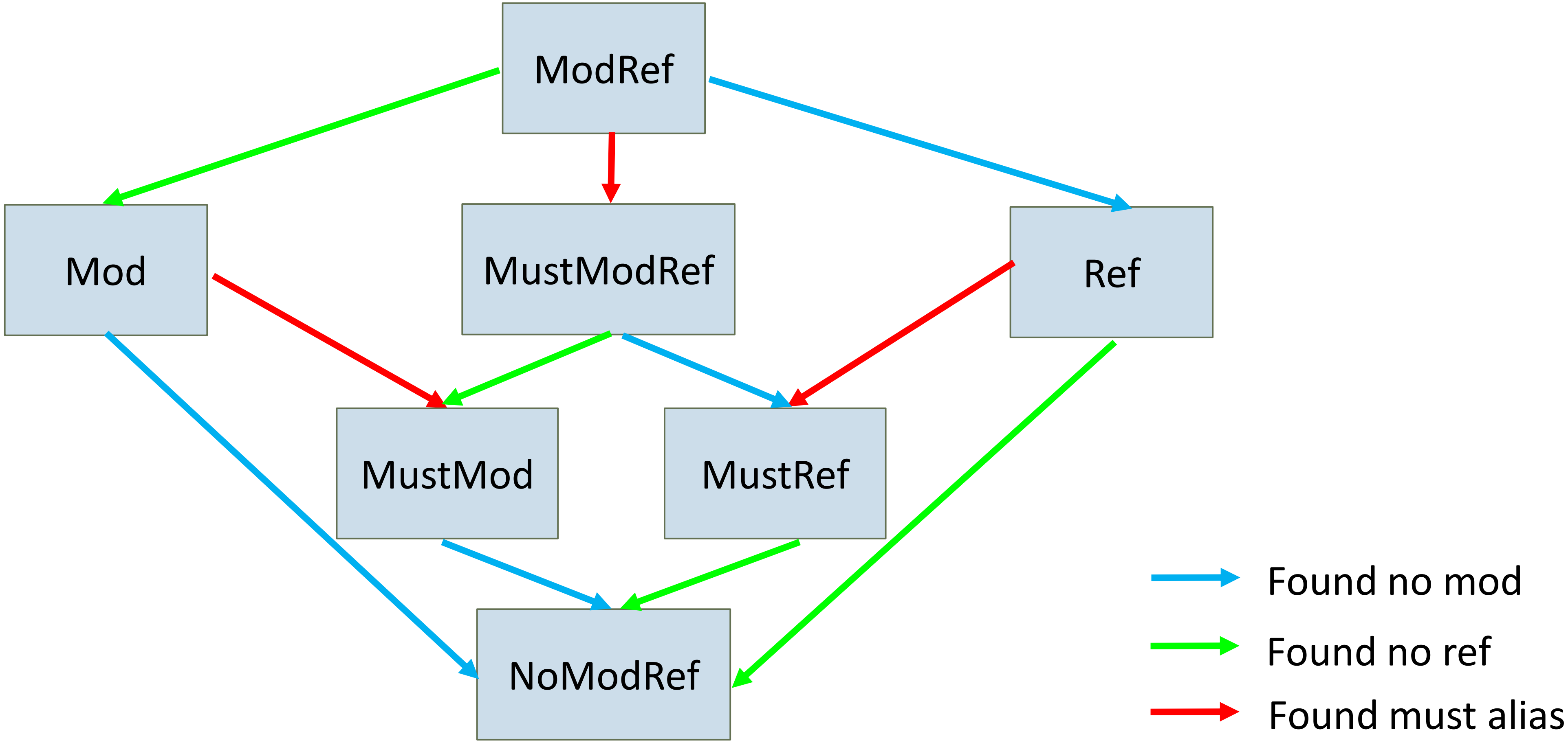
MustAlias can include NoAlias for calls?

- Call `foo` is `argmemonly` a
- `isMustSet(getModRefInfo(foo, a))`
- `getModRefInfo(foo, a)` can have both Mod and Ref set.

```
char *a, *b;
char *c = malloc;


for {
  foo (a, c);
  b = *a + 5;
  *a ++;
}
```

```
char *a, *b, tmp;
char *c = malloc; // noalias(a, c)
// promote to scalar
tmp = *a;
for {
  foo (&tmp, c);
  b = tmp + 5;
  tmp ++;
}
*a = tmp;
```

New ModRef Lattice



Common Misconceptions of Must in ModRefInfo

- MustMod = may modify, must alias found, NOT must modify
    - E.g., foo has readnone attribute => ModRef(foo(a), a) = NoModRef.
- MustRef = may reference, must alias found, NOT must reference
- MustModRef = may modify and may reference, must alias found, NOT must modify and must reference

Key takeaways

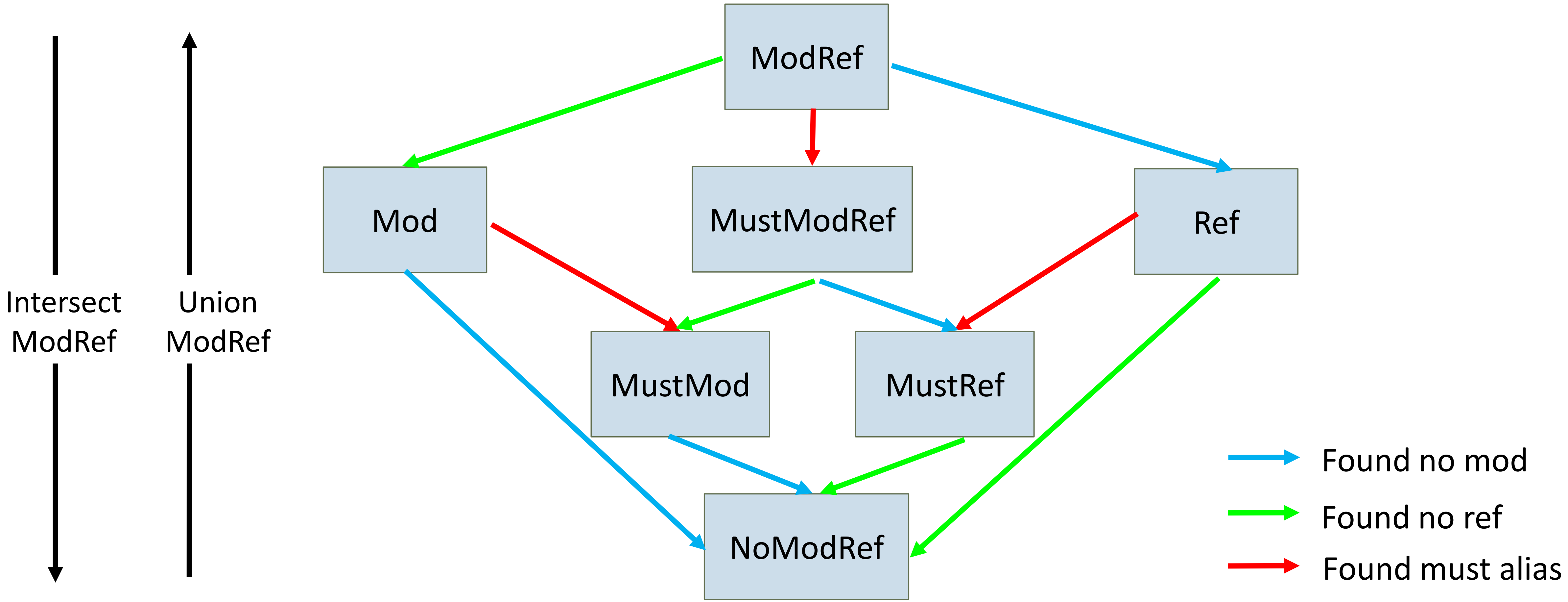
- ModRef is the most general response: may modify or reference
- Mod is cleared when we’re sure a location is not modified
- Ref is cleared when we’re sure a location is not referenced
- Must is set when we’re sure we found a MustAlias
- NoModRef means we’re sure location is neither modified or referenced, i.e. written or read
  - The “Must” bit in the ModRefInfo enum class is provided for completeness, and is not used

ModRefInfo API

- Checks:
  - isNoModRef
  - isModOrRefSet
  - isModAndRefSet
  - isModSet
  - isRefSet
  - isMustSet
- Retrieve ModRefInfo from FunctionModRefBehavior
  - createModRefInfo

- New value generators:
  - setMod
  - setRef
  - setMust
  - setModAndRef
  - clearMod
  - clearRef
  - clearMust
  - unionModRef
  - intersectModRef

New ModRef Lattice



Disclaimers / Implementation details

- GlobalsModRef relies on a certain number of bits available for alignments. To mitigate this, Must info is being dropped.
- FunctionModRefBehavior still relies on bit-wise operations. Changes similar to ModRefInfo may happen in the future.

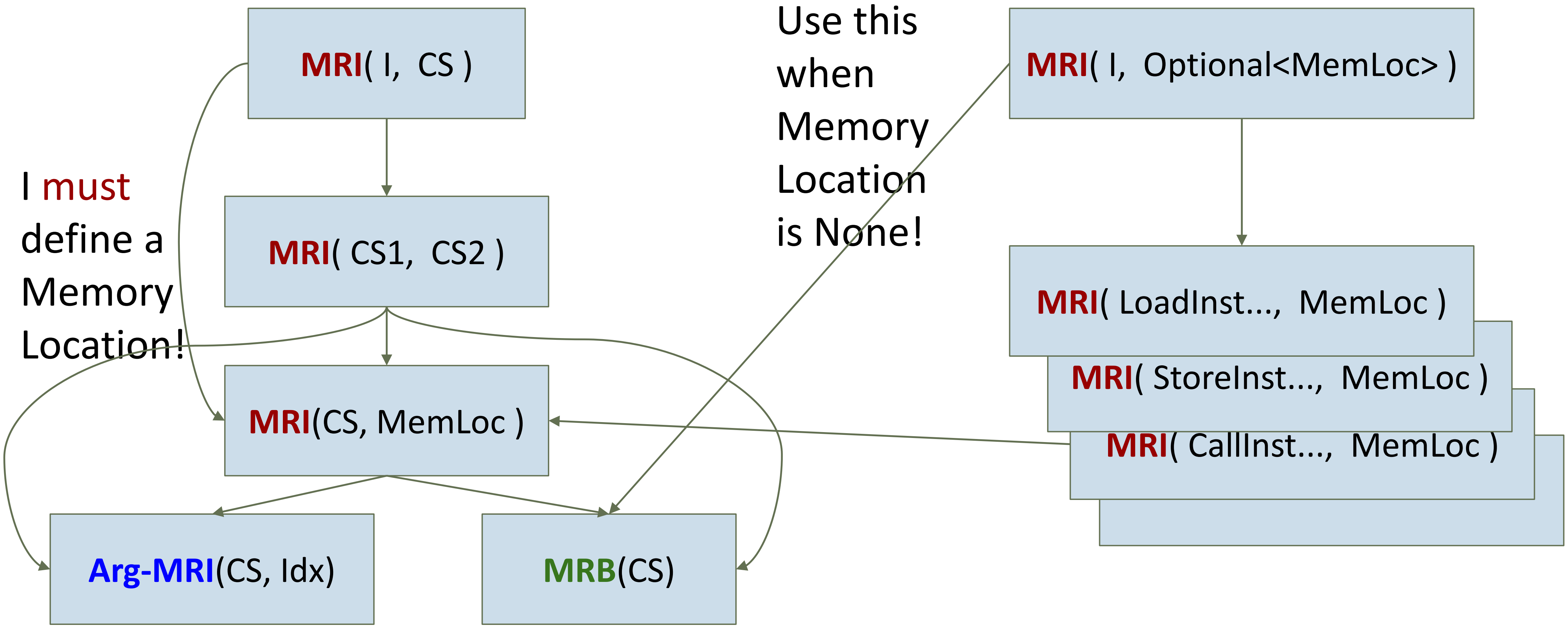
ModRefInfo API overview

- getModRefBehavior (CallSite) → MRB
- getArgModRefInfo (CallSite, ArgIndex) → Arg-MRI
- getModRefInfo(...) → MRI

ModRefInfo API overview

- getModRefBehavior (CallSite) → MRB
- getArgModRefInfo (CallSite, ArgIndex) → Arg-MRI
- getModRefInfo(...) → MRI
  - Instruction, Optional<MemoryLocation>
  - Instruction, CallSite
  - CallSite, CallSite
  - CallSite, MemoryLocation
  - Instruction, CallSite

ModRefInfo API overview



getModRefInfo for instruction I, optional mem. loc

- Special cases memory accessing instructions:
  - LoadInst, StoreInst, CallInst.
- Use ModRefBehavior if I == CS and Loc == None

getModRefInfo for two call sites CS1, CS2

- NoModRef: CS1 does not write to memory CS2 reads or writes
- NoModRef: CS2 does not write to memory CS1 reads or writes
- Ref: CS1 may read memory written by CS2
- Mod: CS1 may write memory read or written by CS2
- ModRef: CS1 may read or write memory read or written by CS2
- Must: is set only if either:
  - CS2 only accesses and modifies arguments & MustAlias is found between CS1 and all CS2 args
  - CS1 only accesses and modifies arguments & MustAlias is found between CS2 and all CS1 args

getModRefInfo for call site CS, memory Loc

- Filter using CS properties
  - CS does not access memory => NoModRef
  - CS does not write => clearMod
  - CS does not read => clearRef
  - CS only accesses arguments, check alias of all arguments against Loc
- Must only set if CS only accesses arguments and MustAlias found with all args.

getModRefInfo for CS, instruction I

- If I is a call, use the getModRefInfo for two call sites CS1, CS2
- If I is a Fence, return *ModRef*
- If I defines a memory location Loc, use getModRefInfo for CS, Loc
  - If I does not define a memory location, this method will assert!
- Default case: *NoModRef* - only taken if above result is *NoModRef*

Assumptions in LLVM

- Cannot allocate > half address space

Summary

Summary: Pointers ≠ Integers

- Two pointer types:
  - Logical (malloc/alloca): data-flow provenance
  - Physical (inttoptr): control-flow provenance

- AA: what's the NoAlias/MustAlias/PartialAlias/MayAlias relation between 2 memory accesses?
- ModRef: what's the (Must)NoModRef/Mod/Ref/ModRef relation between 2 operations?

- p ≠ (int*)(int)p
- There's no "free" GVN for pointers
- Use new pointer analyses APIs to reduce compilation time