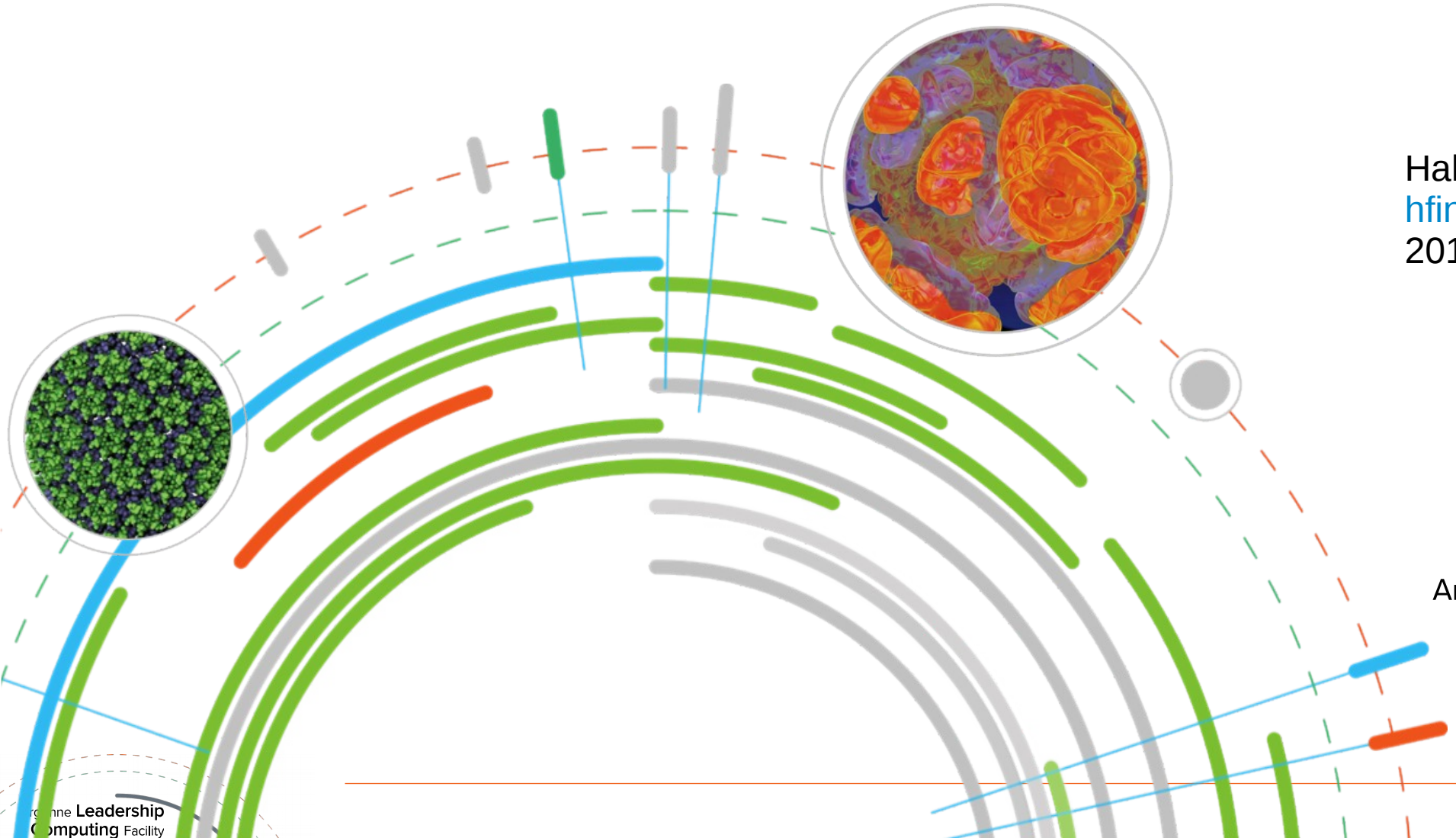
Restrict-qualified pointers in LLVM



Hal Finkel
hfinkel@anl.gov
2017-02-04

Argonne **Leadership Computing** Facility

Argonne NATIONAL LABORATORY

What is “restrict”?

```
void foo(float * restrict x, float * restrict y, float * restrict z, float * restrict v, float * restrict out, int n) {
  ...
}
```

“restrict” only a keyword in C. Use __restrict in C++

Not that restrict goes on the variable, not the pointer type!
Not: restrict float *out

restrict means: Within the scope of the restrict-qualified variable, memory accessed through that pointer, or any pointer based on it, is not accessed through any pointer not based on it.

“based on” and Capturing

In C99, the definition of “based on” for pointers is completely general.

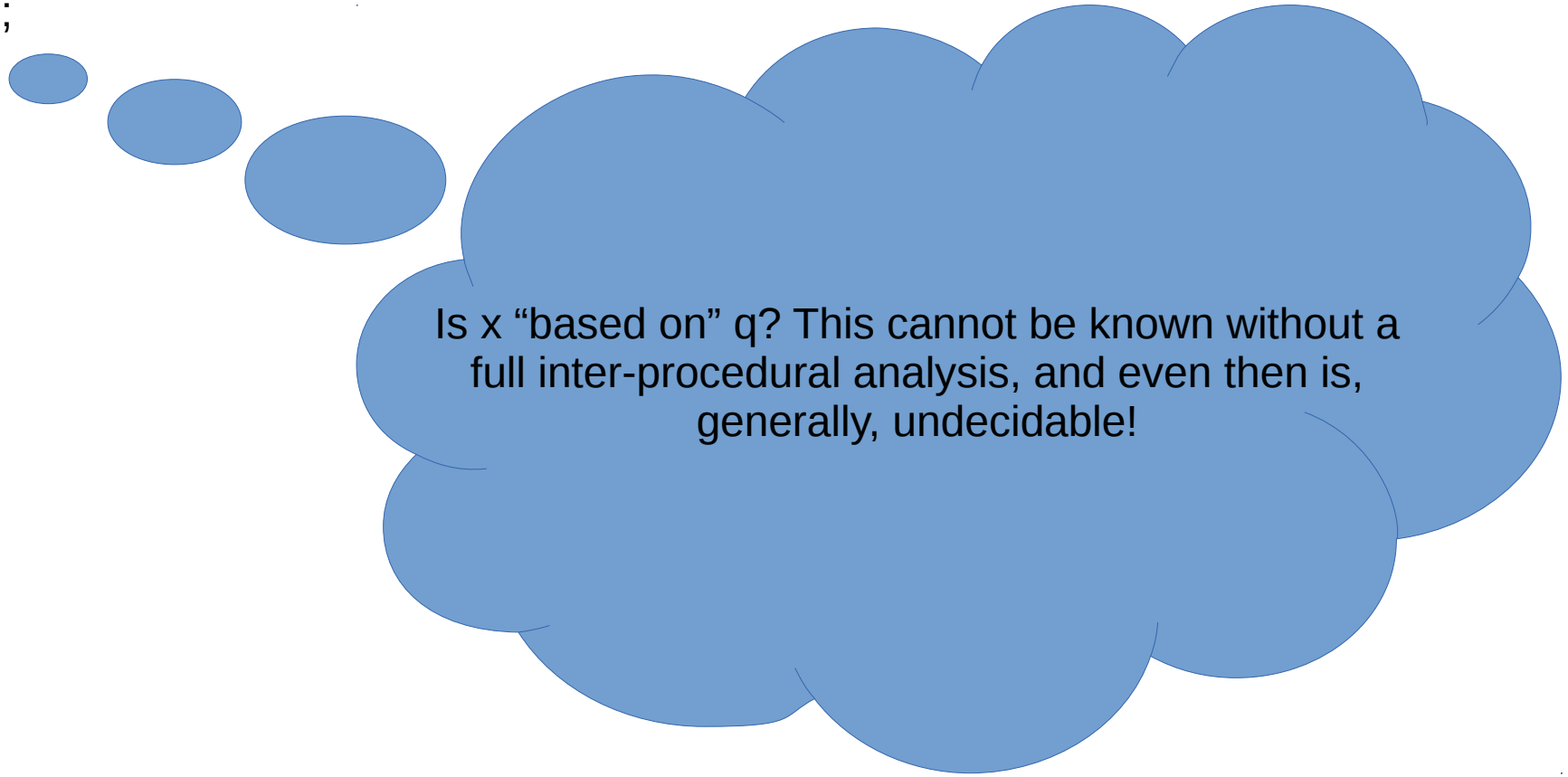
```
void foo(char *restrict p) {
 uintptr_t x = 0;
 uintptr_t y = (uintptr_t)p;
 for (int i = 0; i < 8*sizeof(char*); ++i, x <<= 1) {
 int bit = (y>>(8*sizeof(char*)-1)) & 1;
 x += bit ? 1 : 0;
 }
 char *q = (char*)x;
 // q is "based on" p.
}
```

This makes the compiler's job of figuring out which pointers in a function are allowed to alias with the restricted pointer difficult in mostly-non-useful ways.

“based on” and Capturing

Pointer capturing, the root of all evil...

```
int *bar(int * __restrict__ x);
void foo(int * __restrict__ q) {
 int *x = bar(q);
 *q = *x;
}
```



A common misconception...

Code from the LCALS 1.0.2 benchmark (https://codesign.llnl.gov/LCALS.php):

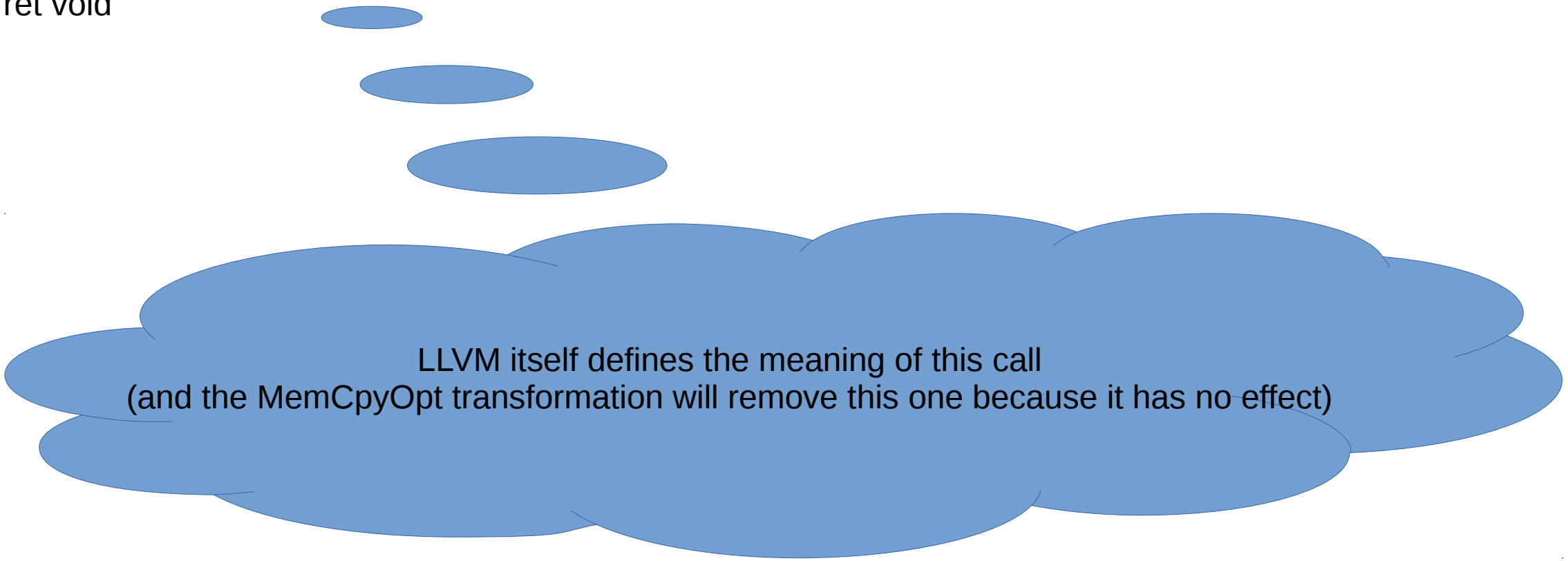
```
const Real_type& operator [] (Index_type i) const
{
   return( (const Real_type* __restrict__) dptr)[i];
}
```

According to the specification this doesn't mean anything (because the restrict is not on a declaration of a pointer variable).

Background: Intrinsics

Intrinsics are “internal” functions with semantics defined directly by LLVM. LLVM has both target-independent and target-specific intrinsics.

```
define void @test6(i8 *%P) {
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %P, i8* %P, i64 8, i32 4, i1 false)
  ret void
}
```



Background: Attributes

Properties of functions, function parameters and function return values that are part of the function definition and/or callsite itself.

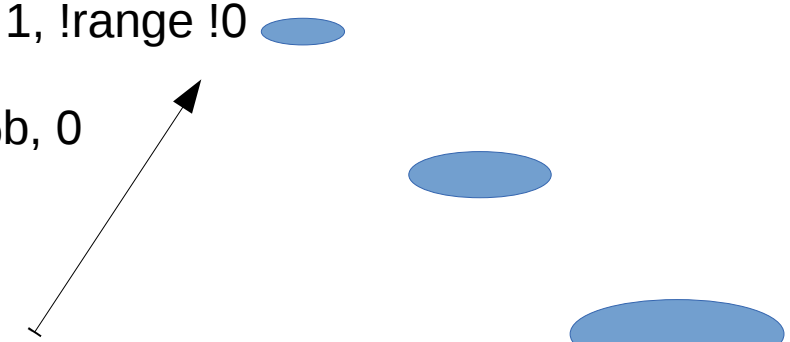
```
define i32 @foo(%struct.x* byval  %a) nounwind  {
  ret i32 undef
}
```

The object pointed to by %a is passed “by value” (a copy is made for use by the callee).
This is indicated by the “byval” attribute, which cannot generally be discarded.

Background: Metadata

Metadata represents optional information about an instruction (or module) that can be discarded without affecting correctness.

```
define zeroext i1 @_Z3fooPb(i8* nocapture %x) {
entry:
  %a = load i8* %x, align 1, !range !0
  %b = and i8 %a, 1
  %tobool = icmp ne i8 %b, 0
  ret i1 %tobool
}

!0 = !{i8 0, i8 2}
```

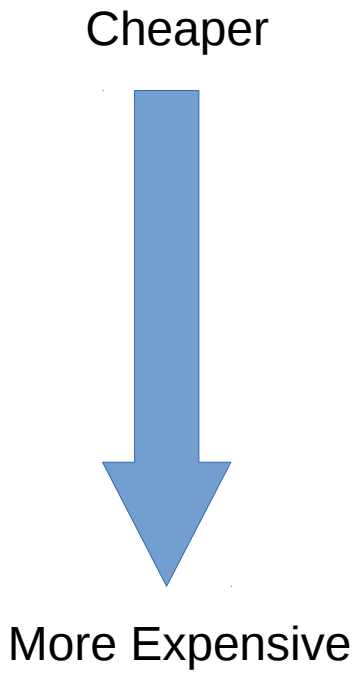


Range metadata provides the optimizer with additional information on a loaded value.
%a here is 0 or 1.

A note on expense

In what follows, we'll review these new

- **Attributes** (essentially free, use whenever you can)
- **Metadata** (comes at some cost: processing lots of metadata can slow down the optimizer)
- **Intrinsics** (intrinsics like @llvm.assume introduce extra instructions and value uses which, while providing potentially-valuable information, can also inhibit transformations: use judicially!)



We now also have “operand bundles”, which are like metadata for calls, but it is illegal to drop them…
call void @y() ["deopt"(i32 10), "unknown"(i8* null)]

(for example, used for implementing deoptimization)
These are essentially free, like attributes, but can block certain optimizations!

!alias.scope and !noalias Metadata

An alias scope is an (id, domain), and a domain is just an id. Both !alias.scope and !noalias take a list of scopes.

```
; Two scope domains:
!0 = distinct !{ !0}
!1 = distinct !{ !1}

; Some scopes in these domains:
!2 = distinct !{ !2,  !0}
!3 = distinct !{ !3,  !0}
!4 = distinct !{ !4,  !1}

; Some scope lists:
!5 =  !{ !4} ; A list containing only scope !4
!6 =  !{ !4,  !3,  !2}
!7 =  !{ !3}

; These two instructions don't alias:
%0 = load float* %c, align 4, !alias.scope !5
store float %0, float* %arrayidx.i, align 4, !noalias !5

; These two instructions also don't alias (for domain !1, the
; set of scopes in the !alias.scope equals that in the !noalias
; list):
%2 = load float* %c, align 4, !alias.scope !5
store float %2, float* %arrayidx.i2, align 4, !noalias !6

; These two instructions don't alias (for domain !0, the set of
; scopes in the !noalias list is not a superset of, or equal to,
; the scopes in the
; !alias.scope list):
%2 = load float* %c, align 4, !alias.scope !6
store float %0, float* %arrayidx.i, align 4, !noalias !7
```

From restrict to !alias.scope and !noalias

An example: Preserving noalias (restrict in C) when inlining:

```
void foo(double * restrict a, double * restrict b, double *c, int i) {
  double *x = i ? a : b;

  *c = *x;
}
```

The actual scheme also checks for capturing (because the pointer “based on” relationship can flow through captured variables)

*x gets:
!alias.scope: 'a', 'b'
(it might be derived from 'a' or 'b')

*c gets:
!noalias: 'a', 'b'
(definitely not derived from'a' or 'b')

*a would get:
!alias.scope: 'a'
!noalias: 'b'

The need for domains comes from making the scheme composable: When a function with noalias arguments, that has !alias.scope/!noalias metadata from an inlined callee, is itself inlined.

But, alas, there is a “bug”...

The current scheme cannot differentiate between the two situations:

```
void foo(float * restrict a, float * restrict b) {
  for (int i = 0; i < 1600; ++i)
    a[i] = b[i] + 1;
}
```

and:

```
void inner(float * restrict a, float * restrict b) {
  *a = *b + 1;
}
void foo(float * a, float * b) {
  for (int i = 0; i < 1600; ++i)
    inner(a+i, b+i);
}
```

To make the behavior truly correct (i.e. usable from the vectorizer and other loop transformations needing cross-iteration dependence information), we need to use a scheme that can contain dominance information...

A requirement:

Note that we need to represent:

```
void foo(T * restrict x, T * restrict y) {
  for (int i = 0; i < 1600; ++i)
    x[2*i+1] = y[2*i] + 1;
}
```

after being inlined from the call:

```
foo(q, q)
```

A @llvm.noalias intrinsic

```
declare <type>* @llvm.noalias.p<address space><type>(<type>* %ptr, metadata %scopes)
```

How does this work? This...

```
declare void @hey() #1

define void @hello(i8* noalias nocapture %a, i8* noalias nocapture readonly %c, i8* nocapture %b) #1 {
entry:
  %l = alloca i8, i32 512, align 1
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %a, i8* %b, i64 16, i32 16, i1 0)
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %b, i8* %c, i64 16, i32 16, i1 0)
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %a, i8* %c, i64 16, i32 16, i1 0)
  call void @hey()
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %l, i8* %c, i64 16, i32 16, i1 0)
  ret void
}

define void @foo(i8* nocapture %a, i8* nocapture readonly %c, i8* nocapture %b) #1 {
entry:
  tail call void @hello(i8* %a, i8* %c, i8* %b)
  ret void
}
```

A @llvm.noalias intrinsic

declare <type>* @llvm.noalias.p<address space><type>(<type>* %ptr, metadata %scopes)

Will become....

```
define void @foo(i8* nocapture %a, i8* nocapture readonly %c, i8* nocapture %b) #1 {
entry:
  %l.i = alloca i8, i32 512, align 1
  %0 = call i8* @llvm.noalias.p0i8(i8* %a, metadata !0)
  %1 = call i8* @llvm.noalias.p0i8(i8* %c, metadata !3)
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %0, i8* %b, i64 16, i32 16, i1 false) #1, !noalias !5
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %b, i8* %1, i64 16, i32 16, i1 false) #1, !noalias !5
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %0, i8* %1, i64 16, i32 16, i1 false) #1, !noalias !5
  call void @hey() #1, !noalias !5
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %l.i, i8* %1, i64 16, i32 16, i1 false) #1, !noalias !5
  ret void
}
```

(Patches under review; should hopefully be committed soon...)

```
!0 = !{!1}
!1 = distinct !{!1, !2, !"hello: %a"}
!2 = distinct !{!2, !"hello"}
!3 = !{!4}
!4 = distinct !{!4, !2, !"hello: %c"}
!5 = !{!1, !4}
```

Implementing this in LLVM...

Currently, LLVM only supports restrict on function arguments, although we have a way to preserve that information if the function is inlined. But we need more...

Here's some LLVM IR (intermediate representation):

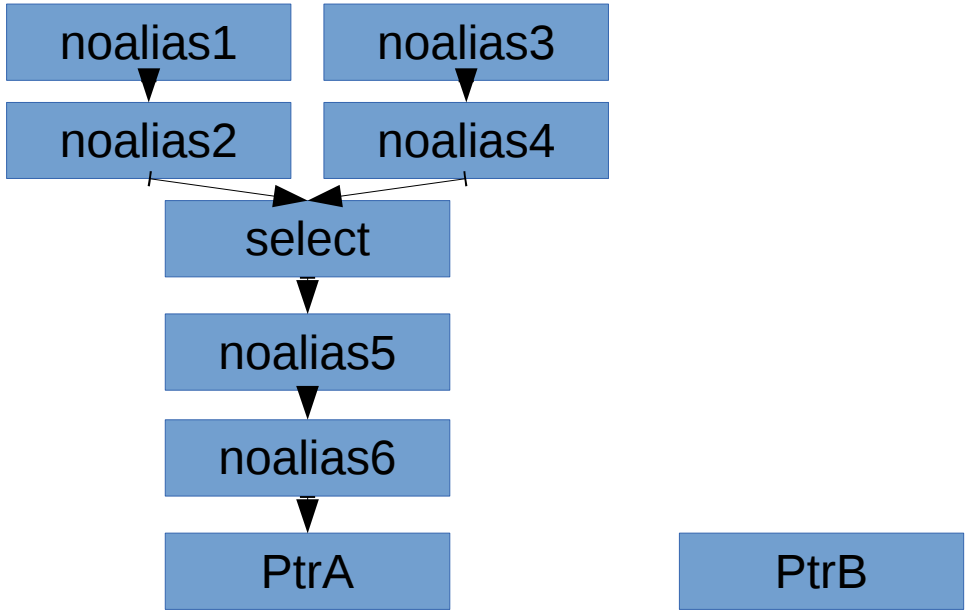
```
define void @foo(float* nocapture %a, float* nocapture readonly %c) #0 {
entry:
  %0 = call float* @llvm.noalias.p0f32(float* %a, metadata !0) #1
  %1 = load float, float* %c, align 4, !noalias !0
  %arrayidx.i = getelementptr inbounds float, float* %0, i64 5
  store float %1, float* %arrayidx.i, align 4, !noalias !0
  %2 = load float, float* %c, align 4
  %arrayidx = getelementptr inbounds float, float* %a, i64 7
  store float %2, float* %arrayidx, align 4
  ret void
}
```

Things derived from this pointer, %0, are in some alias set in a list specified by !0.

The memory being accessed here is definitely not in that alias set if the pointer is not based on %0.

Can't say anything about this one (this is how inlining works)

The AA algorithm for @llvm.noalias



- If PtrA is compatible with noalias6, and PtrB is also compatible, but does not derive from noalias6, then NoAlias.
- If PtrA is compatible with noalias5, and PtrB is also compatible, but does not derive from noalias5, then NoAlias.
- If PtrA is compatible with noalias2 and noalias4, and PtrB is also compatible, but does not derive from either, then NoAlias.
- If PtrA is compatible with noalias2 and noalias3, and PtrB is also compatible, but does not derive from either, then NoAlias.
- If PtrA is compatible with noalias1 and noalias4, and PtrB is also compatible, but does not derive from either, then NoAlias.
- If PtrA is compatible with noalias1 and noalias3, and PtrB is also compatible, but does not derive from either, then NoAlias.

The AA algorithm for @llvm.noalias

How does the algorithm work in practice...

When the aliasing analysis infrastructure is asked whether an access P and an access Q may alias, it will:

1) See if P and Q have !noalias metadata with any overlapping scopes; if not, give up.
2) For each P and Q, walk backward looking for the “underlying objects” for each pointer, collecting the list of scopes associated with @llvm.noalias intrinsics it finds along the way
3) If either P or Q is derived from a @llvm.noalias intrinsic with a scope listed in its !noalias metadata, and the other pointer is not, then they can't alias
4) When determining whether this derivation is possible, if the true “underlying object(s)” of the other pointer cannot be determined (meaning the pointer comes from a load, a function return value, etc.), and result of the @llvm.noalias call is captured before the base pointer value is generated, then we need to conservatively assume that they might alias.

The patches that implement all of this in LLVM are currently under code review...

What does this buy us?

```
$ cat /tmp/foo.c
void bar(float *);
void foo(int m, int n, float *a, float *b) {
  for (int i = 0; i < m; ++i) {
    float * restrict ra = a;
    float * restrict rb = b;

    for (int j = 0; j < n; ++j) {
      ra[j] = rb[j] + 1.0;
    }

    bar(ra);
  }
}
```

trunk vs. trunk with @llvm.noalias support.
We remove this memcheck!

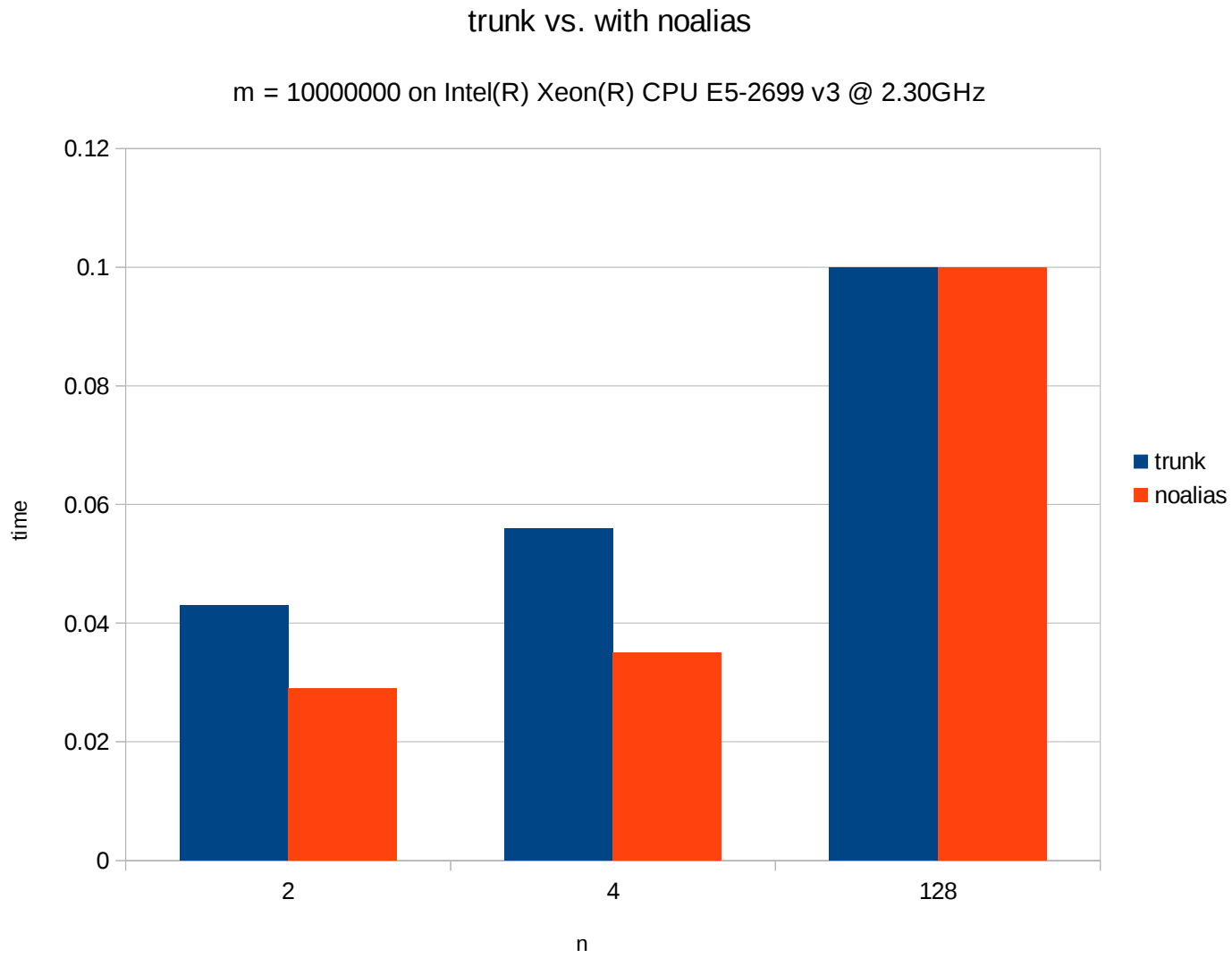
(we also don't hoist the memcheck, but that's another story...)

```
vector.body.preheader:                            ; preds = %min.iters.ch
  br label %vector.body

vector.body:                                      ; preds = %vector.body.
  %index = phi i64 [ %index.next, %vector.body ], [ 0, %vector.body.prehe
```

```
vector.memcheck:                                  ; preds = %min.iters.c
  %bound0 = icmp ugt float* %scevgep32, %a
  %bound1 = icmp ugt float* %scevgep, %b
  %memcheck.conflict = and i1 %bound0, %bound1
  br i1 %memcheck.conflict, label %for.body4.us.preheader, label %vector

vector.body.preheader:                            ; preds = %vector.memc
  br label %vector.body

vector.body:                                      ; preds = %vector.body
  %index = phi i64 [ %index.next, %vector.body ], [ 0, %vector.body.preh
```

What does this buy us?

```
$ cat /tmp/foo.c
void bar(float *);
void foo(int m, int n, float *a, float *b) {
  for (int i = 0; i < m; ++i) {
    float * restrict ra = a;
    float * restrict rb = b;

    for (int j = 0; j < n; ++j) {
      ra[j] = rb[j] + 1.0;
    }

    bar(ra);
  }
}
```



So does this fix the “bug”...

Can we differentiate between the two situations:

```
void foo(float * restrict a, float * restrict b) {
  for (int i = 0; i < 1600; ++i)
    a[i] = b[i] + 1;
}
```

and:

```
void inner(float * restrict a, float * restrict b) {
  *a = *b + 1;
}
void foo(float * a, float * b) {
  for (int i = 0; i < 1600; ++i)
    inner(a+i, b+i);
}
```

Yes, but…
We'll still need to generalize AA queries to support, in addition to:

- Fixed size
- Unknown size (i.e. the whole array)

To also have:

- Unknown size across all iterations of a loop L.

We can use the location of the noalias intrinsic, because it a place in the dominance tree, to answer this. If the relevant noalias is outside the loop, then the answer might be yes, otherwise, it is irrelevant.

Conclusions

- We can extend LLVM to support important usages of restrict-qualified pointers common in HPC
- This can provide important performance improvements
- We can use this to resolve a (unfortunately long standing) conceptual problem with the current noalias metadata and vectorization

- Special thanks to everyone from the LLVM community who has helped with this!
- For sponsoring this work, thanks to: ALCF, ANL, and DOE

➔ ALCF is supported by DOE/SC under contract DE-AC02-06CH11357

ALCF has open LLVM-related postdoc positions:
401877: http://bit.ly/2i7mVLl - 401878: http://bit.ly/2hShmQz

Some thoughts about what to do for C++

Why restrict does not work well in C++

Here's one way to get it...

```
void foo(T * __restrict a, T * __restrict b, ...) {
  for (int i = 0; i < N; ++i) {
    a[i] = b[i] + c[i] + d[i] + e[i];
  }
}
```

This isn't standard C++
(although is supported by almost all compilers)

And doesn't really give us what we want...

What we really want is to abstract the data structure; here's the simplest case which probably still won't give us what we want...

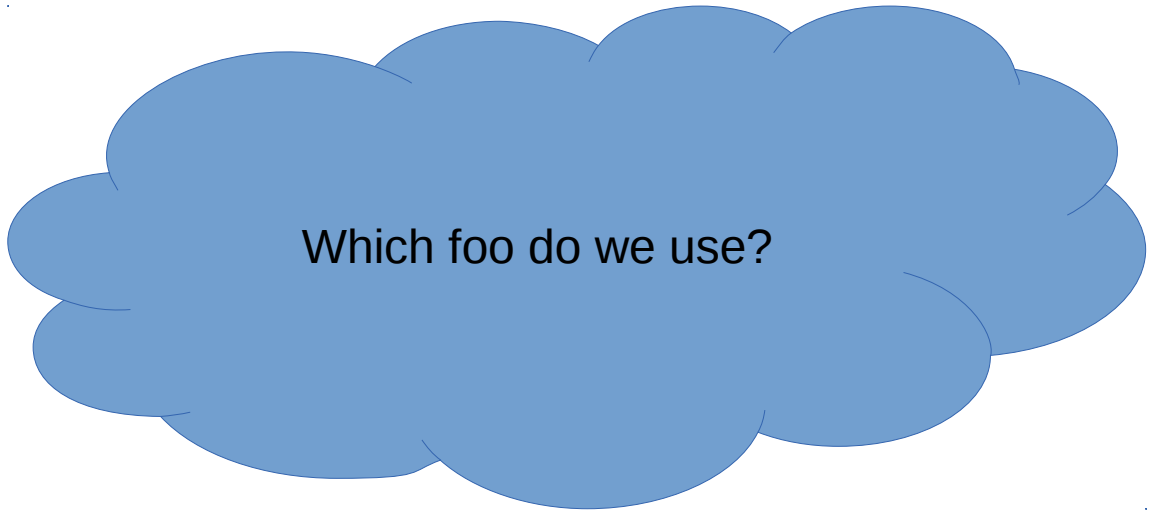
```
void foo(std::vector<T> &a, std::vector<T> &b, ...) {
  for (int i = 0; i < N; ++i) {
    a[i] = b[i] + c[i] + d[i] + e[i];
  }
}
```

Even proving that:
&a != &b
likely does not help here

Why restrict does not work well in C++

- Overloading?

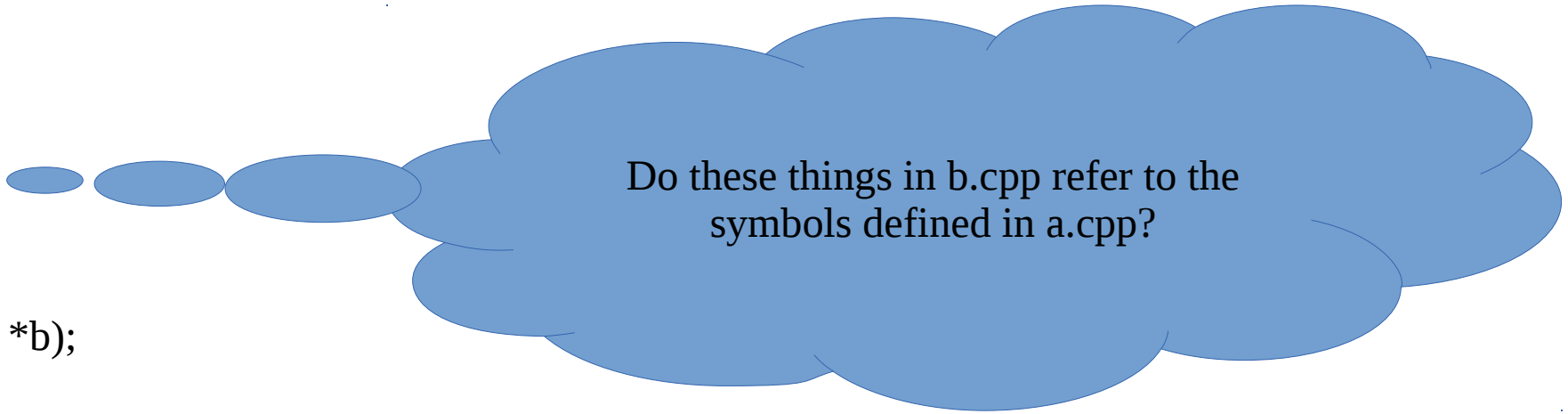
```
void foo(int * __restrict__ * a);
void foo(int * * b);

int main() {
 int a;
 int *b=&a;
 int *__restrict__ *c=&b;
 int * *d =&b;
 foo(c);
 foo(d);
}
```



Why restrict does not work well in C++

- Mangling ?

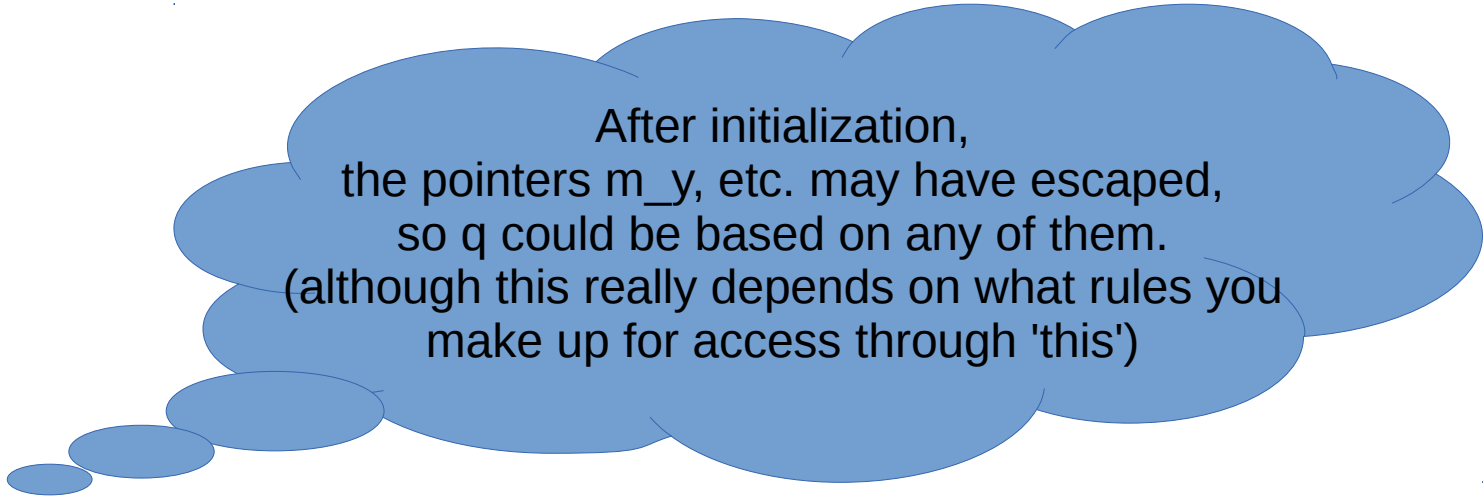
```
a.cpp:
int * __restrict__ p;
int * __restrict__ * q;

void foo(float * __restrict__ a, float * __restrict__ b) {
  ...
}

b.cpp:
extern int *p;
extern int ** q;

void foo(float *a, float *b);
```



Why restrict does not work well in C++

- Initialization and escape...

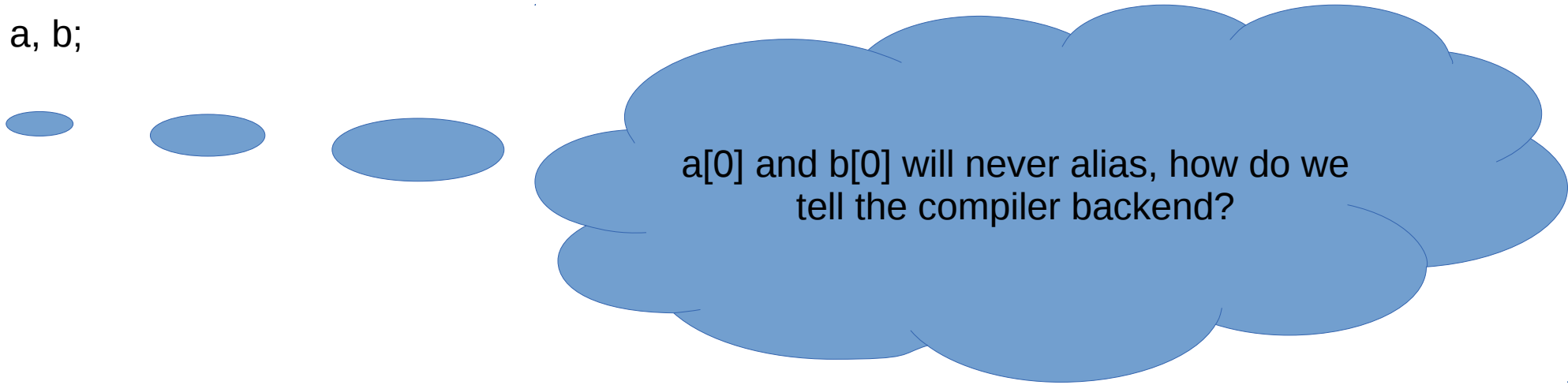
```
typedef double * __restrict__ d_p ;

class Functor
{
public:
 Functor(d_p x, d_p y, d_p z, d_p u)
  : m_x(x), m_y(y), m_z(z), m_u(u) {}

void operator(double *q, int k)
 {
 m_x[k] = m_u[k] + m_r*( m_z[k] + m_r*m_y[k] ) + ...;
 }

private:
d_p m_x;
const d_p m_y;
const d_p m_z;
const d_p m_u;
...
```



Why restrict does not work well in C++

- Aliasing and object identity...

```
std::vector<T> a, b;
...
a[0] = b[0];
```



Proposal (N4150)

Hal Finkel, Hubert Tong, Chandler Carruth, Clark Nelson, Daveed Vandevoorde,

Michael Wong

- Add a new alias-set attribute...

```
as-attribute-argument-clause:
  ( as-set-list )

as-set-list:
  as-set-list ; as-set-specifier
  as-set-specifier
as-set-specifier:
  as-set-name as-predicate_opt
  *
  ...
as-predicate:
  [ assignment-expression ]

as-set-name:
  qualified-id_opt
```

The attribute can be ignored!

Not part of the type system!

Promotes separation of concerns
(and, generally, good design principles)

Explicitly names the pointer sets
(and provides proper scoping)

Naturally handles nested pointer types.

Some examples...

- User-defined malloc-like function...

```
struct tagA {};
struct tagB {};

[[ alias_set() ]] void *my_malloc(size_t); // The return value is the address of an object that is disjoint from all others (just like malloc).

[[ alias_set(tagX) ]] T *x;
x = my_malloc(n);
v = *x; // x is restricted to the universe of tagX (and the compiler may infer that it is disjoint from any other pointer in tagX as well).

 [[ alias_set(tagA), alias_set(tagB) ]] int *x; // objects accessed using x are restricted to the union of the 'tagA'  universe and 'tagB' universe.
```

Some examples… (N4150)

- And it can be kind of like restrict (but better)...

```
struct tag {};

void foo([[ alias_set(a) ]] double *a, [[ alias_set(a) ]] double *b, [[ alias_set(tag) ]] double *c);

// The pointer values of a and b, and any derived from a or b might refer to the same object. The pointer value of c  can only refer to objects in the 'tag' universe (assuming a dereference). However, if foo is inlined into its caller, the alias_set guarantees conferred by the alias_set a still only apply to aliasing comparisons between dereferences that came from the inlined function, not between those and dereferences in the caller.
```

Some examples… (N4150)

- Containers (keeping alias-sets out of the interface!)

```
class container {
 protected:
 struct as {};
 [[ alias_set(as[this]) ]] int *start, *end;
 // These pointers, or those based on them, might refer to the same objects. No other pointers dereferenced by
functions that could access container::as will do so.
};

[[ alias_set(tagY) ]] T* bar([[ alias_set(tagX) ]] T * x) {
 return x; // this will cause undefined behavior if the return value is ever dereferenced in a context where it is
restricted to the tagY universe.
}
```

Some examples… (N4150)

- Containers (cont.)...

```
// assume that std::vector's internal pointers have an alias_set as in the preceding container example
using some  tag that is private to std::vector.
 void foo([[ alias_set(tagX) ]] T *x, std::vector<T> &V);

 [[ alias_set(tagX) ]] *x = &V[0];
 foo(x, V); // this is valid; the alias_set guarantees from std::vector's only apply to aliasing comparisons
where the  vector's alias_set tag is in scope and accessible. So if two of std::vector's operator [int]
were inlined, then the two dereferences contained therein could be dealiased, but nothing could be
said about the dereference within the  std::vector member function and the *x from within foo.
```

References and Nested Pointers (N4150)

- References

```
[[alias_set(tagX)]] double &x = ...;
```

- Nested Pointers

```
[[alias_set(tagX; ; tagZ; ...)]] double *******x;
```

Implicit, like malloc's return.

And all the rest are also tagZ
(note that type-based rules are still in effect)