SYNOPSYS®
Silicon to Software™

`ptr_provenance` and `@llvm.noalias`

The Tale of Full Restrict

Dobbelaere Jeroen

November 2021

Overview

- What is ‘`restrict`’?
- Where do we come from ?
- Thou shallt cherish thy declarations!
- Retrace your steps and learn your provenance.
- Get to know your provenance – *faster*.
- Copying blobs and concealed provenance.
- The future…

What is `restrict` ?

- A standardized(1) mechanism to indicate that pointers are ‘not aliasing’.
- Available in C since C99 (often also as a compiler extension for c++: `__restrict`).

- A simplified definition of ‘`restrict`’:

```
int *restrict p;
```

A promise that, inside the scope of ‘`p`’, all accesses to the set of objects that ‘`p`’ can point to, are done with a pointer based on ‘`p`’ (if at least one such object is also changed in this scope).

```
// dst will not alias with lhs, nor with rhs
// lhs may alias with rhs (only read)
void madd(int *restrict dst, int *restrict lhs, int *restrict rhs) {
    for (int i=0; i<512; ++i)
        dst[i] = dst[i] + lhs[i]*rhs[i];
}
// madd(a, b, b) is allowed;     madd(a, a, b) is not allowed
```

1) See ‘6.7.3.1 Formal definition of restrict’ (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2573.pdf)

Typical use cases

```
void add(int *restrict dst, int *restrict rhs) {
  // dst and rhs do not alias
  for (int i=0; i<512; ++i)
    dst[i] = dst[i]+rhs[i];
}

void foo(int *restrict a, int *restrict b, int *restrict c) {
  add(a, b); // a and b will not alias
  add(b, c); // b and c will not alias
             // a and c will not alias
}
```

Typical use cases

```
void add(int *        dst, int *        rhs) {
  // dst and rhs may    alias
  for (int i=0; i<512; ++i)
    dst[i] = dst[i]+rhs[i];
}

void foo(int *restrict a, int *restrict b, int *restrict c) {
  add(a, b); // a and b will not alias
  add(b, c); // b and c will not alias
             // a and c will not alias
}
```

Typical use cases

```
void add(int *restrict dst, int *restrict rhs) {
  // dst and rhs do not alias
  for (int i=0; i<512; ++i)
    dst[i] = dst[i]+rhs[i];
}

void foo(int *        a, int *        b, int *        c) {
  add(a, b); // a and b must not alias
  add(b, c); // b and c must not alias
             // a and c may       alias
}
```

Typical use cases

```
void foo(int *a, int *b, int *c) {
  {
    // a and b must not alias
    int *restrict dst = a;
    int *restrict rhs = b;
    for (int i=0; i<512; ++i)
      dst[i] = dst[i] + rhs[i];
  }

  {
    // b and c must not alias
    int *restrict dst = b;
    int *restrict rhs = c;
    for (int i=0; i<512; ++i)
      dst[i] = dst[i] + rhs[i];
  }
    // a and c may      alias
}
```

What is `restrict` ?

- A standardized[1] mechanism to indicate that pointers are ‘not aliasing’.
- Available in C since C99 (often also as a compiler extension for c++: `__restrict`).

- A simplified definition of ‘`restrict`’:

```
int *restrict p;
```

A promise that, inside the scope of ‘`p`’, all accesses to the set of objects that ‘`p`’ can point to, are done with a pointer based on ‘`p`’ (if at least one such object is also changed in this scope).

1) See ‘6.7.3.1 Formal definition of restrict’ (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2573.pdf)

What is `restrict` ?

- A standardized[1] mechanism to indicate that pointers are ‘not aliasing’.
- Available in C since C99 (often also as a compiler extension for c++: `__restrict`).

- A simplified definition of ‘`restrict`’:

```
struct restr_it { int *restrict p; } a;
```

A promise that, inside the scope of ‘a’, all accesses to the set of objects that ‘p’ can point to, are done with a pointer based on ‘p’ (if at least one such object is also changed in this scope).

1) See ‘6.7.3.1 Formal definition of restrict’ (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2573.pdf)

Typical use cases

```
struct restr_it {
    int *restrict p;
};

void add(struct restr_it dst, struct restr_it rhs) {
  // dst.p and rhs.p do not alias
  for (int i=0; i<512; ++i)
    dst.p[i] = dst.p[i]+rhs.p[i];
}

void foo(struct restr_it a, struct restr_it b, struct restr_it c) {
    add(a, b); // a.p and b.p will not alias
    add(b, c); // b.p and c.p will not alias
               // a.p and c.p will not alias
}
```

Where do we come from ?

- `restrict` only supported on function arguments

> **Hal Finkel -** *Thu Nov 13 16:44:54 PST 2014*
>
> Hi everyone,
>
> As many of you might know, LLVM now has scoped noalias metadata (http://llvm.org/docs/LangRef.html#noalias-and-alias-scope-metadata) -- it allows us to preserve noalias function argument attributes when inlining, in addition to allowing frontends to add otherwise non-deducible aliasing properties to memory accesses. This currently works well, but needs a change and an intrinsic, as I'll explain below.

- **2014** First discussion about improving support.(1)
- **2015** First set of patches to support local restrict and fix the inlining issue.(2)
- **2017, 2018** Getting more experiences with the local restrict patches…
  - Started thinking on how to fix the observed problems.
- **2018** Discussions at a round-table at the LLVM Dev Conf
- **2019** `RFC: Full 'restrict' support in LLVM` (3)

  `[PATCH 01/27] [noalias] LangRef:` (4)

(1) https://lists.llvm.org/pipermail/llvm-dev/2014-November/078755.html
(2) https://reviews.llvm.org/D9375
(3) https://lists.llvm.org/pipermail/llvm-dev/2019-October/135672.html
(4) https://reviews.llvm.org/D68484

Thou shalt cherish thy declarations!

restrict -> noalias

LLVM-11

```
void foo(int * restrict p0, int * restrict p1) {
    *p0=*p1;
}
```

```
define void @foo(i32* noalias %0, i32* noalias %1) {
  %3 = load i32, i32* %1, align 4, !tbaa !2
  store i32 %3, i32* %0, align 4, !tbaa !2
  ret void
}
```

noalias -> !alias.scope and !noalias

LLVM-11

```
void foo(int * restrict p0, int * restrict p1) {
    *p0=*p1;
}

void test_inline_foo(int *p0, int *p1) {
    foo(p0, p1);
}
```

```
define void @foo(i32* noalias %0, i32* noalias %1) {
  %3 = load i32, i32* %1, align 4, !tbaa !2
  store i32 %3, i32* %0, align 4, !tbaa !2
  ret void
}

define void @test_inline_foo(i32* %0, i32* %1) {
  %3 = load i32, i32* %1, align 4, !tbaa !2, !alias.scope !6, !noalias !9
  store i32 %3, i32* %0, align 4, !tbaa !2, !alias.scope !9, !noalias !6
  ret void
}
```

```
!6 = !{!7}
!7 = distinct !{!7, !8, !"foo: argument 1"}
!8 = distinct !{!8, !"foo"}
!9 = !{!10}
!10 = distinct !{!10, !8, !"foo: argument 0"}
```

Handled in `ScopedNoAliasAA.cpp`

noalias -> !alias.scope and !noalias

LLVM-12

```
void foo(int * restrict p0, int * restrict p1) {
    *p0=*p1;
}

void test_inline_foo(int *p0, int *p1) {
    foo(p0, p1);
}
```

```
define void @foo(i32* noalias %0, i32* noalias %1) {
  %3 = load i32, i32* %1, align 4, !tbaa !2
  store i32 %3, i32* %0, align 4, !tbaa !2
  ret void
}

define void @test_inline_foo(i32* %0, i32* %1) {
  tail call void @llvm.experimental.noalias.scope.decl(metadata !6)
  tail call void @llvm.experimental.noalias.scope.decl(metadata !9)
  %3 = load i32, i32* %1, align 4, !tbaa !2, !alias.scope !6, !noalias !9
  store i32 %3, i32* %0, align 4, !tbaa !2, !alias.scope !9, !noalias !6
  ret void
}
```

```
!6 = !{!7}
!7 = distinct !{!7, !8, !"foo: argument 1"}
!8 = distinct !{!8, !"foo"}
!9 = !{!10}
!10 = distinct !{!10, !8, !"foo: argument 0"}
```

`@llvm.experimental.noalias.scope.decl`

```
declare void @llvm.experimental.noalias.scope.decl(metadata !Scope)
```

- Identifies where a noalias scope was introduced in the control flow.

- Duplicating the intrinsic normally implies duplicating the associated scope.
  - Loop Unroll, Loop Rotate, …

- Notes:
  - This part was extracted from the full restrict patches.(1)
  - Also one of the goals of the `@llvm.noalias` intrinsic in the local restrict patches(2)

(1) https://reviews.llvm.org/D68484
(2) https://reviews.llvm.org/D9375

Retrace your steps and learn your provenance.

`@llvm.noalias` (full restrict patches[1])

```
i32* @llvm.noalias(i32*%p, i8*%p.decl, i32**%p.addr, i64 Id, metadata !Scope)
```

- Used when **reading** a restrict pointer value.

- Adds an implicit `!alias.scope` property.
- Simplifies the `!noalias` metadata on memory accesses.
  (represents visible restrict variables)

- Introduces a dominance dependency on **`@llvm.noalias.decl`**[2]

1) https://reviews.llvm.org/D68484
2) this is equivalent to `@llvm.experimental.noalias.scope.decl`

@llvm.noalias

(full restrict patches)

```
define void @foo(i32* noalias %rp0, i32* %p1, i32* %p2, i64 %i) {
  %rp0.addr = alloca i32*, align 8
  %p1.addr = alloca i32*, align 8
  %p2.addr = alloca i32*, align 8
  %i.addr = alloca i64, align 8
  %rp1 = alloca i32*, align 8
  %0 = call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** %rp0.addr, i64 0, metadata !3), !noalias !6
  store i32* %rp0, i32** %rp0.addr, align 8, !tbaa !8, !noalias !6
  store i32* %p1, i32** %p1.addr, align 8, !tbaa !8, !noalias !6
  store i32* %p2, i32** %p2.addr, align 8, !tbaa !8, !noalias !6
  store i64 %i, i64* %i.addr, align 8, !tbaa !12, !noalias !6
  %1 = bitcast i32** %rp1 to i8*
  call void @llvm.lifetime.start.p0i8(i64 8, i8* %1) #4, !noalias !6
  %2 = call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** %rp1, i64 0, metadata !14), !noalias !6
  %3 = load i32*, i32** %p1.addr, align 8, !tbaa !8, !noalias !6
  store i32* %3, i32** %rp1, align 8, !tbaa !8, !noalias !6
  %4 = load i32*, i32** %rp1, align 8, !tbaa !8, !noalias !6
  %5 = call i32* @llvm.noalias.p0i32.p0i8.p0p0i32.i64(i32* %4, i8* %2, i32** %rp1, i64 0, metadata !14), !tbaa !8, !noalias !6
  %6 = load i64, i64* %i.addr, align 8, !tbaa !12, !noalias !6
  %mul = mul nsw i64 3, %6
  %add = add nsw i64 %mul, 10

  %arrayidx = getelementptr inbounds i32, i32* %5, i64 %add
  store i32 42, i32* %arrayidx, align 4, !tbaa !12, !noalias !6
  %7 = load i32*, i32** %p2.addr, align 8, !tbaa !8, !noalias !6
  store i32 43, i32* %7, align 4, !tbaa !12, !noalias !6
  %8 = bitcast i32** %rp1 to i8*
  call void @llvm.lifetime.end.p0i8(i64 8, i8* %8) #4
  ret void
}
```

```
void foo(int *restrict rp0, int *p1, int *p2, long i) {
  int *restrict rp1 = p1;
  rp1[3*i+10] = 42;
  *p2 = 43;
}
```

```
!3 = !{!4}
!4 = distinct !{!4, !5, !"foo: rp0"}
!5 = distinct !{!5, !"foo"}
!6 = !{!4, !7}
!7 = distinct !{!7, !5, !"foo: rp1"}
!14 = !{!7}
```

@llvm.noalias (full restrict patches)

```
define void @foo(i32* noalias %rp0, i32* %p1, i32* %p2, i64 %i) {
  %rp0.addr = alloca i32*, align 8
  %p1.addr = alloca i32*, align 8
  %p2.addr = alloca i32*, align 8
  %i.addr = alloca i64, align 8
  %rp1 = alloca i32*, align 8
  %0 = call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** %rp0.addr, i64 0, metadata !3), !noalias !6
  store i32* %rp0, i32** %rp0.addr, align 8, !tbaa !8, !noalias !6
  store i32* %p1, i32** %p1.addr, align 8, !tbaa !8, !noalias !6
  store i32* %p2, i32** %p2.addr, align 8, !tbaa !8, !noalias !6
  store i64 %i, i64* %i.addr, align 4, !tbaa !12, !noalias !6
  %1 = bitcast i32** %rp1 to i8*
  call void @llvm.lifetime.start.p0i8(i64 8, i8* %1) #4, !noalias !6
  %2 = call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** %rp1, i64 0, metadata !14), !noalias !6
  %3 = load i32*, i32** %p1.addr, align 8, !tbaa !8, !noalias !6
  store i32* %3, i32** %rp1, align 8, !tbaa !8, !noalias !6
  %4 = load i32*, i32** %rp1, align 8, !tbaa !8, !noalias !6
  %5 = call i32* @llvm.noalias.p0i32.p0i8.p0p0i32.i64(i32* %4, i8* %2, i32** %rp1, i64 0, metadata !14), !tbaa !8, !noalias !6
  %6 = load i64, i64* %i.addr, align 8, !tbaa !12, !noalias !6
  %mul = mul nsw i64 3, %6
  %add = add nsw i64 %mul, 10

  %arrayidx = getelementptr inbounds i32, i32* %5, i64 %add
  store i32 42, i32* %arrayidx, align 4, !tbaa !12, !noalias !6
  %7 = load i32*, i32** %p2.addr, align 8, !tbaa !8, !noalias !6
  store i32 43, i32* %7, align 4, !tbaa !12, !noalias !6
  %8 = bitcast i32** %rp1 to i8*
  call void @llvm.lifetime.end.p0i8(i64 8, i8* %8) #4
  ret void
}
```

```
void foo(int *restrict rp0, int *p1, int *p2, long i) {
  int *restrict rp1 = p1;
  rp1[3*i+10] = 42;
  *p2 = 43;
}
```

```
!3 = !{!4}
!4 = distinct !{!4, !5, !"foo: rp0"}
!5 = distinct !{!5, !"foo"}
!6 = !{!4, !7}
!7 = distinct !{!7, !5, !"foo: rp1"}
!14 = !{!7}
```

`@llvm.noalias` – After SROA

(full restrict patches)

```
define void @foo(i32* noalias %rp0, i32* %p1, i32* %p2, i64 %i) {

  %0 = call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !3)

  %1 = call i32* @llvm.noalias.p0i32.p0i8.p0p0i32.i64(i32* %p1, i8* %0, i32** null, i64 0, metadata !3), !tbaa !6, !noalias !10

  %mul = mul nsw i64 3, %i
  %add = add nsw i64 %mul, 10

  %arrayidx = getelementptr inbounds i32, i32* %1, i64 %add
  store i32 42, i32* %arrayidx, align 4, !tbaa !12, !noalias !10

  store i32 43, i32* %p2, align 4, !tbaa !12, !noalias !10

  ret void
}
```

```
void foo(int *restrict rp0, int *p1, int *p2, long i) {
  int *restrict rp1 = p1;
  rp1[3*i+10] = 42;
  *p2 = 43;
}
```

```
!3 = !{!4}
!4 = distinct !{!4, !5, !"foo: rp1"}
!5 = distinct !{!5, !"foo"}

!10 = !{!11, !4}
!11 = distinct !{!11, !5, !"foo: rp0"}
```

Get to know your provenance…

Faster !

@llvm.noalias – After SROA

```
define void @foo(i32* noalias %rp0, i32* %p1, i32* %p2, i64 %i) {

  %0 = call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !3)

  %1 = call i32* @llvm.noalias.p0i32.p0i8.p0p0i32.i64(i32* %p1      , i8* %0, i32** null, i64 0, metadata !3), !tbaa !6, !noalias !10

  %mul  = mul nsw i64 3, %i
  %add = add nsw i64 %mul , 10

  %arrayidx = getelementptr inbounds i32, i32* %1, i64 %add
  store i32 42, i32* %arrayidx, align 4, !tbaa !12, !noalias !10

  store i32 43, i32* %p2, align 4, !tbaa !12, !noalias !10

  ret void
}
```

```
void foo(int *restrict rp0, int *p1, int *p2, long i) {
  int *restrict rp1 = p1;    // p1+2*i+5
  rp1[3*i+10] = 42;
  *p2 = 43;
}
```

```
!3 = !{!4}
!4 = distinct !{!4, !5, !"foo: rp1"}
!5 = distinct !{!5, !"foo"}

!10 = !{!11, !4}
!11 = distinct !{!11, !5, !"foo: rp0"}
```

`@llvm.noalias` – After SROA (2)

```
define void @foo(i32* noalias %rp0, i32* %p1, i32* %p2, i64 %i) {

  %0 = call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !3)

  %mul = mul nsw i64 2, %i

  %add.ptr = getelementptr inbounds i32, i32* %p1, i64 %mul

  %add.ptr1 = getelementptr inbounds i32, i32* %add.ptr, i64 5

  %1 = call i32* @llvm.noalias.p0i32.p0i8.p0p0i32.i64(i32* %add.ptr1, i8* %0, i32** null, i64 0, metadata !3), !tbaa !6, !noalias !10

  %mul2 = mul nsw i64 3, %i
  %add = add nsw i64 %mul2, 10

  %arrayidx = getelementptr inbounds i32, i32* %1, i64 %add
  store i32 42, i32* %arrayidx, align 4, !tbaa !12, !noalias !10

  store i32 43, i32* %p2, align 4, !tbaa !12, !noalias !10


  ret void
}
```

```
void foo(int *restrict rp0, int *p1, int *p2, long i) {
  int *restrict rp1 = p1+2*i+5; // EXTRA COMPUTATIONS
  rp1[3*i+10] = 42;
  *p2 = 43;
}
```

Too opaque – optimizations are blocked ☹
Too transparent – wrong code is produced ☹

```
!3 = !{!4}
!4 = distinct !{!4, !5, !"foo: rp1"}
!5 = distinct !{!5, !"foo"}

!10 = !{!11, !4}
!11 = distinct !{!11, !5, !"foo: rp0"}
```

`@llvm.noalias` – Computation vs Provenance

```
define void @foo(i32* noalias %rp0, i32* %p1, i32* %p2, i64 %i) {

  %0 = call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !3)

  %mul = mul nsw i64 2, %i

  %add.ptr = getelementptr inbounds i32, i32* %p1, i64 %mul

  %add.ptr1 = getelementptr inbounds i32, i32* %add.ptr, i64 5

  %1 = call i32* @llvm.noalias.p0i32.p0i8.p0p0i32.i64(i32* %add.ptr1, i8* %0, i32** null, i64 0, metadata !3), !tbaa !6, !noalias !10

  %mul2 = mul nsw i64 3, %i
  %add = add nsw i64 %mul2, 10

  %arrayidx = getelementptr inbounds i32, i32* %1, i64 %add
  store i32 42, i32* %arrayidx, align 4, !tbaa !12, !noalias !10

  store i32 43, i32* %p2, align 4, !tbaa !12, !noalias !10

  ret void
}
```

```
void foo(int *restrict rp0, int *p1, int *p2, long i) {
  int *restrict rp1 = p1+2*i+5; // EXTRA COMPUTATIONS
  rp1[3*i+10] = 42;
  *p2 = 43;
}
```

```
!3 = !{!4}
!4 = distinct !{!4, !5, !"foo: rp1"}
!5 = distinct !{!5, !"foo"}

!10 = !{!11, !4}
!11 = distinct !{!11, !5, !"foo: rp0"}
```

`@llvm.noalias` – After PropagateAndConvertNoAlias

```
define void @foo(i32* noalias %rp0, i32* %p1, i32* %p2, i64 %i) {

  %0 = call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !3)

  %mul = mul nsw i64 2, %i

  %add.ptr = getelementptr inbounds i32, i32* %q1, i64 %mul

  %add.ptr1 = getelementptr inbounds i32, i32* %add.ptr, i64 5

  %1 = call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32* %p1, i8* %0, i32** null, i32** undef, i64 0, metadata !3),
                                                                                                    !tbaa !6, !noalias !10
  %mul2 = mul nsw i64 3, %i
  %add = add nsw i64 %mul2, 10

  %arrayidx = getelementptr inbounds i32, i32* %add.ptr1, i64 %add
  store i32 42, i32* %arrayidx, ptr_provenance i32* %1, align 4, !tbaa !12, !noalias !10

  store i32 43, i32* %p2, align 4, !tbaa !12, !noalias !10


  ret void
}
```

```
void foo(int *restrict rp0, int *p1, int *p2, long i) {
  int *restrict rp1 = p1+2*i+5; // EXTRA COMPUTATIONS
  rp1[3*i+10] = 42;
  *p2 = 43;
}
```

```
!3 = !{!4}
!4 = distinct !{!4, !5, !"foo: rp1"}
!5 = distinct !{!5, !"foo"}

!10 = !{!11, !4}
!11 = distinct !{!11, !5, !"foo: rp0"}
```

`@llvm.noalias` – More optimizations

```
define void @foo(i32* noalias %rp0, i32* %p1, i32* %p2, i64 %i) {

  %0 = call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !3)

  %1 = call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32* %p1, i8* %0, i32** null, i32** undef, i64 0, metadata !3),
                                                                                                        !tbaa !6, !noalias !10
  %mul2 = mul nsw i64 5, %i
  %add = add nsw i64 %mul2, -15

  %arrayidx = getelementptr inbounds i32, i32* %p1, i64 %add
  store i32 42, i32* %arrayidx, ptr_provenance i32* %1, align 4, !tbaa !12, !noalias !10

  store i32 43, i32* %p2, align 4, !tbaa !12, !noalias !10

  ret void
}
```

```
void foo(int *restrict rp0, int *p1, int *p2, long i) {
  int *restrict rp1 = p1+2*i+5; // EXTRA COMPUTATIONS
  rp1[3*i+10] = 42;
  *p2 = 43;
}
```

```
!3 = !{!4}
!4 = distinct !{!4, !5, !"foo: rp1"}
!5 = distinct !{!5, !"foo"}

!10 = !{!11, !4}
!11 = distinct !{!11, !5, !"foo: rp0"}
```

Potential optimization, but currently not done at LLVM-IR ☹.
SelectionDAG is able to combine everything ☺.

ptr_provenance[1]

- Optional extra parameter for `load/store`:

```
store i32 42, i32* %p, ptr_provenance i32* %prov.p, !noalias !7

%L = load i32, i32* %p, ptr_provenance i32* %prov.p, !noalias !7
```

- Safe for most optimizations: change the pointer computation, not the provenance
- Safe clone: set to `UnknownProvenance` when available
  - optimization passes must be aware of it for best results
- Shorter lookup path for `ScopedNoAliasAA.cpp`, …

1) Extracted ptr_provenance infrastructure: https://reviews.llvm.org/D111159

`ptr_provenance`[1]

- Optional extra parameter for `load/store`
- '`@llvm.experimental.ptr.provenance`' [2] intrinsic
  - Combines computation with provenance, needed for pointer escapes

- In full restrict, the idea was that following the computation and the ptr_provenance would always end up on the same base object. But…

- Currently, a more generic version is proposed[1]:
  - This should allow us to support N2676[3]
  - It should also make it possible to support optimizing '`if (p == q) *q=1;`'[4] while keeping the correct provenance.

1) Extracted ptr_provenance infrastructure: https://reviews.llvm.org/D111159
2) In the full restrict patches, this is: `@llvm.noalias.arg.guard`
3) N2676: a provenance-aware memory object model for C
4) https://bugs.llvm.org/show_bug.cgi?id=44313

Copying blobs and concealed provenance.

Concealed provenance

```
int foo(int* restrict p, int *r, long i) {
  *p = 99;       // p visible; depends on p
  {
    int *restrict q = r;
    q[i] = 42; // p, q visible; depends on q, not on p
  }
  return *p;     // p visible; depends on p
}
```

```
define dso_local i32 @foo(i32* noalias nocapture %p, i32* nocapture %r, i64 %i) local_unnamed_addr #0 {
  %0 = tail call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !3)
  %1 = tail call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32* %p, i8* %0, i32** null, i32** undef, i64 0, metadata !3),
                                                                                !tbaa !6, !noalias !3

  store i32 99, i32* %p, ptr_provenance i32* %1, align 4, !tbaa !10, !noalias !3
  %2 = tail call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !12)
  %3 = tail call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32* %r, i8* %2, i32** null, i32** undef, i64 0, metadata !12),
                                                                                !tbaa !6, !noalias !14
  %arrayidx = getelementptr inbounds i32, i32* %r, i64 %i
  store i32 42, i32* %arrayidx, ptr_provenance i32* %3, align 4, !tbaa !10, !noalias !14
  ret i32 99
}
```

Concealed provenance

```
int foo(int* restrict p, int *r, long i) {
  *p = 99;      // p visible; depends on p
  {
    int *restrict q = p;
    q[i] = 42; // p, q visible; depends on q and p
  }
  return *p;    // p visible; depends on p
}
```

```
define dso_local i32 @foo(i32* noalias nocapture %p, i32* nocapture readnone %r, i64 %i) local_unnamed_addr #0 {
  %0 = tail call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !3)
  %1 = tail call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32** %p, i8* %0, i32** null, i32** undef, i64 0, metadata !3),
                                                                                !tbaa !6, !noalias !3
  store i32 99, i32* %p, ptr_provenance i32* %1, align 4, !tbaa !10, !noalias !3
  %2 = tail call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !12)
  %3 = tail call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32* %1, i8* %2, i32** null, i32** undef, i64 0, metadata !12),
                                                                                !tbaa !6, !noalias !14
  %arrayidx = getelementptr inbounds i32, i32* %p, i64 %i
  store i32 42, i32* %arrayidx, ptr_provenance i32* %3, align 4, !tbaa !10, !noalias !14
  %4 = load i32, i32* %p, ptr_provenance i32* %1, align 4, !tbaa !10, !noalias !3
  ret i32 %4
}
```

Concealed provenance

```
struct RP { int *restrict rp; };

int foo(struct RP p, long i) {
  *p.rp = 99;   // p visible; depends on p.rp
  {
    struct RP q = p; // struct copy
    q.rp[i] = 42; // p, q visible; depends on q.rp, p.rp
  }
  return *p.rp; // p visible; depends on p.rp
}
```

```
define dso_local i32 @foo(i32* nocapture %p.coerce, i64 %i) local_unnamed_addr #0 {
  %0 = tail call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !3)
  %1 = tail call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32** %p.coerce, i8* %0, i32** null, i32** undef, i64 0, metadata !3),
                                                                                !tbaa !6, !noalias !3
  store i32 99, i32* %p.coerce, ptr_provenance i32* %1, align 4, !tbaa !11, !noalias !3
  %2 = tail call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !13)
  %3 = tail call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32* %1, i8* %2, i32** null, i32** undef, i64 0, metadata !13),
                                                                                !tbaa !6, !noalias !15
  %arrayidx = getelementptr inbounds i32, i32* %p.coerce, i64 %i
  store i32 42, i32* %arrayidx, ptr_provenance i32* %3, align 4, !tbaa !11, !noalias !15
  %4 = load i32, i32* %p.coerce, ptr_provenance i32* %1, align 4, !tbaa !11, !noalias !3
  ret i32 %4
}
```

Concealed provenance

```
struct RP { int *restrict rp; };

int foo(struct RP p, long i) {
  *p.rp = 99;   // p visible; depends on p.rp
  {
    struct RP q = p; // struct copy
    q.rp[i] = 42; // p, q visible; depends on q.rp, p.rp
  }
  return *p.rp; // p visible; depends on p.rp
}
```

```
define dso_local i32 @foo(i32* %p.coerce, i64 %i) #0 {
  %p = alloca %struct.RP, align 8
  %i.addr = alloca i64, align 8
  %q = alloca %struct.RP, align 8

  ...

  %4 = call i8* @llvm.noalias.decl.p0i8.p0s_struct.RPs.i64(%struct.RP* %q, i64 0, metadata !17), !noalias !15
  %5 = call %struct.RP* @llvm.noalias.copy.guard.p0s_struct.RPs.p0i8(%struct.RP* %p, i8* %0, metadata !18, metadata !3)
  %6 = bitcast %struct.RP* %q to i8*
  %7 = bitcast %struct.RP* %5 to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 8 %6, i8* align 8 %7, i64 8, i1 false), !tbaa.struct !19, !noalias !15
  %rp1 = getelementptr inbounds %struct.RP, %struct.RP* %q, i32 0, i32 0
  %8 = load i32*, i32** %rp1, align 8, !tbaa !10, !noalias !15
  %9 = call i32* @llvm.noalias.p0i32.p0i8.p0p0i32.i64(i32* %8, i8* %4, i32** %rp1, i64 0, metadata !17), !tbaa !10, !noalias !15
  %10 = load i64, i64* %i.addr, align 8, !tbaa !6, !noalias !15
  %arrayidx = getelementptr inbounds i32, i32* %9, i64 %10
  store i32 42, i32* %arrayidx, align 4, !tbaa !13, !noalias !15

  ...
```

```
; Encoded offsets with restrict pointers
!18 = !{i64 8, i64 0, i64 8, i64 1}

!3 = !{!4}
!4 = distinct !{!4, !5, !"foo: p"}
!16 = distinct !{!16, !5, !"foo: q"}
!17 = !{!16}
```

`@llvm.noalias.copy.guard` (full restrict patches[1])

```
%struct.X* @llvm.noalias.copy.guard(%struct.X* %p, i8*%p.decl,
                                    metadata !Offsets, metadata !Scope)
```

- Used when copying aggregates (struct, array) that contain one or more restrict pointers.

- Provides an (encoded) list of offsets where those pointers are residing.
- Adds an implicit `!alias.scope` property to those pointers.

- Allows SROA to expand `memcpy`/aggregate copies in a correct way, introducing `@llvm.noalias` when necessary.

Discussions with a related/similar goal:

- https://reviews.llvm.org/D100717: [InstCombine] Transform memcpy to ptr load/stores if TBAA says so
- https://lists.llvm.org/pipermail/llvm-dev/2021-June/150892.html: [RFC] Introducing a byte type to LLVM
- https://lists.llvm.org/pipermail/llvm-dev/2021-October/153295.html: Demystifying the byte type

1) https://reviews.llvm.org/D68484

The Future…

The Future

- Full Restrict upstreaming:
  - Try it out ?
    - Convenience ‘single patch’: https://reviews.llvm.org/D69542
  - Want to help reviewing ?
    - `ptr_provenance`: https://reviews.llvm.org/D111159 (2 out of 13 accepted)
  - Questions, remarks, suggestions ?
    - Mailing list
    - LLVM AA Tech call (every four weeks on Tuesday)
- Other:
  - N4150: alias set proposal(1)

1) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4150.pdf

Related bugs

- `restrict` related:
  - ~~https://bugs.llvm.org/show_bug.cgi?id=39282~~ Loop unrolling incorrectly duplicates noalias metadata
  - ~~https://bugs.llvm.org/show_bug.cgi?id=27955~~ Miscompilation of program using 'restrict' due to overaggressive vectorization

  } Fixed in LLVM-12 (`@llvm….noalias.scope.decl`)

  - https://bugs.llvm.org/show_bug.cgi?id=32581 restrict pointers from structures passed by value ignored
  - https://bugs.llvm.org/show_bug.cgi?id=39240 clang/llvm looses restrictness, resulting in wrong code
  - https://bugs.llvm.org/show_bug.cgi?id=45863 __restrict on struct member has no effect
  - https://bugs.llvm.org/show_bug.cgi?id=47575 noalias/restrict on global pointers

  } Fixed with Full Restrict

  - https://bugs.llvm.org/show_bug.cgi?id=44373 Optimization with `restrict`: is `p == q ? p : q` "based" on `p`?

- `ptr_provenance` related:
  - https://bugs.llvm.org/show_bug.cgi?id=44313 Wrong optimizations for pointers: `if (q == p) use p` -> `if (q == p) use q`
  - https://bugs.llvm.org/show_bug.cgi?id=34548 InstCombine cannot blindly assume that inttoptr(ptrtoint x) -> x

That’s all !

A big **thank you** to:

Dirk, Bruno, Eric, Wouter, Gert, Mark, David, Hal, Johannes, Alina, Troy, David, Nikita, Roman, Jonas, … and many others

Backup slides

Conditional selection of restrict pointers

```
int foo(int c, int *restrict p, int *restrict q, int i) {
  int *pq = c ? p : q;

  *p =42;
  pq[i] = 43;
  return *p;
}

int test_a(int *p, int *q, int i) {
  return foo(0, p, q, i);
}

int test_b(int *p, int *q, int i) {
  return foo(1, p, q, i);
}
```

clang-13

```
int foo(int c, int *restrict p, int *restrict q, int i) {
  int *pq = c ? p : q;

  *p =42;
  pq[i] = 43;
  return *p;
}

int test_a(int *p, int *q, int i) {
  return foo(0, p, q, i);
}

int test_b(int *p, int *q, int i) {
  return foo(1, p, q, i);
}
```

```
define i32 @test_a(i32* %0, i32* %1, i32 %2) {
  tail call void @llvm.experimental.noalias.scope.decl(metadata !7)
  store i32 42, i32* %0, align 4, !tbaa !3, !alias.scope !10, !noalias !7
  %4 = sext i32 %2 to i64
  %5 = getelementptr inbounds i32, i32* %1, i64 %4
  store i32 43, i32* %5, align 4, !tbaa !3, !alias.scope !12
  %6 = load i32, i32* %0, align 4, !tbaa !3, !alias.scope !10, !noalias !7
  ret i32 %6
}

define i32 @test_b(i32* %0, i32* %1, i32 %2) {
  tail call void @llvm.experimental.noalias.scope.decl(metadata !13)
  store i32 42, i32* %0, align 4, !tbaa !3, !alias.scope !16, !noalias !13
  %4 = sext i32 %2 to i64
  %5 = getelementptr inbounds i32, i32* %0, i64 %4
  store i32 43, i32* %5, align 4, !tbaa !3, !alias.scope !18
  %6 = load i32, i32* %0, align 4, !tbaa !3, !alias.scope !16, !noalias !13
  ret i32 %6
}
```

```
!7 = !{!8}
!8 = distinct !{!8, !9, !"foo: argument 1"}
!9 = distinct !{!9, !"foo"}
!10 = !{!11}
!11 = distinct !{!11, !9, !"foo: argument 0"}
!12 = !{!11, !8}

!13 = !{!14}
!14 = distinct !{!14, !15, !"foo: argument 1"}
!15 = distinct !{!15, !"foo"}
!16 = !{!17}
!17 = distinct !{!17, !15, !"foo: argument 0"}
!18 = !{!17, !14}
```

Full Restrict

```c
int foo(int c, int *restrict p, int *restrict q, int i) {
  int *pq = c ? p : q;

  *p =42;
  pq[i] = 43;
  return *p;
}

int test_a(int *p, int *q, int i) {
  return foo(0, p, q, i);
}

int test_b(int *p, int *q, int i) {
  return foo(1, p, q, i);
}
```

```
; using -fno-noalias-arguments to avoid double declarations of the restrict variables
; (once for the local argument, once after inlining a 'noalias' argument)


define i32 @test_a(i32* %0, i32* %1, i32 %2) {
  %4 = tail call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !13) #3
  %5 = tail call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !16) #3
  %6 = tail call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32* %0, i8* %4, i32** null, i32** undef, i64 0, metadata !13) #3
  %7 = tail call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32* %1, i8* %5, i32** null, i32** undef, i64 0, metadata !16) #3
  store i32 42, i32* %0, ptr_provenance i32* %6, align 4, !tbaa !8, !noalias !18
  %8 = sext i32 %2 to i64
  %9 = getelementptr inbounds i32, i32* %1, i64 %8
  store i32 43, i32* %9, ptr_provenance i32* %7, align 4, !tbaa !8, !noalias !18
  ret i32 42
}


define i32 @test_b(i32* %0, i32* %1, i32 %2) {
  %4 = tail call i8* @llvm.noalias.decl.p0i8.p0p0i32.i64(i32** null, i64 0, metadata !19) #3
  %5 = tail call i32* @llvm.provenance.noalias.p0i32.p0i8.p0p0i32.p0p0i32.i64(i32* %0, i8* %4, i32** null, i32** undef, i64 0, metadata !19) #3
  store i32 42, i32* %0, ptr_provenance i32* %5, align 4, !tbaa !8, !noalias !22
  %6 = sext i32 %2 to i64
  %7 = getelementptr inbounds i32, i32* %0, i64 %6
  store i32 43, i32* %7, ptr_provenance i32* %5, align 4, !tbaa !8, !noalias !22
  %8 = load i32, i32* %0, ptr_provenance i32* %5, align 4, !tbaa !8, !noalias !22
  ret i32 %8
}
```