Opaque Pointers Are Coming

Nikita Popov @ LLVM CGO 2022

About Me

- Sr. Software Engineer on Platform Tools team at Red Hat
- I maintain the LLVM Compile-Time Tracker

One PointerType to Rule Them All

```
i8*
i32*
[0 x i32]*                ptr
i8* (i8*)*
%ty*
```


Red Hat

… apart from address spaces

```
i8*
i32*
```

```
ptr
```

```
i8 addrspace(1)*
i32 addrspace(1)*
```

```
ptr addrspace(1)
```

Why?

Pointer element types carry no semantics

```
define void @test(i32* %p)
```

Pointer element types carry no semantics

```
define void @test(i32* %p)
```

⇒ Does not imply that %p is 4-byte aligned

```
define void @test(i32* aligned 4 %p)
```

Pointer element types carry no semantics

```
define void @test(i32* %p)
```

⇒ Does not imply that %p is 4-byte aligned

```
define void @test(i32* aligned 4 %p)
```

⇒ Does not imply that %p is 4-byte dereferenceable

```
define void @test(i32* dereferenceable(4) %p)
```

Pointer element types carry no semantics

```
define void @test(i32* %p)
```

⇒ Does not imply that %p is 4-byte aligned

```
define void @test(i32* aligned 4 %p)
```

⇒ Does not imply that %p is 4-byte dereferenceable

```
define void @test(i32* dereferenceable(4) %p)
```

⇒ Does not imply any aliasing semantics (TBAA metadata does)

Pointer element types carry no semantics

```
define void @test(i32* %p)
```

⇒ Does not imply that %p is 4-byte aligned

```
define void @test(i32* aligned 4 %p)
```

⇒ Does not imply that %p is 4-byte dereferenceable

```
define void @test(i32* dereferenceable(4) %p)
```

⇒ Does not imply any aliasing semantics (TBAA metadata does)

⇒ Does not imply it will be accessed as `i32`!

Pointers can be arbitrarily bitcasted

```
define i64 @test(double* byval(double) %p) {
  %p.p0i8 = bitcast double* %p to i8**
  store i8* null, i8** %p.p0i8
  %p.i64 = bitcast double* %p to i64*
  %x = load i64, i64** %p.i64
  ret i64 %x
}
```

Only types at certain uses matter

```
define i64 @test(double* byval(double) %p) {
  %p.p0i8 = bitcast double* %p to i8**
  store i8* null, i8** %p.p0i8
  %p.i64 = bitcast double* %p to i64*
  %x = load i64, i64** %p.i64
  ret i64 %x
}
```

Only types at certain uses matter

```
define i64 @test(ptr byval(double) %p) {

  store ptr 0.0, ptr %p

  %x = load i64, ptr %p
  ret i64 %x
}
```

Why?

- Memory usage: Don’t need to store bitcasts
- Compile-time: Don’t need to skip bitcasts in optimizations

Compile-Time Improvements (CTMark)

NewPM-O3:

Benchmark	Old	New
kimwitu++	51006M	49058M (-3.82%)
sqlite3	48144M	47289M (-1.78%)
consumer-typeset	47326M	43628M (-7.82%)
Bullet	116728M	114131M (-2.23%)
tramp3d-v4	111031M	105986M (-4.54%)
mafft	45658M	44875M (-1.71%)
ClamAV	70951M	71269M (+0.45%)
lencod	83910M	83417M (-0.59%)
SPASS	57310M	56069M (-2.16%)
7zip	169700M	166307M (-2.00%)
geomean	72445M	70529M (-2.65%)

Disclaimer: There may be differences in optimization behavior.

Link to data

Compile-Time Improvements (rustc)

Primary benchmarks

Benchmark & Profile	Scenario	% Change
html5ever opt	full	-6.65%
tokio-webpush-simple opt	full	-5.67%
syn opt	full	-5.52%
piston-image opt	full	-5.38%
clap-rs opt	full	-5.27%
style-servo opt	full	-5.07%
inflate opt	full	-4.82%
ripgrep opt	full	-4.73%
regex opt	full	-4.50%
cargo opt	full	-4.47%
hyper-2 opt	full	-4.34%
webrender-wrench opt	full	-3.81%
cranelift-codegen opt	full	-3.65%
inflate check	full	3.34%
encoding opt	full	-3.07%
futures opt	full	-2.82%
webrender opt	full	-2.44%
regex debug	full	-1.41%

Disclaimer: There may be differences in optimization behavior.

Link to data

Max-RSS Improvements (rustc)

Primary benchmarks

Benchmark & Profile	Scenario	% Change
html5ever opt	full	-10.23%
cargo opt	full	-5.56%
tokio-webpush-simple opt	full	-4.46%
clap-rs opt	full	-3.86%
webrender-wrench opt	full	-3.85%
style-servo opt	full	-3.12%
ripgrep opt	full	-2.98%
piston-image opt	full	-2.67%
cranelift-codegen opt	full	-2.51%
webrender debug	full	-2.45%
regex opt	full	-2.40%
piston-image debug	full	2.14%
hyper-2 opt	full	-2.04%
unicode_normalization debug	full	1.86%
clap-rs debug	full	-1.83%
webrender opt	full	-1.35%
encoding debug	full	1.33%
unicode_normalization doc	full	1.27%

Disclaimer: There may be differences in optimization behavior.

Link to data

Why?

- Memory usage: Don’t need to store bitcasts
- Compile-time: Don’t need to skip bitcasts in optimizations
- Performance:
  - Optimizations **should** ignore pointer bitcasts

Why?

- Memory usage: Don’t need to store bitcasts
- Compile-time: Don’t need to skip bitcasts in optimizations
- Performance:
  - Optimizations **should** ignore pointer bitcasts
  - …and many do (e.g. cost models says they’re free)

Why?

- Memory usage: Don’t need to store bitcasts
- Compile-time: Don’t need to skip bitcasts in optimizations
- Performance:
    - Optimizations **should** ignore pointer bitcasts
    - …and many do (e.g. cost models says they’re free)
    - …but many don’t (e.g. limited instruction/use walks)

Why?

- Memory usage: Don’t need to store bitcasts
- Compile-time: Don’t need to skip bitcasts in optimizations
- Performance:
  - Bitcasts can’t affect optimization if they don’t exist

Equivalence modulo pointer type

```
define i32* @test(i8** %p) {
  store i8* null, i8** %p
  %p.i32 = bitcast i8** %p to i32**
  %v = load i32*, i32** %p.i32
  ret i32* %v
}
```

EarlyCSE can’t optimize this!
(But full GVN can.)

Equivalence modulo pointer type

```
define ptr @test(ptr %p) {
  store ptr null, ptr %p
  %v = load ptr, ptr %p
  ret ptr %v
}
```

EarlyCSE **can** optimize this!

Equivalence modulo pointer type

```
define ptr @test(ptr %p) {
  store ptr null, ptr %p
  %v = load ptr, ptr %p
  ret ptr %v
}

; RUN: opt -S -early-cse < %s

define ptr @test(ptr %p) {
  store ptr null, ptr %p
  ret ptr null
}
```

Why?

- Memory usage: Don’t need to store bitcasts
- Compile-time: Don’t need to skip bitcasts in optimizations
- Performance:
  - Bitcasts can’t affect optimization if they don’t exist
  - Pointer element type difference cannot prevent CSE / forwarding / etc.

Offset-based reasoning

```
define internal i32 @add({ i32, i32 }* %p) {
  %p0 = getelementptr { i32, i32 }, { i32, i32 }* %p, i64 0, i32 0
  %v0 = load i32, i32* %p0
  %p1 = getelementptr { i32, i32 }, { i32, i32 }* %p, i64 0, i32 1
  %v1 = load i32, i32* %p1
  %add = add i32 %v0, %v1
  ret i32 %add
}

define i32 @caller({ i32, i32 }* %p) {
  %res = call i32 @add({ i32, i32 }* %p)
  ret i32 %res
}
```

Offset-based reasoning

```
; RUN: opt -S -argpromotion < %s

define internal i32 @add(i32 %p.0.val, i32 %p.4.val) {
  %add = add i32 %p.0.val, %p.4.val
  ret i32 %add
}

define i32 @caller({ i32, i32 }* %p) {
  %1 = getelementptr { i32, i32 }, { i32, i32 }* %p, i64 0, i32 0
  %p.val = load i32, i32* %1, align 4
  %2 = getelementptr { i32, i32 }, { i32, i32 }* %p, i64 0, i32 1
  %p.val1 = load i32, i32* %2, align 4
  %res = call i32 @add(i32 %p.val, i32 %p.val1)
  ret i32 %res
}
```

Offset-based reasoning

```
; RUN: opt -S -argpromotion < %s

define internal i32 @add(i32 %p.0.val, i32 %p.4.val) {
  %add = add i32 %p.0.val, %p.4.val
  ret i32 %add
}

define i32 @caller({ i32, i32 }* %p) {
  %1 = getelementptr { i32, i32 }, { i32, i32 }* %p, i64 0, i32 0
  %p.val = load i32, i32* %1, align 4
  %2 = getelementptr { i32, i32 }, { i32, i32 }* %p, i64 0, i32 1
  %p.val1 = load i32, i32* %2, align 4
  %res = call i32 @add(i32 %p.val, i32 %p.val1)
  ret i32 %res
}
```

Used to be based on GEP indices

Red Hat

Getelementptr index ambiguity

```
; Equivalent despite different indices:
getelementptr { [1 x i32], i32 }, ptr %p, i64 0
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0, i64 0
```

Getelementptr index ambiguity

```
; Equivalent despite different indices:
getelementptr { [1 x i32], i32 }, ptr %p, i64 0
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0, i64 0

; Equivalent despite different indices:
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0, i64 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 1, i32 0, i64 -1
```

Getelementptr index ambiguity

```
; Equivalent despite different indices:
getelementptr { [1 x i32], i32 }, ptr %p, i64 0
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0, i64 0

; Equivalent despite different indices:
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0, i64 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 1, i32 0, i64 -1
```

⇒ Requires careful restriction to ensure uniqueness
⇒ Can't support bitcasts

Getelementptr index ambiguity

```
; Equivalent despite different indices:
getelementptr { [1 x i32], i32 }, ptr %p, i64 0
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0, i64 0

; Equivalent despite different indices:
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0, i64 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 1, i32 0, i64 -1
```

⇒ Requires careful restriction to ensure uniqueness
⇒ Can't support bitcasts

⇒ Very hard to ensure correctness with opaque pointers

Offset-based reasoning

```
define internal i32 @add({ i32, i32 }* %p) {
  %p0 = getelementptr { i32, i32 }, { i32, i32 }* %p, i64 0, i32 0
  %v0 = load i32, i32* %p0  ← Load of i32 at offset 0
  %p1 = getelementptr { i32, i32 }, { i32, i32 }* %p, i64 0, i32 1
  %v1 = load i32, i32* %p1  ← Load of i32 at offset 4
  %add = add i32 %v0, %v1
  ret i32 %add
}

define i32 @caller({ i32, i32 }* %p) {
  %res = call i32 @add({ i32, i32 }* %p)
  ret i32 %res
}
```

Offset-based reasoning

```
define internal i32 @add({ i32, i32 }* %p) {
  %p0 = getelementptr { i32, i32 }, { i32, i32 }* %p, i64 0, i32 0
  %v0 = load i32, i32* %p0  ← Load of i32 at offset 0
  %p1 = getelementptr { i32, i32 }, { i32, i32 }* %p, i64 0, i32 1
  %v1 = load i32, i32* %p1  ← Load of i32 at offset 4
  %add = add i32 %v0, %v1
  ret i32 %add
}

define i32 @caller({ i32, i32 }* %p) {
  %res = call i32 @add({ i32, i32 }* %p)
  ret i32 %res
}
```

Derive “struct type” from access pattern, rather than IR type information

Why?

- Memory usage: Don’t need to store bitcasts
- Compile-time: Don’t need to skip bitcasts in optimizations
- Performance:
  - Bitcasts can’t affect optimization if they don’t exist
  - Pointer element type difference cannot prevent CSE / forwarding / etc.
  - Opaque pointers require/encourage generic offset-based reasoning

Why?

- Memory usage: Don’t need to store bitcasts
- Compile-time: Don’t need to skip bitcasts in optimizations
- Performance: …
- Implementation simplification:
  - Don’t need to insert bitcasts all over the place

Red Hat

Type-System recursion

```
%ty = type { %ty* }
```

⇒ This requires struct types to be mutable

Type-System recursion

```
%ty = type { %ty* }
```

⇒ This requires struct types to be mutable

```
%ty = type { ptr }
```

⇒ All types can be immutable

Why?

- Memory usage: Don’t need to store bitcasts
- Compile-time: Don’t need to skip bitcasts in optimizations
- Performance: …
- Implementation simplification:
  - Don’t need to insert bitcasts all over the place
  - Removes recursion from the type system

Why?

- Memory usage: Don’t need to store bitcasts
- Compile-time: Don’t need to skip bitcasts in optimizations
- Performance: …
- Implementation simplification:
  - Don’t need to insert bitcasts all over the place
  - Removes recursion from the type system
  - Enables follow-up IR changes to remove more types

How?

IR changes

Add explicit type where semantically relevant.

```
load i32* %p
load i32, i32* %p

getelementptr i32* %p, i64 1
getelementptr i32, i32* %p, i64 1

define void @test(i32* byval %p)
define void @test(i32* byval(i32) %p)
```

`Type::getPointerElementType()`

~~Type::getPointerElementType()~~

Code changes

Use value types:

```
Load->getPointerOperandType()->getPointerElementType()
⇒ Load->getType()

Store->getPointerOperandType()->getPointerElementType()
⇒ Store->getValueOperand()->getType()

Global->getType()->getPointerElementType()
⇒ Global->getValueType()

Call->getType()->getPointerElementType()
⇒ Call->getFunctionType()
```

Migration helpers

```
assert(Load->getType() ==
       Load->getPointerOperandType()->getPointerElementType());
⇒
assert(cast<PointerType>(Load->getPointerOperandType())
       ->isOpaqueOrPointeeTypeEquals(Load->getType()));
```

Code changes

`PointerType::get(ElemTy, AS)` still works in opaque pointer mode!

The element type is simply ignored.

Code changes

`PointerType::get(ElemTy, AS)` still works in opaque pointer mode!
The element type is simply ignored.

⇒ As long as `getPointerElementType()` is not called,
code usually “just works” in opaque pointer mode.

Pointer equality does not imply access type equality

```
define ptr @test(ptr %p) {
  store i32 0, ptr %p
  %v = load i64, ptr %p
  ret ptr %v
}
```

Pointer equality does not imply access type equality

```
define ptr @test(ptr %p) {
  store i32 0, ptr %p
  %v = load i64, ptr %p
  ret ptr %v
}
```

Need to explicitly check that load type == store type.
Not implied by same pointer operand anymore!

Red Hat

Frontends

Need to track pointer element types in their own structures now – can't rely on LLVM PointerType!

Frontends

Need to track pointer element types in their own structures now – can’t rely on LLVM PointerType!

Clang: `Address`, `LValue`, `RValue` store pointer element type now.

Opaque pointer mode

Automatically enabled if you use `ptr` in IR or bitcode.

Opaque pointer mode

Automatically enabled if you use `ptr` in IR or bitcode.

Manually enabled with -opaque-pointers.

```
define i32* @test(i32* %p)
  ret i32* %p
}
```

Opaque pointer mode

Automatically enabled if you use `ptr` in IR or bitcode.

Manually enabled with -opaque-pointers.

```
define i32* @test(i32* %p)
  ret i32* %p
}

; RUN: opt -S -opaque-pointers < %s
define ptr @test(ptr %p)
  ret ptr %p
}
```


Red Hat

Opaque pointer mode

Automatically enabled if you use `ptr` in IR or bitcode.

Manually enabled with -opaque-pointers.

Upgrading (very old) bitcode to opaque pointers is supported!

Migration

- Proposed by David Blaikie in 2015

Migration

- Proposed by David Blaikie in 2015
- Massive migration scope, with many hundreds of direct and indirect pointer element type uses across the code base.
  - Some trivial to remove.
  - Some require IR changes or full transform rewrites.

Migration

- Proposed by David Blaikie in 2015
- Massive migration scope, with many hundreds of direct and indirect pointer element type uses across the code base.
  - Some trivial to remove.
  - Some require IR changes or full transform rewrites.
- Opaque pointer type `ptr` only introduced in 2021 by Arthur Eubanks

Migration

- Proposed by David Blaikie in 2015
- Massive migration scope, with many hundreds of direct and indirect pointer element type uses across the code base.
  - Some trivial to remove.
  - Some require IR changes or full transform rewrites.
- Opaque pointer type `ptr` only introduced in 2021 by Arthur Eubanks
- Now (end of March 2022): All pointer element type accesses in LLVM and Clang eradicated.

Migration

- Proposed by David Blaikie in 2015
- Massive migration scope, with many hundreds of direct and indirect pointer element type uses across the code base.
  - Some trivial to remove.
  - Some require IR changes or full transform rewrites.
- Opaque pointer type `ptr` only introduced in 2021 by Arthur Eubanks
- Now (end of March 2022): All pointer element type accesses in LLVM and Clang eradicated.
- Next step: Enable opaque pointers by default.
  - Caveat: Requires updating ~7k tests.

Migration

- Proposed by David Blaikie in 2015
- Massive migration scope, with many hundreds of direct and indirect pointer element type uses across the code base.
  - Some trivial to remove.
  - Some require IR changes or full transform rewrites.
- Opaque pointer type `ptr` only introduced in 2021 by Arthur Eubanks
- Now (end of March 2022): All pointer element type accesses in LLVM and Clang eradicated.
- Next step: Enable opaque pointers by default.
  - Caveat: Requires updating ~7k tests.
- Typed pointers expected to be removed after LLVM 15 branch.

Future: Type-less GEP

All of these are equivalent:

```
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0, i64 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 1, i32 0, i64 -1
```

Future: Type-less GEP

All of these are equivalent:

```
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0, i64 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 1, i32 0, i64 -1
getelementptr i32, ptr %p, i64 1
getelementptr i8, ptr %p, i64 4
```

Future: Type-less GEP

All of these are equivalent:

```
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 0, i32 0, i64 1
getelementptr { [1 x i32], i32 }, ptr %p, i64 1, i32 0, i64 -1
getelementptr i32, ptr %p, i64 1
getelementptr i8, ptr %p, i64 4
```

Offset-based algorithms will realize these are equivalent, but…

Future: Type-less GEP

Nothing in the `-O3` pipeline realizes that %p1 and %p2 can be CSEd:

```
define void @test(ptr %p) {
  %p1 = getelementptr i8, ptr %p, i64 4
  %p2 = getelementptr i32, ptr %p, i64 1
  call void @use(ptr %p1, ptr %p2)
  ret void
}
```

Future: Type-less GEP

Offset-based GEP makes these trivially equivalent:

```
define void @test(ptr %p) {
  %p1 = getelementptr ptr %p, i64 4
  %p2 = getelementptr ptr %p, i64 4
  call void @use(ptr %p1, ptr %p2)
  ret void
}
```

Future: Type-less GEP

How far should we go?

```
%p.idx = getelementptr ptr %p, 4 * i64 %idx

; or

%off = shl i64 %idx, 2
%p.idx = getelementptr ptr %p, i64 %off
```

The End

- Docs: https://llvm.org/docs/OpaquePointers.html
- Reach me at:
  - npopov@redhat.com
  - https://twitter.com/nikita_ppv