Opaque Pointer Types

To a world without pointer to pointer bitcasts

Motivation

Proximal Motivation

```
222748: Canonicalize “store (cast V), P” -> “store V, (cast P)”
220138: Canonicalize “cast (load P)” -> “load (cast P)”
```

- LLVM IR looks like C - typed pointers
- Typed pointers don’t actually provide guarantees

```
void f(char *c) {
  *(int*)c = 3;
}
```

Noise/misleading information can only hurt optimizations

General Motivation: Simplification

- 150 stripPointerCasts
- uncountable other code dealing with this complexity
  - eg: r251291, Loop Vectorizer - skipping “bitcast” before GEP

General Motivation: Performance

- Compile time: reduce the size of IR by removing pointer bitcast instructions
- Runtime: fix optimizations that weren’t ignoring/looking through bitcasts when they should

Strategy

- Small increments as much as possible
- Split out each piece
- Try to do as much before the actual core change as possible
- Try to do as much after as possible (leave in code that will become dead/unneeded, remove it separately)

Easier to review, easier to roll back, easier to migrate - all the usual good stuff

Textual IR and Bitcode Migration

Start With What We Know

‘load’ and ‘getelementptr’ instructions will need an explicit type operand once that information is not conveyed by the pointer operand.

```
load i32* %x
  -> load i32, ptr %x
getelementptr <4 x i32*> %x, i32 3
  -> getelementptr i32, <4 x ptr> %x, i32 3
```

Updating Test Cases

- 1000s of test cases
- Updated for each change to the IR syntax
- With the power of regular expressions...

```
pat = re.compile(r"((?:=|:|^)\s*load (?:atomic )?(?:volatile
)?(.*?))(| addrspace\(\d+\) *)\*($| *(?:%
|@|null|undef|blockaddress|getelementptr|addrspacecast|bitca
st|inttoptr|\[\[[a-zA-Z]|\{\{).*$")
```

- Even changing CHECK lines (in Clang, Polly, and LLVM)

Migration Plan

After transforming a few obvious cases, developed a plan:

Attemp to serialize and deserialize every LLVM IR regression test case (read textual IR, write bitcode, read bitcode, write textual IR) without using pointer element types.

Whitelist obvious exceptions that should remain during the migration:

- IR Verifier
- IR/Bitcode parser error checks/diagnostics

Diving Deeper

Corresponding bitcode changes to match the IR

Further bitcode changes:
  For example, ‘store’ would only store the pointer type, not the value type. Instead reconstituting the value type from the pointer type. That won’t work.

Call/Invoke

Usually call/invoke is fine (doesn't rely on element type):

```
  call void %x()
```

Unless it's variadic, in which case the return type was replaced with the function type (as a function /pointer/ type):

```
  call void (...)* %x()
```

Call/Invoke

What about a function returning a function pointer type?

```
call void ()* ()* %x() ; void (*x)()
```

If we just encode the return type as a function type instead, no ambiguity and just be explicit for vararg:

```
call void (...) %x() ; void x(...);
call void ()* %x() ; void (*x)();
call void %x() ; void x();
```

The Rest

- getelementptr operator
  involved a bit more work on the migration regexes
- global alias
  surprised to find this needed a value type

Wrinkle 1: ByVal

---

byval:

```
%struct.foo = type { [1024 x i32] }
declare void @x(%struct.foo byval align 8)
```

Option 1:

```
declare void @x(ptr byval 4096 align 8)
```

Option 2:

```
declare void @x(ptr byval %struct.foo align 8)
```

Wrinkle 2: GlobalVariable/Function Forward References

Deserialization (and parsing) of references to GlobalVariables and Functions

If the reference came before the declaration, a stub entity was created of the appropriate type (variable or function, depending on the element type).

```
  store i32 %j, i32* @foo, align 4
…
  @foo = global i32 0, align 4
```

What About Global Aliases?

```
@foo = alias i32, i32* @b_val
```

Type information was insufficient - a GlobalVariable was created for the alias

ReplaceAllUsesWith to replace the variable with the alias

Can we do the same thing between functions and variables?

Types Are Hard

Aliases are easy, the value types match.

1) wait until the pointers actually change - then the type will match: they'll all be opaque pointer & everything will look like the alias case(ish)
2) use bitcasts…
   We could create the same type of GlobalVariable for every forward reference, and bitcast it to the desired type, then RAUW that global later on (with more bitcasts).

API Migrations

Flushing out PointerType::getElementType calls within IR/bitcode roundtripping also meant API changes:

- Explicit type member of llvm::GlobalValue
- Backwards compatible (keeping existing implicit type parameters where still in use)
- Sub-optimal attempt at migrating GEP factories by forcing an explicit parameter but allowing null for old callers

Next Steps

Optimizations

- Simplifying optimizations that already handle these cases where necessary. (where it’s not /necessary/ to change them in lock-step, let the code become dead with the removal of element types and clean it up after)

- Fix optimizations that have been leaning on this information - harder, but possibly rewarding in terms of enabling optimization opportunities hidden behind this crutch.

Frontends

Frontends easily rely on element types during IR generation for convenience.

Fundamentally fixable - just pass around or recompute the element type wherever needed.
May involve a lot of plumbing.

How To Migrate Your Code

- Add an assertion/unreachable in PointerType::getElementType
- Run your tests/code (preferably in frontend-only mode, or just your one optimization rather than a full pipeline)
- Everywhere the assertion fires, push the API requirement up:

Migration API Examples

```
-  auto *PTy = cast<PointerType>(TypeMap.get(SGA->getType()));
-  return GlobalAlias::create(PTy->getElementType(), PTy->getAddressSpace(),
+  auto *Ty = TypeMap.get(SGA->getValueType());
+  return GlobalAlias::create(Ty, SGA->getType()->getPointerAddressSpace(),
                              SGA->getLinkage(), SGA->getName(), &DstM);
```

```
-  Type *Ty = Globals[j]->getType()->getElementType();
+  Type *Ty = Globals[j]->getValueType();
```

What To Expect

- Some churn: Byval is hopefully the only remaining invasive/mass IR change.
- Less mechanical/more risky changes to tease out element type dependence in optimizations
- Tedious, but likely safe/uninteresting changes to Clang
- Introduction of an opaque PointerType (possibly a swap, keeping the typed version around for IR/bitcode parsing)
- A moderately large change to flip the switch
- A bunch of cleanup patches to remove now-dead code