Paths towards unifying LLVM and MLIR

Nicolai Hähnle
Part I

Where do you see yourself the LLVM project 10 Years From Now?
“The LLVM project”

Communication

~

Intermediate Representation
What is “LLVM IR”?

Dialect

Substrate
Dialect

Instruction set
Types
Semantics

E.g.: LangRef.rst, mlir/Dialect.td

Substrate

Set of C++ classes to represent and manipulate code in one or more dialects

E.g.: llvm::Instruction, LLVM::Value, LLVM::BasicBlock, …

N:M relationship
Substrates Matter

- Code artefacts (transforms, analyses, helpers, …) are written against a specific substrate
- Can represent LLVM in MLIR, but can’t run InstCombine or loop transforms on it
Substrates Matter

%2535 = call i32 @llvm.amdgon.ubfe.i32(i32 %.4.vec.extract1487, i32 16, i32 16)
%2536 = select i1 false, i32 %.4.vec.extract1487, i32 %2535
%2537 = select i1 false, i32 0, i32 %2536
%2538 = uitofp i32 %2537 to float
%2539 = call i32 @llvm.amdgon.ubfe.i32(i32 %.4.vec.extract1487, i32 0, i32 16)
%2540 = select i1 false, i32 %.4.vec.extract1487, i32 %2539
%2541 = select i1 false, i32 0, i32 %2540
%2542 = uitofp i32 %2541 to float
%2543 = insertelement <4 x float> undef, float %2530, i64 0
%2544 = insertelement <4 x float> %2543, float %2534, i64 1
%2545 = insertelement <4 x float> %2544, float %2538, i64 2
%2546 = insertelement <4 x float> %2545, float %2542, i64 3
%scale.i152 = fnul reassoc @nnn nsz arcp contract afn <4 X float> %2546, <float 0x3f80000000000000, float 0x3f80000000000000, float 0x3f80000000000000, float 0x3f80000000000000>, <float 0x3f800000000000000, float 0x3f800000000000000, float 0x3f800000000000000, float 0x3f800000000000000>
%2547 = call i8 @lgc.descriptor.table.addr(i32 1, i32 1, i32 0, i32 2, i32 1, i32 0) #4
%2548 = getelementptr i18, i18 %2547, i64 0
%2549 = bitcast i8 @lgc.descriptor.table.addr(i32 1, i32 1, i32 0, i32 2, i32 1, i32 0) to i18
%2550 = bitcast i8 @lgc.descriptor.table.addr(i32 1, i32 1, i32 0, i32 2, i32 1, i32 0) to i18
%2551 = getelementptr i18, i18 %2550, i64 0
%2552 = bitcast i8 @lgc.descriptor.table.addr(i32 1, i32 1, i32 0, i32 2, i32 1, i32 0) to i18
%2553 = load <4 x i32>, <4 x i32> %2552, align 16
%2554 = load <8 x i32>, <8 x i32> %2552, align 32
%2555 = shufflevector <4 x float> %25152, <4 x float> poison, <2 x i32> <i32 0, i32 1>
%2556 = fmul reassoc nmn nsz arcp contract afn <2 x float> %2492, %2555
%2557 = shufflevector <4 x float> %25152, <4 x float> poison, <2 x i32> <i32 2, i32 3>
%2558 = fadd reassoc nmn nsz arcp contract afn <2 x float> %2556, %2557
%2559 = extractelement <2 x float> %2558, i64 0
%2560 = extractelement <2 x float> %2558, i64 1
%2561 = call reassoc nmn nsz arcp contract afn <4 x float> @llvm.amdgon.image.sample.l.2d.v4f32.f32(i32 15, float %2559, float %2560, float 0.000000e+00)
%2562 = extractelement <4 x float> %2561, i64 0
%2563 = fmul reassoc nmn nsz arcp contract afn float %2562, %2562
Benefits of a Unified Substrate

- Interoperability of code
- Eliminate a hard pass-ordering boundary

- Accessibility for developers
  - Shared knowledge and shared “language” within the LLVM project
  - Reduced friction for “full (compiler) stack developers”
Part II
The Unification Landscape
and the Delta Matrix
Thought Experiment: Port LLVM Code to the MLIR Substrate

Port passes from LLVM to MLIR substrate, from left to right.
Reality Check: The Optimization Pass Pipeline
Thought Experiment: Port LLVM Code to the MLIR Substrate

- Port passes from LLVM to MLIR substrate, from left to right

- Many substrate round-trips (compile-time cost!) or code duplication (maintenance cost!)
Thought Experiment: Make Code Generic over the Substrate

• Generic code already exists. For example:

```cpp
template <typename NodeT, bool IsPostDom>
class DominatorTreeBase {
}
```

Expand use of this technique. For example:

```cpp
template <typename BlockT, typename InstructionT>
void hoistAllInstructionsInto(BlockT *DomBlock, InstructionT *InsertPt,
                              BlockT *BB);
```

Eventually:

```cpp
namespace llvm {
using BasicBlock = mlir::Block;
using Instruction = mlir::Operation;
// ...
}
```
void llvm::hoistAllInstructionsInto(BasicBlock *DomBlock, Instruction *InsertPt, BasicBlock *BB) {
    // (...)
    for (BasicBlock::iterator II = BB->begin(), IE = BB->end(); II != IE;) {
        Instruction *I = &II;
        I->dropUndefImplyingAttrsAndUnknownMetadata();
        if (I->isUsedByMetadata())
            dropDebugUsers(*I);
        if (I->isDebugOrPseudoInst()) {
            // Remove DbgInfo and pseudo probe Intrinsics.
            II = I->eraseFromParent();
            continue;
        }
        I->setDebugLoc(InsertPt->getDebugLoc());
        ++II;
    }
    DomBlock->getInstList().splice(InsertPt->getIterator(), BB->getInstList(),
        BB->begin(),
        BB->getTerminator()->getIterator());
}
template <typename BlockT, typename InstructionT>
void llvm::hoistAllInstructionsInto(BlockT *DomBlock, InstructionT *InsertPt,
                                    BlockT *BB) {
    // (...)}

    for (typename BlockT::iterator II = BB->begin(), IE = BB->end(); II != IE;) {
        InstructionT *I = &*II;
        I->dropUndefinedImpliedAndUnknownMetadata();
        if (I->isUsedByMetadata())
            dropDebugUsers(*I);
        if (I->isDebugOrPseudoInst()) {
            // Remove DbgInfo and pseudo probe Intrinsics.
            II = I->eraseFromParent();
            continue;
        }
        I->setDebugLoc(InsertPt->getDebugLoc());
        ++II;
    }
    DomBlock->getInstList() splice(InsertPt->getIterator(), BB->getInstList(),
                                    BB->begin(),
                                    BB->getTerminator()->getIterator());
Tooling for Templates is Weak

- Error messages
- Compile times
- Language servers

```c
inline void llvm::Instruction::setDebugLoc(llvm::DebugLoc Loc)
```

Set the debug location information for this instruction.

```c
I->setDebugLoc(InsertPt->getDebugLoc());
```

VS.

```c
<unknown> InstructionT::setDebugLoc
I->setDebugLoc(InsertPt->getDebugLoc());
++IT;
```
C++ concepts

```cpp
template <typename BlockT>
concept Block = requires (BlockT BB) {
    ...
};

template <concepts::Block BlockT, concepts::Instruction InstructionT>
void hoistAllInstructionsInto(BlockT *DomBlock, InstructionT *InsertPt,
                               BlockT *BB);
```

- Example: “Block” concept
  - CFG predecessors and successors
  - Instruction list and terminators
  - Basic block arguments
  - ...
- Not as good as type classes or traits; and requires C++20
### The Delta Matrix

<table>
<thead>
<tr>
<th></th>
<th>Clang</th>
<th>Flang</th>
<th>SimplifyCFG</th>
<th>InstCombine</th>
<th>Attributor</th>
<th>Coroutines</th>
<th>AddressSanitizer</th>
<th>Alias analysis</th>
<th>Target codegen</th>
<th>Assembly printer &amp; parser</th>
<th>Bitcode writer &amp; reader</th>
<th>IR linker</th>
<th>Ildb</th>
<th>JIT infrastructure</th>
<th>llvm</th>
<th>reduce</th>
</tr>
</thead>
</table>

...
## The Delta Matrix

<table>
<thead>
<tr>
<th></th>
<th>Constants</th>
<th>Multiple defined values</th>
<th>Phi nodes vs. B3 args</th>
<th>Builder interface</th>
<th>Regions</th>
<th>Attributes</th>
<th>Debug info</th>
<th>Metadata</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clang, Flang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InstCombine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attributor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coroutines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AddressSanitizer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alias analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target codegen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assembly printer &amp; parser</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitcode writer &amp; reader</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IR linker</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lldb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>llvm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Part III
Column Refactorings

"Columns with Excavator" by DALL-E 2
Example 1: Dialects in LLVM

1. First-class support for extended instructions and types, with near-MLIR-compatible definitions
2. Cleanup of “the LLVM dialect” into constituent dialects
3. Extensibility for frontends, including external frontends
First-class support for extended instructions and types

- Intrinsics are second-class citizens in textual IR

```
x = add i32 y, z
```

vs.

```
x = call i32 @llvm.smin.i32(i32 y, i32 z)
```

- Intrinsics are second-class citizens in C++

```
if (auto *Store = dyn_cast<StoreInst>(I)) {
    use(Store->getPointerOperand());
}
```

vs.

```
if (auto *Intr = dyn_cast<IntrinsicInst>(I)) {
    if (Intr->getIntrinsicID() == Intrinsic::prefetch) {
        use(Intr->getArgOperand(0));
    }
}
```

- Intrinsics are compile-time and space inefficient

```
store volatile i32 x, ptr p
```

vs.

```
call void @llvm.memset.p0.i64(ptr %p, i8 %y, i64 4, i1 true)
```
Cleanup of “the LLVM dialect”
Extensibility

Today:

```c
call void @llvm.downstream.custom()
```

```c
auto *Intr = cast<IntrinsicInst>(I);
assert(Intr->getIntrinsicID() == Intrinsic::not_intrinsic);
```
Example 2: Constant values vs. materialization

- Value → User → Constant
  - ConstantData
    - ConstantInt, ConstantFP, ...
    - ConstantDataSequential
  - ConstantAggregate
    - ConstantStruct, ConstantArray, ConstantVector
  - ConstantExpr
  - GlobalValue
    - GlobalIndirectSymbol
    - GlobalObject
      - Function
      - GlobalVariable
  - BlockAddress
  - DSOLocalEquivalent
  - UndefValue
  - PoisonValue

\[
%1 = \text{add i32 } 0, 42
\]

vs.

\[
%1 = \text{arith.constant } 42 : \text{i32}
\]

\[
%2 = \text{arith.addi } 0, %1 : \text{i32}
\]
Deciding on a Unified Design

- Constant equality via object identity
- Easier extensibility
- Debug info on constants
- Multi-threading

What is the right design in the long term?
Where is the (near-term) value?
Eliminating `llvm::Constant`

- Value → User → Constant
  - ConstantData
    - ConstantInt, ConstantFP, ...
  - ConstantDataSequential
  - ConstantAggregate
    - ConstantStruct, ConstantArray, ConstantVector
  - ConstantExpr
  - GlobalValue
    - GlobalIndirectSymbol
  - GlobalObject
    - Function
    - GlobalVariable
  - BlockAddress
  - DSOLocalEquivalent
  - UndefValue
  - PoisonValue

1. Remove from `GlobalVariable` initializers
   - Standalone representation of constant data
2. Remove from metadata
3. Introduce symbol reference instruction(s), detach `GlobalValue` from the Value hierarchy
4. Introduce constant instruction(s) for plain data, remove `Constant`
Example 3: Regions

(N/A) vs.

Function

BasicBlock

vs.

FuncOp

other operations

Region

Block

```plaintext
scf.reduce(%cf) : f32 {
^bb0(%lhs: f32, %rhs: f32):
  %1 = arith.addf %lhs, %rhs : f32
  scf.reduce.return %1 : f32
}
```
The near-term value of regions in LLVM

- Wave-wide mode in AMDGPU: side-stepping regular control flow
  - Analogous to “unmasked” in ISPC
- Waterfall loops in AMDGPU: highly regular loop with unusual and complex loop condition that does not benefit from generic loop transforms
- Frontends for C-like languages: scopes and lambdas
Regions for the LLVM substrate

1. LLVM::Region exists but means something quite different; rename it
2. Insert Region into the Function/BasicBlock relationship
   • Change BB->getParent() to return Region instead of Function; replace by BB->getFunction() where necessary and appropriate
   • Iterate over F->getBody() instead of *F
3. Define a concept of region passes
4. Audit existing function passes: which ones should really be region passes?
5. Add regions to instructions
Recap

- Dialects and substrates
- Benefits of a unified substrate
- Thought experiments on unification of LLVM and MLIR
- The delta matrix
- Feasibility and near-term value of some column refactorings
The End
COPYRIGHT AND DISCLAIMER

©2022 Advanced Micro Devices, Inc. All rights reserved.

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.’ AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
Dialect implementation that is generic over the substrate

- Using a generic builder to create a generic instruction.
- Behind the scenes, the template parameter is resolved to the substrate-specific instruction class

```cpp
builder.template create<generic::AddInst>(lhs, rhs);
```

- Using `dyn_cast<T>` with a generic instruction.
- The result type is a function of both the generic target instruction and the substrate-specific operand type, so the result type can be substrate-specific as well.

```cpp
for (auto &inst : basicBlock) {
    if (auto *alloca = dyn_cast<generic::AllocaInst>(&inst)) {
        ...
    }
}
```