Modular

Mojo Debugging
Extending MLIR & LLDB for new languages
Agenda

01  Mojo 🔥

02  Extending LLDB

03  DebugInfo in MLIR
Mojo at a glance

Pythonic system programming language
- Driving SoTA in compiler and language design
- Forget everything you know about Python! :-)

One year old and still in development
- Freely available on Linux, Mac and Windows
- Full LLVM-based toolchain + VSCode LSP support
- Powers CPU and GPU AI programs written in the same language

Available now
- modular.com

```python
fn mandelbrot_kernel[
    simd_width: Int
](c: ComplexSIMD[float_type, simd_width]) ->
    SIMD[float_type, simd_width]:
    ""
    A vectorized implementation of the
    inner mandelbrot computation.
    ""
    let cx = c.re
    let cy = c.im
    var x = SIMD[float_type, simd_width](0)
    var y = SIMD[float_type, simd_width](0)
    var y2 = SIMD[float_type, simd_width](0)
    var iters = SIMD[float_type, simd_width](0)
    var t: SIMD[DT[y[bool, simd_width]] = True
    for i in range(MAX_ITERS):
        if not t.reduce_or():
            break
        y2 = y * y
        y = x.fma(y + y, cy)
        t = x.fma(x, y2) <= 4
        x = x.fma(x, cx - y2)
        iters = t.select(iters + 1, iters)
    return iters
```
And now it has a debugger! 🔥
Mojo Debugging

Mojo Code → DebugInfo Dialect → LLVM Dialect → LLVM IR → Language Extension → Editor
Extending LLDB
Supporting Mojo in LLDB

- LLDB doesn't have first class support for non-clang based languages.
Supporting Mojo in LLDB

• Some previous experiences:
  • Fork LLDB (e.g. swift)
    • Maintenance pain
    • Compiler integration
      • Expr eval
      • Conditional breakpoints
      • IR-based features
Supporting Mojo in LLDB

- Some previous experiences:
  - Pretend to be C++ (e.g. go-lldb)
    - Very buggy experience...
Supporting Mojo in LLDB

- Some previous experiences:
  - Limit to pretty printing (e.g. rust-lldb)
    - No compiler integration
    - Very limited expr eval
    - No cool features
Full compiler integration via a runtime plugin

Vanilla LLDB

•  > plugin load libMojoLLDB.so
Full compiler integration via a runtime plugin

Problems:
- LLDB didn't export all internal symbols
  - Needed by our DWARF parser
- Core LLDB assumed all languages are clang-based
Full compiler integration via a runtime plugin

Major upstream changes
- Namespaced internal plugin symbols
  - E.g. lldb_plugin::dwarf
- Selective export of private symbols via CMake (PR 68013)
- Removed assumptions on clang
  - Arbitrary typesystems
Current status of the debugger

Linux + MacOS
JIT + AOT debugging
Variable printing
Stepping
Breakpoints
REPL / Jupyter Notebooks

Coming soon:
Expression evaluation
REPL debugging
Windows support
Smart formatters
Better inline support
Debug adapter protocol

Modern interface for IDE integration
• VS Code, Visual Studio, Eclipse, Vim, Emacs, etc.
• lldebug

lldebug / VS Code-first
• UX focused
• Lower learning curve

Two-click JIT debugging
Extending the debug adapter protocol

Richer variable metadata for nicer features

```
30 var int_vec = List[Int](capacity=3)  int_vec = (size 1)[2]
31 var input = 2  input = 2
32 int_vec.append(input)  int_vec = (size 1)[2], input = 2
33 int_vec.append(3)  int_vec = (size 1)[2]
```

Inline variables: talking to the LSP
Future work - notebook debugging

- Used very much!
- New debugging interface
  - Need to enable REPL debugging within an editor
- Python-interop in Mojo
  - debugging and variable printing
Future work
Mojo-based formatters

Decorator-based formatters

- Based on common archetypes

```swift
@lldb_formatter_list
struct List<T>:
    var data: AnyObject[T]
    var size: Int
    var capacity: Int
```

```
> point_vec: (size 2)
<table>
<thead>
<tr>
<th></th>
<th>[0]: {x:1, y:-1}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[1]: {x:2, y:-2}</td>
</tr>
</tbody>
</table>
```
Future work
Mojo-based formatters

Function-based summary providers
• Matches python experience
• Requires interpretation

```rust
struct URL {
    fn __repr__(self) -> String {
        ...
    }
}
```

> url: http://localhost/Index.html
03
DebugInfo in MLIR
Infrastructure for keeping track of source level concepts in the IR across transformations.
Debug Info in LLVM

Source Level Debugging with LLVM

Debug metadata on instructions
- E.g. DILocation
- Source location & scope

Debug intrinsics
- E.g. dbg.declare, dbg.value
- Keep track of the location / value of source variables

%i.addr = alloca i32, align 4, !dbg !3
call void @llvm.dbg.declare(
    metadata ptr %i.addr,
    metadata !1,
    metadata !DIExpression())

!1 = !DILocalVariable(name: "i", ...)
!2 = !DISubprogram(name: "main", ...)
!3 = !DILocation(scope: !2, line: 2, ...)
Debug Info in the LLVM Dialect

Lowering target for custom debug info constructs in MLIR for codegen via LLVM.
Direct LLVM Mappings

Scoped location tracking with DIAttrs

Variable value tracking with DbgValueOp & DbgDeclareOp

Two-way translation to/from LLVM IR

```mlir
%0 = llvm.mlir.constant(1:i32) : i32
%1 = llvm.allocate %0 x i32
     : (i32) -> !llvm.ptr
llvm.intrinsic.dbg.declare #var = %1
     : !llvm.ptr loc(#loc1)

#loc = loc("test.mlir":2:4)
#loc1 = loc(fused[#di_subprogram][#loc])
#main = #llvm.di_subprogram<
    name = "main", ...>
#var = #llvm.di_local_variable<
    scope = #main, name = "i", ...>
```
DebugInfo Dialect

Exploration of a generic MLIR solution for keeping track of source level information.
Keeping track of source level concepts in IR across transformations

**DebugInfo Dialect**

01. **Scope-Based Locations**
   Augments MLIR’s native Location tracking with source scopes.

02. **Source Variable Tracking**
   Tracks value / location of source variables, and their lifetimes.

03. **Transformation Hooks**
   Utilities to incentivize maintaining debuginfo across transformations.
   Conversions into LLVM dialect.
Scoped Locations

MLIR has operation locations
• Missing source scope information

DebugInfo provides attributes for describing scopes
• DILevel
• DILocation
• DILocationName

Scope is fused onto operation locations

```c
#subprogram = #debuginfo.subprogram<
  compileUnit = #compile_unit,
  scope = #file,
  name = #main_name,
  linkageName = "main()",
  file = #file,
  line = 1,
  scopeLine = 1,
  subprogramFlags = Definition
> : !subroutine
#loc0 = loc("test.mojo":2:5)
#loc1 = loc(fused<#subprogram>[#loc0])
```
Variable Tracking

Tracks the value / location of a source variable starting from a program point.

In-line operation `debuginfo.value`:
- Relates an IR Value with a variable in the source program
- Similar to the LLVM intrinsic `dbg.value` (with subtle semantic differences)

```plaintext
%0 = ...
ddebuginfo.value #var = %0 : index
%1 = ...
ddebuginfo.value #var = %1 : index

#var = #debuginfo.local_variable<
  scope = #subprogram,
  name = "x",
  file = #file,
  line = 2
> : !debuginfo.unresolved<index>
```
DI Expressions

A high-level counterpart to DWARF expressions.

Models the transformations needed to map the IR value to a source variable.

// Mojo
var x: Int = 42

// MLIR
%index42 = kgen.param.constant = <42>
debuginfo.value #var = %index42 : index

#var = #debuginfo.local_variable<
   scope = #foo,
   name = "x",
   file = #file,
   line = 2
> : !debuginfo.unresolved#index>
DI Expression Representation

DWARF & LLVM DI Expressions are modeled with a stack machine

DebugInfo uses a typed attribute tree
- An "inner" operator is parameterized on other DIExpressions
- A "leaf" operator models constants or inputs (e.g. `debuginfo.expr.$irvalue`)

```cpp
#var = debuginfo.local_variable<...>
    : !debuginfo.unresolved<index>

#deref =

%0 = stack_allocation 1 x index
debuginfo.value #var #deref = %0
    : pointer<index>
```
Example: DeRef

Value may become stack allocated
- Source type is still `index`
- IR type is now a pointer type

DI Expression helps reconcile this difference
- Encodes the "inverse" operation to get back the source representation

```c
#var = #debuginfo.local_variable<...> : !debuginfo.unresolved<index>

#deref = #debuginfo.expr.deref<#irValue> : !debuginfo.unresolved<index>
#irValue = #debuginfo.expr.irvalue : !debuginfo.ti.ptr<
            !debuginfo.unresolved<index>>

%0 = stack_allocation 1 x index
debuginfo.value #var #deref = %0 : pointer<index>
```
Incentivizing DebugInfo Update

Handle representational changes.

Type-checked `debuginfo.value`
- Operand matches `expr.irvalue` type

Leaf-replacer utility
- Register the "opposite" logic of the pass using DI Expression operators
- Cached & type-checked

```cpp
DebugInfo::DIExprAttr
stackAllocLeafConversion(
    DebugInfo::DIType irType) {
    // newIRType is a pointer to `irType`.
    // New leaf (expr.irvalue) has newIRType.
    // Wrap with operator opposite of stack allocation (expr.deref) & return.
}

DebugInfo::DIExprLeafReplacer
leafReplacer(stackAllocLeafConversion);

leafReplacer.apply(  
    debugValueOp.getConversionExpr())
```
Lifetime Tracking

Mojo eager destruction
- Value is destroyed after last use
- Can no longer rely on scopes for local variable lifetime

In-line operation `debuginfo.kill`
- A special kind of `debuginfo.value`
  denoting a dead value
- Lowers into "killed" DbgValueOp in LLVM

```rust
fn main()
{
    var text = "Hello World"
    print(text)
    # `text` destroyed here.
    # `debuginfo.kill` inserted.
    foo()
}
```
Inheriting Locations

Beware of assigning locations for derived ops (esp. merged or deduplicated ops)
- Heavily influences stepping behavior
- E.g. changes to MLIR canonicalizer
What We Learned

- Not all debug info are preserved equally through the LLVM backend
  - Stack-allocated values & DbgDeclare work best
- IR & debug info co-design for efficient transformations
  - Don't forget tests
- DebugInfo representation can still be iterated upon
  - Ongoing LLVM work too
Mojo Debugging

Source Code → DebugInfo Dialect → LLVM Dialect → LLVM IR → Language Extension → Editor

MLIR → LLVM → DWARF → LLDB