SemIR: Carbon’s high-level semantic IR

Richard Smith
@zygoloid

EuroLLVM 2024
What is Carbon?

New language, emphasizing:

- C++ interoperability
- Efficient compilation
- Compile-time evaluation
- ...

Carbon learning from C++
Carbon *toolchain* learning from Clang
Clang AST

- Lex
- Token
- Parse
- ActOn* calls
- Sema
- Clang AST
- CodeGen
- LLVM IR

ActOn* calls
Clang AST

```
int max(int a, int b) {
    return a > b ? a : b;
}
```
int max(int a, int b) {
    return a > b ? a : b;
}
Clang AST

Problems:

- No clear distinction of syntax vs semantics
  - Challenge for tooling
- Semantic information incomplete: control flow, destructors
  - Duplicated work in lowering, constant evaluation, static analysis
  - ClangIR (MLIR) should eventually fix this
- Substantial effort required to make AST types small
  - `clang::DeclRefExpr` still $\geq$ 32 bytes
Carbon SemIR

- Data-oriented: dense arrays, side tables
- Parse tree stored compactly in memory
  - 12 bytes per parse node (might be 8 soon)
  - ~1 parse node per token
- SemIR purely semantic
  - Backreferences to parse tree
  - SSA form
  - Fixed instruction size, two operands, 16 bytes per instruction
fn Max(a: i32, b: i32) -> i32 {
    return if a > b then a else b;
}
Explicit control flow with block arguments
Carbon SemIR

```carbon
fn @Max(%a: i32, %b: i32) -> i32 {
  !entry:
  %a.ref.loc10_13: i32 = name_ref 'a', %a
  %b.ref.loc10_17: i32 = name_ref 'b', %b
  %.1: <function> = interface_witness_access @impl.%.1, element0 [template = @impl.%Greater]
  %.loc10_15.1: <bound method> = bound_method %a.ref.loc10_13, %.1
  %.loc10_15.2: init bool = call %.loc10_15.1(%a.ref.loc10_13, %b.ref.loc10_17)
  %.loc10_10.1: bool = value_of_initializer %.loc10_15.2
  %b.ref.loc10_17: i32 = name_ref 'b', %b
  !if.expr.then: %a.ref.loc10_24: i32 = name_ref 'a', %a
  br !if.expr.result(%a.ref.loc10_24)
  !if.expr.else:
  %b.ref.loc10_31: i32 = name_ref 'b', %b
  br !if.expr.result(%b.ref.loc10_31)
  !if.expr.result:
  %.loc10_10.2: i32 = block_arg !if.expr.result
  return %.loc10_10.2
}
```

- Explicit control flow with block arguments
- High-level expression type and category modeled
Carbon SemIR

```rust
def @Max(%a: i32, %b: i32) -> i32 {
    !entry:
    %a = name_ref 'a', %a
    %b = name_ref 'b', %b
    %.1: <function> = interface_witness_access @impl.%1, element0 [template = @impl.%Greater]
    %.loc10_15.1: <bound method> = bound_method %a, %1
    %.loc10_15.2: init bool = call %.loc10_15.1(%a, %b, %loc10_17)
    %.loc10_10.1: bool = value_of_initializer %.loc10_15.2
    %.loc10_15.3: bool = converted %.loc10_15.2, %.loc10_10.1
    !if.expr.then br !if.expr.then else br !if.expr.else
    !if.expr.then:
        %a = name_ref 'a', %a
    br !if.expr.result(%a)
    !if.expr.else:
        %b = name_ref 'b', %b
    br !if.expr.result(%b)
    !if.expr.result:
        %.loc10_10.2: i32 = block_arg !if.expr.result
    return %.loc10_10.2
}
```

- Explicit control flow with block arguments
- High-level expression type and category modeled
- Compile-time and runtime code coexist
Why not MLIR?

MLIR is great:

- Mutability
- Def-use chains
- Built-in analyses and transforms
- Extensible and flexible
Why not MLIR?

MLIR is great:

- Mutability
- Def-use chains
- Built-in analyses and transforms
- Extensible and flexible

Greatness comes at a cost:

- Performance: lots of pointer chasing
- Memory usage: \( \text{sizeof(mlir::Operation)} \) with two operands is:
  - 72 bytes directly in object
  - 32 bytes in separate allocations
  - \( \geq 5 \times \text{sizeof(SemIR::Inst)} \)
Lowering

Current plan, and development builds:
Lowering

Current plan, and development builds:

Possible future, for peak performance:
Conclusion

• Custom data model helped achieve performance and memory goals
• Data-oriented design is extra work, but often worth it
• Use MLIR when it makes sense, but like any tool, it’s not universally applicable