SemIR: Carbon's high-level semantic IR

Richard Smith
@zygoloid

EuroLLVM 2024

What is Carbon?

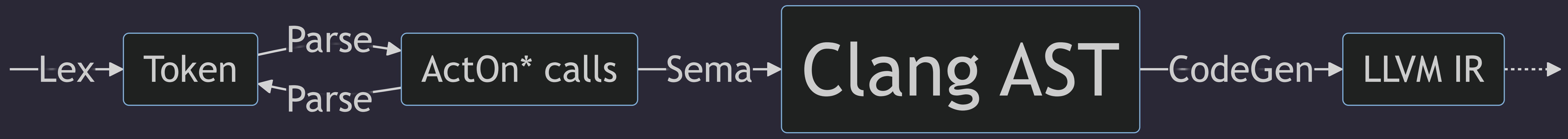
New language, emphasizing:

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

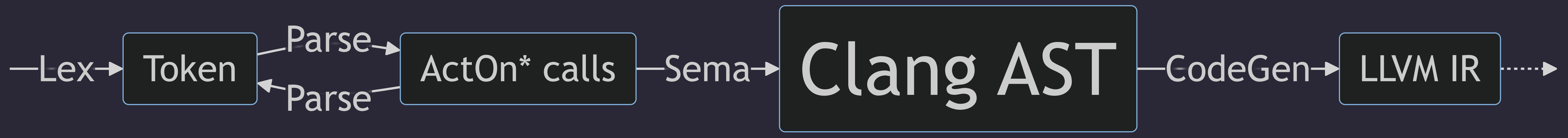
Carbon learning from C++

Carbon *toolchain* learning from Clang

Clang AST

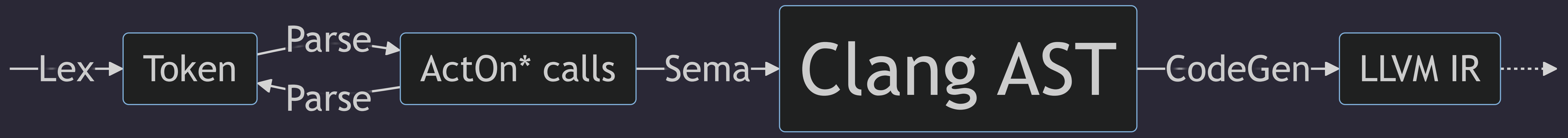

Lex
Token
Parse
Parse
ActOn* calls
Sema
Clang AST
CodeGen
LLVM IR

Clang AST




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

Clang AST



C++

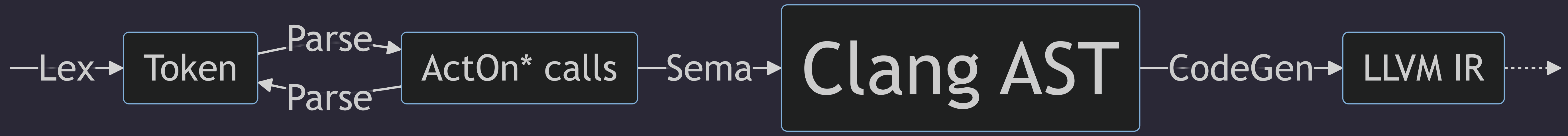
```cpp
int max(int a, int b) {
  return a > b ? a : b;
}
```

Clang AST

```
`-FunctionDecl 0xbb972d8 <<source>:1:1, line:3:1> line:1:5 max 'int (int, int)'
  |-ParmVarDecl 0xbb97170 <col:9, col:13> col:13 used a 'int'
  |-ParmVarDecl 0xbb971f0 <col:16, col:20> col:20 used b 'int'
  `-CompoundStmt 0xbb97500 <col:23, line:3:1>
    `-ReturnStmt 0xbb974f0 <line:2:3, col:22>
      `-ImplicitCastExpr 0xbb974d8 <col:10, col:22> 'int' <LValueToRValue>
        `-ConditionalOperator 0xbb974a8 <col:10, col:22> 'int' lvalue
          |-BinaryOperator 0xbb97448 <col:10, col:14> 'bool' '>'
          | |-ImplicitCastExpr 0xbb97418 <col:10> 'int' <LValueToRValue>
          | | `-DeclRefExpr 0xbb973d8 <col:10> 'int' lvalue ParmVar 0xbb97170 'a' 'int'
          | `-ImplicitCastExpr 0xbb97430 <col:14> 'int' <LValueToRValue>
          |   `-DeclRefExpr 0xbb973f8 <col:14> 'int' lvalue ParmVar 0xbb971f0 'b' 'int'
          |-DeclRefExpr 0xbb97468 <col:18> 'int' lvalue ParmVar 0xbb97170 'a' 'int'
          `-DeclRefExpr 0xbb97488 <col:22> 'int' lvalue ParmVar 0xbb971f0 'b' 'int'
```

- Represents both syntax and semantics

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 ≥ 32 bytes

Carbon SemIR

$\xrightarrow{\text{Lex}}$ Tokens $\xrightarrow{\text{Parse}}$ Parse tree $\xrightarrow{\text{Check}}$ **Carbon SemIR** $\xrightarrow{\text{Lower}}$ LLVM IR $\dashrightarrow$

- 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

Carbon SemIR

Carbon

```
fn Max(a: i32, b: i32) -> i32 {
  return if a > b then a else b;
}
```

Carbon SemIR

```
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
  %.loc10_15.3: bool = converted %.loc10_15.2, %.loc10_10.1
  if %.loc10_15.3 br !if.expr.then else br !if.expr.else

!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
}
```

Carbon SemIR

Carbon SemIR

```
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
  %.loc10_15.3: bool = converted %.loc10_15.2, %.loc10_10.1
  if %.loc10_15.3 br !if.expr.then else br !if.expr.else

!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

Carbon SemIR

```
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
  %.loc10_15.3: bool = converted %.loc10_15.2, %.loc10_10.1
  if %.loc10_15.3 br !if.expr.then else br !if.expr.else

!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
}
```

Carbon SemIR

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

Carbon SemIR

Carbon SemIR

```
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
  %.loc10_15.3: bool = converted %.loc10_15.2, %.loc10_10.1
  if %.loc10_15.3 br !if.expr.then else br !if.expr.else

!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
- 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:

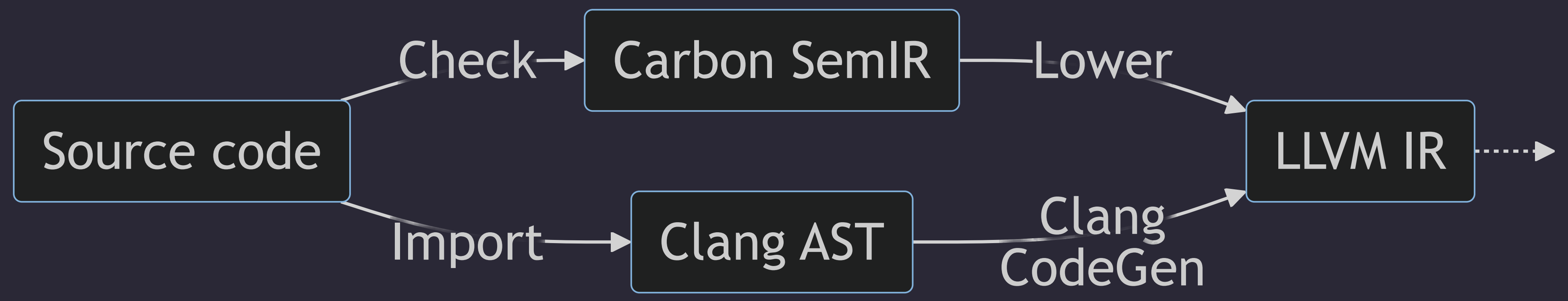
- Mutability
- Def-use chains
- Built-in analyses and transforms
- Extensible and flexible

Greatness comes at a cost:

- Performance: lots of pointer chasing
- Memory usage: `sizeof(mlir::Operation)` with two operands is:
  - 72 bytes directly in object
  - 32 bytes in separate allocations
  - ≥ 5x `sizeof(SemIR::Inst)`

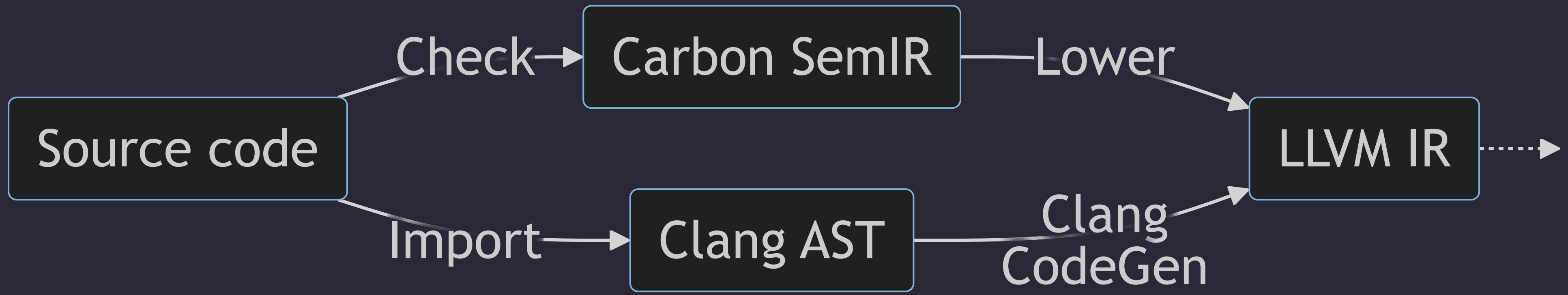
Lowering

Current plan, and development builds:

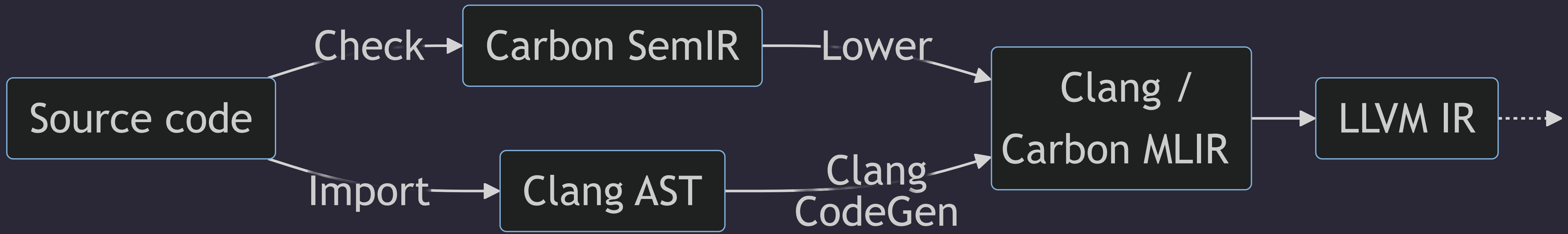


Lowering

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