Extendable and Decoupled Multi-Level Intermediate Representation (MLIR) Compilation with JavaScript

Pedro Ramalho[1 [0009−0001−6518−0378]], Manuel Cerqueira da Silva[1 [0009−0000−8383−9693]], João Bispo[1 [0000−0002−3017−9449]], Nuno Paulino[1 [0000−0001−5547−0323]]

INESC TEC and Faculty of Engineering, University of Porto

Abstract. The promise of the MLIR framework is to be a universal Intermediate Representation (IR) that can be used on all processing steps of the compiler, from the high-level Abstract Syntax Tree (AST) down to the low-level representation. To achieve this, MLIR introduces the concept of dialects, to encapsulate specific abstractions, and allows the co-existence of any combination of dialects in the same IR.
However, MLIR faces composability and extensibility issues, which hinder the original objective of providing a "reusable and extensible compiler infrastructure". Extending the MLIR ecosystem with a new dialect usually requires modifications to the framework, immediately causing fragmentation. Additionally, the MLIR ecosystem has a steep learning curve, which hinders adoption by new developers.
In this paper, we propose an approach that intends to lower the entry barrier and improve the composability and extensibility of MLIR, as well as a concrete solution for the first steps of obtaining and parsing MLIR. Since the MLIR language itself is open to extensions, this requires a more agile and dynamic approach to processing it. Our solution is based on a compiler, built on top of the LARA framework, which does not pre-impose restrictions on the code that it can parse and transform.

Keywords: MLIR · LLVM · ANTLR · compilers · source-to-source compilation.

1 Introduction

Modern compilation technologies are addressing the growing complexity of programming languages and hardware targets. A key development is the Multi-Level Intermediate Representation (MLIR), which unifies various levels of abstraction within a single framework, offering flexibility for diverse compilation needs.

However, developing custom MLIR dialects is complex due to the extensive C++ codebase, presenting a steep learning curve that hinders adoption, especially in research. This paper proposes an approach to streamline dialect development using ANother Tool for Language Recognition (ANTLR) [1] to create

an MLIR grammar that simplifies parsing and decouples it from MLIR's strict environment, lowering the entry barrier.

We discuss the background and challenges of MLIR dialect development, our ANTLR-based parsing approach, and its implications, proposing directions for future research.

2 MLIR Overview

MLIR is a subproject of LLVM, designed as a unified framework for compiler development targeting various computing platforms like CPUs, GPUs, DPUs, TPUs, FPGAs, AI ASICs, and quantum systems. It supports multiple dialects and progressive conversions towards machine code, retaining high-level abstraction information to enhance analysis and transformation accuracy.

MLIR can be represented in three forms: human-readable text for debugging, in-memory for transformations and analyses, and compact serialized for storage and transport.

2.1 Dialects and Operations

Dialects in MLIR define new operations, attributes, and types within unique namespaces. Multiple dialects can coexist in one module and be converted between each other. For instance, LLVM's intrinsics are represented in the LLVM dialect.

Operations are extensible, identified by unique strings, and can have multiple results, operands, successors, enclosed regions, properties, and attributes. They can use custom or generic assembly formats for printing. For example, the operation below represents a power operation in the `ignota` dialect, with specific attributes and types. Every MLIR operation has an associated, mandatory source location:

```
%r = "ignota.powf"(%base, %exponent) {inplace = true}
(f32, f32) -> f32 loc("file/path":10:1)
```

2.2 Challenges of Extending MLIR

Creating custom operations and types in MLIR typically involves developing a dialect. The `standalone-dialect` [6] is a useful template for this purpose.

However, MLIR's scattered documentation complicates learning, and understanding its integration with the complex Low Level Virtual Machine (LLVM) infrastructure requires significant effort. Expanding MLIR's ecosystem is challenging due to the complexity of defining new operations and passes. Additionally, MLIR tools depend on specific LLVM versions, leading to potential compatibility issues. MLIR's strict environment enforces consistency but limits flexibility, restricting developers to their native capabilities and hindering integration with external tools.

3 Proposed MLIR Processing Methodology

Figure 1 illustrates the proposed methodology. We use `mlir-opt` to output operations in the generic format, using the flag `--print-op-generic`[1], ensuring standardization across all MLIR operations and avoiding the impracticality of parsing dialect-specific formats.

Following this, our in-house tool, AnyWeaver, uses an ANTLR-based parser to convert the generic MLIR code into an AST. We can then modify the MLIR using a top-level JavaScript file (Analysis Script in figure 1), which processes the AST and outputs MLIR code in the generic format.

A limitation of this approach is its incompatibility with dialects that extend MLIR semantics. For example, CIRCT [5] uses non-Static Single-Assignment (SSA) semantics, causing errors when outputting in the generic format.

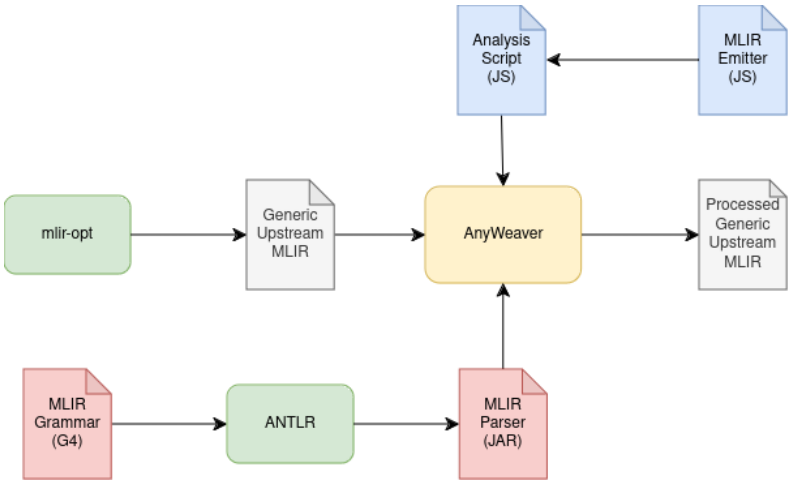

Fig. 1. Processing Flow Using an MLIR Grammar and AnyWeaver

3.1 The LARA Framework and AnyWeaver

LARA is a framework, written in Java, for building source-to-source compilers (or transpilers). While the compiler is a Java application, compilation passes are written in JavaScript, as libraries, and interpreted by the compiler, in order to promote composability and extensibility.

Usually, each LARA compiler supports a single language, and developing a new compiler usually requires some effort, since it is necessary to provide an AST for the language, as well as the points in the code that can be selected

[1] This is a hidden flag that is listed when using `--help-hidden`.

and corresponding attributes. Any change to one of these components implies creating a new version of the compiler.

AnyWeaver is a LARA compiler that does not predefine a language or an AST, but instead, is capable of loading AST-like structures and working over them, without prior knowledge of the underlying structure.

3.2 A Transformation Example

The following example illustrates how to transform an operation from one dialect into an equivalent operation in another dialect. This transformation is facilitated by using ANTLR to parse the MLIR code and a JavaScript-based transformation script.

```
import Query from "lara-js/api/weaver/Query.js";
import Io from "lara-js/api/lara/Io.js";
import Parsers from "anyweaver-js/api/weaver/Parsers.js";

// Parse MLIR code
const mlirAst =
    Parsers.antlr(Io.readFile("test.mlir"), "root", "Mlir",
        "pt.up.fe.specs.mlir.grammar");

// Add AST to current root
Query.root().addAst(mlirAst);

// Find all operations with name "toy.reshape", change to "ignota.reshape"
for(const op of Query.search("GenericOperation")) {
  if(op.getValue("name") !== "\"toy.reshape\"") {
    continue;
  }

  op.setValue("name", "\"ignota.reshape\"");
}

// Print name of operations
for(const op of Query.search("GenericOperation")) {
  console.log(op.getValue("name"));
}
```

The provided code makes use of an ANTLR grammar through the Weaver framework, which allows querying and manipulating the AST of the parsed MLIR code. The script performs a transformation where all operations named `"toy.reshape"` are renamed to `"ignota.reshape"`.

4 Conclusion and Future Work

This paper addresses MLIR's composability and extensibility issues, and its steep learning curve for new developers. We proposed an approach to lower the entry barrier and improve the framework's adaptability. Our solution, which utilizes the LARA framework, provides a flexible method for parsing and transforming MLIR without pre-imposed restrictions.

Future work will focus on expanding the ANTLR grammar in order to fully support MLIR modules.

Acknowledgments. This work has been financially supported by A-IQ READY project, which has received funding within the Chips Joint Undertaking (Chips JU) and National Authorities under grant agreement No. 101096658, and was also supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under project 2022.06780.PTDC (DOI:10.54499/2022.06780.PTDC).

References

1. Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. “MLIR: Scaling compiler infrastructure for domain specific computation.” In 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 2-14. IEEE, 2021.
2. Terence Parr. 2013. The Definitive ANTLR 4 Reference (2nd. ed.). Pragmatic Bookshelf.
3. João Bispo, and João MP Cardoso. Clava: C/C++ source-to-source compilation using LARA. SoftwareX, Volume 12, 2020, Article 100565.
4. William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko presented their work on raising C to Polyhedral MLIR in the paper titled "Polygeist" at the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT ’21).
5. CIRCT Contributors. CIRCT: Circuit IR Compilers and Tools. Available at: https://circt.llvm.org/. Accessed: 11 July 2024.
6. MLIR Contributors. Standalone Dialect in MLIR. Available at: https://github.com/llvm/llvm-project/tree/main/mlir/examples/standalone. Accessed: 11 July 2024.