Planned Improvements to the Chapel Compiler

Michael P. Ferguson
Hewlett Packard Enterprise
USA
michaelferguson@acm.org

ABSTRACT

The current architecture of the Chapel compiler makes it difficult to add new features such as separate compilation and IDE integration that are frequently requested by Chapel users. This talk will discuss plans for an improved compiler architecture that can better support these features.

KEYWORDS

compiler architecture, chapel, parallel programming

1 INTRODUCTION

The Chapel compiler has supported the development of the Chapel language and enables high performance applications. However it can be difficult to work with due to long compile times, limited error reporting, lack of Integrated Development Environment (IDE) integration, and missing support for separate compilation. Additionally, there is a steep learning curve to become familiar with the compiler implementation.

One cause of these problems is the structure of the Chapel compiler itself. Both the program representation and the pass structure need improvement.

The compiler uses an Abstract Syntax Tree (AST) representation to represent programs throughout compilation. It uses this AST representation until LLVM Intermediate Representation (LLVM IR) [3] or C source code is generated. Since the same AST classes need to represent programs at many different points in compilation, these AST nodes have different constraints at different moments during compilation. This fact makes it difficult to understand all of the assumptions embedded in the AST or even to document them. This AST representation is not particularly well suited for traditional analyses needed for optimization such as alias analysis. In addition, this AST representation is only really intended to aid the compiler and it isn't suitable for tools such as a linter or semantic highlighter.

In terms of pass structure, the compiler divides the task of compilation into 41 passes. Each of these passes operates in order across the whole program and mutates the AST in-place to achieve its goals. This pass structure makes it difficult to extend the compiler into other use cases such as IDE integration.

2 IDE INTEGRATION

Integrated Development Environment (IDE) integration is a frequently requested feature from Chapel users. The Jupyter system [1] and the Language Server Protocol [2] are two example systems that enable compilers to support IDE integration across many editors. When combined with an appropriate program editor, these systems enable productivity-enhancing operations such as semantic highlighting, code completion, and explaining some program context on mouse-over.

To support IDE integration, the Chapel compiler will have to become more like a server than a program that processes some input files and produces some output files. While this server is running, it will need to have the ability to respond to small queries from the user as quickly as possible.

3 SEPARATE COMPILATION

Separate compilation for Chapel is challenging because there are generic functions and no equivalent to header files. However, it is possible to still have a form of separate compilation by redefining *compile* and *link* from what they historically mean in C-like languages. In particular, *compile* can include generic code in a library and *link* can instantiate this generic code.

Implementing such a separate compilation strategy is difficult in the current compiler because:

- the *compile* step needs to be able to compile a library without also compiling its dependencies, but currently the compiler can only operate as a whole-program compiler.
- the *link* step should not go through the entire compilation process; rather it should focus on instantiating any newly needed generics, connecting function calls to their concrete implementations, and generating dispatch tables.

Improving the compiler architecture can help to make adding these features easier.

4 PLANNED ARCHITECTURE IMPROVEMENTS

To improve the situation, we are planning to migrate the compiler to a new architecture. The new architecture will divide compilation into three phases:

- a Responsive Compiler phase
- a mid-level IR phase
- a low-level IR phase

The *Responsive Compiler* phase is focused on interactivity and incremental compilation. It is based on a talk describing elements of the Rust compiler [4]. In this phase, compilation proceeds by computing the results of side-effect-free *queries*. These queries will depend on other queries and a framework will manage re-using query results when possible. This strategy will be new to the Chapel compiler.

A mid-level IR is a program representation that is not as high-level as ASTs but still higher level than something like LLVM. Swift, Rust, and Julia compilers use mid-level or high-level IRs [5]. At the same time, most of the passes within the existing Chapel compiler could be considered operating on a mid-level IR. However in the future we hope to improve the program representation and pass structure by migrating this portion of the compiler to MLIR [5].

Even before we do that, we will need to adjust these passes to process a function at a time.

Finally, the compiler will generate a low-level IR and perform optimizations on it as it does now. Using [3] as this low-level IR has the benefit of the compiler being able to use and control a variety of optimizations.

5 BUILDING A COMMUNITY OF COMPILER DEVELOPERS

As it stands now, the Chapel compiler is difficult enough to get familiar with that it is rare that a new contributor to the Chapel project will be able to do something with the compiler.

While improving the compiler architecture, this project aims to improve this situation in three ways.

First, the new development for the first part of the Chapel compiler will focus on creating a library for Chapel compilation. That way, community members can use this library to implement their own tools that work with Chapel source code, such as linters, code refactoring tools, and IDE integrations.

Second, the library for Chapel compilation will include API documentation that is generated from source code comments. That way, this documentation is less likely to go out of date. The aim is to include this compiler API documentation on the main Chapel documentation page.

Third, the other phases of compilation will rely more on MLIR and LLVM. These are technologies that community members interested in compilers may already be familiar with. By using them more in the construction of the Chapel compiler, we can reduce the barrier to entry for compiler developers in the community to contribute to the Chapel compiler.

6 CONCLUSION

We are beginning a focused effort towards improving the architecture of the Chapel compiler. We are expecting the improved compiler to be more equipped to address feature requests of IDE integration and separate compilation. Additionally, we hope that this improved compiler will form a good foundation for future efforts and support a growing community of compiler developers.

REFERENCES

[1] 2021. *Jupyter Client 6.2.* https://jupyter-client.readthedocs.io/

[2] 2021. *Language Server Protocol.* https://microsoft.github.io/language-server-protocol/

[3] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[4] Nicholas Matsakis. 2019. *Responsive Compilers talk at PLISS 2019.* https://www.youtube.com/watch?v=N6b44kMS6OM

[5] Tatiana Shpeisman and Chris Lattner. 2019. *MLIR: Multi-Level Intermediate Representation Compiler Infrastructure.* https://llvm.org/devmtg/2019-04/slides/Keynote-ShpeismanLattner-MLIR.pdf