Studying link-time optimizations in programming language development to facilitate the continuum of static and dynamic modules

David Callanan

21444104

Final Year Project – 2026

B.Sc. Single Honours in Computational Thinking

Department of Computer Science

Maynooth University

Maynooth, Co. Kildare

Ireland

A thesis submitted in partial fulfilment of the requirements for the

B.Sc. Single Honours in Computational Thinking.

Supervisor: Dr. Phil Maguire

Contents

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of the B.Sc. Computational Thinking qualification, is entirely my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

I hereby acknowledge and accept that this thesis may be distributed to future final year students, as an example of the standard expected of final year projects.

Signed: _____ Date: _____

Acknowledgements

I would like to thank my parents/guardians/siblings/classmate/friends for their ...

Abstract

The logic of computer software is primarily expressed using programming languages, but the balance between versatile and efficient programming languages remains a major challenge. Changes to the execution environment of a program can impact performance, or worse, cause incompatibilities that require major refactoring. Developers often maintain multiple branches of a codebase to keep performance to acceptable levels. One particular difference between execution environments is whether individual modules are statically linked at compile-time or dynamically resolved at runtime. In this research project, we propose a prototype programming language called "Essence C" that better supports the continuum of static and dynamic modules, by allowing the programmer to write code once that is agnostic to the linking strategy. By taking advantage of LLVM's intermediate representation, it is possible to advance the state-of-the-art of link-time optimizations across module boundaries.

Chapter 1 – Introduction

1.1 Overview

1.2 Motivation

1.3 Methodology

Thesis structure, and a resesarch strategy.

To address the objectives of this research, a design-science methodology is employed with a focus on continued interation and evaluation of "Essence C". This approach was selected because the project is concerned with the implementation of a novel programming language that advances the flexibility and performance of modules in software development, and thus requires a design and implementation process that is informed by ongoing evaluation.

The structure of this thesis is organized to reflect this methodology. Chapter 2 reviews existing literature.

1.4 Success Criteria

Success will be evidenced by the following deliverables:

(1) A functional prototype compiler that implements the designed module system;

(2) Documentation of the module system and other core language features;

(3) A simple IDE extension that provides syntax highlighting to the programmer;

(4) A body of research into the continuum of static and dynamic modules in software development, with kernel development as a case study;

(5) An implementation of link-time optimization techniques that leverage the designed module system;

(6) An evaluation of the effectiveness of these optimizations through the benchmarking of sample programs;

(7) A monorepo containing extensive git history, demonstrating ongoing development progress;

(8) A final written report detailing the research, design, implementation and evaluation of the project.

Chapter 2 – Technical Background

2.1 Literature Review

2.1.1 Static vs dynamic dispatch

Methods of class instances (or equivalent constructs) dispatch calls either statically or dynamically. When the exact implementation of a method is known at compile-time, static dispatch is typically used, involving a direct assembly call to the method. Since the method is known at compile-time, the compiler can use an immediate addressing, which is most efficient. In contrast, when the exact implementation of a method is not decided until runtime (or if the compiler cannot prove otherwise), dynamic dispatch is used, which involves an indirect call through a vtable or similar structure.

2.1.2 Static vs dynamic linking

There is much overlap in the literature between dispatch and linking, but linking focuses on larger units of code, often referred to as modules or libraries, as opposed to class instances. This concept exists in C where there is no inbuilt notion of static vs dynamic dispatch.

When the exact source code (implementation) of a module is known at compile-time, static linking is typically used, involving the direct inclusion of the module's code into the final executable. Then, the compiler can use immediate addressing to invoke functions or access data from this module. In contrast, when the exact source code of a module is not decided until runtime, dynamic linking is used which typically requires a Procedure Linkage Table (PLT) and Global Offset Table (GOT).

However, various techniques are already employed to optimize dynamic linking:

(1) Some architectures use the "relaxation" technique to replace indirection at runtime by modifying the assembly instructions.

(2) PLT Rewriting provided by compilers.

(3) Bespoke replacement (e.g. linux kernel static_call).

However, two primary concerns of interest are still evident:

(1) There is always some level of pointer indirection that cannot be avoided (nvm);

(2) Various techniques such as inlining become impossible due to this indirection.

Terrible PIMPLE idiom. Headers of headers.

2.1.3 Module loading in operating system kernels

2.1.4 Link-time optimizations

Traditionally, many optimizations were performed at the language level, before the program was compiled to object files. For instance, inlining could not be performed at link-time because there was insufficient information in the object files to reason about the semantics of the program. While dedicated link-time optimizations were introduced, their scope was limited for this reason.

However, LLVM IR sits in between the level of abstraction of source code and object files, and maintains higher-level semantics. By keeping the program in this intermediate representation longer (and treating this as modern object files), recent developments have made link-time optimizations more powerful, allowing for, for instance, inlining across module boundaries.

2.1.5 Inlining and IR-level linking

2.2 Technical Material

2.2.1 LLVM intermediatea representation

Chapter 3 – The Problem

TODO

Chapter 4 – The Solution

4.1 Design Elements

4.1.1 Unification of Modules and Classes

Languages have historically offered various features like classes, modules, packages, and so on, that have much overlap in their functionality. One of the core design decisions was to unify these elements into a single system that consists of a smaller unit called a "map" (which bares some resemblence with Rust- and go-like structs), and a larger organizational unit called a "module". Most importantly, every module still uses a map for instance-management purposes, and so the same capabilities are available whether or not a program is split into more modules or not. This differs from languages like C, where separate linkage units has an impact on the final prograsm. The decision for this unification stems from Rust's philosophy of "zero-cost abstractions", an acknowledgement that organizational differences in a program that aids with design and iteration should not impact the performance of the final artifact.

4.1.2 Map Enum Equivalence

Three features in programming languages worth considering are structs (or classes), functions (or methods), and enums (specifically, tagged unions). Structs and functions have been unified into a single feature called a "map".

One notable outcome is that there is no concept of multiple function parameters; instead, there is a single parameter which reuses the existing tuple syntax to achieve multiple parameters.

4.1.3 Additional benefit of inlining: interacting with existing languages without indirection.

4.2 Picking of overall solution

4.3 Design

4.3.1 Associativity

In most programming languages, addition and subtraction are treated as unary operations in an additive batch, whereras operations like multiplication and division are instead treated as binary operations in a multiplicative tree. However, it turns out that any associative operation (such as multiplication) can be written in the former form, where operations in the same family (such as division) control the unary effect via an operator that prefixes the operand. As such, a consistent syntax was employed in this language, allowing both the unary and binary approach for all associative operations and their respective intra-family operations. This includes the following families:

- Addition and subtraction (like existing languages);
- Multiplication and division;
- Logical AND;
- Logical OR;

The syntax was designed to follow the crystal/pistol pattern, some terminology devised to mean the following:

- Crystal: A unary batch of operator/operand pairs (an extension of how existing languages handle addition and subtraction).
- Pistol: A binary tree of operations (like how existing languages handle multiplication and division).

For precedence reasons, we have four separate rules: additive crystal -> additive pistol -> multiplicative crystal -> multiplicative pistol, with each embedding the next in the hierarchy. Those further to the right of the chain enjoy higher precedence.

4.4 Implementation

4.4.1 Mutability

LLVM IR is in SSA form, where registers are immutable. One cannot reuse a register, and must instead create a new register for every mutation. Then, one takes advantage of the phi instruction to pick which register is the correct one at runtime. However, the alloca (stack allocation) model operates under the familiar mindset of mutability, as it works with raw memory addresses. Conveniently, it was observed that the standard mem2reg optimization pass does a fantastic job at converting stack allocations into SSA registers where legal, bypassing the tough development of a custom SSA mutability mechanism. Thus variables are entirely implemented using stack allocations.

4.4.2 Tooling and Infrastructure

Parser Trace Explorer

Parsing issues were frequently encountered during the development cycle, and manual debugging became infeasible as complexity increased. Hence, a *Parser Trace Explorer* was developed, consisting of a JSON trace logger in the compiler frontend, and a separate web-app tool for exploring these logs. A simple SolidStart + Tanstack Router tech stack was selected for this tool due to existing familiarity with this stack, and because of the straightforward requirements of the tool.

Since parsers try a vast number of rules and are inherently recursive, the logs quickly exceed $500,000$ lines for small source files. To deal with this, the logs are structured as a tree, and the user of the tool is expected to navigate this tree by selecting the precise rule attempts that the parser is supposed to succeed on, thus hiding irrelevant attempts. A dedicated lexer, in hindsight, would have reduced the exponential growth of the logs, but might not have easily parsed some of the unique language features (such as semicolons acting as both terminators and comments).

4.4.3 Defensive Programming

As the language has strict semantics, the backend has no need to recover from illegal states. As such, defensive programming is used, where the program is crashed with a useful error message whenever an invariant is violated. This could be due to semantic violations in the user's code, or due to bugs in the compiler itself.

4.4.4 Debugging

While the primary mechanism of performing external communication is to link with external code via the C ABI (which may be LLVM IR), debugging functionality is often preferable within the language directly in case that there are issues with the interoperability layer itself. Thus, two basic logging functions were implemented, “log” and “log_d”. The former takes a null-terminated ascii string and prints this output in an implementation-specific manner. The latter takes an arbitrarily-sized value (like an integer or pointer) and prints its raw binary representation (in 64-bit chunks), including both its hex representation, ascii representation, and decimal representation. In this specificic language implementation, the external “puts” function was seletcted to implement this functionality, available in many environments. Todo: A flag was implemented to disable “puts” usage in case the environment does not support it, in which case, logs are dropped.

Chapter 5 – Evaluation

TODO

Chapter 6 – Conclusion

TODO

6.0.1 Contribution to state-of-the-art

- inlining - built in language constructs to avoid likes of kernel static_call - continuum of dispatch, not just static vs dynamic, but by merging the notion of class instances (and fusing classes and modules), if we detect multiple instances we can switch to an even more dynamic dispatch mechanism. - more efficient inter-language interoperability

Chapter 7 – Parsing Library

Prior to this project I had already developed a simple parsing library in JavaScript, which allows for:

(1) Parsing terminal rules using regular expressions.

(2) Forming non-terminal rules by combining other rules in the following ways, sufficient to develop any complex grammar:

   (1) “or” making rule optional

   (2) “join” sequencing rules

   (3) “multi” repeating rules zero or more times

   (4) “opt” making rule optional

(3) Map parsed data to custom structures using the “mapData” function.