University of Nevada, Reno

Foundations for a Data Oriented Compiler Infrastructure

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Computer Science and Engineering

by

Joshua Aaron Dahl

Dr. Frederick C. Harris Jr./Thesis Advisor

August, 2025

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

Joshua Aaron Dahl

Entitled

Foundations for a Data Oriented Intermediate Representation

be accepted in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Dr. Frederick C. Harris Jr., Ph.D., Advisor

Ms. Nancy LaTourrette, Commitee Member

Dr. Thomas Quint, Ph.D., Graduate School Representative

Dr. Markus Kemmelmeier, Ph.D., Dean, Graduate School

August, 2025

Abstract

Despite the fact that most college graduates possess the necessary skills to assemble a compiler, few actually take on the task. One major barrier is the lack of an accessible and well-designed Intermediate Representation (IR) that can support and guide new compiler authors. A thoughtfully constructed IR could significantly reduce the difficulty of compiler development and encourage more individuals to engage with this domain. To address this issue, we conduct a survey of existing compiler IR, paying particular attention to lesser-known or obscure designs that may offer untapped potential. In addition, we explore the feasibility of employing Entity Component Systems (ECS) to improve both performance and usability within compiler architecture. By combining insights from these analyses with a system capable of running code across a wide range of platforms, we lay the groundwork for a new IR; specifically aimed at lowering the barrier to entry into compiler design and empowering more developers to participate in this domain.

Acknowledgements

I would like to thank my committee Dr. Frederick C. Harris Jr., Ms. Nancy LaTourrette, and Dr. Thomas Quint for their invaluable guidance throughout my graduate studies. In addition, I would like to thank Dr. Sergiu Dascalu, Dr. Sushil Louis, and Dr. Batyr Charyyev for their supplemental assistance and the opportunities they have created for me.

I would also like to extend my deepest gratitude to my mother, whose unwavering support and gentle (yet persistent) reminders kept me focused and on track throughout this journey. Her encouragement made all the difference, especially when distractions threatened to derail my progress.

This material is based in part upon work supported by the National Science Foundation under grant numbers OIA-2019609 and OIA-2148788. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Tables

List of Figures

1 Introduction

Designing compilers is notoriously tricky. It is a sentiment echoed frequently across programming forums—when users are asked to name the most complex subject in computer science, someone inevitably brings up compilers [16]. However, what makes compiler construction so challenging?

The typical undergraduate curriculum covers many of the foundational topics necessary for building a compiler: computer architecture and assembly, algorithms, automata theory, and the design principles behind programming languages. In theory, this gives students all the pieces they need. Yet, despite this preparation, only one team in the history of the University of Nevada, Reno's Senior Capstone class has ever attempted to assemble those components into a complete compiler. Why is that?

We believe the key issue is scale. Even a simple compiler requires integrating many complex systems, making the task feel overwhelming. To make compiler construction more approachable, we propose a few strategies and supporting technologies that can significantly reduce the complexity:

1. Multilevel Intermediate Representations (IRs): Technologies like MLIR [80] and Rust's compiler architecture [103] offer the ability to define IRs that closely resemble the source language's constructs. These high-level IRs are composed of operations defined in lower-level IRs, making it easier to express language semantics naturally. The lower-level representations can then be substituted for higher-level ones, modularizing the compilation process in a similar way that Lego modularizes building.

2. Compile-Time Evaluation: Features such as Zig's comptime [133] and C++'s constexpr/consteval [24] allow for running code during compilation. When this capability is built directly into a compiler's IR—and combined with multilevel IRs—it enables language designers to embed optimization logic as part of the language definition itself, treating optimizations as first-class constructs rather than separate compiler passes.
3. Entity-Component Systems (ECS): ECS frameworks offer a data-oriented design pattern that can be a compelling alternative to traditional object-oriented techniques like visitor patterns or rigid inheritance hierarchies. By using ECS, developers can write ad-hoc analysis and optimization passes that are more modular and easier to reason about, fitting naturally into an IR-driven compiler structure.

Although platforms like LLVM cite{lattner2004} and GCC cite{gcc} are robust and widely used, we chose not to extend these existing compiler toolchains. Instead, our primary objective is to lower the barrier to entry in compiler development. Unfortunately, the architecture and APIs of these existing systems are often complex and, in many cases, poorly documented. This complexity can make it difficult for newcomers to get started or to meaningfully contribute. These systems also evolve rapidly, with internal APIs and components changing frequently. Such changes can easily break dependent projects, making long-term maintenance a challenge.

In contrast, our approach is to build a small, lightweight compiler infrastructure with a stable, fixed API that defers many implementation details to compile time and is paired with a consistent standard for implementing essential core features, making it straightforward and more approachable for less experienced developers or those seeking to integrate a compiler into a larger application.

The remainder of this thesis explores our preliminary investigations into these topics through the following chapters, all of which are either published or structured for submission as publications: Chapter 2 surveys existing compiler intermediate representations, identifying the most promising ideas to combine. Chapter 3 introduces Entity Component Systems in greater detail and examines how they can be used to reduce compiler complexity. Chapter 4 presents Mizu, a lightweight, low-level virtual machine designed for broad portability; Mizu provides a foundation for compile-time evaluation within our IR and serves as a simple abstract machine for compilers. Finally, Chapter 5 summarizes the work and outlines a plan to unify these features into a cohesive compiler infrastructure, which will form the basis for my PhD dissertation.

2 Background on Intermediate Representations

Abstract

This survey investigates a broad spectrum of intermediate representations in compiler design, emphasizing alternatives to standard forms like Abstract Syntax Trees, Control Flow Graphs, and Static Single Assignment, which dominate mainstream imperative languages. To support this exploration, we conducted a systematic review beginning with a curated set of obscure yet foundational seed papers, followed by multi-level citation snowballing using CrossRef, and a hybrid filtering pipeline combining exact keyword and fuzzy phrase matching across abstracts and full texts. Framed around three core research questions: identifying underexplored IRs, evaluating their potential for optimization, and assessing their suitability for code generation. We highlight representations such as Program Dependence Graphs, Program Expression Graphs, Lambda Calculus, and Interaction Nets. These IRs offer insights beyond the dominant Static Single Assignment paradigm.

2.1 Introduction

While studying compiler design, it is important to understand why Intermediate Representations (IRs) matter. Traditionally, if a developer wanted to write a compiler targeting a specific machine architecture, they would create one directly tailored for that combination of source language and hardware. However, this approach quickly becomes inefficient. For each new source language or target architecture, a new compiler had to be developed from scratch for each of the language pairs.

To address this, the introduction of a robust, general-purpose IR significantly reduces the complexity of compiler development. Instead of creating a unique compiler for every language-architecture pair, developers can write one compiler from each source language to the IR, and then separate compilers from the IR to each target architecture. This modular approach drastically reduces the number of compilers that need to be written, from a number equal to the product of the number of source languages and target architectures, to a number equal to the sum of the two. In other words, rather than writing $\mathrm{SourceLanguages} \times \mathrm{TargetArchitectures}$ compilers, one only needs to create $\mathrm{SourceLanguages} + \mathrm{TargetArchitectures}$ compilers.

Beyond simplification, this strategy enables the definition of general optimizations on the IR itself. These optimizations can then be leveraged by all compilers that make use of the IR, leading to more efficient and consistent output across platforms.

In 2013, a survey was conducted on IRs in imperative language compilers [110]. However, with the increasing popularity and influence of functional and logic programming paradigms, there is a growing need to expand this survey to accommodate a broader range of programming paradigms better.

In July 2025, the TIOBE index reported that all ten of the most searched-for programming languages were imperative in nature [119]. A similar trend was observed earlier, in the first quarter of 2024, where the top ten programming languages with the most code pushed to GitHub were also all imperative [86]. Most of these popular imperative languages rely on an IR based on Three-Address Code (3AC) (see Section 2.3.2.2), with the notable exception of the GCC family of languages, which use an Abstract Syntax Tree-like representation (see Section 2.3.1.1). This prevalence indicates that 3AC and Static Single Assignment (see Section 2.3.2.3) forms have

effectively become the dominant IRs in non-academic, production-level programming environments.

Given this mainstream dominance, a central goal of this survey is to explore and highlight IRs that break away from this imperative orientation, particularly those that might offer utility or innovation beyond the prevailing 3AC/SSA paradigm. Towards that end, we propose a set of research questions aimed at better understanding the landscape of IRs, particularly beyond those traditionally used in imperative compilers.

1. What IRs exist beyond the standard Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Static Single Assignment (SSA) forms that are found in almost every industrial compiler? While these conventional IRs have proven effective, they may not fully address the needs of modern programming paradigms, such as functional and logic programming, or support emerging hardware architectures and parallel execution models.
2. Which IRs provide the most support for enabling beneficial optimizations? Such as:
   - Global Value Numbering, which allows for the detection and elimination of redundant computations.
   - Automatic Parallelization, where the IR supports the analysis and transformation of code into forms that can be executed in parallel.
   - Automatic Vectorization, enabling efficient use of SIMD (Single Instruction, Multiple Data) instructions.
   - Dead Code Elimination, a fundamental optimization for removing code that has no impact on program behavior.

3. Which IRs are most amenable to code generation? That is, which IRs retain sufficient semantic detail and structural clarity to support the effective translation into target machine code, preferably across a wide range of architectures?

The following section outlines the methodology used to conduct this review, detailing how we identified and analyzed a wide range of IRs. In Section 2.3, we categorize the IRs we discovered into three distinct groups: Graphical, Linear, and Mathematical. Following this, Section 2.4 examines the identified IRs through the lens of our research questions, evaluating their capabilities in terms of optimization support and code generation potential. Finally, Section 2.5 presents a summary of our key findings and conclusions.

2.2 Methodology

We began our study by manually searching for obscure seed papers relevant to IRs in compiler and systems research. During this process, we identified a preliminary list of DOIs:

- https://doi.org/10.1145/2480741.2480743
- https://doi.org/10.1515/9781400881932
- https://doi.org/10.1145/96709.96718
- https://doi.org/10.1109/CGO51591.2021.9370308
- https://doi.org/10.1006/inco.1994.1013
- https://doi.org/10.1007/978-3-540-24644-2_14
- https://doi.org/10.1016/0743-1066(92)90054-7

Using these documents as seeds, we traced back two levels of references and forward six levels of citations. We limited this snowballing to artifacts with associated

DOIs. This enabled the use of the CrossRef API [27] for automation and simultaneously filtered out most gray literature. The intent of this process was to assemble a broad, yet relevant, candidate pool while remaining focused within the IR domain.

All collected DOIs from the snowballing process were consolidated into a single file. Duplicate entries across sources were removed, resulting in a clean list of unique DOIs, ready for the next phase of analysis. Each of these 1,685,170 DOIs underwent a basic bibliographic validation. Papers were excluded if they fell outside of acceptable parameters, such as:

1. Publication dates earlier than 1910,
2. Undesirable document types (anything other than journal articles, conference papers, book chapters, dissertations, technical reports, or reference entries),
3. Missing or mismatched bibliographic fields,
4. Papers focused on niche IR subfields which do not directly generate executable code, including disassembler-specific IRs [58, 73, 85, 104] and IRs developed primarily for machine learning graphs [30, 75, 105].
5. Papers focusing on IRs designed for non-classical (quantum) computing [15].

This filtering ensured that only academically valid and topically relevant literature would advance to the next stage.

Next, we evaluated the abstracts of the remaining papers using a thematic keyword model tailored to IR-related topics such as Static Single Assignment (SSA), code generation, and optimization. Keywords were grouped into thematic categories based on concepts commonly associated with compiler design and program analysis. Each abstract was tokenized into lowercase words and scanned for exact matches against a set of curated single-word keywords. This step is computationally efficient and effective

for capturing concise technical terms and acronyms. If an abstract did not meet the minimum group-match threshold during this fast check, it proceeded to a second-stage analysis.

The second stage used fuzzy matching via fuzz.partial_ratio from the RapidFuzz library [7]. Here, multi-word keyword phrases were compared against the abstract text, with a similarity threshold of 80% for acceptance. This step allowed for natural linguistic variation, paraphrasing, and formatting differences by identifying terms like “three-address code,” “dead code elimination,” and “program dependence graph.”

The combination of both strategies (exact and fuzzy matching) ensures a balance between speed and semantic depth. Exact matching allows early exits for clearly relevant papers, while fuzzy matching prevents discarding borderline but valuable documents due to superficial differences in phrasing.

For papers passing the abstract check, we conducted full-text analysis. PDF documents were parsed, and their body text was extracted for evaluation using the same thematic keyword framework. This deeper level of filtering is particularly important for papers, such as those on optimization techniques, that introduce IR concepts more in the body, which may not be mentioned in the abstract.

Papers that successfully passed full-text evaluation were compiled for final manual review. Each entry was annotated with summary data, including relevance scores and matched thematic groups. These summaries were presented to human reviewers to ensure traceability, transparency, and accuracy. The goal of this stage was to finalize a high-quality, focused corpus for inclusion in the systematic review.

2.3 Taxonomy

We have categorized the Intermediate Representations (IRs) into three main categories. The first is **Graphical**, which includes all IRs represented as a set of nodes connected by edges. The second is **Linear**, encompassing IRs that can be structured as arrays—this includes strings, which are technically arrays as well—and that closely correspond to either a source or target language. The third category is **Mathematical**, where IRs rely primarily on mathematical substitution to define and execute computations.

2.3.1 Graphical Intermediate Representations

Graphical representations are fundamentally based on core concepts from graph theory. A graph, typically denoted as $G = (V, E)$, consists of two main components: a set of nodes (or vertices), V, and a set of edges (or links), E. Each node in V represents some unit of data—for example, each node could correspond to a specific instruction that a program is meant to execute.

The edges in E are usually represented as pairs (V_1, V_2), where V_1 and V_2 are elements from the set V. These elements are referred to as the endpoints of the edge, and the edge itself signifies a connection between the two corresponding nodes. For instance, one might model the relationship between sequential instructions in a program by connecting an instruction node to the one that follows it via an edge.

A subgraph represents a smaller graph $G' = (V', E')$, where V' is a subset of V and E' is a subset of E. For example, a subgraph of the graph shown in Figure 2.1 could be $G' = (\{a, b, c\}, \{(a, b), (a, c)\})$, which corresponds to the leftmost three nodes and the two edges connecting them, as illustrated in Figure 2.2.

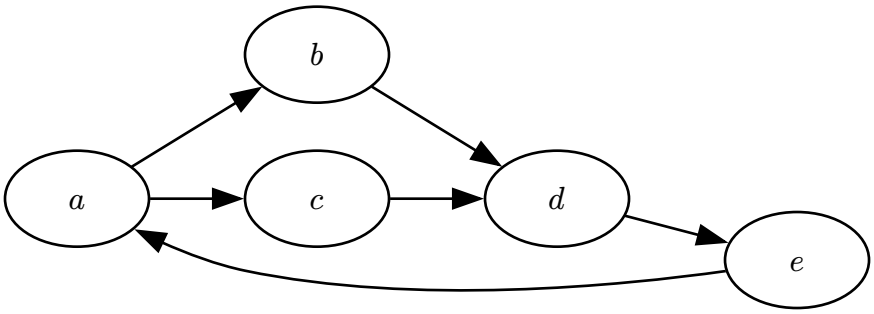

Figure 2.1: A directed graph with arbitrarily named nodes.

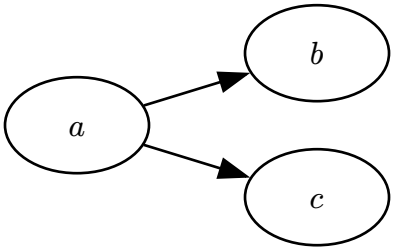

Figure 2.2: One possible subgraph of the graph in Figure 2.1.

One can traverse a graph by selecting a specific starting node and then moving along one of its edges to reach another connected node. When this traversal involves only a single edge, it is referred to as a single-hop traversal.

Graphs can be classified as either directed or undirected. In an undirected graph, the order of the endpoints in an edge is irrelevant, meaning it is possible to traverse from either endpoint to the other without restriction. In contrast, a directed graph features edges as ordered pairs. In this case, the edges (V_1, V_2) and (V_2, V_1) are treated as two distinct edges, where the left endpoint indicates the starting point of a traversal and the right endpoint indicates its destination.

In a directed graph, starting nodes—those that appear on the left side of an edge—are referred to as **predecessors** of the nodes they connect to. Conversely, ending nodes —those on the right side of an edge—are known as **successors** of the nodes from which

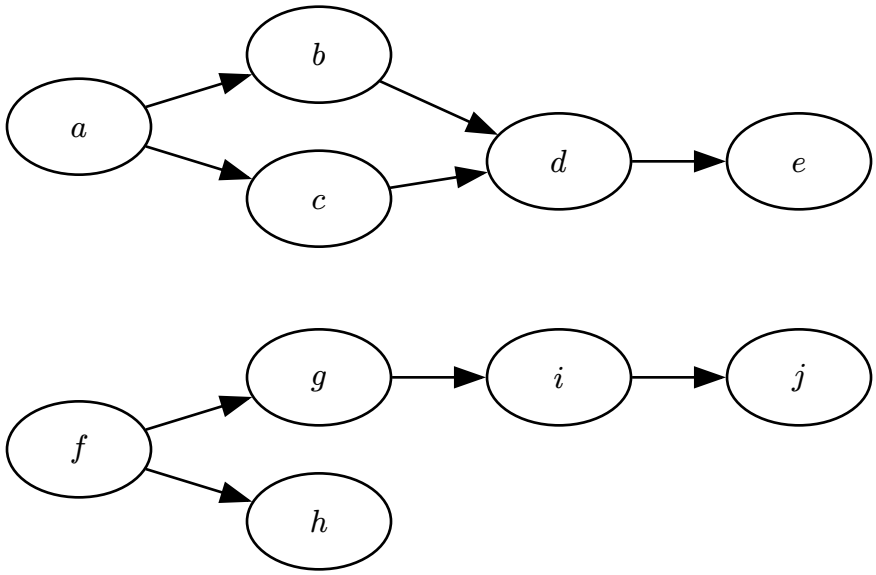

Figure 2.3: A directed acyclic graph (top) and a tree (bottom).

they are reached. For example, in Figure 2.1, nodes `b` and `c` are successors of node `a`, making `a` their predecessor. Similarly, nodes `b` and `c` serve as predecessors to node `d`.

A path represents a sequence of single-hop traversals. For example, in Figure 2.1, the path represented by the edges (a, b), (b, d), (d, e), and (e, a) is a sequence of valid single-hop traversals. A path is only considered valid if all of its edges exist, and in a directed graph, the path must flow from predecessor nodes to successor nodes. Paths are often represented by simply listing the nodes they pass through. Using this notation, the previous path would be written as (a, b, d, e, a).

When the first and last nodes of a path are the same, this particular type of path is called a cycle. If a graph contains no cycles, it is referred to as an acyclic graph. An example of such a graph is shown at the top of Figure 2.3. Furthermore, if an acyclic graph contains no nodes with more than one predecessor, it is referred to as a tree. An example is also presented in Figure 2.3.

According to Johnson [68], Directed Acyclic Graphs and trees have a well-defined starting point, called the root node (such as nodes `a` and `f` in Figure 2.3). A node `a` is said to dominate another node `b` if every path from the root node to `b` passes through `a`. By this definition, a node dominates itself; however, if a node `a` dominates another node `b` where `a` is not equal to `b`, then `a` is said to strictly dominate `b`. Additionally, a node `d` immediately dominates another node `e` (as shown in Figure 2.3) if there is no intermediate node `r` such that `d` strictly dominates `r` and `r` strictly dominates `e` — in other words, there is no node positioned between `d` and `e` in the dominance hierarchy.

2.3.1.1 Abstract Syntax Tree/Parse Tree

To understand a parse tree, one must first grasp a few basic concepts in parsing [1]. Parsing is generally divided into two main phases: lexical analysis and syntactic analysis. In the lexical analysis phase, an input string is segmented into its smallest meaningful units, known as tokens. For example, the string "-A + B" would be broken down into the tokens "-", "A", "+", and "B". These tokens are then passed on to the syntactic analysis phase, where this flat list is organized into a structured form. This structure is typically represented by either a parse tree or an abstract syntax tree.

Parse trees represent the structure of a parse and can be understood as a record of the grammar productions traversed during parsing. However, many nodes in a parse tree (for example, a semicolon at the end of a statement in a C-like language) are irrelevant to the actual meaning of the parse. Abstract Syntax Trees (ASTs) address this by being a reduced subset of the parse tree, with all such irrelevant nodes removed. ASTs can be constructed either by skipping these irrelevant productions during parsing or by pruning them from an existing parse tree. Figure 2.4 illustrates an example of a parse tree alongside two corresponding ASTs.

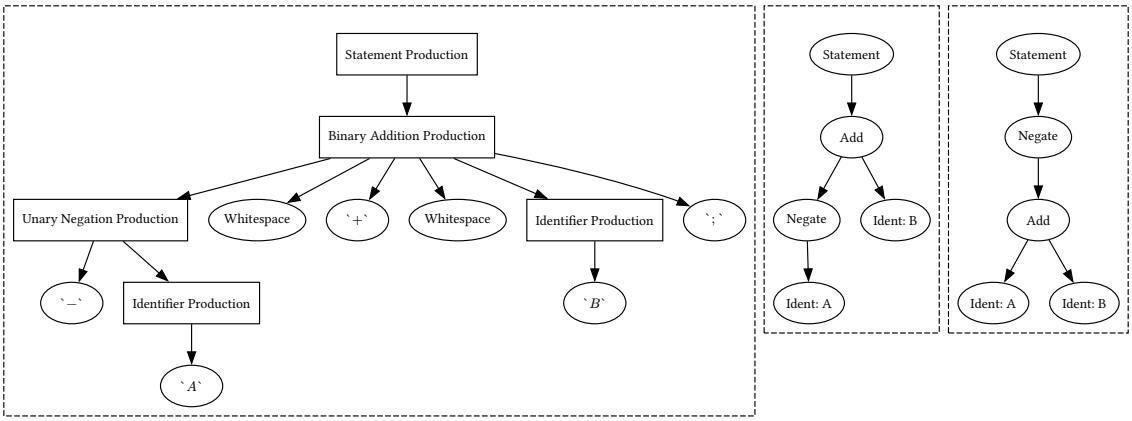

Figure 2.4: A potential parse tree (left), an abstract syntax tree (AST) (center), and a mathematically incorrect AST (right)—resulting from an improper application of operator precedence—are shown for the expression "$-A + B$;" in a hypothetical C-like language. Notice that the parse tree includes more irrelevant details and often fails to convey the meaning of specific nodes. In contrast, AST nodes tend to carry more semantic information while eliminating unnecessary elements. The label "Ident" is a shortened form of "Identifier."

These two closely related representations are among the simplest intermediate representations currently in use. Many straightforward systems that do not employ more complex intermediate forms still rely on one of these trees. Furthermore, any parsing system more advanced than regular expressions will use one of these tree structures, at least implicitly. In general, an AST can be described as a tree structure where each node represents an operator or a programming construct. The children of each node correspond to the operands of that operator or the components of the programming construct. This hierarchical organization of nodes reflects the structure of the expression being represented. For example, the two different interpretations of the expression $-A + B$ (namely $(-A) + B$ and $-(A + B)$) are encoded as two distinct trees, as illustrated on the right side of Figure 2.4.

One limitation of parse trees is that a single grammar can produce more than one parse tree for the same string of tokens. This means there may be multiple possible

parse trees for a given input, making it challenging to determine which parse tree is the correct one. Additionally, common subexpressions—such as those found in the expression `(a + b) * -(a + b)`—are often processed separately in parse trees. This can lead to redundant work, potentially wasting memory and execution time either during program runtime or in later optimization stages aimed at eliminating such redundancy.

The GCC [51] family of compilers employs a two-level IR system consisting of **GENERIC** and **GIMPLE**, both of which are similar to an AST [87].

GENERIC represents functions as abstract syntax trees in a language-independent manner. This means that code from any supported language is first translated into a standardized tree form, allowing the compiler to process it uniformly regardless of its source language.

GIMPLE is a simplified subset of GENERIC, explicitly designed for optimization. It retains the structure of the original parse tree but transforms complex expressions into a normalized three-address code form. In this form, intermediate values are assigned to temporary variables, making the code easier for optimization passes to analyze and manipulate.

Compiler front-ends may still use language-specific constructs in their GENERIC representations, provided they implement hooks to convert those constructs into GIMPLE. The transformation from GENERIC to GIMPLE is handled by a dedicated compiler pass known as the gimplifier. This pass recursively breaks down complex statements into sequences of simpler ones, preparing the IR for subsequent optimizations.

GIMPLE plays a foundational role in enabling advanced optimizations within GCC, including polyhedral optimization and interprocedural analyses on function-local SSA

(Static Single Assignment) form. One example of such usage is detailed in Jan Hubička's work on interprocedural optimization in GCC [51].

2.3.1.2 Directed Acyclic Graphs

Directed Acyclic Graphs (DAGs) were developed as a solution to the problem of identifying and eliminating common subexpressions in programs [22]. In expressions where there is no assignment or other state-manipulating operation, any textually identical subexpressions are guaranteed to yield the same result (at least in many older languages). This property allows a compiler to collapse such duplicate subexpressions into a single node in the graph, thereby avoiding redundant computation.

This transformation can be illustrated through the contrast between an AST and a DAG. In an AST, the subexpression (A + B) is represented multiple times if it appears more than once in the source code. However, in a DAG, such repeated subexpressions are represented by a single node, as shown in Figure 2.5. The figure presents both an AST and a corresponding DAG for the expression $(A + B) * -(A + B)$. In the AST (left), the expression $A + B$ appears twice, while in the DAG (right), it appears only once. This consolidation enables the compiler to evaluate the subexpression a single time and reuse the result, provided the values of A and B do not change between uses.

Formally, DAGs are structurally similar to ASTs, but they differ in that they collapse equivalent subgraphs into a single representative node. This transformation preserves semantic correctness under the assumption of referential transparency—i.e., that the values involved in the subexpression are not modified between evaluations.

DAGs are typically constructed using the same parsing techniques employed for building ASTs. However, during DAG construction, each time a new node is to be added, the compiler first searches the existing graph for an equivalent node. If a match

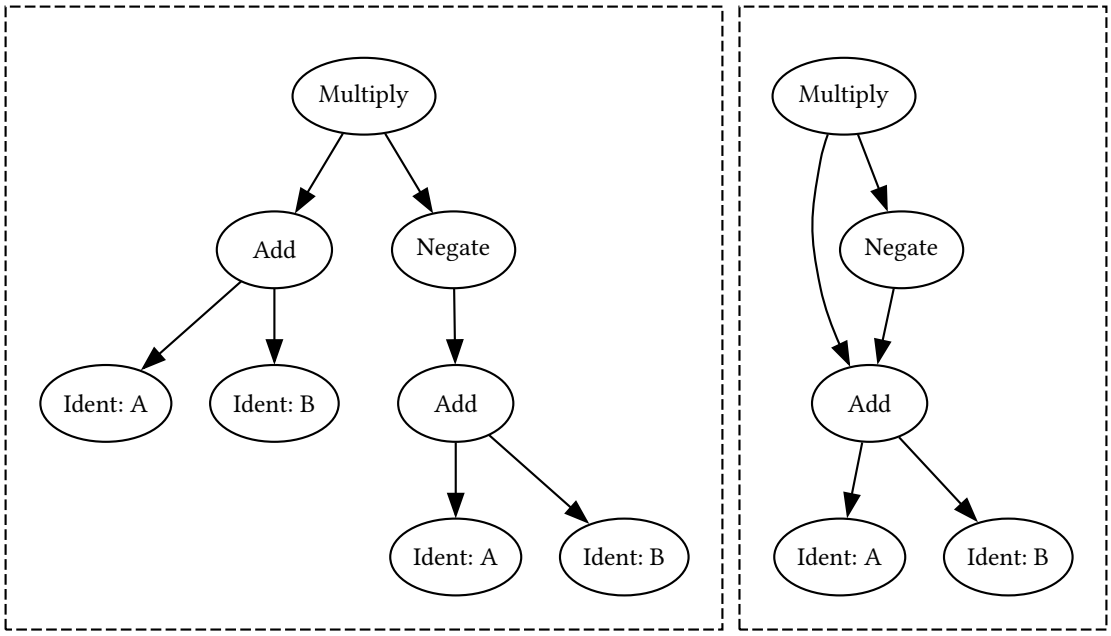

Figure 2.5: A potential AST (left) and DAG (right) for the expression "$(A + B) * -(A + B)$" in a hypothetical C-like language. Notice how the nodes for "A + B" only appear once in the DAG, whilst appearing twice in the AST.

is found, that existing node is reused rather than creating a new one. This process ensures that identical subexpressions are not redundantly represented.

Despite their utility, DAGs have limitations. Care must be taken to ensure that the variables involved in collapsed subgraphs remain unchanged between uses. If the value of a variable changes, say in another thread, collapsing expressions that reference it could result in incorrect behavior.

DAGs find practical application in several compilers. For instance, the *lcc* compiler [49] uses a DAG-based intermediate representation. Rather than constructing the entire DAG upfront, *lcc* incrementally generates only the required fragments during parsing, processes them, and then discards them before continuing with the rest of the program [50]. Additionally, DAGs are instrumental in detecting and removing redundant computations, making them a valuable tool in optimizing compilers [22].

2.3.1.3 Control Flow Graph

The primary purpose of Control Flow Graphs (CFGs) is to facilitate the analysis of a program's execution flow by explicitly representing flow relationships between different parts of the code. They help answer important questions such as: "Is this an inner loop?", "If an expression is removed from a loop, where can it be correctly and profitably placed?", or "Which variable definitions can affect this use?" [2]. Control flow graphs were developed to provide a visual and structural representation of the program's control flow, enabling easier analysis and understanding of the program's runtime behavior.

To construct a CFG, one begins by identifying Basic Blocks: the foundational units of the graph. The Dragon Book [1] defines a basic block as a sequence of instructions with a single entry point and no branching, except at the end. Identifying basic blocks begins with identifying leaders: special instructions that mark the beginning of a new block. Leaders are instructions that meet one of the following criteria:

- They are the first instruction of the program.
- They are the target of a conditional or unconditional jump, or an exception-handling instruction [4].
- They immediately follow a conditional or unconditional jump or an exception throw.

For example, in the code snippet enumerated in Listing 2.1, the statements on lines 2, 4, 5, 7, 8, 10, 12, and 15 are leaders. A basic block consists of a leader and all subsequent instructions up to, but not including, the next leader.

```
try {
  a = b + c;
  x = 0;
  while(x < a)
    x = a * d;

  if(x == y)
    z = e;
  else
    throw

exception();

  y = z + 1;
  return y;
} catch(exception) {
  return 0;
}
```

Listing 2.1: Some example code written in a hypothetical C(++)-like language containing a while loop, if statement, and throw-try-catch trifecta. Assume all variables are integers that have been previously defined.

Once the basic blocks have been identified, each block becomes a node in the control flow graph. Edges are then added to represent possible transitions between blocks. There are typically three types of edges:

1. An edge from a node to itself, representing loops where the program control returns to the same block.
2. An edge from one node to another, representing conditional or unconditional jumps.
3. An edge from a node to an exit node, indicating the end of execution paths.

A visual example of a full CFG constructed from the code in Listing 2.1 is shown in Figure 2.6.

Despite their usefulness, CFGs have several limitations. One major drawback is their ignorance of data dependencies. While CFGs effectively capture control dependencies, they do not represent data dependencies. This can lead to incorrect

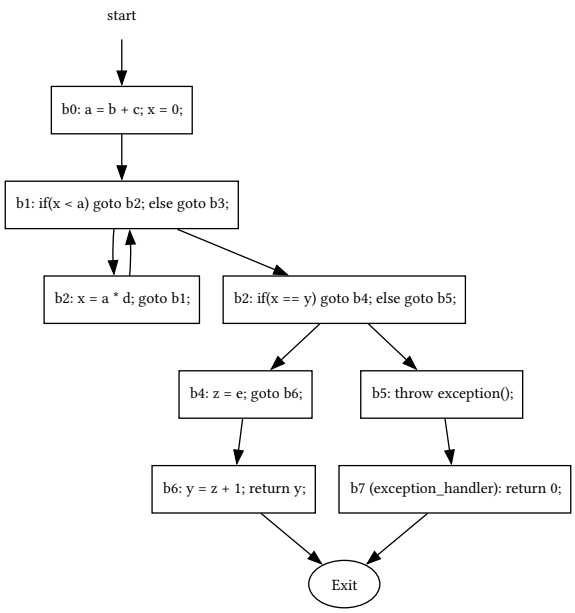

Figure 2.6: The Control Flow Graph, which is derived from the code shown in Listing 2.1, is not minimal. Certain Basic Blocks—specifically those labeled “b4” and “b6”—could potentially be merged through further optimization; the same applies to Basic Blocks “b5” and “b7”. However, the optimizations required to enable these combinations are not universally applicable. For example, merging “b6” with another block would require moving lines 12 and 13 inside the conditional statement on line 7 and eliminating the exception handling mechanism. These changes are not always valid. In particular, if the else branch assigned a value to the variable “z” instead of throwing an exception, Basic Block “b6” would need to remain separate.

results—particularly when one statement depends on the result of another that is not directly connected in the graph.

Another limitation is scalability. Constructing CFGs for large or complex programs can be computationally expensive, and as program size grows, these graphs may become unwieldy and difficult to manage effectively.

CFGs also tend to be overinclusive. They often represent both relevant and irrelevant dependencies. For instance, statements located within the same loop may appear connected in the graph even if they do not depend on each other in practice. On the flip side, CFGs lack semantic context. While they show the paths that control might

follow during execution, they do not capture the actual meaning or runtime behavior of the code.

Despite these limitations, CFGs are practically valuable in specific contexts. One prominent example is in the construction of Static Single Assignment form (see Section 2.3.2.3). In this setting, CFGs help in placing φ-functions correctly and enable efficient analysis of definition-use chains.

2.3.1.4 Dataflow Graph

Dataflow Graphs (DFGs) were initially developed as a strategy to extract greater performance from "supercomputers" with multiple processors [40]—what we now consider standard, consumer-grade CPUs. The key design principle behind Dataflow is its ability to more naturally express concurrency, allowing multiple operations to proceed simultaneously without requiring explicit coordination by the programmer.

In modern computing, DFGs are foundational to the execution models of many machine learning runtimes, especially those compatible with the ONNX standard [94]. These graphs help maximize concurrency during model execution, making them well-suited to the parallel processing requirements of modern ML workloads.

Structurally, Dataflow Graphs closely resemble DAGs, but with a crucial difference in interpretation. In a DFG, tokens—which represent values—are added to the edges of the graph. An operation (node) can only execute once a token is present at each of its input edges. Upon execution, the node produces a new token at each of its output edges. This "trickling" of tokens through the graph is analogous to the way electricity flows through a circuit or water through a plumbing system.

Some implementations of Dataflow Graphs refine the structure of nodes by using Activity Templates [41]. In this representation, each node is a tuple consisting of an operation code and an array of outputs. Each output is itself a tuple indicating the destination Activity Template and the index of the input to which the output should be wired.

Not all programming constructs translate cleanly into the Dataflow paradigm. In particular, global variables and unstructured control flow (such as goto and switch statements) are incompatible and must be eliminated during preprocessing [45, 101].

Nonetheless, structured control flow can be supported using special node types. For example, Figure 2.7 shows a Dataflow Graph for the expression "while (X > 0) X -= 3". This requires two additional constructs:

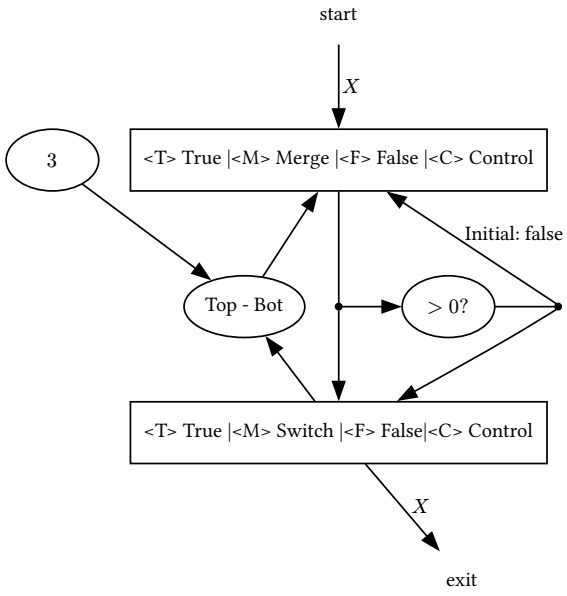

Figure 2.7: A Dataflow Graph for the expression while(X $>$ 0) X -= 3″. Two special types of nodes are required for control flow: Merge nodes, which forward the token along from one of their inputs depending on whether a control parameter is true or false, and switch nodes, which forward their input token to one of their outputs depending on whether their control parameter is true or false.

- Merge nodes, which forward a token from one of their inputs based on a control condition.
- Switch nodes, which direct an input token to one of multiple outputs depending on a control value.

One consequence of the Dataflow model is that every part of a program—including control flow—must be expressible as a value-producing expression. As a result, statements that do not yield values must either be transformed or eliminated, often requiring pre-compilation or rewriting of the source program.

2.3.1.4.1 Value Dependence Graph

One notable implementation of dataflow is the Value Dependence Graph (VDG) [128]. VDGs were developed to address the limitations of traditional dependence analysis techniques, particularly when analyzing complex, data-dependent programs. These traditional techniques often struggled to capture the nuances required for effective parallelization and optimization. In contrast, VDGs provide a more accurate and flexible representation of data dependencies, leading to improved program performance.

Structurally, VDGs are similar to DFGs but introduce a key distinction: they are bipartite graphs, in which every edge connects a node (representing primitive operations, control structures like γ-nodes for conditional branches, function calls, and λ-nodes for closures) to a port (representing values). This bipartite design allows for more granular dependency tracking between computations and values. VDGs can be seen as an extension of the intermediate representation (IR) used by the Fuse partial evaluator [127], expanded to support imperative features such as store and load

operations for variable updates. Notably, because of the IR's functional roots, traditional loops are transformed into tail-recursive function calls.

The construction of a VDG begins with a CFG (see Section 2.3.1.3), which is then transformed by replacing control flow edges with γ- and λ-nodes. The contents of each basic block are expanded into corresponding VDG nodes using a process referred to as symbolic execution.

Despite their strengths, VDGs are not without limitations. In particular, they do not handle loop and function termination dependencies elegantly or comprehensively, which can affect their applicability in specific analyses or optimizations.

The Value State Dependence Graph (VSDG) [66] was developed to address several limitations found in the VDG; One of the main being VDG's difficulty in preserving a program's terminating properties and in supporting target code generation. The standard process—converting a VDG into a demand-based Program Dependence Graph (see Section 2.3.1.5.1), then into a traditional CFG, and finally generating code, which can be both complex and indirect.

Formally, the VSDG can be understood as an extension of the VDG, augmented with state-dependence edges that explicitly model sequential computation. These edges ensure that execution order is respected, making the graph suitable for representing imperative constructs. Significantly, state-dependence edges can be added incrementally until the VSDG corresponds to a unique CFG, which helps in maintaining control flow integrity throughout compilation or analysis.

The construction of the VSDG often begins with structural analysis [108], a form of interval analysis that transforms irreducible control flow into reducible forms. This

transformation is essential because many compiler optimizations and code generation techniques work more effectively on reducible control flow [109].

VSDGs have a wide range of applications: One key use is in simultaneous register allocation and code motion [66]. Additionally, VSDGs contribute to code size reduction, particularly by utilizing multiple memory instructions [67]. Despite these strengths, VSDGs have some limitations, including the complexity involved in correctly constructing state-dependence edges and the challenges they pose for visualization and reasoning compared to more traditional representations like CFGs.

2.3.1.4.2 Program Expression Graph

A Program Expression Graph (PEG) [118] is a dataflow-based representation of a program's expression-level structure, where nodes correspond to operations, such as addition or subtraction, or data values like variables or constants, and edges represent the flow of data between these nodes. PEGs are constructed by grouping parts of the CFG (see Section 2.3.1.3) into PEG-like sub-CFGs based on special nodes called Φ-nodes (which represent branching conditions) and θ-nodes (which represent loop conditions). These sub-CFGs are then recursively converted into PEGs; this construction process includes identifying equal conditions for branch fusion and loop fusion, making the approach straightforward to implement. After optimization, a similar reverse process is used to reconstruct the CFG from the PEG.

An extension of this model, called the Equality PEG (E-PEG), groups nodes into equivalence classes. Unlike traditional PEGs, E-PEGs support equality-based reasoning through a system of trigger patterns and callbacks managed by a "saturation engine" that dynamically monitors and applies optimizations. This mechanism allows the E-PEG to simultaneously explore and represent multiple optimization paths as a single

“blob” of programs. All optimizations can be applied to this “blob,” after which the best program is extracted using a global heuristic. This approach enables compilers to make more globally informed decisions, such as selectively inlining functions based on broader optimization effects.

PEGs and E-PEGs have been used for applications like translation validation [113, 120], providing formal assurance that program transformations preserve correctness.

2.3.1.5 Control Flow/Dataflow Hybrids

Control flow and dataflow analyses are both instrumental when applying optimizations to a program. Consequently, the majority of Graphical IRs combine these two techniques to leverage their combined strengths. One such IR that exemplifies this combination is the Value State Dependence Graph, as previously presented in Section 2.3.1.4.1.

2.3.1.5.1 Program Dependence Graph

A Program Dependence Graph (PDG) [47] integrates both control flow and data dependencies into a single graph, providing a comprehensive representation of the dependencies between variables in a program. The PDG was initially developed to serve as a helpful program representation for optimizing compilers targeting vector or parallel machines. Such compilers need to perform not only traditional optimizations but also transformations to detect and exploit parallelism effectively.

There are two main types of control dependencies in a PDG: First, data dependence occurs when the value computed by one statement depends on the value produced by another. Second, control dependence exists between a statement and the predicate that directly controls its execution. For instance, in the conditional statement `if(A) B = C`

`* D;`, the execution of $B = C * D$ is controlled by the outcome of the condition A. Ferrante, Ottenstein, and Warren [47] describe several variations of the graph. Nodes within the PDG can represent entire statements or predicate expressions, or more granularly, individual operators and operands. The authors also discuss a more functional style of PDG, which they refer to as a Dataflow Graph augmented with a few additional links.

The construction process begins with an existing CFG (see Section 2.3.1.3). For handling data dependencies, the original approach assumes each basic block within the CFG is represented as a DAG (see Section 2.3.1.2). The PDG is then built by identifying common subsets of control dependencies, which are factored out by introducing region nodes. Specifically, for each non-region node with multiple control dependence predecessors, a region node is created representing this set of predecessors. All nodes sharing this same set of control dependence predecessors are then linked to the region node, which replaces the original multiple dependencies as their single control dependence (which can run in linear time [70]).

Next, the leaf nodes of the DAGs corresponding to each basic block are marked as "merge" nodes. Initially, every variable in the program is assigned the value "undefined." The algorithm determines the set of assignments or initializations—commonly known as Reaching Definitions—that may affect a variable's value at any program point. Edges are added between these definitions and the merge nodes, making explicit the chains of usage-to-definition relationships in the graph. Input/output operations are treated as operations on an implicit file object to maintain correct sequencing. Subscripted array accesses are represented using a "select" operator with inputs for the array and offset, and one output for the selected element. Similarly,

subscripted array assignments use an “update” operator with inputs for the array, offset, and replacement value, producing a modified array output. Loops are represented as single operators with operands for initial, final, and increment values, producing two outputs: an index value stream and a predicate value stream. Loop variables are remapped to run from 1 to N in steps of 1.

Ferrante, Ottenstein, and Warren [47] note that the PDG representation may require roughly 50% more memory than simpler linear representations. However, with modern hardware, this overhead is often acceptable rather than a strict limitation. Data dependence is most straightforward to determine when pointers, shared variables, or procedure calls with non-pass-by-value semantics cause no side effects. Aliasing (when two array-like structures refer to overlapping memory) and side effects introduce significant challenges in accurately modeling dependencies in the PDG. To handle implicit aliasing arising from procedure parameter bindings, interprocedural dataflow analysis must be performed. This analysis is also used to detect side effects. In languages like C, pointers can complicate or even preclude PDG construction because pointers may reference arbitrary memory locations. However, principled use of pointers in well-structured C programs may still allow for practical PDG analysis.

PDGs have also proven valuable in code generation [112] and vectorization [9, 106]. The process often involves linearizing the PDG and then interpreting or scheduling its operations to generate executable code [46].

2.3.1.5.1.1 Click’s IR

One notable implementation influenced by PDG concepts is Click’s IR [19]. This graph-based IR is designed with simple semantics and implemented in a memory-efficient C++ style. It employs a directed graph structure in which vertices—labeled

with opcodes—have ordered inputs and unordered outputs, while edges remain unlabeled. While the authors acknowledge the similarity to PDGs, a key point of divergence lies in their use of region nodes. These nodes consolidate control values from predecessor blocks into a single output, effectively acting as switches to manage control flow and replace conventional basic blocks. Additionally, the IR follows Single Static Assignment form (see Section 2.3.2.3), with virtual registers defined accordingly. Primitive nodes such as `IF` and `PHI` carry control inputs that explicitly indicate their basic block membership. Compared to the PDG, this IR is more compact, imposes fewer restrictions on evaluation order, and includes just enough control information at merge points to support executable semantics.

2.3.1.5.1.2 Program Dependence Web

The Program Dependence Web (PDW) [97] was developed to enable flexible interpretations of programs in control-driven, data-driven, and demand-driven styles. By integrating both control and data dependencies, the PDW allows imperative programs to be mapped onto dataflow architectures and demand-driven graph reducers. It also supports specific optimizations more effectively than SSA-form, as it incorporates control dependence information absent in SSA. This makes the PDW particularly suitable for converting traditional imperative code into representations aligned with modern computational paradigms.

A PDW is constructed by augmenting a Program Dependence Graph (PDG) with elements of SSA, beginning with the replacement of merge nodes with φ-functions and renaming variables to enforce single assignment. These φ-functions are classified into gating functions: γ (if-then-else), μ (loop iteration), and η (loop result). Switch nodes are then introduced to manage value flow into control regions, enabling data-driven

interpretation. Switch placement is guided by a breadth-first traversal of the PDG's control dependence subgraph, ensuring definitions entering a region are properly gated. Data operator nodes spanning regions receive additional switch networks representing conditional execution paths.

The PDW enables multiple execution modes: control-driven using original PDG control edges, data-driven via switches, and demand-driven through gating functions. In dataflow extraction, control dependencies and γ-functions are removed after being subsumed by switches. Demand-driven interpretation works in reverse—starting at outputs and querying inputs—allowing gating functions to select inputs based on predicate evaluation dynamically. Despite its versatility, the PDW's complexity and construction overhead, with an $O(N^3)$ time cost, limit its general applicability [110].

2.3.1.5.2 Dependence Flow Graph

The development of the Dependence Flow Graph (DFG) [99] was motivated by certain limitations encountered when using traditional dataflow graphs. In these graphs, information propagates throughout the entire graph without regard for the control flow, rather than being restricted to where the information is actually needed for optimization. This leads to inefficiencies. For instance, when a part of the graph is updated at some point in a program, the entire control flow graph below that point (or above it, in backward analyses) may need to be reanalyzed, even if only a few points within that region are affected by the update.

Def-use chains were introduced as a partial solution to these problems. By allowing information to flow directly between definitions and their uses without passing through unrelated statements, def-use chains improve the flow of information. However, def-use chains come with their own set of drawbacks. Firstly, they are not suitable for backward

dataflow problems, such as the elimination of redundant computations, because they lack sufficient information about the program's control structure. Secondly, the absence of control flow information affects the precision of analysis even in forward dataflow problems, like constant propagation. Although def-use chains can become large, the adoption of Static Single Assignment (SSA) form helps to alleviate this issue [69].

The DFG addresses these shortcomings by providing more precise structural properties, which can be leveraged in formal correctness proofs. Furthermore, the DFG algorithm is more efficient; it performs work only for relevant dependencies at each node, resulting in an asymptotic complexity of O(EV), where E and V represent edges and vertices, respectively. This contrasts with the PDG algorithm, which performs O(V) work each time a node is processed.

Formally, DFGs extend a standard dataflow graph by incorporating a concept of "global" storage, along with load and store operations that manipulate this storage. These operations emit a secondary "imperative" token in addition to their computed value, signaling that their execution has completed. Control flow is implemented through special switch and merge nodes. A switch node takes an imperative token and a boolean input, producing an imperative token along one of two outputs depending on the boolean value. A merge node takes two input imperative tokens but does not require both to be ready before forwarding a token, effectively merging two possible execution paths into one.

While loops can theoretically be represented using recursion, for optimization purposes, the authors introduce special loop nodes—loop nodes, which behave like merges, and until nodes, which behave like switches. It is also notable that the semantics of a standard dataflow graph are inverted in the DFG: edges are viewed as

single assignment registers, and producing a token on an edge is analogous to storing a value in the corresponding register.

2.3.2 Linear Intermediate Representations

Linear IRs are typically represented as some form of linear array. This array can take various forms: Sometimes it consists of tokens, other times it comprises tuples that encapsulate more abstract concepts, and occasionally it is represented as a textual language (a primary example being Assembly languages, which act as a human-readable intermediary to machine code). Regardless of the specific format, linear IRs can be read sequentially from the beginning, stepping through one element at a time—much like the source languages from which they are derived. However, similar to those source languages, it is entirely possible for a computer executing a linear IR to deviate from a strictly linear progression. Jump and branch operations, which mimic those found in assembly languages, are common in this type of IR. Additionally, it is not unusual for nodes within graphical IRs to be represented as some form of linear IR; For example, the Basic Blocks in a Control Flow Graph (see Section 2.3.1.3) serve as a prime example of this practice.

2.3.2.1 Polish Notation

Polish Notation [1], one of the earliest forms of intermediate representations in computing, was not initially designed to serve that role. Instead, it emerged as a way to eliminate the ambiguity often found in standard mathematical notation. For example, the expression $2x/3y - 1$ is ambiguous without parentheses—depending on how it is interpreted, it can yield vastly different results [74]. When $x = 5$ and $y = 6$, some might calculate the result as 19, while others arrive at $-0.\overline{44}$. This discrepancy arises from two possible interpretations: either $((2x)/3)y - 1$ or $(2x)/(3y) - 1$.

Reverse Polish Notation (RPN), also known as postfix notation (in contrast to the original prefix Polish Notation), addresses this issue by eliminating the need for parentheses. It does so by placing operators after their operands. The two aforementioned expressions, when written in RPN, become:

- $2\ x\ *\ 3\ /\ y\ *\ 1\ -$
- $2\ x\ *\ 3\ y\ *\ /\ 1\ -$

This postfix structure not only resolves ambiguity but also aligns neatly with the operations performed by stack-based computers. In such systems, operands are pushed onto a stack, and operations are performed by popping values off the stack and pushing the result back on. RPN mirrors this computational model perfectly.

RPN is straightforward to derive from an AST (see Section 2.3.2.1); by performing a post-order traversal of the AST—visiting the children nodes before the parent node—the corresponding RPN expression is naturally produced.

Despite its advantages, Polish Notation has its shortcomings. Neither prefix nor postfix variants support control flow constructs such as loops or conditionals. To address this, Extended Polish Notation [111] was introduced, incorporating control structures and making it more suitable for complex program representation.

However, as computing evolved, stack-based architectures gave way to register-based machines, leading to a decline in the use of Polish Notation in modern compilers targeting native machine code. Nevertheless, more sophisticated descendants of Polish Notation, often referred to as bytecodes, continue to be widely used in interpreted environments. A prominent example is the Java Virtual Machine's Bytecode [55].

2.3.2.2 Three Address Codes

How can we capture the elegance and simplicity of Polish Notation in the context of a register-based computing model? Three Address Codes (3ACs) [1] define computations as a linear sequence of 4-tuples. Each tuple consists of an operation followed by three elements that typically refer to registers, memory addresses, or immediate values. This structure is well-suited to expressing computations in a way that's both readable and amenable to translation into low-level code.

For example, consider the following translation of an expression originally written in Polish Notation. Using a naïve approach, we might represent it in 3AC as:

```
(load, r0, 2, ∅)
(load, r1, x, ∅)
(*, r2, r0, r1)
(load, r3, 3, ∅)
(/, r4, r2, r3)
(load, r5, y, ∅)
(*, r6, r4, r5)
(load, r7, 1, ∅)
(-, r8, r6, r7)
```

This representation is quite close in structure to what one might write in an actual assembly language, and this resemblance is one of 3AC's key advantages. It bridges high-level structure and low-level implementability, making it ideal for compiler backends.

3AC representations are typically generated by performing a linear inorder traversal of an AST (see Section 2.3.1.1). This process naturally produces the required sequence of instructions in a way that respects the structure of the original expression.

However, this initial version is not particularly efficient in its use of registers. With a small quantity of care, we can rewrite the same computation to reuse registers and reduce the total number needed:

```
(load, r0, 2, ∅)
(load, r1, x, ∅)
(*, r0, r0, r1)
(load, r1, 3, ∅)
(load, r2, y, ∅)
(*, r1, r1, r2)
(/, r0, r0, r1)
(load, r1, 1, ∅)
(-, r0, r0, r1)
```

This optimized form produces the same result but uses fewer registers by overwriting temporary values as soon as they are no longer needed. An intriguing question arises: can we automatically transform a less efficient form of 3AC (as shown in the first example) into a more efficient version (as shown in the second)? Single Static Assignment form.

2.3.2.3 Single Static Assignment Form

Static Single Assignment (SSA) form [32] was developed to provide a more efficient and optimized representation of programs; by imposing a more structured form on a program's variables and assignments, SSA enables the use of advanced analysis and transformation techniques that significantly improve the quality of the generated object code.

In SSA form, each variable is assigned exactly once. To achieve this, each occurrence of a variable from the original program is replaced with a new, uniquely

named version (e.g., $V_1, V_2, ...$). This single-assignment rule simplifies data flow analysis and facilitates various optimizations.

However, complications arise when control flow diverges and then rejoins—such as in `if` statements or loops. If different versions of a variable are assigned in each branch, the program must reconcile which version to use after the branches rejoin. SSA handles this by inserting φ-functions, which act as special assignments that choose between different variable versions depending on the control path taken. For example, a phi-function might look like: $V_3 = \varphi(V_1, V_2)$, indicating that V_3 should take the value of V_1 if control came from one path, and V_2 otherwise.

The translation of a program into SSA form requires the prior construction of a CFG (see Section 2.3.1.3), which captures how execution paths in the program split and merge. A basic method for constructing SSA form involves two primary steps. First, trivial φ-functions are inserted at join points within the CFG. These functions have the form $V_i = \varphi\big(V_j, V_k, ...\big)$ and are used to merge different variable definitions coming from multiple control paths. Second, all variables are renamed so that each assignment and use refers to a uniquely identified version. This step ensures that every occurrence of a variable—whether in an expression or an assignment—is replaced with a distinct version V_i, corresponding to a specific definition.

However, this straightforward method often introduces more φ-functions than necessary, which can hinder the effectiveness of later optimization stages. A more sophisticated approach leverages the concept of dominance in the CFG [31]. Using dominance frontiers, the translation to minimal SSA form is performed in three steps. First, the dominance frontier mapping of the control flow graph is constructed. Next, this mapping is used to accurately determine where φ-functions are actually needed for

each variable. Finally, variable renaming is carried out, where each use of a variable is replaced with the appropriate version V_i, ensuring that every assignment is unique and each use refers to the correct definition.

Despite its advantages, SSA form has limitations. In the worst-case scenario, computing dominance frontiers can take $O(R^2)$ time, and the complete conversion to SSA form may take $O(R^3)$, where R is the maximum among the number of nodes, edges, variable assignments, or variable mentions in the program. Moreover, SSA form cannot be directly executed or interpreted. Programs in SSA form must be destructured —converted back to a form where multiple assignments to the same variable are permitted—before they can be run [12]

Despite its limitations, the SSA form has proven to be highly valuable in enabling a range of powerful compiler optimizations and analyses. One such optimization is code motion, which improves program efficiency by relocating computations to more optimal positions within the code [32]. It has also been used to facilitate automatic parallelization [114]. SSA also facilitates the detection of program equivalence, an important capability for both optimization and program verification [131].

Another significant benefit of SSA is its support for the elimination of partial redundancies, where redundant computations that are not present on all execution paths are identified and removed, streamlining code execution [102]. Moreover, SSA enhances constant propagation, which involves replacing variables with known constant values throughout the code, improving execution speed and reducing runtime complexity [126].

2.3.2.3.1 SSA Graph

The SSA Graph [53, 130] extends the traditional SSA form to a graph structure where edges flow from each operand of an operation to its source operation. This representation has been used in several important compiler optimizations, including instruction selection [42, 107], operator strength reduction (reformulating certain costly computations in terms of less expensive ones) [23], and rematerialization (recognizing when it is cheaper to recompute a value that does not fit into physical registers than to store and reload it) [13]. For example, instruction selection benefits from this approach by leveraging the detailed operand-to-operation connections to improve code generation efficiency.

2.3.2.3.2 Gated Single Assignment

Another notable SSA variant is the Gated Single Assignment (GSA) form [97]; GSA modifies SSA by replacing the classic φ-functions with gating functions, which enables direct interpretation of program dependence graphs. This form introduces specialized functions: the γ-function, which attaches a condition along with the two arguments of a φ-function to represent the end of if-else branches; the μ-function, appearing at loop headers to select between initial and loop-carried values; and the η-function, which determines a variable's value at the end of a loop.

The construction of GSA begins by first building the SSA form [121]. Then, the control dependencies of definitions reaching a φ-function are gathered and transformed into gating functions. An improved algorithm constructs these gating functions from scratch by building gating path expressions [116] for each node in the CFG (see Section 2.3.1.3). This approach treats any path in the CFG as a string of edges, using path expressions—simple regular expressions over edges—to represent these

paths. Both this new algorithm and earlier ones traverse the CFG to identify gating conditions for each reaching definition. However, the original algorithms have a time complexity of $O(E \times N)$, where E is the number of edges and N is the number of nodes. In contrast, the newer algorithm aims to be more efficient, approaching linear time. It achieves this by using path compression [117], which flattens the traversal trees to speed up subsequent operations.

GSA serves multiple purposes: it acts as an intermediate step in constructing PDGs (see Section 2.3.1.5.1), offers a more compact form for value numbering [60], and is also used to validate LLVM compilation passes [120]. Despite its advantages, GSA introduces complexity by requiring three types of φ-like functions, which may be seen as a limitation depending on the context.

2.3.2.3.3 LLVM

The Low Level Virtual Machine (LLVM) [79], provides the infrastructure most modern programming languages are built on, including:

- Python https://github.com/exaloop/codon https://github.com/numba/numba https://www.modular.com/max/mojo
- (Objective)C(++) https://clang.llvm.org/
- Fortran https://flang.llvm.org/docs/
- D https://github.com/ldc-developers/ldc
- Swift https://developer.apple.com/swift/
- GHC Haskell https://www.haskell.org/ghc/
- Rust https://www.rust-lang.org/
- Zig https://ziglang.org/
- Julia https://julialang.org/

- Solidity https://github.com/hyperledger/solang

LLVM is currently used as the intermediate representation (IR) for at least five of the ten most popular programming languages, demonstrating its centrality to contemporary language development. Beyond language implementation, LLVM also plays a crucial role in advanced compiler optimization techniques, including polyhedral optimizations [56], making it a powerful tool not only for generating efficient machine code but also for enabling high-level program analysis and transformation.

2.3.2.4 MLIR

Multi-Level Intermediate Representation (MLIR) [80] was created in response to the recognition that modern machine learning frameworks consist of numerous compilers, graph technologies, and runtime systems that lacked a shared infrastructure or consistent design principles. This fragmentation resulted in poor error abstraction and made it challenging to work effectively across different application domains, hardware targets, and execution environments. However, unlike most other eliminated machine learning IRs, the creators of MLIR sought to develop a generalized infrastructure that would lower the cost of building compilers while supporting a wide variety of domain-specific use cases.

MLIR is hosted in the same GitHub repository as LLVM, prompting a natural question about how it differs from LLVM itself. The key distinction lies in their approaches to modeling data structures and algorithms. While LLVM primarily focuses on scalar optimizations and homogeneous compilation, MLIR aims to represent a rich set of data structures and algorithms as first-class values and operations. It is designed to be highly customizable, allowing new operations to be added over time.

MLIR's core features include several important components. Types in MLIR are structured, multi-dimensional, and designed to access memory in a way that is injective by construction, helping to prevent aliasing issues. Operations serve as the basic building blocks of MLIR's intermediate representation (IR), encompassing arithmetic, logical, and control flow operations. Regions group operations together similarly to blocks in C-like languages and can be attached to control flow operations; their design was inspired by Click's IR (see Section 2.3.1.5.1.1). Attributes provide additional information linked to operations, such as metadata or control flow details. Dialects logically group operations, attributes, and types to establish common namespaces and functionalities. MLIR includes several built-in dialects, ranging from those that closely map to LLVM IR to others that support high-level tensor manipulation primitives.

Unlike most other Static Single Assignment (SSA) implementations, MLIR is "functional," which means that regions must end with a terminating operation that defines a value for the region, and consequently for its controlling operation, to evaluate to. This functional approach makes φ-functions mostly implicit within the system.

2.3.3 Mathematical Intermediate Representations

The final set of intermediate representations rely on mathematical substitution as the foundational mechanism for defining and executing computations. However, directly compiling mathematical substitution is inefficient, so most real-world systems transform it into more practical forms. One common technique is the use of closures, which package function code together with an environment that captures any free variables. Another approach is to convert to Continuation-Passing Style (see Section 2.3.3.2), which restructures functions to receive an explicit continuation,

making control flow and substitution more manageable. Alternatively, compilers may apply Lambda Lifting [71], converting lambdas into global functions by making free variables explicit parameters, which allows them to be more easily represented in Linear or Graphical Intermediate Representations.

2.3.3.1 Lambda Calculus

The primary motivation behind Lambda Calculus's [3] development was to create a formal system for expressing functions and performing computations. Additionally, it aimed to establish a formal foundation for combinatorial logic—a type of logic that focuses on constructive proofs rather than classical logical operators.

In Lambda Calculus, functions are expressed as anonymous expressions known as lambda terms or lambda abstractions. These terms are built using variables, parentheses, and the Greek letter lambda (λ). The λ symbol introduces a function that takes a single argument. For example, the expression $\lambda x.x$ defines a function that takes an input x and returns x itself.

Although lambda terms take only one input at a time, this is not a limitation. A function can return another function, which makes it possible to construct multi-parameter functions such as $\lambda x.\lambda y.x + y$, effectively defining a two-parameter addition function.

With just these basic constructs, it is possible to represent any computation. For instance, one can define Boolean logic purely in terms of functions:

- $\text{true} = \lambda x.\lambda y.x$
- $\text{false} = \lambda x.\lambda y.y$

Using these definitions, an `if-else` construct can be encoded as:

- if-else $= \lambda b.\lambda x.\lambda y.bxy$

Function application in Lambda Calculus is straightforward: An argument is applied simply by writing it next to the function. Evaluating the result involves substituting the function's parameter with the given argument—a process known as beta reduction ($b\eta$-reduction). For example, applying the `ifelse` function to the inputs True, M, and N:

$$\text{if-else true } M \ N$$

$$\text{becomes } (\lambda b.\lambda x.\lambda y.b \ x \ y) \ (\lambda x.\lambda y.x) \ M \ N$$

$$\text{// Notice how every instance of b is replaced with } (\lambda x.\lambda y.x)$$

$$\text{becomes } (\lambda x.\lambda y.(\lambda x.\lambda y.x) \ x \ y) \ M \ N$$

$$\text{becomes } (\lambda y.(\lambda x.\lambda y.x) \ M \ y) \ N$$

$$\text{becomes } (\lambda x.\lambda y.x) \ M \ N$$

$$\text{becomes } (\lambda y.M) \ N$$

$$\text{becomes } M$$

This process demonstrates how the function selects and returns the appropriate value based on the input boolean.

However, Lambda Calculus also has its limitations. It lacks built-in support for basic data types such as numbers, booleans, and lists. These must all be encoded using functions—like Church numerals for representing numbers—which can be both inefficient and difficult to read. Additionally, $b\eta$-reduction does not translate well to modern machine code, making Lambda Calculus more suitable as an intermediate representation for interpreters rather than compilers.

2.3.3.2 Continuation Passing Style

Continuation-Passing Style (CPS) [115] was developed to make the control flow of programs explicit, especially in the contexts of programming language theory, compiler design, and functional programming. By transforming the way functions return results, CPS provides a robust framework for reasoning about program execution and implementing advanced control structures.

In CPS, functions do not return results in the conventional sense. Instead, each function takes an additional argument called a continuation—a function that represents the rest of the computation to perform after the current function completes. When a function finishes its computation, it does not return the result; instead, it calls the continuation function with the result as an argument. This means all function calls in CPS are tail calls, which eliminates the need for traditional return mechanisms and creates a uniform structure ideal for expressing advanced control features such as exceptions, coroutines, and backtracking.

The CPS transformation can be formally constructed in the Lambda Calculus by rewriting expressions so that they explicitly take and apply continuations. Might [89] presents this process in terms of three functions: the T-c transformation assumes that a continuation is already available and rewrites expressions so that the result is passed directly to this continuation. In contrast, T-k is more flexible: it takes a function that, given a CPS-compatible value, returns a CPS expression. This makes T-k particularly useful for composing nested or complex expressions. The M transformation handles atomic expressions such as variables and lambda abstractions, converting them into forms that are compatible with CPS. Specifically, M rewrites lambdas so they accept continuations and pass their results to them. The transformation process uses T-c when

the context provides an existing continuation (such as at the top level) and resorts to T-k for constructing more intricate or nested CPS structures.

There is a deep structural similarity between CPS and Static Single Assignment form (see Section 2.3.2.3), allowing for translation between the two. Kelsey [72] shows that the one-to-one binding of variables in CPS and the precise control flow through continuations align closely with SSA's approach of unique variable assignment and block-structured control flow. In his transformation, CPS continuations are treated as SSA blocks. Those labeled as jump—which in CPS distinguish control-flow branches such as conditionals and loops—correspond to SSA's labeled blocks containing φ-functions to reconcile incoming control paths. CPS bindings and function applications map directly to SSA variable assignments and control-flow structures. Tail calls to continuations become jumps, akin to gotos in imperative languages, while CPS's conditional structures translate directly to SSA's if-statements. Furthermore, loops expressed via recursion in CPS are transformed into SSA loops using recursive blocks and φ-functions to merge values across iterations.

However, this transformation is not without limitations. Some CPS programs, particularly those involving non-local continuations (such as those constructed using call-with-current-continuation in Scheme), cannot be faithfully represented in SSA form. SSA's structure lacks the flexibility to model such arbitrary control transfers. Likewise, specific patterns in SSA, particularly irreducible control flow, pose challenges when converting from SSA back to CPS. These edge cases highlight fundamental differences between the models and show that while the two Intermediate Representations are very similar, they are not equivalent.

2.3.3.3 Interaction Nets

Interaction Nets [77] were created as a new kind of programming language characterized by several distinct features. They are based on a simple graph rewriting semantics, which provides a clear and intuitive foundation for computation. A notable aspect of interaction nets is the complete symmetry between constructors and destructors, ensuring a balanced and elegant structure. Additionally, they incorporate a type system designed to enable deterministic and deadlock-free microscopic parallelism, making them particularly well-suited for concurrent computations. One of the main selling points of interaction nets is their inherent parallelizability, which has contributed to a modest modern resurgence in interest, as evidenced by ongoing projects such as the HVM [62] implementation.

Interaction Nets can be understood as an extension of Proof Nets [54] applied to the domain of programming languages. Each symbol—or agent—in the net is assigned a specific port known as the principal port, which is distinguished from the auxiliary ports. Interaction occurs when agents connect through their principal ports, and each matching pair of symbols has an associated reduction rule that defines their interaction. The system also introduces a type system that restricts agent interactions based on their types. The language defines several constant types, including atom, list, nat, d-list, stream, and tree. Moreover, each port is typed as either input (denoted T-) or output (T+). For a reduction rule to be well-typed, the symbols on the left side of the rule must have opposite port types, and the resulting right side must also be well-typed.

One of the significant challenges with interaction nets lies in their construction. There is no widely established method for constructing interaction nets directly, and existing implementations [59, 62], typically build custom languages that map closely—

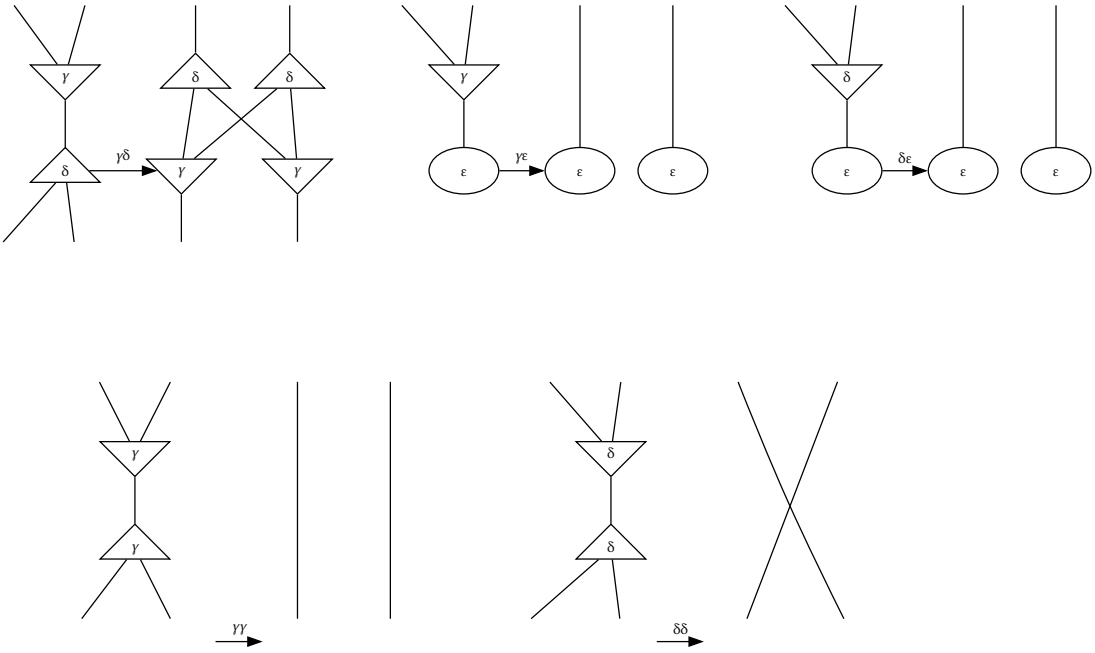

Figure 2.8: Five of the six interaction net substitution rules, $\gamma\delta$-interaction (top left), $\gamma\varepsilon$-interaction (top center), $\delta\varepsilon$-interaction (top right), $\gamma\gamma$-interaction (bottom left), and $\delta\delta$-interaction (bottom right). The sixth is $\varepsilon\varepsilon$-interaction in which both symbols are erased.

but not trivially—to interaction nets. This suggests that mapping existing programming languages onto interaction nets can be difficult. A related concept, interaction combinators [76], serves as a simplified model of interaction nets. While the original formulation of interaction nets included symbols tailored to a Lisp-like list processing language, interaction combinators reduce the system to just three symbols: γ (constructor), δ (duplicator), and ε (eraser). These are governed by six interaction rules that define their behavior which are presented in Figure 2.8.

2.4 Discussion

Intermediate representations (IRs) in compilers serve as critical abstractions for analyzing, transforming, and optimizing programs. Graph-based IRs like Graphical Intermediate Representations, Directed Acyclic Graphs (DAGs), and Program

Dependence Graphs (PDGs) visually model computations through nodes and edges, emphasizing structural and semantic relationships between operations. Variants such as Control Flow Graphs (CFGs) and Dataflow Graphs (DFGs) specialize in capturing execution paths and data dependencies, respectively. At the same time, hybrids and extensions like Value Dependence Graphs, Program Dependence Webs (PDWs), and Dependence Flow Graphs offer enriched semantic modeling for research and parallelism.

Linear IRs, including Three-Address Code (TAC) and Polish Notation, represent computations sequentially, making them easier to serialize and map directly to machine code. Single Static Assignment (SSA) form, and its extensions like Gated Single Assignment (GSA) and SSA Graphs, underpin many modern compiler optimizations by simplifying variable tracking and data flow. Popular frameworks like LLVM IR and MLIR build upon SSA to enable modular, multi-target compilation and support for domain-specific representations across multiple abstraction levels.

Functional and mathematical IRs, such as Lambda Calculus, Mathematical IRs, and Interaction Nets, abstract away control flow in favor of expression evaluation and substitution. These representations, while less common in imperative language compilers, are foundational in the functional programming ecosystem, enabling formal reasoning, parallelism, and verification. Though many IRs vary in complexity and domain focus, from industry standards like LLVM to research-oriented models like Click's IR or Program Expression Graphs, their collective role is to bridge the gap between source code and efficient executable programs.

2.4.1 Optimization

IRs in compilers play a crucial role in enabling various forms of program optimization. Graph-based IRs like Control Flow Graphs (CFG), Dataflow Graphs (DFG), Program Dependence Graphs (PDG), and Directed Acyclic Graphs (DAG) make control and data dependencies explicit, allowing for optimizations such as dead code elimination, instruction reordering, and parallelization. Hybrid models that combine control and data aspects, like Control Flow/Dataflow Hybrids and the Dependence Flow Graph, enhance capabilities like branch prediction and instruction-level parallelism. These representations also support advanced analyses such as speculative execution and memory disambiguation.

Several specialized IRs further refine optimization potential. SSA (Single Static Assignment) and its variants, like SSA Graph and Gated SSA, simplify dataflow analysis and register allocation by making variable versions and dependencies clear. Representations like Click's IR, LLVM, MLIR, and PEG are designed for aggressive or domain-specific optimizations, supporting techniques like loop unrolling, superoptimization, and fusion. Others, such as Three-Address Code and Linear IRs, offer efficient paths to low-level machine code, supporting local optimizations and strength reduction.

Functional and mathematical representations, including Lambda Calculus, Mathematical IRs, and Interaction Nets, are well-suited for transformations like β-reduction, inlining, and partial evaluation. These IRs emphasize referential transparency and algebraic simplification, making them ideal for compilers targeting functional languages or high-performance computing contexts. While traditional forms like ASTs and Polish Notation offer limited optimization potential, they still serve as

foundational structures for parsing and initial program analysis. As the compilation process advances from analysis and transformation toward code generation, the role of intermediate representations shifts accordingly.

2.4.2 Code Generation

IRs serve various roles in the code generation pipeline, ranging from abstract syntax trees (ASTs) to highly optimized low-level forms. High-level representations like ASTs or Program Expression Graphs (PEGs) must first be transformed into lower-level IRs due to their lack of machine semantics. Graph-based IRs, such as Control Flow Graphs (CFGs), Dataflow Graphs (DFGs), and Control/Dataflow hybrids, are valuable for structuring computation and scheduling, often aligning well with hardware targets, particularly in parallel and SIMD/SIMT environments. Some specialized forms, like the Program Dependence Graph (PDG) and Program Dependence Web (PDW), are more relevant for transformations and speculative execution strategies than direct code generation.

Linearized IRs, such as Three-Address Code, SSA form, and its variants (SSA Graph, Gated SSA), are more suitable for direct code generation due to their close mapping to machine instructions and ease of register allocation and instruction scheduling. While SSA form requires conversion (e.g., φ-node elimination) before final codegen, it remains a backbone in modern compiler infrastructures like LLVM and GCC. Similarly, representations like Polish Notation and Click's IR can be interpreted or directly lowered into machine code in production environments. Linear IRs often represent the final stage before generating assembly code.

Modern compiler frameworks like LLVM and MLIR further abstract and streamline the code generation process. LLVM provides direct support for multiple architectures

through a unified IR, offering robust optimization and instruction selection backends. MLIR, though not a final codegen IR itself, acts as a flexible intermediate layer supporting heterogeneous targets such as CPUs, GPUs, and FPGAs. Functional IRs, including Mathematical IRs and Lambda Calculus, typically undergo transformations (e.g., to continuation-passing style) before reaching low-level forms. Experimental models like Interaction Nets, though not mainstream, offer potential for highly parallel execution through runtime-managed rewrites.

2.5 Conclusion

This review explores a wide array of intermediate representations (IRs) beyond standard forms like ASTs, CFGs, and SSA, directly addressing our three research questions. For RQ1, we identify lesser-known or advanced IRs such as Program Dependence Graphs (PDGs), Dependence Flow Graphs, Program Dependence Webs (PDWs), Program Expression Graphs (PEGs), Click's IR, and mathematical or functional IRs like Lambda Calculus and Interaction Nets—each offering unique structural or semantic insights not found in conventional IRs.

Regarding RQ2, IRs like SSA (and its variants), PDGs, and hybrid control/data flow models prove most beneficial for optimizations, enabling techniques such as instruction-level parallelism, loop unrolling, memory disambiguation, and superoptimization. Functional IRs support algebraic simplifications and β-reduction, aiding optimizations in functional or high-performance computing domains.

In response to RQ3, linear IRs such as Three-Address Code, SSA, and even Polish Notation show strong suitability for code generation due to their machine-level alignment and straightforward mapping to assembly. Frameworks like LLVM and MLIR

exemplify how modern infrastructures leverage these IRs to support robust codegen pipelines across diverse hardware targets.

Acknowledgement

This material is based in part upon work supported by the National Science Foundation under grant numbers OIA-2019609 and OIA-2148788. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

3 Entity Component Systems: Challenges and Benefits

This chapter first appeared as a conference publication in ACM's SIGPLAN Onward! [38].

J. Dahl, and F. C. Harris Jr., "An Argument for the Practicality of Entity Component Systems as the Primary Data Structure for an Interpreter or Compiler," in Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, in Onward! '25. Singapore: Association for Computing Machinery, 2025.

Abstract

In this paper, we examine how Entity Component Systems, a data structure that has been gaining popularity in game engines, can benefit compiler and interpreter design. It does not consistently provide the same performance benefits that games utilize it for; however, it does make writing optimization passes easier. Additionally, its dogmatic focus on simple Plain-Old-Data structures makes serialization much easier. These benefits do come at a memory cost, the severity of which we still need to compare against more mature language implementations.

3.1 Introduction

Compiler construction is often considered one of the more challenging topics in computer science, frequently appearing near the top of lists that discuss the field's most difficult subjects [16]. This is somewhat surprising given the typical undergraduate curriculum: students are generally required to take courses in computer organization, assembly language programming, algorithms, and formal languages. In addition, many

programs include a capstone project or a bachelor's thesis, which encourages students to synthesize and apply knowledge from all of these domains.

Given this solid foundation, it would seem that most computer science graduates possess the necessary tools to build a compiler. So why, then, does the prospect of actually constructing one feel so overwhelming? We hypothesize that integrating a greater number of compiler development tools in a well-documented and user-friendly manner can help bridge the significant mental gap that would-be compiler developers often face. Toward this end, we examine how data-oriented design techniques can facilitate and simplify the process of compiler construction.

A significant source of inspiration for this project was a *CppNorth* talk by Chandler Carruth [17, 18], in which he discusses how Google is "modernizing" compiler development for their new *Carbon* programming language [14]. In particular, Google is attempting to leverage data-oriented programming techniques—approaches that have gained popularity in the video game industry for their performance benefits [11, 43, 122]—to improve compiler efficiency. However, Google's implementation is tightly coupled with the specific syntactic structure of *Carbon*. In this work, we experiment with similar data-oriented techniques, but apply them across three languages in three different domains and of increasing complexity with differing syntactic characteristics: a basic expression calculator, JSON [26], and a simplified imperative version of Lox [93].

For this preliminary analysis, we applied a strict design constraint: to implement all functionality using only a single generic data-oriented structure, an Entity Component System (ECS) supplemented by a single string containing the original input. The core idea of data-oriented design is to decompose data into the smallest logical groupings that are frequently accessed or manipulated together. These

groupings are then tightly packed into memory. This memory layout is particularly well-suited to the caching mechanisms of modern CPUs. When a program accesses a value in memory, the CPU typically loads not just the specific value, but also several surrounding bytes into the cache. These groups of nearby memory (commonly 64 bytes in size, known as cache lines [96]) enable faster subsequent access to nearby data. In practice, the data packed into ECS structures is usually composed of simple, Plain-Old-Data (POD) types [52] (types which can be treated as binary blobs that can be moved using `memcpy`), which are not only efficient to store and manipulate but also straightforward to serialize and deserialize.

Our goal was to explore whether an ECS is a useful data structure for compilers and interpreters. We began with the hypothesis that, much like in game development, an ECS could offer improved performance. Unfortunately, our evidence neither supports nor refutes this hypothesis; So, we also present some benefits to usability that emerged from using an ECS and a slight tweak we made to treat Component arrays as hashtables. Thus, this work has two primary research questions:

1. Does an ECS improve the performance of a compiler or interpreter?
2. Does an ECS make it “easier” to write compilers and interpreters?

Section 3.2 explores what exactly an ECS is, along with several other common interpreter architectures. Section 3.3 details the modifications we made to the ECS framework to enhance its utility as well as the design challenges we encountered while implementing interpreters for two simple languages and a more general-purpose (albeit still simple) language. Then, in Section 3.4, we outline the benefits that emerged during the development of these interpreters, while Section 3.5 addresses some of the

drawbacks we observed. Finally, Section 3.6 offers a summary of our findings and enumerates why we believe ECS can be a valuable tool in this context.

3.2 Background

Treewalk and bytecode interpreters are two foundational architectures used to implement programming language interpreters. A treewalk interpreter operates directly on the abstract syntax tree (AST) parsed from source code. The interpreter traverses the tree recursively, executing code by evaluating nodes according to their syntactic roles. This method is often favored in the early stages of language development due to its simplicity and close alignment with the structure of the source language [93]. While intuitive and easy to implement, treewalk interpreters typically suffer from performance inefficiencies, especially in large or compute-intensive programs, due to the overhead of recursive traversal.

In contrast, bytecode interpreters convert the AST into a lower-level, often linear intermediate representation (IR), where instructions are typically one byte in size—hence the name. This IR is then executed by a language runtime, commonly referred to as a virtual machine. Compared to raw ASTs, the IR is generally more compact and easier to analyze and manipulate, which enables more advanced optimizations [50]. Bytecode interpreters strike a balance between the flexibility of high-level languages and the efficiency of lower-level execution models, making them a popular choice for production-grade language runtimes such as the Java Virtual Machine [84] and Python's CPython interpreter [100].

We employ an ECS to design a new Attribute Bubble Interpreter that operates halfway between a traditional tree-walk interpreter (which directly traverses an

Abstract Syntax Tree, or AST) and a bytecode interpreter (where the AST is transformed into a more linear representation). The ECS data structure is centered around three primary concepts: Components, Systems, and Entities [63]. Components are stored in tightly packed arrays and consist of small, closely related pieces of data. Traditionally, Components are purely data-oriented and contain no behavior. However, it is not uncommon to include simple helper methods for lightweight computations.

The bulk of the processing logic—sometimes all of it, depending on how strictly one adheres to ECS principles—is handled by Systems. Systems are functions that operate on specified subsets of Components, usually in bulk, and are responsible for implementing the behavior of the game or simulation. Each System receives a context object that includes all relevant Component arrays, allowing it to process Entities efficiently.

Entities, by contrast, act as indices into these Component arrays and represent unique objects in the simulation. They function similarly to keys in a relational database [20], where each column corresponds to a Component array. Just as some database rows may lack values in specific columns, Entities in an ECS may not be associated with every possible Component. Table 3.1 presents a sample ECS setup that stores the data required to position, move, and accelerate multiple Entities in a simple Asteroids-like game [6]. Meanwhile, Listing 3.1 shows the implementation of a System that performs a discrete approximation of the kinematics necessary for such a game.

```
void kinematics_System(Context& ctx) {
  for(uint e = 0; e < ctx.entity_count(); ++e) {
    if(!ctx.has_component<Position>(e)) continue;
    if(!ctx.has_component<Velocity>(e)) continue;

    Position& p = ctx.get_component<Position>(e);
    Velocity& v = ctx.get_component<Velocity>(e);
    if(ctx.has_component<Acceleration>(e)) {
      auto& a = ctx.get_component<Acceleration>(e);
      v.x += a.x;
      v.y += a.y;
      v.z += a.z;
    }
    p += v; // Assuming there is an element-wise overload
  }
}
```

Listing 3.1: This System updates the ECS defined in Table 3.1. It processes all matching Entities in bulk within the module/context. Note that only a subset of Entities—specifically those possessing both `Position` and `Velocity` components—are processed. In this example, however, that subset happens to include all Entities.

Entity	Position	Velocity	Acceleration	IsPlayer?	IsRock?
1	(0, 0, 0)	(0, 0, 0)	∅	✓	∅
2	(5, 0, 0)	(0, 0, 0)	(-1, 0, 0)	∅	✓
3	(0, 5, 5)	(2, 0, 0)	∅	∅	✓
4	(0, 2, 2)	(0, 5, 0)	∅	∅	✓

Table 3.1: An example ECS storing several Entities (a player and three rocks) that each have a position and velocity. There are several additional Components which may (✓) or may not (∅) be present for a particular Entity. The `IsPlayer?` and `IsRock?` components are tags: valueless Components that encode some boolean property about the Entity.

Each concept in an ECS corresponds neatly to familiar constructs in programming language design. Components resemble attributes in an IR, and Systems mirror analysis

or optimization passes. Entities are analogous to parser tokens or AST nodes; and the Context that wraps the Component arrays is comparable to a Module.)

However, when working with an ECS, a common issue arises when Components are added to only a few Entities, as is the case for the `IsPlayer?` and `Acceleration` components in Table 3.1. In such cases, large arrays are often created that contain mostly "null" or unused data, leading to significant memory waste. The most widely adopted solution to this problem in games is the use of archetypes [21].

In this approach, each unique combination of Components defines an archetype. Whenever a Component is added to or removed from an Entity, the Entity is moved to a different archetype that matches its new Component set. All Entities sharing the same archetype—i.e., the same set of Components—have their Components stored together in tightly packed arrays. This organization eliminates the need to reserve space for absent Components, thereby avoiding the inefficiency of storing null data to maintain a fixed array structure. When rewritten using archetypes, the example in Table 3.1 instead is represented by a table per archetype as shown in Table 3.2.

One alternative to archetypes is the use of Sparse Sets [21]. In this approach, the data for each Component type is tightly packed, while a separate sparse array of indices maps Entities to their corresponding Component data. The key idea behind this method is to trade off a small amount of memory used to store indices (typically only 2 to 4 bytes each) instead of wasting significantly more memory on partially filled Component arrays, where each Component might occupy 10 bytes, 100, or even more. An example of how this might be used to tightly pack the `Acceleration`s from Table 3.1 is shown in Figure 3.1.

Archetype 1

Entity	Position	Velocity	IsPlayer?
1	(0, 0, 0)	(0, 0, 0)	✓

Archetype 2

Entity	Position	Velocity	Acceleration	IsRock?
2	(5, 0, 0)	(0, 0, 0)	(-1, 0, 0)	✓

Archetype 3

Entity	Position	Velocity	IsRock?
3	(0, 5, 5)	(2, 0, 0)	✓
4	(0, 2, 2)	(0, 5, 0)	✓

Table 3.2: Table 3.1 rewritten to use archetypes. Notice that each archetype only stores a Component if all Entities within that archetype possess it, resulting in all Component arrays being densely packed. Additionally, every System must now loop over all archetypes in a module before iterating through each Entity within those archetypes, which adds more boilerplate code to their definitions.

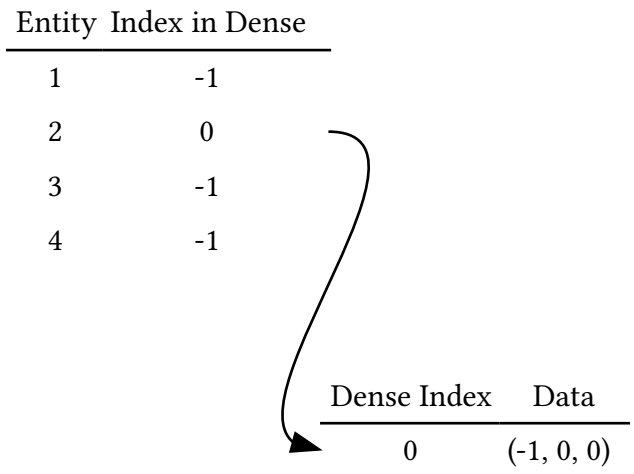

Figure 3.1: A Sparse Set that maps sparse Entity data to tightly packed `Acceleration` values. An index of −1 indicates that no data is associated with a given Entity. Assuming 4-byte integers and 12-byte (three f32) `Acceleration` values, this approach reduces wasted memory from $3 \times \text{sizeof}(\text{Acceleration}) = 36$ bytes down to $4 \times \text{sizeof}(\text{int}) = 16$ bytes. The factor of 4 (instead of 3 as in the first calculation) arises because the entire sparse array is considered wasted space, rather than just the extra null data. The main drawback is that mostly filled Component arrays end up wasting an additional $\text{sizeof}(\text{int})$ bytes for each Component.

3.3 Design

Our implementation introduces a subtle variation on the typical Sparse Set approach: Rather than storing sparse indices alongside each Component's storage, we instead associate sparse indices with each Entity, replacing the traditional use of an Entity bitmask, which is used to indicate which Components are associated with each Entity. This centralized sparse structure, visualized in Figure 3.2 and Figure 3.3, serves a dual purpose: It allows for efficient lookup of Component offsets and simultaneously indicates whether or not a Component exists for a given Entity. This tweak allows us to manipulate the order in which Components are stored at will. We use this ability to "sort" some Component arrays into hashtables, a feature that would be difficult to reproduce if using archetypes.

In general, the order in which Entities and Components are added will not necessarily be the same. If they get far enough apart, they will wind up in different CPU

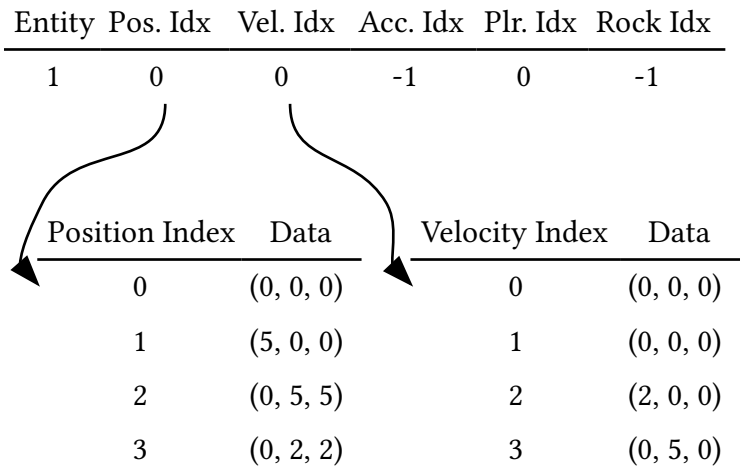

Figure 3.2: Entity #1 in Table 3.1 shown with its per Entity indices as well as mappings to the dense `Position` and `Velocity` arrays. Again, −1 indicates that a Component is not associated with the Entity. Additionally, in this scheme, external storage for Tag Components is not necessary: we can represent their presence (0) and absence (-1) in the Sparse Entity Indices.

cache lines, likely causing a cache miss and degrading the performance ECS was designed to improve. It is entirely possible to add Entities first and then add the same Component to each one in a random order. In such a situation, we lose all of our guarantees about associated data being close together, as shown in Figure 3.3. To address this issue, we provide a mechanism for sorting Components so they are in the same order as their Entities.

However, to swap two Components, we must linearly scan the Entity Sparse Indices to determine which Entity each Component is associated with, resulting in an operation with time complexity $O(n^2 \log n)$. To address this, our approach first sorts a

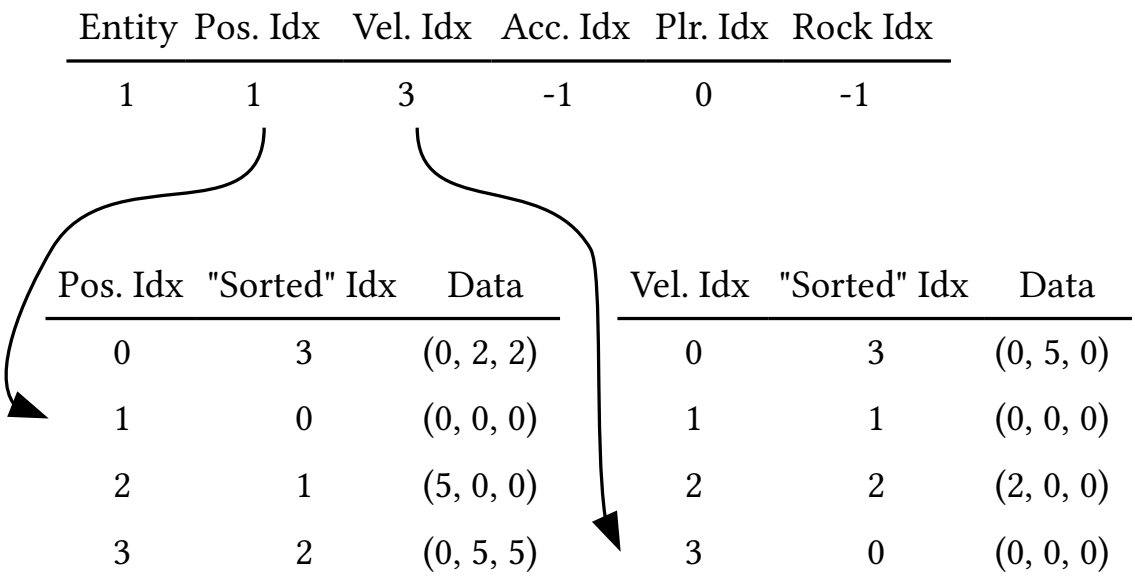

Figure 3.3: Figure 3.2 with its `Position` and `Velocity` component arrays shuffled to simulate more random insertion. In this example, if two `Position` or `Velocity` components fit within a single cache line, iterating through the shuffled `Position` and `Velocity` data would result in approximately 3 and 2 cache misses, respectively—compared to just one each if the data were sorted. This difference is significant when you consider that the main operation performed in Listing 3.1 (three additions) can ideally be completed in just 3 CPU cycles on an x86 machine, whereas a single cache miss to main memory can cost 100–200 CPU cycles—the equivalent processing time for roughly 33-66 entities.

proxy array of indices—functionally similar to the previously described Sparse Sets, where an additional array maps indices back to the original data. This sorted proxy array is then transformed into a series of swap instructions. This two-step process—sorting the proxy and executing the swaps—allows us to leverage fast sorting algorithms (Introsort [8, 25] in our case) without needing to consider the number of swaps the sorting algorithm will perform.

Despite our efforts to minimize the number of swaps, for Components that are expected to be sorted frequently, this process can quickly become prohibitively expensive. To mitigate this, we also provide a utility (enumerated in Listing 3.2) for storing Entities along with Component data, eliminating the need to search for the associated Entity. This reduces the computational complexity of this $O(n^2 \log n)$ operation to the usual $O(n \log n)$ at the cost of increasing the memory consumed by each Component in that Component array.

```
template<typename T>
struct with_Entity {
  T value;
  entity_t Entity = invalid_entity;
  // Methods excluded for brevity
};
```

Listing 3.2: A simple Component adapter that associates Entity information with a Component. We have template machinery in place to detect this wrapper and utilize the saved Entity instead of scanning for it.

In order to stress our ECS implementation, we have implemented three language processors of increasing complexity. The design of each presented us with several challenges, which we will now discuss and fix.

3.3.1 Calculator

The first language we implemented was a simple calculator interpreter, where users can type in basic math expressions to be evaluated. Additionally, variables can be defined and assigned values for use in future calculations.

A key challenge in this context was that, to maximize speed, memory allocations had to be minimized as much as possible. To address this, our System stores the original input string and produces views (pairs of position and length) into that input. This approach ensures that most strings are allocated only once, at the start of the program. However, a downside to this method is that very large programs may not fit into memory.

To provide some perspective, assuming a world record typing speed of 360 words (3,600 characters) per minute [57], it would take about 193 days of continuous typing with no rest to input a one-gigabyte-sized program. Given that modern machines typically have sixteen to thirty-two gigabytes of available memory, it is likely that the vast majority of programs humans are likely to type can comfortably fit into memory. For those programs that exceed this limit, solutions are still being considered as future work.

Another challenge we encounter is that when creating trees (or other types of graphs), the memory associated with the nodes is often scattered, which results in poor cache locality. If different types of Components are created for each graph node, they will be clustered into separate memory groups. As a result, when traversing the graph, the program would need to jump between these different clusters of node types, which leads to a reduction in cache locality. Our solution to this issue is to ensure that all

Components associated with a specific graph or tree element are of the same type. This allows them to be packed closely together in memory, improving cache locality.

To optimize this, it makes sense to minimize the number of "structure Components." We thus construct an AST for the parsed language using two primary (classes of) Components. One of these is an `operation` component, which stores a four-tuple that the AST relies on to build most of its trees. For expression nodes, the first and second elements of the tuple (or just the first in the case of unary operators) are used to represent the child nodes. In contrast, control flow nodes in more advanced languages (see Section 3.3.3) use all four slots, with the first slot being the condition and the remaining slots used for the block(s) of code to be executed and a marker to indicate the end of each block.

`operation` components represent one of the primary sources of conflicting pressures in this design. On one hand, smaller Components allow more of them to fit within a single cache line, reducing the likelihood of cache misses. On the other hand, having many different Component types means that each type is stored in a separate memory segment. Switching between these segments increases the chances of cache misses—unless the access pattern involves just a few types of Components.

This creates a tension: we want many small Components to minimize cache misses due to size, but we also want fewer, larger Components to minimize cache misses due to memory fragmentation. Designers must strike a balance between these competing factors. In our case, we introduced a third consideration: ease of implementation. We found that consolidating everything into a single operation structure made writing Systems significantly easier. Exploring this trade-off in greater depth is part of our planned future work.

The second class of Components in our AST design are Tag Components: Components that store no data but serve to indicate the specific type of tree node being represented, such as a variable, an addition expression, an if-statement, and so on. These Tag Components offer a more dynamic and flexible alternative to using `std::variants` (C++'s tagged unions) when representing different kinds of tree nodes. The definitions both for `operation` and several of the most pertinent tags are provided in Listing 3.3.

```
// The operation Component utilized by the Calculator
struct operation {
  entity_t left, right; // entity_t is a pointer sized uint
};

// The operation Component utilized by Lox
struct operation {
  // Condition, Then, Else, Marker
  Entity a = 0, b = 0, c = 0, d = 0; // Wrapper class
};

struct add {}; // Recall that tags store no data!
struct subtract {};
struct multiply {};
struct divide {};
struct power {};
struct assignment {};
```

Listing 3.3: Definitions for the `operation` component (both for the Calculator and Lox (see Section 3.3.3)) as well as the tags used to differentiate `operations` in the calculator.

It might seem that this approach shifts the problem of accessing different memory clusters in a cache-unfriendly manner to the Tag Components. However, we have a

trick up our sleeve: Checking if a Component is present only requires looking at the Entity Sparse Indices. This means that the only thing that needs to be loaded from memory are the Entity Sparse Indices, which are the most heavily accessed element in an ECS and thus very likely to already be loaded into the cache.

As a side note, the variant alternativeness of Tag Components also proves beneficial when writing a parser. Instead of having to define a rigid `std::variant` containing all the possible types that can flow through the parse, you only need to worry about passing around Entities.

3.3.2 JSON

Building on the lessons we learned from implementing a simple calculator interpreter, we moved on to a language that requires deeper levels of nested structure: JSON [26]. While not a programming language, JSON's ability to nest alternating types of structures was precisely the kind of additional structure we needed to ensure our system could support.

The first challenge we encountered was that JSON arrays need to store a collection of child elements. However, to maintain convenient serializability (which we will argue is a primary benefit of this system in Section 3.4.2) we had to ensure that all data stored in the ECS is represented as POD types without any pointers to external memory.

Fortunately, these POD types can still store Entities (which are represented simply as integers). With this constraint in mind, we implemented two core Components: `list` and `list_entry`. The `list_entry` component is responsible for storing the Entities that represent the next and previous elements in a linked list. Meanwhile, the `list` component is attached to the list head (the Entity representing a JSON array or object) and manages the overall structure of the linked list built out of `list_entry`

components. This structure, including `list` components as well as how they are used to derive arrays and objects, is illustrated in Listing 3.4. These Components are designed to be generic and inheritable since attempting to use a single list type for both arrays and objects could potentially introduce conflicts when these elements are nested.

```
struct list_entry {
  Entity next = invalid_entity, previous = invalid_entity;
  // Helper methods excluded for brevity
};
template<std::derived_from<list_entry> T>
struct list {
  T children = {};
  Entity children_end = invalid_entity;
  // Methods excluded for brevity
};

struct array_entry : public list_entry {};
struct array : public list<array_entry> {};

struct object_entry : public list_entry {};
struct object : public list<object_entry> {};
```

Listing 3.4: Generic `list` and `list_entry` components. Followed by how they are used to implement the JSON `array` and `object` components. Two separate versions are necessary so that they can each be checked for using `has_component` and `get_component`.

While it is well known that linked lists are generally less efficient than contiguous arrays [90], our approach demonstrates that it is still possible to represent nested structures using only an ECS (by building additional "graphs" on top of each other). Even in scenarios where our strict design constraints do not apply, a conversion from a

more efficient `std::vector`-based implementation to this ECS-linked-list model could still prove useful for serialization purposes.

With JSON arrays and objects now knowing about their children, we had another obstacle to overcome: Objects connected to identifiers (such as variables, functions, and types), or in the case of JSON, object members, needed to be efficiently looked up. In a standard ECS, this kind of lookup requires a linear scan. However, for large numbers of Entities, linear scans become prohibitively expensive. To address this, we leveraged our existing capability to reorder (sort) component arrays, enabling us to organize a Component's storage into a hashtable.

After analyzing a performance benchmark for modern hashing algorithms [82], we chose to implement Hopscotch Hashing [61] due to its consistently fast lookup times, and the fact that it is an open addressing (no extra list) method, so it can be implemented by simply reordering a Component array. For the hashing algorithm itself, we selected the FNV-1A algorithm [91], which is specifically designed to be fast while maintaining a low collision rate: both critical characteristics for a high-performance implementation with a bounded maximum number of collisions (Hopscotch Hashing stores a fixed size number of neighbors in a bitmask and thus does not support more collisions than the size of said bitmask).

Our hashtable component arrays diverge from the conventional hashtable model in that elements are added in the same manner as any other Components (typically via a list append). Thus, the table must be rehashed before a search can occur since its properties are invalidated after each addition. In typical usage, the hashtable is populated during a parse, rehashed once parsing is complete, and then used like a standard table. However, Component lookup operations return the associated Entity (as

opposed to the Component data itself), enabling other Components tied to that Entity to be queried in the usual ECS fashion.

3.3.3 Imperative Lox

With our ability to perform calculations and look up references now prepared, all the necessary parts were in place to implement a more general-purpose programming language.

In his book *Crafting Interpreters* [93], Robert Nystrom introduces Lox, a simple, dynamically typed, object-oriented programming language. In addition to the two implementations of Lox included in the book, the language has inspired a large number of third-party implementations across a variety of programming languages. Nystrom also provides a suite of benchmark programs, making Lox an ideal candidate for performance benchmarking.

3.3.3.1 Attribute Bubble Interpretation

One of the most apparent advantages of using an ECS is the ease with which new types of structures can be dynamically attached to Entities. For example, during interpretation, it is convenient to attach runtime-type information to AST nodes as they are processed. Listing 3.5 illustrates how our implementation is capable of querying this runtime-type information efficiently. The ECS facilitates this kind of dynamic attachment through a single function call, making the process both straightforward and flexible. Consequently, we treat the ECS itself as the primary data store for running programs.

Since Components in our ECS are stored in flat component arrays, topologically sorting the AST into a specific traversal order transforms the traversal process into a

```
runtime_type determine_runtime_type(Entity e) {
  if(e.has_component<runtime_type>())
    return e.get_component<runtime_type>();
  else if(e.has_component<Lox::null>())
    return runtime_type::Null;
  else if(e.has_component<double>())
    return runtime_type::Number;
  else if(e.has_component<bool>())
    return runtime_type::Boolean;
  else if(e.has_component<Lox::string>())
    return runtime_type::String;
  else return runtime_type::Invalid;
}
```

Listing 3.5: Code used to determine the runtime-type of an Entity in an Attribute Bubble interpreter.

simple linear iteration. By arranging the elements in reverse order, as shown in Listing 3.6, we gain a slight efficiency boost, as comparisons against zero can take advantage of specialized instructions provided by some CPU instruction set architectures.

```
// This loop will run slightly faster...
for(uint i = N - 1; --i; ) { /* useful code*/ }


// Than this loop
for(uint i = 0; i < N; ++i) { /* useful code*/ }
```

Listing 3.6: x86 provides special instructions for checking if an integer is equal to zero. Thus, the top loop will run slightly faster than the bottom loop.

Among the various reversed traversal schemes we could use to sort our programs topologically, we chose reverse post-order, ensuring that each child node appears after

its parent. This ordering is beneficial when searching for a containing block. Rather than requiring every Component type to store a reference to its parent explicitly, we can rely on the assumption that a backward linear scan, as depicted in Listing 3.7, will eventually locate the relevant parent block in the rare occasions when they are needed. Moreover, with this ordering in place, a backward linear scan becomes a depth-first search.

However, to ensure that reverse iteration aligns with a valid execution sequence, the Entities (not just their Components) must be sorted. To accomplish this, we employ a sorting scheme for Entities that mirrors the one used for Components. Entities themselves can be easily reordered through simple swaps; however, any Components that store Entities must be made aware of the change.

To resolve an issue in our AST storage scheme—where functions and regular code blocks were not distinguished, causing unintended execution during reverse traversal—we introduced a special marker at the start of each block. This marker references the

```
Entity current_block(Module& module, entity_t root) {
  entity_t target = root;
  do {
  // Decrement until we find a block or run out of Entities
  while((root - 1) > 0 && !module.has_component<block>(--root));
  // If target is larger than the number of children in the root... it can
not be root's child
  } while(root > 0 && root + module.get_component<children>(root).total <
target);
  return root;
}
```

Listing 3.7: The code we use to find the block of code an Entity belongs to. Notice how the code is little more than a reverse linear scan.

block's Entity and, after sorting the AST, becomes the block's last Entity numerically. During reverse iteration, if a marker does not match the currently executing block, we skip to the correct block using the stored reference. This mechanism applies not only to functions but also to control structures like loops and if-statements, with markers added to loop bodies and the latter branch of conditionals to ensure correct traversal behavior.

A related issue arises on the other end of a function call: when invoking a function, we do not initially know where to begin the reverse iteration. To resolve this, we perform a recursive traversal of the AST, during which we calculate and store the number of child Entities associated with each Entity. This preprocessing step enables us to determine the exact size of each function's body. Then, when a function is called, we can consult this stored child count to know how many Entities to skip forward.

Once the AST has been topologically sorted (and child counts identified), we can carry out all further semantic analysis as linear iterations over the ECS. In our Lox implementation, we perform three additional semantic analysis passes:

1. The first is a lookup pass, which resolves lexeme references to their corresponding Entities. For example, when a variable `X` is used, this pass identifies where `X` is defined in the program and stores a reference to that Entity.
2. The second pass is reference verification, which checks the results of the lookup pass and reports errors for any unresolved variables or functions. This function is provided in full in Listing 3.8.
3. The third is a function arity pass, which ensures that each function call has the correct number of arguments as defined in its declaration, while reporting errors when mismatches are detected.

```
bool verify_references(Module& module) {
  bool valid = true;
  for(Entity e =
module.get_component<children>(1).reverse_iteration_start(1); e--; ) {
    if(e.has_component<call>()) {
      auto& call = e.get_component<call>();
      auto& ref = call.parent.get_component<Entity_reference>();
      if(!ref.looked_up()) {
        ERROR("Failed to find function.");
        valid = false;
      }
    } else if(e.has_component<assign>()) {
      auto& op = e.get_component<operation>();
      auto& ref = op.a.get_component<Entity_reference>();
      if(!ref.looked_up()) {
        ERROR("Failed to find variable.");
        valid = false;
      }
    } else if(e.has_component<variable>()) {
      auto& ref = e.get_component<Entity_reference>();
      if(!ref.looked_up()) {
        ERROR("Failed to find variable.");
        valid = false;
      }
    }
  }
  return valid;
}
```

Listing 3.8: The complete code for the semantic analysis pass that is responsible for determining if variables and functions exist. Notice how the System uses the AST's root's (Entity #1) child count to determine where to begin reverse iteration. Also, notice how easily a Component (AST node type) can be checked for and then accessed.

When all of these features are combined, they result in what we refer to as an Attribute Bubble Interpreter. In an Attribute Bubble Interpreter, all data is stored in preallocated memory locations corresponding to AST nodes within the ECS, and values (attributes) bubble down through the AST (starting at the last Entity and ending at the first). Listing 3.9 details a snippet of the interpreter code showing how attributes bubble through the multiply instruction.

This architecture eliminates the need for a traditional call stack, while function calls still operate correctly because each function has dedicated memory reserved in the ECS. However, recursion is not supported (with the exception of tail recursion). This limitation arises because each function call clears the previously allocated memory, preventing nested invocations from preserving state.

Similarly, this approach restricts classes to a single instance each. Just like function bodies, class bodies rely on preallocated memory, which prevents multiple instances from existing simultaneously.

This "pure" ECS-based model works reasonably well for simple imperative languages. However, implementing slightly more complex language features—even those found in minimally complex languages like C—would require additional side structures. As a result, this interpreter does not fully implement the Lox language as initially designed. Instead, it supports a simplified subset we call "Imperative Lox," where most of the functional and object-oriented features from the original language are either ignored or treated as errors.

3.3.3.2 Performance

To evaluate the performance of our Attribute Bubble Interpreter, we used Nanobench [81], a C++ benchmarking library that repeatedly runs a code snippet and

```
bool interpret_multiply(Module& module, Entity target) {
  auto& op = target.get_component<operation>();
  if(
    determine_runtime_type(module, op.a) == runtime_type::Number
    && determine_runtime_type(module, op.b) == runtime_type::Number
  ) {
    target.get_or_add_component<runtime_type>() = runtime_type::Number;
    target.get_or_add_component<double>() = op.a.get_component<double>() *
op.b.get_component<double>();
    return true;
  }
  // Error handling omitted for brevity
}

std::pair<bool, bool> interpret(TrivialModule& module, entity_t root, Entity returnTo = 0) {
  bool valid = true;
  bool should_return = false;

  for(
    Entity e = module.get_component<children>(root).reverse_iteration_start(root);
    (e--).Entity > root && valid && !should_return;
  ) {
    // Many branches omitted for brevity...
    else if(e.has_component<multiply>())
      valid &= interpret_multiply(module, e);
    // Many branches omitted for brevity...
  }

  // If this is a function call... mark that we returned null
  if(returnTo.Entity > 0) returnTo.get_or_add_component<runtime_type>() = runtime_type::Null;
  return {valid, should_return};
}
```

Listing 3.9: The main interpret function used to bubble attributes, along with the multiply instruction. Notice that a `runtime_type` and `double` are directly attached to the Entity representing the AST node for the multiply instruction. Attaching values to AST nodes is the primary mechanism utilized by an Attribute Bubble interpreter.

averages the execution time. We used it to test our Lox implementation against the C (bytecode) and Java (tree-walk) implementations from *Crafting Interpreters* [93].

We benchmarked two types of Lox programs: The first is a simple example that performs basic arithmetic operations and prints the results. This program is referenced throughout the later sections and shown in detail in Listing 3.10.

```
var x = 5; var y = 6; var z = 7;

print x + y * z / (z - y);
```

Listing 3.10: A simple Lox program that performs a simple computation and prints the result.

The second group consists of benchmarks selected from those provided by Nystrom [92]. We chose only those that do not rely on functional or object-oriented features, as Attribute Bubble interpretation does not support them. `benchmark/equality.lox` runs several loops ten million times, repeatedly creating constants and performing equality comparisons, many of which are redundant. `logical_operator/and.lox` performs several short-circuit evaluations, while `operator/comparison.lox` executes multiple logical comparisons. However, in both of these benchmarks, the majority of execution time is spent on print instructions. Finally, `if/truth.lox` evaluates a series of if statements.

All benchmarks were executed on an Intel i7-13700K processor, running on a MAG Z790 Tomahawk Wifi with 32 GB of DDR5-6400 RAM, and a 1 TB Samsung 980 Pro SSD, running Ubuntu 24.04. The code used to implement and run these benchmarks was compiled with GCC 13.3.0 using CMake's default Release flags and is publicly available in our GitHub repository [33]. We also ran an additional benchmark [36] to measure the startup time of a program with an empty main function. The average startup time of 1,098,828.17 ns was added to each ECS benchmark result to account for the fact that

these tests run within the benchmark executable and therefore do not incur their own startup overhead.

The results of these benchmarks are presented in Figure 3.4. For simple programs, the Attribute Bubble Interpreter outperforms both the treewalk and bytecode interpreters, showing significantly better performance. However, as programs become less linear, their performance degrades considerably. These findings thus do not support our hypothesis that using an ECS improves execution performance. In fact, the performance of the non-linear test (`benchmark/equality.lox`) undermines what is often considered the primary advantage of ECS in the context of game development.

We believe this performance gap arises from a fundamental difference in access patterns: games typically involve straightforward linear iteration over Entities, whereas compilers require much more random access to data. While interpretation still involves a good deal of direct iteration, it also frequently requires jumping to other parts of the ECS—for example, when a function is invoked or a loop is iterated. Similarly, following variable references or other indirections often breaks the sequential access pattern. Each of these jumps and indirections invalidates the cache locality that the games industry heavily focuses on. However, in compilers, we have plenty of operations that do not march straight through the data without stopping.

With all the focus on the poor results of the `benchmark/equality.lox` test case, it is easy to overlook that the more linear test cases (where the ECS was allowed to "march straight through") outperformed the competition, averaging 1.3 times faster than C and 34 times faster than Java. While performance may not always be the primary reason to choose an ECS, it remains a significant consideration, especially since we believe many optimization passes will find themselves in this "march straight

Benchmark	ECS	C	Java
Listing 3.10	1.125 ms	1.572 ms	39.513 ms
logical_operator/and.lox	1.150 ms	1.474 ms	39.171 ms
operator/comparison.lox	1.192 ms	1.424 ms	39.472 ms
if/truth.lox	1.138 ms	1.556 ms	39.163 ms
benchmark/equality.lox	24842.964 ms	2051.956 ms	2888.242 ms

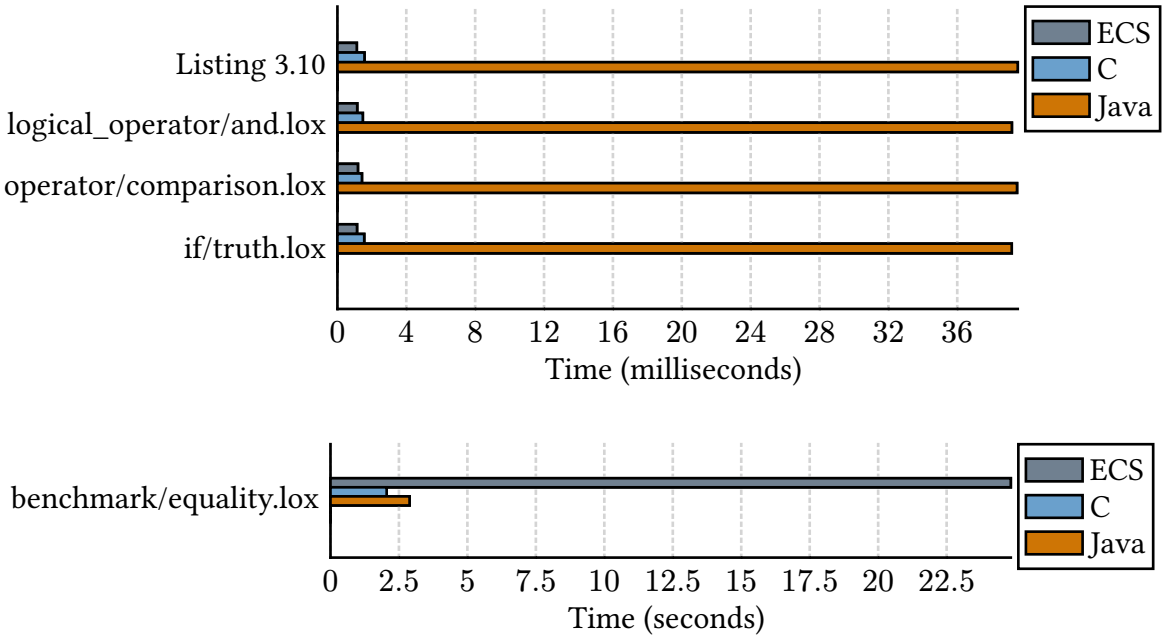

Figure 3.4: Comparison in time (lower is better) presented as a table (top) and graphically (bottom) of our Lox interpreter (ECS/Gray) compared against Nystrom's C (Blue) and Java (Orange) implementations of Lox running the example code in Listing 3.10 as well as several benchmarks taken from the Lox test set [92]. `benchmark/equality.lox` took substantially longer than the others and is thus separated; also note its different timescale.

through" scenario. Additionally, we discovered several notable usability benefits, particularly if some of our stricter constraints are relaxed, which we believe make ECS appealing even for cases that do not find themselves on the linear "happy path."

3.4 Additional Benefits

Combined with the lack of support for object-oriented and functional programming paradigms, the Attribute Bubble Interpreter ultimately serves more as a proof of concept than a practical tool. It demonstrates that interpretation is possible within the strict limitations we imposed (a single ECS and a single string), but realistically offers little outside of academic novelty. That said, the ECS architecture itself still provides several notable benefits outside the scope of performance, which warrant further consideration.

3.4.1 Storing Multiple Intermediate Representations Attached Together

The Attribute Bubble Interpreter highlights the power of dynamically adding Components within an ECS-based System. We have demonstrated that it is possible to topologically sort a tree in such a way that a simple linear iteration over the structure effectively simulates a traditional traversal. Once the AST is in this linearized form, we can attach Three Address Code (3AC) tuples to each node—using the node's ID as the result and the first two fields of the operation as operands—effectively embedding linear code directly within the tree. Because we can perform linear iterations over this structure, executing the code becomes as straightforward as iterating to traverse the tree, skipping over any nodes that do not contain associated 3AC instructions.

Our 3AC generation function is slightly more complex than a basic implementation, as it also does some basic variable assignment optimization. Due to the structure of our AST, directly generating 3AC can produce multiple layers of redundant assignments, an issue illustrated in Listing 3.11. To address this, we apply a simple substitution pass, provided in full in Listing 3.12, that analyzes how many times each

```
// Before
23 <- 5 immediate
24 <- 23
19 <- 6 immediate
20 <- 19
15 <- 7 immediate
16 <- 15
12 <- 20
11 <- 16
10 <- 11 - 12
9 <- 16
8 <- 20
7 <- 8 * 9
6 <- 7 / 10
5 <- 24
4 <- 5 + 6
```

```
// After
23 <- 5 immediate
19 <- 6 immediate
15 <- 7 immediate
10 <- 15 - 19
7 <- 19 * 15
6 <- 7 / 10
4 <- 23 + 6
```

Listing 3.11: An example of the 3AC we generate before (left) and after (right) removing redundant assignments for the Lox code in Listing 3.10. The numbers represent Entity IDs; in both cases, the value attached to Entity #4 would be printed (if using an Attribute Bubble interpreter).

AST node is assigned a value. Any assignment to a node that only receives a single value throughout the program is considered redundant and is removed.

To perform this substitution, we first count the number of assignments associated with each node. Notably, the counters used for this process can be efficiently discarded with a single function call (the last line of Listing 3.12), allowing us to reclaim memory with minimal overhead.

For reference, Appendix A contains the complete AST, including all attached 3AC instructions, corresponding to the example presented in Listing 3.11.

3.4.2 Simple Serialization for Simple Intermediate Representations

A keen reader may have noticed that we did not address control flow in the previous section. While our current implementation omits them, we have conducted some preliminary experiments involving the construction of a Control Flow Graph (CFG) to support the construction of Single Static Assignment (SSA) form. Although our ECS structure is capable of representing graphs, implementing a CFG in practice

```
std::unordered_map<entity_t, entity_t> substitutions;
for(Entity e =
module.get_component<children>(1).reverse_iteration_start(1); e--; ){
  if(!e.has_component<addresses>(module)) continue;

  auto& addresses = e.get_component<addresses>(module);
  if(substitutions.contains(addresses.a)) addresses.a =
substitutions[addresses.a];
  if(substitutions.contains(addresses.b)) addresses.b =
substitutions[addresses.b];

  auto assignments =
addresses.res.get_component<assignment_counter>().count;
  if(assignments == 1 && addresses.b == 0 && addresses.a != 0) {
    substitutions[addresses.res] = addresses.a;
    e.remove_component<addresses>(module);
  }
}

module.release_storage<assignment_counter>();
```

Listing 3.12: A snippet of the code responsible for building a 3AC representation that optimizes away redundant assignments. It removes 3AC entries when there is only one assignment to the result, and the entry does not represent loading an immediate value.

requires allocating additional side structures, such as `std::vectors` to track predecessor and successor blocks.

Until now, arrays of block children and parameters have been managed using a linked list approach, with `list_entry` components attached directly to the relevant child Entities. However, extending this method to support arbitrary graph structures—where a single node can have multiple parents—introduces significant complexity. This would require duplicating component types (as illustrated in Figure 3.5), dynamically managing which unique copy to use, and relying on the assumption that no node exceeds the fixed number of supported list chains for parent connections.

This issue would be trivial to resolve if we allowed parent nodes to store a `std::vector` of their children. However, our current ECS design is intended to serialize directly to disk, which precludes the use of such dynamic memory facilities.

As long as each Component stores only plain data—replacing all pointers to dynamic memory with Entity IDs and attaching other flat Components—the entire structure can be easily moved from memory to disk. Our entire serialization code [37] for ASTs and 3AC (neither of which has this duplicate list chain problem) is concise, comprising less than 50 lines of code in each direction. Additionally, most of the data is transferred efficiently using large `memcpy` operations, making the process both fast and straightforward.

3.4.3 Linear Traversals with Dynamic Polymorphism

Earlier, we discussed the semantic analysis performed by the Attribute Bubble Interpreter. Listing 3.8 presented the complete code used to verify whether references to variables and functions are valid. We also mentioned a simple optimization applied during the construction of our 3AC, with the relevant code provided in Listing 3.12.

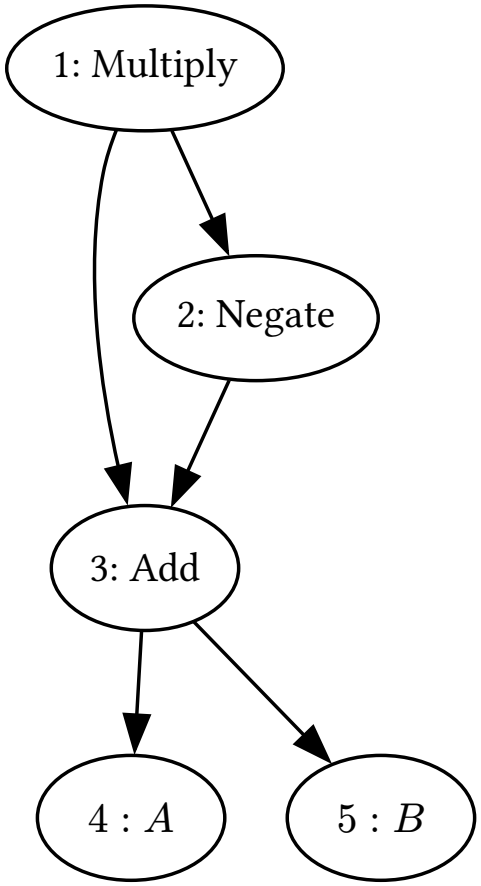

Figure 3.5: Consider Entity #2: we can easily create a `parent` component referencing Entity #1. But with Entity #3, things get more complex. Should we update the `parent` component to store both Entity #1 and Entity #2, or introduce a new `parent_2` Component? A similar issue arises for children: Entity #3 needs a `list_entry` for both parents. While we could define a `list_entry_2`, the core problem remains—no matter how many Component types exist, a program can always be generated that requires one more. This is not an issue if heap allocation is allowed, but that introduces non-ECS or string structures, complicating serialization.

Both of these common operations (reference verification and optimization) would typically require some form of tree traversal and likely a complex visitor pattern. However, in our implementation, they are handled as straightforward ECS-style Systems. These Systems consist of functions that apply bulk processing to every Entity

matching a specific set of Components. We believe that this simplicity in writing analysis and optimization passes is one of the greatest strengths of using an ECS.

Additionally, these ECS-style Systems can easily constrain which types of Entities they should run on. An Entity takes a role similar to a void pointer, but we can cheaply ask if the Entity has specific Components (say a call, assignment, or variable declaration tag). This allows for cheap and dynamic polymorphism where new Components can be attached to an Entity at will.

3.5 Limitations and Future Work

Throughout this paper, we have mentioned some of the limitations our strict requirements produced. However, while the ECS architecture in general provides some notable benefits, it also has some notable weaknesses. One of the key drawbacks is the memory overhead associated with Component storage. In our implementation, each Component requires an additional eight bytes to store Entity index information, which gets allocated for every single Entity. Thus, there are dueling pressures to minimize the number of Component types to reduce memory overhead while also being able to attach arbitrary information to solve specific problems, which inflates the Component count.

Furthermore, we attach source information to every Entity. While this approach simplifies access, it could potentially be optimized using a hashtable. However, in scenarios where every source entry is unique, a hashtable might actually introduce more overhead than it eliminates.

These issues tie into one of the more holistic limitations of this paper: The narrow scope of the programs we analyzed. Our longest test program spans only twenty lines

of code. Given this limited scale, we have not been able to thoroughly evaluate how certain costly operations, such as the tree-sorting step, scale with increased code size. This raises an important question: Does the performance gain from linear iteration and the simplicity it brings to implementing optimization passes outweigh the cost of tree canonicalization?

To properly address these questions, we need to develop implementations for larger, more complex languages and run them through more comprehensive benchmark suites. An implementation of the C programming language is already in development, which should provide more insights into how well our ECS-based System scales. We hypothesize that our linear iteration optimization passes will scale more favorably with program length than traditional tree traversal methods. However, to validate the performance and memory efficiency of our System, it will be necessary to compare it against established compilers like Clang or GCC.

3.6 Conclusion

In conclusion, we have explored how Entity Component Systems (ECSs)—a data structure commonly used in game engines—can be applied in the context of compiler and interpreter design. While ECS does not offer the same consistent performance advantages to languages as it does to games, primarily due to the inherently more random access patterns found in programming languages, it still presents meaningful benefits. Notably, it simplifies the implementation of optimization passes and enhances the overall serializability of the System. Although our findings are preliminary and based on relatively small programs, they remain promising. We are eager to continue

this work and evaluate the approach more thoroughly on larger, more complex codebases.

Acknowledgement

This material is based in part upon work supported by the National Science Foundation under grant numbers OIA-2019609 and OIA-2148788. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

4 An Embeddable Virtual Machine

This chapter first appeared as a conference publication in SERA 2025 [39].

J. Dahl, Q. Contaldi, K. Partovi and F. C. Harris, Jr. “Mizu: A Lightweight Multi-Threaded Threaded-Code Interpreter that Can Run Almost Anywhere with a C++ Compiler” The 23rd IEEE/ACIS International Conference on Software Engineering, Management and Applications (SERA 2025) May 29-31, 2025 Las Vegas, NV.

Abstract

We present Mizu, a threaded-code interpreter for an assembly-like language designed to be embedded inside compilers. Mizu has three primary goals: to be lightweight, portable, and extensible. We explore Mizu’s lightweight core instruction set, along with several of its extensions and some of our methods for further extension, and the platforms it has been ported to. Additionally, we demonstrate how two high-level features—threading and foreign function interfaces—can be spelled at the level of an assembly language. Finally, we compare Mizu to several other popular interpreted languages and find that it’s performance sits comfortably between JIT interpreters (approximately 4 times slower on a CPU straining task) and regular interpreters (approximately 8.5 times faster on the same task) all with only fourty-six core instructions.

4.1 Introduction

Recently, compile-time execution has become an increasingly important goal for compilers targeting imperative languages. For example, since 2011, C++ has been expanding its support for the `constexpr` keyword [24], which encourages code to be

executed at compile time rather than runtime. Similarly, in 2016, Zig emerged as a language with `comptime` [133] as one of its central features. Consequently, compilers must now function as interpreters. While interpreters have been extensively researched, little work has focused on their role as ancillary components of a compiler.

Mizu [34] is designed to serve as a backend target for compilers, in a similar role to LLVM's Intermediate Representation (IR) [79]. While LLVM IR was designed to be paired with a compiler backend that reduces it to native machine code, *Mizu* is intended to function in environments where compiling is difficult, such as microcontrollers where running code from certain regions of memory is sometimes explicitly blocked.

The system's name is derived from the Japanese word for water: 水 (Mizu), symbolizing its designed ability to adapt and fit into a wide variety of contexts. It is thus designed with three primary goals: to be lightweight, portable, and extensible.

In our effort to create a system that is both lightweight and extensible, we drew inspiration from RISC-V [28], which offers several minimal instruction sets known as "extensions." To ensure portability, we implemented these instructions as simple C++ functions, all sharing a single tail-call optimized signature. These functions are then chained together into a threaded-code interpreter. Instructions are essentially functions with a predefined signature. Thus, adding a new instruction is straightforward: first define a function that specifies the desired behavior; then add a special macro at the end to properly dispatch the next instruction. Adding new third-party instructions, such as manipulating a compiler's data structures, is easy with this design. *Mizu* supports loading and interfacing with shared libraries, also making integrations possible after the design phase.

In addition to these primary goals, *Mizu* explores implementing features found in higher-level interpreters, such as threading and interfacing with foreign functions—all the way down to the assembly instruction level. Section 4.2 reviews alternative interpreters in the literature as well as some background on threaded-code, while Section 4.3 delves into the design of *Mizu*'s Instruction Set and Abstract Virtual Machine. Section 4.4 provides a comparison of *Mizu* to several popular peer interpreters and assesses its performance on its supported platforms. Finally, Section 4.5 discusses some of *Mizu*'s limitations and highlights areas for future exploration.

4.2 Related Work

Mizu is a threaded code interpreter based on the subroutine-threaded code model, developed by Curley [29] and explained by Berndl [10]. In this architecture, each virtual instruction is a function that ends with a return statement. The virtual program's instructions are loaded and translated into a sequence of call instructions, each targeting an instruction function. Calls are executed natively, threading the code through each function.

Shifting focus, Anderson and Matson [5] aimed to bypass Python's Global Interpreter Lock (GIL) by integrating OpenMP support into Numba [78], a JIT compiler for Python enabling multi-threaded execution, thereby allowing parallel execution of Python code. Numba generated multi-threaded code using LLVM, improving performance to match C code. However, this approach adds complexity, requiring developers to learn both OpenMP directives and Numba's JIT compilation process. We illustrate a simpler method of implementing multi-threading into an interpreter.

Wihuri [129] investigates parallelism and multithreading in a JavaScript Web Application for meteorological visualization. Motivated by the growing complexity of web apps and the need to improve performance, the study identifies bottlenecks such as slow weather animation updates, high XML parsing times, and rendering delays. Proof-of-concept parallelism techniques using Web Workers and parallel JavaScript APIs were implemented. Parallelism improved some areas, namely parsing XML in separate threads. However, JavaScript's architectural constraints limited its efficiency. The study highlights JavaScript's limitations for multithreading and offers practical insights. Conversely, the findings are specific to a single application and do not explore redesigns of JavaScript's event loop for better parallelism.

Maintaining a focus on "web" technologies, the design of *Mizu* is inspired by WASM3 [124], a threaded-code interpreter for WebAssembly. WASM3 employs a subroutine threaded-code model, which links together an array of function pointers that are then recursively executed, with each function calling the next in the array. To efficiently implement this using C, the instruction functions are designed so their invocations can be tail-call optimized; otherwise, the interpreter will quickly fill the physical stack with "useless" function frames. Similarly, *Mizu* is an interpreter designed to emulate a custom instruction set inspired by RISC-V.

4.3 Design

Programs in *Mizu* are stored as an array of `opcodes`. Each `opcode` contains a function pointer to a *Mizu* instruction, as well as three 16-bit register values: `out`, `a`, and `b`. These values represent the indices of the registers used for saving to or loading from.

Listing 4.1 shows the full implementation of the add instruction, which shows how register saving and loading is performed.

```
void* mizu::instructions::add(
    mizu::opcode* pc,
    uint64_t* registers,
    uint8_t* stack_boundary,
    uint8_t* sp)
{
    registers[pc->out] = registers[pc->a]
        + registers[pc->b];
    MIZU_NEXT();
}
```

Listing 4.1: A demonstrative example of a *Mizu* instruction which takes a program counter (`pc`), array of `registers`, `stack_boundary`, and a pointer to the bottom (top in theoretical terms) of the stack (`sp`). Every instruction is expected to share this signature and to end with the `MIZU_NEXT` macro, which handles dispatching to the next instruction in the program.

Mizu targets a theoretical emulated 64-bit register machine, that is deeply influenced by `rv64im` architecture [28]. There are two main distinctions between this machine and other typical abstract machines:

1. First, *Mizu* does not provide a mechanism for a "data segment." Since *Mizu* is designed to be embedded in other applications, it assumes that these applications will supply pointers to already-allocated memory for it to manipulate.
2. Second, *Mizu* does not distinguish between register types. Rather than providing different types of registers for integers, floats, vectors, etc., all registers are treated as raw 64-bit blobs that instructions can interpret as necessary.

As a result, *Mizu* provides a huge 256 registers. It is currently assumed that pointers to raw data or function pointers within the executing program will be placed in higher-numbered registers, while temporary data and function arguments are placed in the lower-numbered registers. In addition to the registers, *Mizu* provides a stack with a configurable static size that functions can use. Both the registers and the stack are stored in the same "region" in memory, with registers counting upwards and the stack counting downwards. A pointer is used to mark the boundary for bounds-checking purposes.

4.3.1 Calling Convention

In the *Mizu* calling convention, registers do not differ based on the data stored in them, but they are assigned specific meanings. Register zero always holds the value zero, and its value is reset by the instruction dispatch machinery placed in a macro at the end of every instruction. Registers one through ten are designated as temporary registers, while registers twelve and above are used for passing arguments and storing host resources. Register eleven is reserved specifically as the return address register. Additionally, all registers are caller-saved, meaning that callee functions are free to modify any of the registers without the risk of overwriting important data.

4.3.2 Core Instruction Set

Mizu's Core Instruction Set handles jumps, branching, and integer operations. All forty-six of *Mizu*'s core instructions are summarized in Table 4.1. Typically, an instruction loads the values from its `a` and `b` registers and then stores the result of some computation in its `out` register.

Instruction	Description (C++)
label	`; // noop`[1]
find_label	`out = &matching_label`[1]
halt	`std::exit(0)`
debug_print(_binary)	`printf("%\n", a)`[2]
load_immediate	`out = immediate`[1]
load_upper_immediate	`out = immediate << 32`[13]
convert_to_u8|16|32|64	`out = (uint8|16|32|64_t)a`[4]
stack_load_u8|16|32|64	`out = (uint8|16|32|64_t)` stack_pointer[a][4]
stack_store_u8|16|32|64	`stack_pointer[b] = (uint8|16|32|64_t) a`[4]
stack_push(_immediate)	`stack_pointer -= a|immediate`[5]
stack_pop(_immediate)	`stack_pointer += a|immediate`[5]
jump_relative(_immediate)	`out = pc + 1; pc += a|immediate`[5]
jump_to	`out = pc + 1; pc = a`
branch_relative(_immediate)	`out = pc + 1; if(a) pc += b`[6]
branch_to	`out = pc + 1; if(a) pc = b`
set_if_equal	`out = a == b`
set_if_not_equal	`out = a != b`
set_if_less(_signed)	`out = a < b`
set_if_greater_equal(_signed)	`out = a >= b`
add	`out = a + b`
subtract	`out = a - b`
multiply	`out = a * b`
divide	`out = a / b`
modulus	`out = a % b`
shift_left	`out = a << b`
shift_right(_arithmetic)	`out = a >> b`[7]
bitwise_xor	`out = a ^ b`
bitwise_and	`out = a & b`
bitwise_or	`out = a | b`

Table 4.1: The Core *Mizu* Instructions.

[1] `a` and `b` treated as a single integer immediate value
[2] Prints the register in several formats, including binary if the instruction is used.
[3] Since immediates are only 32 bits, two instructions are needed to load a 64-bit immediate.
[4] The values are cast to the provided types. Stack accesses are bounds-checked in debug mode.
[5] Either `a` as a register or `a` and `b` treated as a single integer immediate value.

Two instructions that deviate from the typical pattern are `label` and `find_label`. Because *Mizu* programs are simple arrays, there is no guarantee that they will be located at a stable address in memory, especially if they are loaded from disk. Therefore, we cannot assume where in memory a specific jump location will be stored. To address this, the `label` instruction acts as a no-op target for the `find_label` instruction to locate. The `a` and `b` operands of both instructions are treated as a single 32-bit immediate integer. The `find_label` instruction searches through the program, scanning a configurable "maximum label search distance" number of instructions. If no label is found with matching `a` and `b` operands, it then performs a linear scan upwards, storing the address of the found `label` in `out`. Due to the runtime cost, these instructions should be invoked as infrequently as possible. In our example code, `find_label` instructions are placed at the start of the program. Following this pattern is not always possible; however, it's strongly recommended to avoid placing `find_label` inside loops (direct iteration or recursive).

Jumps and branches are also notable deviations. Jump instructions can take one of three types of operands: a register storing an offset, an immediate value (which is the combination of the `a` and `b` registers into a 32-bit integer), or a register holding the memory address of an instruction to jump to (obtained via the `find_label` instruction for example). These instructions then update the program counter, which points to the current `opcode`. These three jump instructions are respectively named `jump_relative`, `jump_relative_immediate`, and `jump_to`, depending on the type of operand they use. For relative jump instructions, offsets are given as relative values. A zero re-executes

[6] Or a signed immediate stored in place of `b`.
[7] The arithmetic version sign-extends the result.

the current instruction, negative one re-executes the previous, and positive two skips over one instruction and executes the next.

Jump-like instructions store a pointer to the next instruction in their `out` register. Function calls are thus easy, since they can be represented using the `jump_to` instruction, which jumps to the function and stores the next instruction in the `return_address` register, which can then be `jump_to`ed when the function should return.

There are three corresponding branch instructions that take a register storing a condition. This condition is directly passed into a C++ `if`-statement: if the condition register holds zero, the branch is not followed, but if it contains one or greater, the branch is followed. These branch instructions differ from RISC-V, which compares two registers before potentially jumping; instead, they behave more similarly to x86, where separate comparison instructions (such as `set_if_<something>` in *Mizu*) are used, and the result of these comparisons are then considered by the branch instructions. This design eliminates the need for jump instructions by using branch instructions with a non-zero condition register. Yet, we retained jump instructions to avoid the overhead of an `if`-statement and maintaining a non-zero register value.

4.3.3 Floating Point Extension Instructions

Mizu's built-in extensions include several instructions for floating-point operations and unsafe or host memory manipulation. The floating-point instructions have two variants: one for 32-bit IEEE 754 single-precision floating-point numbers [64], and another for 64-bit IEEE 754 double-precision floating-point numbers.

Note that all floating-point instructions assume both operands are represented using the same type. If the operands are of different types, the conversion instructions,

such as `convert_f32|64_to_f64|32` or `convert(_signed)_to_f32|64`, can correct the discrepancy. The instruction summary can be found in Table 4.2.

Instruction	Description (C++)
stack_load_f32|64	`out = (float) stack_pointer[a]`
stack_store_f32|64	`stack_pointer[a] = (float) b`
convert(*signed*)to_f32|64	`out = (float) a`
convert(*signed*)from_f32|64	`out = (uint64_t|int64_t) a`
convert_f32|64_to_f64|32	`(float)out = (double)a` `(double)out = (float)a`
add_f32|64	`out = a + b`
subtract_f32|64	`out = a - b`
multiply_f32|64	`out = a * b`
divide_f32|64	`out = a / b`
max_f32|64	`out = std::max(a, b)`
min_f32|64	`out = std::min(a, b)`
sqrt_f32|64	`out = std::sqrt(a)`
set_if_equal_f32|64	`out = a == b`
set_if_not_equal_f32|64	`out = a != b`
set_if_less_f32|64	`out = a < b`
set_if_greater_equal_f32|64	`out = a >= b`
set_if_negative_f32|64	`out = std::signbit(a)`
set_if_positive_f32|64	`out = !std::signbit(a)`
set_if_infinity_f32|64	`out = std::isinf(a)`
set_if_nan_f32|64	`out = std::isnan(a)`

Table 4.2: The Floating Point *Mizu* Instructions.

4.3.4 Unsafe Extension Instructions

Mizu integrates with a host application and assumes it will manipulate memory blocks provided by the host. It supports a small set of "unsafe" operations for direct memory manipulation, but a lack of checks leaves proper usage to the programmer.

The simplest of these instructions is `unsafe::allocate`, which creates a block of memory with a size specified in register `a` and stores a pointer to this memory in the `out` register. This allocated memory may be freed using the `unsafe::free_allocated` instruction, which also sets the register containing the freed pointer to the value stored in register `b`.

By default, `opcode` values are set to zero, so register `b` defaults to register zero, which always stores zero. This results in the freed pointer being swapped with null by default.

Before a pointer is freed, data can be copied into it using the `unsafe::copy_memory` instruction. This instruction takes a pointer to the destination memory (`out`), a pointer to the source memory (`a`), and the number of bytes to copy (`b`).

Copying host memory is only useful if it can be moved into a *Mizu* register or *Mizu*'s stack, which is why the `unsafe::pointer_to_stack` and `unsafe::pointer_to_register` instructions exist. These allow for the creation of a pointer that can be used with `unsafe::copy_memory`.

An example program snippet showing these operations in action—where one hundred 64-bit integers are copied from host memory to *Mizu* stack memory—can be found in Listing 4.2. A summary of all these instructions is provided in Table 4.3.

```
// x204 = sizeof(uint64_t)
opcode{load_immediate, 204}.set_immediate(sizeof(uint64_t)), // type size
constant
// a0 (size) = 100
opcode{load_immediate, registers::a(0)}.set_immediate(100),
// t0 = sizeof(uint64_t[100])
opcode{multiply, registers::t(0), 204, registers::a(0)}, // 204 ==
sizeof(uint64_t)
opcode{stack_push, 0, registers::t(0)},
// t1 = stack
opcode{unsafe::pointer_to_stack, registers::t(1)},
// t2 = host
opcode{load_immediate,
registers::t(2)}.set_host_pointer_lower_immediate(host_ptr),
opcode{load_upper_immediate,
registers::t(2)}.set_host_pointer_upper_immediate(host_ptr),
// std::memcpy(stack, host, sizeof(uint64_t[100]))
opcode{unsafe::copy_memory, registers::t(1), registers::t(2),
registers::t(0)},
```

Listing 4.2: Snippet from a *Mizu* program which reserves space on *Mizu*'s stack for 100 64-bit integers and then copies them from an arbitrary host pointer.

Instruction	**Description (C++)**
unsafe::allocate	`out = malloc(a)`
unsafe::free_allocated	`free(a); a = b`
unsafe::pointer_to_stack	`out = &stack_pointer[a]`
unsafe::pointer_to_register	`out = &a`
unsafe::copy_memory	`std::memcpy(out, a, b)`

Table 4.3: The Unsafe *Mizu* Instructions.

4.3.5 Parallel Extension Instructions

Mizu offers instructions for creating and coordinating program threads. It supports two primary methods: C++11 threading for cross-platform portability—a key reason *Mizu* uses C++ over more universally portable languages like C—and a custom coroutine-based emulation layer for platforms without hardware thread support. On resource-constrained platforms without hardware threading, the coroutine fallback uses a simple algorithm that runs one operation from each active thread before moving to the next.

When a new thread is forked, it inherits a copy of the parent thread's registers and stack, enabling inter-thread communication immediately. Each of the three types of fork instructions follows the same semantics as the corresponding jump instructions. The key difference is that the forked thread "jumps" to a new instruction while the parent continues to the next one.

When threads need to communicate post-creation, *Mizu* provides instructions to create and manipulate a communication channel. The channel must be created before the forked thread, ensuring that it is present in a register for both parent and child threads.

Channels support sending and receiving register-sized data blobs. When created, they allow specifying a maximum queue size. Threads attempting to send to a full channel or receive from an empty one will be blocked until another thread changes the channel's state.

For synchronization instructions, the coroutine fallback checks whether the necessary state is available. If not, it moves the waiting thread's program counter back by one instruction, causing the blocking operation to retry on the next activation.

A summary of these instructions is shown in Table 4.4.

Instruction	Description (C++)

fork_relative(_immediate)	`std::thread([env]{`
	`  env = env.clone();`
	`  start(env, pc + a | immediate);`
	`})`

fork_to	`std::thread([env]{`
	`  env = env.clone();`
	`  start(env, a);`
	`})`

join_thread	`a.join(); a = b`
sleep_microseconds	`std::thread::sleep_for(a)`
channel_create	`out = channel({.capacity = a})`
channel_close	`a.close(); a = b`
channel_receive	`out = a.receive()`
channel_send	`a.send(b)`

Table 4.4: The Parallel *Mizu* Instructions.

4.3.6 Foreign Function Extension Instructions

The final set of extensions in *Mizu* is focused on extensibility, though it introduces greater potential for unsafe behavior. Rather than embedding all features as instructions in the host application, *Mizu* enables dynamic loading of shared libraries and direct function calls.

Shared library loading is handled through a wrapper over platform-specific system calls. Function calls are executed using either *libffi* [83] or a custom trampoline system that supports up to four arguments. The trampoline fallback is used only when *libffi* is unavailable for a platform, making this extension the least portable among *Mizu*'s features.

Shared objects can be loaded using the `ffi::load_library` instruction or through `mizu::loader::load_library`. After loading, function pointers are retrieved via `ffi::load_library_function` or `mizu::loader::lookup`.

Function pointers are invoked similarly to *Mizu*'s internal calls: arguments are placed into registers, but instead of jumping to a code block, `ffi::call` or `ffi::call_with_return` is used. These instructions read the function address from register `a`, the interface description from register `b`, and call the function using the underlying ABI (via *libffi* or trampoline). If `ffi::call_with_return` is used, the return value is stored in `out`.

To prepare for a call, the function signature must first be defined using a sequence of `ffi::push_type_*` instructions. These define the return type (beginning with `ffi::push_type_void` for void) followed by the parameter types. After all types are specified, the `ffi::create_interface` instruction builds the interface object and places a pointer to it in `out`, clearing the temporary type stack.

A summary of these foreign function instructions is shown in Table 4.5.

Instruction	Description (C++)
ffi::push_type_void	`interface.push_back(void)`
ffi::push_type_pointer	`interface.push_back(void*)`
ffi::push_type_i32	`interface.push_back(int32_t)`
ffi::push_type_u32	`interface.push_back(uint32_t)`
ffi::push_type_i64	`interface.push_back(int64_t)`
ffi::push_type_u64	`interface.push_back(uint64_t)`
ffi::push_type_f32	`interface.push_back(float)`
ffi::push_type_f64	`interface.push_back(double)`
ffi::create_interface	`out = std::move(interface.create())`
ffi::call	`((b) a)(a0, a1, a2, ...)`
ffi::call_with_return	`out = ((b) a)(a0, a1, a2, ...)`
ffi::load_library	`out = dlopen(a)`[8]
ffi::load_library_function	`out = dlsym(a, b)`[8]

Table 4.5: The Foreign Function *Mizu* Instructions.

4.4 Benchmark Results

Designing a new interpreter is useful, but its effectiveness is limited without support for multiple platforms. To prove this, we conducted two benchmarks: one to compare the performance of *Mizu* against several other popular interpreters, and another to assess *Mizu*'s performance across all of its supported platforms.

For both tests, we used a custom-built x86 machine (Intel i7-13700K, 32GB DDR5-6400 RAM, Samsung 980 Pro 1TB SSD) running Ubuntu Linux. For the platform-specific test, we also utilized the same x86 machine running Windows and an M2 Pro 32GB Mac Mini.

To benchmark, we used Nanobench [81], a C++ library that runs a code snippet multiple times and averages the execution time. *Mizu* was linked into the runner code,

[8]The last parameter must store a pointer to a null-terminated C string.

while test programs—one recursively calculating the 40th Fibonacci number to strain a single CPU core, and another using bubble-sort to sort 1100 random integers, stressing memory access—were loaded from disk. This setup allowed for a fair comparison with other languages, which were also benchmarked using Nanobench via C's `system` function. The code for the benchmarks, along with additional detailed results that we do not have space to include here, are available on our GitHub [35].

The first benchmark's results are presented in Figure 4.1. *Mizu* runs an average of 4× slower than JIT interpreters; however, it outperforms regular interpreters by an average of 8.5× on the Recursive Fibonacci problem. In comparison to native code, *Mizu* is at least an order of magnitude slower. *Mizu* may even outperform JIT interpreters in bulk processing tasks involving large amounts of memory accesses (although the sample size is small). Overall, the results align closely with our expectations for a low-level, assembly-like interpreter.

The results from the second benchmark, shown in Figure 4.2, display a similar pattern across each platform for the two test programs, though the magnitude of the results differ. This suggests that *Mizu*'s performance is stable and not subject to significant fluctuations. It also demonstrates that *Mizu* runs on the web, though its performance there is not impressive.

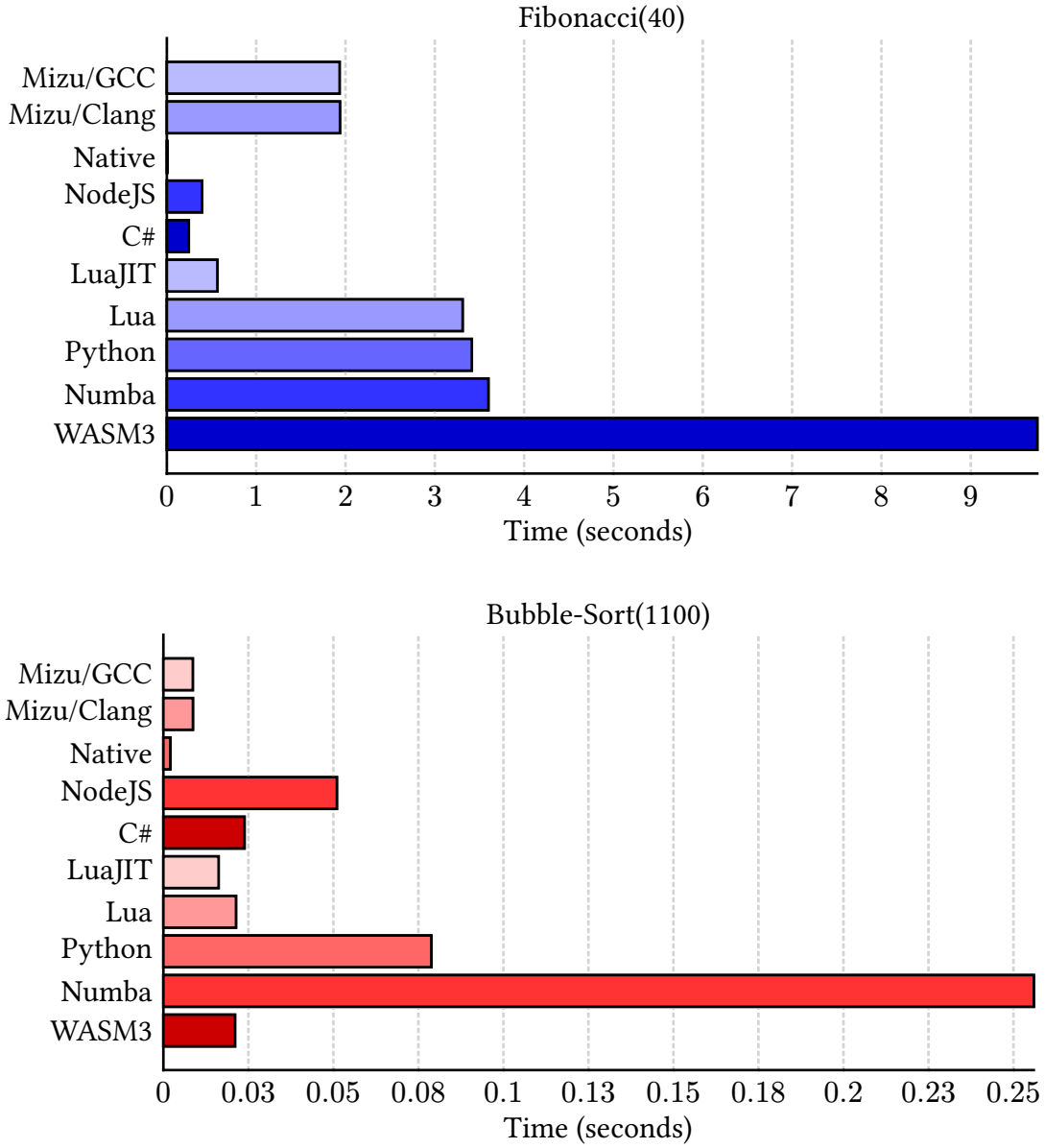

Figure 4.1: Comparison of the execution time, in seconds (lower is better), for *Mizu* (compiled with GCC and Clang) compared to a Native Executable, NodeJS [95], C# (DotNet8) [88], LuaJIT [98], Lua [65], Python [100], Numba [78], and WASM3 [124].

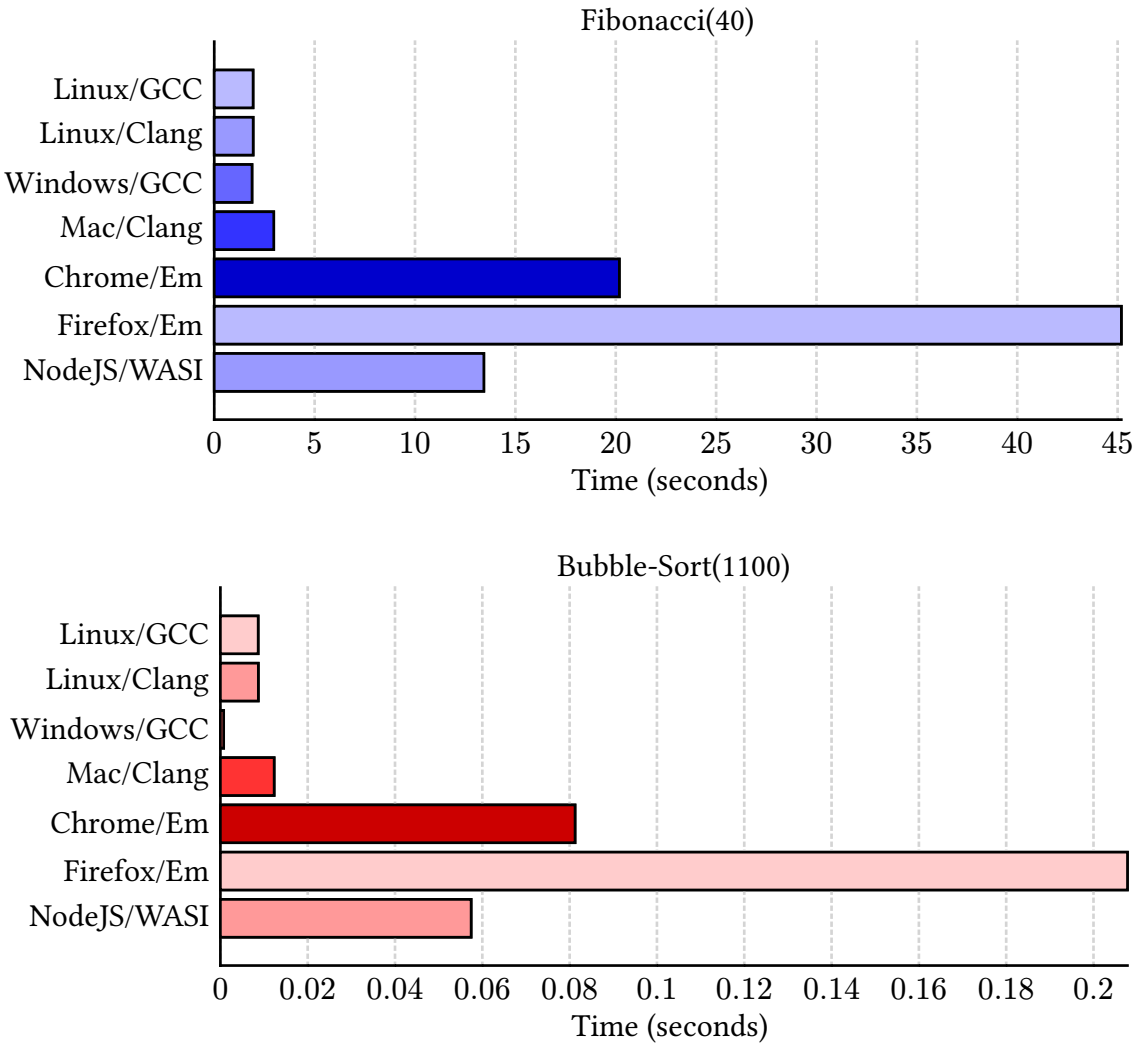

Figure 4.2: Comparison of the execution time, in seconds, for *Mizu* running on (from bottom to top) Linux (compiled with GCC), Linux (compiled with Clang), Windows (compiled with GCC), M2 Mac (compiled with Apple-Clang), Chrome on Linux (compiled with Emscripten [132]), Firefox on Linux (compiled with Emscripten), and NodeJS on Linux (compiled with WASI-SDK [125]).

These results imply that a low-level interpreter like *Mizu* can perform well compared to higher-level interpreters. More work is needed to further solidify this hypothesis.

4.5 Limitations & Future Work

Mizu relies on a threaded-code architecture, depending on compilers supporting tail call optimization. We added "almost" to the title of this paper because Microsoft's Visual C++ compiler does not seem to support tail call optimization. As a result, any non-trivial *Mizu* program running on that platform will quickly run out of stack space, causing a segmentation fault. This doesn't limit Windows support, as alternate compilers exist; however, we planned to demonstrate support for several microcontrollers, but discovered that `avr-gcc` (compiling for Arduino) also does not support tail-call optimization. This seems to be a well-known issue [123]. Similar problems on a RISC-V microcontroller are still being investigated.

Mizu does not provide an arbitrary program loader or define a portable file format. This limitation is acceptable for its intended use as an embedded submodule within a compiler. But if *Mizu* were to become an export target for these same compilers, providing such a loader and portable file format would become a core necessity. Because `opcodes` store a pointer, the binary representation of a *Mizu* program lacks portability across machines with different pointer sizes. While the foundation exists to address this, it is not robust beyond dumping test programs to a file and reloading them from disk.

4.6 Conclusion

In conclusion, *Mizu* has been designed as a lightweight, portable, and extendible interpreter that can be embedded within larger projects. We have demonstrated the simplicity of its instructions, the ease with which they can be extended, and the overall performance of the technique. Work needs to be done before *Mizu* can be considered

fully mature (especially in regard to its binary interface), but it is already performing comparably to its peers. We have also shown that it is possible to reduce higher-level concepts, such as threading and foreign function interfacing, to the level of an assembly language.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under grant numbers OIA-2019609 and OIA-2148788. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

5 Conclusions & Future Work

In addition to the material published about *Mizu* (see Chapter 4), we have developed a portable binary format that combines a *Mizu* program with raw binary data intended to be copied to the bottom of the program's stack. To support this functionality, we also introduced several new instructions that allow interaction with and referencing of pointers to the bottom of *Mizu*'s stack. These additions address what Chapter 4 identified as the most significant limitation of Mizu (the lack of usability as a compiler target). With this improvement in place, the primary challenges now lie in its overall immaturity and the lack of thorough testing.

Both of these issues could be significantly mitigated through a case study that integrates Mizu into a working compiler. To that end, the remainder of this chapter outlines our plans for the Data-Oriented Intermediate Representation (DOIR), a multilevel intermediate representation designed with built-in compile-time evaluation capabilities.

DOIR is intended to target the Mizu abstract machine and adopts a functional programming perspective. In DOIR, each line corresponds to a single assignment to a virtual register, using one of seven distinct forms: constants, blocks (which may contain nested operations and yield values), aliases, namespaces, types, and undefined field declarations. A basic example of each of these forms is illustrated in Listing 5.1.

Registers in DOIR may be explicitly numbered or named, and each assignment can include source location metadata. Functions are treated as first-class entities and are constructed in a manner similar to blocks. Their types combine explicit return and argument types, and they are capable of capturing registers from their enclosing scope.

The entire system is designed with minimalism in mind; in fact, the current draft of DOIR's grammar is compact enough to fit on a single page, as shown in Listing 5.2. This intermediate representation is built for flexibility in both code generation and transformation, emphasizing explicit control over scope, visibility, and the semantics of functions.

The plan for DOIR includes a mechanism analogous to β-reduction (see Section 2.3.3.1), in which function calls are progressively replaced with their corresponding bodies. This reduction continues until the program is transformed to a level where each function corresponds directly to a single machine language instruction. At that point, the result can be assembled into an executable binary.

```
%1 : i32 = 5 // #1 Constant assignment (name : type = value)
%2 : block = { // #2 Block assignment (%2 stores "quoted" information
about the contents of the block)
  %1 : i32 = 6
  _ : _ = yield(%1) // #3 Function execution (_ in a name will use the
next numbered register, _ in a type will request inference)
}
%3 : alias = %2 // #4 Alias assignment (%3 is resolved to %2)
math : namespace = { // #5 Namespace assignment (note that registers can
be named, not just numbered)
  vec2 : type = { // #6 Type assignment
    x : f32 // #7 Undefined assignment (invalid outside of types)
    y : f32
  }
}
```

Listing 5.1: A basic example of the seven forms DOIR will support.

```
program <- - assignment*
assignment <- Identifier- ':'- Type- '='- (Constant wsc / Block wsc /
function_call)? (SourceInfo wsc)? Terminator-

deducible_type <- Type- / 'deduced'- 'type'-
identifier_type_pair <- Identifier- ':'- deducible_type
FunctionType <- Identifier- '('- (identifier_type_pair ','-)* ')'
Type <- (FunctionType / Identifier)

Block <- '{'- assignment* '}'
function_call <- ('inline'-/'tail'-/'flatten'-)? Identifier- '('-
(FunctionType- / Identifier- / Block-)* ')'wsc

Terminator <- ';' / '\n' / '\n\r'
Identifier <- !Keywords [%a-zA-Z_][a-zA-Z0-9_.]*
Keywords <- 'deduced' / 'flatten' / 'inline' / 'tail'
SourceInfo <- '<' ((!':' .)* ':')? IntegerConstant ('-' IntegerConstant)?
':' IntegerConstant ('-' IntegerConstant)? '>'

- <- ([ \n\r\t] / LongComment / LineComment)*
wsc <- ([ \t] / LongComment / LineComment)*
LongComment <- '/*' (!'*/'.)* '*/'
LineComment <- '//' (!'\n' .)*
```

Listing 5.2: The current draft Parsing Expression Grammar (PEG) [48] for DOIR. PEGs are similar to other standard grammar notations, such as BNF, except that they define a scanner (tokens) in addition to grammar productions, which are marked by a trailing arrow (<-), alternatives are delimited with a forward slash (/), and quoted strings or regular expressions define tokens. Additionally, they are inherently unambiguous: each alternative of a production is tried, and if it fails, the next one is tried, or in other words, ambiguity resolution is defined by which production is listed first.

In this sense, DOIR can be viewed as a kind of "better assembler"—one that offers higher-level abstractions without sacrificing control or efficiency. To evaluate and quantify what "better" truly means in this context, we intend to conduct a series of user studies.

This thesis presents the majority of the core components we intend to use in the implementation of DOIR. At the heart of the system is the Entity-Component-System defined in Chapter 3, which serves as its structural and, to a lesser extent, performance backbone. Mizu (see Chapter 4), the first "assembly" language we plan to target, will play a key role by providing a foundation for low-level execution along with some support for compile-time evaluation. The next phase of the project involves building a parser for the designed language and integrating it with a Mizu-targeting backend.

Our initial focus will be on developing a single-function version of DOIR to refine the "better assembly" syntax and ensure a solid foundation before implementing β-reduction. Once that core functionality is in place, we plan to add support for RISC-V assembly. This step should be relatively straightforward, particularly because Mizu was designed with RISC-V as a reference model.

Following that, the project will progress toward building multiple layers of reductions that incrementally map representations closely aligned with C down to lower-level constructs. Once DOIR can support the most common features of C, we will gain access to the rich ecosystem of benchmarks developed for the C language. This will allow us to rigorously evaluate our system by comparing it against established compiler toolchains such as GCC [51] and LLVM [79].

This is the limit of the current development plan, and thus, future directions become more speculative. One possible avenue for future development would be to

support additional machine architectures, such as x86 (which is likely a necessity for any future adoption) and Arm, as well as provide options for transpiling DOIR code to higher-level targets like C or WebAssembly [124].

The end goal of this phase is a small, lightweight compiler infrastructure with a stable, fixed API that defers many implementation details to compile time. This lightweight core should be capable of consuming backends as libraries, and significant design work is being put towards ensuring these "backend libraries" are as easy to write as possible.

An earlier iteration of the DOIR design incorporated several parallelism primitives inspired by those found in Verse [44]. Given DOIR's functional nature, an intriguing possibility is the development of optimization passes that transform side effects into pure argument-based manipulation. This could potentially allow us to leverage existing research on the automatic parallelization of functional languages.

Looking further ahead, it may be feasible to build additional abstraction layers stacked on top of the C layer, enabling support for other programming languages such as Go, Haskell, or even Datalog (hopefully all also delivered as libraries). These ideas remain exploratory, but they are all promising directions for future work.

References

[1] Aho, A.V., Lam, M.S., Sethi, R. and Ullman, J.D. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley. ISBN: 9780321486813.

[2] Allen, F.E. 1970. Control flow analysis. *Proceedings of a Symposium on Compiler Optimization*. Association for Computing Machinery. 1–19. https://doi.org/10.1145/800028.808479.

[3] Alonzo Church 1941. *The calculi of lambda conversion*. Princeton University Press. ISBN: 978-0691083940.

[4] Amighi, A., C. Gomes, P. de, Gurov, D. and Huisman, M. 2012. Sound Control-Flow Graph Extraction for Java Programs with Exceptions. *Software Engineering and Formal Methods*. G. Eleftherakis, M. Hinchey, and M. Holcombe, eds. Springer Berlin Heidelberg. 33–47. https://doi.org/10.1007/978-3-642-33826-7_3.

[5] Anderson, T. and Mattson, T. 2021. Multithreaded parallel Python through OpenMP support in Numba. *Proceedings of the 20th Python in Science Conference*. M. Agarwal, C. Calloway, D. Niederhut, and D. Shupe, eds. 140–147. https://doi.org/ 10.25080/majora-1b6fd038-012 .

[6] Atari 2021. Asteroids: Recharged. *https://atari.com/products/asteroids-recharged*. Accessed: 2025-06-22.

[7] Bachmann, M. 2021. RapidFuzz: Fuzzy String Matching in Python. *https://github.com/rapidfuzz/RapidFuzz*. Accessed: 2025-07-08.

[8] Basili, V.R., Caldiera, G. and Rombach, H.D. 1997. The Goal Question Metric Approach. *Software Engineering Journal*. 27, 8 (1997), 983–1001. https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-\#.

[9] Baxter, W. and Bauer, H.R. 1989. The program dependence graph and vectorization. *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery. 1–11. https://doi.org/10.1145/75277.75278.

[10] Berndl, M., Vitale, B., Zaleski, M. and Brown, A. 2005. Context threading: a flexible and efficient dispatch technique for virtual machine interpreters.

International Symposium on Code Generation and Optimization. 15–26. https://doi.org/10.1109/CGO.2005.14.

[11] Bevy-Contributors 2025. Bevy Engine. *https://github.com/bevyengine/bevy/releases/tag/v0.10.0*. Accessed: 2025-04-15.

[12] Boissinot, B., Darte, A., Rastello, F., Dinechin, B.D. de and Guillon, C. 2009. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. *2009 International Symposium on Code Generation and Optimization*. 114–125. https://doi.org/10.1109/CGO.2009.19.

[13] Briggs, P., Cooper, K.D. and Torczon, L. 1992. Rematerialization. *SIGPLAN Not.* 27, 7 (Jul. 1992), 311–321. https://doi.org/10.1145/143103.143143.

[14] Carbon Developers 2025. Carbon Language: An experimental successor to C++. *https://github.com/carbon-language/carbon-lang*. Accessed: 2025-04-15.

[15] Cardama, F.J., Vázquez-Pérez, J., Piñeiro, C., Pichel, J.C., Pena, T.F. and Gómez, A. 2025. Review of intermediate representations for quantum computing. *The Journal of Supercomputing*. 81, 2 (Jan. 2025), 418. https://doi.org/10.1007/s11227-024-06892-2.

[16] Carlton, B. 2011. Answer to: Most difficult subject/theory in Computer Science?. *https://softwareengineering.stackexchange.com/a/41674*. Accessed: 2025-04-15.

[17] Carruth, C. 2023. Modernizing Compiler Design for Carbon Toolchain (lecture). *https://www.youtube.com/watch?v=ZI198eFghJk*. Accessed: 2025-04-15.

[18] Carruth, C. 2023. Modernizing Compiler Design for Carbon's Toolchain (slides). *https://chandlerc.blog/slides/2023-cppnow-compiler*. Accessed: 2025-04-15.

[19] Click, C. and Paleczny, M. 1995. A simple graph-based intermediate representation. *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*. Association for Computing Machinery. 35–49. https://doi.org/10.1145/202529.202534.

[20] Cloutier, T. 2023. Databases are the endgame for data-oriented design. *https://spacetimedb.com/blog/databases-and-data-oriented-design*. Accessed: 2025-04-24.

[21] Colson, D. 2020. How to Make a Simple Entity-Component-System in C++. *https://www.david-colson.com/2020/02/09/making-a-simple-ecs.html*. Accessed: 2025-04-15.

[22] Cooper, K.D. and Torczon, L. 2022. *Engineering a Compiler*. Morgan Kaufmann. ISBN: 9780128154120.

[23] Cooper, K.D., Simpson, L.T. and Vick, C.A. 2001. Operator strength reduction. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sep. 2001), 603–625. https://doi.org/10.1145/504709.504710.

[24] cppreference 2025. constexpr specifier - cppreference.com. *https://en.cppreference.com/w/cpp/language/constexpr*. Accessed: 2025-07-08.

[25] cppreference 2025. std::sort - cppreference.com. *https://en.cppreference.com/w/cpp/algorithm/sort*. Accessed: 2025-04-15.

[26] Crockford, D. 2025. Introducing JSON. *https://www.json.org/json-en.html*. Accessed: 2025-04-15.

[27] Crossref 2025. Crossref REST API Documentation. *https://www.crossref.org/documentation/retrieve-metadata/rest-api/*. Accessed: 2025-07-09.

[28] Cui, E., Li, T. and Wei, Q. 2023. RISC-V Instruction Set Architecture Extensions: A Survey. *IEEE Access*. 11, (2023), 24696–24711. https://doi.org/10.1109/ACCESS.2023.3246491.

[29] Curley, C. 1993. Life in the FastForth lane. *https://www.forth.org/fd/curley2.html*. Accessed: 2025-08-07.

[30] Cyphers, S., Bansal, A.K., Bhiwandiwalla, A., Bobba, J., Brookhart, M., Chakraborty, A., Constable, W., Convey, C., Cook, L., Kanawi, O., Kimball, R., Knight, J., Korovaiko, N., Kumar, V., Lao, Y., Lishka, C.R., Menon, J., Myers, J., Narayana, S.A., Procter, A. and Webb, T.J. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. (2018). https://doi.org/10.48550/arXiv.1801.08058.

[31] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320.

[32] Cytron, R., Lowry, A. and Zadeck, F.K. 1986. Code motion of control structures in high-level languages. *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery. 70–85. https://doi.org/10.1145/512644.512651.

[33] Dahl, J. 2025. benchmark_lox.cpp.bench - Benchmark Test from the DOIR Project. *https://github.com/data-oriented-ir/DOIR/blob/benchmark/tests/benchmark_lox.cpp.bench*. Accessed: 2025-04-24.

[34] Dahl, J. 2025. Mizu Source Code. *https://github.com/joshuadahlunr/MizuVM/tree/paper*. Accessed: 2025-07-08. https://doi.org/10.5281/zenodo.15578022.

[35] Dahl, J. 2025. MizuBenchmark – Cross-Language Benchmarking Suite. *https://github.com/doir-lang/MizuBenchmark*. Accessed: 2025-07-08.

[36] Dahl, J. 2025. Platform Executable Startup Cost. *https://gist.github.com/joshuadahlunr/fa99761edcfbdaaf4aec8ef93396ef42*. Accessed: 2025-06-27.

[37] Dahl, J. 2025. serialize.cpp - Part of the DOIR Project. *https://github.com/data-oriented-ir/DOIR/blob/ECS4Compilers/src/Lox/serialize.cpp*. Accessed: 2025-04-24.

[38] Dahl, J. and Harris Jr., F.C. 2025. An Argument for the Practicality of Entity Component Systems as the Primary Data Structure for an Interpreter or Compiler. *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Association for Computing Machinery.

[39] Dahl, J., Contaldi, Q., Partovi, K. and Harris Jr., F.C. Mizu: A Lightweight Multi-Threaded Threaded-Code Interpreter that Can Run Almost Anywhere with a C++ Compiler. *The 23rd IEEE/ACIS International Conference on Software Engineering, Management and Applications*.

[40] Dennis, J.B. 1980. Data Flow Supercomputers. *Computer*. 13, 11 (1980), 48–56. https://doi.org/10.1109/MC.1980.1653418.

[41] Dennis, J.B. 1974. First version of a data flow procedure language. *Programming Symposium*. B. Robinet, ed. Springer Berlin Heidelberg. 362–376. ISBN: 978-3-540-37819-8.

[42] Ebner, D., Brandner, F., Scholz, B., Krall, A., Wiedermann, P. and Kadlec, A. 2008. Generalized instruction selection using SSA-graphs. *SIGPLAN Not.* 43, 7 (Jun. 2008), 31–40. https://doi.org/10.1145/1379023.1375663.

[43] Epic Games 2022. Large Numbers of Entities with Mass in Unreal Engine 5. *https://dev.epicgames.com/community/learning/talks-and-demos/37Oz/large-numbers-of-entities-with-mass-in-unreal-engine-5*. Accessed: 2025-04-15.

[44] Epic Games 2025. Verse Language Reference. *https://dev.epicgames.com/documentation/en-us/fortnite/verse-language-reference*. Accessed: 2025-07-16.

[45] Erosa, A. and Hendren, L. 1994. Taming control flow: a structured approach to eliminating goto statements. *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*. 229–240. https://doi.org/10.1109/ICCL.1994.288377.

[46] Ferrante, J. and Mace, M. 1985. On linearizing parallel code. *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery. 179–190. https://doi.org/10.1145/318593.318636.

[47] Ferrante, J., Ottenstein, K.J. and Warren, J.D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (Jul. 1987), 319–349. https://doi.org/10.1145/24039.24041.

[48] Ford, B. 2004. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.* 39, 1 (Jan. 2004), 111–122. https://doi.org/10.1145/982962.964011.

[49] Fraser, C.W. 1991. A retargetable compiler for ANSI C. *SIGPLAN Not.* 26, 10 (Oct. 1991), 29–43. https://doi.org/10.1145/122616.122621.

[50] Fraser, C.W. and Hanson, D.R. 1995. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 0805316701.

[51] Free Software Foundation 2025. GNU Compiler Collection (GCC). *https://gcc.gnu.org/*. Accessed: 2025-05-15.

[52] Gaede, Frank, Hegner, Benedikt and Stewart, Graeme A. 2020. PODIO: recent developments in the Plain Old Data EDM toolkit. *EPJ Web Conf.* 245, (2020), 5024. https://doi.org/10.1051/epjconf/202024505024.

[53] Gerlek, M.P., Stoltz, E. and Wolfe, M. 1995. Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Program. Lang. Syst.* 17, 1 (Jan. 1995), 85–122. https://doi.org/10.1145/200994.201003.

[54] Girard, J.-Y., Lafont, Y. and Regnier, L. 1995. Proof nets. *Advances in Linear Logic*. Cambridge University Press. ISBN: 978-1316702291.

[55] Gosling, J. 1995. Java intermediate bytecodes: ACM SIGPLAN workshop on intermediate representations (IR'95). *SIGPLAN Not.* 30, 3 (Mar. 1995), 111–118. https://doi.org/10.1145/202530.202541.

[56] Grosser, T., Groesslinger, A. and Lengauer, C. 2012. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*. 22, 4 (2012), 1250010. https://doi.org/10.1142/S0129626412500107.

[57] Guinness World Records 2004. Fastest realtime court reporter (stenotype writing). *https://www.guinnessworldrecords.com/world-records/fastest-realtime-court-reporter-%28stenotype-writing%29*. Accessed: 2025-04-15.

[58] Hasabnis, N. and Sekar, R. 2016. Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers. *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery. 311–324. https://doi.org/10.1145/2872362.2872380.

[59] Hassan, A., Mackie, I. and Sato, S. 2015. An Implementation Model for Interaction Nets. *Electronic Proceedings in Theoretical Computer Science*. 183, (May 2015), 66–80. https://cgi.cse.unsw.edu.au/~eptcs/content.cgi?TERMGRAPH2014. Accessed: 2025-08-08. A PDF version can be found at: https://doi.org/10.4204/eptcs.183.5.

[60] Havlak, P. 1994. Construction of thinned gated single-assignment form. *Languages and Compilers for Parallel Computing*. U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds. Springer Berlin Heidelberg. 477–499. https://doi.org/10.1007/3-540-57659-2_28.

[61] Herlihy, M., Shavit, N. and Tzafrir, M. 2008. Hopscotch Hashing. *Distributed Computing*. G. Taubenfeld, ed. Springer Berlin Heidelberg. 350–364. https://doi.org/10.1007/978-3-540-87779-0_24.

[62] HigherOrderCO 2025. HVM: A Virtual Machine for Interaction Nets. *https://github.com/HigherOrderCO/HVM*. Accessed: 2025-07-07.

[63] Härkönen, T. 2019. Advantages and Implementation of Entity-Component-Systems. *https://trepo.tuni.fi/handle/123456789/27593*. Accessed: 2025-04-15. Bachelor's thesis. Tampere University, Tampere, Finland.

[64] IEEE 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*. 0, (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229.

[65] Ierusalimschy, R., Figueiredo, L.H. de and Filho, W.C. 1996. Lua—An Extensible Extension Language. *Software: Practice and Experience*. 26, 6 (1996), 635–652. https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P.

[66] Johnson, N. and Mycroft, A. 2003. Combined Code Motion and Register Allocation Using the Value State Dependence Graph. *Compiler Construction*. G. Hedin, ed. Springer Berlin Heidelberg. 1–16. https://doi.org/10.1007/3-540-36579-6_1.

[67] Johnson, N. and Mycroft, A. 2004. Using Multiple Memory Access Instructions for Reducing Code Size. *Compiler Construction*. E. Duesterwald, ed. Springer Berlin Heidelberg. 265–280. https://doi.org/10.1007/978-3-540-24723-4_18.

[68] Johnson, N.E. 2004. *Code size optimization for embedded processors*. https://doi.org/10.48456/tr-607 Technical Report Number: UCAM-CL-TR-607, University of Cambridge, Computer Laboratory, Cambridge, United Kingdom.

[69] Johnson, R. and Pingali, K. 1993. Dependence-based program analysis. *SIGPLAN Not.* 28, 6 (Jun. 1993), 78–89. https://doi.org/10.1145/173262.155098.

[70] Johnson, R., Pearson, D. and Pingali, K. 1994. The program structure tree: computing control regions in linear time. *SIGPLAN Not.* 29, 6 (Jun. 1994), 171–185. https://doi.org/10.1145/773473.178258.

[71] Johnsson, T. 1985. Lambda lifting: Transforming programs to recursive equations. *Functional Programming Languages and Computer Architecture*. J.-P. Jouannaud,

ed. Springer Berlin Heidelberg. 190–203. https://doi.org/10.1007/3-540-15975-4_37.

[72] Kelsey, R.A. 1995. A correspondence between continuation passing style and static single assignment form. *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*. Association for Computing Machinery. 13–22. https://doi.org/10.1145/202529.202532.

[73] Kirchner, K. and Rosenthaler, S. 2017. bin2llvm: Analysis of Binary Programs Using LLVM Intermediate Representation. *Proceedings of the 12th International Conference on Availability, Reliability and Security*. Association for Computing Machinery. https://doi.org/10.1145/3098954.3103152.

[74] Knill, O. 2014. Ambiguity in Mathematical Notation. *https://people.math.harvard.edu/~knill/pedagogy/ambiguity/index.html*. Accessed: 2025-07-09.

[75] Kunft, A., Katsifodimos, A., Schelter, S., Bress, S., Rabl, T. and Markl, V. 2019. An intermediate representation for optimizing machine learning pipelines. *Proc. VLDB Endow.* 12, 11 (Jul. 2019), 1553–1567. https://doi.org/10.14778/3342263.3342633.

[76] Lafont, Y. 1997. Interaction Combinators. *Information and Computation*. 137, 1 (1997), 69–101. https://doi.org/https://doi.org/10.1006/inco.1997.2643.

[77] Lafont, Y. 1989. Interaction nets. *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery. 95–108. https://doi.org/10.1145/96709.96718.

[78] Lam, S.K., Pitrou, A. and Seibert, S. 2015. Numba: a LLVM-based Python JIT compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. Association for Computing Machinery. https://doi.org/10.1145/2833157.2833162.

[79] Lattner, C. and Adve, V. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665.

[80] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N. and Zinenko, O. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. *2021 IEEE/ACM International*

Symposium on Code Generation and Optimization (CGO). 2–14. https://doi.org/10.1109/CGO51591.2021.9370308.

[81] Leitner-Ankerl, M. 2023. ankerl::nanobench: Simple, Fast, Accurate Single-Header Microbenchmarking for C++. *https://nanobench.ankerl.com/*. Accessed: 2025-04-15.

[82] Leitner-Ankerl, M. 2016. Very Fast HashMap in C++: Benchmark Results (Part 3). *https://martin.ankerl.com/2016/09/21/very-fast-hashmap-in-c-part-3/*. Accessed: 2025-04-15.

[83] libffi Contributors 2025. libffi - Portable Foreign Function Interface Library. *https://github.com/libffi/libffi*. Accessed: 2025-07-08.

[84] Lindholm, T., Yellin, F., Bracha, G. and Buckley, A. 2013. *The Java Virtual Machine Specification, Java SE 7 Edition*. Pearson Education. ISBN: 9780133260465.

[85] Liu, X., Yin, W., Yin, Q. and Jiang, L. 2010. A SSA-based intermediate representation technique. *2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering*. 98–101. https://doi.org/10.1109/CMCE.2010.5609916.

[86] Madnight 2024. GitHut 2.0 – GitHub Language Statistics (Q1 2024). *https://madnight.github.io/githut/#/pushes/2024/1*. Accessed: 2025-07-08.

[87] Merrill, J. 2003. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. *https://ftp.cygwin.com/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf*. Accessed: 2025-07-09.

[88] Microsoft 2025..NET (dotnet.microsoft.com) – Free · Cross-Platform · Open Source Developer Platform. *https://dotnet.microsoft.com/*. Accessed: 2025-07-08.

[89] Might, M. 2011. Continuation Passing Style (CPS) Conversion. *https://matt.might.net/articles/cps-conversion/*. Accessed: 2025-07-16.

[90] Mrena, M., Varga, M. and Kvassay, M. 2022. Experimental Comparison of Array-based and Linked-based List Implementations. *2022 IEEE 16th International Scientific Conference on Informatics (Informatics)*. 231–238. https://doi.org/10.1109/Informatics57926.2022.10083495.

[91] Noll, L.C. 2009. FNV Hash Function. *http://www.isthe.com/chongo/tech/comp/fnv/*. Accessed: 2025-04-15.

[92] Nystrom, B. 2025. Benchmark Test from the Crafting Interpreters Repository. *https://github.com/munificent/craftinginterpreters/blob/master/test/*. Accessed: 2025-04-24.

[93] Nystrom, R. 2021. *Crafting interpreters*. Genever Benning. ISBN: 978-0990582939.

[94] ONNX Community 2019. Open Neural Network Exchange (ONNX). *https://onnx.ai/*. Accessed: 2025-07-07.

[95] OpenJS Foundation 2025. Node.js – JavaScript Runtime. *https://nodejs.org/en*. Accessed: 2025-07-08.

[96] Ostrovsky, I. 2010. Gallery of Processor Cache Effects. *https://igoro.com/archive/gallery-of-processor-cache-effects/*. Accessed: 2025-04-15.

[97] Ottenstein, K.J., Ballance, R.A. and MacCabe, A.B. 1990. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. *SIGPLAN Not.* 25, 6 (Jun. 1990), 257–271. https://doi.org/10.1145/93548.93578.

[98] Pall, M. and LuaJIT Project 2025. LuaJIT – Just-in-Time Compiler for Lua. *https://luajit.org/*. Accessed: 2025-07-08.

[99] Pingali, K., Beck, M., Johnson, R., Moudgill, M. and Stodghill, P. 1991. Dependence flow graphs: an algebraic approach to program dependencies. *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery. 67–78. https://doi.org/10.1145/99583.99595.

[100] Python Software Foundation 2025. CPython: The Python Programming Language. *https://github.com/python/cpython*. Accessed: 2025-04-17.

[101] Ramsey, N. 2022. Beyond Relooper: recursive translation of unstructured control flow to structured control flow (functional pearl). *Proc. ACM Program. Lang.* 6, 90 (Aug. 2022). https://doi.org/10.1145/3547621.

[102] Rosen, B.K., Wegman, M.N. and Zadeck, F.K. 1988. Global value numbers and redundant computations. *Proceedings of the 15th ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages. Association for Computing Machinery. 12–27. https://doi.org/10.1145/73560.73562.

[103] The Rustc Dev Guide Team 2024. Part 3: Middle-End and Back-End. *https://rustc-dev-guide.rust-lang.org/part-3-intro.html*. Accessed: 2025-07-16.

[104] Sahasrabuddhe, S.D., Raja, H., Arya, K. and Desai, M.P. 2007. AHIR: A Hardware Intermediate Representation for Hardware Generation from High-level Programs. *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*. 245–250. https://doi.org/10.1109/VLSID.2007.28.

[105] Santhanam, K., Krishna, S., Tomioka, R., Fitzgibbon, A. and Harris, T. 2021. DistIR: An Intermediate Representation for Optimizing Distributed Neural Networks. *Proceedings of the 1st Workshop on Machine Learning and Systems*. Association for Computing Machinery. 15–23. https://doi.org/10.1145/3437984.3458829.

[106] Sarkar, V. 1991. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*. 35, 5.6 (1991), 779–804. https://doi.org/10.1147/rd.355.0779.

[107] Schäfer, S. and Scholz, B. 2007. Optimal chain rule placement for instruction selection based on SSA graphs. *Proceedingsof the 10th International Workshop on Software & Compilers for Embedded Systems*. Association for Computing Machinery. 91–100. https://doi.org/10.1145/1269843.1269857.

[108] Sharir, M. 1980. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*. 5, 3 (1980), 141–153. https://doi.org/10.1016/0096-0551(80)90007-7.

[109] Stanier, J. 2023. Removing and Restoring Control Flow with the Value State Dependence Graph. *https://sussex.figshare.com/articles/thesis/Removing_and_restoring_control_flow_with_the_Value_State_Dependence_Graph/23317190*. Accessed: 2025-07-07. Doctoral thesis. University of Sussex, Sussex, United Kingdom.

[110] Stanier, J. and Watson, D. 2013. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.* 45, 3 (Jul. 2013). https://doi.org/10.1145/2480741.2480743.

[111] Stauffer, J.D. 1978. LCL: a compiler and language for logical mask checking. *https://www.osti.gov/biblio/6758481*. Accessed: 2025-08-07. Technical Report Number: 6758481, Sandia Labs., Albuquerque, N.Mex. (USA).

[112] Steensgaard, B. 1993. Sequentializing program dependence graphs for irreducible programs. *https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a0d78dc17daca1c56e6c999ad27351d291d09754*. Accessed: 2025-07-07. Technical Report Number: MSR-TR-93-14, Microsoft Research, Redmond, Washington, United States.

[113] Stepp, M., Tate, R. and Lerner, S. 2011. Equality-Based Translation Validator for LLVM. *Computer Aided Verification*. G. Gopalakrishnan and S. Qadeer, eds. Springer Berlin Heidelberg. 737–742. https://doi.org/10.1007/978-3-642-22110-1_59.

[114] Stoltz, Gerlek and Wolfe 1994. Extended SSA with factored use-def chains to support optimization and parallelism. *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*. 43–52. https://doi.org/10.1109/HICSS.1994.323280.

[115] Sussman, G.J. and Steele, G.L. 1998. Scheme: An Interpreter for Extended Lambda Calculus. *Higher-Order and Symbolic Computation*. 11, 4 (1998), 405–439. https://doi.org/10.1023/A:1010035624696.

[116] Tarjan, R.E. 1981. A Unified Approach to Path Problems. *J. ACM*. 28, 3 (Jul. 1981), 577–593. https://doi.org/10.1145/322261.322272.

[117] Tarjan, R.E. 1979. Applications of Path Compression on Balanced Trees. *J. ACM*. 26, 4 (Oct. 1979), 690–715. https://doi.org/10.1145/322154.322161.

[118] Tate, R., Stepp, M., Tatlock, Z. and Lerner, S. 2009. Equality saturation: a new approach to optimization. *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery. 264–276. https://doi.org/10.1145/1480881.1480915.

[119] TIOBE Software BV 2025. TIOBE Index for July 2025. *https://web.archive.org/web/20250709134336/https://www.tiobe.com/tiobe-index/*. Accessed: 2025-07-08.

[120] Tristan, J.-B., Govereau, P. and Morrisett, G. 2011. Evaluating value-graph translation validation for LLVM. *Proceedings of the 32nd ACM SIGPLAN*

Conference on Programming Language Design and Implementation. Association for Computing Machinery. 295–305. https://doi.org/10.1145/1993498.1993533.

[121] Tu, P. and Padua, D. 1995. Efficient building and placing of gating functions. *SIGPLAN Not.* 30, 6 (Jun. 1995), 47–55. https://doi.org/10.1145/223428.207115.

[122] Unity Technologies 2023. Unity's Data-Oriented Technology Stack (DOTS). *https://unity.com/dots*. Accessed: 2025-04-15.

[123] wasm3 contributors 2025. Wasm3 Performance — "Wasm3 on MCUs" Section. *https://github.com/wasm3/wasm3/blob/main/docs/Performance.md#wasm3-on-mcus*. Accessed: 2025-07-08.

[124] wasm3 contributors 2025. wasm3 – High-performance WebAssembly Runtime. *https://github.com/wasm3/wasm3*. Accessed: 2025-07-08.

[125] WebAssembly Community Group 2025. WASI-SDK – WebAssembly System Interface SDK. *https://github.com/WebAssembly/wasi-sdk*. Accessed: 2025-07-08.

[126] Wegman, M.N. and Zadeck, F.K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (Apr. 1991), 181–210. https://doi.org/10.1145/103135.103136.

[127] Weise, D., Conybeare, R., Ruf, E. and Seligman, S. 1991. Automatic online partial evaluation. *Functional Programming Languages and Computer Architecture*. J. Hughes, ed. Springer Berlin Heidelberg. 165–191. https://doi.org/10.5555/645420.652530.

[128] Weise, D., Crew, R.F., Ernst, M. and Steensgaard, B. 1994. Value dependence graphs: representation without taxation. *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery. 297–310. https://doi.org/10.1145/174675.177907.

[129] Wihuri, A. 2023. A case study on parallelism and multithreading in a meteorological JavaScript web application. *https://aaltodoc.aalto.fi/items/71feb1c5-fa75-4af9-a5d0-6afaf34659d8*. Accessed: 2025-07-08.

[130] Wolfe, M. 1992. Beyond induction variables. *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. Association for Computing Machinery. 162–174. https://doi.org/10.1145/143095.143131.

[131] Yang, W., Horwitz, S. and Reps, T. 1989. Detecting Program Components With Equivalent Behaviors. *https://minds.wisconsin.edu/handle/1793/59110*. Accessed: 2025-07-09.

[132] Zakai, A. 2011. Emscripten: an LLVM-to-JavaScript compiler. *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. Association for Computing Machinery. 301–312. https://doi.org/10.1145/2048147.2048224.

[133] Zig Software Foundation 2025. Zig Documentation: comptime. *https://ziglang.org/documentation/master/#comptime*. Accessed: 2025-07-08.

Appendices

A A Full Tree Representation

Below is the AST generated for the Lox code in Listing 3.11. Child nodes are indented one level beneath their parents. Each node ends with two numbers in parentheses: the first is the node's Entity ID, and the second is the total number of children it has (both direct and indirect). Some nodes also include three numbers in square brackets after the parentheses. These represent the attached 3AC: the result and two operands of the operation (a 0 indicates the absence of an operand). Nodes are stored in memory in ascending order of their Entity IDs. Notice that a reverse iteration starting at Entity 24 corresponds to the execution of the program. Similarly, a reverse iteration that skips nodes without attached 3AC also corresponds to an execution.

```
{ (1, 23)
  declare:var:x (24, 0)
    assign: (21, 2)
        var:x -> 24 (22, 0)
    =
        5 (23, 0) [23, 0, 0]
    declare:var:y (20, 0)
    assign: (17, 2)
        var:y -> 20 (18, 0)
    =
        6 (19, 0) [19, 0, 0]
    declare:var:z (16, 0)
    assign: (13, 2)
        var:z -> 16 (14, 0)
```

```
    =
        7 (15, 0) [15, 0, 0]
    print (3, 9)
        + [add] (4, 8) [4, 23, 6]
            var:x -> 24 (5, 0)
            / [divide] (6, 6) [6, 7, 10]
                * [multiply] (7, 2) [7, 19, 15]
                    var:y -> 20 (8, 0)
                    var:z -> 16 (9, 0)
                - [subtract] (10, 2) [10, 15, 19]
                    var:z -> 16 (11, 0)
                    var:y -> 20 (12, 0)

    declare:fun:clock (2, 0) {} // A builtin function
}
```

B Publication List

B.1 Journal Publications

1. J. Dahl, E. Marsh, C. Lewis, and F. C. Harris Jr, “uMuVR: A Multiuser Virtual Reality and Body Presence Framework for Unity,” International Journal for Computers & Their Applications, vol. 30, no. 1, 2023.

B.2 Conference Publications

1. J. Dahl, E. Marsh, C. Lewis, and F. Harris, “MuVR: A Multiuser Virtual Reality Framework for Unity,” in Proceedings of 31st International Conference on Software Engineering and Data Engineering, F. Harris, A. Redei, and R. Wu, Eds., in EPiC Series in Computing, vol. 88. EasyChair, 2022, pp. 61–70. doi: https://doi.org/10.29007/jdlg.
2. E. Marsh, J. Dahl, A. Kamran Pishhesari, J. Sattarvand, and F. C. Harris, Jr. “A Virtual Reality Mining Training Simulator for Proximity Detection,” in ITNG 2023 20th International Conference on Information Technology-New Generations, S. Latifi, Ed., Cham: Springer International Publishing, 2023, pp. 387–393. doi: https://doi.org/10.1007/978-3-031-28332-1_44.
3. Q. Contaldi, J. Dahl, B. Charyyev, and F. C. Harris, Jr. “Analyzing ‘Existential Dread’ on the Internet in Response to Elmo,” in The 22nd International Conference on Information Technology-New Generations (ITNG 2025), S. Latifi, Ed., Cham: Springer Nature Switzerland, 2025, pp. 167–177. doi: https://doi.org/10.1007/978-3-031-89063-5_15.

4. J. Dahl, Q. Contaldi, K. Partovi and F. C. Harris, Jr. “Mizu: A Lightweight Multi-Threaded Threaded-Code Interpreter that Can Run Almost Anywhere with a C++ Compile” The 23rd IEEE/ACIS International Conference on Software Engineering, Management and Applications (SERA 2025) May 29-31, 2025 Las Vegas, NV.