Universiteit
Leiden
The Netherlands

Bachelor Computer Science

Reviewing the Educational Value of the ChocoPy Compiler Framework

Luuk Daleman

Supervisor:
Miguel O. Blom
Rob V. van Nieuwpoort

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

March 6, 2025

Abstract

Several universities have adopted ChocoPy, a Python to RISC-V compiler framework, as the practical component of their compiler construction course. This thesis focuses on the educational value of this framework by providing a thorough review grounded in empirical analysis. Our goal is to contribute to enhancing compiler education tools. Our evaluation reveals that while ChocoPy excels in areas such as documentation quality, skeleton code structure, and testing infrastructure, there are several opportunities for enhancement in its educational approach. We identify three key areas for improvement: the documentation of semantic analysis error messages, adherence to RISC-V calling conventions, and the absence of optimization techniques in the curriculum. To incorporate the latter we propose to add an additional assignment, balancing the workload by reducing the scope of semantic analysis. To validate our proposed improvements, we extended ChocoPy with a Three-Address Code intermediate representation and implemented several optimization techniques. Through extensive benchmarking across five workloads, we demonstrate that these optimizations can reduce the total number of executed RISC-V instructions by an average of approximately 36%, with improvements ranging from approximately 6% to 56%. Based on these results, we provide recommendations for integrating optimization techniques into the curriculum, emphasizing both educational value and performance impact. Our findings suggest that incorporating these optimizations would significantly enhance the educational value of ChocoPy by providing students with hands-on experience in modern compiler optimization techniques while maintaining a balanced approach to compiler construction education.

Contents

1 Introduction

Compiler construction combines many subjects in the computer science field in interesting and useful practical application. Thus, a well-structured and challenging practical assignment should accompany the theory. The ChocoPy framework [PSH19] is used by multiple international universities to teach compiler construction. The ChocoPy framework consists of the ChocoPy programming language (a subset of Python) skeleton code as basis to help implement a compiler and a reference implementation of the compiler. The framework consists of three assignments; each implements a different stage of the compiler pipeline. The complete compiler pipeline consists of Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Representation and Optimizations, and Target-specific Machine Code Generation and Optimizations. In the assignments, Lexical Analysis and Syntax Analysis are integrated into a single assignment, while Semantic Analysis and Machine Code Generation are addressed as separate, individual assignments. Intermediate representation is not addressed in the framework.
ChocoPy has undergone formal evaluation twice, both conducted by researchers at Lund University [Kar21, GA22]. The first evaluation primarily compared the execution time performance of ChocoPy programs with Python, using an alternative back-end targeting the x86_64 architecture. The second evaluation focused on extending the back-end to support lists, a feature already present in the standard ChocoPy implementation. The primary objective of this thesis is to evaluate the current ChocoPy framework and explore possibilities for improvement. Ultimately, this research aims to contribute to a more effective, engaging, and comprehensive educational compiler construction framework. To do so, the primary question is posed:

"What is the educational value of the ChocoPy framework and how can this be improved?"

To address this question, we evaluate the ChocoPy framework with a particular emphasis on its educational value. Additionally, we explore ways to enhance the framework by extending its functionality, particularly through the introduction of optimizations.
To thoroughly understand the educational value of the framework, the following sub-questions must be answered:

RQ1: "*How effectively are the aspects of the educational value handled in the ChocoPy framework?*"

RQ2: "*How can the deficiencies in the educational value of the ChocoPy framework, as identified through evaluation, be addressed and improved?*"

These sub-questions aim to provide insight into the suitability of the framework as an educational tool and identify areas for improvement. This analysis will naturally lead to a second set of research questions, as we observe that optimizations are notably absent from the framework.

RQ3: "*How can an optimization exercise be implemented into the ChocoPy framework?*"

RQ4: "*Which optimizations should be present in the framework based on the educational value and impact on total RISC-V instructions executed*"

This approach is motivated by the appealing nature of optimizations, which are both theoretically rich and practically engaging. Our goal is to maximize the number of theoretical concepts addressed in the practical assignments. Furthermore, we speculate that students will find optimizations engaging, which could enhance their learning experience and deepen their understanding of compiler construction.

1.1 Thesis Overview

We begin by implementing the assignments provided by the ChocoPy framework. In Chapter 4, we analyze the educational value of the framework, with an emphasis on identifying its key strengths and areas requiring improvement. We provide suggestions for the latter in Chapter 5. This includes an investigation into the possibilities of extending ChocoPy with an optimization assignment and corresponding recommendations. In Chapter 6, we extend the framework by introducing Three-Address Code (3AC) as an intermediate representation (IR) with optimizations. Specifically, we implement Function Inlining, Dead Code Elimination, Constant Propagation, Constant Folding, and Register Allocation. Finally, we compare the optimized implementation of ChocoPy to the reference implementation by analyzing the total number of RISC-V instructions executed across five benchmarks included in the framework. These benchmarks serve to provide accurate recommendations on the optimizations to include and how the assignment may be structured.

2 Background

This section presents the technical aspects and key definitions essential to this thesis. We first examine the concept of educational value and relevant programming language principles necessary for understanding ChocoPy. Additionally, we provide an overview of compilers and the compilation process, establishing the context for ChocoPy as a programming language, its compiler, and the broader ChocoPy framework.

2.1 Educational Value

The concept of educational value encompasses several aspects. M. Mernik and V. Zumer [MZ03] identify certain educational values, which we adopt and expand upon. We incorporate the following aspects:

1. **Balance between Theory and Practice**

   The degree to which theoretical concepts are complemented by their application in practical components, ensuring a cohesive integration of abstract and hands-on learning.

2. **Relevance to Modern Practices**

   This aspect assesses the framework's alignment with contemporary techniques and tools, ensuring relevance to current practices and applications in the field. While important, this is not the most important aspect. This should provide context and motivation for further study to the student, not implement the practical for them.

3. **Translational Challenges from Theory to Practice**

   The identification and resolution of obstacles that students may encounter when applying theoretical knowledge to practical tasks. This includes mitigating challenges that are not inherently related to the core theory, such as by revising or simplifying assignments, or by providing instructional materials.

4. **Quality and Comprehensiveness of Supporting Documentation**

   The extent to which the provided documentation facilitates understanding and guides students in navigating assignments and overcoming challenges effectively.

5. **Motivation and Learning Support**

   Mechanisms that promote student engagement, motivation to complete the practical assignments, and provide support during the learning process, such as feedback systems, progress tracking, and other motivational tools. Additionally, it examines the extent to which diverse learning styles and varying paces of student progress are accommodated.

2.2 Programming languages

Programming languages are varied and are quite complex. Typing, abstraction level and executable creation may differ greatly between them. These subjects are key to understanding compiler construction and this thesis.

2.2.1 Type Systems

Type systems serve as formal mechanisms for associating data types with variables within programming language paradigms. Type classification is characterized by two orthogonal dimensions: type determination timing (static/dynamic) and type mutability (strong/weak) [MMMP90].
In static type systems, variable types are explicitly declared by the programmer at compile-time, enabling early type verification. Conversely, dynamic type systems defer type resolution to runtime, with type assignments performed by the compiler's runtime mechanism.
Strong typing enforces strict type immutability, prohibiting runtime type transformations. Whereas weak typing permits implicit type conversions and type reassignments later in the program [Pie02].
Type width refers to the number of bytes required to store a specific data type. A value of a type with fewer bytes can be stored in a type with a larger byte width without loss of information, a process known as promotion [Str13]. This introduces ambiguity in type mutability and permits implicit type casting in languages like C++. To address this, guidelines such as MISRA C [Lim04] prohibit implicit casting to ensure type safety.

2.2.2 Abstraction

High-level and low-level programming languages are distinguished by their level of abstraction from the underlying hardware. High-level languages provide greater abstractions to simplify programming, often at the expense of increased overhead [FBC+09]. In contrast, low-level languages offer minimal abstraction, granting developers finer control over hardware resources and enabling more efficient code [ALSU06]. However, this increased control comes with added complexity.
The process of creating executables differs between high-level and low-level programming languages. High-level languages often use interpreters that translate source code line by line at runtime, simplifying debugging but adding performance overhead. Low-level languages are typically compiled into machine-specific executables, optimized for the system instruction set architecture (ISA) to eliminate runtime overhead.

2.2.3 Instruction Set Architecture

An ISA defines "the specification of the machine's native language: that is its instructions and their actions" [Wri09]. Most computers are equipped with Intel or AMD processors that use a closed-source complex instruction set computer (CISC) architecture, specifically x86-64. In contrast, reduced instruction set computer (RISC) architectures aim to minimize the number of available instructions compared to CISC. This approach comes with a trade-off, as RISC architectures typically require longer sequences of instructions to perform the same operation. In return it provides a small but complete instruction set [WLPA11]. This does not necessarily imply the architecture is inherently slower [BMVS15].

2.2.4 RISC-V

RISC-V was developed in 2010 at the University of California, Berkeley, and was specifically "designed to support computer architecture research and education" [WLPA11]. As its name suggests, RISC-V is based on a reduced instruction set computer (RISC) architecture. It provides a

base set of instructions that can be extended. This thesis focuses on RISC-V 32-bit assembly with integer multiplication and division extension (RV32IM).
The RISC-V ISA includes 31 general-purpose registers, each capable of holding integer values. The RISC-V calling convention outlines how these registers should be used. Each register has a specific role and a saving convention, which dictates whether the caller or the callee function is responsible for storing the register's contents in memory during function calls. Roles specify which registers should be used for a generic task, such as function arguments and return values.

2.3 Compilers

A compiler is a software tool that translates human-readable programming languages into machine-readable code [ALSU06]. This process is performed through a pipeline, see Figure 1, which analyzes the input for syntax and semantic errors, applies optimizations when applicable, and generates the corresponding target machine code.

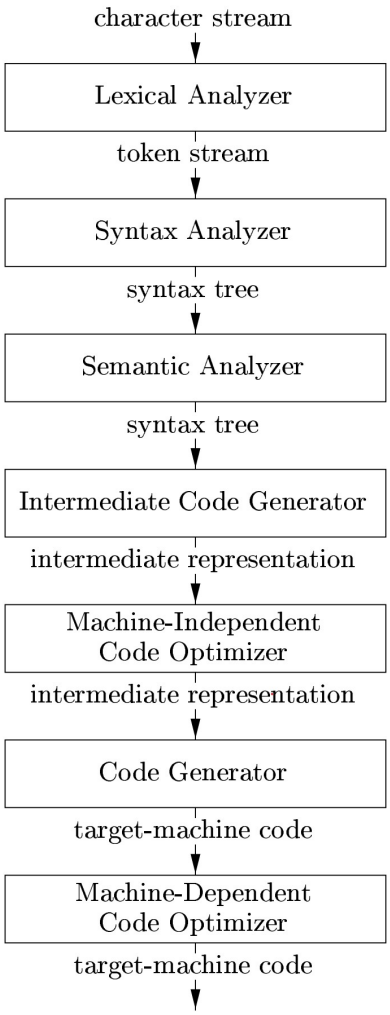

Figure 1: Compiler pipeline overview depicting the various stages of the compiling process [ALSU06]

2.3.1 Front-end

The front-end of a compiler consists of the lexical analysis (lexer), syntax analysis (parser) and semantic analysis. The stream of incoming characters from the source file is tokenized by the lexer. The lexer first groups the character stream into lexemes. For each identified lexeme, a corresponding token is generated. These tokens contain metadata about the original lexeme and are subsequently passed to the parser. Among other functions, lexical analysis facilitates the specification of keywords and enables the detection and handling of errors arising from unrecognized or invalid character combinations.
The parser creates a parse tree from the token stream. This tree represents the syntactic structure and order of the source file according to the grammar rules of the language. Tokens are matched to grammar rules, and errors are reported when no valid rule applies.
Lastly, the semantic analyzer validates the parse tree against the semantic rules defined by the language. This stage ensures that the program is meaningful and adheres to contextual constraints, such as type compatibility, scope resolution, and proper use of identifiers. Upon completion of this stage, the source file is validated, and the parse tree is prepared for transfer to the back-end for further processing.

2.3.2 Back-end

The back-end consists of intermediate representation (IR) generation and target-specific machine code generation. IR serves as an abstracted form of machine code, facilitating machine-independent optimizations. Different types of IR exist, varying in their levels of abstraction. High-level IR closely resembles the source code, while low-level IR is more akin to machine code.
The final step involves translating the IR into target-specific machine code. This process accounts for the architecture and instruction set of the target system to ensure the generation of valid, executable code. To enhance performance, various optimizations may be applied during this stage.

2.3.3 Optimizations

Compiler optimizations are applied in the back-end to enhance performance. These include machine-independent optimizations (applied during intermediate representation generation) and machine-dependent optimizations (applied during target-specific machine code generation). Optimizations may target goals such as space, energy, or execution efficiency, however, this thesis focuses exclusively on execution efficiency. In this thesis, we discuss the optimizations function inlining, dead code elimination, constant propagation, constant folding, and register allocation.

1. Register allocation assigns registers to variables based on heuristics. These techniques identify variables that are simultaneously live or conflicting and assign them accordingly.

2. Dead code elimination removes code that does not affect program output. This includes finding dead variables, which might be declared but are not significant for the program output.

3. Constant propagation replaces variables with their known constant values to simplify computations.

4. Constant folding computes constant expressions at compile time, replacing them with their results.

5. Function inlining replaces function calls with the function's body to eliminate call overhead.

Three-address code (3AC) serves as an intermediate representation (IR) that facilitates optimization processes. This IR resembles machine code but allows flexibility in its specific implementation. 3AC lacks a hierarchical structure, necessitating the formation of basic blocks to enable effective optimizations. Basic blocks are linear sequences of statements executed sequentially, which can be interconnected to form a control-flow graph (CFG), representing the potential execution paths of the program.

2.4 ChocoPy

ChocoPy is a framework developed at the University of California, Berkeley, for use in compiler construction courses [PSH19]. It defines a restricted subset of the Python programming language and targets machine code for the RISC-V RV32IM architecture. The framework is implemented in three assignments.

2.4.1 ChocoPy as Programming Language

Python is a programming language created by Guido van Rossum [VRD09]. It was designed with an emphasis on ease of learning and readability. Python features strong, dynamic typing and is traditionally interpreted. ChocoPy is a strong, statically typed, restricted subset of Python and, as such, does not support all Python features. Notable omissions include packages, dictionaries, exceptions, and default arguments. As a subset, all ChocoPy programs are valid Python code. The language employs native Python type hints to enforce strong, static typing, allowing ChocoPy programs to be compiled. ChocoPy's static typing and compilation model make it more suitable for illustrating how high-level code is transformed into lower-level assembly, helping students understand the underlying processes of program execution and optimization in a way that Python does not.

2.4.2 ChocoPy Compiler

The ChocoPy compiler employs the JFlex lexer [Kle23] for lexical analysis and the CUP parser [CSAP14] for syntactic analysis. These tools generate an abstract syntax tree (AST), a specialized form of a parse tree. Both frameworks are explicitly designed for compatibility and are supported by extensive documentation, enhancing their accessibility for educational use. The compiler targets RV32IM assembly code, chosen to align with the educational goal of using a modern architecture. To evaluate correctness and performance on common x86 machines, the compiler's output is executed on a simulator. ChocoPy utilizes the Venus simulator [VEP22] for this purpose.

2.4.3 ChocoPy as Framework

The ChocoPy framework, implemented in Java, is structured into three assignments, each addressing one or two stages of the compiler pipeline. The first assignment includes both lexical and syntactical analysis, building an AST for the program. The second assignment is focused on semantic analysis, where the generated AST is validated and detailed error messages are implemented. In the final assignment, the validated AST is translated directly into RISC-V machine code. Additionally,

this assignment allows the import of custom RISC-V assembly files to define standard (built-in) functions, with several predefined functions already provided by the framework.
The framework provides extra functionality for executing the Java program, including an obfuscated reference implementation and unit tests. The obfuscated reference serves as an executable program, though its source code remains inaccessible. The unit tests serve as an automatic grading system and not all unit tests are shown to the students. For the second and third assignments, students have the option to use either their own implementation from the preceding compiler stages or the provided reference implementation.
Additionally, it enables interaction with the Venus simulator, allowing users to execute ChocoPy programs or analyze the number of instructions executed. Finally, the framework provides a web-based editor, enabling students to write ChocoPy programs and execute the three assignment stages. Programs can be parsed, semantically analyzed, and compiled into machine code with their own implementation or the reference implementation. The machine code generation initiates a web-based Venus environment, which supports step-through debugging and provides a detailed view of registers and memory, including their respective values.

3 Related work

In this section, several related works are highlighted to provide context and background for the contributions of this thesis. This section reviews relevant works about compiler construction, works related to ChocoPy, as well as research on enhancing compiler construction courses.
The book "Compilers: Principles, Techniques, and Tools" [ALSU06], commonly referred to as the "Dragon Book", is a prominent work in the field of compiler construction and is widely used as a textbook in many compiler design courses. It provides a comprehensive overview of both the theoretical foundations and practical techniques involved in building compilers. Several concepts and methodologies presented in this thesis are aligned with the principles outlined in this book, particularly in the context of compiler construction and optimization. Another influential book about compilers and programming languages is "types and programming languages". This book adopts a more theoretical and mathematical approach, focusing on type systems, programming language semantics, and formal reasoning about compilers. It serves as a valuable complement to the Dragon Book, providing deeper insight into the theoretical underpinnings of programming languages. This book has been used to provide a theoretical framework for ChocoPy's type system and related concepts, while the Dragon Book has informed the practical aspects of compiler implementation and optimization.
The authors of ChocoPy published a paper [PSH19] that provides insights into the design motivation of the framework and briefly addresses performance considerations. The authors aimed to create a subset of a widely-used programming language to enhance its relevance to real-world applications. Furthermore, they selected a modern and sufficiently simple assembly language by implementing the aforementioned RV32IM architecture. The paper also notes that student-implemented compilers could "easily outperform the official Python implementation". However the methodology used is unclear. Presumably the benchmarks were conducted on a RISC-V computer or emulator rather than the Venus simulator, as the latter incurs a significant performance penalty. Additionally, it is unclear whether the reference code generated by the framework also outperforms Python. If this is the case, student implementations could outperform Python without requiring optimizations, raising questions about the framework's ability to motivate students to explore advanced optimization techniques. We seek to investigate how student implementations could surpass the reference implementation by employing techniques that are currently challenging to implement due to the framework's limited optimization support. Such enhancements would likely achieve significant performance gains over the Python interpreter when executed in a native environment.
Lund University in Sweden uses a modified version of ChocoPy in their compiler construction course [Kar21]. Their version uses a different toolchain and, more importantly, compiles directly to x86_64. In this paper, a short evaluation of the ChocoPy framework is done with these changes and was deemed to be good for their compiler construction course. This version was also benchmarked against Python and found to be much faster, which would be for good educational value. We use a different back-end (RISC-V) with a simulator, which makes the performance measured in the paper unobtainable due to the overhead. Lund University also later extended their version of ChocoPy with lists and classes on the heap with garbage collection [GA22]. ChocoPy was researched on the opportunities to be extended, extended with these features and benchmarked against Python. This paper is not relevant to our work as in our framework classes and lists are already featured. Garbage collection is not featured, but is outside of the scope of this thesis.
A recent study by Zhang et al. [ZHZ+21] explores the potential of shifting the focus in undergraduate

compiler construction courses toward machine code generation and optimizations. This approach involves team-based competitions. Performances varied across teams, with some able to outperform GNU GCC -O2 in 10 predefined test cases. This approach offers a compelling perspective on structuring compiler courses, emphasizing hands-on, competitive learning to enhance student engagement. We want to explore opportunities to shift the balance of the current assignments and dedicate more time to code generation and optimization to create a similar engaging experience. However, we do not intend to base grading primarily on competition between students, as this approach may not accommodate diverse learning paces and styles, potentially diminishing educational value.

LISA is a tool used and evaluated in a paper by M. Mernik and V. Zumer [MZ03]. The tool is designed to enhance understanding of compiler front-end construction and meant such as "lex" and "yacc". The paper examines various aspects of its educational value, or "didactical benefits", including support for diverse learning styles and paces, as well as its potential to increase learner motivation. These aspects were carefully analyzed and, where applicable, integrated into our definition of educational value.

The Amsterdam Compiler kit (ACK) [Tan83] is a historically significant compiler toolkit. ACK is based on the core principle of UNCOL (Universal Computer-Oriented Language), a theoretical intermediate language designed to facilitate portability across different source languages and machine architectures. ACK was also used in compiler courses at universities like the Vrije Universiteit Amsterdam and the University of Amsterdam. As a comprehensive compiler toolkit, ACK ensures seamless integration of all stages of the compilation process, making it well-suited for use in compiler courses. It exemplifies the modular nature of compiler design by supporting the translation of multiple programming languages into a shared IR. The IR used in ACK is Experimental Machine 1 (EM-1). This language is a stack-based machine without registers; however, it can be used to effectively allocate registers. This is different from the 3AC used in this thesis, as this tries to minimize the use of a stack. Nonetheless, ACK provides a historically significant and conceptually distinct implementation of compiler construction and IR.

4 Educational value of ChocoPy

This section discusses the educational value of the ChocoPy framework along with specifying strengths and weaknesses. Suggestions to improve the latter will be discussed in the next section. The analysis is conducted with reference to the criteria previously established in regards to evaluating educational value in Chapter 2.1. Notable strengths of the framework include comprehensive documentation, well-structured skeleton code, a robust feedback and progression system through unit tests, a web-based editor with supportive features, and a reference implementation for verifying correct output. Identified areas for improvement include better documentation of error messages in the semantic analysis, incorrect calling convention in machine code generation, and addressing the absence of optimization techniques.

4.1 Qualities

All assignments are comprehensively documented and well-structured, providing students with clear guidance throughout the process. All relevant features of the framework are explained in detail, ensuring that students have a solid understanding of its functionality. Additionally, partially functional example code is provided, which allows students to quickly familiarize themselves with the program. This minimizes the time spent on setup and enables students to focus directly on applying theoretical concepts to the implementation. This increases the educational value through the quality and comprehensiveness of supporting documentation, along with the translation of challenges from theory to practice.

The inclusion of the reference implementation and the modularity of the pipeline design enhance the educational value of the framework by demonstrating the practical application of modular design principles. This approach also allows students to learn at their own pace and in their preferred style. Additionally, it accommodates students who may not have completed previous assignments or lack confidence in their own implementations by enabling them to focus on understanding new concepts without being hindered by prior work.

The education value is further increased through motivation and learning support by the included unit tests and web-based editor. The integration of unit tests to quantify students' progress, offers a clear measure of their accomplishments. This structured approach not only enhances the learning experience, but also motivates students to complete the assignment by providing tangible milestones. The web-based editor further supports diverse learning styles and paces by enabling students to experiment with and debug both their own implementations and the reference implementation. This functionality facilitates a deeper understanding of language specifications and machine code generation.

JFlex and CUP simplify the lexing and parsing process, while providing relevance to modern compiler tools, increasing the educational value. Additional, newer and more powerful tools, such as ANTLR [PQ95], could be included in the project to improve the relevance to modern practices. However, this would undermine the challenges in translating theory into practice and ultimately lose more educational value. RISC-V was chosen for its relevance to modern architectures [PSH19] and thus it also enhances the educational value.

The code generation assignment assigns priority to multiple features to be implemented, creating a checkpoint system. This approach supports diverse learning styles and paces by providing flexible

guidance to students through a complex assignment, while allowing autonomy and encouraging independent judgment.

4.2 Areas of Improvement

The front-end assignments are largely complete and offer high educational value. However, the semantic analysis assignment requires improvement in the aspect of translational challenges from theory to practice. Currently, the error message strings checked by the unit tests, and thus the grading system, must match exactly. Problematically, these strings are undocumented and can only be found through the unit tests or experimentation with the reference implementation.
In machine code generation assignment we identified a deficiency regarding the proper implementation of the register calling convention. The implementation guide explains the calling convention used by ChocoPy well and therefore does not diminish educational value this way. However, this does not align fully with the RISC-V calling convention. This documentation details function arguments to be only passed via the stack rather than through the designated argument registers. This also applies to built-in functions, except the "alloc" and "abort" functions. This deviation from the prescribed calling convention diminishes the educational value of the assignment, as the theoretical concepts of calling conventions are not adequately reflected in the practical implementation. Consequently, students are not fully exposed to an essential aspect of low-level function calls, which undermines the goal of balancing theory and practice and quality of documentation.
Lastly, we find the balance of theory and practice lacking. The ChocoPy framework does not incorporate any optimizations in the machine code generation, nor does it have an IR generation stage with optimizations. While the framework offers a bonus exercise on optimizations, this exercise is structured as a competition, rewarding only the best implementations. This approach has reduced educational value as it does not accommodate all learning styles and paces, leading some students to forgo participation due to limited time or motivation. Furthermore, implementing machine-independent optimization techniques directly on RISC-V code is impractical, as it adds unnecessary complexity and detracts from the foundational understanding of optimization techniques.
The absence of optimizations in the primary implementation diminishes the educational value of the assignment, as it disrupts the balance between theoretical concepts and practical application. In addition, optimizations are often regarded as particularly engaging and intellectually stimulating by students. Introducing optimizations as a core part of the practical would provide students with a more comprehensive understanding of real-world compiler design practices and enhance the learning experience. We find the lack of optimizations to be the largest diminishment of the educational value and significant focus will be put on solving this.

4.3 Conclusion

To determine the educational value of ChocoPy, we systematically correlated the qualities and areas for improvement of ChocoPy with their corresponding educational values, as summarized in Table 1. Certain factors may appear in multiple categories, as they may contribute to or detract from the educational value in multiple aspects.
The framework demonstrates a well-structured design and significant educational value. Supported by its alignment with modern practices, the quality and comprehensiveness of its documentation, and its capacity to effectively foster motivation and support learning. These strengths substantially

Aspects of Educational Value	**Qualities of ChocoPy**	**Areas of Improvement for ChocoPy**
Balance between Theory and Practice	Complete compiler pipeline Extensive semantic analysis	Absence of optimizations Non-conventional function argument passing
Relevance to Modern Practices	Python subset JFlex and CUP RISC-V	N/A
Translational Challenges from Theory to Practice	Documentation on the assignments, the language specification, and the guide to RISC-V Skeleton code Checkpoint system in code generation	Lacking documentation on error messages
Quality and Comprehensiveness of Supporting Documentation	Documentation on the assignments, the language specification, and the guide to RISC-V	N/A
Motivation and Learning Support	Reference implementation Unit tests Web-editor	N/A

Table 1: Evaluation of the Educational Value of ChocoPy. **N/A**: no significant areas for improvement were identified for this aspect of educational value.

contribute to its overall quality as an educational tool. Nonetheless, certain areas require further development. Specifically, the framework could be improved in the transitional challenges from theory to practice, balance between theory and practice. These findings will guide our efforts to enhance the educational value of ChocoPy. We identify the absence of optimizations as the most significant weakness in the educational value of the assignment. Therefore, the second part of this thesis will primarily focus on overcoming this limitation by incorporating optimizations into the framework.

5 Improvements

This section presents solutions to address the identified areas of improvement, namely the error messages in the semantical analysis, the incorrect calling convention and lack of optimizations. Suggestions on how to include an optimization assignment are also provided.

5.1 Error Messages

To address the issue with undocumented error messages, we propose the following solutions. We recommend documenting these messages in the assignment materials and/or including a file in the framework with predefined string variables for use in the code. The latter option would eliminate copying errors, thereby enhancing the educational value more effectively than the former.

5.2 Calling Convention

The calling convention used by the ChocoPy framework should be improved to better represent the RISC-V calling convention. We propose updating the documentation to accurately describe the function argument register calling convention and changing the reference implementation to reflect this. Modifying the reference implementation itself to align with these updates would require significant effort, so we acknowledge that such changes may not be feasible. However, the documentation should explicitly highlight this deviation from the standard RISC-V calling convention. Providing a clear reference or summary would ensure that students understand the theoretical underpinnings, even if they are not directly reflected in the provided reference implementation.

5.3 Optimizations and Extending ChocoPy

To enhance the educational value of ChocoPy, an intermediate code stage should be introduced to facilitate the application of optimization techniques. In the current implementation, the framework already handles the process of translating (high-level) IR into (low-level) machine code. Consequently, the proposed optimization stage would focus exclusively on applying optimization techniques, without requiring students to translate the AST into IR and the IR into RISC-V code. By isolating the optimization process, students can focus on understanding and implementing optimization concepts, ensuring a more targeted and effective learning experience. A new assignment should be created for this.

5.4 New Assignment

Introducing an entirely new assignment to the framework is not feasible given the current scope and workload of the existing assignments. Due to this, the framework currently only discusses optimizations in a bonus exercise.
We believe a bonus exercise does not motivate all students to give it a proper attempt, still leaving the theory and practice out of balance. To address this, we propose reducing the scope of the semantic analysis assignment while retaining its core objectives, to allow the addition of another two- to three-week assignment on optimizations to the framework. The primary goal of the semantic analysis assignment is to teach how semantic checks are performed and how meaningful error

messages can be generated. These messages can be categorized into types such as definition errors, type errors, and class member errors. We recommend allowing flexibility in the specific messages students implement, with requirements for a minimum number per category and a total minimum. This approach preserves the original assignment's educational objectives, while reducing total development time.
The new optimization assignment should be positioned after the machine code generation assignment. At this point, students will have developed a solid understanding of the fundamental concepts involved in compiler design, as well as the potential for optimizations. This sequential progression will ensure that students are well-prepared to tackle optimization techniques, as they will have already gained the necessary background knowledge and practical experience to effectively apply them.

5.4.1 New Provisions in the Framework

The new assignment should introduce a low-level IR similar to 3AC. Since RISC-V instructions are already in 3AC form, students should be familiar with this assembly at this point; we recommend designing an IR modeled closely after RISC-V instructions. As outlined previously, the translations from the AST to the new IR and from the new IR to machine code should be pre-implemented. Additionally, detailed documentation should accompany this assignment, akin to the current framework, and include a list of potential optimizations. To maximize educational value, the provided optimizations should encompass a diverse and impactful selection, encouraging motivation through meaningful achievements and exposing students to varied techniques. Similar or low-impact optimizations would fail to effectively balance theory and practice or adequately engage students, as the lack of tangible results from their efforts could be highly demotivating. To ensure high-quality recommendations for this assignment, we will extend ChocoPy with 3AC and optimizations and benchmark the generated code to the reference implementation.
If desired, a bonus exercise can still be incorporated, offering rewards for additional optimization implementations or organizing a competition, as in the current framework. This approach would now enhance the educational value, as all students would have already engaged with the foundational theory and begun with an equal baseline.

6 Three-Address Code Extension

In this section, we address the lack of IR generation and optimization present in ChocoPy. We do this by implementing 3AC with translations between the AST and machine code and equipping the 3AC with optimizations, namely register allocation, constant folding, constant propagation, dead code elimination and function inlining. Furthermore, to provide accurate recommendations on what optimizations to include in a new assignment, various optimizations are benchmarked. Finally, we will provide recommendations on the most suitable optimizations for a new optimization assignment, along with a suggestion on implementation order. This is based on the benchmarking data and the educational value of implementing each optimization.

6.1 Method

This section outlines the newly proposed pipeline and implementation of the optimization stage, as depicted in Figure 2. The process begins with the transformation of the AST into 3AC. Subsequently, a control-flow graph (CFG) is constructed, followed by the execution of liveness analysis and optional machine-independent optimizations. These optimizations can have a synergistic effect. For instance, constant propagation and constant folding may enhance one another, as the former generates additional constants that the latter can simplify, and vice versa. Finally, we perform register allocation, and translate the optimized 3AC into RISC-V assembly. This high-level overview serves as the foundation for a more in-depth exploration of specific design decisions and implementation strategies, which we discuss in subsequent sections. Notably, built-in functions are not optimized as these are hand-written RISC-V assembly code and the contents are inserted separately from the user-generated code. Consequently, these strings of instructions are exactly the same as the reference implementation.

6.1.1 3AC instructions

The 3AC instructions we implemented in this thesis are based on the RISC-V instructions. Additionally, we corrected ChocoPy's calling convention by employing the appropriate registers, according to the RISC-V calling convention [WLPA11]. The primary objective of the 3AC is to abstract complex instruction sequences and memory management required for functional RISC-V machine code. To aid in this abstraction, instructions such as "param", "call" and "ret" are incorporated, following the approach in the Dragon book [ALSU06]. These additions result in more concise code that can often be translated directly to RISC-V instructions. Sometimes, a 3AC instruction directly match their RISC-V equivalents, making translation trivial. Additionally, the compactness of 3AC simplifies reasoning and transformations, as chains of instructions can be condensed into a single 3AC instruction. Compared to the AST, 3AC offers significant advantages, as it employs simple, sequential instructions that are much closer to the target machine code. This structure provides a more suitable foundation for the application of optimizations.

6.1.2 Function inlining

To inline a function, we first check to see if the function contains recursion. Recursively called functions are inlined a fixed number of times, while non-recursively called functions are inlined until

no function calls are left. Additionally, function arguments are directly substituted, eliminating the need for intermediary function argument registers. Lastly, class methods are not inlined due to time constraints and challenges with polymorphism.

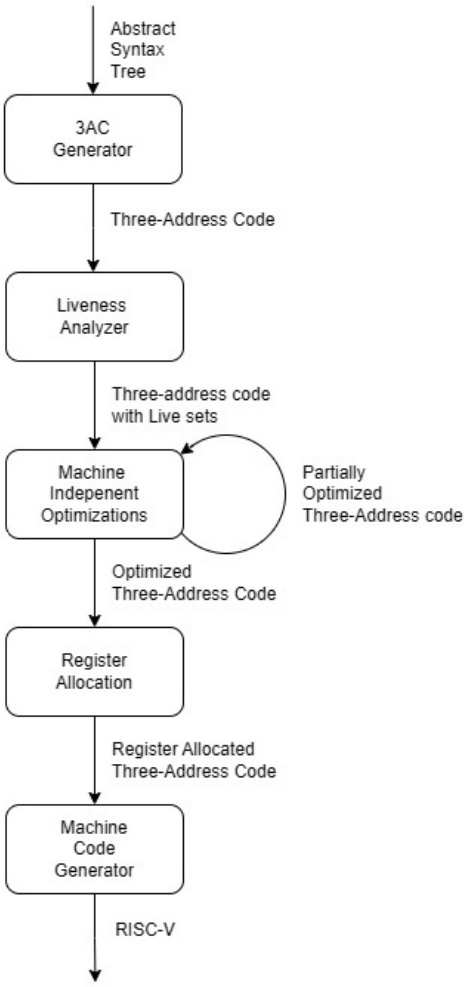

Figure 2: An overview of the intermediate code stage pipeline. Various Machine Independent Optimizations may be performed sequentially. In our implementation this includes constant propagation, constant folding, dead code elimination, and function inlining.

6.1.3 Constant Propagation and Folding

These optimizations are trivial to implement and in terms of their transformation of the assembly. Constant folding does not require additional considerations. However, constant propagation requires an accompanying heuristic, which is highly comparable to liveness analysis, as it determines the points in the program where a variable retains a constant value. This similarity may diminish the educational value of the optimization by introducing redundancy, thereby disrupting the balance between theory and practice.

6.1.4 Dead Code Elimination

The implementation of dead code elimination is carried out in two independent stages. First, unreachable code is identified and eliminated by analyzing the CFG. Second, the liveness analysis is used to detect dead variables, allowing the removal of unnecessary instructions.

6.1.5 Register Allocation

This thesis employs a basic [CAC+81] register allocation algorithm [ALSU06]. An adjacency matrix is built utilizing the live sets from the liveness analysis, enabling efficient register assignments. Registers designated for passing function arguments and return values are reserved to ensure proper functionality, if function calls are present. Priority is given to assigning caller-saved registers, providing greater flexibility during function calls by requiring only the preservation of variables needed after the call. This approach reduces the number of instructions required for saving and restoring register contents. Lastly, register spilling is not implemented. This omission reduces the educational value as the theory is not fully executed in practice. However, this reduces the complexity of the register allocation optimization and the framework, furthermore, this is not required for the used benchmarks. After the registers are successfully assigned, the machine code generator generates valid RISC-V machine code.

6.2 Experiments

In this section we will record the impact of applying various optimization techniques to ChocoPy's code generation process. The performance is determined by counting the total amount of executed RISC-V instructions. By analyzing the results, the educational value of different optimizations can be determined and substantiated recommendations on the new assignment can be provided. We hypothesize that register allocation and function inlining will have the greatest impact on the performance. Additionally, we anticipate that the combination of multiple optimizations will result in cumulative benefits. Through this experimental framework, we seek to identify which optimizations are most beneficial, as well as to understand their educational significance in the context of compiler design and optimization techniques.

6.2.1 Experiment Setup

Quantifying the total RISC-V instructions executed is handled by the Venus simulator used by the framework through a command-line argument. The execution of all benchmark programs is deterministic, and therefore the count remains consistent, not requiring replication for statistical reliability. The actual execution time is not measured. However, reducing the number of executed instructions will give a decent estimate of real-world performance improvements.
The experiments will employ five benchmark programs included in the framework, each designed to evaluate different aspects of the programming language and code generation process. These benchmarks assess key computational features, including recursion, string and list manipulations, and object-oriented programming. The benchmarks and their primary tested features are as follows:

- **Exp** computes repeated exponentiation using a recursive function. This benchmark primarily evaluates recursion within very short functions.

- **Prime** iteratively computes prime numbers and primarily assessing control flow mechanisms such as while loops and conditional statements.
- **Sieve** implements an algorithm similar to the Sieve of Eratosthenes for prime number computation. This benchmark primarily tests list manipulation and object-oriented programming, including polymorphism.
- **Stdlib** converts a number to a string and back, testing extensive if-else chains, string and list manipulation, recursion, and built-in function calls.
- **Tree** constructs a tree data structure by inserting nodes. This benchmark primarily evaluates recursion and object-oriented programming.

The implemented optimizations, namely function inlining, constant propagation, constant folding, dead code elimination, and register allocation, will be tested individually and in combination on these benchmarks. This approach facilitates a comprehensive understanding of the interactions between different optimizations and their cumulative effects within the optimization pipeline.

6.3 Results

This section discusses the results from running the benchmarks with various configurations of optimizations. This aims to show the effectiveness of individual optimizations and combinations of these varying optimizations. First we will show the impact of these variations compared to the reference implementation of the framework. Afterwards, we will take the register allocation configuration as the baseline and compare the additional optimizations to highlight and clarify their effectiveness.

In Figure 3, we observe notable improvements in most of the benchmarks with increased active optimizations, demonstrating the synergistic effects of applying the optimizations collectively. Register allocation alone yields an average performance improvement of 24.30%, establishing a significant baseline compared to the reference implementation. However, the Stdlib benchmark does not exhibit notable reductions in the number of executed instructions in this configuration. As the performance does increase with other optimizations, we suspect the extensive usage of built-in function calls for string and list operations diminishes the effectiveness of register allocation in this workload. The greatest performance gains are observed when all optimizations are applied together, resulting in an average reduction of 35.73% in the total number of executed RISC-V instructions, with a maximum reduction of 56.20% and a minimum of 6.26%. An exception to this trend is observed in the Sieve benchmark. Analysis of the generated assembly code reveals that the machine code generation introduces a significant number of load and store operations surrounding function calls. These instructions are primarily introduced as safety measures to ensure program correctness. However, multiple of these operations were observed to be unnecessary in specific situations, indicating possible further refinement through complementary machine-dependent optimization techniques. The function calls remain due to the lack of inlining member method calls.

To illustrate the impact of the various optimization configurations more effectively, Figure 4 uses the configuration with only register allocation as the baseline. This comparison demonstrates that combining multiple optimizations generally leads to improved performance. However, individual optimizations do not always reduce the total instruction count. For example, constant folding

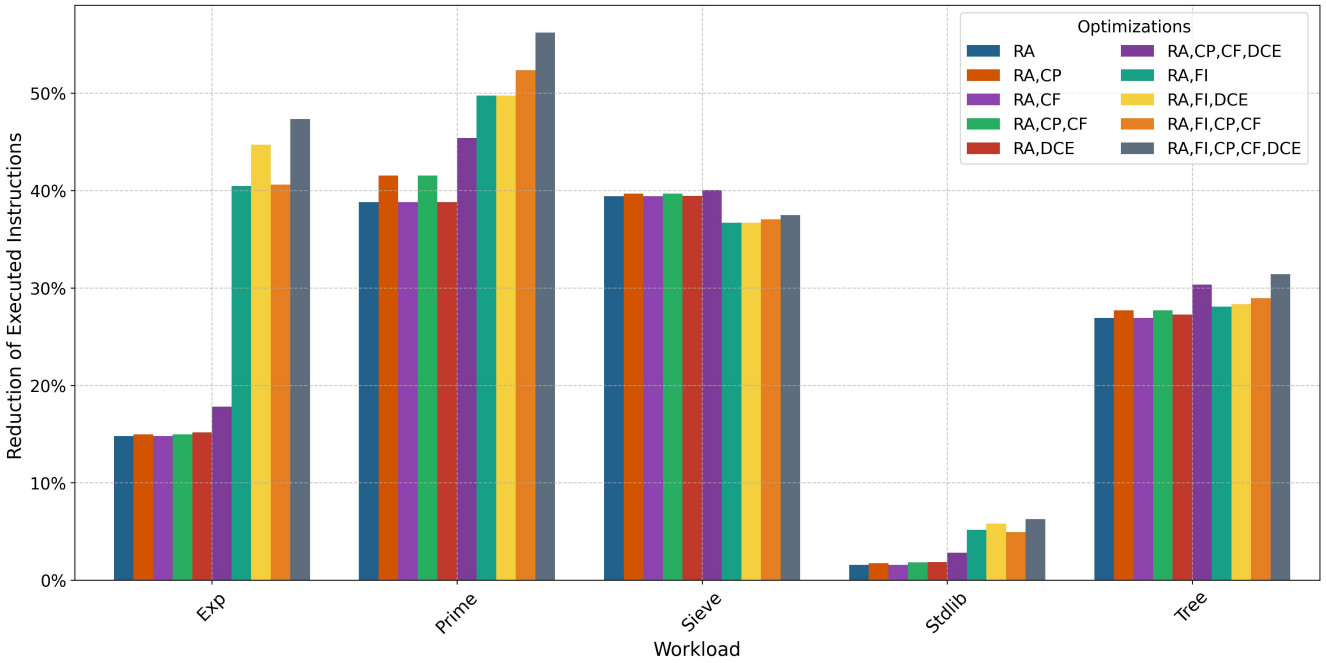

Figure 3: Relative reduction in executed instructions across benchmarks with different optimization configurations. Not all combinations have been graphed to limit the size and complexity of the graph. **RA**: Register Allocation, **CP**: Constant Propagation, **CF**: Control Flow Optimization, **DCE**: Dead Code Elimination, **FI**: Function Inlining.

does not change instruction counts. From the perspective of the Venus instruction counter, this transformation merely replaces one instruction with another, such as converting a binary operation into a move or load operation. Other optimizations are generally ineffective on user-generated code, such as dead code elimination [ALSU06], especially in these workloads. When combined, constant propagation and constant folding decreases total executed instructions only in the Stdlib benchmark by 0.07% compared to only constant propagation. This combination does not report significant results for these workloads. Function inlining represents a notable outlier in its impact on instruction counts. This can significantly decrease the executed instructions, but it can also lead to an increase depending on workload. Once more the outlier result of the Sieve benchmark can be observed. As mentioned above, this is due to an increased amount of load and store operations. The class method calls present in this benchmark are not optimized, which is also the case for Tree which does not benefit as much from inlining as Exp or Sieve. Moreover, we can observe Stdlib gaining a noticeable performance increase with these additional optimization techniques. This is on par with the Tree benchmark, indicating register allocation as an ineffective optimization for this benchmark.

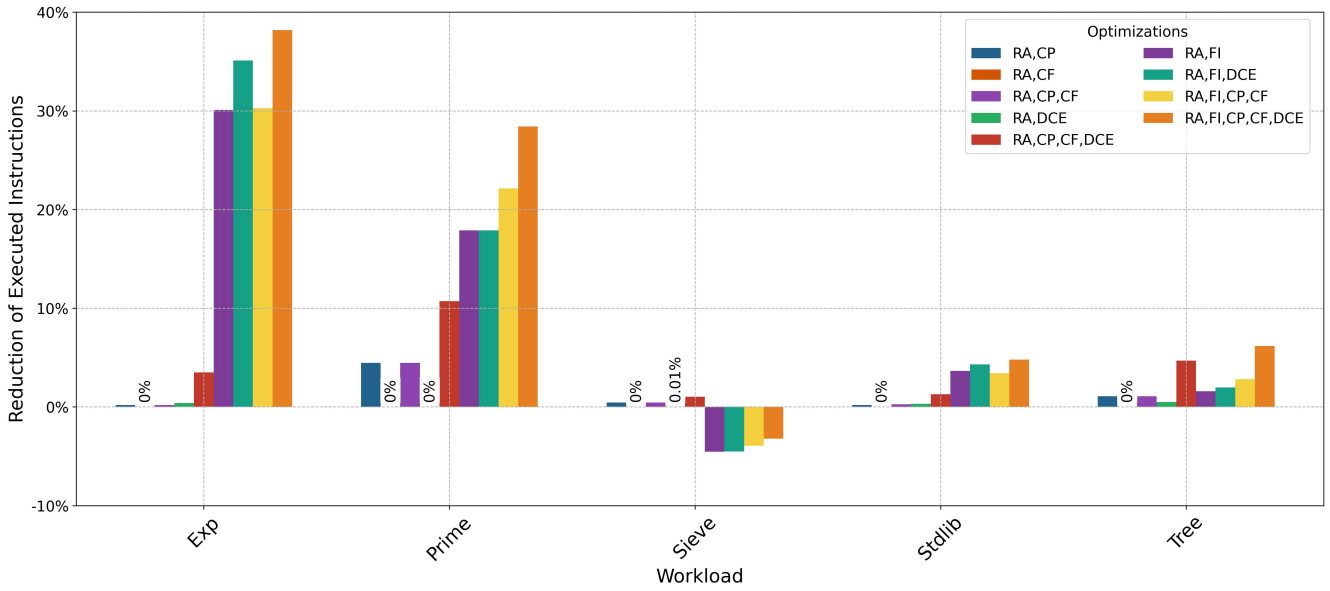

Figure 4: The relative reduction in instruction count compared to only Register Allocation. These are the same data points as in Figure 3, emphasizing the impact of the other optimizations.

The configuration with all optimizations except function inlining, achieves a maximum of 10.72% decreased instructions executed with an average of 4.25% and a minimum of 1.03%. Including function inlining leads to a significant maximum of 38.17% and an average of 14.86%, but also a minimum with an increase of executed instructions of -3.22%. This maximum observed in the Exp benchmark effectively illustrates the impact of this optimization. The benchmark features shorts functions and recursion, meaning stackframe construction and destruction is a large part of the executed instructions. These instructions are significantly reduced or even completely removed, leading to a significant reduction of executed instructions.

6.4 Recommended optimizations

This research aims to enhance balance between theory and practice by introducing compiler optimizations. Implementing these optimizations is expected to produce a measurable impact on performance, thereby motivating students to improve their designs and implement additional optimizations, which will further enhance the educational value. Additionally, the process of implementing these optimizations introduces new theoretical concepts, while avoiding unnecessary repetition, to further increase the overall educational impact.
The first optimization proposed is register allocation, as it introduces a foundational and broad concept within compiler construction theory. Benchmark results demonstrate that this optimization significantly reduces the number of executed instructions, contributing substantially to the overall performance improvement observed. Additionally, the heuristics and graph-based structures required for register allocation can serve as a foundation for other optimizations. Following register allocation, function inlining should be implemented. This optimization contributes the second largest impact to the benchmarking results. Additionally, its theoretical framework is distinct from other optimizations, making it a valuable and independent concept for students to study. This optimization does not guarantee a decrease in executed instructions count, negatively impacting educational value as

this might demotivate students. However, this does teach an important lesson about the trade-offs in compiler optimizations, demonstrating that while function inlining can reduce function call overhead, it may also lead to code bloat and increased executed instructions in certain cases. It should also be noted that both register allocation and function inlining are typically applied once during compilation and therefore lack the iterative or synergistic effects of combining multiple optimizations.

Subsequent optimizations, while not yielding significant standalone reductions in executed instructions, emphasize the importance of combining optimizations to achieve synergistic effects. We recommend implementing dead code elimination as the next optimization. When the heuristics from register allocation are correctly implemented, dead code elimination becomes relatively trivial to implement. Importantly, this reuse does not diminish the educational value due to theoretical repetition, as it does not require the implementation of a concept that is overly similar to those already covered. Additionally, dead code elimination reduces executed instructions and cleans up the code, making debugging easier for students.

The final two techniques, constant propagation and constant folding, do not individually meet the criteria for a highly impactful optimization. Constant propagation relies heavily on heuristics similar to those used for register allocation and dead code elimination, while constant folding does not independently yield measurable performance improvements in our benchmarks. However, when applied in combination with other optimizations, they contribute to a noticeable reduction in the total number of executed instructions. We propose packaging these two techniques together, as constant folding is straightforward to implement and complements constant propagation effectively. This approach ensures that students still gain experience implementing repeatable, synergistic optimizations. To compensate for the limited impact of these techniques, existing benchmarks can be modified or additional benchmarks can be introduced. These benchmarks should be specifically designed to provide more optimization opportunities for these techniques. Nonetheless, this approach does not fully address the imbalance between theory and practice caused by the overlapping heuristics required for constant propagation. However, we argue the educational value gained by implementing synergistic optimizations and relevant benchmarks overall increases the total educational value.

7 Conclusion

This thesis evaluates the ChocoPy framework as an educational tool and explores opportunities for improvement. For the evaluation we look specifically at how effectively the aspects of educational value are handled by the Chocopy Framework (RQ1). ChocoPy meets all five criteria of educational value with several aspects of the framework. These criteria are balance between theory and practice, relevance to modern practices, translational challenges from theory to practice, quality and comprehensiveness of supporting documentation, and motivation and learning support.
To enhance the educational value (RQ2), we identified three key areas. Here, improvements could enhance the balance between theory and practice, as well as mitigate translational challenges from theory to practice:

- **Semantic Analysis:** The framework should provide a reference, in documentation or code, for required error messages, as searching for exact wording is not a core theoretical aspect of compiler error handling.
- **Calling Convention:** The current implementation of function argument passing does not fully align with the RISC-V calling convention and requires revision.
- **Optimizations:** The absence of optimization techniques reduces the educational value, necessitating their inclusion. We implemented, extensively analyzed, and provided recommendations on this.

Incorporating optimizations into the framework as an assignment (RQ3) requires some reorganization of the current course structure to accommodate additional content without overwhelming students. To address this, we recommend reducing the scope of the semantic analysis assignment by limiting the number of required error messages to be implemented. This adjustment would make room for an additional two- to three-week assignment focused on optimizations. By providing pre-implemented translations between the IR, front-end, and machine code, students can focus solely on optimization techniques.
To guide which implementations to include and the order of implementations (RQ4), we implemented and benchmarked 3AC with several optimizations. We recommend introducing register allocation and function inlining first due to their strong theoretical foundation and significant impact on benchmark results. Dead code elimination should be implemented next, as it simplifies the 3AC, thereby easing the implementation of other optimizations. Implementing this optimization alongside constant propagation and constant folding further reduces executed instructions and enhances the educational value by teaching the importance of synergistic effects between optimizations. To further demonstrate the effectiveness of these benchmarks, new benchmarks or modifications to the existing benchmarks could be introduced. When all optimizations are applied, we observed an average reduction of 35.73% in the total number of executed instructions, with a maximum reduction of 56.20% and a minimum of 6.26%. These results highlight the substantial benefits of incorporating optimization techniques into the curriculum, further strengthening ChocoPy as a comprehensive and effective educational tool.
Finally, we can confidently conclude that ChocoPy holds significant educational value. By implementing the proposed changes, its effectiveness can be further enhanced, providing a more comprehensive and impactful learning experience. This not only addresses our main research question but also strengthens ChocoPy as a comprehensive and effective educational tool.

7.1 Discussion and Future work

Our assessment of the educational value of ChocoPy primarily focuses on the structure and implementation of the framework. However, this evaluation lacks formal surveys to measure the impact of using ChocoPy on students' knowledge acquisition. The original ChocoPy paper did conduct an informal survey, however, a more rigorous and formal study would provide valuable insights. Future research could involve the development of a standardized, scientific methodology for evaluating educational frameworks and assignments such as ChocoPy. Several papers on compiler construction practical assignments were reviewed, but identifying a comprehensive and clearly defined set of criteria for assessing educational value proved challenging.

Benchmarking the constructed ChocoPy compiler could also be improved and extended. Our current approach measures the number of executed instructions. This benchmarking could be expanded by utilizing RISC-V hardware to compare native performance or by employing emulators such as QEMU to better compare ChocoPy's output to other real languages, such as Python or C++. Unfortunately, due to time constraints, these additional benchmarking methodologies were not incorporated into the conducted experiments. Furthermore, enhancing the framework to guide students through the installation process and enabling comparisons with real-world programming languages would make the optimization assignment more engaging. Similar to efforts at Lund University, the inclusion of additional ISA backends in the framework would provide students with practical experience in comparing performance across different architectures. This expansion would also demonstrate the modularity of a compiler by allowing students to target different architectures as part of the optimization assignment. However, this is a large effort if no tools exist yet. We recommend running the assembly through emulation as relative performance to Python should remain consistent, additionally, students are not likely to have prior experience with such software. Furthermore, our implementation has significant potential for improvement through the enhancement of existing optimizations and the inclusion of additional techniques. One notable limitation of the current implementation is that function inlining does not support the inlining of class methods. Extending this functionality would require analyzing the program for potential conflicts arising from inheritance and handling them appropriately. This feature was not implemented due to time constraints. We think this functionality would dramatically increase the performance on these benchmarks. Calling class methods requires even more instructions to complete and the methods in the benchmarks are similarly short to the functions in the Exp benchmark.

The current 3AC generation introduces opportunities for further optimization. Specifically, the code generation simplifies binary operations by splitting them into multiple instructions, facilitating ease of implementation. This design choice makes the implementation of copy propagation a particularly impactful optimization, as it could significantly reduce the number of redundant instructions. This optimization was also not implemented due to time constraints. We did not want to expand the 3AC to mitigate this splitting as we believe this to be the responsibility of optimizations.

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[BMVS15] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Saalingam. ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures. *ACM Transactions on Computer Systems (TOCS)*, 33(1):1–34, 2015.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[CSAP14] Dan Wang Andrew W. Appel C. Scott Ananian, Frank Flannery and Michael Petter. *CUP*. Georgia Institute of Technology, June 2014. Available at https://www2.cs.tum.edu/projects/cup/docs.php.

[FBC+09] Daniel Frampton, Stephen M Blackburn, Perry Cheng, Robin J Garner, David Grove, J Eliot B Moss, and Sergey I Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 81–90, 2009.

[GA22] Ellen Grane and Samer Alkhodary. Extending a chocopy compiler to include lists and classes on the heap. 2022.

[Kar21] Tobias Karlsson. Chocopy compiler. Technical report, Jan 2021. https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2020/Reports/Karlsson.pdf.

[Kle23] Gerwin Klein. *JFlex*, Mar 2023. Available at https://jflex.de/.

[Lim04] MISRA Limited. *Misra-C:2004: Guidelines for the use of the C language in Critical Systems*. MISRA Limited, Watling Street, Nuneaton, Warwickshire CV10 0TU, UK, 2004.

[MMMP90] Ole Lehrmann Madsen, Boris Magnusson, and Birger Mølier-Pedersen. Strong typing of object-oriented languages revisited. *ACM SIGPLAN Notices*, 25(10):140–150, 1990.

[MZ03] M. Mernik and V. Zumer. An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46(1):61–68, 2003.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, first edition, February 2002.

[PQ95] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[PSH19] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. ChocoPy: a programming language for compilers courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, SPLASH-E 2019, page 41–45, New York, NY, USA, 2019. Association for Computing Machinery.

[Str13] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, fourth edition, June 2013.

[Tan83] Tanenbaum, Andrew S and Van Staveren, Hans and Keizer, EG and Stevenson, Johan W. A practical tool kit for making portable compilers. *Communications of the ACM*, 26(9):654–660, 1983.

[Tu19] Gan Tu. ChocoPyCompiler. https://github.com/Gan-Tu/ChocoPyCompiler, May 2019. GitHub.

[VEP22] Keyhan Vakil, Julian Early, and Rohan Padhye. *Venus RISC-V Simulator*. 2022.

[VRD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[WLPA11] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116:1–32, 2011.

[Wri09] Stephen Wright. *Formal construction of instruction set architectures*. PhD thesis, Citeseer, 2009.

[ZHZ+21] Yu Zhang, Chunming Hu, Mingliang Zeng, Yitong Huang, Wenguang Chen, and Yuanwei Wang. Encouraging compiler optimization practice for undergraduate students through competition. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, pages 4–10, 2021.

A Appendix

The implementation of ChocoPy was carried out with the assistance of a reference implementation by Gan Tu, available on GitHub [Tu19]. Access to the Git repository containing all code related to this thesis can be requested by contacting the supervisors via email.