

Automatic Conversion of C and C++ Programs to the BuildIt Multi-Stage Programming Framework

by
Aryan Kumar

S.B. Computer Science and Engineering, Massachusetts Institute of Technology, 2025

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 2025

© 2025 Aryan Kumar. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Aryan Kumar
Department of Electrical Engineering and Computer Science
May 16, 2025

Certified by: Saman Amarasinghe
Professor of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Automatic Conversion of C and C++ Programs to the BuildIt Multi-Stage Programming Framework

by

Aryan Kumar

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

ABSTRACT

BuildIt allows users to write C++ programs that can execute in multiple stages, where the output of one stage is the program source for the next stage, ending with some final output produced. This is particularly useful for writing specialized code and generating code for domain-specific languages. While there are other approaches to multi-stage programming, BuildIt has several advantages: it takes a library-based approach (so it requires no modifications to the compiler and is thus highly portable), and it has excellent ease of use as all the user has to do is change the declared types of variables in their C++ program. The goal of this thesis is to further improve BuildIt's ease of use by simplifying this step: in particular, by developing a tool that will automatically convert existing C and C++ programs to the BuildIt framework. We show how to use Clang tooling in conjunction with modifications to the Clang compiler to perform non-trivial modifications to source, namely type-modification, to automatically convert code to its unstaged BuildIt equivalent. As the unstaged BuildIt code can be specialized by staging certain variables, this tool will ultimately enable more easily staging and optimizing C/C++ repositories with the BuildIt framework.

Thesis supervisor: Saman Amarasinghe

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to express my gratitude to my mentor and direct advisor, Ajay Brahmakshatriya. He has sat through numerous meetings with me, helping me debug, answering my questions, and teaching me what it means to be a researcher and truly be passionate about compilers. His continual guidance throughout the research process has been instrumental in shaping my work and helping me grow as a researcher. I would also like to extend my thanks to my faculty supervisor, Professor Saman Amarasinghe, for first introducing me to the field of compilers and performance engineering through his class 6.172 Performance Engineering, and for accepting me into his lab. I have learned an incredible amount about this field of research and I am grateful for the input he has provided on my research.

I am sincerely grateful to all the professors, teaching assistants, and researchers I have met during my time at MIT. I entered MIT as a student eager to learn, and thanks to all of these individuals I can say this goal has been successful. I have learned a great deal, have been exposed to new fields, and have been continuously challenged. My time at MIT has been a period of significant personal and academic growth, and I will always remember it fondly.

Thank you to all of my friends for making my last four years enjoyable. From the late-night problem-set sessions to our excursions into Boston, I have truly cherished our time together. My friends' support, laughter, and understanding have helped me grow as a person, and has made my time here much more fulfilling.

Lastly, I would like to give my heartfelt thanks to my family. Thank you for always believing in me, for providing me with the best opportunities, and for all of your sacrifices. Without your unconditional support and encouragement, this thesis would not have been possible.

Contents

<i>List of Figures</i>	9
<i>List of Tables</i>	11
1 Introduction	13
1.1 Problem and Motivation	13
1.2 Contributions	15
1.3 Thesis Overview	15
2 Background and Related Work	17
2.1 Clang	17
2.2 BuildIt	19
2.3 Existing Solutions for Source Modifications	20
3 System Description	23
3.1 System Overview	23
3.2 System Motivation	25
3.3 System Components	30
3.3.1 Clang Rewriter Tool	30
3.3.2 Modified Clang C/C++ Compiler	33
3.3.3 Stub Generator Script	36
3.3.4 Driver Code	39
3.4 Support for Various Features	40
3.4.1 Structs	40
3.4.2 Variadic Functions	43
3.4.3 Recursive Functions	44
3.4.4 Global Variables	46
3.5 Challenges and Limitations	47
4 Results and Evaluation	49
4.1 Results	49
4.1.1 Example 1	49
4.1.2 Example 2	52
4.2 Evaluation	62

5	Conclusion	65
5.1	Future Work	65
5.2	Summary	66
	<i>References</i>	67

List of Figures

1.1	The BuildIt conversion tool's pipeline.	15
2.1	A simple C++ program and its generated AST shown below. The AST consists of a FunctionDecl node, consisting of two children ParmVarDecl nodes corresponding to function parameters a and b , a CompoundStmt node corresponding to the first line of the function, and a ReturnStmt node corresponding to the last line of the function. These children nodes themselves have children, capturing further structure within the program.	18
2.2	A power function staged in Buildit is shown on the left, where the base is a dynamic variable and the exponent is a static variable. The generated code is shown on the right, with 15 passed as the static exponent argument. It is important to note that the static variable exponent is no longer present, and only the dynamic variable base remains in the generated code.	20
3.1	System diagram of the BuildIt conversion tool.	24
3.2	An add function along with its generated AST.	26
3.3	The AST produced by running the Clang Plugin on the add function shown in figure 3.2.	27
3.4	A BuildIt add function and its (truncated) generated AST. This is the intended result of performing source-modifications on the code in figure 3.2. Note, minor reformatting was performed to improve visualization.	28
3.5	A simple power program is shown on the left, and the emitted source code from running the Clang Tool on it is shown on the right.	29
3.6	A simple power program is shown on the left, and the emitted source code from running the Clang Tool on it is shown on the right. Declaring multiple variables in a single statement results in an error with the Clang Tool, as can be seen in the first line of the function body on the right.	30
3.7	The code contained inside dbd_support.h.	32
3.8	A preprocessed program is shown on the left, and the result of running the Clang Rewriter Tool on it is shown on the right.	33
3.9	A simple program is shown on the left. The result of running the program through the Rewriter Tool and modified Clang compiler produces an object file for the code shown on the right. The puts() declaration's types are changed to match its function call, though it is missing a definition.	37
3.10	An example of a linker error dumped from compiling the program shown in 3.9.	38

3.11	The header file generated by our script based on the linker error message in 3.10.	38
3.12	The driver code, used to generate code for functions, struct-types, and global variables, is shown.	39
3.13	The left and right image show a program and its equivalent code in BuildIt, respectively. This shows BuildIt's treatment of struct types.	41
3.14	The result of applying the Rewriter Tool to the program shown on the left in 3.13.	42
3.15	The left shows a simple program that calls the variadic function <code>printf</code> . The right shows the emitted output from running the Rewriting Tool on the program on the left.	44
3.16	A recursive program and the result of performing source rewrites on it is shown.	45
3.17	A program with global variables, and the result of running the Rewriter Tool on it.	46
4.1	A user program that calls the recursive function factorial, and makes use of global variables, structs, and variadic functions.	50
4.2	The result of the preprocessing the program shown in 4.1. We have truncated the code included from <code>stdio.h</code> header to only include the relevant <code>printf</code> declaration, for display purposes.	50
4.3	The result of performing source rewrites to the program shown in 4.2.	51
4.4	The driver for the program in 4.1.	52
4.5	The generated code produced from the BuildIt program's executable. This should be equivalent to the code in 4.1.	53
4.6	The time taken to compile the programs from example 1 and example 2 in section 4.1, from a standard approach and from converting and compiling the program with the BuildIt conversion tool. Note, no linker errors are thrown when converting example 1 and 2 so no stubs need to be generated.	63

List of Tables


4.1	Lines of Code at Different Stages in the Conversion Process	62
-----	---	----

Chapter 1

Introduction

1.1 Problem and Motivation

As computation workloads from machine learning and data-science applications continue to grow, it has become increasingly important for developers to write specialized, high-performance code. Often, developers will need to write code that targets hardware features or intrinsics to ensure good performance for their applications. Using domain-specific languages (DSLs) is one way for developers to achieve this, as DSLs enable users to write optimized code for specific use cases while still maintaining some programming generalizability. In particular, DSLs abstract away many lower-level programming details, making them much simpler for developers to use. GraphIt [1], Halide [2], and TensorFlow [3] are notable examples of DSLs that have been released in recent years. Interestingly, these DSLs are simply multi-stage programming frameworks with two stages, where the first stage corresponds to code in the high-level DSL, and the second stage corresponds to high-performance code in C++/CUDA. In the simplest sense, multi-stage programming refers to code that executes in several stages, where the output of one stage is the code for the next stage. Multi-stage programming provides a much more general way to generate specialized code, and can be used to implement DSLs [4, 5].

BuildIt [6] is a type-based programming framework that enables users to write multi-staged C++ programs. BuildIt’s support for writing multi-staged programs makes it particularly useful for writing specialized, high-performance code and for generating code for DSLs. For instance, BuildIt has been used to implement a GraphIt compiler [7]. While there are other approaches to multi-stage programming, BuildIt has several advantages: it takes a library-based approach, so it requires no modifications to the compiler and is thus highly portable; and it has excellent ease-of-use. To use BuildIt for multi-stage C++ programming, all the user has to do is change the declared types of variables in their C++ program (by wrapping it in the appropriate BuildIt template) so that BuildIt can determine what stage to bind each variable .

The goal of this thesis is to further improve BuildIt’s ease of use by enabling automatic conversion of existing C and C++ programs to the BuildIt framework. Concretely, this thesis demonstrates how to use Clang tools and modify the Clang C/C++ compiler to automatically convert all types in a program to be a BuildIt compatible type, and thus produce BuildIt code that is equivalent to the original source. The converted BuildIt code represents an unstaged program that users can begin staging variables in to specialize their code. The execution of the BuildIt program will then generate optimized C/C++ code as output. Thus, our conversion tool performs the tedious work needed to convert a program to its unstaged BuildIt equivalent, so that users can then stage and optimize their code. We show that the tool we develop can convert sample programs to the equivalent BuildIt code successfully, demonstrating the viability of our conversion tool. We restrict support of our tool to C and C-like C++ programs to make the problem scope appropriate for a thesis.

At a high-level, the tool converts code to BuildIt through a combination of source rewrites and modified parsing by a Clang compiler. The rewriting process prepares the code for parsing by the modified compiler, and is crucial for supporting conversion of many C language features. The modified compilation process changes types in the program to BuildIt types. The conversion pipeline is shown in figure 1.1.



Figure 1.1: The BuildIt conversion tool’s pipeline.

1.2 Contributions

We make two main contributions in this thesis. First, this work details how to use Clang tooling in conjunction with modifications to the Clang compiler to perform structural modifications to C and C++ source. In doing so, we highlight limitations of different Clang tooling and also share insights that may be useful for performing source modifications in other works. Secondly, this work demonstrates how to convert existing C and C++ programs to the equivalent BuildIt code. We support converting C and C++ programs with various features, including variables, functions, structs, and global variables, leaving support for C++ features such as classes and templates for future work. This conversion is useful because there are multiple steps required to optimize a program with BuildIt: it must first be converted to the equivalent BuildIt code, and then certain variables must be staged and specialized. This thesis completes the first step, and its work can be extended to allow users to stage/specialize certain variables when automatically converting their code to BuildIt, thus enabling users to stage and specialize their code with the BuildIt multi-stage framework. Ultimately, our tool brings us one step closer to the goal of being able to automatically stage and optimize large C and C++ codebases with BuildIt.

1.3 Thesis Overview

Chapter 2 provides relevant background on Clang and Buildit, and also discusses existing approaches to source code modifications, particularly for programs written in C and C++. Chapter 3 discusses the design and implementation of our BuildIt conversion tool. We

discuss the various components—including the Clang Rewriter tool, our modifications to the Clang compiler, and the driver code—along with particular work done to support standard C programming features. In chapter 4, we apply the BuildIt conversion tool to sample C programs, and show that our tool produces equivalent BuildIt code. Chapter 5 describes future directions of this work, particularly how the conversion tool can be extended to enable automatic staging and optimization of large C/C++ code-bases with BuildIt. Chapter 5 also concludes the thesis.

Chapter 2

Background and Related Work

In this chapter, we provide relevant background on Clang and BuildIt, and cover related works on source modifications.

2.1 Clang

Unlike Python programs which are interpreted live at runtime, C and C++ programs must first be compiled into an executable, which can then be run by the machine. Clang [8] is an open-source C/C++ compiler that is available as part of the LLVM-project, with significant tooling built around it. The Clang compiler works by running several passes through the source code. In the first few passes, Clang parses and constructs an abstract-syntax-tree (AST) representation of the code. The AST captures the hierarchical structure of the code, as can be seen in Figure 2.1, where the higher-level node corresponds to a function declaration and the children nodes correspond to the parameters and statements within the function body.

Clang exposes many tools that can act on the AST representation to perform either introspection or modifications on the code. For example, Clang provides a `RecursiveASTVisitor` Class whose methods can be overridden to visit certain AST nodes and perform custom actions on them. This can be utilized by user-written Clang tools [9] (stand-alone code that

```

1 // function returns the sum of two numbers
2 int add(int a, int b) {
3     int sum_of_numbers = a + b;
4     return sum_of_numbers;
5 }
6
7 '-FunctionDecl add 'int' (int, int)'
8   |-ParmVarDecl used a 'int'
9   |-ParmVarDecl used b 'int'
10  '-CompoundStmt
11    |-DeclStmt
12    | '-VarDecl used sum_of_numbers 'int'
13    |   '-BinaryOperator 'int' '+'
14    |     |-ImplicitCastExpr 'int' <LValueToRValue>
15    |       |-DeclRefExpr 'int' lvalue ParmVar 'a' 'int'
16    |     |-ImplicitCastExpr 'int' <LValueToRValue>
17    |       |-DeclRefExpr 'int' lvalue ParmVar 'b' 'int'
18    '-ReturnStmt
19      '-ImplicitCastExpr 'int' <LValueToRValue>
20        '-DeclRefExpr 'int' lvalue Var 'sum_of_numbers' 'int'

```

Figure 2.1: A simple C++ program and its generated AST shown below. The AST consists of a FunctionDecl node, consisting of two children ParmVarDecl nodes corresponding to function parameters *a* and *b*, a CompoundStmt node corresponding to the first line of the function, and a ReturnStmt node corresponding to the last line of the function. These children nodes themselves have children, capturing further structure within the program.

can perform user-defined actions on a program) or Clang Plugins [10] (code which can be loaded in during compilation to perform user-defined actions on a program). For instance, a particularly powerful Clang tool combines a Clang Rewriter with a RecursiveASTVisitor to visit specific AST nodes in a program and perform rewrites to their source. Internally, Clang’s code also uses the AST structure to parse, analyze, and compile the code.

After the AST is constructed, Clang’s next set of passes optimize the code. The Clang compiler at this stage may restructure control flow, perform constant folding, function inlining, etc. It is important to note, however, that Clang does not change the original behavior of the code while optimizing. Finally, in the last stage, Clang generates an executable for the program which can actually run on the target machine.

2.2 BuildIt

As previously stated, BuildIt [6] is a type-based programming framework that allows users to write multi-staged C++ programs. This enables users to generate specialized code and implement DSLs. Whereas other approaches to multi-stage programming require esoteric compiler modifications, BuildIt takes a library-based approach, requiring simple modifications to the code rather than the compiler. This results in BuildIt code being easier to write and being more portable.

The BuildIt framework requires all types in a program to have the type `static_var<T>` or `dyn_var<T>`, corresponding to BuildIt’s two main stages, the static (first) stage and dynamic (second) stage. Variables with type `static_var<T>` have type `T` in the static stage, and any control flow or statements comprising of variables of this type are resolved. BuildIt uses repeated execution of the static stage to produce an AST to generate code for the next stage. At the end of the static stage, no `static_var` variables exist, and the code is passed to the dynamic stage. At this point, any variables with type `dyn_var<T>` have type `T` and the program can execute like any standard C or C++ program. Figure 2.2 shows an example taken from [6] of a simple power function staged in BuildIt. It is worth noting that BuildIt can execute code in more than two stages by wrapping the BuildIt types `static_var<T>` and `dyn_var<T>` with `dyn_var`.

To convert a program to the equivalent BuildIt program, all the types must be wrapped in `dyn_var`. Thus, in this thesis we focus on automatically wrapping all types of variables, functions, structs, etc. with the `dyn_var` type. However, for a user to begin optimizing their program, they must further specify certain variables to have the `static_var` type in order to specialize the program, and generate optimized code. We leave enabling this to be an extension project of our BuildIt conversion tool.

It is also worth differentiating BuildIt from the compile-time features of C++, such as templating and constant-folding. BuildIt allows users to view the generated code, whereas Clang produces an object-file with no introspection capabilities. In addition, BuildIt is a

<pre> 1 dyn_var<int> power(dyn_var<int> base, static_var<int> exp){ 2 dyn_var<int> x=base, res=1; 3 while (exp > 0) { 4 if (exp % 2 == 1) 5 res = res * x; 6 x = x * x; 7 exp = exp / 2; 8 } 9 return res; 10 }</pre>	<pre> 1 int power_15(int base){ 2 int res = 1; 3 int x = base; 4 res = res * x; 5 x = x * x; 6 res = res * x; 7 x = x * x; 8 res = res * x; 9 x = x * x; 10 return res; 11 }</pre>
--	---

Figure 2.2: A power function staged in Buildit is shown on the left, where the base is a dynamic variable and the exponent is a static variable. The generated code is shown on the right, with 15 passed as the static exponent argument. It is important to note that the static variable exponent is no longer present, and only the dynamic variable base remains in the generated code.



much more general multi-stage programming framework, where code can execute in more than two stages.

2.3 Existing Solutions for Source Modifications

There is an ample amount of literature on performing modifications to source code in C/C++. For instance, Clang provides various types of tooling such as Clang Plugins and Clang tools. The tools' API enable performing user-defined actions on the code, such as syntax-checking or minor modifications to the source like reformatting [9, 10]. However, there is little available in the Clang API that enables readily making structural changes to code.

A number of academic projects have also focused on modifying C/C++ source through Clang. Wright et al. discusses using the Map-Reduce framework to efficiently refactor large amounts of code in parallel [11]. The authors use Clang tooling to perform source rewrites, though their work mostly focuses on using Map-Reduce to parallelize the refactoring workload. Their work performs source modifications similar to Clang-Tidy [12], a Clang tool used for linting code.

There are also works that make more substantial changes than refactoring to source. Antal et al. use Clang Matchers and other Clang tooling to backport code from C++11 to C++03 [13]. Their approach involves writing Clang tools to match specific features in C++11 and replace it with the equivalent C++03 code in the source. Balogh et al. [14] also utilize Clang AST Matchers. Their work focuses on building OP2-Clang, a translator from high-level C/C++ source to efficient parallel code in a target language (such as CUDA, OpenMP, etc). In general, Clang’s infrastructure supports generating high-performance code from a higher level abstraction. A notable such example is OpenCilk [15] which uses LLVM-IR to generate efficient, parallel code for the Cilk C++ library.

This work thus largely deviates from the focus of prior works on source modification, as we do not focus on making minor changes to source for the purposes of refactoring or attempt to generate optimized lower-level code. Rather, we focus on using Clang’s tooling along with modifications to the Clang compiler to make significant structural modifications to C and C++ source. Specifically, we show how to modify all of the types in a program to convert it to the BuildIt framework.

Chapter 3

System Description

3.1 System Overview

The BuildIt conversion tool we develop enables converting C and C-like C++ programs to the equivalent BuildIt code. Specifically, our conversion tool appropriately wraps all types in a source program with the BuildIt `dyn_var<T>` type. The result is a BuildIt program that will generate source code equivalent to the original program.

The BuildIt conversion tool consists of four core components: a Clang Rewriter tool, a modified Clang C/C++ compiler, a script that generates stubs from linker errors, and finally driver code. The bulk of the code transformation is performed with the first two components. The Clang Rewriter tool performs source rewrites on the code. It includes in the appropriate BuildIt headers; performs source rewrites necessary to convert features like structs, recursive functions, global variables, and variadic functions; and finally it encapsulates the user's program in a namespace, so the modified Clang compiler knows what code to transform. The Clang Rewriter tool is essentially run as the first step in the conversion process, completing the necessary preparation before the modified Clang compiler can run.

The modified Clang compiler performs the type modifications in the source, wrapping all types of variables, functions, etc. with the BuildIt `dyn_var<T>` type. As Clang internally

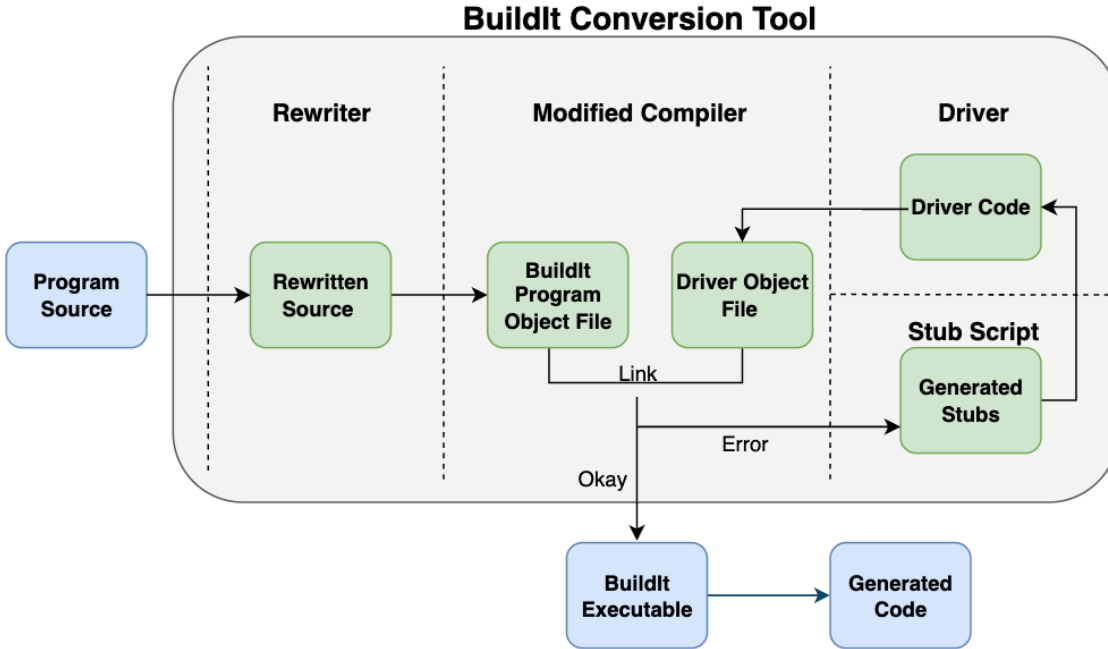


Figure 3.1: System diagram of the BuildIt conversion tool.

parses a program and assigns types, we modified the compiler to replace the assigned type `T` with `dyn_var<T>` and instantiate the `dyn_var<T>` template if needed. Importantly, the types must be modified as Clang parses them, since Clang very soon solidifies the AST and all types in the program.

After the Rewriter tool and modified compiler have run their transformations over the code, the code can be compiled and linked. However, user programs often call functions from standard C headers, and these function calls and their declarations have been modified to now take `dyn_var` arguments and return `dyn_var` types. The stub generator script produces definitions for such functions, simply returning a call to the original function in its body. Finally, the driver represents the last step in the BuildIt conversion process. The driver uses BuildIt’s API to generate code for a specific function, typically the main function, and also to generate code for any global variables or user-defined types (e.g. structs) that were present in the original program.

The result of the BuildIt conversion tool is a program where all types are wrapped in

`dyn_var`, with an accompanying driver to generate code. Running the executable generates standard C/C++ code that is equivalent to the original source. A system diagram of the conversion tool is shown in figure 3.1.

We summarize the process for converting a program to BuildIt as follows:

1. Preprocessed files are generated from the original source files. This can be done by using the `-E` flag in Clang. This resolves any pre-processing directives in the code, including in code from header files.
2. The Clang rewriter tool is run on each of the pre-processed files, generating rewritten files.
3. The modified Clang compiler is used to produce object files for each of the rewritten files and the driver code. The object files are then linked together to produce an executable.
4. At the linking step, errors may be emitted due to missing definitions for functions from header files. A Python script is fed the linker errors to produce a header file with generated stubs.
5. The modified Clang compiler is used to produce an object file for the driver code with the generated header file force included in. Clang supports force including headers using the `-include` flag.
6. All of the object files are linked together to produce an executable. The executable contains the result of converting the original program to BuildIt (by wrapping all of the types with `dyn_var`). The executable can be run to generate C/C++ code equivalent to the original source.

3.2 System Motivation

It is worth discussing why we take this complicated approach to source modifications rather than just using a Clang Plugin or Clang Rewriter tool. Indeed, using Clang's tooling

```

1 int add(int x, int y){
2     int res = x + y;
3     return res;
4 }
5
6
7 FunctionDecl add 'int (int, int)'
8   |-ParmVarDecl used x 'int'
9   |-ParmVarDecl used y 'int'
10  '-CompoundStmt
11    |-DeclStmt
12      |-VarDecl used res 'int' cinit
13        '-BinaryOperator 'int' '+'
14          |-ImplicitCastExpr <LValueToRValue>
15            |-DeclRefExpr 'int' lvalue ParmVar 'x' 'int'
16              '-ImplicitCastExpr 'int' <LValueToRValue>
17                '-DeclRefExpr 'int' lvalue ParmVar 'y' 'int'
18          '-ReturnStmt
19            '-ImplicitCastExpr 'int' <LValueToRValue>
20              '-DeclRefExpr 'int' lvalue Var 'res' 'int'

```

Figure 3.2: An add function along with its generated AST.

would be much simpler than making modifications to the internals of the Clang compiler, which requires significant knowledge of Clang’s extensive code-base. We briefly discuss prior approaches taken with Clang tooling, and why the tooling is inadequate to perform our desired modifications to the source.

A Clang Plugin enables running user-defined actions on a program during its compilation. Our first approach relied on writing a Clang Plugin that used a RecursiveASTVisitor Class to visit variable and function declaration nodes in the program’s AST. The Plugin code wrapped these nodes’ type with `dyn_var`. Figures 3.2, 3.3, and 3.4 show the original, transformed, and intended AST for a simple add program that the Clang Plugin transforms.

While the Clang Plugin does successfully change the type of the function’s parameters and return value in 3.3, the AST nodes in the function body do not change to reflect the parameters’ new type. For instance, the function body’s AST nodes still refer to parameter `x` and `y`’s type as `int`. In fact, the result of the Plugin’s AST modification shown in 3.3 differs from the intended result shown in 3.4. We found that while the Clang Plugin does successfully wrap types with `dyn_var`, Clang has already produced a solidified AST at this

```

1 FunctionDecl add 'builder::dyn_var<int> (builder::dyn_var<int>, builder
  ::dyn_var<int>)'
2 |-ParmVarDecl used x 'int'
3 |-ParmVarDecl used y 'int'
4 '-CompoundStmt
5   |-DeclStmt
6   | '-VarDecl used res 'int' cinit
7   |   '-BinaryOperator 'int' '+'
8   |     |-ImplicitCastExpr 'int' <LValueToRValue>
9   |     | '-DeclRefExpr 'int' lvalue ParmVar 'x' 'int'
10  |     | '-ImplicitCastExpr 'int' <LValueToRValue>
11  |     | '-DeclRefExpr 'int' lvalue ParmVar 'y' 'int'
12  '-ReturnStmt
13    '-ImplicitCastExpr 'int' <LValueToRValue>
14    '-DeclRefExpr 'int' lvalue Var 'res' 'int'

```

Figure 3.3: The AST produced by running the Clang Plugin on the add function shown in figure 3.2.

point in the compilation process. Changes to specific nodes have an isolated effect, where other AST nodes are not updated to reflect the new template types of variables. In fact, replacing primitive types with the `dyn_var` template type necessitates the creation of new AST nodes, as can be seen in 3.4, which is not possible with a Clang Plugin. Thus, using simply a Clang Plugin is infeasible to perform our desired source modifications.

In our second approach, we utilized a Clang Tool—a stand-alone program that can perform user-defined actions on code. Our Clang Tool was composed of a `RecursiveASTVisitor` Class and a `Source Rewriter` Class, enabling the program to visit specific AST nodes and perform source rewrites to their contents. As this Clang Tool directly modifies and emits source text as output, it does not encounter the same issue with the Clang Plugin. The Clang Tool simply wraps the `dyn_var` text around types in the source text, and then emits rewritten source text which can then be compiled.

The flow of our Clang Tool concretely looks as follows: Clang parses the program and produces an AST; the `RecursiveASTVisitor` visits the `VarDecl` and `FunctionDecl` AST nodes (corresponding to variable and function declarations, respectively); the `Rewriter` extracts the type in the AST node and replaces it with the text "`dyn_var<T>`" where `T` refers to the original type; the tool then emits the modified C/C++ source text, which can be compiled. While

```

1 dyn_var<int> add(dyn_var<int> x, dyn_var<int> y){
2     int res = x + y;
3     return res;
4 }
5
6 FunctionDecl add 'dyn_var<int> (dyn_var<int>, dyn_var<int>)'
7 |-ParmVarDecl used x 'dyn_var<int>':'dyn_var<int>' destroyed
8 |-ParmVarDecl used y 'dyn_var<int>':'dyn_var<int>' destroyed
9 '-CompoundStmt
10   |-DeclStmt
11   |   '-VarDecl used res 'dyn_var<int>':'dyn_var<int>' nrvo cinit
12         destroyed
13   |   '-ExprWithCleanups 'dyn_var<int>':'dyn_var<int>'
14   |   '-ImplicitCastExpr 'dyn_var<int>':'dyn_var<int>'
15         <ConstructorConversion>
16   |   '-CXXConstructExpr 'dyn_var<int>':'dyn_var<int>' 'void
17         (const builder &)' noexcept(false)
18   |   ...
19   |   ...
20   |   '-ImplicitCastExpr 'const dyn_var<int>':
21         'const dyn_var<int>' lvalue <NoOp>
22   |   '-DeclRefExpr 'dyn_var<int>':'dyn_var<int>'
23         lvalue ParmVar 'x' 'dyn_var<int>'
24   |   '-ImplicitCastExpr 'const dyn_var<int>':
25         'const dyn_var<int>' lvalue <NoOp>
26   |   '-DeclRefExpr 'dyn_var<int>':'dyn_var<int>'
27         lvalue ParmVar 'y' 'dyn_var<int>'
28   '-ReturnStmt
29     '-CXXConstructExpr 'dyn_var<int>':'dyn_var<int>' 'void
30       (const dyn_var<int> &)'
31     '-ImplicitCastExpr 'const dyn_var<int>':'const dyn_var<int>'
32       xvalue <NoOp>
33     '-DeclRefExpr 'dyn_var<int>':'dyn_var<int>'
34       lvalue Var 'res' 'dyn_var<int>'
35
36

```

Figure 3.4: A BuildIt add function and its (truncated) generated AST. This is the intended result of performing source-modifications on the code in figure 3.2. Note, minor reformatting was performed to improve visualization.

<pre> 1 int power(int base, int exponent){ 2 int res = 1; 3 int x = base; 4 5 while (exponent > 0){ 6 if (exponent % 2 == 1){ 7 res = res * x; 8 } 9 10 x = x * x; 11 exponent = exponent / 2; 12 } 13 14 return res; 15 } 16 </pre>	<pre> 1 #include "builder/dyn_var.h" 2 builder::dyn_var<int> power(3 builder::dyn_var<int> base, 4 builder::dyn_var<int> exponent){ 5 6 builder::dyn_var<int> res = 1; 7 builder::dyn_var<int> x = base; 8 while (exponent > 0){ 9 if (exponent % 2 == 1){ 10 res = res * x; 11 } 12 x = x * x; 13 exponent = exponent / 2; 14 } 15 return res; 16 } </pre>
---	---

Figure 3.5: A simple power program is shown on the left, and the emitted source code from running the Clang Tool on it is shown on the right.

this approach does correctly rewrite basic C/C++ code as shown in figure 3.5, numerous edge cases quickly arise as is shown in figure 3.6. In the latter example, there are two variables `res` and `x` declared in the same statement. Since both variables' type comes from the same text `"int"`, this text gets wrapped with `dyn_var` twice.

Though this edge-case was simple to fix, it highlights the delicate nature of source-rewrites and that other patches will likely be needed to fix similar issues. For instance, we find the text `"int x, *y;"` also needs special care as variable `y`'s type appears in two locations, so the rewrite `"dyn_var<int> x, *y;"` would be incorrect. We found the number of edge cases (involving global variables, unnamed structs, etc.) quickly blows up, rendering this Clang Rewriting Tool approach hacky and infeasible.

We therefore conclude that Clang tooling is insufficient to perform our type-based source modifications. This motivates our desire to modify the Clang compiler, as it enables modifying the types internally as they are parsed, avoiding the issue the Clang Plugin faced with types being modified too late or the Clang Rewriting Tool's issue with modifying types through complicated source-rewrites. The BuildIt conversion tool will still need additional components though. Namely, the conversion tool will utilize a Clang Rewriter Tool to perform source-

<pre> 1 int power(int base, int exponent){ 2 int res = 1, x = base; 3 4 while (exponent > 0){ 5 if (exponent % 2 == 1){ 6 res = res * x; 7 } 8 9 x = x * x; 10 exponent = exponent / 2; 11 } 12 13 return res; 14 } 15 16 </pre>	<pre> 1 #include "builder/dyn_var.h" 2 builder::dyn_var<int> power(3 builder::dyn_var<int> base, 4 builder::dyn_var<int> exponent){ 5 6 builder::dyn_var<int> lder:: 7 dyn_var<int> res = 1, x = base; 8 while (exponent > 0){ 9 if (exponent % 2 == 1){ 10 res = res * x; 11 } 12 x = x * x; 13 exponent = exponent / 2; 14 } 15 return res; 16 } </pre>
--	---

Figure 3.6: A simple power program is shown on the left, and the emitted source code from running the Clang Tool on it is shown on the right. Declaring multiple variables in a single statement results in an error with the Clang Tool, as can be seen in the first line of the function body on the right.

rewrites to support features like recursive functions, structs, and global variables for which the conversion process involves more than just wrapping types with `dyn_var`.

Now that we have provided the motivation for our system’s design, we can discuss its implementation in the following sections.

3.3 System Components

3.3.1 Clang Rewriter Tool

The BuildIt conversion process begins with a Clang Rewriter Tool that performs source rewrites on the user’s original code. Like the Clang Tool described earlier, this Rewriter Tool combines a RecursiveASTVisitor with a Rewriter class. The RecursiveASTVisitor class enables visiting specific nodes in the user program’s AST. These AST nodes can correspond to things such as variable declarations, function declarations, or struct members, and are represented by Clang classes which store a significant amount of metadata, such as the node’s source location, identifier name, etc. The Clang Rewriter class uses this information to insert,

modify, or delete parts of the source text corresponding to the AST node. The Rewriter Tool finally emits the modified C/C++ source as output.

To prepare the user's code for the modified Clang compiler, the Rewriter Tool performs four main transformations:

- It includes in the `dbd_support.h` header at the top of the code. The header file is shown in figure 3.7. `dbd_support.h` includes in the BuildIt headers, so that the BuildIt `dyn_var` type will be defined when it is later used by the modified Clang compiler. The header also contains the declaration `extern dyn_var<int> __dummy_dynamic_by_default`, so that the modified Clang compiler can extract the `dyn_var` template class from a dummy variable while parsing. Lastly, `dbd_support.h` defines a `struct register_function`. This struct is used to generate code for functions in the driver. In particular, we instantiate a `struct register_function` variable with a function and its name. At runtime, this adds the function to the `registered_functions` list defined in the driver, allowing the driver to generate code for those functions.
- It encapsulates the original code in the `buildit_application` namespace. While the types in a user's program and user-included headers should be modified, the BuildIt headers inserted by the Rewriter Tool must not be altered by the modified Clang compiler. Thus, we denote the code to be altered by the modified Clang compiler with the `buildit_application` namespace.
- It registers the main function to be generated by the driver.
- It performs additional rewrites to support conversion of structs, global variables, etc. This is discussed in a later section.

Figure 3.8 shows the result of running the Rewriter Tool on a program. The Rewriter Tool is implemented as follows. First, it visits the `FunctionDecl` AST nodes using the `RecursiveASTVisitor` Class. If the node corresponds to the main function, it inserts an

```

1 #ifndef DBD_SUPPORT_H
2 #define DBD_SUPPORT_H
3
4 #include "builder/dyn_var.h"
5 #include "builder/builder.h"
6 #include "builder/builder_context.h"
7 #include "blocks/c_code_generator.h"
8 extern builder::dyn_var<int> __dummy_dynamic_by_default;
9
10 extern std::vector<std::function<block::block::Ptr(void)>> *
    registered_functions;
11 struct register_function {
12     template <typename T>
13     register_function(T fimpl, std::string fname) {
14         auto f = [=] (void) -> auto {
15             builder::builder_context context;
16             context.run_rce = true;
17             return context.extract_function_ast(fimpl, fname);
18         };
19         if (registered_functions == nullptr) {
20             registered_functions = new std::vector<std::function<block::
21 block::Ptr(void)>>();
22         }
23         registered_functions->push_back(f);
24     };
25 };
26 #endif

```

Figure 3.7: The code contained inside dbd_support.h.

instantiation of `struct register_function` (with `main` passed as argument) immediately after the `main` function's definition using the Clang Rewriter class. This registers the `main` function to have its code generated in the driver. After performing the rewrites, the Rewriting Tool emits an include statement for `dbd_support.h`, the text `"buildit_application {"`, followed by the modified source, and finally emits a `"}"` to close the `buildit_application` namespace. This includes the desired header and encapsulates the user-program and any user-included headers inside the `buildit_application` namespace. The result is modified C/C++ source code that is prepared to be compiled and altered by the modified Clang compiler in the next step.

<pre> 1 # 1 "thesis_example.cpp" 2 # 1 "<built-in>" 1 3 # 1 "<built-in>" 3 4 # 482 "<built-in>" 3 5 # 1 "<command line>" 1 6 # 1 "<built-in>" 2 7 # 1 "thesis_example.cpp" 2 8 9 int power (int base, 10 int exponent) { 11 int res = 1; 12 for (int i = 0; i < exponent; 13 i++) { 14 res *= base; 15 } 16 return res; 17 } 18 int main(int argc, 19 char* argv[]) { 20 int x = power(5, 2); 21 return 0; 22 } 23 24 25 26 </pre>	<pre> 1 #include "dbd_support.h" 2 3 namespace buildit_application { 4 # 1 "thesis_example.cpp" 5 # 1 "<built-in>" 1 6 # 1 "<built-in>" 3 7 # 482 "<built-in>" 3 8 # 1 "<command line>" 1 9 # 1 "<built-in>" 2 10 # 1 "thesis_example.cpp" 2 11 12 int power (int base, int exponent) { 13 int res = 1; 14 for (int i = 0; i < exponent; i 15 ++) { 16 res *= base; 17 } 18 return res; 19 } 20 int main(int argc, char* argv[]) { 21 int x = power(5, 2); 22 return 0; 23 } 24 static register_function reg0(main, 25 "main"); </pre>
--	---

Figure 3.8: A preprocessed program is shown on the left, and the result of running the Clang Rewriter Tool on it is shown on the right.

3.3.2 Modified Clang C/C++ Compiler

After performing rewrites to the source code, the next step is to modify all the types in the user program during compilation. In this subsection, we describe the changes we make to the Clang compiler’s semantic analysis phase—where type checking and type assignment occurs—in order to wrap all types with the `dyn_var` template, as the code is parsed.

During the parsing of a declaration, the Clang compiler constructs a `Declarator` and calls `Sema::GetTypeForDeclarator()`, which computes a preliminary `QualType` (qualified-type) and produces a `TypeSourceInfo` capturing the fully elaborated type (including pointers, qualifiers, etc) along with the source-location data. The function returns the `TypeSourceInfo` object. By using LLDB, Clang’s version of GDB, we were able to step through Clang’s

parsing and verify that this function is responsible for assigning types for all declarations. Therefore, our modifications to the compiler focus on this function, particularly with the goal of changing the type stored in the `TypeSourceInfo` object.

Before the `Sema::GetTypeForDeclarator()` function can wrap types with `BuildIt`'s `dyn_var` template, it must first extract the template's information. The function is first modified to extract the `dyn_var` template name and template class from the declaration `extern builder::dyn_var<int> __dummy_dynamic_by_default` inserted by the Rewriter Tool. As the function parses declarations, it checks if it is parsing a variable with reserved identifier name `__dummy_dynamic_by_default`, extracting the template information if so. As this inserted declaration appears prior to the `buildit_application` namespace in the rewritten code, the function will be able to extract the `dyn_var` template information before it needs to modify any type.

`Sema::GetTypeForDeclarator()` is then modified to wrap the types in declarations with `dyn_var`. After the `TypeSourceInfo` (which contains the full-type information) is computed inside the function, we extract the declaration's original type from it. It is used to first instantiate the `dyn_var<T>` class template with the original type passed as template argument, and then create the desired `dyn_var` type by using the `dyn_var` template name, original type as template argument, and the new template instantiation. The new `dyn_var` type is used to override the original type present in the `TypeSourceInfo` object. As `Sema::GetTypeForDeclarator()` is responsible for type-assignment for all declarations, correct modification of this function results in all types in a program being wrapped with `dyn_var`.

The examples below show the changes made by the compiler to the type of various declarations:

- `int x` becomes `dyn_var<int> x`
- `int power(int, int)` becomes `dyn_var<int> power(dyn_var<int>, dyn_var<int>).`

Notice that the function's arguments and return type are wrapped with `dyn_var`

- `struct myType* x` becomes `dyn_var<struct myType*>`. Notice the entire compound type is wrapped with `dyn_var`

Type modification is not always performed on a declaration. In particular, it is only performed in `Sema::GetTypeForDeclarator()` if the declaration is in the `buildit_application` namespace. This can be determined by iterating over the declarator contexts encapsulating the declaration, and checking each for the `buildit_application` namespace. In addition, type-modification is not performed for `void` types, types from the `builder` namespace (this is BuildIt's namespace for its types and functions), the `register_function` struct type, or variable's with the name `type_name`. The `void` type is treated specially in BuildIt and does not need to be wrapped with `dyn_var`. In the latter three cases, no type-modification is performed as they are produced from source rewrites in the previous phase and should be left as is.

We encountered numerous bugs while implementing changes to the Clang compiler. This included:

- Not instantiating the `dyn_var` template class. We presumed creating a template type with a template name and template arguments would be sufficient, though it requires an explicit template instantiation beforehand.
- Wrapping the `QualType` (qualified-type) computed in the body of `Sema::GetTypeForDeclarator()` with `dyn_var`. This type is incomplete, not capturing elements of compound types such as pointers or arrays.
- Replacing `TypeSourceInfo`'s original type with the new `dyn_var` type without updating the meta-data. The `TypeSourceInfo` class captures not only the type, but the type's source-location information as well. Since the original type is replaced with a type that does not exist in the source, the source-location information becomes invalid. In a later pass, Clang traverses the constructed AST and accesses the invalid source-information, leading it to crash. As source-location metadata is only utilized to provide additional

information to the user (e.g. for debugging), we omit the code that accesses it without affecting the compiler's parsing.

We ran into many bugs because the Clang compiler's internal passes are not designed for modification, with little documentation for functions or an API available to perform user-defined actions. This makes modifying the Clang compiler incredibly difficult and error-prone, which was only exacerbated by the size and inter-dependency of the codebase. Changes require significant care, knowledge, and debugging effort on the part of the developer. Nevertheless, simple changes to Clang's compiler are incredibly powerful, as in our case modifications to a single function altered type-assignment of an entire program.

To summarize, the modified Clang compiler is used to parse the rewritten C/C++ programs, and produce object files where the types in the program have been converted to the BuildIt `dyn_var` type. This is the main goal of the conversion tool, and the remaining steps focus on generating stubs (for missing definitions) and code generation.

3.3.3 Stub Generator Script

User programs frequently include standard C and C++ library headers. Since types in the user program will be wrapped in `dyn_var`, arguments to external function calls will also change in type. This necessitates new function declarations and definitions for functions from C/C++ headers. The former is already accomplished by the first two steps of the conversion process. This is because the Rewriter Tool wraps any user-included header in the `buildit_application` namespace, so the modified Clang compiler will automatically change the header function's declarations to have `dyn_var` parameters and return type. This is shown in figure 3.9. However, we are still missing the corresponding function definitions for the new function declarations. Upon linking, this results in missing function definition errors thrown for any external functions called by the user program.

The purpose of the stub generator script is to create the necessary function definitions from the dumped linker errors. The stub generator script is specifically a Python program

<pre> 1 // declaration included from 2 <stdio.h> 3 int puts(const char *); 4 5 int main() { 6 char message[] = "Hello"; 7 puts(message); 8 return 0; 9 } 10 11 12 13 14 15 16 </pre>	<pre> 1 #include "dbd_support.h" 2 3 namespace buildit_application { 4 // declaration included from 5 // <stdio.h> 6 dyn_var<int> puts(dyn_var<const char 7 *>); 8 9 dyn_var<int> main() { 10 dyn_var<char []> message = "Hello 11 "; 12 puts(message); 13 return 0; 14 } 15 16 static register_function reg0(main, " 17 main"); 18 } </pre>
--	--

Figure 3.9: A simple program is shown on the left. The result of running the program through the Rewriter Tool and modified Clang compiler produces an object file for the code shown on the right. The puts() declaration's types are changed to match its function call, though it is missing a definition.

that uses regular expression matching to extract all the functions (and their associated arguments) with a missing definition in the linker errors. For each such function, it creates a definition that simply returns a call to `buildit_runtime::{function_name}`. The script then defines `buildit_runtime::{function_name}` using BuildIt's `as_global`, informing BuildIt to replace calls to `buildit_runtime::{func_name}` with some given text when generating code. The script's generated definitions are outputted in a header file. Figure 3.9 produces the linker error shown in figure 3.10 regarding the puts() function. From this, the stub generator script produces the header file shown in figure 3.11. In this example, the call to `buildit_application::puts()` will now resolve as there is a matching function definition. Moreover, during BuildIt's code generation phase, `buildit_runtime::puts` is replaced with the text "puts", so a call to `buildit_application::puts()` will simply return a call to puts(). Any external function calls in the program will thus link and produce a call to the original external function in the generated code.

```

1 ld: Undefined symbols:
2   buildit_application::puts(builder::dyn_var<char const*>), referenced
3   from:
4       buildit_application::power(builder::dyn_var<int>,
5       builder::dyn_var<int>) in power_rewriter.o
6 clang: error: linker command failed with exit code 1 (use -v to see
7 invocation)

```

Figure 3.10: An example of a linker error dumped from compiling the program shown in 3.9.

```

1 #include "builder/dyn_var.h"
2 #include "blocks/c_code_generator.h"
3 using builder::dyn_var;
4
5 // Generated stubs for undefined linker symbols
6
7 namespace buildit_runtime {
8 static dyn_var<int(const char*)> puts = builder::as_global("puts");
9 }
10
11 namespace buildit_application {
12 dyn_var<int> puts(dyn_var<const char*> x0) {
13     return buildit_runtime::puts(x0);
14 }
15 }

```

Figure 3.11: The header file generated by our script based on the linker error message in 3.10.

```

1 #include "builder/dyn_var.h"
2 #include "blocks/c_code_generator.h"
3
4 using builder::dyn_var;
5
6 std::vector<std::function<block::block::Ptr(void)>> *
   registered_functions;
7 int main(int argc, char* argv[]) {
8     // manually print out any headers
9     std::cout << "#include <stdio.h>\n\n";
10    // manually print any global variable definitions
11    std::cout << "int x;\n";
12    // manually print any struct definitions
13    std::cout << "struct my_type {\nint z;\n};\n";
14
15    for (auto f: *registered_functions) {
16        auto ast = f();
17        block::c_code_generator::generate_code(ast, std::cout, 0);
18    }
19    return 0;
20 }

```

Figure 3.12: The driver code, used to generate code for functions, struct-types, and global variables, is shown.

3.3.4 Driver Code

The driver is the last component of the BuildIt conversion tool. The driver generates code for functions and user-defined types in the user's program, after all types in the program have been wrapped with BuildIt's `dyn_var`. The driver invokes BuildIt's `generate_code` function to do so, producing C/C++ code resulting from execution of BuildIt's static stage. Figure 3.12 shows the contents of the driver.

The `registered_functions` list stores all of the functions to generate code for. Functions are registered by declaring a `register_function` struct (defined in `dbd_support.h`) with the function provided as argument. At runtime, the struct's constructor is called, which appends the function to the `registered_functions` list in the driver. The driver generates code for each function in this list. By default, code will be generated for the main function as the Rewriter Tool automatically registers it. The user can specify additional functions to generate code for by simply adding an `annotate("buildit_outline")` attribute to the given

function's declarations and definition in the user program. Alternatively, the driver code can be modified to invoke `generate_code` on additional functions.

The driver also generates code for struct definitions and global variables present in the user program. Currently, the user must print definitions for global variables and structs manually in the driver. However, there is work being done in BuildIt to support automatically generating struct and global variable definitions. The driver must also manually print out any headers the program needs.

3.4 Support for Various Features

In this section, we discuss changes made to the Rewriter Tool to support various C/C++ language features. These features require special treatment, as they cannot simply be wrapped with the `dyn_var` type.

3.4.1 Structs

Struct types are a commonly used programming construct in C and C++ programs, enabling users to group multiple variables into a single type. A struct type must first be defined in a program, and can then be used to instantiate variables of that type. BuildIt handles struct types specially. In addition to wrapping any variables with struct type, the struct definition also needs to be modified. This latter modification is necessary as BuildIt generates code for user-defined types. The necessary modifications are illustrated in figure 3.13. In particular, the struct definition is modified to include a static member `type_name` which is initialized to the struct's name. This ensures that the struct retains its original name when BuildIt generates code for it. (BuildIt otherwise generates an arbitrary name for the struct). Each struct member's type is also wrapped with `dyn_var`, and the member is set to `builder::with_name(member_name)`. This ensures the struct member has the same name in the generated code.

<pre> 1 struct my_type { 2 int *x; 3 }; 4 5 int main() { 6 struct my_type hello; 7 return 0; 8 } 9 10 11 </pre>	<pre> 1 struct my_type { 2 static constexpr const char* type_name = 3 "my_type"; 4 dyn_var<int*> x = 5 builder::with_name("x"); 6 }; 7 8 dyn_var<int> main() { 9 dyn_var<struct my_type> hello; 10 return 0; 11 } </pre>
---	---

Figure 3.13: The left and right image show a program and its equivalent code in BuildIt, respectively. This shows BuildIt’s treatment of struct types.

The conversion tool supports converting structs to BuildIt as follows. In the first phase, the Rewriter Tool inserts the static member `type_name` and initializes each struct member to `builder::with_name(member_name)`. In the second phase, the modified Clang compiler automatically wraps the type of struct members with the `dyn_var` template. It also wraps the type of any variables with struct type with `dyn_var`. However, this approach runs into an issue when transforming recursively defined structs (i.e. structs that have a member with type pointing to the same struct). When BuildIt attempts to generate code for a recursively defined struct, it will parse each of the struct’s members, including the struct member with type pointing to the same struct. BuildIt errors here as the struct member’s type appears incomplete to it (since BuildIt is still in the process of parsing the struct).

Therefore, the approach for converting structs needs to be modified. In the new approach, The Rewriter Tool performs the following source modifications for each struct it traverses:

- It first initializes each struct member to `builder::with_name(member_name)`.
- Immediately after each struct’s first declaration or definition in a file, it closes the `buildit_application` namespace, opens the `builder` namespace, and creates a struct `external_type_namer` with a static member `type_name`. The `external_type_namer` interface was specifically added to BuildIt to avoid code generation issues for recursive struct definitions, as it changes BuildIt’s parsing and registry of structs.

```

1 #include "dbd_support.h"
2
3 namespace buildit_application {
4 struct my_type {
5     int* x = builder::with_name("x");
6 };
7 }
8 namespace builder {
9 template <>
10 struct external_type_namer<buildit_application::my_type> {
11 static constexpr const char* type_name = "struct my_type";
12 };
13 }
14 namespace buildit_application {
15
16 int main() {
17     struct my_type hello;
18     return 0;
19 }
20 static register_function reg0(main, "main");
21
22 }

```

Figure 3.14: The result of applying the Rewriter Tool to the program shown on the left in 3.13.

- The `builder` namespace is closed and the original `buildit_application` namespace is opened.

The new approach is depicted in figure 3.14. After the rewrites, the modified Clang compiler further transforms the rewritten code, wrapping the types of struct members and any variables with struct type with the `dyn_var` template. Finally, the driver generates code for struct definitions seen in the user program.

The Rewriter Tool is able to make the modifications described above by traversing RecordDecl AST nodes and checking if the node corresponds to a struct. If so, it iterates over the struct's members and inserts the text corresponding to their initializations, and finally emits text at the end of the struct's definition for creating the `struct external_type_namer`.

3.4.2 Variadic Functions

Variadic functions—functions with an unspecified number of arguments—are frequently called by user programs. A notable example of a variadic function defined in a standard C/C++ header is `printf()`. While BuildIt does not allow users to define variadic functions themselves, it does allow for calling externally defined variadic functions (such as `printf`). However, BuildIt cannot simply take a declaration of the form `int printf(const char * , ...)` and produce the equivalent BuildIt declaration by wrapping the arguments and return type with `dyn_var`.

The solution for supporting variadic functions is the following. The Rewriter Tool deletes all variadic function declarations in the user program. After performing source rewrites, the Rewriter Tool emits the variadic function names defined with `builder::as_global` at the top of the `buildit_application` namespace, along with the usually emitted text. An example of the source rewrites performed by the tool is shown in figure 3.15. These changes enable the user program to successfully call external variadic functions. Deleting the variadic function declarations ensures the modified Clang compiler does not crash from wrapping the ellipsis argument with `dyn_var`. Furthermore, any calls to external variadic functions will resolve as these functions have an `as_global` declaration. During code generation, BuildIt replaces calls to these external functions with the text passed to `as_global` (i.e. the function’s name). This results in the generated code successfully calling the external variadic function with no issues.

The Rewriter tool makes the described changes by visiting the `FunctionDecl` node and checking if the declaration is variadic. If so, it deletes the declaration and adds the function’s name to a set. At the end of the rewriting process, it emits a definition with BuildIt’s `as_global` for each function in this set.

<pre> 1 // included in from stdio.h 2 int printf(const char * , ...); 3 4 int main() { 5 printf("Hello"); 6 return 0; 7 } 8 9 10 11 12 13 14 15 </pre>	<pre> 1 #include "dbd_support.h" 2 3 namespace buildit_application { 4 // included in from stdio.h 5 builder::dyn_var<void(void)> printf = 6 builder::as_global("printf"); 7 ; 8 9 int main() { 10 printf("Hello"); 11 return 0; 12 } 13 static register_function reg0(main, 14 "main"); 15 } </pre>
--	--

Figure 3.15: The left shows a simple program that calls the variadic function `printf`. The right shows the emitted output from running the Rewriting Tool on the program on the left.

3.4.3 Recursive Functions

Recursive functions express complicated computation as solving a number of easier sub-problems, making them a particularly powerful programming tool. However, BuildIt cannot generate code for recursive functions with the usual approach. As BuildIt explores all control flow paths through repeated execution, this leads it to become stuck in infinite recursion when evaluating the recursive function.

Hence, recursive functions are treated specially. The user must first add a `buildit_outline` annotation attribute to any recursive function declarations or definitions in their user program. The Rewriter Tool then performs the following modifications to source:

- It appends the text `"_buildit_impl"` to the recursive function's name in all function declarations and definitions. This effectively renames the recursive function.
- Immediately before the first declaration or definition for a recursive function in each file, it inserts a static definition for the function with BuildIt's `as_global`. This ensures calls to the originally-named function still resolve, as the function's declarations/definitions have all been renamed.

<pre> 1 int __attribute__((annotate(" buildit_outline"))) factorial(int x) { 2 if (x == 0) return 1; 3 return x * factorial(x - 4 1); 5 } 6 7 int main(int argc, 8 char* argv[]) { 9 int x = factorial(5); 10 return 0; 11 } 12 13 14 15 16 17 18 19 20 </pre>	<pre> 1 #include "dbd_support.h" 2 3 namespace buildit_application { 4 static builder::dyn_var<int (int)> 5 factorial = 6 builder::as_global("factorial"); 7 8 int __attribute__((annotate(" 9 buildit_outline"))) 10 factorial_buildit_impl(int x) { 11 if (x == 0) return 1; 12 return x * factorial(x - 1); 13 } 14 15 static register_function reg0(16 factorial_buildit_impl, "factorial"); 17 18 int main(int argc, char* argv[]) { 19 int x = factorial(5); 20 return 0; 21 } 22 23 static register_function reg1(main, 24 "main"); 25 } </pre>
---	---

Figure 3.16: A recursive program and the result of performing source rewrites on it is shown.

- Immediately after a recursive function's definition, it inserts a `static struct register_function` initialized with the function. This registers the recursive function to have its code generated in the driver.

The Rewriter makes these changes by visiting the `FunctionDecl` node and checking if it has the `buildit_outline` annotation. If so, it renames the function and inserts any text before/after the node if necessary. Figure 3.16 shows the result of performing rewrites on a program with a recursive function.

When the final BuildIt code is executed, any calls to the originally-named recursive function resolve due to its `as_global` definition. Importantly, BuildIt does not try to further evaluate the call and instead replaces it with the text in `as_global`'s argument. This produces a call to the originally-named function in the generated code. The driver generates code for the recursive function's definition using its original name, thus resulting in a valid program.

<pre> 1 2 int x; 3 double z = 5, *y; 4 char c = 'a'; 5 6 7 8 9 </pre>	<pre> 1 #include "dbd_support.h" 2 3 namespace buildit_application { 4 5 int x = builder::as_global("x"); 6 double z = builder::as_global("z"), 7 *y = builder::as_global("y"); 8 char c = builder::as_global("c"); 9 } </pre>
---	--

Figure 3.17: A program with global variables, and the result of running the Rewriter Tool on it.

3.4.4 Global Variables

Global variables have the greatest possible scope in a program. To support this programming feature, BuildIt allows declaring variables with `as_global` in the global scope. During compilation of the BuildIt program, this resolves all references to the variable. In the generated code, BuildIt replaces use of the variable with the text passed as argument to `as_global`. Thus, the solution for supporting global variables is relatively straightforward:

- The Rewriter Tool visits the VarDecl AST nodes. If the variable is defined in the global scope, we insert an initialization for it using `as_global` (with the name of the variable passed as argument). The original initialization is deleted if it exists. This ensures BuildIt can resolve references to the global variable during compilation, and BuildIt generates code where references to the global variable are replaced with the same text. Note, we skip extern variable declarations in the global scope.
- The driver generates definitions for global variables in the generated code. Declaring a variable with `as_global` only resolves references to it in the BuildIt code, but does not produce a definition for it in the generated code. The driver must therefore generate definitions for global variables (which it does so currently by printing them out manually).

Figure 3.17 shows an example of the source-rewrites performed on a global variable.

3.5 Challenges and Limitations

Although Clang’s infrastructure was crucial to the development of the BuildIt conversion tool, it imposed a significant bottleneck on our project. First and foremost, Clang tooling’s API exposes very limited methods for making significant source modifications. The API mostly enables performing checks on code or making surface level modifications, such as refactoring. Since the Clang API was insufficient, our approach shifted to modifying the compiler. However, this was an incredibly esoteric process due to the little documentation, large size and interdependency of the compiler’s codebase. It becomes readily apparent that the Clang’s compiler is not designed to be modified and that in general, it is quite difficult to perform actions not intended by the Clang developers. Having greater Clang support for performing source modifications in the future would greatly improve our ability to systematically convert code to the BuildIt framework.

There are also limitations to our current conversion approach. Namely, we use source rewrites for each programming feature the conversion tool supports. As discussed previously, source rewrites are quite tricky to perform due to the numerous edge cases they induce, increasing the likelihood of bugs in our conversion process. This becomes an even greater issue as each subsequent C or C++ programming feature we add support for will require performing additional rewrites. While using source rewrites are unideal, it is necessary as converting most programming features to BuildIt requires most substantial changes than just wrapping types with `dyn_var`. It is also worth mentioning that while our BuildIt conversion tool does save significant effort on the part of the user, it still requires some manual changes to convert code to BuildIt. In particular, users must add annotations to recursive functions, and manually print out global variable and struct definitions along with any required headers in the driver. Lastly, the BuildIt conversion tool currently only supports a subset of the C and C++ programming language. It can not convert programs with classes, templates, and union types to the BuildIt framework.

Chapter 4

Results and Evaluation

In this chapter, we show the step-by-step conversion of sample programs to BuildIt using the conversion tool, and then conclude with a brief evaluation.

4.1 Results

4.1.1 Example 1

Figure 4.1 shows a user program that contains a struct, global variable, and a call to the variadic function `printf`. The user program also contains the recursive function `factorial`, that the user has added the `buildit_outline` annotation attribute to. As the first step, the user's code is preprocessed to produce the program shown in figure 4.2.

Then, the Rewriter Tool is run on the preprocessed program to produce the code shown in figure 4.3. This code is then compiled with the modified Clang compiler to produce an object file. The accompanying driver code shown in figure 4.4 is similarly compiled to produce an object file.

Finally, both object files are linked together to produce an executable. Note there are no linker errors since the user program does not make any external function calls to any non-variadic functions. Running the executable generates the code shown in figure 4.5, which

```

1 #include <stdio.h>
2
3 int x;
4 struct my_type {
5     int z;
6 };
7
8 int __attribute__((annotate("buildit_outline"))) factorial(int y) {
9     if (y == 0) return 1;
10    return y * factorial(y - 1);
11 }
12
13 int main() {
14     x = factorial(5);
15     my_type struct1;
16     struct1.z = x;
17     printf("%d", struct1.z);
18     return 0;
19 }

```

Figure 4.1: A user program that calls the recursive function factorial, and makes use of global variables, structs, and variadic functions.

```

1 // included from stdio.h
2 int printf(const char * , ...) __attribute__((__format__ (__printf__,
3 // ...
4
5 int x;
6 struct my_type {
7     int z;
8 };
9
10 int __attribute__((annotate("buildit_outline"))) factorial(int y) {
11     if (y == 0) return 1;
12     return y * factorial(y - 1);
13 }
14
15 int main() {
16     x = factorial(5);
17     my_type struct1;
18     struct1.z = x;
19     printf("%d", struct1.z);
20     return 0;
21 }

```

Figure 4.2: The result of the preprocessing the program shown in 4.1. We have truncated the code included from stdio.h header to only include the relevant printf declaration, for display purposes.

```

1 #include "dbd_support.h"
2
3 namespace buildit_application {
4 builder::dyn_var<void(void)> printf = builder::as_global("printf");
5
6 int x = builder::as_global("x");
7 struct my_type {
8     int z = builder::with_name("z");
9 };
10 }
11 namespace builder {
12 template <>
13 struct external_type_namer<buildit_application::my_type> {
14 static constexpr const char* type_name = "struct my_type";
15 };
16 }
17 namespace buildit_application {
18
19 static builder::dyn_var<int (int)> factorial = builder::as_global("
    factorial");
20 int __attribute__((noinline)) factorial_buildit_impl(int y) {
21     if (y == 0) return 1;
22     return y * factorial(y - 1);
23 }
24 static register_function reg0(factorial_buildit_impl, "factorial");
25
26
27 int main() {
28     x = factorial(5);
29     my_type struct1;
30     struct1.z = x;
31     printf("%d", struct1.z);
32     return 0;
33 }
34 static register_function reg1(main, "main");
35 }

```

Figure 4.3: The result of performing source rewrites to the program shown in 4.2.

```

1 #include "builder/dyn_var.h"
2 #include "blocks/c_code_generator.h"
3
4 using builder::dyn_var;
5
6 std::vector<std::function<block::block::Ptr(void)>> *
   registered_functions;
7 int main(int argc, char* argv[]) {
8     // manually print out any headers
9     std::cout << "#include <stdio.h>\n\n";
10    // manually print any global variable definitions
11    std::cout << "int x;\n";
12    // manually print any struct definitions
13    std::cout << "struct my_type {\nint z;\n};\n";
14
15    for (auto f: *registered_functions) {
16        auto ast = f();
17        block::c_code_generator::generate_code(ast, std::cout, 0);
18    }
19    return 0;
20 }

```

Figure 4.4: The driver for the program in 4.1.

is equivalent to the code shown in figure 4.1.

4.1.2 Example 2

In this example, we convert an interpreter for the BF language [16] to BuildIt. The BF language consists of 8 characters in its grammar, and a sequence of these 8 characters constitutes a BF program. Our BF interpreter reads a program as input and sequentially executes each character according to the BF language. This simple interpreter is adapted from [17], and contains global variables, structs, and a call to the variadic function `printf`. We show the original program, the result of preprocessing and rewriting it, its driver, and the final generated code below.

We choose to convert a BF interpreter to BuildIt as it is an ideal candidate for specialization. [6] shows that if the input program and program counter are set as static variables in a BuildIt BF interpreter, the generated program will be a BF compiler. In our case, the conversion tool automatically converts the BF interpreter to BuildIt (with all variables set to

```

1 #include <stdio.h>
2
3 int x;
4 struct my_type {
5     int z;
6 };
7 int factorial (int arg0) {
8     if (arg0 == 0) {
9         return 1;
10    }
11    int var2 = arg0 * factorial(arg0 - 1);
12    return var2;
13 }
14
15 int main (void) {
16     x = factorial(5);
17     struct my_type var0;
18     var0.z = x;
19     printf("%d", var0.z);
20     return 0;
21 }

```

Figure 4.5: The generated code produced from the BuildIt program’s executable. This should be equivalent to the code in 4.1.

execute in the dynamic stage). Thus, with some manual user effort or through an extension to this project, the appropriate variables in the BuildIt BF interpreter can be staged (by setting them as static) to produce a BF compiler. This shows both the power of specialization and the usefulness of our tool.

```

1 // Original BF Interpreter
2 #include <stdio.h>
3
4 #define MAX_PROG 65536
5 #define MAX_CELLS 65536
6
7 struct BFIInst {
8     char cmd;
9     BFIInst* next;
10    BFIInst* jmp;
11 };
12
13 struct Mem {
14     char val;
15     Mem* next;
16     Mem* prev;
17 };
18

```

```

19 // zero-initialized pools
20 static BFIInst prog_pool[MAX_PROG];
21 static int      prog_count = 0;
22
23 static Mem      cell_pool[MAX_CELLS];
24 static int      cell_count = 0;
25
26 static BFIInst* alloc_inst() {
27     return (prog_count < MAX_PROG)
28         ? &prog_pool[prog_count++]
29         : NULL;
30 }
31
32 static Mem* alloc_cell() {
33     return (cell_count < MAX_CELLS)
34         ? &cell_pool[cell_count++]
35         : NULL;
36 }
37
38 int main(int argc, char* argv[]) {
39     if (argc < 2) {
40         printf("Usage: %s \"<bf-program>\"\n", argv[0]);
41         return 1;
42     }
43
44     // 1) Read program from argv[1]
45     const char *program = argv[1];
46
47     // 2) Build the linked list of instructions
48     BFIInst *p = NULL, *n = NULL, *j = NULL, *pgm = NULL;
49     for (const char *src = program; *src; ++src) {
50         char ch = *src;
51         int valid = (ch=='<' || ch=='>' || ch=='+' || ch=='-' ||
52                     ch==',' || ch=='.' || ch=='[' || (ch==']'&&j));
53         if (!valid) continue;
54
55         n = alloc_inst();
56         if (!n) {
57             printf("Error: program too large\n");
58             return 1;
59         }
60         // link into list
61         if (p) p->next = n; else pgm = n;
62         n->cmd = ch;
63         n->next = NULL;
64         n->jmp = NULL;
65         p = n;
66
67         if (ch == '[') {
68             n->jmp = j;
69             j = n;
70         }
71         else if (ch == ']') {
72             n->jmp = j;

```

```

73         j                = j->jmp;
74         n->jmp->jmp        = n;
75     }
76 }
77 // discard unmatched '['
78 while (j) {
79     p        = j;
80     j        = j->jmp;
81     p->jmp    = NULL;
82     p->cmd    = ' ';
83 }
84
85 // 3) Allocate the initial tape cell
86 Mem *m = alloc_cell();
87 if (!m) {
88     printf("Error: tape out of memory\n");
89     return 1;
90 }
91 // m->val == 0, m->next == m->prev == NULL by zero-init
92
93 // 4) Execute, printing output with printf
94 for (n = pgm; n; n = n->next) {
95     if (n->cmd == '+') {
96         ++m->val;
97     } else if (n->cmd == '-') {
98         --m->val;
99     } else if (n->cmd == '.') {
100         printf("%c", m->val);
101     } else if (n->cmd == ',') {
102         continue;
103     } else if (n->cmd == '[') {
104         if (m->val == 0) n = n->jmp;
105     } else if (n->cmd == ']') {
106         if (m->val != 0) n = n->jmp;
107     } else if (n->cmd == '<') {
108         if (!(m = m->prev)) {
109             printf("Error: at start of tape\n");
110             return 1;
111         }
112     } else if (n->cmd == '>') {
113         if (!m->next) {
114             Mem *c = alloc_cell();
115             if (!c) {
116                 printf("Error: tape out of memory\n");
117                 return 1;
118             }
119             c->prev = m;
120             c->next = NULL;
121             m->next = c;
122         }
123         m = m->next;
124     }
125 }
126

```

```

127     return 0;
128 }

1 // Preprocessed and Rewritten BF Interpreter
2 #include "dbd_support.h"
3
4 namespace buildit_application {
5 //...
6 builder::dyn_var<void(void)> printf = builder::as_global("printf");
7 //...
8 struct BFInst {
9     char cmd = builder::with_name("cmd");
10    BFInst* next = builder::with_name("next");
11    BFInst* jmp = builder::with_name("jmp");
12 };
13 }
14 namespace builder {
15 template <>
16 struct external_type_namer<buildit_application::BFInst> {
17     static constexpr const char* type_name = "struct BFInst";
18 };
19 }
20 namespace buildit_application {
21
22 struct Mem {
23     char val = builder::with_name("val");
24     Mem* next = builder::with_name("next");
25     Mem* prev = builder::with_name("prev");
26 };
27 }
28 namespace builder {
29 template <>
30 struct external_type_namer<buildit_application::Mem> {
31     static constexpr const char* type_name = "struct Mem";
32 };
33 }
34 namespace buildit_application {
35
36
37 static BFInst prog_pool[65536] = builder::as_global("prog_pool");
38 static int prog_count = builder::as_global("prog_count");
39
40 static Mem cell_pool[65536] = builder::as_global("cell_pool");
41 static int cell_count = builder::as_global("cell_count");
42
43 static BFInst* alloc_inst() {
44     return (prog_count < 65536)
45         ? &prog_pool[prog_count++]
46         : __null;
47 }
48
49 static Mem* alloc_cell() {
50     return (cell_count < 65536)

```



```

51         ? &cell_pool[cell_count++]
52         : __null;
53     }
54
55     int main(int argc, char* argv[]) {
56         if (argc < 2) {
57             printf("Usage: %s \"<bf-program>\"\n", argv[0]);
58             return 1;
59         }
60
61         const char *program = argv[1];
62
63         BFIInst *p = __null, *n = __null, *j = __null, *pgm = __null;
64         for (const char *src = program; *src; ++src) {
65             char ch = *src;
66             int valid = (ch=='<' || ch=='>' || ch=='+' || ch=='-' ||
67                         ch==',' || ch=='.' || ch=='[' || (ch==']'&&j));
68             if (!valid) continue;
69
70             n = alloc_inst();
71             if (!n) {
72                 printf("Error: program too large\n");
73                 return 1;
74             }
75
76             if (p) p->next = n; else pgm = n;
77             n->cmd = ch;
78             n->next = __null;
79             n->jmp = __null;
80             p = n;
81
82             if (ch == '[') {
83                 n->jmp = j;
84                 j = n;
85             }
86             else if (ch == ']') {
87                 n->jmp = j;
88                 j = j->jmp;
89                 n->jmp->jmp = n;
90             }
91         }
92
93         while (j) {
94             p = j;
95             j = j->jmp;
96             p->jmp = __null;
97             p->cmd = ',';
98         }
99
100         Mem *m = alloc_cell();
101         if (!m) {
102             printf("Error: tape out of memory\n");
103             return 1;
104         }

```

```

105
106     for (n = pgm; n; n = n->next) {
107         if (n->cmd == '+') {
108             ++m->val;
109         } else if (n->cmd == '-') {
110             --m->val;
111         } else if (n->cmd == '.') {
112             printf("%c", m->val);
113         } else if (n->cmd == ',') {
114             continue;
115         } else if (n->cmd == '[') {
116             if (m->val == 0) n = n->jmp;
117         } else if (n->cmd == ']') {
118             if (m->val != 0) n = n->jmp;
119         } else if (n->cmd == '<') {
120             if (!(m = m->prev)) {
121                 printf("Error: at start of tape\n");
122                 return 1;
123             }
124         } else if (n->cmd == '>') {
125             if (!m->next) {
126                 Mem *c = alloc_cell();
127                 if (!c) {
128                     printf("Error: tape out of memory\n");
129                     return 1;
130                 }
131                 c->prev = m;
132                 c->next = __null;
133                 m->next = c;
134             }
135             m = m->next;
136         }
137     }
138
139     return 0;
140 }
141 static register_function reg0(main, "main");
142
143 }

```

```

1 // Driver for BF interpreter
2 #include "builder/dyn_var.h"
3 #include "blocks/c_code_generator.h"
4
5 using builder::dyn_var;
6
7 std::vector<std::function<block::block::Ptr(void)>> *
8     registered_functions;
9 int main(int argc, char* argv[]) {
10     std::cout << "#include <stdio.h>\n\n";
11     std::cout << "struct BFIInst {\nchar      cmd;\nBFIInst* next;\nBFIInst*
        jmp;\n};\n";
12     std::cout << "struct Mem {\nchar      val;\nMem*      next;\nMem*      prev;\n";

```

```

    }; \n \n";
12
13     std::cout << "static BFIInst prog_pool[65536]; \n";
14     std::cout << "static int      prog_count = 0; \n";
15     std::cout << "static Mem      cell_pool[65536]; \n";
16     std::cout << "static int      cell_count = 0; \n";
17
18     for (auto f: *registered_functions) {
19         auto ast = f();
20         block::c_code_generator::generate_code(ast, std::cout, 0);
21     }
22     return 0;
23 }

```

```

1 // Generated code for BF Interpreter
2 #include <stdio.h>
3
4 struct BFIInst {
5     char      cmd;
6     BFIInst* next;
7     BFIInst* jmp;
8 };
9 struct Mem {
10     char      val;
11     Mem*      next;
12     Mem*      prev;
13 };
14
15 static BFIInst prog_pool[65536];
16 static int      prog_count = 0;
17 static Mem      cell_pool[65536];
18 static int      cell_count = 0;
19 int main (int arg0, char* arg1[]) {
20     struct BFIInst* var11;
21     struct Mem* var13;
22     struct Mem* var16;
23     if (arg0 < 2) {
24         printf("Usage: %s \"<bf-program>\" \n", arg1[0]);
25         return 1;
26     }
27     struct BFIInst* var4 = 0;
28     struct BFIInst* var5 = 0;
29     struct BFIInst* var6 = 0;
30     struct BFIInst* var7 = 0;
31     char const* var8 = arg1[1];
32     while (var8[0]) {
33         char var9 = var8[0];
34         if (!( (((((((var9 == 60) || (var9 == 62)) || (var9 == 43)) || (
var9 == 45)) || (var9 == 44)) || (var9 == 46)) || (var9 == 91)) ||
((var9 == 93) && var6)))) {
35             if (prog_count < 65536) {
36                 var11 = (&(prog_pool[(prog_count = prog_count + 1) - 1]));
37             } else {

```

```

38     var11 = 011;
39 }
40 var5 = var11;
41 if (!(var5)) {
42     printf("Error: program too large\n");
43     return 1;
44 }
45 if (var4) {
46     var4->next = var5;
47 } else {
48     var7 = var5;
49 }
50 var5->cmd = var9;
51 var5->next = 011;
52 var5->jmp = 011;
53 var4 = var5;
54 if (var9 == 91) {
55     var5->jmp = var6;
56     var6 = var5;
57 } else {
58     if (var9 == 93) {
59         var5->jmp = var6;
60         var6 = var6->jmp;
61         (var5->jmp)->jmp = var5;
62     }
63 }
64 }
65 var8 = var8 + 1;
66 }
67 while (var6) {
68     var4 = var6;
69     var6 = var6->jmp;
70     var4->jmp = 011;
71     var4->cmd = 32;
72 }
73 if (cell_count < 65536) {
74     var13 = (&(cell_pool[(cell_count = cell_count + 1) - 1]));
75 } else {
76     var13 = 011;
77 }
78 if (!(var13)) {
79     printf("Error: tape out of memory\n");
80     return 1;
81 }
82 var5 = var7;
83 while (var5) {
84     if (var5->cmd == 43) {
85         var13->val = var13->val + 1;
86     } else {
87         if (var5->cmd == 45) {
88             var13->val = var13->val - 1;
89         } else {
90             if (var5->cmd == 46) {
91                 printf("%c", var13->val);

```

```

92     } else {
93         if (!(var5->cmd == 44)) {
94             if (var5->cmd == 91) {
95                 if (var13->val == 0) {
96                     var5 = var5->jmp;
97                 }
98             } else {
99                 if (var5->cmd == 93) {
100                     if (var13->val != 0) {
101                         var5 = var5->jmp;
102                     }
103                 } else {
104                     if (var5->cmd == 60) {
105                         if (!(var13 = var13->prev)) {
106                             printf("Error: at start of tape\n");
107                             return 1;
108                         }
109                     } else {
110                         if (var5->cmd == 62) {
111                             if (!(var13->next)) {
112                                 if (cell_count < 65536) {
113                                     var16 = (&(cell_pool[(cell_count = cell_count +
114                                     1) - 1]));
115                                 } else {
116                                     var16 = 011;
117                                 }
118                                 if (!(var16)) {
119                                     printf("Error: tape out of memory\n");
120                                     return 1;
121                                 }
122                                 var16->prev = var13;
123                                 var16->next = 011;
124                                 var13->next = var16;
125                             }
126                             var13 = var13->next;
127                         }
128                     }
129                 }
130             }
131         }
132     }
133     var5 = var5->next;
134 }
135 return 0;
136 }
137 }

```

Notice that in the generated code, there is no call to the helper functions `alloc_cell()` or `alloc_inst()`. In general, when BuildIt generates code for any function, it will further explore any function calls it makes and effectively inline that code. The only exception to

Table 4.1: Lines of Code at Different Stages in the Conversion Process

Program	Original Source	Preprocessed Source	Rewritten Source	Generated Source
Example 1	19	609	735	21
Example 2	128	716	843	137

this is for recursive function calls. Calls to recursive functions still appear in the generated code due to our special treatment of them.

4.2 Evaluation

To evaluate our BuildIt conversion tool, we consider its impact on compilation performance, debugging, and code formatting. As converting and compiling a program with the BuildIt conversion tool is a multi-step process, it does increase the time it takes to compile a program and produce an executable. In fact, our changes to the Clang compiler result in additional code being executed for each declaration in the program. A comparison of the time taken to compile a program with a standard approach versus with the conversion tool is shown in figure 4.6. We find the time taken by the overall conversion process is significantly more than the time taken to normally compile a program, with much of the overhead coming from the modified Clang compiler. This partially stems from the fact that all types in the user program are parsed and modified, including code from large header files. Table 4.1 shows the length of the rewritten code that the modified Clang compiler parses. Future work could attempt to mitigate the conversion tool’s overhead, particularly by optimizing the compiler’s type-modification process. However, the runtime of the generated program—which is considerably more important in most applications—remains unchanged.

The BuildIt conversion tool also affects the formatting and debugging of the intermediate code. Throughout the conversion process, the tool inserts types and text that do not exist in the original code, resulting in incorrect source location information for much of the program. If Clang crashes while compiling the rewritten program, it will use the invalid source location

Compilation Time Comparison

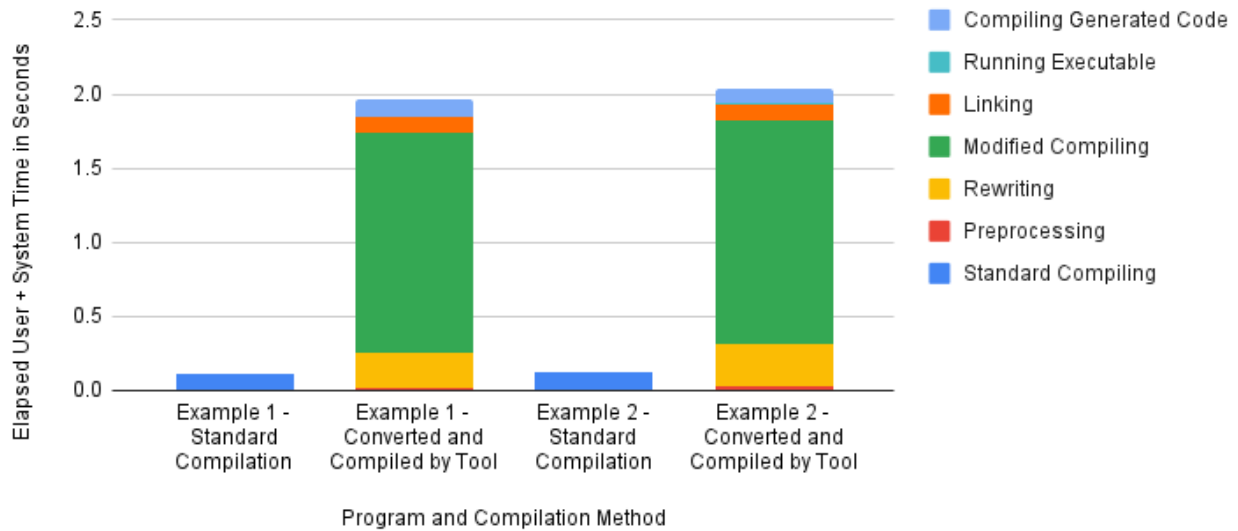


Figure 4.6: The time taken to compile the programs from example 1 and example 2 in section 4.1, from a standard approach and from converting and compiling the program with the BuildIt conversion tool. Note, no linker errors are thrown when converting example 1 and 2 so no stubs need to be generated.

information to emit error messages. Furthermore, the Rewriter Tool often inserts text that is not properly indented into the code. This makes debugging the intermediate programs more challenging. However, this is not a problem in actuality. C and C++ programs should be compiling properly before they are provided to the conversion tool, so there should be no compilation issues during the conversion process. Additionally, the final generated code is in standard C/C++ and is compiled with a standard compiler. Therefore, any runtime or compile time errors from the generated program produce standard error messages.

Based on our testing of the BuildIt conversion tool, it should very much be considered in development. While the tool does reliably convert sample programs (consisting of a few hundred lines of code) to BuildIt, it sometimes runs into issues when converting larger programs with many header files to BuildIt. Header files in particular present an issue, as they may consist of thousands of lines of code and utilize less-common C and C++ programming constructs. In addition, the conversion tool does not yet support many common C and C++

features such as union types, classes, and templates. This means user programs can not directly use these features or include headers that use these features. Thus, there is much ongoing work on this project.

Lastly, it is worth noting that the conversion tool produces the equivalent BuildIt program in an object file format. This means that a user can not directly edit the converted BuildIt program and begin staging variables to specialize the code. The generated code from BuildIt's execution will thus be unoptimized code equivalent to the original source. We will therefore need to extend the tool to allow the user to specify variables to stage when converting their code to BuildIt. This is discussed in the future work section.

Chapter 5

Conclusion

5.1 Future Work

Our conversion tool currently supports converting C and C++ programs with a subset of the language features. Future work can extend the tool’s support to converting broader C and C++ programs, including programs with classes, templates, and union types. Support for these features will likely involve source rewrites, since converting these features to BuildIt requires more substantial changes than simply wrapping types with `dyn_var`. Union types can most likely be converted in a similar manner to structs. However, the rewriting process for classes and templates is likely to be much more complicated.

In keeping with the original goal of enabling staging and specialization of programs, future work can be done to allow users to stage certain variables when automatically converting their code to BuildIt. In this case, the conversion tool will wrap user-specified variables with the BuildIt `static_var` type and the remaining types with `dyn_var` when converting code. The resulting program executes in two stages, with the static stage executing first to produce code for the dynamic stage. Users can thus make use of BuildIt’s multi-stage framework to optimize and specialize their code. Supporting this in the conversion tool will likely require changes to the modified Clang compiler to check and wrap certain variables

with the `static_var` type, and also require changes in the driver to provide values for the staged variables when generating code.

If the previously described work is completed, it should be possible to automatically stage and specialize large C and C++ programs, including repositories. There are many widely-used C and C++ repositories like nginx [18] and memcached [19] that are developed with performance in mind. Future work could attempt to stage and optimize these repositories with BuildIt, which would be incredibly impactful if successful.

5.2 Summary

In this thesis, we discussed the development of a tool that automatically converts C and C++ source code to the BuildIt multi-stage programming framework. This was achieved by wrapping every type in a program with BuildIt’s `dyn_var` type, producing equivalent BuildIt code. Our approach combined a Clang Rewriter tool with a modified Clang compiler, demonstrating how to use and extend Clang to perform non-trivial source transformations. Along the way, we highlighted limitations to Clang’s tooling and shared practical insights into Clang’s internals. We demonstrated our conversion tool’s viability by successfully converting and compiling example programs, showing the generated code is equivalent to the original code. We also showed how to convert various key C languages features—including structs, global variables, recursive functions, and more—to BuildIt, enabling conversion of a broad host of C and C++ programs. As our tool automatically converts a program to its (unstaged) Buildit equivalent, it performs the tedious work necessary before a user can begin staging variables and specializing their code with BuildIt. Ultimately, we hope it brings us closer to the broader goal of being able to automatically stage and optimize C and C++ repositories with the BuildIt framework.

References

- [1] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. “Graphit: A high-performance graph DSL”. *Proceedings of the ACM on Programming Languages* (Oct. 2018). URL: <https://doi.org/10.1145/3276491>.
- [2] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 519–530. URL: <https://doi.org/10.1145/2491956.2462176>.
- [3] M. Abadi et al. *TensorFlow: Large-scale machine learning on heterogeneous systems*. Software available from <http://tensorflow.org/>. 2015.
- [4] W. Taha. “A gentle introduction to multi-stage programming”. *Domain Specific Program Generation* Springer (2004), pp. 30–50.
- [5] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. “Terra: a multi-stage language for high-performance computing”. *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation* (2013), pp. 105–116.
- [6] A. Brahmakshatriya and S. Amarasinghe. “BuildIt: A type based multistage programming framework for code generation in C++”. In: *Proceedings of the 2021 International Symposium on Code Generation and Optimization*. 2021.
- [7] A. Brahmakshatriya and S. Amarasinghe. “GraphIt to CUDA compiler in 2021 LOC: A case for high-performance DSL implementation via staging with BuildDSL”. In: *Proceedings of the 2022 International Symposium on Code Generation and Optimization*. 2022.
- [8] The LLVM Project. *Clang: A C language family frontend for LLVM*. URL: <https://clang.llvm.org/>.
- [9] The LLVM Project. *Clang Tools*. URL: <https://clang.llvm.org/docs/ClangTools.html>.
- [10] The LLVM Project. *Clang Plugins*. URL: <https://clang.llvm.org/docs/ClangPlugins.html>.
- [11] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan. “Large-Scale Automated Refactoring Using ClangMR”. In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 548–551. DOI: [10.1109/ICSM.2013.93](https://doi.org/10.1109/ICSM.2013.93).
- [12] The LLVM Project. *Clang Tidy*. URL: <https://clang.llvm.org/extra/clang-tidy>.

- [13] G. Antal, D. Havas, I. Siket, Á. Beszédes, R. Ferenc, and J. Mihalicza. “Transforming C++11 Code to C++03 to Support Legacy Compilation Environments”. In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2016, pp. 177–186. DOI: [10.1109/SCAM.2016.11](https://doi.org/10.1109/SCAM.2016.11).
- [14] G. Balogh, G. Mudalige, I. Reguly, S. Antao, and C. Bertolli. “OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling”. In: *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 2018, pp. 59–70. DOI: [10.1109/LLVM-HPC.2018.8639205](https://doi.org/10.1109/LLVM-HPC.2018.8639205).
- [15] T. B. Schardl and I.-T. A. Lee. “OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code”. In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. PPOPP ’23. Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 189–203. URL: <https://doi.org/10.1145/3572848.3577509>.
- [16] Esolangs. *Brainfuck Language Wiki*. URL: <https://esolangs.org/wiki/Brainfuck>.
- [17] Rosetta Code. *Brainfuck Interpreter in C*. URL: https://rosettacode.org/wiki/Execute_Brain****/C.
- [18] Nginx. *Nginx – A High Performance Web Server and Reverse Proxy*. URL: <https://github.com/nginx/nginx>.
- [19] Memcached. *Memcached – A High Performance Distributed Memory Object Caching System*. URL: <https://github.com/memcached/memcached>.