CJQ: A COMPILER FOR THE JQ PROGRAMMING LANGUAGE

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science - Computer Science

by

John Rubio

Aug 2024

ABSTRACT

`jq` is a functional programming langauge, known for its ability to filter, transform, and manipulate JSON data with concise syntax. The standard implementation of `jq` includes a source-to-bytecode compiler and a stack-based virtual machine (VM).

This thesis introduces `cjq` v0.1, a compiler that translates `jq` bytecode into LLVM intermediate representation (IR). By leveraging LLVM's compiler infrastructure, `cjq` reliably enhances runtime performance for small to medium-sized JSON inputs when compared to the standard `jq` implementation. However, the current *tracing module* in `cjq` becomes a bottleneck for certain types of large-scale JSON inputs.

`cjq` combines the functional elegance of `jq` with the performance benefits of compiled languages, aiming to deliver a robust tool for modern data processing tasks. Future work involves optimizing the tracing module to improve scalability and exploring further enhancements to extend `cjq`'s capabilities.

Keywords: jq, JSON, LLVM, bytecode, interpreter, compiler

BIOGRAPHICAL SKETCH

John Rubio received his Bachelors in Mathematical Statistics with a minor in Computer Science from Wayne State University University in 2020, graduating summa cum laude, and worked in industry for 2 years before his fascination with compilers got the better of him and he returned to school. John plans to return to industry after the MS for at least a year, and plans to remain actively involved in the community of researchers working on programming languages, compilers, and hardware accelerators.

This document is dedicated to all Cornell graduate students.

ACKNOWLEDGEMENTS

In any large-scale endeavor by one person, it always takes a village to support that person. Here are the people in my village, who I'm deeply indebted to:

- My wife, Bekah, for being deeply supportive of my aspirations to go to graduate school and willing to move across the country with me as I pursued my goal of becoming a member of the Computer Science research community
- My parents, brother, and sister, for being by my side through a long, convoluted path that only makes sense looking backward
- My advisors Adrian Sampson and Zhiru Zhang, who've provided me with the patience, support, and help I needed to take on the daunting task of building a new language implementation.
- My MS cohort: I'm going to miss you all

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In this chapter, I'll motivate and outline the contributions described throughout the rest of this thesis.

1.1 Motivation

The `jq` programming language is renowned for its efficiency in processing and transforming JSON[10] data using concise and expressive filters. Traditionally, `jq` has been implemented with a stack-based interpreter, which provides flexibility and ease of implementation but may not fully exploit modern optimizations for performance.

The `cjq` project aims to provide a new `jq` implementation, transitioning away from the current stack-based interpreter execution model to a compiler-driven approach. The primary goal is to develop a new compiler for `jq` that leverages LLVM optimizations to eliminate overhead associated with stack-based semantics and to generate machine code more aligned with a register-based execution model.

Specifically, the objectives of this endeavor are:

1. Designing and implementing a compiler for `jq` that translates `jq` filters into optimized machine code.
2. Exploring a "tracing" technique to compile `jq` bytecode into efficient LLVM intermediate representation (IR).

3. Evaluating and comparing the performance of the `cjq` compiler against the standard `jq` interpreter across a variety of benchmarks.
4. Developing a user-friendly tool that facilitates efficient JSON document transformations without the need to repeatedly specify `jq` filters for each transformation.

1.2 Thesis outline

In Chapter 2, I provide a brief overview of the `jq` programming language and the standard `jq` implementation, which will serve as the necessary background to discuss the implementation details of the `cjq` compiler.

In Chapter 3, I present `cjq`, a novel bytecode-to-LLVM compiler implementation of the `jq` programming language[1]. The `cjq` implementation consists of three main phases, each of which will be discussed in this chapter: tracing and lowering, bootstrapping, and optimization.

In Chapter 4, I conduct a performance evaluation of `cjq` and compare the results to the standard `jq` implementation. I utilize a combination of benchmarks, including those provided by the maintainers of the standard `jq` implementation[11] as well as handcrafted benchmarks to increase coverage of the features in the `jq` programming language.

Finally, in Chapter 5, I consider the current results and discuss possible future directions for `cjq`.

[1]While the architecture and implementation are novel, many internals are borrowed from the standard `jq` implementation.

CHAPTER 2

BACKGROUND

Mini-languages like awk and sed have a long history, dating back to Bell Labs in the 1970s [1, 5]. Due to factors such as minimal overhead, interactive programming support, platform independence, quicker development times, and the relative ease of implementation compared to compilers, the decision was made to implement the standard versions of these programming languages as interpreters [1]. `jq`, belonging to this lineage, emerged to address the special case of text processing for JSON documents [12]. In this section, I'll briefly introduce the `jq` programming language and provide an overview of the standard `jq` implementation.

2.1 The `jq` Programming Language

`jq` is a functional programming language characterized by dynamic typing and lazy evaluation. It supports higher-order functions, though they are considered second-class citizens within the language[12]. The `jq` programming language was specifically designed for processing JSON documents, offering a concise and expressive syntax tailored for efficient querying and transforming of JSON data. In this section, I provide an overview of the essentials of the `jq` language that will serve as a foundation for understanding the internals of the standard implementation of `jq`[1].

[1]For a more in-depth description of the `jq` programming language, please refer to the user manual.

`jq` Basics

Fundamentally, the `jq` tool is a simple one - it takes zero or more JSON values as input and produces zero or more JSON output values[2]. `jq`'s ability to transform JSON data in highly flexible ways is the reason why `jq` programs are referred to as *filters*. When discussing `jq` filters, assume zero or more arbitrary JSON values as input and zero or more JSON values as output.

The simplest `jq` filter is the **identity filter** `.` which, when applied, returns the input JSON document unaltered.

```
echo '"foo"' | jq '.'    # returns "foo"
```

The simplest *useful* `jq` filter is the **Object Identifier-Index filter**. This filter allows users to index into JSON objects. When provided with a JSON object as input, `.foo` retrieves the value associated with the key "foo" if it exists; otherwise, it returns `null` [4].

```
echo '{"foo" : 42}' | jq .foo    # returns 42
```

Similarly, `jq` supports operations like array indexing and slicing. `jq`'s array-manipulation syntax resembles syntax found in general-purpose scripting languages[3].

```
Command   jq '.[0]'
Input     [{"name":"JSON", "good":true},
          {"name":"XML", "good":false}]
 Output   {"name":"JSON", "good":true}
```

[2] `jq` can also produce errors.
[3] The below examples are borrowed from the `jq` manual

```
Command      jq '.[2:4]'
Input        ["a","b","c","d","e"]
Output       ["c", "d"]
```

```
Command      jq '.[-2:]'
Input        ["a","b","c","d","e"]
Output       ["d", "e"]
```

Part of what makes `jq` useful as a JSON processing tool is that every `jq` expression can be a generator that can produce zero, one, or many outputs [12]. One such example is the **Array/Object Value Iterator `.[]`**. Syntactically, this feature is the same as the index operator. The only difference is that no value is supplied to the index operator. The semantic behavior of the Array/Object Value Iterator is that it takes an array or an object as input and returns all of the values that were originally contained in the structure as output.

```
Command  jq '.[]'
Input   [{"name":"JSON", "good":true},
         {"name":"XML", "good":false}]
Output   {"name":"JSON", "good":true}
         {"name":"XML", "good":false}
```

```
Command   jq '.[]'
Input     []
Output    none
```

```
Command   jq '.[]'
Input     {"a": 1, "b": 1}
Output    1
          1

Command   jq '.foo[]'
Input     {"foo":[1,2,3]}
Output    1
          2
          3
```

JSON arrays and objects can also be constructed using `jq`'s array and object constructors. These features are about what one would expect - to construct a JSON array, zero or more expressions must be wrapped in `[]`.

```
Command jq '[.[].a]'
Input [{"a" : 1}, {"a" : 2}]
Output [1, 2]
```

Likewise, to construct objects, zero or more key-value pairs must be wrapped in {}.

```
Command jq '{"first_name" : .[0], "last_name": .[1]}
Input ["Alan", "Turing"]
Output {"first_name" : "Alan", "last_name" : "Turing"}
```

If two filters are separated by a comma, the same input is processed through both filters sequentially. The output streams of the two filters are then concate-

nated in order: first, all outputs from the left filter, followed by all outputs from the right filter. For example, the filter ".foo, .bar" produces separate outputs for both the foo and bar fields [4].

```
Command  jq '.foo, .bar'
Input    {"foo": 42, "bar": "something else", "baz": true}
Output    42
          "something else"
```

```
Command   jq '.[4,2]'
Input     ["a","b","c","d","e"]
Output    "e"
          "c"
```

As a functional programming language, jq must support function composition - this is implemented via the | operator. The output(s) of an expression can be passed to another using the | operator. expressionA | expressionB applies expressionA to some JSON input, then expressionB to all the outputs of expressionA.

```
echo 1 | jq '.+1, 10 | .+2'
```

In the above code snippet, the integer, 1 is supplied to the jq subexpression .+1, 1. Thus, the subexpression .+1, 10 adds one to the current input and pipes the result, 2, as well as a newly introduced value, 10 to the next subexpression using the jq pipe operator. The next subexpression, .+2, is applied to each of the inputs, 2 and 10, and thus the outputs of this jq program are 4 and 12.

As seen in previous examples, `jq` supports all of the typical arithmetic operations such as addition, subtraction, multiplication, division, and modulo. For example, the filter "`.[] % 2`" yields `[0, 1, 0, 1]` when the JSON array `[2, 3, 4, 5]` is supplied as input.

`jq` also has many built-in functions which substantially increase the power of `jq` as a command-line tool. To add all elements of an array, the `add` filter can be applied to an input array. For example, the filter "`add`" yields `3` when the JSON array `[1,2]` is supplied as input , the filter "`sort`" yields `[1,2,3]` when the JSON array `[3,2,1]` is supplied as input, and the filter "`keys`" yields `["key1", "key2"]` when the JSON object {`"key1" : 1, "key2" : 2` } is supplied as input.

Control flow is handled via the typical `if-then-else` construct. Conditionals in `jq` can be illustrated by the below example which is a `jq` filter that calculates the factorial **`n!`** given a positive integer **`n`**. Note that indentation is only used below for the sake of clarity.

```
jq 'def update:
        if .[0] > 1 then
            [.[0] - 1, .[0] * .[1]]
        else
            empty
        end;
    def factorial(n):
        [n, 1] | last(recurse(update))[1];
    factorial(.)'
```

Using a combination of user-defined and built-in functions, the input is

eventually passed to the `update` function which uses the conditional logic to determine whether the base case, `empty` which stops recursion, or the recursive case should be executed.

Iteration is not required in the same way it often is in general-purpose languages. For example, given a JSON array of integers, the average can be calculated by simply using the filter "`add / length`". However, `jq` provides support for operating over streams of values with many built-in functions for performing higher-order operations, such as `fold` and `map`. Folding operations can be performed using `reduce` and `foreach` while mapping operations are done with `map`. `jq` also provides the more familiar `while` filter for those who wish to express indefinite iteration using an imperative paradigm. It is worth noting that the `while` filter does not contradict the functional nature of `jq`. Under the hood, `while` is simply a recursive function as shown below:

```
def while(cond; update):
 def _while:
     if cond then ., (update | _while) else empty end;
 _while;
```

This example highlights a general design principle of the `jq` programming language: tasks accomplished via iteration in other languages are either solved by pipelining filters together (i.e. function composition) or recursion in `jq` [4].

`jq` supports user-defined functions, allowing filters to reuse logic. These functions can be applied to a singular input JSON value and can take multiple arguments. Here is an example of defining and using a simple function in `jq`:

```
echo '{"a": 1, "b": 2}'
```

```
| jq 'def add2(x; y): x + y; add2(.a; .b)'
```

In this snippet, the function `add2` is defined to take two arguments, `x` and `y`. The function simply returns the sum of these two arguments. When applied to the input JSON object {`"a": 1, "b": 2`}, the function is called with `.a` and `.b` as arguments, resulting in the output `3`.

Fundamental to `jq`'s design is the notion that each filter has an input and an output. Consequently, variables are often unnecessary in `jq`. To illustrate this, most programming languages require at least a couple of variables to calculate the average value of a list of numbers. As discussed earlier in this section, `jq` requires zero variables to accomplish this computation since the stream of values is provided as the input to the filter "`add / length`". Still, situations do arise when variables are necessary so `jq` does provide support for the use of variables[4]. In `jq`, the syntax for assigning a value to a variable is "`<value> as $v | ...`". The semantics of this `jq` filter are that the input "`.`" will be passed to the next expression via the | operator along with `$x` which is assigned the value `<value>`. `jq` variables are lexically-scoped bindings - that is they are scoped over the entirety of the expression in which they are defined. For example, in this filter `.realnames as $names | (.posts[] | {title, author: $names[.author]})`, the variable `$names` will be available when it is used but not in the filter: `(.realnames as $names | .posts[]) | {title, author: $names[.author]}` since `$names` is referenced outside of the expression in which it was defined [4].

In this section, I provided a brief overview of the `jq` programming language. I discussed the core features of the language one would need to understand to begin writing `jq` filters. This description of the `jq` language was not intended

to provide a comprehensive description jq. Rather, this overview aimed to acquaint those unfamiliar with jq to the extent that I can now discuss the internals and implementation of the jq programming language. For a thorough description of the jq programming language, please refer to the jq manual[4].

2.2 The Standard jq Implementation

Like most production-quality interpreters, the standard jq implementation is a highly optimized bytecode interpreter[12]. Architecturally, the jq bytecode interpreter consists of a source-to-bytecode compiler and a VM that executes a dynamic sequence of bytecode instructions. In this section, I describe the major components of the standard jq architecture and discuss how generators and backtracking are implemented.

2.2.1 The Standard jq Compiler

When a jq filter is passed to the standard jq interpreter, the interpreter first compiles the jq source code to an intermediate representation referred to as the block representation [12]. The block representation of a jq program is similar in structure to an Abstract Syntax Tree (AST). Internally, each jq block consists of a pair of pointers pointing to the first and last instructions within each basic block in the block representation structure. Each instruction roughly corresponds to a bytecode instruction in the next phase. Once the block representation is constructed, the linker binds all unresolved references to built-in functions and programmer-defined symbols using jq's module system [12]. Lastly,

the compiler emits bytecode from the linked block representation.

The primary benefit of `jq`'s VM operating on bytecode rather than `jq`'s block representation is the fact that the bytecode opcodes are stored in a dynamic array of 16-bit integers, leading to higher cache locality[3]. In contrast, the instructions in `jq`'s block representation are stored as a linked list, which has low cache locality and would therefore hurt performance.

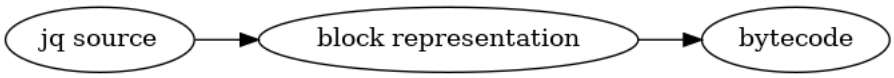


Figure 2.1: The compilation pipeline of the `jq` compiler, illustrating the transformation from source code to bytecode with the block representation acting as an intermediate representation.

2.2.2 The `jq` Type System

A convenient consequence of `jq`'s intended use case is that its type system is the same as JSON's. Therefore, the set of data types that `jq` supports includes numbers, strings, booleans, arrays, objects (i.e., key-value pairs where the keys are always strings), and null[4]. For error-handling purposes, the standard `jq` implementation also supports an invalid type. Internally, the standard `jq` implementation converts JSON input from raw text to what is referred to as a `jv` value. `jv` values are simply C structures[12]. This is important because JSON data is flexible, and `jq` must support many complex operations for all valid JSON inputs. To accommodate the flexible nature of JSON data, `jv` values are designed to support all of JSON's data types as well as arbitrarily complex JSON inputs[4].

[4]See `jq` source code for details.

2.2.3 The `jq` Bytecode Virtual Machine

Once the `jq` source code has been compiled to bytecode, the interpreter is ready to execute the `jq` program. The `jq` virtual machine is stack-based [12]. This means that the manner in which the input JSON value(s) are manipulated use stack-based semantics. Furthermore, the `jq` stack is made up of *stack blocks*. Each stack block can either be a JSON value (a `jv` value), a call frame, or a "fork point" [12]. `jv` values can be objects, arrays, numbers, booleans, null, or an invalid type value. Like in other programming languages, call frames or activation frames are execution contexts for function calls, including a function's local variables, the return address, and other state information. Fork points mark the locations that the execution flow must return to in order to perform an operation on the next value in a stream of values. This is how `jq` implements generators and lazy evaluation - through backtracking.

When discussing the`jq` VM, it is essential to remember that `jq` is a mini-language designed to process JSON documents. To that end, there is an implied control flow that the `jq` VM implements, as depicted in the listing below.

Listing 2.1: Implied Control Flow

```
for each JSON input provided:
    for each generator in the jq filter:
        while generator is still producing outputs:
            pretty_print output;
            do backtrack;
```

The `jq` Interpreter Stack

Each variably-sized stack block in the `jq` stack points to the previously pushed block through a *next pointer*. A stack block can be a *next block* to more than one other stack block. To avoid the cost of adjusting each pointer whenever the stack needs to be grown, the next pointers are implemented as negative integers relative to the end of the `jq` stack[12]. If a stack block has no next block, its next pointer is zero.

In contrast with ordinary stack data structures, the `jq` stack is a directed forest of stack blocks [12]. This means that despite the fact that there is a single allocated region of memory for the `jq` stack, from the program's perspective, there can be multiple stacks in memory simultaneously. This approach allows for easier memory management despite the potential complexity of `jq` programs. Moreover, a directed forest-style stack allows for multiple entry points. This is useful for expressions that do not utilize the output of an expression on the left-hand side of a | operator.

To implement generators, `jq` must support backtracking. This means that the VM must be able to reverse execution to a specific point in the program (the fork point) before it resumes forward execution. This corresponds to popping blocks off the stack until the most recent fork point is reached. For example, if `1` is supplied as input to the `jq` filter "`.+1, .+2`", the `jq` source code to bytecode compiler will emit the following sequence of bytecode instructions:

Listing 2.2: Bytecode sequence compiled from the jq filter ".+1, .+2"

```
0000 TOP
0001 FORK 0011
```

```
0003 PUSHK_UNDER 1
0005 DUP
0006 CALL_BUILTIN _plus
0009 JUMP 0017
0011 PUSHK_UNDER 2
0013 DUP
0014 CALL_BUILTIN _plus
0017 RET
```

See figure 2.2 for a depiction of how the `jq` stack is modified as the VM executes the above `jq` program.

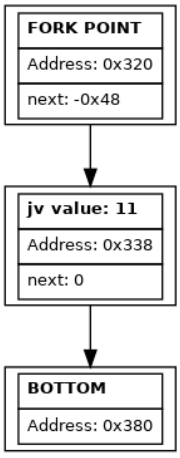
The second bytecode instruction marks a fork point in the program. The `FORK` instruction pushes a fork point to the stack. This means that when the VM eventually backtracks to this fork point, program execution will resume in a forward fashion starting from the instruction at bytecode index `0011`, which is the instruction `PUSHK_UNDER 2`. Thus, after backtracking the VM executes the "`.+2`" expression on the right-hand side of the "`,`" operator. Specifically, the VM executes the below sequence of bytecode instructions after backtracking which adds `2` to the input before returning and pretty printing the result.

```
0011 PUSHK_UNDER 2
0005 DUP
0006 CALL_BUILTIN _plus
0009 JUMP 0017
0017 RET
```

Prior to backtracking, the VM adds `1` to the input before jumping to the

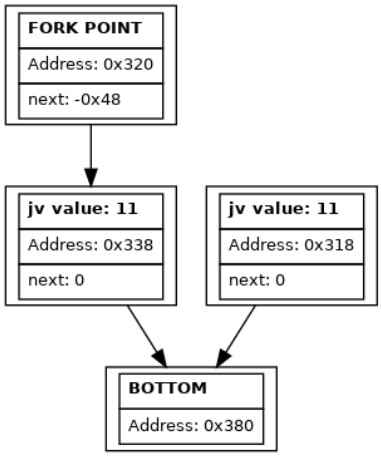
`RET` instruction which returns and pretty prints the result. Afterward, the program execution flow backtracks from the `RET` instruction to the `FORK` instruction. The `FORK` instruction marks the point where program execution flows forward again. Again, this is the point in the program execution where control jumps to the `0011 PUSHK_UNDER 2` instruction, indicating that the program is now ready to add `2` to the input.

One of `jq`'s design principles is that every expression acts as a generator. When a generator processes a value in a stream of values, the program flow backtracks to the nearest preceding fork point before resuming normal execution. When a generator runs out of values to process, program flow will backtrack to the nearest preceding, active generator and program execution will resume from there. This continues until each generator has processed all of its input values. This behavior illustrates why `jq` is said to utilize ”pervasive backtracking”[12].



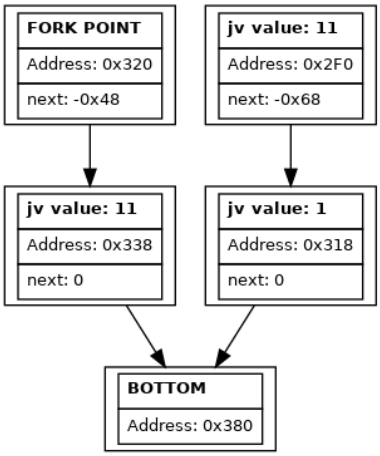
After TOP

(i)



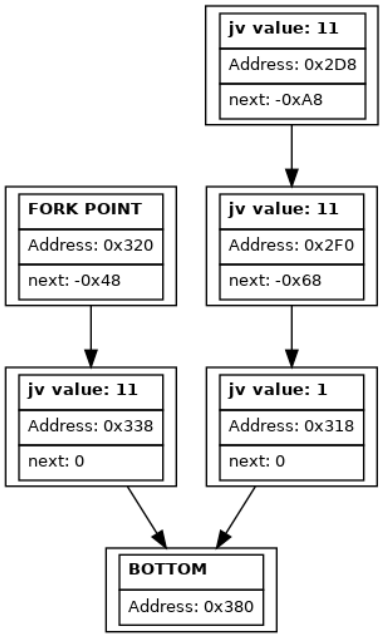
After FORK

(ii)



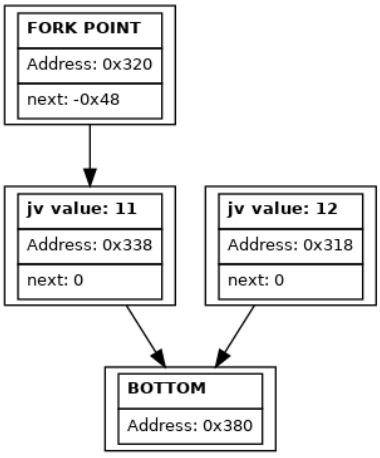
After PUSHK_UNDER 1

(iii)



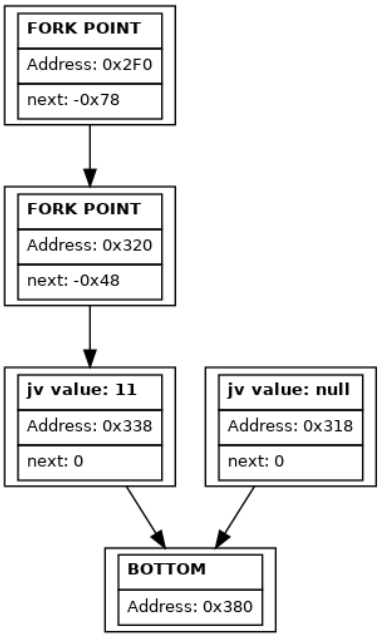
After DUP

(iv)



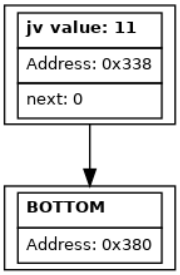
After CALL_BUILTIN _plus

(v)



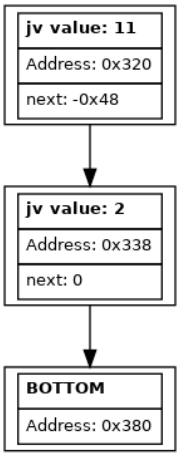
After RET

(vi)



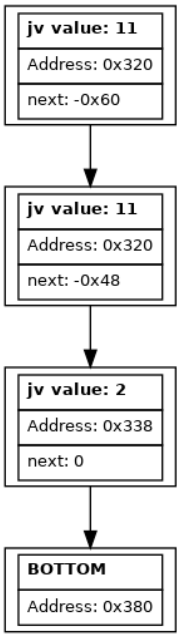
After backtracking

(vii)



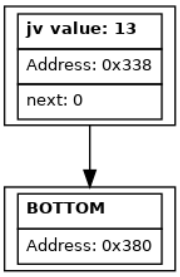
After PUSHK_UNDER 2

(viii)



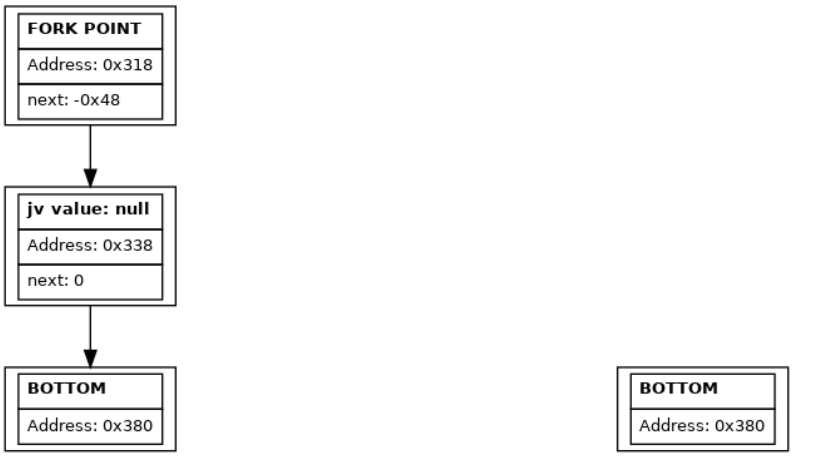
After DUP

(ix)



After CALL_BUILTIN _plus

(x)



(xi) (xii)

Figure 2.2: Evolution of the `jq` stack for the `jq` filter "`.+1, .+2`" with input `11`.

`jq` Calling Conventions

`jq` function calls at the bytecode level are handled in a manner that preserves `jq`'s functional nature and maintains specific contexts throughout execution. In `jq`, all arguments to functions are themselves functions, specifically closures. For example, in the expression `f(g)`, `g` is a closure that is passed as an argument to the function `f`. This implies that `g`'s closure value will be provided to `f` when it is invoked.

Under the hood, the `jq` interpreter represents closures as `<frame`

`reference, bytecode>` pairs. The frame reference is an integer that points to the environment or scope in which the closure was created. This corresponds to another frame relative to the current stack frame and ensures that the closure has access to the correct variables and context from its creation environment. The bytecode is a sequence of instructions that the `jq` VM will execute when the closure is called. This bytecode corresponds to the function logic that the closure encapsulates.

CHAPTER 3

CJQ: A BYTECODE-TO-LLVM COMPILER

This chapter introduces `cjq`, a novel `jq` bytecode-to-LLVM IR compiler. `cjq`'s architecture consists of two main components: the *tracing module* and the *bootstrapping module*. The tracing module records the sequence of dynamic bytecode opcodes executed by the standard `jq` VM and uses this trace to generate equivalent LLVM IR. The bootstrapping module implements the functions called from the generated LLVM IR, known as *opcode functions*. Additionally, the bootstrapping module deserializes runtime data that was previously serialized by the tracing module. Lastly, the bootstrapping module and the generated LLVM IR are compiled using Clang[2]. To minimize overhead from `jq`'s stack-based semantics and improve performance, I had Clang apply a series of strategic optimizations during compilation. In this section, I provide an overview of the tracing module, the bootstrapping module, and the optimizations applied to the linked LLVM IR.

3.1 The Tracing Module

Tracing in chunks

`cjq` differs from most LLVM frontends in the way it lowers from source code to LLVM IR. LLVM frontends for commonly used programming languages often employ a progressive lowering strategy [6]. This often involves lowering

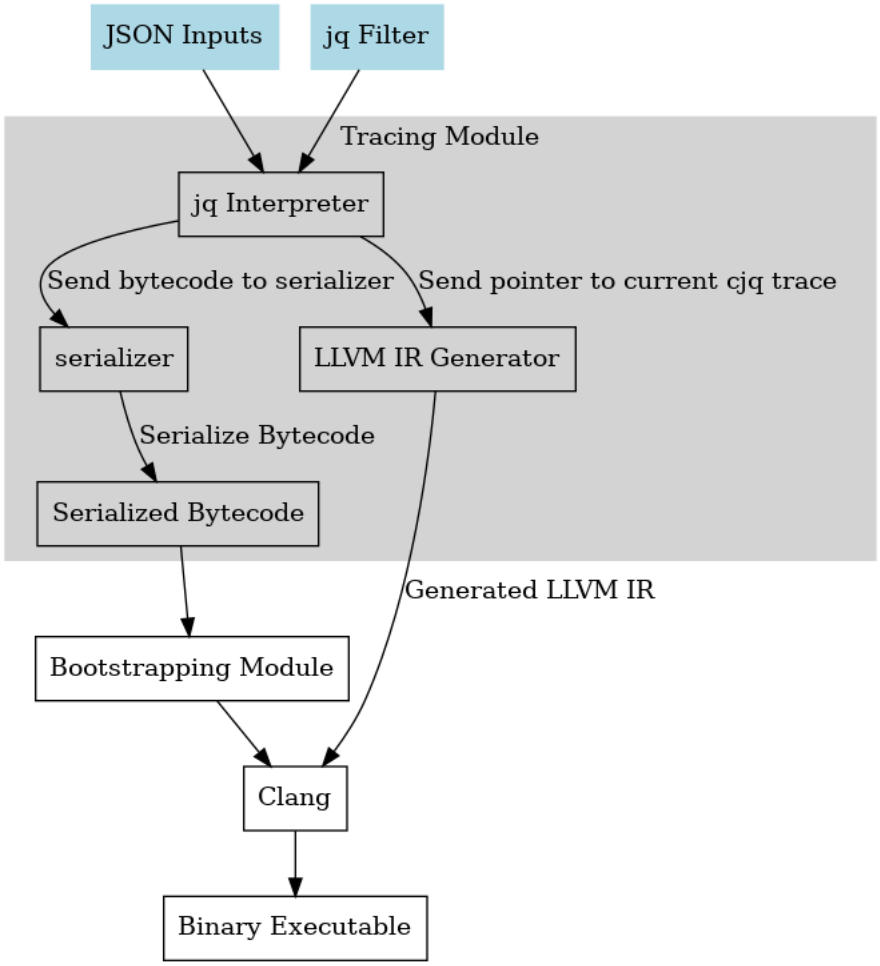


Figure 3.1: The compilation pipeline of the `cjq` compiler.

from source code to a source-specific IR, [1] then further lowering to LLVM IR.[2] `jq` is unique in the sense that it does not directly translate bytecode to a separate intermediate representation. Rather, the first step in the `cjq` pipeline is to interpret the `jq` bytecode using the same VM as the standard implementation. While the `jq` VM executes bytecode instructions, the tracing module records the dynamic sequence of opcode values, VM entry points, points in program execution when new JSON inputs are passed in, and points in program execution when the VM halts. As I will show, this tracing information is passed to

[1] The purpose of the source-specific IR is to perform optimizations that require source-level information that would otherwise be difficult to preserve.[6]

[2] In practice, there can be several more lowering steps before LLVM IR is generated.

a submodule known as the *lowering submodule* which builds an LLVM program from the series of ”trace chunks” passed in.

LLVM IR Generation

The `jq` VM records dynamic state information in chunks where a *chunk* is simply a set of dynamic arrays. When one of the chunk's dynamic arrays reaches some pre-determined size, *N*, CJQ *flushes* the values stored in the chunk. This means that `cjq` calls the LLVM IR Builder[7] via the llvmlite API[8]. The trace is passed to a Python program running llvmlite and, for each recorded opcode in the chunk, the LLVM IR Builder adds a call to an equivalent pre-defined opcode function. At runtime, the sequence of opcode functions will be used to modify the state of the `jq` stack in an equivalent way to the `jq` VM modulo LLVM optimizations. To preserve memory, the tracing module frees the current chunk before continuing to record dynamic state information.

Listing 3.1: Information stored in the cjq trace structure.

```
typedef struct {
    uint64_t count;
    uint64_t capacity;
    uint8_t* ops;
} opcode_list;

typedef struct {
    uint64_t count;
    uint64_t capacity;
    uint64_t* entry_locs;
```

```
} jq_next_entry_list;

typedef struct {
    uint64_t count;
    uint64_t capacity;
    uint64_t* input_locs;
} jq_next_input_list;

typedef struct {
    opcode_list* opcodes;
    jq_next_entry_list* entries;
    jq_next_input_list* inputs;
    uint64_t jq_halt_loc;
} trace;
```

Opcode Functions

A critical requirement of `cjq` is to preserve the implied control flow depicted in Listing 2.1. The `jq` interpreter accomplishes this through a VM that processes a stream of bytecode instructions. The `jq` VM is basically one large switch statement. Given a particular bytecode instruction, one or more updates to the `jq` program state are made. This process continues until all inputs have been fully processed. As the `cjq` tracing module records the dynamic sequence of opcodes, it builds an LLVM program that must effectively perform the same sequence of computations as the `jq` interpreter. To achieve this, an opcode function must be implemented for each bytecode opcode in the `jq` VM's instruction

set.

When an opcode function is executed, it performs the same mutations to the program state as the VM when it processes the corresponding bytecode opcode. In addition to ordinary opcode functions, `cjq` requires helper opcode functions. Essentially, helper opcode functions perform the updates to the program state that take place after each of the loops from Listing 2.1[3]. A helper opcode function is called for each VM entry point, new JSON value passed to the VM, and point in program execution when the VM halts. Listing 3.2 below shows the LLVM program generated from `cjq` compiling the resultant bytecode from the `jq` filter `".+1, .+2"` to LLVM IR. Note that this LLVM program performs the same sequence of computations as the `jq` VM when it processes the bytecode from Listing 2.2. The sequences of computations are identical because they're both compiled from the same `jq` source program.

Listing 3.2: Sequence of opcode functions generated by `cjq` after compiling the resultant bytecode from the jq filter ".+1, .+2"

```
define void @jq_program(i8* %.1) {
entry:
  call void @_get_next_input(i8* %.1)
  call void @_init_jq_next(i8* %.1)
  call void @_opcode_TOP(i8* %.1)
  call void @_opcode_FORK(i8* %.1)
  call void @_opcode_PUSHK_UNDER(i8* %.1)
  call void @_opcode_DUP(i8* %.1)
  call void @_opcode_CALL_BUILTIN(i8* %.1)
  call void @_opcode_JUMP(i8* %.1)
```

[3]See `cjq` source code for details.

```
  call void @_opcode_RET(i8* %.1)
  call void @_init_jq_next(i8* %.1)
  call void @_opcode_BACKTRACK_RET(i8* %.1)
  call void @_opcode_BACKTRACK_FORK(i8* %.1)
  call void @_opcode_PUSHK_UNDER(i8* %.1)
  call void @_opcode_DUP(i8* %.1)
  call void @_opcode_CALL_BUILTIN(i8* %.1)
  call void @_opcode_RET(i8* %.1)
  call void @_init_jq_next(i8* %.1)
  call void @_opcode_BACKTRACK_RET(i8* %.1)
  call void @_update_result_state(i8* %.1)
  ret void
}
```

Lowering

The "flushing" step refers to passing a pointer to a chunk to the lowering submodule. The lowering submodule is simply a Python program running llvmlite. In short, the lowering submodule appends opcode function calls to an LLVM IR program each time the current trace is flushed. If this were exactly true, the memory required by `cjq` would quickly become a bottleneck. To avoid this, the lowering submodule leverages the fact that there are only 43 unique bytecode instructions in the `jq` VM's instruction set. This implies that the dynamic sequence of opcodes is really a dynamic sequence of random numbers from the set $\{0, 1, 2, \ldots, 42\}$[4]. Moreover, the subsequences within the random se-

[4]There are actually more possible values due to the way backtracking is implemented but the number of possible values is roughly the same

quence of numbers are not entirely random because `jq` programs possess inherent structure and, like all programs, exhibit varying degrees of locality. This observation can be leveraged by caching common dynamic subsequences to avoid wasting memory.

To be precise, the lowering submodule performs the following steps each time the current trace is flushed:

1. The dynamic sequence of opcode values is recorded as a list of 8-bit values. This includes the helper opcodes representing VM entry points, points in program execution when new JSON inputs are passed in, and points in program execution when the VM halts. Each recorded opcode value is assigned a backtracking flag as well.
2. The list of 8-bit opcode values and backtracking flags are passed to a function that generates subsequences from the list. The subsequence generator divides the list into subsequences in a pseudorandom fashion[5]. This set of subsequences is unioned with a global set keeping track of all recorded subsequences that have been flushed so far.
3. Using the updated global set of subsequences, the list of dynamic opcode values is replaced by a list of references to subsequences in the global set. Importantly, the sequence of opcode values in the list of references to subsequences matches the sequence of opcode values in the original list.
4. To prevent the LLVM program from becoming too large, loops are generated to avoid unnecessary redundancy. This is accomplished by using a variant of run-length encoding (RLE) to compress the list of subsequence

[5]The current heuristic for generating a subsequence is to generate subsequences of non-decreasing opcode values. Empirically, this has resulted in subsequence functions containing roughly five opcode function calls.

functions. For example, if the sequence `(1,2,3)` appears ten times in a row, this subsequence will appear as `((1,2,3), 10)` in the compressed list.

5. The final step involves continuing to build the LLVM program using the list of subsequences. An important step, not previously discussed, is the creation of a mapping from each subsequence of opcode values to a unique *subsequence function*. A subsequence function is simply a function that sequentially calls each of the opcode functions corresponding to the opcode values in the subsequence. Subsequence functions are used to package common subsequences of opcode function calls, thus helping to avoid the size of the generated LLVM IR program from becoming a bottleneck.

The above steps repeat until the VM finishes executing the `jq` program. Continuing with the example jq filter "`.+1, .+2`", the generated LLVM IR is shown in Listing 3.3 below. Note that this generated LLVM program is equivalent to the one shown in Listing 3.2. For brevity, some of the function declarations and definitions are not shown.

Listing 3.3: LLVM IR generated by the lowering submodule from the `jq` filter ".+1, .+2"

```
define void @jq_program(i8* %.1) {
entry:
  call void @_subsequence_func1(i8* %.1)
  br label %block_0

block_0:                                          ; preds = %entry
  call void @_subsequence_func0(i8* %.1)
```

```
  br label %block_1

block_1:                                          ; preds = %block_0
  call void @_update_result_state(i8* %.1)
  ret void
}

...

; Function Attrs: noinline
define void @_subsequence_func0(i8* %.1) #1 {
subsequence_0:
  call void @_opcode_BACKTRACK_RET(i8* %.1)
  call void @_opcode_BACKTRACK_FORK(i8* %.1)
  call void @_opcode_PUSHK_UNDER(i8* %.1)
  call void @_opcode_DUP(i8* %.1)
  call void @_opcode_CALL_BUILTIN(i8* %.1)
  call void @_opcode_RET(i8* %.1)
  call void @_init_jq_next(i8* %.1)
  call void @_opcode_BACKTRACK_RET(i8* %.1)
  ret void
}
```

3.2 The Bootstrapping Module

The bootstrapping module is where the callee functions called from the generated LLVM IR are defined. For example, a common opcode function call that appears in the generated LLVM IR is `_opcode_CALL_JQ`. The bootstrapping module implements `_opcode_CALL_JQ`. With all of the opcode functions implemented, Clang can compile the generated LLVM IR with the bootstrapping module to produce a binary executable that performs an effectively equivalent sequence of computations to the `jq` interpreter.

As discussed in Chapter 2.2, the `jq` interpreter compiles the `jq` source code to bytecode. One of the goals of `cjq` was to avoid compiling the `jq` source code in both the tracing module and the bootstrapping module. To achieve this goal, the bytecode generated during the tracing stage is serialized to disk. The bootstrapping module later deserializes the generated bytecode, thus eliminating the need to recompile the `jq` source code. As a result, `cjq` users are only required to provide compatible[6] JSON data as input to the resultant binary executable to perform the JSON transformations they desire.

At runtime, the JSON input is read and converted to a `jv` value in the same manner described in Chapter 2.2. Next, the bytecode that was serialized during the tracing stage is deserialized. At this point, the program state has everything it needs to execute the `jq` program. A pointer to the program state is passed to the `jq_program` function, the function implemented by the generated LLVM IR during the tracing stage, and the sequence of opcode functions are called. For the `cjq` program state to survive across function calls, the bootstrapping

[6]Here, compatible means that the JSON input has the same format as the JSON input provided in the tracing stage.

module uses heap-allocated structures. This differs from the `jq` interpreter. As depicted in Listing 2.1, the `jq` VM is effectively a triply-nested loop. Thus, all program state is able to be declared outside of the outer-most for loop, allowing program state to be stack-allocated.

3.3 Optimizations

A logical question to pose at this point might be: *If `cjq` produces an LLVM program that just makes calls to functions that perform the same computations as the VM when it reads in the corresponding bytecode, shouldn't `jq` be roughly as fast as `cjq`[7]?* Without optimizing the result of the bootstrapping module, `cjq` has shown to be roughly the same speed as standard `jq`. However, the key insight to improving `cjq`'s performance lies in the strategic optimizations applied during the bootstrapping phase.

3.3.1 Exposing `jq`'s Stack-Based Execution Model

Within each opcode function is a sequence of stack-based computations to update the program state. For example, the opcode function `_opcode_PUSHK_UNDER` performs the following sequence of computations:

Listing 3.4: Sequnce of computations executed when _opcode_PUSHK_UNDER is called. Note that unimportant details of this code are not shown.

```
void _opcode_PUSHK_UNDER(void*) {
```

[7]For simplicity, assume that the overhead associated with the indirection of a switch statement is roughly equivalent to that of a function call.

```
    // Getting state info
    // ...

    // Grab constant integer from symbol table
    jv v = jv_array_get(sym_table, pc++);

    // Pop top of stack. Store popped value in temp
    jv v2 = stack_pop();

    // Push constant integer
    stack_push(v);

    // Push popped value
    stack_push(v2);
}
```

For the `jq` VM, performing analyses and optimizations on the bytecode at runtime would likely outweigh any performance gains. Conversely, when Clang compiles the generated LLVM IR with the bootstrapping module, Clang's powerful analyses and optimizations can be statically applied both within and across opcode functions. Specifically, Clang can perform global and interprocedural analyses, enabling optimizations that reduce the overhead of the `jq` VM's stack-based execution model. Examples include function inlining, constant propagation, dead code elimination, and loop unrolling. These optimizations are possible both globally and across function boundaries due to the interprocedural analysis pass.

Thus, the standard implementation of `jq` performs as a JIT might in interpreter mode, while `cjq` performs as an optimized, compiled trace would. By leveraging LLVM optimizations, the compiled `cjq` program can achieve significant performance improvements, making it competitive with, or even superior to, the original stack-based execution model.

CHAPTER 4

EVALUATING THE CJQ IMPLEMENTATION

This chapter compares the performance of `cjq` to the performance of the standard `jq` implementation across a variety of benchmarks and input sizes. I will use the results discussed here to justify future directions I plan to take the `cjq` project.

4.1 Evaluation

I evaluated the performance of `cjq` and `jq` by measuring their execution runtime on a set of benchmarks written by the maintainers of the standard `jq` implementation. I also included some handcrafted benchmarks of my own. These benchmarks possess a high degree of overlap with the language features discussed in Chapter 2.1.

I conducted the evaluation on WSL2 running Ubuntu 22.04, on a machine with an AMD Ryzen 7 5800H processor (eight cores at 3.2 GHz) and 16 GB of RAM. `jq` was compiled with GCC 11.4.0, and `cjq` was compiled with Clang 16.0.4.

Nine benchmarks were used in this evaluation. Five of these were pulled from a set of benchmarks provided by the maintainers of the standard `jq` implementation[11]. Measurements were taken using hyperfine[9]. For accuracy, all benchmarks were run at least ten times.

1. **schema-1**: This benchmark calls a `jq` script that returns a JSON document

describing the schema of the input JSON data[1]. The JSON input provided is one megabyte in size [2].

2. **length-jeopardy**: This benchmark uses the built-in `length` function to compute the length of a 54MB JSON file[11], **jeopardy.json**.
3. **identity-jeopardy**: This benchmark uses the built-in `identity` function which is piped to the Linux `wc` command. This benchmark also uses jeopardy.json as input.
4. **index-len**: This benchmark uses a combination of object indexing and the built-in `length` function on a 181MB JSON file[11], **citylots.json**.
5. **nested-indexing**: This benchmark uses a combination of array and object indexing to extract all valid data members from citylots.json.

I handcrafted the remaining four benchmarks. All `jq` filters and JSON inputs for these benchmarks can be found online[3].

1. **flatten-25**: This benchmark uses the built-in `flatten`[4] function to "flatten" a 25MB JSON file.
2. **openlib**: This benchmark uses a combination of many of the `jq` language features discussed in Chapter 2.1 to perform a series of complex extractions and transformations on the input JSON data.
3. **bsearch_1M**: This benchmark simply performs binary search on a sorted list of integers of size one million.

[1] In the online description of the standard `jq` benchmarks, a different JSON input is used. I used a smaller JSON input for this benchmark to reduce startup time.
[2] All benchmark programs and inputs can be found online[11]
[3] https://github.com/jdroob/cjq/tree/main/cjq/tests

4. **paths-5**: This benchmark uses the built-in `paths` function[4] to compute all compatible paths in a 5MB JSON file.

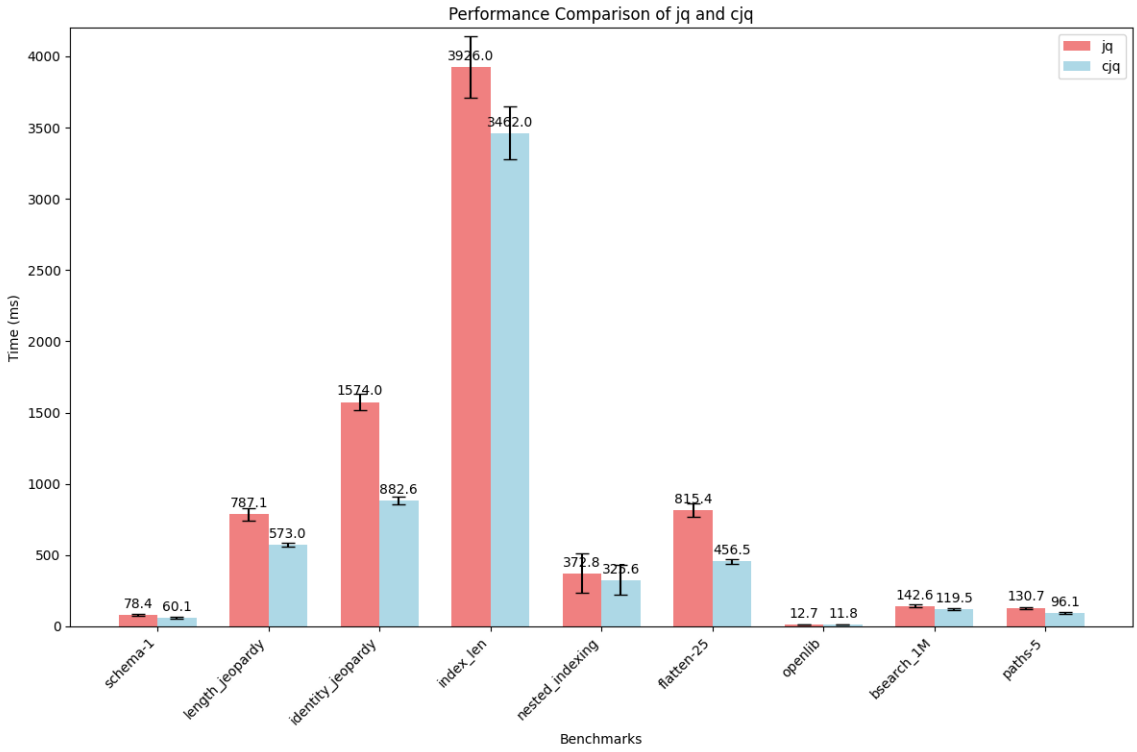


Figure 4.1: Runtime of `cjq` and `jq` for a set of benchmarks (lower is better)

The results are shown in Figure 4.1. In all benchmarks, `cjq` has a lower execution time than `jq`. `cjq`'s improved performance can likely be attributed to utilizing LLVM optimizations to reduce overhead associated with `jq`'s stack-based semantics.

The **identity_jeopardy** benchmark showed the largest performance improvement from `jq` to `cjq`, seeing a 44% performance improvement. This is likely due to there being no actual transformations of the input JSON. As a result, many of the default stack-based operations before pretty printing are likely being optimized away. Benchmarks with more complex filters such as **index_len**, **nested_indexing**, and **openlib** show the smallest performance im-

provements at 12%, 13%, and 7%, respectively. While these performance improvements are not insignificant, they may suggest that there is an upper limit to the amount of overhead associated with stack-based semantics that can be optimized away via LLVM optimizations.

One important caveat to consider in evaluating `cjq`'s performance is its startup time, which can vary significantly depending on the filter and input size. Startup times for `cjq` range from a few seconds to several hours. This variability arises primarily due to generating LLVM IR from traced opcodes and subsequently compiling the generated LLVM IR using Clang.

`cjq` excels at processing large JSON inputs when the transformations being performed do not require explicitly[4] evaluating a condition for each element within a JSON object or array. For example, in the `flatten` benchmark shown above, despite the JSON input size of 25 MB, the generated bytecode implicitly encodes the recursive behavior required to "flatten" nested arrays, resulting in a small number of dynamic opcodes relative to the input size. Conversely, when the recursive or iterative nature of the filter must be explicitly expressed in the bytecode, the result is a rapidly increasing number of dynamic opcodes. For example, when an integer is passed to the `jq` filter `[range(.)]`, there exists an explicit bytecode sequence that checks whether the current value is equal to the provided argument. This results in the number of dynamic opcodes increasing rapidly with the input, leading to a longer tracing phase and subsequent compilation time. While the tracing module utilizes a compression step to help mitigate the rate at which code size grows, `jq` filters that require explicit conditional bytecode sequences remain a bottleneck in the current version of `cjq`.

[4]Here, by explicit I mean there exists a sequence of bytecode instructions that evaluate the condition.

CHAPTER 5

CONCLUSION AND FUTURE THOUGHTS

This thesis introduced `cjq`, a novel compiler implementation of the `jq` programming language. `cjq` represents an initial step towards compiling `jq`, emphasizing correctness and runtime performance, at times, at the cost of increased startup time.

This contribution comes with some potential future directions:

1. Transitioning from stack-based bytecode to register-based bytecode and developing a register-based virtual machine are potential enhancements. These improvements, along with strategically encoding the implied control flow depicted in Listing 2.1, could help bridge the semantic gap between the current stack-based bytecode and register-based semantics. This transition may pave the way for implementing a traditional bytecode-to-LLVM IR compiler.
2. Exploring more efficient tracing techniques and advanced compression algorithms to mitigate current bottlenecks in the tracing phase.

BIBLIOGRAPHY

[1] Alfred V Aho, Brian W Kernighan, and Peter J Weinberger. Awk—a pattern scanning and processing language. *Software: Practice and Experience*, 9(4):267–279, 1979.

[2] Clang Team. *Clang Documentation*, 2024. https://clang.llvm.org/docs/index.html.

[3] M. Dahm. Byte code engineering. In C.H. Cap, editor, *JIT'99*, Informatik aktuell. Springer, Berlin, Heidelberg, 1999.

[4] Stephen Dolan. jq 1.7 manual. https://jqlang.github.io/jq/manual/, 2023. [Online; accessed 11-January-2024].

[5] Michael Hauben and Ronda Hauben. sed-history. https://web.archive.org/web/20180627160704/http://sed.sourceforge.net/sedfaq2.html#s2.1, 2018. [Online; accessed 20-February-2024].

[6] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *ArXiv*, 2020.

[7] LLVM Project. Llvm language reference manual. https://llvm.org/docs/, 2024. https://llvm.org/docs/.

[8] llvmlite Project. llvmlite documentation. https://llvmlite.readthedocs.io/en/latest/, 2024. https://llvmlite.readthedocs.io/en/latest/.

[9] David Peter. hyperfine: A command-line benchmarking tool. https://github.com/sharkdp/hyperfine, 2023. Accessed: 2024-07-18.

[10] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

[11] Nicholas Williams and William Langford. jq benchmarks. https://

github.com/jqlang/jq/wiki/X---Experimental-Benchmarks, 2023. [Online; accessed 18-January-2024].

[12] Nicholas Williams and William Langford. jq wiki. https://github.com/jqlang/jq/wiki, 2023. [Online; accessed 18-January-2024].