VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Performance of Kihi

Callum Li

Supervisor: Michael Homer

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours in Software
Engineering.

Abstract

This report outlines the progress of the Kihi honours project. Specifically, this report analyses the performance of three Kihi implementations, two of which were implemented for this project and the other a pre-existing NodeJS implementation. This report also provides all the necessary background knowledge required to understand the report, including an introduction to the Kihi language itself and adjacent concepts such as functional programming and compositional programming. An overview of the design of the two project implementations is also provided followed by an evaluation of all three implementations. Finally, an outline of potential future work is given.

Contents

Figures

Chapter 1

Introduction

The goal of this project is to optimise the performance of the Kihi programming language. The initial component of this project, and the topic of this report, is the design and construction of an implementation of the Kihi language. This actually resulted in two implementations due to multiple possible execution methods. The first implementation uses a term rewriting method and the second a stack machine based method. This report provides an analysis and evaluation of these two implementations. The evaluation compares the two implementations against each other and also against a third pre-existing NodeJS based implementation. Finally, an outline of potential future work is given.

Chapter 2

Background

2.1 Introduction to Kihi

This project principally concerns the Kihi programming language. Kihi is a compositional and functional programming language consisting of only six types of terms. These characteristics make it an interesting subject for an optimisation based project and its simplicity lends itself well to inquiry within a time-limited honours project. However, this simplicity also limits the space of possible optimisations, a challenge that will be discussed and addressed later.

The language's simplicity is best shown by its grammar and semantics. A Kihi program is simply a sequence of terms, where terms are one of five operators or an abstraction. An abstraction is itself a sequence of terms captured by parenthesis. The complete grammar is given in figure 2.1.

```
⟨program⟩ ::= { ⟨term⟩ }

⟨abstraction⟩ ::= '(' ⟨program⟩ ')'

⟨term⟩ ::= ⟨abstraction⟩
  |  'apply'
  |  'left'
  |  'right'
  |  'copy'
  |  'drop'
```

Figure 2.1: Kihi Grammar

The semantics of the language are equally simple. An operational semantics of the languages using term rewriting can be defined quite simply as shown in figure 2.3. The figure illustrates the effect of each operator. An explanation of these operators is also given in figure 2.2. The operational semantics shown in 2.3 also includes two execution rules: term rewriting and stack machine. The first allows for reduction to occur anywhere. However, the latter restricts reduction to the rightmost term. This mimics a stack based approach that will be discussed in greater detail later.

1. **Apply** releases the sequence of terms captured by an abstraction.
2. **Left** places the second abstraction at the start of the first
3. **Right** places the second abstraction at the end of the first
4. **Copy** copies an abstraction
5. **Drop** deletes an abstraction

Figure 2.2: English explanation of operators

.

$$p \in program = t_1 : t_2 : \ldots : t_n$$

$$t \in term = \{\text{apply}, \text{left}, \text{right}, \text{copy}, \text{drop}\} \cup \{(p) \mid p \in program\}$$

$$\frac{t_2 = (p)}{\text{apply} : t_2 \rightsquigarrow p} \qquad (\textsc{Apply})$$

$$\frac{t_2 = (p_1) \qquad t_3 = (p_2)}{\text{left} : t_2 : t_3 \rightsquigarrow ((p_2) : p1))} \qquad (\textsc{Left})$$

$$\frac{t_2 = (p_1) \qquad t_3 = (p_2)}{\text{right} : t_2 : t_3 \rightsquigarrow (p1 : (p_2))} \qquad (\textsc{Right})$$

$$\frac{t_2 = (p_1)}{\text{copy} : t_2 \rightsquigarrow t_2 : t_2} \qquad (\textsc{Copy})$$

$$\frac{t_2 = (p_1)}{\text{drop} : t_2 \rightsquigarrow} \qquad (\textsc{Drop})$$

$$\frac{p_1 \rightsquigarrow p_2}{p : p_1 : p' \rightsquigarrow p : p_2 : p'} \qquad (\textsc{Term Rewriting})$$

$$\frac{p_1 \rightsquigarrow p_2}{p : p_1 \rightsquigarrow p : p_2} \qquad (\textsc{Stack Machine})$$

Figure 2.3: Term rewriting operational semantics for Kihi

2.2 Compositional programming languages

A language is compositional if the concatenation of two programs written in that language produces a program that utilises the output of one program as the input to the other, also known as composition. These types of languages are often compared against applicative languages where the juxtaposition of terms denotes application in contrast to composition, such is the case in Haskell. For instance, in Haskell the program "`f g`" denotes f applied to g while in Forth, a concatenative language, the program "`g f`" denotes f applied to the output of g [1].

As mentioned earlier Kihi is also a compositional language. However, in contrast to Forth, Kihi is prefix concatenative as opposed to postfix concatenative. This means that values in Kihi appear to the right of the operator. In practice, this results in values and execution flowing from right to left. This property also means that values to the left of all operators will not be involved in any further computation and can be freely output. This compositional characteristic can also be shown as a result of the operational semantics of the language, see figure 2.4.

$$\frac{p_1 \rightsquigarrow p_1' \qquad p_2 : p_1' \rightsquigarrow p_2'}{p_2 : p_1 \rightsquigarrow p_2'} \qquad \text{(Composition)}$$

Figure 2.4: Composition

2.3 Functional programming languages

A functional programming language is one where execution is the evaluation of expressions. This is often compared against imperative languages where execution is the evaluation of statements which can modify some global state [2]. Kihi falls in the former category. A Kihi program can be understood as a sequence of expressions where the input to each expression is the result of the previous. In fact, this closely resembles the monadic style of computation provided by Haskell. These expressions also have no effect on the global state of the program since the base language has no side effects.

There are many touted benefits to functional programming languages. One such notable benefit is referential transparency. Put simply, this means evaluating any particular expressions will always yield the same value. This allows functional programs to be understood through relatively simple equational reasoning [2]. In contrast, deductive reasoning is often required to understand imperative programs. Figure 2.5 nicely illustrates this point. Specifically, in order to determine whether the output of the first and second print statements are identical an understanding the omitted code is required due to the mutability of the `hello` variable. This touches on the concept of immutability which underlies functional programming languages and enables referential transparency.

```
public class Main {
    public static void main(String[] args) {
        String hello = "hello world!"
        System.out.println(hello);
        ... omitted code ...
        System.out.println(hello);
    }
}
```

Figure 2.5: Simple Java program

2.4 Optimisation

Optimisation quintessentially amounts to improving some metric. In the context of programming languages these metrics are often real world quantities such as time, memory consumption, source code size, CPU utilisation [3]. However, more abstract quantities can and are also used such as instruction count, cache misses, or even reduction steps. In fact, optimising compilers often have to rely on these abstract quantities when optimising since measuring concrete quantities requires executing the program. The choice of metric is then essential to any optimisation process and more likely than not it will be abstract.

This is not to say that concrete performance is unimportant. In fact, the opposite is more true. Often these abstract quantities serve as tractable models of real world performance metrics such as those mentioned earlier. With respect to this project, the foremost goal is to reduce the time required to execute any given Kihi program. However, some difficulty arises from the functional nature of the language; any preprocessing optimisation step could simply replace the program with the result of executing it since there are no side effects.

2.5 Related work

2.5.1 Factor

Factor is concatenative language that features an optimising compiler [4]. In contrast to Kihi, it provides much higher level primitives and operators that allow it to solve practical real world problems. This higher level of abstraction makes many optimisations possible that are simply not available to Kihi. For instance, Factor features a high level optimisation stage where method inlining, overflow check elimination, and escape analysis are used to optimise the program. These optimisations do not make sense at the level that Kihi operates at. Ultimately, this suggests that future work should extend the basic facilities of Kihi to include higher level features. These features would come at a performance cost which could then be subsequently optimised away.

2.5.2 Limits of optimising compilers

There are numerous discussions on the limits faced by optimising compilers today. Bernstein presents the idea that optimising compilers fail to optimise critical code sufficiently or at the level of expert humans and that only non performance critical code benefits from machine optimisation [5]. However, in a language like Kihi which is essentially incomprehensible to humans the role of a machine optimisation is much more critical. In fact, it will often be the only form of optimisation possible. This issue could be alleviated through the introduction of higher level primitives that would make Kihi code more understandable, but that would bring its own computation costs.

Another interesting point is that optimisation is often a trade-off between time and performance [6]. Most compilers are not able to run the resulting code and rely on heuristics to determine which optimisations to perform. This is further complicated by the order dependence of optimisations. When one optimisation is applied often other optimisations become impossible. If time were not an issue, then all possible sequences of optimisations could be tried and evaluated until the best was found. This idea of trading off between time and performance is discussed in further detail in the future work section on peephole optimisation.

Chapter 3

Design and Implementation

3.1 Execution methods

As mentioned briefly before, there are two main methods that can be used to execute Kihi programs: a term rewriting or a stack machine based approach. The term rewriting approach is essentially described by the operational semantics shown in figure 2.3 and can be implemented very directly. The pseudocode shown in figure 3.1 describes a simple implementation. The essence of that algorithm is finding reducible terms and replacing them with their reduced forms.

The stack based approach understands execution as a series of stack transformations with each term transforming the stack in a particular way [7]. An operational semantics for this approach is shown in figure 3.3. This approach is inherently more sequential than the term rewriting approach since evaluation is restricted to the rightmost term. As the operational semantics show, when an abstraction is encountered it is pushed onto the stack, and when an operation is found, values from the stack are consumed and a result is pushed onto the stack or in the case of apply appended to the program. Pseudocode for this approach is shown in 3.2.

A basic complexity analysis of these two methods, assuming each term is evaluated at some point, reveals that the term rewriting method has a time complexity of $O(n^2)$ since each reducible term needs to be found before being executed. Since there are n terms and finding a term takes n operations the overall time complexity is $O(n^2)$. In contrast, the stack based approach has a significantly better time complexity of $O(n)$ since the position of the next term is known to always be the topmost term on the program stack.

However, these two execution styles are not equivalent for all programs. There are three classes programs: those which are partially evaluated, fully evaluated, and looping. The former class are programs whose result contains unevaluated operators. For instance, `left (apply)` is a partially evaluated since it contains left, in contrast to `(apply)` which is fully evaluated since it only contains values. Partially evaluated programs are essentially equivalent to partially evaluated functions in other programming languages such as Haskell. Where these two execution styles differ is when the result is partially evaluated the results may differ. This difference arises due to a restriction of the terms the stack based method is able to consider. It is only able to evaluate the topmost term and when that term is not provided sufficient arguments the program must stop. However, there may still be reducible terms below that term which would be reduced using the term rewriting based method thus causing a potential difference in the final results. This is the crux of the difference between these two execution methods. Whether this is of consequence depends on whether a program which ultimately can only be partially evaluated should be considered valid.

```
execute_program(input: String):
    terms: []Term := parse_program(input)

    reductible_term: Index := find_reductible_term(terms)
    while (reductible_term != -1) {
        reduce_term(terms, reductible_term)
    }

find_reductible_term(terms: []Term):
    for i: Index in 0..|terms| {
        if terms[i] is 'apply'
            and terms[i+1] is abstraction => return i
        else if terms[i] is 'left'
            and terms[i+1] and terms[i+2] are abstractions => return i
        else if terms[i] is 'drop'
            and terms[i+1] => return i
        ... and so on for each inference rule
        }
    }
    return -1

reduce_term(terms: []Term, term: Index):
    if terms[index] is 'apply'
        terms[index..index+1] = terms[index+1]
    else if terms[index] is 'left'
        terms[index..index+2] = [terms[index+2], ...terms[index+1]]
    else if terms[index] is 'drop'
        terms[index..index+1] = []
    ... and so on for each inference rule
```

Figure 3.1: Kihi Term Rewriting Pseudocode

Potential differences can also arise in looping programs for similar reasons. However since they never halt and thus have no well-defined result they are less significant. The simplest of these programs is `apply copy (apply copy)` which cycles between three states:

$$apply : copy(apply : copy) \rightsquigarrow apply : (apply : copy) : (apply : copy)$$
$$\rightsquigarrow apply : copy(apply : copy) \rightsquigarrow ...$$

The states involved in these cycles may differ depending on the execution method for the same reasons outlined above. For instance, simply image some reducible terms below the infinite loop shown above. However, some looping programs are more interesting than others. The one shown above cycles between a finite set of states. This is typically errant behaviour but not always. One can imagine intentionally writing a program that outputs a constant stream of ones. However, some looping programs do not cycle. For instance, imagine a counting program which runs forever but outputs an ever incrementing number. This program in Kihi is shown in figure 3.4.

3.2 Implementation

Ultimately, both execution styles were implemented. Rust was the language chosen for this task due to its low level capabilities which would be beneficial for this type of performance

```
execute_program(input: String):
    program: []Term := parse_program(input)

    stack: []Term := []

    while(|terms| != 0) {
        term := terms.pop()
        if term is abstraction => stack.push(term)
        else if term is 'apply' => program.append( stack.pop() )
        else if term is 'left' => {
            arg1 = stack.pop()
            arg2 = stack.pop()
            stack.push( [arg2] ++ arg2 )
        }
        ... and so on for each inference rule
    }
```

Figure 3.2: Kihi Stack Machine Pseudocode

$$p \in program = t_1 : t_2 : \ldots : t_n \qquad s \in (p_1) : (p_2) : \ldots : (p_n)$$

$$t \in term = \{(p), \text{apply}, \text{left}, \text{right}, \text{copy}, \text{drop}\}$$

$$\frac{s, p_1 \rightsquigarrow s', p_1'}{s, p_2 : p_1 \rightsquigarrow s', p_2 : p_1'} \qquad \text{(EXECUTION-1)}$$

$$\frac{s, p_1 \rightsquigarrow s', \varnothing}{s, p_2 : p_1 \rightsquigarrow s', p_2} \qquad \text{(EXECUTION-2)}$$

$$\frac{}{s, (p) \rightsquigarrow s : (p), \varnothing} \qquad \text{(ABSTRACTION)}$$

$$\frac{}{s : (p_1), \text{apply} \rightsquigarrow s, p_1} \qquad \text{(APPLY)}$$

$$\frac{}{s : (p_1) : (p_2), \text{left} \rightsquigarrow s : ((p_2), p_1)), \varnothing} \qquad \text{(LEFT)}$$

$$\frac{}{s : (p_1) : (p_2), \text{right} \rightsquigarrow s : (p_1, (p_2)), \varnothing} \qquad \text{(RIGHT)}$$

$$\frac{}{s : (p_1), \text{copy} \rightsquigarrow s : (p_1) : (p_1), \varnothing} \qquad \text{(COPY)}$$

$$\frac{}{s : (p_1), \text{drop} \rightsquigarrow s, \varnothing} \qquad \text{(DROP)}$$

Figure 3.3: Stack based operational semantics for Kihi

```
apply right copy right (apply apply left (right right (apply) copy)) (
    apply left (apply apply left left ()) copy right (apply apply left
     (apply) apply left left () apply left (copy)) apply left left ())
     (drop)
```

Figure 3.4: Counting program in Kihi

critical project. Consideration was also given to other low level languages such as C, C++, and Go but concerns about the level of abstraction and safety lead to the choice of Rust. C and C++ provides few mechanisms to prevent safety problems and C lacks higher level abstractions. Whereas Go's abstraction is too far disconnected to actual machine hardware.

These execution methods take the form of two interpreters. They share a parsing stage that parses a given string into a program. A program is represented internally as a vector of terms. The interpreters then directly execute this internal representation. An alternative architecture of a bytecode compiler and virtual machine was also considered but dismissed due to concerns regarding implementation complexity. However, it is still under consideration and might exist in the future. A machine code compiler is also still under consideration.

These two interpreters fundamentally operate as described in the pseudocode. However, the term rewriting method combines the term search and reduction steps into a single operation. Additional variables were also introduced in order to log additional metrics. However, these should have little effect on the performance of the program since the overhead they have is insignificant compared to the relatively complicated reduction code. However, logging I/O to standard output can be a bottleneck due to the limited speed of I/O so logging can be turned off with a command flag.

Chapter 4

Evaluation

4.1 Method

The evaluation was carried out by measuring the amount of real elapsed time taken to fully executed some benchmark programs. Three implementations are compared: the term rewriting and stack based interpreters created for this project and a pre-existing NodeJS implementation. Real elapsed time was chosen as the metric due to its simplicity and compatibility with the various implementations. It is also compatible across all languages allowing for potential future work to compare Kihi against more popular languages such as those presented in the ”Computer Language Benchmarks Game” [3]. Other metrics such as reduction step count, memory utilisation, and CPU utilisation were also considered but were deemed inappropriate. The ultimate goal of this project is to improve the speed of the language and the simplest and most direct measure of that is real world elapsed time. This method may not work for many popular general purpose languages since the elapsed time is greatly dependent on usage of particular language features. However, the simplicity of Kihi means all programs written in Kihi will need to utilise all the language features offered by Kihi. Later work may mean reduction steps can be reduced but at the moment no work has been done to optimise it so it would not provide much insight.

The chosen programs are described in figure 4.1. They utilise all of Kihi’s features and each class of program is represented. In the case of infinitely looping programs a finite number of iterations is performed and the time taken to reach that state is measured. However, looping programs which cycle between a finite number of states without output were avoided simply since they serve no practical purpose in a side effect free language.

1. **Count**; Outputs all the natural numbers (as church numerals) in order.
2. **Duplicate**; Given a sequence of numbers from 1 to 25 output the same sequence with each element repeated.
3. **Select first**; Select the first number from a given sequence of numbers, specifically the numbers from 1 to 25.
4. **Select last**; Select the last number from a given sequence of numbers, specifically the numbers from 1 to 25.

Figure 4.1: Benchmark programs

The executables were run with their priority set to 99 and a FIFO scheduling policy using `chrt -f 99` and the measurements were made with `perf stat -r 10 -ddd` on a computer with a i5-4670K CPU @ 3.40GHz and 8GB of 2400MHz ram. The implementation and code can be found at `https://gitlab.ecs.vuw.ac.nz/project489-2019/licall/kihi-vm` and the specific commit used in this report is 9f0ee77030a60f5d6d5b9d2ef4fc5f9b3291da81. The first command, `chrt -f 99`, reduces the effect of scheduling and other running applications on the results of the measurements. `perf stat -r 10 -ddd` provides detailed information about how an executable runs providing information such as elapsed real time, user time, and system time. It also provides information about the number of cache misses and context switches but these were unused in this analysis.

4.2 Results

	Term rewriting	Stack machine	NodeJS
Count (1000 outputs)	12.2074s ± 0.0714s	12.5954s ± 0.0804s	12.783s ± 0.124s
Duplicate	0.091431 ± 0.000432	0.238114 ± 0.000231	0.07078 ± 0.00366
Select first	0.044579 ± 0.000316	0.134209 ± 0.000184	0.063090 ± 0.000549
Select last	0.134329 ± 0.000134	0.250103 ± 0.000138	0.067948 ± 0.000169

Table 4.1: Real time elapsed for various Kihi programs and implementations

	Real time	User time	System time
Term rewriting	12.433s	12.207s	0.069s
Stack machine	12.447s	12.266s	0.0029s
Node	13.637s	26.178s	3.915s

Table 4.2: Various time metrics for count program

4.3 Analysis

These results are relatively surprising. The stack machine implementation, although theoretically better, performs noticeably worse than the term rewriting based implementation. It performs worse across the board across all the benchmark programs. The difference is not significant for the 'count' program but is more than twice as slow than term rewriting for 'duplicate' and 'select first'. This suggests there may be a flaw in the implementation and that further profiling is needed to determine which parts of the algorithm are most computational expensive. A potential explanation is that the stack based approach might have a larger initial computation cost, this is supported by the similar results for the longer running 'count' program.

Overall, the term rewriting is comparable to the NodeJS implementation. However, the NodeJS implementation is more consistent across benchmark programs than the term rewriting based implementation. Term rewriting's results vary from 0.04s to 0.13s for the short running programs ('duplicate', 'select first', and 'select last') but the respective NodeJS results are all within the 0.6s to 0.7s range. This suggests that the NodeJS approach uses a different execution style. The term rewriting based approach starts searching for terms from the left, which implies programs which only have reducible terms on the far right will perform worse. The fact that NodeJS is spared from this variability means that its execution method is more stable.

A preliminary inspection of the results suggests that the NodeJS implementation uses multithreading. Specifically, since the CPU time used exceeds the real world elapsed time. A multithreaded evaluator would enable a Kihi program to be reduced from many points simultaneously, not only improving overall running time but also consistency. This aligns with the results found in this experiment.

Chapter 5

Future work

5.1 Higher level abstractions

The results have shown that a multithreaded model may be able to improve performance and consistency. This should relatively simple to implement due to the functional and side effect free nature of the language. This means a trivial parallel implementation would only require slicing a program into multiple parts since there are no side effects, each part could then be executed simultaneously with output flowing as needed from one part to another.

Research into related work in this space has shown that the most significant optimisations exist at higher levels of abstractions. This has proven to be especially true for Kihi which lacks any expensive high level abstractions. Furthermore, providing Kihi with higher level abstractions would allow for more complicated programs to be written and benchmarked. Thus, a reasonable future goal would be the introduction of such high level features. The high level features that are being considered are user defined functions and built-in integer and string data types. The former would allow for users to define modular programs, which would allow for optimisations such as inlining and memoisation. A built-in integer data type would significantly improve the performance of programs that are currently restricted to using church numerals. These new data types would also significantly improve quality of life for programmers writing Kihi.

5.2 Peephole optimisation

Another worthwhile area of investigation is peephole optimisation. This optimisation technique is very well suited to a concatenative language such as Kihi. It consists of simply finding a series of terms with a faster but equivalent series of terms. There is significant research being done in this area and it would interesting to see those ideas applied to Kihi. For instance, Bansal and Aiken propose a method which can be used to automatically generate superoptimisers [8]. The crux of their idea is to find equivalent terms through a memoised search process. This can be easily applied to Kihi and used to remove null operators or replace longer sequences with shorter ones. The idea of peephole optimisation is also used by the Factor optimising compiler [4].

5.3 Timeline

The current plan is to add multithreaded capabilities to both implementations and redo the evaluation by the end of June 2020. The first week of July will then be dedicated to adding the user defined functions and the week after will be dedicated to introducing additional data types.

Bibliography

[1] What is the Forth programming language? [Online]. Available: https://www.forth.com/forth/

[2] Functional programming - HaskellWiki. [Online]. Available: https://wiki.haskell.org/Functional_programming

[3] Which programming language is fastest? — Computer Language Benchmarks Game. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/

[4] S. Pestov, D. Ehrenberg, and J. Groff, "Factor: A dynamic stack-based programming language," p. 15.

[5] Cr.yp.to: 2015.03.14: The death of optimizing compilers. [Online]. Available: http://blog.cr.yp.to/20150314-optimizing.html

[6] Laurence Tratt: What Challenges and Trade-Offs do Optimising Compilers Face? [Online]. Available: https://tratt.net/laurie/blog/entries/what_challenges_and_trade_offs_do_optimising_compilers_face.html

[7] J. Purdy. The Big Mud Puddle: Why Concatenative Programming Matters. [Online]. Available: http://evincarofautumn.blogspot.com/2012/02/why-concatenative-programming-matters.html

[8] S. Bansal and A. Aiken, "Automatic Generation of Peephole Superoptimizers," p. 10.