Perfecting Nothingness by Refactoring Whitespace

Rutger Witmans[1], Vadim Zaytsev[1,2]

[1]Technical Computer Science, University of Twente, Enschede, The Netherlands
[2]Formal Methods & Tools, University of Twente, Enschede, The Netherlands

Abstract

In this paper we explore the possibilities of refactoring code in Whitespace, a programming language which only recognises whitespace characters as code, and tolerates textual comments that describe the program. The paper presents a list of possible refactorings applicable to Whitespace, and describes a tool that automates some of these refactorings. The functionality of the tool is demonstrated with concrete examples.

Refactoring is a important systematic process of improving code without creating new functionality, improving long-term properties of the code such as readability, maintainability, changeability, testability, extendability and safety. We argue that, despite the lack of real-life applications for Whitespace specifically, it is beneficial to apply refactoring methodology to it, since lessons learnt from esoteric languages can be ported elsewhere — in this case, to assemblers and similarly restrictive software languages.

Keywords

Whitespace, program refactoring, esoteric languages, second generation languages

1. Motivation

There are many ways to classify software languages [16]. One of them is a spectrum from the most mainstream and widespread languages, to most exotic and esoteric ones. The mainstream side can be represented by the TIOBE index [12], the current top ten being Python, C, Java, C++, C#, Visual Basic, JavaScript, SQL, PHP and Go. On the other side, esoteric languages, as the name suggests, are designed for one specific purpose of local interest: to have the smallest compiler, as it was the case with BRAINF*CK [7], to use statements that are as far from all other languages as possible, as it was with INTERCAL [27], or to provide a feasibly tiny playground for implementing legacy languages, as it happened with BABY-COBOL [30]. One of such languages is WHITESPACE [4], and it was designed from a driving principle that whitespace — the part of the source code which is traditionally ignored by the compiler as insignificant — is precisely the only part of the code which is significant, and the rest of the code such as visible punctuation, letters and numbers, are insignificant and skipped by the compiler. The language was designed by Edwin Brady around 2003 [4], and has enjoyed some attention in the meantime, leading to the existence of many implementations and programs to try software evolution techniques and tools on.

Refactoring [18, 19] can be used as a standalone technique, often applied manually by developers (with automation support from the IDE) with the original intent — to improve the design of existing code [6]. However, it is also very useful as a part of composite techniques. For instance, one can apply it as a program transformation on elements of a test suite, possibly augmenting it with more test cases with known execution outcomes. In the past, this is exactly what the second author has tried to do [9] to augment the labour-intensive process of testing the Raincode Assembler Compiler [3, 29] with mutative fuzzing. The endeavour was ultimately unsuccessful: fuzzing only worked on the level of macros (where it did contribute somewhat, and found at least one off-by-one bug in the compiler), but the original goal of testing the instruction implementations failed. The main reason was the difficulty to define any kind of refactoring transformations that make sense: changing even one bit of the test program had potentially numerous and hardly predictable effects.

Several years later, we try a different approach: instead of codeveloping all the elements of the fuzzing infrastructure, we focus only on refactorings; and instead of facing a gigantic language requiring 1500+ pages of documentation just to cover the byte-level basics, we focus on one tiny esoteric language with similar properties — namely, difficulty of defining what constitutes a refactoring within it. If by any chance our results will happen to help some WHITESPACE developer to improve readability of their code, that could only make the world a better place to live in.

SATToSE'23: Post-proceedings of the 15th Seminar on Advanced Techniques and Tools for Software Evolution, June 2023, Fisciano, Italy
r.c.h.witmans@student.utwente.nl (R. Witmans);
vadim@grammarware.net (V. Zaytsev)
https://github.com/rwitmans (R. Witmans);
http://grammarware.net (V. Zaytsev)
0000-0001-7764-4224 (V. Zaytsev)

2. Background

In the vast realm of software languages, where intricacies of syntax and verbosity often shape the landscape, an esoteric programming language stands out, challenging these conventions. Whitespace [4], a language conceived by Edwin Brady around 2003, transcends the norms and redefines the very essence of coding, highlighting the beauty that can be found in minimalism and turning whitespace perception inside out.

At its core, Whitespace embodies a design that is deliberately unconventional. While mainstream programming languages rely on an array of keywords, operators, and identifiers [28], Whitespace takes a dramatically different approach. Its code comprises just three fundamental characters: space (which we will denote as `\_`), tabulation (`\t`) and newline (`\n`). In this minimalistic arrangement, Whitespace signifies the essence of functionality through its whitespace, presenting a stark departure from the verbose nature of traditional software languages.

In essence, Whitespace stands as a testament to purity in programming. In contrast to languages where code is defined by intricate syntax, Whitespace's elegance lies in embracing emptiness. Its programs are essentially assembler-level algorithms expressed as sequences of whitespace characters, where patterns of spaces and tabs elaborate computational operations. This purity extends beyond aesthetics; it mirrors a programming philosophy that peers into the essence of computation, devoid of distractions. This brings Whitespace closer to IBM High Level Assembler [3, 29], Intel IA32 Assembler [11], PowerPC Assembler [1], ARM Assembler [13], Forth [21], to some extent even to C [23] or at least its more restrictive variants like SAC [8].

Central to Whitespace's uniqueness is its distinctive mode of operation. The language is built exclusively around a stack-based approach — just like Forth [21] and some of the most disciplined assemblers. Commands, represented by sequences of whitespace characters, manipulate this stack, orchestrating computations that resonate with a certain enigmatic beauty. The interplay between whitespace and functionality is an embodiment of Whitespace's duality, where the visually unassuming code conceals complex calculations beneath its surface.

For programmers accustomed to conventional languages, engaging with Whitespace presents a unique challenge. Crafting meaningful programs requires a fundamental shift in the programming mindset, relying on whitespace characters to carry the program's meaning and on non-whitespace characters to provide comments. The latter aspect brings it conceptually closer to literate programming [14, 20].

In this paper, our focus lies on the potential for code refactoring, exploring how this process can amplify the elegance and functionality of this distinctive programming paradigm. However, first we explain the basics of the language itself, to make this story self-contained.

Whitespace supports the following instructions:

- `\_\_` = `push` (a value on the stack)
- `\_\n\_` = `dup` (duplicate the top of the stack)
- `\_\t\_` = `lift` (copy the n^{th} item on the stack onto the top of the stack)
- `\_\n\t` = `swap` (the top two values on the stack)
- `\_\n\n` = `drop` (the top of the stack)
- `\_\t\n` = `dropN` (slide n items off the stack, keeping the top item)
- `\t\_\_\_` = `add` (two top values on the stack)
- `\t\_\_\t` = `sub` (subtract the top value on the stack from the one after it)
- `\t\_\_\n` = `mul` (multiply two top values on the stack)
- `\t\_\t\_` = `div` (integer division of two top values on the stack)
- `\t\t\_` = `store` (in heap)
- `\t\t\t` = `retrieve` (from heap)
- `\n\_\_` = `label` (define for later use)
- `\n\_\n` = `jump` (unconditionally)
- `\n\_\t` = `call` (to return later)
- `\n\t\_` = `jumpZ` (conditionally, if zero is on the top of the stack)
- `\n\t\t` = `jumpNeg` (conditionally, if the top of the stack is negative)
- `\n\t\n` = `return` (get back from a call)
- `\n\n\n` = `stop` (the programme)
- `\t\n\_\_` = `writeC` (output the character at the top of the stack)
- `\t\n\_\t` = `writeN` (output the number at the top of the stack)
- `\t\n\t\_` = `readC` (read a character and place it in the location given by the top of the stack)
- `\t\n\t\t` = `readN` (read a number and place it in the location given by the top of the stack)

For this project, we wanted to rely on a tool which parses Whitespace code and turns it into an intermediate representation, and after the refactorings turn the newly refactored intermediate representation back into Whitespace code. We have decided to use the Rust library "`whitespace-rs`" [5]. This tool has all the features necessary for testing, creating and transforming Whitespace code. The library has created its own intermediate representation, thus saving us the hassle of coming up with such a representation. The library is able to run our Whitespace programs, giving us the ability to test for changes in behaviour.

To show what our Intermediate Representation (IR) looks like, we first have to explain how Whitespace

Listing 1: "The IR of Whitespace"

```
Push {value: Integer},
PushBig {value: BigInteger},
Duplicate,
Copy {index: usize},
Swap,
Discard,
Slide {amount: usize},
Add,
Subtract,
Multiply,
Divide,
Modulo,
Set,
Get,
Label,
Call {index: usize},
Jump {index: usize},
JumpIfZero {index: usize},
JumpIfNegative {index: usize},
EndSubroutine,
EndProgram,
PrintChar,
PrintNum,
InputChar,
InputNum,
```

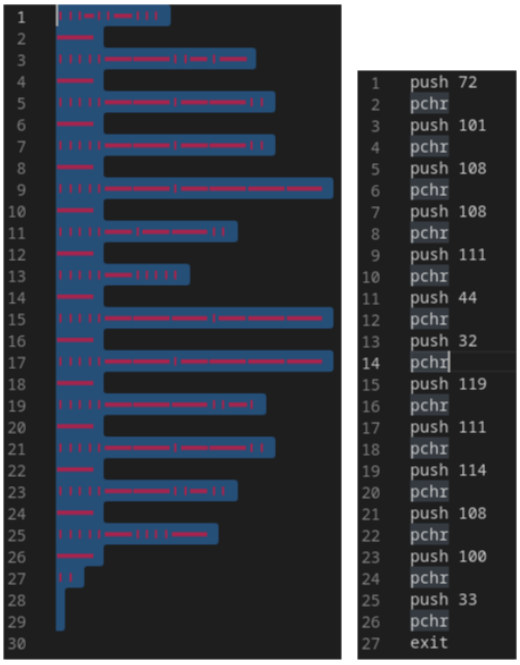

Figure 1: *Hello, World* in Whitespace and Whitespace IR

works. Whitespace has five different types of commands. These types all have a different Instruction Modification Parameter (IMP). The IMP is a unique sequence of whitespace characters that selects one of these instruction types. After choosing an instruction type, you now enter the corresponding combination of whitespace characters to select the instruction you want. Some of these instructions have parameters, which are a sequence of tabs and spaces, terminated with a Line Feed character. All Whitespace programs end with three line feed characters, indicating that there is no more code to parse. Combining all these instructions gives us a total of 24 instructions in the Whitespace language.

With this in mind, in Listing 1 you will see the IR of the Whitespace library we have decided to use. All the 24 commands have their own unique and human-understandable name (in our explanation earlier we have leaned towards decades-long terminology established by Forth, but here we see some less context-aware choices like using `Discard` instead of `drop`; its pretty-printed version differs yet from this internal representation). Using this IR, we are able to create our refactorings in an easier-to-understand language.

In Figure 1, on the left you will see a complete working program in Whitespace, specifically, a "Hello, world!" program. In the example, you see a combination of spaces (the vertical stripes), tabulations (the horizontal stripes) and newline characters at the end of each line. For most people, it is not clear how this program should behave. For example, some lines contain more than one instruction. That is why we would like to use the IR. In Figure 1, on the right you can find the IR version of the same Whitespace program. Here it becomes clear that every letter first gets pushed onto the stack by their ASCII code and then printed. When it finally finished printing the last character, the program exits. Using the IR to create Whitespace programs was convenient for us since it sped up the time it took to create and analyse test programs.

3. Possible Refactorings

Since the world has moved way past the list of refactorings proposed by Opdyke [19] and Fowler et al [6] in the 1990s, we have mostly relied on developer-created grey sources like *Refactoring Guru* [24]. We first looked at what refactoring categories are possible. The following categories are available:

- **[CM]** Composing Methods
- **[MF]** Moving Features between Objects
- **[OD]** Organising Data
- **[SE]** Simplifying Conditional Expressions
- **[SM]** Simplifying Method Calls
- **[DG]** Dealing with Generalisation

- **[CS]** Code Smells

We are ruling out everything that has to do with object programming patterns since Whitespace is an assembler-like language. Such languages generally miss the object programming data structures needed to perform such refactorings. Thus **[MF]**, **[OD]** and **[DG]** are high-level refactorings which we will not translate that well. Whitespace does have instructions which are called labels. These labels are points in the code you can jump to where a certain piece of code gets executed. This functionality borrows some refactoring ideas from the **[SM]** and **[CM]**. Whitespace also has instructions for conditional jumps. This makes some of the ideas for **[SE]** possible. Next to this, the list of code smells look promising enough to deliver at least some refactorings for us to perform, so we will be looking into **[CS]** as well. We will thus be looking through the following categories:

- **[CM]** Composing ~~Methods~~ Labels
- **[SE]** Simplifying Conditional Expressions
- **[SM]** Simplifying ~~Method Calls~~ Label Jumps
- **[CS]** Code Smells

With these chosen categories, we have created a list of refactorings which can be performed on Whitespace code. The following list is the refactorings is our result so far:

- **[EM]** Extract method
- **[IM]** Inline method
- **[RM]** Rename method
- **[CC]** Consolidate conditional expression
- **[CD]** Consolidate duplicate conditional fragments
- **[RD]** Remove dead code
- **[RC]** Remove clone/duplicate methods

3.1. [EM] Extract method

The extract method refactoring is a refactoring where a grouped sequence of instructions gets extracted into its own method so that this new method describes with its method name what the sequence of instructions is supposed to do. This is useful when you have a large method which does multiple sub-tasks to perform its functionality. Making it clear what the function does in these sub-steps is nice for the next reader of the code, so the readers are able to easily deduce what your code does.

3.2. [IM] Inline method

The inline method refactoring is the opposite of this. If some functionality of a method is small, there is the possibility of performing that function on the spot. Refactoring code with this method gets rid of code which clutters the program without bringing new functionality. We see that these first two methods of refactoring have opposite ideas in mind, yet both methods are able to be utilized exclusively from each other. For some methods, you might have made use of too many methods. This makes it unclear how the method works. On the other hand, using too few methods overwhelms the reader and makes the reader get lost in certain details which are not important. Because of this balance, it will be tricky to automate this process. While it is possible to automate this based on self-defined predicates, we will not be doing this in our paper because this is beyond the scope of this research.

3.3. [RM] Rename method

The rename method refactoring is quite self-explanatory. The purpose of this refactoring normally is to rename the method in order to make it more clear what the method does. In the case of Whitespace, this is impossible. Labels do not have ordinary names. Instead, they are made up of a combination of tabs and spaces. Because of this, the naming of labels is purely there to keep uniqueness. However, since the naming does not matter, we instead rename the labels to keep them as small as possible. Not only will this increase the number of labels we will have available to us, but it will also allow us to keep the Whitespace code as small as possible.

See § 4.1.1 for implementation details of this refactoring.

3.4. [CC] Consolidate conditional expression

The consolidate conditional expression refactoring is a refactoring method where one looks at all the different branches and then checks what branches lead to the same instructions. We then group these branches into a singular conditional statement that performs these actions. Grouping these conditionals gives clarity to code, especially if you name this expression. While this can be done in Whitespace using labels and performing the conditional logic under one of these labels, we would like to argue that this refactoring is still too subjective. We cannot easily decide whether a conditional statement is complex and needs changing. We have thus decided not to implement this refactoring into the tool.

3.5. [CD] Consolidate duplicate conditional fragments

The consolidate duplicate conditional fragments refactoring checks whether all branches execute the same piece of code and then extracts this piece out of the branches. This refactoring makes clear what piece of code always needs to be executed no matter what conditional branch

you might have taken. This clears up confusion about what the if-statement tries to separate resulting in cleaner code. We have chosen not to implement this method.

3.6. [RD] Remove dead code

To explain removing dead code, we will first explain what dead code is. "Dead or inactive code is any code that has no effect on the application's behaviour" [15]. With this definition, we see that we want to remove code that has no effect on the application we are writing. While this is trivial to do as a human, as a robot it is quite hard to notice when code is unused. Because of this, we will eliminate unused methods instead, to keep complications lower.

See § 4.1.2 for implementation details of this refactoring.

3.7. [RC] Remove clone/duplicate methods

Last up, we will be removing duplicate methods. Duplicate methods are two methods which have the exact same functionality. For Whitespace, this will mean to us that there are two different labels which are followed by the same code and are not (co)recursive. Our tool is going to remove these methods since duplicate methods only cause confusion and do not have any benefits to a programmer.

See § 4.1.3 for implementation details of this refactoring.

4. Evaluation

4.1. Implementation details

In our proof of concept [25], we have implemented several refactorings, which we explain below in this section.

4.1.1. [RM] Rename method

Given the small size of the space of possible label names in Whitespace, the obvious automated refactoring would be one that minimises label names. Serendipitously, this functionality was already included in `whitespace-rs`, so we could simply rely on their implementation to achieve our first working refactoring [5].

4.1.2. [RD] Remove dead code

For our dead code removal, we have created a plan to detect unused methods and then remove these methods. Our plan is as follows:

- Look at all our jump instructions and store to what label they jump to.
- If there is a label which is not jumped towards, we will eliminate this label with the code corresponding with this label.

Using this approach we are easily able to detect if methods are not called. There are some downsides to this method which we will now point out. If there are two methods which will reference each other that are not called through the main method, they will both still be seen as used code. This can be fixed by storing the label in which the method is called, and seeing whether this name space is reached via the main method. If it is, then this piece of code is not dead, otherwise, you can mark it as dead code.

That would not fix the second issue, however. If the code mentions a jump to a certain label, but it would never take this jump, then this called method would still be seen as a used piece of code. However, This cannot be true since this part of the code is never reached. One would have to guarantee that this piece cannot be reached using more complicated techniques.

Finally, we are just looking at dead methods and not dead code in general. If code is specifically told to stop the execution and there are calls to other methods after stopping execution, these called methods should be seen as dead. However, since we have not put in checks to detect this behaviour, these methods are not removed.

A combination of [RD] with [RM] can be seen on Figure 2 (in pure Whitespace) or on Figure 3 (in Whitespace IR).

4.1.3. [RC] Remove clone/duplicate methods

For removing duplicate methods, we have created a plan to detect these instances. Our plan is as follows:

- Analyse the code of all the methods.
- Group methods that have duplicate code.
- Remove grouped methods until there is one left.
- Change all jumps from the removed labels to the grouped method that is left.

With this, we have created a way to remove duplicate code without changing behaviour. There is one issue left with this implementation. If two instructions are swapped which are interchangeable, this plan would not be comprehensive to detect all method duplication. The way to fix this interchangeable code problem is to find all patterns where code can be interchanged without changing behaviour and detect duplicate code using these patterns.

A combination of [RC] with [RM] can be seen on Figure 4 (in pure Whitespace) or on Figure 5 (in Whitespace IR).

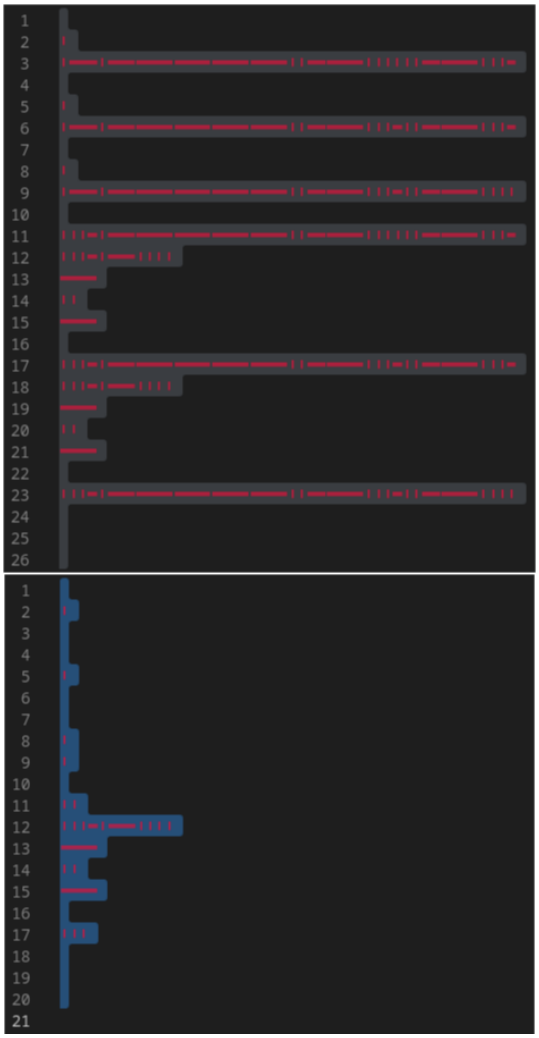

Figure 2: Duplicate method removal: before and after

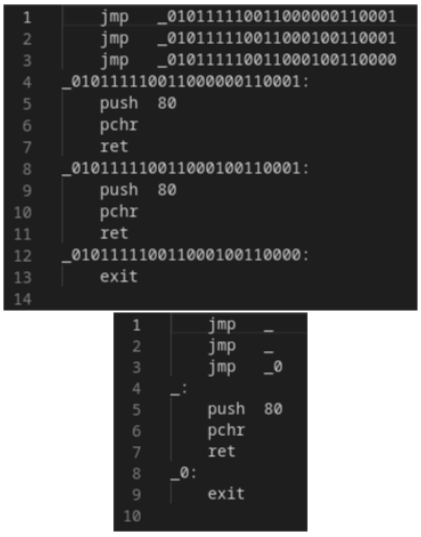

Figure 3: Duplicate method removal: before and after, in IR

4.2. Testing

With all of the refactorings finished, we needed some test programs to test whether the refactorings are applied correctly and kept their behaviour. This turned out to be a problem, since writing valid Whitespace code is not human-friendly. However, we solved this problem by writing in the format of the library their IR. The library was then able to recognise this format and transform the IR into a whitespace-encoded file, solving the issue of writing Whitespace code.

4.2.1. Testing [RD]

In software languages that permit low-level branching constructs, there are many ways to use a "method", and thus dead code detection must run very advanced code analysis algorithms and apply domain-specific heuristics. For example, HLASM (IBM High Level ASseMbler) has an EX(ECUTE) instruction which can modify the target address of a branching instruction at runtime. COBOL and BabyCobol [30] have a statement that can ALTER a target of an existing GO TO statement at runtime. Older versions of FORTRAN had computable GOTOs, as does BabyCobol. Luckily, Whitespace is a bit more straightforward, and making a call graph of all `Call` and `Jump` locations to all the `Label`s, is sufficient, if we take fall-throughs into account.

4.2.2. Testing [RC]

Code clone management has been a topic of research for many years [22]. Researchers and practitioners identify many different clone types and clone detection method families [10]. For this research, we opted for type-1 equivalence (precise character-level equality) of labelled sections after all possible `[RM]` and `[RD]` refactorings have been applied. Our tool marks clone pairs and in the second pass removes one of them and replaces all calls to it with the calls to the remaining one.

5. Advanced Refactorings

In this section, we consider other possible refactorings of Whitespace code which were not directly followed from the *Refactoring* book [6] and its derivatives, but conform to the general expectations about code refactoring, which improving design while preserving observable behaviour.

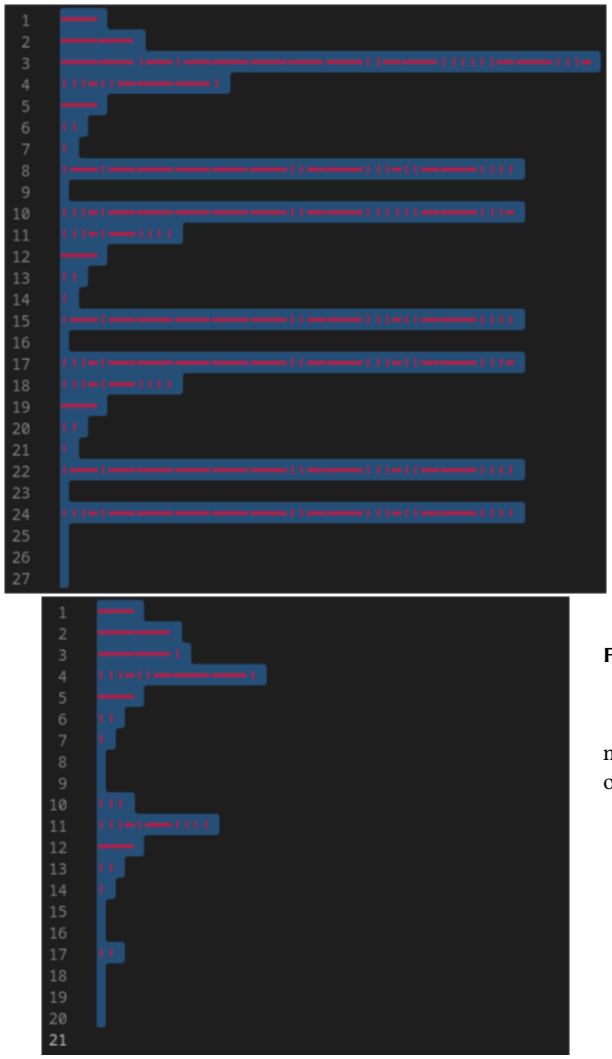

Figure 4: Unused method removal: before and after

Figure 5: Unused method removal: before and after, in IR

5.1. Control Flow Optimisation

Control flow optimisation is not uncommon in compiler construction since at least 1960s [2, 17]. It is an analysis and transformation technique known for enhancing program efficiency and readability, and especially performance. In the context of Whitespace, control flow optimisation takes on unique significance. This section delves into the intricacies of control flow optimisation within the confines of Whitespace, discussing its methods, implementation, rationale, and potential outcomes.

In Whitespace, control flow is inherently intertwined with the stack-based execution model of the language. Optimisation strategies focus on streamlining the sequence of commands that manipulate the stack, ultimately leading to more efficient execution. Several methods can be employed:

- **Elimination of Redundant Operations:** Identifying and removing unnecessary stack manipulations, such as consecutive pushes and pops, contributes to a leaner execution flow.
- **Conditional Jump Simplification:** Streamlining conditional jumps by minimizing the number of jumps or strategically placing jumps to avoid redundant checks.
- **Loop Optimisation:** Modifying loop structures to minimise stack usage and control operations, thus improving execution speed and resource consumption.

Control flow optimisation in Whitespace necessitates a meticulous analysis of the whitespace patterns corresponding to commands. This analysis guides the identification of opportunities for optimisation. Techniques borrowed from compiler optimisation, such as data flow analysis and control dependence analysis, can be adapted to the Whitespace context. These techniques enable the identification of redundant or unnecessary commands, leading to code simplification.

Implementation of optimised control flow involves rewriting whitespace commands to eliminate redundancy, reorganise control structures, and minimise stack operations. Given the absence of dedicated optimisation tools for analysing Whitespace code, this process

is largely manual, demanding a profound understanding of the language's mechanics and the intricacies of the specific program.

Control flow optimisation in Whitespace serves a dual purpose: enhancing program efficiency and promoting code clarity. A leaner, more streamlined execution path reduces runtime overhead, translating to improved performance for computationally intensive tasks. Furthermore, optimised code is less prone to errors arising from complex control structures, thus augmenting program reliability.

The rationale behind control flow optimisation aligns with the overarching goal of refining Whitespace's unique art form of code expression. While Whitespace emphasises minimalism and obscurity, control flow optimisation seeks to harmonise this peculiarity with improved functionality and efficiency. The optimisation process bridges the gap between the language's enigmatic syntax and the pursuit of elegant, optimised solutions.

The outcomes of control flow optimisation manifest in quantifiable performance gains, as optimised programs execute more swiftly and consume fewer resources. Additionally, the optimisation process unearths hidden patterns within whitespace sequences, deepening the understanding of Whitespace's computational nature.

Future directions in control flow optimisation for Whitespace involve the development of automated tools that aid programmers in identifying optimisation opportunities and implementing changes. These tools would alleviate the manual effort currently required, broadening the applicability of control flow optimisation and making it accessible to a wider audience.

Control flow optimisation within Whitespace stands as a testament to the adaptability of optimisation principles across diverse programming paradigms. The methods employed, the implementation strategies pursued, and the underlying rationale collectively contribute to the enhancement of Whitespace's unique code architecture, underscoring the symbiotic relationship between minimalism and efficiency. As the realm of esoteric languages continues to evolve, optimising control flow in Whitespace remains an active field of exploration, promising further insights into the interplay of form and function in code.

5.2. Code Partitioning for Parallelism

Within the realm of Whitespace, the exploration of parallelism introduces a novel perspective on optimizing program execution. By partitioning Whitespace programs into segments that can execute in parallel, the potential for leveraging modern multi-core systems for improved performance becomes apparent. This section delves into the theoretical underpinnings, practical considerations, and potential outcomes of code partitioning for parallelism within the Whitespace context.

The pursuit of parallelism in Whitespace stems from its stack-based execution model, where operations are inherently ordered and dependent on the state of the stack. However, careful analysis can reveal segments of code that exhibit independence in terms of stack interactions. These independent segments provide the foundation for potential parallel execution, as they can be decoupled without affecting the overall program logic.

The critical task in code partitioning for parallelism involves identifying segments of Whitespace code that can be executed in parallel without introducing data dependencies. This necessitates an intricate understanding of the program's control flow, stack interactions, and the interplay between Whitespace commands. Independent segments may arise from distinct branches of conditional logic, separate loops, or operations that manipulate entirely separate data on the stack.

While the identification of independent segments is foundational, the practical implementation of parallel execution in Whitespace presents challenges. Traditional parallel programming paradigms, such as those found in languages like C or Python, are inapplicable due to Whitespace's unique stack-centric nature. Parallel execution introduces the potential for race conditions, where multiple segments may access and manipulate the shared stack simultaneously.

To circumvent race conditions and ensure consistent execution, mechanisms for synchronisation and communication between parallel segments must be devised. This requires the establishment of well-defined points for interaction, such as specific stack configurations at which parallel segments can synchronise and exchange data. These synchronisation points mitigate conflicts arising from concurrent stack manipulations.

Code partitioning for parallelism in Whitespace holds the promise of enhanced performance on modern multi-core systems. By concurrently executing independent segments, the overall runtime can be significantly reduced. This holds particular relevance for computationally intensive tasks, where the efficiency gains are most pronounced.

For anyone willing, it is possible to develop specialised tools or compilers tailored to this unique context, automating identification of independent segments, introducing synchronisation mechanisms, facilitating generation of parallel-execution Whitespace code. Just like on other endeavours based on esoteric languages, the exploration of theoretical frameworks for analysing the parallelism potential within Whitespace programs could yield insights into optimal partitioning strategies applicable more generally.

Code partitioning for parallelism introduces a novel dimension to the realm of Whitespace programming. By identifying and leveraging independent segments of

code, the potential for harnessing modern multi-core systems to enhance program performance is illuminated. As Whitespace continues to captivate programmers with its minimalist elegance, the pursuit of parallelism within its stack-centric realm opens doors to new vistas of optimisation and efficiency.

5.3. Whitespace Code and Quality Metrics

In the landscape of programming languages, the evaluation of code quality is paramount to fostering maintainability, reliability, and performance. Within the unique context of Whitespace, the establishment of code metrics and quality metrics unveils a new dimension of assessment. This section delves into the rationale behind the application of metrics in the Whitespace paradigm, the metrics themselves, their interpretation, and the implications for code refinement.

The application of code metrics and quality metrics to Whitespace serves a dual purpose: to quantify the inherent complexities of code and to provide objective criteria for evaluating its quality. As Whitespace's minimalist nature conceals intricate computational operations, metrics offer a lens through which to comprehend the program's underlying intricacies. Additionally, metrics serve as a means to guide code refinement, providing a tangible framework for optimizing stack usage, execution efficiency, and logical coherence.

In the context of Whitespace, the development of metrics necessitates an alignment with its stack-centric architecture. Some relevant metrics include:

1. **Stack Depth:** Quantifying the maximum and average stack depth during program execution provides insights into memory consumption and potential bottlenecks.
2. **Command Sequence Length:** Measuring the length of command sequences contributes to understanding the program's complexity and potential for optimisation.
3. **Instruction Density:** Calculating the density of instructions relative to whitespace characters reveals the efficiency of command utilisation.
4. **Branching Complexity:** Assessing the branching complexity through metrics like conditional-to-total command ratio sheds light on the program's logical structure.
5. **Stack Operations:** Quantifying the frequency of stack manipulations, such as pushes and pops, offers insights into computational steps and potential redundancies.

Interpreting metrics within the Whitespace context requires a nuanced understanding of the idiosyncrasies of the language. Metrics do not merely serve as benchmarks; they provide a framework for uncovering patterns, correlations, and opportunities for optimisation. For instance, high stack depth values may indicate memory inefficiencies, whereas excessive branching complexity might signal the need for code restructuring.

The insights garnered from metrics lay the foundation for targeted code refinement in Whitespace. A high stack depth, for instance, may inspire efforts to minimise stack operations through algorithmic optimisations. Excessive branching complexity could encourage the simplification of control structures to enhance code readability and maintainability. Metrics guide the balance between the elegance of Whitespace's minimalist syntax and the pursuit of optimised, well-structured programs.

The integration of metrics into Whitespace programming potentially leads to the development of tooling that automates metric calculation and analysis. Such tools could guide programmers towards code improvements, recommend optimisations, and facilitate the evaluation of codebases. Challenges include adapting traditional metrics to Whitespace's unique attributes and defining thresholds that align with the language's distinctive goals.

Metrics in the Whitespace landscape transcend conventional notions of code evaluation. They illuminate the intricate dance of whitespace commands, offering a quantifiable means to understand, analyze, and enhance program quality. As Whitespace continues to captivate enthusiasts with its enigmatic charm, the application of metrics underscores the symbiotic relationship between quantitative analysis and qualitative code refinement.

6. Concluding Remarks

In this paper, based on recent graduation project [26], we have described our approach to refactoring in Whitespace. The tool is also available for public use under GPL-3 license [25]. To answer our main research question about what refactorings would make sense in Whitespace, we first looked at different refactoring categories. From there we identified seven refactorings that are possible on Whitespace code. To address the question of implementability, we chose three refactorings and implemented them into a tool. Our tool reads Whitespace code, performs refactorings on this code using the generated IR, and transforms the IR back into Whitespace code. This shows that it is possible to create a tool which detects and applies possible refactorings on Whitespace. This work shows that even with minimal circumstances, it is always possible to refactor code even in minimal assembler-like languages. Furthermore, refactoring code is always useful, be that code clarity or a minimal code footprint. We conclude that refactoring Whitespace code is possible and that refactoring Whitespace code improves the readability and usability of such code.

The refactorings proposed in this paper, are a work in progress. Further research and development are needed to fully realise the envisioned functionality. Next to this, more refactorings can be implemented, such as the different conditional refactorings mentioned in § 3.

Furthermore, while some testing has been performed, there could be more tests added. Generating tests to show results that accurately depict the tool is something worth considering to be done. Finally, profiling the tool itself could help identifying useful optimisations to improve its usability, especially regarding possible future extensions For anyone who would like to look at the tool or work on it further, you can find the tool over at GitHub [25].

References

[1] T. Afzal, M. Breternitz, M. Kacher, S. Menyhert, M. Ommerman, W. Su, Motorola PowerPC Migration Tools-Emulation and Translation, in: Digest of Papers on Technologies for the Information Superhighway (COMPCON), IEEE, 1996, pp. 145–150. doi:10.1109/CMPCON.1996.501761.

[2] F. E. Allen, Control Flow Analysis, ACM Sigplan Notices 5 (1970) 1–19. doi:10.1145/390013.808479.

[3] V. Blagodarov, Y. Jaradin, V. Zaytsev, Tool Demo: Raincode Assembler Compiler, in: T. van der Storm, E. Balland, D. Varró (Eds.), Proceedings of the Ninth International Conference on Software Language Engineering (SLE), 2016, pp. 221–225. doi:10.1145/2997364.2997387.

[4] E. Brady, Whitespace, https://web.archive.org/web/20150623025348/http://compsoc.dur.ac.uk/whitespace, 2003.

[5] CensoredUsername, whitespace-rs, https://github.com/CensoredUsername/whitespace-rs, 2016.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design or Existing Code, Addison-Wesley Professional, 1999.

[7] S. Graue, R. Berge, C. Pressey, et al., brainfuck — esolang, https://esolangs.org/wiki/Brainfuck, 2005.

[8] C. Grelck, Single Assignment C (SAC) High Productivity Meets High Performance, in: V. Zsók, Z. Horváth, R. Plasmeijer (Eds.), Revised Selected Papers of the Fourth Central European Functional Programming School (CEFP), Springer, Berlin, Heidelberg, 2012, pp. 207–278. URL: https://doi.org/10.1007/978-3-642-32096-5_5. doi:10.1007/978-3-642-32096-5_5.

[9] A. Gül, V. Zaytsev, Mutative Fuzzing for an Assembler Compiler, in: D. Di Nucci, C. De Roover (Eds.), Post-proceedings of the 18th Belgium-Netherlands Software Evolution Workshop (BENEVOL), volume 2605 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020, pp. 18–24. URL: http://ceur-ws.org/Vol-2605/18.pdf.

[10] A. Hamid, V. Zaytsev, Detecting Refactorable Clones by Slicing Program Dependence Graphs, in: D. Di Ruscio, V. Zaytsev (Eds.), Post-proceedings of the Seventh Seminar in Series on Advanced Techniques and Tools for Software Evolution (SATToSE 2014), volume 1354 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2015, pp. 37–48. URL: http://ceur-ws.org/Vol-1354/paper-04.pdf.

[11] K. R. Irvine, Assembly Language for Intel-Based Computers, 4th ed., Pearson Education, 2002.

[12] P. Jansen, et al., TIOBE Index for April 2023, https://www.tiobe.com/tiobe-index/, 2023.

[13] P. Knaggs, S. Welsh, ARM: Assembly Language Programming, Bournemouth University, School of Design, Engineering, and Computing, 2004. URL: http://www.rigwit.co.uk/ARMBook/ARMBook.pdf.

[14] D. E. Knuth, Literate Programming, The Computer Journal 27 (1984) 97–111. doi:10.1093/comjnl/27.2.97.

[15] C. de Kruif, Using δ-NFGs to Identify and Eliminate Dead Code in C# Programs, Bachelor's thesis, Universiteit Twente, 2022. URL: http://purl.utwente.nl/essays/91890.

[16] R. Lämmel, D. Mosen, A. Varanovich, Method and Tool Support for Classifying Software Languages with Wikipedia, in: M. Erwig, R. F. Paige, E. Van Wyk (Eds.), Proceedings of the Sixth International Conference on Software Language Engineering, volume 8225 of *LNCS*, Springer, 2013, pp. 249–259. doi:10.1007/978-3-319-02654-1_14.

[17] J. Nievergelt, On the Automatic Simplification of Computer Programs, Communications of the ACM 8 (1965) 366–370.

[18] W. F. Opdyke, Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems, in: Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA), 1990.

[19] W. F. Opdyke, Refactoring Object-Oriented Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[20] N. Ramsey, Literate Programming Simplified, IEEE software 11 (1994) 97–105. doi:10.1109/52.311070.

[21] E. D. Rather, D. R. Colburn, C. H. Moore, The Evolution of Forth, in: History of programming languages, Part II, 1996, pp. 625–670. doi:10.1145/234286.1057832.

[22] C. K. Roy, J. R. Cordy, Benchmarks for Software Clone Detection: A Ten-Year Retrospective, in: Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering, IEEE Computer Society, 2018, pp. 26–37. doi:10.1109/SANER.2018.8330194.

[23] H. Schildt, The annotated ANSI C Standard American National Standard for Programming Languages—C: ANSI/ISO 9899-1990, McGraw-Hill, Inc., 1990.

[24] A. Shvets, Refactoring: Clean Your Code, https://refactoring.guru/refactoring, 2022.

[25] R. Witmans, whiteref, https://github.com/rwitmans/whiteref/, 2023.

[26] R. Witmans, Improving Nothingness: Refactorings on Whitespace, Bachelor's thesis, Universiteit Twente, 2023. URL: http://purl.utwente.nl/essays/94374.

[27] D. R. Woods, J. M. Lyon, The INTERCAL Programming Language Reference Manual, https://www.muppetlabs.com/~breadbox/intercal-man/, 1973.

[28] V. Zaytsev, Language Design with Intent, in: D. Batory, J. Gray, V. Kulkarni (Eds.), Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MoDELS), IEEE, 2017, pp. 45–52. doi:10.1109/MODELS.2017.16.

[29] V. Zaytsev, Modelling of Language Syntax and Semantics: The Case of the Assembler Compiler, Proceedings of the 16th European Conference on Modelling Foundations and Applications in the Journal of Object Technology (ECMFA@JOT) 19 (2020). doi:10.5381/jot.2020.19.2.a5.

[30] V. Zaytsev, Software Language Engineers' Worst Nightmare, in: R. Lämmel, L. Tratt, J. De Lara (Eds.), Proceedings of the 13th International Conference on Software Language Engineering (SLE), ACM, 2020, pp. 72–85. doi:10.1145/3426425.3426933.