

Automated Transformations and Alternate Translations: A Case Study

Doni Pracner^{1,*}, Nataša Sukur¹ and Zoran Budimac¹

¹University of Novi Sad, Faculty of Sciences, Department of Mathematics and Informatics, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia

Abstract

An important part of software maintenance is understanding its logic. This is especially true for old systems and new developers, and can be additionally complicated when the source is in low-level structures. In this paper we take a look at one specific process that translates bytecode into an intermediate language (WSL) and then automatically transforms the code into high-level structures. Specifically we evaluate how different translations of the same input programs (while using the same transformations) influence the end results.

Keywords

Automated Maintenance, Hill Climbing, Bytecode, Translations, Program transformations

1. Introduction

Modern systems are mostly integrated with complex software that evolves as it is used. This leads to new complexities that were introduced through many small changes and make the software harder to change or replace. When a need for reengineering emerges, often the first step is to fully understand the software and its functionalities, so that the modifications do not cause any new defects, but rather cause improvement and that they are done in a logical manner [1, 2].

Understanding the original code can be hard, especially if all that is available is low-level versions of the program. In these cases the ideal solution would be to somehow transform these into high-level versions. One type of tools for these types of tasks are decompilers, but those are usually focused on a single type of inputs and often specialise in reversing certain compilers, while in other cases might lead to wrong outputs or even uncompileable code. In our previous work we have shown an open source multistage, adaptable process with tools that are designed to be used independently and guarantee to keep the original semantics [3]. The main stages are 1) translation from low-level code to an intermediate language and 2) transformation of that low-level translation to a high-level program. The current implementation of the transformation

SQAMIA 2023: Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 10–13, 2023, Bratislava, Slovakia


*Corresponding author.

✉ doni.pracner@dmi.uns.ac.rs (D. Pracner); natasa.sukur@dmi.uns.ac.rs (N. Sukur); zjb@dmi.uns.ac.rs (Z. Budimac)

🌐 <https://perun.pmf.uns.ac.rs/pracner/> (D. Pracner)

🆔 0000-0002-3428-3470 (D. Pracner); 0000-0003-4701-9289 (N. Sukur); 0000-0001-5688-6320 (Z. Budimac)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

process is called *Hill Climbing in Fermat* or *HCF* for short. The transformer was made in such a way that it is independent of the translator and makes no assumptions on the inputs, just that it should transform low-level structures into more readable counterparts. It was also made to be fully automatic and require no domain-knowledge from the developer to use it.

In this paper we will look at how different translations of the same input programs can influence the transformation process. Specifically we will use a set of MicroJava bytecode programs and different combinations of switches in our translation tool. While the results are obtained from a single process and samples, some of the conclusions should be applicable to other processes interested in making programs more readable.

The rest of the paper is organised as follows. Section 2 show a brief overview of related work. Section 3 shows the main comparisons of the variants of the translated programs: the metrics of these translations; the metrics of final transformed programs; the executions times; the numbers of transformations selected and tried. In Section 4 we briefly show what are the results that can be achieved with the best switches selected. Finally, conclusions are given in Section 5.

2. Related Work

Understanding existing systems can be a hard task, but is also very crucial for regular maintenance, adding new features, or complete reengineering. Some of the simpler tools types that help in these actions are code visualisers and browsers, these days common in some form in most IDEs [4]. More advanced tools allow customisation and adaptability to the task at hand.

Automating the processes can reduce the work load for developers. Specifically when the inputs are low-level programs the task is much harder for the developers. Tools such as decompilers try to reassemble the original high-level structures. While they can be very good with some inputs, they often rely on knowing the compiler being used and its specific techniques. Working with assembly code is notoriously hard, due to the many optimisation techniques used by compilers [5][6][7]. Even more abstract run-times, with much more data preserved, such as Java bytecode can not be reliably decompiled in general, with some reports assessing even the best decompilers to work at about 80% of the time [8].

In general, program transformations can be considered any actions that take a program as an input and give a different program as the output. Most systems are interested in special types of transformations. For instance, many helper tools have catalogues of refactoring transformations, such as inline procedures, extract procedure, introduce constants, etc. Some systems are designed around the idea of meta-programming, writing systems that are meant to help build systems, such as Rascal [9]. One class of transformations are *semantics-preserving*, which allow for changes in the structure of the code, while retaining all the functionality. These are especially useful for comprehension, since they can be used to find a more understandable version of the program.

2.1. FermaT/WSL

FermaT is a program transformation system, the current implementation of WSL language [10]. It is designed for software maintenance tasks, and among other things contains a large catalogue

of semantics-preserving transformations. It was successfully used to automatically restructure industrial assembly projects (sometimes millions of lines) into maintainable C or COBOL code.

FermaT comes with several metrics built-in. In later sections these will be used to evaluate the input and output programs at various stages: *McCabe's cyclomatic* and *essential* complexity; number of statements; *control flow and data flow*; *size* of the abstract syntax tree generated; and finally *structure*, a custom weighted metric in WSL representing the complexity of structures in the program [11].

3. Comparison of Translation Variants

This section presents the main experiments and results of the paper. The input programs are MicroJava bytecode, and the end results of the *Hill Climbing in Fermat (HCF)* process are high-level versions of these programs in the WSL language. WSL is also used for the intermediate representations and is the one that is used for transformations.

The automated part of HCF is based on a hill climbing algorithm, a pre-selected set of transformations and a fitness function. In our case the fitness function is the structure metric, which gives a weighted sum of the parts of the program and gives a rough idea of “structuredness”. The transformations are evaluated one by one on the input programs. If one of them leads to a better program, i.e. one with a lower metric, it is selected as the base for further transformations. The process continues until no further improvements can be found.

The translation tool *mjc2wsl* can produce several variants of the same input programs. These variations of the tool itself will be described in more detail in Section 3.1. Then the differences between the translated versions for this dataset will be analysed in Section 3.2. After that, Section 3.3 compares the influences of these switches on the transformation process. Section 3.4 shows these influences on execution times and number of transformations tried. Finally, an overview of the results for one of these variants are presented in Section 4.

3.1. Variations in Translations

The tool *mjc2wsl* translates MicroJava bytecode into WSL, which enables the programs to be transformed. The translations are based on the idea of creating a virtual machine integrated with the code. There are local variables that represent the state of the machine, such as the operation stack, procedure stack and the heap. The translation results can be influenced by command line switches. The following three switches are used in this paper, and their combinations produce eight different variants. These translations will be referred to as *alpha-wsl-v8*.

Stack operations `--genPopPush` OR `--genHeadTail` influence how the storage and retrieval of values from the expression and method stack are handled. These structures are internally just lists, and the first option uses the specialised `POP` and `PUSH` commands. The other one uses the more generic `HEAD` and `TAIL` to remove an element and a straight access to the first element for retrieval.

Local variables `--genLocalsAsArray` OR `--genLocalsSeparate`: the first option generates an array of local variables that are accessed by their index; these arrays are then stored the

procedure stack. The second option generates all the local variables as separate names; they are then stored one by one on the procedure stack.

VAR blocks `--genLocalVars` or `--genGlobalVars` influences the temporary variables used for instance in arithmetic operations. The first option will try hard to make all of them in local VAR blocks, while the other one will just use the same global variables for the these needs.

For simpler references in future text, variants of programs will be marked with two letter shorthands, all of which are shown in Table 1. Versions of programs are therefore marked with three two letter codes, in the order given in the table.

Table 1
Abbreviations for the parameters used

<i>Code</i>	<i>Option used</i>
<i>Stack operations</i>	
ht	HEAD and TAIL
pp	POP and PUSH
<i>Temporary variables</i>	
gl	Global
lo	Local VAR blocks
<i>Local variables (procedures)</i>	
ar	Stored together in an array
sp	Separate variables

3.2. Influence of Switches on the Metrics of Translated Programs

In this section we will give an overview of how the previously explained switches influence the metrics. The experiments were run using *mjc2wsl* version 1.0.0, FermaT 18c (internal version number).

The main goal is to compare these variants, so the differences will be expressed as percentage to the best results. This helps to normalise the input program samples, but also the metrics themselves, since the values can vary a lot. For each metric and each program, a best version will be found, and then all of them are compared to it. The formula used is the the difference between the value and the best value for that sample and metric, divided by the best value. These are then averaged per metric and variant group and will be shown in tables further on.

We start with an exception. *McCabe's Essential Complexity* is not influenced by any of combinations of the switches. Any additional IF statements and variants are not counted, as they will be grouped into the same single-entry-single-exit block.

McCabe's Cyclomatic Complexity is affected only by the switch for the temporary variables. The different types of access to the stacks or the different storage of local variables are not affecting this metric. The imlementation of local blocks for temporary variables uses additional jump flags that need to be set and checked later on which results in higher values for this metric.

On average the increase is 27.25%, with a high standard deviation of 23 percentage points. This is due to rare samples that have no or almost no extra jumps in the translation.

Number of statements is most influenced by the first two switches, specifically the combination of *pp-gl* gives the lowest results (Table 2). The final option (*ar* or *sp*) has less influence on the results, mostly just a few percent between them. Which one of these is better is sample dependant. When either of the first two switches is changed (to *ht* or *lo*, respectively) the results of the metric increase about 20%, while if both are changed at the same time it results in almost 40% more statements. The first switch causes this difference since *HEAD/TAIL* need more statements for the same operations than *POP/PUSH*. Local *VAR* blocks also generate extra statements, including the block itself, but also extra jump flags. Finally, the third switch is sample dependant – sometimes extra handling of the individual local variables at the start and end of procedures will be more expensive than the array operations needed for the other versions.

Table 2
alpha-wsl-v8, statements metric

	avg	stdev	max	min
ht-gl-ar	20.23	3.34	26.42	15.22
ht-gl-sp	19.19	2.79	23.77	14.29
ht-lo-ar	37.41	5.33	47.06	28.26
ht-lo-sp	36.37	6.45	47.71	28.21
pp-gl-ar	1.87	2.28	7.14	0.00
pp-gl-sp	0.63	0.94	2.41	0.00
pp-lo-ar	19.05	3.91	24.51	13.04
pp-lo-sp	17.81	4.85	26.51	9.43
percentage difference to lowest results				

Control Flow/Data Flow metric values behave similar to the number of statements when switches are changed, but the values are proportionally larger. The *pp-gl* combination is again showing the lowest values, with *pp-lo* about 20% higher, *ht-gl* about 40% higher, and *ht-lo* about 55%. *CFDF* is more sensitive to the *HEAD/TAIL* variation since it introduces more list operations.

Table 3
alpha-wsl-v8, CFDF metric

	avg	stdev	max	min
ht-gl-ar	40.65	6.71	55.71	30.43
ht-gl-sp	39.45	5.06	45.74	31.03
ht-lo-ar	56.23	8.94	67.14	40.58
ht-lo-sp	55.04	8.56	67.96	39.66
pp-gl-ar	2.47	3.06	8.75	0.00
pp-gl-sp	0.93	1.40	3.59	0.00
pp-lo-ar	18.05	4.26	22.50	10.14
pp-lo-sp	16.51	4.64	23.51	8.62
percentage difference to lowest results				

Size metric (of the abstract syntax tree) always has the best results on all of the samples for the variation *pp-gl-sp*, as shown in Table 4. Next up is *pp-gl-ar*, which is on average 8% worse (± 2.5 percentage points). As was the case with the previous two metrics, there are groups based on the first two switches follow it, but the differences between them are bigger. In short, *pp-lo* is about 30% worse, *ht-gl* 50–60%, *ht-lo-sp* 77% and *ht-lo-ar* 85%. The differences are more pronounced since there are more generated nodes in the AST that earlier metrics would ignore.

Table 4
alpha-wsl-v8, size metric

	avg	stdev	max	min
ht-gl-ar	57.51	8.48	69.60	43.63
ht-gl-sp	50.14	9.15	64.25	35.29
ht-lo-ar	84.36	16.70	109.25	58.33
ht-lo-sp	77.00	17.45	104.74	50.00
pp-gl-ar	7.90	2.52	13.73	3.12
pp-gl-sp	0.00	0.00	0.00	0.00
pp-lo-ar	34.75	9.42	50.15	22.13
pp-lo-sp	26.86	8.98	40.52	14.71

percentage difference to lowest results

Structure metric values are very similar to the *size metric* (usually a few percentage points higher), and just like there all of the samples have their best results with *pp-gl-sp* combination (Table 5). Similarities are due to this metric being a weighted sum of the components of the program, and higher weights are mostly correlated with more nodes in the abstract syntax tree.

Table 5
alpha-wsl-v8, structure metric

	avg	stdev	max	min
ht-gl-ar	57.42	4.75	67.35	51.49
ht-gl-sp	50.43	6.74	63.16	39.81
ht-lo-ar	86.93	11.20	110.21	71.88
ht-lo-sp	79.95	13.38	106.52	61.98
pp-gl-ar	7.60	2.62	13.45	3.17
pp-gl-sp	0.00	0.00	0.00	0.00
pp-lo-ar	35.91	7.02	48.30	24.21
pp-lo-sp	28.31	7.42	41.63	17.37

percentage difference to lowest results

Overall each of the switches shows differences in results for the metrics. The highest difference in values is shown with the *ht/pp* switch, followed by the *gl/lo*. The *sp/ar* also shows differences, but sometimes they are not very significant. The combination of *pp-gl-sp* provides the lowest values for most metrics, usually closely followed or sometimes beaten by the *pp-gl-ar* variant. On the other hand, the metrics will not always correlate with readability, as should be expected. For instance, although the *lo* variants usually have worse numeric results, in most cases it is

easier to understand a program that has defined local variables. More importantly, since these translations are meant to be automatically transformed in the next step, it is more important how these switches will influence that process and the transformed end results.

3.3. Influence of Switches on the Final Results

This section will focus on comparing the influence of the translation switches on the final outputs of the automated transformation process. Part of the process was automated using GNU Parallel [12]. All of these variations of translations came from the same bytecode, so the comparison can focus on the sizes of the final programs and therefore discover which versions of the programs lead to the best end results for which metric. The percentage improvements that the automated transformation made are not as relevant for this, since the start points were different, but will be shown in a later section.

Like in the previous section, normalisation is done and percentage differences to the best result for each sample and metric are used. In a few cases, this can lead to problems with the formula, as the best value of CFDF (*Control Flow and Data Flow*) can reach 0, which would lead to a “division by zero” error. To sidestep this problem the particular example has the values offset by one, which maintains the raw difference and should be an acceptable approximation of the percentage difference. Unlike in the previous section, for all shown metrics there are samples that end up being the same as the best sizes in all variants (visible in the *min* column). It is not the same sample across all of these tables, but shows how there are samples that will be transformed to the same end result no matter the translation.

As in the previous section, McCabe’s metrics are less sensitive to the switches, and won’t be shown in tables. *Cyclomatic Complexity* always has the best results with the *pp-lo* switches. Using *pp-gl* gives slightly worse results (up to 16%) on a few of the samples. The *ht* variants lead to varied results, sometimes same as *pp*, but sometimes twice as large. *Essential Complexity* is mostly unaffected by the switches, but global variants tend to have a few percentage point worse results.

Number of statements and *Control Flow/Data Flow* have similar trends in their results, like in the previous section. On average, the best results are achieved with *pp-gl-ar*, with *pp-lo-sp* being close, and in one case being the best. Tables 6a and 6b illustrate this in more detail. All the *ht* versions are very consistently much worse.

Size (of the AST) and *structure* (weighted sum of elements) metrics are another pair with similar trends. Variant *pp-gl-sp* is best on average, with a few samples where *pp-gl-ar* is better. The *ht* variants are generally significantly worse, as can be seen from Tables 7 and 8.

Overall, the high standard deviation numbers in the given tables indicate that there is a high variation of the differences in results within a single group across the different samples. This was also verified by manual inspections. Part of the explanation is that for every metric there were samples that would be transformed into the same form independent of the switches used. On the other hand there would be a few samples with extremely bad results, shown in the *max* columns in the tables.

Observing the results across all metrics at once, McCabe’s metrics (*Essential* more than *Cyclomatic*) are less influenced by the switches. For other metrics, much clearer conclusions can be made.

Table 6Transformed *alpha-wsl-v8*, a) statements and b)CFDF metric

	avg	stdev	max	min		avg	stdev	max	min
ht-gl-ar	80.19	119.08	360.00	0.00	ht-gl-ar	137.55	219.51	720.00	0.00
ht-gl-sp	134.48	120.73	340.00	0.00	ht-gl-sp	277.05	247.39	700.00	0.00
ht-lo-ar	73.13	117.46	360.00	0.00	ht-lo-ar	133.10	218.03	720.00	0.00
ht-lo-sp	108.17	112.14	340.00	0.00	ht-lo-sp	242.23	254.28	700.00	0.00
pp-gl-ar	15.18	20.68	50.00	0.00	pp-gl-ar	19.38	29.15	100.00	0.00
pp-gl-sp	36.26	48.35	136.36	0.00	pp-gl-sp	39.31	48.75	120.00	0.00
pp-lo-ar	2.78	8.61	33.33	0.00	pp-lo-ar	4.39	14.93	60.00	0.00
pp-lo-sp	10.78	18.03	55.93	0.00	pp-lo-sp	12.61	22.00	65.06	0.00

percentage difference to lowest results

Table 7Transformed *alpha-wsl-v8*, size metric

	avg	stdev	max	min
ht-gl-ar	64.67	84.13	228.57	0.00
ht-gl-sp	115.47	122.66	500.00	0.00
ht-lo-ar	64.64	83.07	228.57	0.00
ht-lo-sp	102.31	124.33	500.00	0.00
pp-gl-ar	13.46	18.24	52.38	0.00
pp-gl-sp	16.64	20.05	52.05	0.00
pp-lo-ar	5.96	14.88	51.43	0.00
pp-lo-sp	3.48	6.68	21.83	0.00

percentage difference to lowest results

Table 8Transformed *alpha-wsl-v8*, structure metric

	avg	stdev	max	min
ht-gl-ar	95.98	137.27	394.12	0.00
ht-gl-sp	155.21	166.62	662.50	0.00
ht-lo-ar	96.05	136.56	394.12	0.00
ht-lo-sp	140.31	169.54	662.50	0.00
pp-gl-ar	16.58	23.64	74.24	0.00
pp-gl-sp	19.44	23.52	62.25	0.00
pp-lo-ar	8.05	20.41	74.24	0.00
pp-lo-sp	3.88	7.30	22.29	0.00

percentage difference to lowest results

Using the *ht* option leads to worse results than *pp*, mostly due to the inability of the current versions of the procedure parameters transformations to recognise HEAD/TAIL operations. This could be solved in the future with expansions to these transformations, or building a new specific transformation that changes these to POP/PUSH.

The differences between local and global temporary variables are more pronounced with the *pp* group, where the advantage is clearly with the *lo* versions. With the *ht* group, these tend to be more equal.

For most metrics (apart from McCabe’s) storing the variables in a single array shows better results than handling them separately. However, best results overall for *size* and *structure* are exactly with *pp-lo-sp* making this narrowly the best candidate for transformations in this analysis.

A lot of the trends are reversed compared to the initial metrics of the translated programs. The global variants usually had better results for the translation, yet the local ones end up with better final transformed results. Similarly the advantage that *sp* had over *ar* in the translated ones is mostly overturned in the transformation process.

3.4. Influence of Switches on Execution Times

Another aspect to be considered when choosing the translation variant to work with is the length and complexity of the process. Table 9 shows the execution times for the variants, as well as the number of total transformations tried, and the number of transformations that were selected. The numbers are taken from experiments run on a Intel Xeon E5-2420, clocked at 2.2 GHz, with times taken as totals for transforming all the samples in a single run.

When comparing switches, there is an advantage on the side of *pp* against *ht*, and *lo* against *gl*, both being several times faster. On the other hand *sp* and *ar* vary a lot combined with the first two: *ht-gl-ar* it is twice as fast as *ht-gl-sp*; with the *lo* versions it is about 20% slower, while *pp-gl-ar* is about 60% slower. The number of transformations tried rises with the times, although it is not strictly correlated. For example, the slowest version was 162 times longer than the fastest, while trying “only” 22 times more transformations. This is due to some transformations being slower than others, and on more successful variants they get applied rarer and to shorter pieces of code. The number of *selected* transformations is much more related to the variant at hand, than to the times, with a big difference between the *ht* and *pp* versions.

Table 9
Execution times and counts for *alpha-wsl-v8* transformations

Variant	Time	Tried	Selected
ht-gl-ar	642m	17.394.246	2401
ht-gl-sp	1297m	22.359.953	2293
ht-lo-ar	29m	2.292.549	2405
ht-lo-sp	23m	1.798.537	2231
pp-gl-ar	168m	6.881.829	1669
pp-gl-sp	102m	7.233.491	1539
pp-lo-ar	10m	1.113.809	1851
pp-lo-sp	8m	1.021.643	1823

There is a correlation that the variants with better metrics took significantly less time to execute. This is somewhat inherent to the hill climbing process used – there are more transformations to try on longer programs, therefore when the transformations are successful

the process finds the local minimums faster.

4. Transformation Improvements With Best Switches

The analysis so far has shown that the *pp-lo-sp* variant is the best candidate for transformations (with *pp-lo-ar* being a close second). The previous section was primarily showing results as comparisons between variants. In this section, to give a better idea of what can be expected from the transformation process, the end results of the transformations for this variant will be discussed in comparison to the translated code. Table 10 shows the average values of the original code, the transformed code, and the average percent of improvement per program in the *alpha-wsl-pp-lo-sp* sample set.

Table 10
alpha-wsl-pp-lo-sp transformation metrics

Metric	WSL	WSL-t	% diff
McCabe Cyclo	8.62 ± 5	3.19 ± 3	66.38 ± 11
McCabe Essential	2.88 ± 1	1.06 ± 0	57.69 ± 13
Statements	166.44 ± 144	16.56 ± 23	91.31 ± 4
CFDF	239.69 ± 207	21.94 ± 35	93.62 ± 5
Size	782.06 ± 649	112.88 ± 130	87.88 ± 5
Structure	2367.81 ± 2070	243.88 ± 292	91.25 ± 3

All of the metrics tested show significant improvements, in line with the expectations of the experiments. Most metrics show improvements in the range of 87 – 94%. The two exceptions are McCabe metrics. The least improved is McCabe Essential complexity at 57.69%. Most translated programs don't have very high values for this metric, and often the final transformed program will lower it to 1. The samples have a lot of variety in complexity and size, which is reflected in high standard deviations on the “raw” values of the metrics. On the other hand the percentage results have low standard deviations, showing that the process gives stable results in terms of the improvements for a single program.

5. Conclusion

This paper presented a case study of how changing the translation of programs while using the same transformation system can influence the final results. Specifically this is observed on an automated process that transforms low-level MicroJava bytecode into high-level structures in WSL. The used tool *mjc2wsl* provides several switches which were combined to get different versions of same input programs.

The comparisons were done at several stages of the process. First, the translations themselves were compared using several metrics (Section 3.1). Second, the final transformed versions were also compared using those metrics (Section 3.3). Third, the execution times and the numbers of tried and selected transformations were compared (Section 3.4).

The results show that switches influence all of these. When considering metrics, McCabe's Cyclomatic and Essential Complexity showed less change than others. Metrics of the translated versions were proven to be mostly irrelevant since the metrics on end results reversed many of the trends. On the other hand there was good correlation between execution times, and transformation numbers with the best results metrics-wise.

The main contribution of this paper is the recommendation of a set of switches for the tools at hand, empowered by empiric data and the analysis for these results. This analysis can be used for other tools that could be used for similar transformation processes, or in designing new tools.

Further experiments about these phenomena should be done to confirm the results for a more general use case. Experiments could use different translation tools and different languages for these comparisons. Different transformation tools should also be used. The sample set used could also be expanded or replaced with another.

5.1. Reproducibility

The experiments presented in this paper used GPL licensed open source software and the dataset is also publicly available. FermaT version 18c is available from our git repository¹. The translation tool is available from the repository: *mjc2wsl*, version 1.1.0². The used HCF script is also available as part of the *mjc2wsl* repository in the "src-wsl" folder. The data sets is available in the repository of the translator, in the "samples" folders.

All of the input files, translations, transformations and logs are available as an archive on the project web site³.

Acknowledgments

The authors from the University of Novi Sad gratefully acknowledge the financial support of the Ministry of Science, Technological Development and Innovation of the Republic of Serbia (Grant No. 451-03-47/2023-01/200125)

References

- [1] H. Yang, M. Ward, *Successful Evolution of Software Systems*, Artech House, Norwood, MA, USA, 2003.
- [2] P. Tripathy, K. Naik, *Software Evolution and Maintenance*, John Wiley & Sons, 2014.
- [3] D. Pracner, Z. Budimac, Enabling code transformations with FermaT on simplified bytecode, *Journal of Software: Evolution and Process* 29 (2017) e1857–n/a. URL: <http://dx.doi.org/10.1002/smr.1857>. doi:10.1002/smr.1857.

¹<https://gitlab.com/clatu/fermat3/-/tags/fermat-18-c>

²<https://github.com/quinnuendo/mjc2wsl/releases/tag/v1.1.0>

³<https://perun.pmf.uns.ac.rs/pracner/transformations/>

- [4] J. Siegmund, Program comprehension: Past, present, and future, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 5, 2016, pp. 13–20. doi:10.1109/SANER.2016.35.
- [5] C. Cifuentes, D. Simon, Procedure abstraction recovery from binary code, in: Proceedings of the Conference on Software Maintenance and Reengineering, CSMR '00, IEEE Computer Society, Washington, DC, USA, 2000, pp. 55–64. URL: <http://dl.acm.org/citation.cfm?id=518900.795261>.
- [6] E. J. Schwartz, J. Lee, M. Woo, D. Brumley, Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring, in: Proceedings of the USENIX Security '13 Symposium, 2013, pp. 353–368.
- [7] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, M. Smith, No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations, in: Network and Distributed System Security (NDSS) Symposium 2015, The Internet Society, 2015.
- [8] N. Harrand, C. Soto-Valero, M. Monperrus, B. Baudry, Java decompiler diversity and its application to meta-decompilation, *Journal of Systems and Software* 168 (2020) 110645.
- [9] M. Hills, P. Klint, Php air: Analyzing php systems with rascal, in: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on, IEEE, 2014, pp. 454–457.
- [10] M. Ward, Assembler restructuring in FermaT, in: SCAM, IEEE, 2013, pp. 147–156. doi:<http://dx.doi.org/10.1109/SCAM.2013.6648196>.
- [11] M. Ward, T. Hardcastle, S. Natelberg, WSL Programmer's Reference Manual, 2008.
- [12] O. Tange, Gnu parallel - the command-line power tool, *login: The USENIX Magazine* 36 (2011) 42–47. URL: <http://www.gnu.org/s/parallel>. doi:<http://dx.doi.org/10.5281/zenodo.16303>.