University of Novi Sad
Faculty of Sciences
Department of Mathematics and Informatics

Doni Pracner

Translation and Transformation of Low Level Programs

– Doctoral dissertation –

Prevođenje i transformisanje programa niskog nivoa

– Doktorska disertacija –

Novi Sad, 2018

Preface

Software is ubiquitous, and this is not something that will change, with more and more portable and embedded devices getting into every part of our lives. However, since it has no physical properties like the hardware it runs on, there is less and less software that is being developed from scratch. Similarly, program code that was correct when written will not suddenly decay and produce wrong results. What can change is the environment. The hardware itself can change too much for the code to run, communication (or other) standards can change, new features might be needed, or were not anticipated fully. A huge portion of work is spent on understanding old programs and their maintenance or reengineering for new purposes. For instance, one of the greatest world wide software problems was the year 2000, known as *Y2K*. When designing time storage in the early days of software development it was crucial to be conservative with memory. Therefore, often only the last two digits of the year were used. This meant that for a lot of programs after year 1999, the next one was 1900. As the problem itself was noticed early enough, the negative impact when the year itself arrived was relatively minimal, yet the costs of fixing it in the years prior were substantial. Similarly, hardware itself changes. PC compatible computers started with 16 bits as the basic processor word, and over the years increased to 64. This meant that older software was often not able to use the full power of new hardware, or, even worse, would not work at all.

This PhD thesis presents an approach for working with low-level source code, which can be used for understanding the logic of the code, automated reengineering and basic decompilation. A crucial part of this work is the ability to automate the transformation into higher-level structures, thus significantly reducing the time spent on restructuring, and enabling users with little or no domain knowledge needed to do this work effectively.

The thesis is split into three primary parts: introduction, translation, and transformation, followed by a general conclusion and a fourth part which contains the appendices. The parts themselves are organised in chapters as follows:

Part I presents various topics related to the thesis. Chapter 1 gives a brief introduction, as well as an explicit list of contributions of this work. In Chapter 2 some main ideas and tools for software maintenance and evolution are presented, and a brief overview of software metrics. This is followed in Chapter 3 by a more detailed presentation of the FermaT tool and WSL language that were heavily used in this thesis. Chapter 4 deals with the basics of assembly language, the x86 processor and the MASM/TASM dialect. Chapter 5 presents the ideas of using bytecode and discusses in some detail the MicroJava language, the MicroJava Virtual Machine and the related bytecode.

Part II deals with the translation low-level languages into WSL the two tools made for this. *Asm2wsl*, used for x86 assembly, is presented in Chapter 6. *Mjc2wsl*, used for MicroJava compiled bytecode is then presented in Chapter 7.

Part III deals with the transformations of code done in FermaT. Chapter 8 starts with manual application of transformations and an example of such a process. This is then extended with some ideas on how to automate the process including a hill climbing algorithm. Following are experiments with the hill climbing approach on samples translated from assembly and MicroJava bytecode.

Finally, Chapter 9 presents an overview of the work accomplished in this thesis and the conclusions that were made. It is further compared to some other related approaches, and then ideas for future work are given.

Part IV contains the appendices, including a catalogue of FermaT transformations, specifications for MicroJava, some additional data, and source code.

Acknowledgements

First and foremost I need to thank my mentor, Dr. Zoran Budimac for the time we spent working on this thesis, over the many years. Among other things, he taught me valuable lessons on writing in a clearer way, and being more accessible

to the reader. I would also like to thank the members of the committee, Dr. Miloš Radovanović, Dr. Gordana Rakić and Dr. Zaharije Radivojević, for their time spent on reading and commenting this thesis.

I thank the Faculty of Sciences, University of Novi Sad for some of the facilities that were used during the development of this thesis. This work was also partially funded by the Serbian Ministry of Education, Science and Technological Development.

Special thanks need to be addressed at Dr. Martin Ward, the creator of FermaT and WSL. To start off, these tools are at the core of this work. More importantly, I have to thank him for the many discussions and explanations of the internals of the system that allowed me to be more efficient in my work. Some of his earlier work was an inspiration for many of the achievements displayed here, and others were made in direct collaboration with him. I also thank Dr. Hussein Zedan for discussions on some of these topics.

I am grateful to Dr. Miloš Radovanović, with whom I made some of my initial steps into research in general, and have learned a lot from him over the years. I thank Dr. Gordana Rakić for sending potentially related papers and books my way and encouraging me to progress further. I would also like to thank Saša Tošić and Saša Pešić for sharing their experiences in teaching assembly, which was valuable in the development of *asm2wsl*. I have probably missed at least some people who influenced this work, and for that I am sorry in advance.

I am grateful to many of my colleagues for a pleasant work environment and many discussions over the years, sometimes even related to this work. I am additionally grateful to my friends and brother for many good moments over the years, sometimes a (necessary) distraction from work, sometimes pushing me to make further progress.

Finally, the greatest gratitude has to be directed to my parents, who raised me and supported me through all of these years of endless education. Their influence on who I am and where I am now is immeasurable.

Novi Sad, 2018 Doni Pracner

Rezime

Sa razvojem i pojeftinjenjem računara dolazi do njihove sve veće rasprostranjenosti. Danas su oni integrisani u skoro svaki aspekt naših života. I na najmanjim današnjim računarima se nekad pokreću veoma kompleksni programi. Zbog ove kompleksnosti, a i zbog toga što softver nema fizička svojstva i lako se kopira, danas se sve ređe razvija potpuno nov i nezavistan softver. Samim tim postoji sve veća potreba da se stari softver menja i unapređuje kako bi se mogao integrisati u nova okruženja. Veoma važan deo svakog takvog procesa je razumevanje kako originalni softver radi. Često su dostupne samo konačne, izvršne verzije programa, dok originalni izvorni kôd visokog nivoa možda nije dostupan, možda je zastareo, a možda nikad nije ni postojao (na primer kod programa pisanih u mašinskom jeziku). Slično važi i za dokumentaciju programa.

U okviru ove teze se predstavlja pristup radu sa kôdom niskog nivoa koji omogućava automatsko restrukturiranje i podizanje na više nivoe. Samim tim postaje mnogo lakše razumeti logiku programa, što smanjuje vreme razvoja. Krajnji rezultat procesa je u najboljem slučaju jasan kôd visokog nivoa. U najgorim slučajevima je urađen bar deo procesa reinženjeringa.

Proces predstavljen u tezi se dobrim delom oslanja na sistem *FermaT* i jezik *WSL (eng. Wide Spectrum Language – jezik širokog spektra)*. U ovaj sistem je ugrađen veliki broj transformacija programa koje očuvavaju semantiku i u skladu sa tim su veoma primenljive na restrukturiranje kôda. Sam sistem je već uspešno bio primenjivan u restrukturiranju industrijskih asemblerskih biblioteka u održive C i COBOL programe. Proces je dizajniran tako da bude fleksibilan i sastoji se od više nezavisnih faza. Samim tim je lako menjati proces po potrebi, ali i upotrebiti razvijene alate u drugim procesima. Tipično se mogu razlikovati dva glavna koraka. Prvi je prevođenje u WSL, a drugi su transformacije u samom WSL-u. Za potrebe prevođenja su razvijena dva alata, jedan koji radi sa podskupom x86 asemblera i drugi koji radi sa MikroJava bajtkôdom. Rezultat prevođenja je program niskog nivoa u WSL jeziku.

Transformacije se mogu primenjivati na različite načine, ali primarni cilj ovog istraživanja je bila potpuna automatizacija odabira, tako da i korisnici bez iskustva u radu sa sistemom mogu efikasno da primene ovaj proces za svoje potrebe. Sa druge strane zbog fleksibilnosti procesa, iskusni korisnici mogu lako da ga prošire ili da ga integrišu u neki drugi već postojeći proces. Automatizacija je postignuta *pretraživanjem usponom (eng. hill climbing)*. Algoritam se sastoji od primene transformacija iz prethodno odabranog skupa na program i provere da li je rezultat bolji na osnovu neke *funkcije pogodnosti (eng. fitness function)*. Ako je novi program bolji, uzima se za novu osnovu za primenu daljih transformacija. Inače se odbacuje i nastavlja sa primenom transformacija na trenutni program. Proces se nastavlja dokle god je moguće naći bolji program. Algoritam pretraživanja uspinjanjem do sada nije bio uspešno primenjivan na ovaj tip problema, bar koliko je autoru poznato.

Same transformacije u procesu ne pretpostavljaju ništa u vezi ulaznih programa, odnosno mogu se primenjivati na bilo kakve ulaze i ne zavise od korišćenih prevodilaca. Kvalitet konačnih rezultata će varirati u zavisnosti od tipa ulaza, pošto su primarno odabrane transformacije koje dižu nivo apstrakcije programa. Za neke specifične ulaze ovo može značiti da će promene biti minimalne, ali čak i onda će programi zadržati svoju kompletnu semantiku, odnosno neće dovesti do „kvarenja" programa.

Eksperimenti vršeni na nekoliko tipova ulaznih programa niskog nivoa su pokazali da rezultati mogu biti izuzetni. Za funkciju pogodnosti je korišćena ugrađena metrika koja daje „težinu" struktura u programu. Napravljeni alati generišu različite tipove programa niskog nivoa, a imaju i parametre kojima se može dodatno uticati na način prevođenja različitih struktura. Na sve ove ulaze je primenjen isti automatski transformator. Različiti prevodi istih programa su transformisani sa različitim procentima unapređenja metrika. Istovremeno je važilo da se do boljih rezultata skoro uvek dolazilo u značajno kraćem vremenu. Ovo je posledica samog algoritma koji primenjuje sve transformacije na sve delove programa. Ukoliko su one ranije uspešne, dalji proces se ubrzava jer ima manje mesta za primene. Na osnovu analiza različitih ulaza se dodatno može unaprediti sam proces, ali i pomoći korisniku da odabere ulaze koji su najadekvatniji za ovaj proces.

Kod ulaza za koje je postojao originalni izvorni kôd, krajnje metrike najboljih varijanti prevedenih i transformisanih programa su bile na sličnom nivou. Neki primeri su bolji od originala, dok su drugi bili nešto kompleksniji. Rezultati su uvek pokazivali značajna unapređenja u odnosu na originalni kôd niskog nivoa.

Abstract

With the development and greater availability of computers they become integrated into almost every aspect of our lives. Even the smallest of computers can often run very complex software. Due to this complexity, but also because software has no physical properties and can be easily copied, software is rarely developed from scratch and without external dependencies. This leads to a greater need to reuse and improve old software and integrate it into new environments. An important part of every maintenance process is to understand the logic of the original software. Quite often the only available artefacts are the executable versions of the program, while the original high-level source code is not available, out of date, or has never existed in the first place (for instance with programs written in machine language). Similar can be said for the program documentation.

This thesis presents an approach for working with low-level source code that enables automatic restructuring and raising the abstraction level of programs. This makes it easier to understand the logic of a program, which in turn reduces development time. The end result of the process is, in a best case scenario, a high-level version of the program. At worst, only a part of the reengineering is done automatically.

The presented process relies on the *FermaT* transformation system and the language *WSL (Wide Spectrum Language)*. This system has a great number of built-in semantics-preserving program transformations and as such is very applicable for code restructuring. The system itself was already used on industrial legacy assembly programs and their conversion into maintainable C and COBOL code. The process described in this thesis was designed to be flexible and consists of several independent tools. This makes the process easy to adapt as needed, while at the same time the developed tools can be used for other processes. There are usually two basic steps: translation to WSL; and transformation of the translated WSL. Two tools were developed for translation: one that works with a subset of

x86 assembly, and another that works with MicroJava bytecode. The result of the translation is a low-level program in WSL.

The transformations themselves can be applied in different ways. The primary goal of this thesis was to fully automate the selection of the transformations. This enables users with no domain knowledge to efficiently use this process as needed. At the same time, the flexibility of the process enables experienced users to adapt it to their needs or integrate it into other processes. The automation was achieved with a *hill climbing* algorithm. Transformations from a predefined set are applied to the program at hand, and then a *fitness function* is used to compare the new program with the original. If it is an improvement, it is used for the next round of transformations. Otherwise the new program is disposed of and the transformations are further applied to the current program. The process continues while there are improvements. The hill climbing algorithm was not successfully used before in this type of application, at least as far as the author is aware of.

The transformations used make no assumptions about the input programs, i.e., they can be applied to any type of input and are not dependant on the translators used. The end results can vary in quality for different types of programs, since the transformations were chosen specifically to raise the level of abstraction. This means that for some specific inputs the improvements might be minimal, but even then the programs will keep their semantics, in other words, the program will not be "ruined".

Experiments that were run on several types of input programs showed that the results can be excellent. The fitness function used was a built-in metric that gives the "weight" of structures in a program. The developed tools generate different types of low-level programs, and also feature parameters to additionally change the way some structures are translated. All of these variations were handled with the same automated transformation program. The different translations of the same programs had different percentages of metrics improvements. Better results were almost always achieved in significantly less time. This is a direct consequence of the algorithm that tries all the transformations on all parts of a program. As soon as there is success, the process is sped up since there are fewer application targets. The analysis of the behaviour of different inputs can be used to improve the process itself, but also as a guide for users to choose the most appropriate types of input for the process.

On input samples that had original high-level source code, the end result metrics of the translated and transformed programs were comparable. On some samples the result was even better than the originals, on some others they were somewhat more complex. When comparing with low-level original source code, the end results were always significantly improved.

Contents

List of Figures

List of Tables

Shorthands and Abbreviations Used

asm2wsl assembly to WSL translation tool;

AST abstract syntax tree;

FME FermaT Maintenance Environment, an open source graphical user interface for FermaT and WSL;

GCD Greatest Common Divisor, the algorithm for this is used in some examples;

JVM Java Virtual Machine;

MASM Microsoft Macro Assembler;

MJ, MJVM MicroJava and MicroJava Virtual Machine;

mjc2wsl MicroJava Compiled bytecode to WSL translation tool;

TASM Turbo Assembler, by Borland Inc;

WSL *Wide Spectrum Language*, with FermaT transformation system holding an implementation of it, see Chapter 3.

Sample sets Assembly programs are referred to as *asm-a*. The set of MicroJava programs used is named *alpha-mj*, and the collection of translated versions with eight variations of translations is then called *alpha-wsl-v8*, while the automatically transformed samples are referred to as *alpha-wsl-tr*.

See also Table 7.1 (page 74) for abbreviations used for different switches of the translation tool and the resulting variations of the programs.

Part I

Preliminaries

Chapter 1

Introduction

Software in its executable form is typically stored in some low-level form. One type is machine code specific for the hardware it runs on and executable directly on it. The other is some sort of code for a virtual machine or a low-level interpreter, such as a Java class compiled into bytecode, p-code versions of Pascal programs, or .NET code. The executable form is obtained either from a high-level language compiler, or it was written directly. In a general case, an interpreter that executes the source code directly, without previous compilation to a low-level version can be made for any high-level language. The main advantages of a compiled executable are the reduction in size and an increase in execution speed. The problem with this is that if changes are needed, the appropriate high-level code might not be available, either because it was lost or because there were too many low-level changes at the target machine. The consequence is that the whole process of maintenance becomes a lot harder, since it is harder to understand the logic of a low-level program.

This thesis presents an approach mainly aimed at raising the low-level code to high-level structures and therefore making the understanding and maintenance significantly easier. Parts of it rely on the FermaT transformation system, that is built around the WSL language [Ward, 2004], which supports formal, semantics preserving transformations, and has been successfully used before in industrial applications of software evolution of legacy code. More details about the language and how to work with it are given in Chapter 3. The process in this work is illustrated in Figure 1.1 and consists of two main steps:

- translation of the original low-level code to the WSL language. In this thesis tools for a subset of x86 assembly and MicroJava bytecode are presented;

- transformation of the translated WSL code. The main approach in this work is an automated transformation script, but it can easily be combined with additional manual transformations either before or after.

The process itself is made to be adaptable and the tools in it were designed to solve the subproblems independently of other parts. Therefore it is easy to replace some of them, add new ones to the process, or even reuse them in other processes.

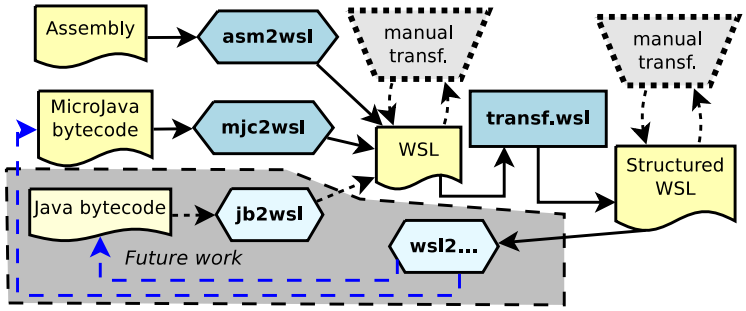


Figure 1.1: Work flow for processing low-level code

1.1 Thesis Contributions

The main contribution of this thesis is the fully automated process that is capable of restructuring low-level code into high-level structures. The goal is to make the programs easier to understand and maintain. As such it can be used in a variety of maintenance applications.

The process is flexible and adaptable. Components from it can be used for other purposes, or easily replaced for other needs. The thesis demonstrates and evaluates the process on two types of low-level inputs. The assembly translation tool is very limited, with its input programs, as will be explained in later chapters, but does provide a different type of input programs, and could be an interesting test bed for future expansions. The MicroJava translation tool, on the other hand, is capable of handling practically any correct input of bytecode and can further provide different translations of the same inputs. Variations of inputs were used to get a deeper understanding of the process and the types of programs it can handle better and faster.

The transformation process uses a *hill climbing* algorithm to automatically choose the transformations. They are tested one by one on different parts of the program, and a *fitness function* is used to evaluate if the result is more desirable and should be used for the continuation of the process. The automation makes it suitable for a user with no domain expertise. At the same time, an experienced user can adapt the process or integrate it into their own workflow.

The quality of the end result is highly dependent on the used fitness function. According to the "no free lunch" theorem for search and optimisation a universally best algorithm that would lead to the best results for any input does not exist [Wolpert and Macready, 1997]. However, for the specific input types used, made from low-level languages with a lot of labels and jumps, the selected *structure* metric is, on average, a good choice. The automated selection can be used on a variety of input types with no changes to this part of the process. In the cases where an input program is "bad" for the algorithm it will not be restructured as much as it could be, but its semantics will not be compromised, nor can the process be stuck in an endless loop.

The *hill climbing* algorithm itself was not successfully used before for this type of application, at least as far as we are aware of. It was used with a similar goal, but with a significantly different starting point and route to the solution. There it showed to be inferior to generic algorithms, but the approach itself was much more in line with general genetic algorithm approaches [Fatiregun, Harman, and Hierons, 2004]. It was also briefly discussed as a potential solution for bug fixing in [Arcuri and Yao, 2008], but the authors were sceptical about it and chose to use genetic algorithms instead.

The following tools and components were developed in this thesis:

- Open source tool *asm2wsl* that translates a subset of x86 assembly to WSL [Pracner and Budimac, 2011b][Pracner and Budimac, 2011a] (Chapter 6).
- Open source tool *mjc2wsl* which translates MicroJava bytecode into WSL [Pracner and Budimac, 2013][Pracner and Budimac, 2017b] (Chapter 7).
- An automated transformation selection program called `hill_climbing` developed in cooperation with Dr. Martin Ward (Section 8.2), available under the GNU Public Licence v3 or later, source code can be found in Appendix D.1.
- Analysis of the process with a series of experiments using the hill climbing approach done on samples translated from assembly (Section 8.3) and MicroJava bytecode (Section 8.4), including a deeper analysis of the variations of MicroJava samples and their influence on the process. An overview of the

input types and their characteristics and the possibilities of working with other inputs is given in Section 8.5.

- The following improvements to FermaT were made in cooperation with its author Dr. Martin Ward:
  - Item simplification was expanded with operations to convert numeric codes to actual characters, and some format strings;
  - new transformations were added to the system: `Stack_To_Var`, `Stack_To_Par`, `All_Proc_Stacks_To_Pars`, `Stack_To_Return`, `Array_To_Vars`, `Proc_To_Funct`, `Align_Nested_Vars` ;
  - some improvements and expansions of the documentation of FermaT;
  - improvements and bug fixes in FermaT in some existing transformations and in general;
  - a new "quiet" mode for WSL scripts was added;
  - new ways of passing command line parameters to WSL programs.

Relevant papers published during the development of this thesis

- Doni Pracner and Zoran Budimac [2017b]. "Enabling code transformations with FermaT on simplified bytecode". In: *Journal of Software: Evolution and Process* 29.5, e1857–n/a. ISSN: 2047-7481. DOI: 10.1002/smr.1857. URL: http://dx.doi.org/10.1002/smr.1857

- Doni Pracner and Zoran Budimac [2017a]. "A Practical Tutorial for FermaT and WSL Transformations". In: *Proceedings of the 6th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications*. Ed. by Zoran Budimac. Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Serbia, 11:01–11:08. URL: http://ceur-ws.org/Vol-1938/

- Doni Pracner and Zoran Budimac [2013]. "Transforming Low-level Languages Using FermaT and WSL". in: *Proceedings of the 2nd Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications*. Ed. by Zoran Budimac. Vol. 1053. CEUR-WS.org, pp. 71–78. URL: http://ceur-ws.org/Vol-1053/

- Doni Pracner and Zoran Budimac [2011a]. "Restructuring Assembly Code Using Formal Transformations". In: *Proc. of Symposium on Computer Languages, Implementations and Tools (SCLIT 2011) held within International Conference on Numerical Analysis and Applied Mathematics ICNAAM 2011*. Ed. by Theodore E. Simos. Vol. 1389. AIP proceedings. Kassandra, Halkidiki, Greece, pp. 845–848. ISBN: 978-0-7354-0956-9

- Doni Pracner and Zoran Budimac [2011b]. "Understanding Old Assembly Code Using WSL". in: *Proc. of the 14th International Multiconference on Information Society (IS 2011)*. Ed. by Marko Bohanec et al. Vol. A. Ljubljana, Slovenia: Institute "Jožef Stefan", Ljubljana, pp. 171–174. ISBN: 978-961-264-035-4. URL: `http://is.ijs.si`

Chapter 2

Software Maintenance, Evolution and Metrics

This chapter covers some basic software evolution and maintenance concepts, and a brief overview of software metrics as they will be used later on.

Software engineering, as most fields related to computer science, is relatively new, yet developing at a rapid rate. One could argue that it started in the 1950s with the advancements of computers and their needs for complex software. The earliest electronic machines had their behaviour hardwired and thus inseparable from hardware. The first programmable machines were therefore under a strong influence of classical electronic engineering. With the relative inexpensiveness of changing computer code, the expensive formalism of many methods were mostly abandoned for more ad-hoc approaches. As early as the 1968 "NATO Software Engineering Conference" [Naur and Randell, 1969] there were already talks of a *software crisis*, that computing power was increasing fast and that software development was too complex to be handled in the required time. David Parnas raised some opposition to the term *crisis*, since it should refer to something short term, and that something like *chronic disease* would be more appropriate [Parnas, 1994]. Edsger Dijkstra, in his 1972 Turing Award acceptance speech [Dijkstra, 1972], even claimed that this complexity is inherent to more powerful machines and that only theoretical computing can be free of these problems. In line with this, *Wirth's law* states that software becomes slower at a more rapid rate than the hardware gets faster. It was named after Niklaus Wirth and his discussion of the topic [Wirth, 1995], but it is also know by other names, such as Page's, May's or Gates' law. Similar conclusions

were made by Frederick Brooks in his *No Silver Bullet* paper, in which he separates accidental complexity (something that can be reduced), from essential complexity (inherent to the problem) and can not be implemented in a simpler way than the problem itself [Brooks, 1995].

Boehm [2006] gives an outline of the history of software engineering through the decades of the second half of the 20th century, the major trends, and what was positive and negative about them. As the author says, not knowing the history makes you repeat it, but the successes should be repeated, which is often not the case.

Another way in which software engineering is very different to other types of engineering that have clear physical limitations and costs, is that a huge part of earlier developed software can still be used and it does not degrade on its own. On the other hand most systems need updates to improve their usefulness or even to remain useful. David Parnas described *software aging* as having two primary causes: *lack of movement*, that is not adapting to the changes of the environment (hardware, software, requirements); and *ignorant surgery*, changes made without proper considerations to the system. These are not mutually exclusive and can lead to even faster deterioration when combined [Parnas, 1994]. A proportionately huge part of work hours is spent on maintenance of existing software, introducing new features and fixing problems, compared to the original development time. Different estimates go from 2/3 of time to even 90%, but this can all depend on the characteristics of the system at hand, and on the details of how the maintenance phase is defined [Wagner, 2014].

2.1 Software Maintanance and Evolution

Software *maintenance* and *evolution* are related, but different terms. Maintenance is usually applied to changes in the software that prevent its failure, or in other words "fixing bugs". Evolution on the other hand is a process of developing a new piece of software from an existing one [Tripathy and Naik, 2014]. Another definition of evolution is "the process of conducting continuous software reengineering", with three stages in the life cycle of software that repeat infinitely: forward engineering, reverse engineering and functional restructuring [Yang and Ward, 2003].

On the other hand, even though these terms are different, the basic tasks present similar or sometimes identical problems, such as understanding and reengineering the system. Most of the tools that will be discussed here and the main points of the work in this thesis can be applied both to maintenance and to evolution.

Legacy systems are a somewhat special category when considering maintenance. Definitions vary a bit, but in general these are pieces of software that are necessary for the functioning of an organisation, typically developed in some obsolete language and/or method, potentially running on obsolete hardware, hard to update due to inconsistent documentation, with original developers potentially unavailable either from moving to a different company or retiring [Bennett, 1995; Tripathy and Naik, 2014; Wagner, 2014]. For these systems one of the highest priorities is to understand their original logic to be able to properly redevelop and modernise them.

2.1.1 Lehmans Laws of Software Evolution

First well-known attempt to observe and understand the behaviour of continuously developing software systems was made by Belady and Lehman while working at IBM, based on empirical data from the OS/360 operating system. The findings had little initial impact on the processes inside the company, but these would form the basis for the field of software evolution [Belady and Lehman, 1976]. The laws themselves were subject to several updates, leading to an increase from three to eight laws in total [Lehman, 1997; Lehman and Belady, 1985].

From the point of view of evolution, all software systems were divided into three types with the *SPE* classification:

S-type – *specified* programs, whose behaviour can be formally specified and therefore the implementations themselves can be formally verified.

P-type – *problem* programs which solve something that can be clearly defined, yet the solution itself might be imprecise, with a certain level of precision, either because there are no known exact solutions, or they are to costly to be performed in full.

E-type – *evolving* and are *embedded* into the real world. These types of programs solve problems that cannot even be clearly defined and are often implementations of human or society activities. They are under the influence of their environment and, at the same time, can influence the environment.

The main observation is that the defined laws apply to any complex E-type software system, independent of the actual processes used, the targets of the system, etc. In some ways this is analogous to the laws of supply and demand which are universally applicable to trade. The laws were partially made under the influences of the laws of thermodynamics, with the introduction of entropy as an inherent property of software systems.

The laws, with the years of introduction, are listed below [Lehman, 1997]:

1. **Continuing Change** (1974) – an E-type system must be continually adapted or it becomes progressively less satisfactory.

2. **Increasing Complexity** (1974) – as a program is evolved its complexity increases unless work is done to maintain or reduce it.

3. **Self Regulation** (1974) – the program evolution process is self regulating with close to normal distribution of measures of product and process attributes.

4. **Conservation of Organisational Stability** (1978) – the average effective global activity rate in an evolving E-type system is invariant over the product's lifetime.

5. **Conservation of Familiarity** (1978) – during the active life of an evolving program, the content of successive releases is statistically invariant.

   In other words, everyone involved with the system, from the developers to the users need to know it to use it effectively. Large changes reduce the familiarity, therefore the average changes are invariant.

6. **Continuing Growth** (1991) – functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

7. **Declining Quality** (1996) – *E-type* programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.

8. **Feedback System** (1996) – *E-type* programming processes constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved (first stated 1974, formalised as law 1996).

Another observation made by Lehman and closely related to the laws is the *Principle of Software Uncertainty* which states that real world outcome of E-type software execution is inherently uncertain with precise area of uncertainty also not knowable.

Lehman and his colleagues made further studies to the laws with the *FEAST* project (Feedback, Evolution, And Software Technology) [Lehman, 2001]. A detailed overview of the changes to the laws as they were developed, and the literature that tried to validate them is given by Herraiz, Rodriguez, Robles, and Gonzalez-Barahona [2013]. The main problem of the laws is that they are not formally defined – there is room for interpretation, which leads to different studies reporting different laws being valid or not for similar systems. Notably, several studies that

looked at open-source solutions had different results. [Fernandez-Ramil, Lozano, Wermelinger, and Capiluppi, 2008; Godfrey and Tu, 2000; Israeli and Feitelson, 2010; Pirzada, 1988]

2.1.2 Maintenance and Evolution Methods

This section briefly presents some of the many methods and in some cases specific tools used for maintenance and evolution. A detailed overview is available for instance in [Tripathy and Naik, 2014].

Decompilation is a process of taking an executable file and trying to produce readable high level source code from it. In general, the result will not be the same as the original source code, usually for the same output there is a whole class of sources that could have generated it, especially when modern compiler optimisations are taken into consideration [Cifuentes and Gough, 1995; Cifuentes and Simon, 2000]. It was also proposed that decompilation could be helpful for more efficient handling of security problems [Cifuentes, Van Emmerik, and Waddington, 2001].

Wrapping (sometimes called *Encapsulation*) is a method applicable to an old, legacy piece of software that is still correctly functioning, but has problems with the changes to the "outer world" (for instance communication protocols have changed). It is kept as is, but then wrapped into a layer of software that emulates the environment the old software is used to and on the other side makes it available to the requested needs [Sneed, 2000]. This technique is relatively cheap and reliable, but has limitations, mainly when there are new requirements for the old software, or if the overheads of the wrapper layers are not acceptable.

Another method is to raise the abstractions of the current software to a model level, then restructure it there, and finally return it to an executable level. When making new software using this approach it is known as *Model Driven Development*, but the same ideas can be applied to reengineering old systems [Wagner, 2014].

Formal methods are those that are based on a mathematical approach, usually with some sort of a proofing system. In software development and maintenance they can be applied at different levels – starting from specifications, through models and down to the actual code. They can be used to make new programs, either as a way to add reliability and traceability to the process, or in more indirect ways, such as optimisation done by modern compilers. They come in different forms, but the main advantage has always been the increased reliability of the end product. The negative side that is often used against them is that they use too much resources for many projects compared to the actual improvements. A sensible conclusion is that while they are not universal (and should not be forced into every project), there are big advantages to be gained and there are certain types of software (such as critical

systems) where they are practically a necessity [Yang and Ward, 2003]. Different types of formal methods find more applications to different types of problems, such as process algebras for concurrency and communication problems, the Z notation for large industrial software or net-based formalisms for visual representations, etc. In model driven development the usage of formal methods can help to ensure that the resulting code conforms to the model [Hemel, Kats, Groenewegen, and Visser, 2010].

New technologies that try to improve development can also have negative consequences on maintenance. For instance, object-oriented programming is a staple of modern development, yet it introduces a host of problems in tracing execution of tests (for example, see the whole Chapter 7 of Ammann and Offutt [2008]), it can invalidate some of the earlier best practices and brings a need for new ways of handling problems. A collection of reengineering patterns for OOP is presented by Demeyer, Ducasse, and Nierstrasz [2008]. This results in development of new tools that are built around the idea of dealing with the new problems, such as Moose [Ducasse, Lanza, and Tichelaar, 2000]. Similar is applicable to most new methodologies, in line with the "no silver bullet" idea, there are no universal solutions to the complexity of the underlying problem. More tools related directly to Java bytecode are covered in Section 5.1.

The *GenProg* system has an approach to automated software repair via *genetic programming* [Le Goues, Forrest, and Weimer, 2013]. The code itself is represented as a sequence of statements and the genetic operators are applied on this level, while the fitness of the programs is decided based on positive and negative run time tests which represent the desired behaviour and the faults. The consequence is that for most applications the fitness evaluation is expensive – the experiments by Le Goues, Nguyen, Forrest, and Weimer [2012] on average spent 62% of the total time on this. The search space is usually a problem with approaches such as this one, but here it is significantly reduced by giving weights to the statements, so that ones that are executed in the negative test cases are more likely to be changed. The mutation operators were made with the assumptions that it is very likely that elsewhere in the program there are correct versions of the faulty code. This means that they do not generate entirely new code, but insert other statements from the program, which also reduces the search space significantly. The system has been successfully used with various C, assembly and Java bytecode programs including real world examples such as web servers and Unix utilities [Le Goues, Nguyen, Forrest, and Weimer, 2012; Schulte, Forrest, and Weimer, 2010]. The main advantages of the approach are that it can be fully automated and has been shown to fix real bugs at relatively low costs. The problems are that since it does not generate new code, there are types of faults that can not be easily fixed. It also relies heavily on the

quality of the test cases, which means that bad tests can result in degradation of the original program. On the other hand, it was shown that the more positive tests it has, the process will likely be faster, since the weights of the test paths will be more reliable and the search space is further reduced. Finally, as the authors themselves note, the patches themselves can be quite non-intuitive and hard to understand. Therefore, they can lead to a software that is harder for future maintenance.

2.1.3 Program Transformations

A *program transformation* can be defined as any operation that takes a computer program and generates another program [Ward, 1989]. Transformations themselves can be applied directly to source code (on the level of tokens or statements) or potentially to some model of it (such as the common abstract syntax tree). Focus is often on semantics-preserving transformations, as will be in the later chapters of this thesis, but in general transformations can introduce changes to the program at hand. For instance, a patch for a software fault could be viewed as a transformation.

Transformations can be used to improve existing programs (or specifications, or models) or to add new features to them. For example, it is possible to extract models from source code that can give a better insight into the original code and allow for high level transformations. *Gra2MoL* is a tool that can extract models from any text that conforms to grammars [Cánovas Izquierdo and García Molina, 2014]. Formal methods can also be used for verifying model transformations done in non-formal ways [Ab. Rahim and Whittle, 2015].

One of the major approaches are rule-based transformation systems, where in general the *rules* are the individual actions that can be applied to the program, and there are *strategies* for the selection and application of those rules. An overview of the principles of such systems can be found in [Visser, 2005]. One example of such a system is *StrategoXT* that was later on integrated into *Spoofax* [Kats and Visser, 2010; Visser, 2004].

The *FermaT* transformation system and the language WSL are an example of a self modifying (or meta-programming) approach, where the program being modified and the modifying program are in the same language [Ward and Bennett, 1995a]. More detail about this system will be given in Chapter 3.

Rascal is a domain specific language (DSL) built around the idea of meta-programming and building tools for program manipulation [Klint, Storm, and Vinju, 2011]. It can be used for refactoring and analysing existing program, or construction of other DSLs. It has been used on C, Java, and PHP among others [Hills and Klint, 2014].

SmaCC (Smalltalk Compiler-Compiler) is a parser generator for Smalltalk, which was successfully used to write custom reengineering and transformation tools [Brant and Roberts, 2009]. It was used on Java, C#, and Delphi, with one listed example of a migration of 1.5 million lines of Delphi to C#. SmaCC was also ported to the language *Pharo*, making it possible to use it with *Moose* for data analysis, or other related software written in *Pharo* [Brant, Lecerf, Goubier, and Ducasse, 2017].

2.2 Software Metrics

Software metrics are measurements of properties of software development and software products. A distinction can be made to use the term metric just for the function that calculates the result and not for the actual measurements, but the term is generally used interchangeably for the end results. While there are metrics that apply to the whole process or the project (effort estimates and similar), the focus in this thesis is on product metrics, those that look at the software itself and its characteristics. Metrics are used here to evaluate the process as a whole, by comparing the results of the initial and transformed programs. More importantly, they are used as a guide in the automated transformation process presented in Section 8.2.

Metrics can be very useful in a process, both as a guide for improvements and to better locate and understand potential problems. It is often repeated "you can not control what you can not measure" [DeMarco, 1986], or a variation to the theme, in the sense that you need comparable data to be able to improve on a process. On the other hand, W. Edwards Deming (and others) have called this a myth [Deming, 1991]. While having data is crucial, there are things that need to be managed even though they can not be measured, and focusing on just having measurements can divert attention from important problems. After all, some qualities of software are hard to express in numbers, and blind faith in them can sometimes do more harm than good [Kaner, Member, and Bond, 2004].

A common problem with metrics is the lack of consistency of definitions and results – due to differences in interpretations, the numbers can vary even for the same language with the change of tools [Lincke, Lundberg, and Löwe, 2008; Novak and Rakić, 2010]. Additional problems can arise in complex projects where multiple languages are used for different components, making comparisons even harder. *SSQSA (Set of Software Quality Static Analysers)* tries to solve this problem by universally representing code from different languages with the enriched Concrete Syntax Tree (eCST). Part of the system is *SMIILE (Software Metrics Independent*

on Input LanguagE) which calculates metrics from the generated eCST, but there are other tools as well [Rakić, 2015].

The metrics mostly used in this thesis are those built-in to the FermaT transformation system and that are applied to WSL. They are listed in Section 3.5, with some basic explanations of how they work.

Chapter 3

FermaT and WSL

FermaT is the current implementation of the language WSL (short for *Wide Spectrum Language*) and the surrounding code transformation libraries. It has been used in several industrial projects of converting legacy code (IBM 370, x86, a less known Herma assembly and others) to human understandable and maintainable C and COBOL [Ward, 1999, 2004, 2013; Ward and Bennett, 1995b; Ward, Zedan, and Hardcastle, 2004]. It also has support for program slicing [Ward and Zedan, 2017] and can be used to derive program code from abstract specifications [Ward and Zedan, 2014]. A companion graphical application *FermaT Maintenance Environment (FME)* [Ladkau, 2007] is also available and can be very useful, especially for initial experiments with the transformation system. Both of these tools are available under the GPL v3 software licence on the project's web site[1] and work on most computer platforms, including Linux, Windows and Mac OS.

This chapter will give an overview of WSL and some of the basic concepts and ideas, but will not go in depth with the complete syntax of the language that is available in the official manual [Ward, Hardcastle, and Natelberg, 2008].

3.1 WSL History

WSL is short for *Wide Spectrum Language* in the sense that it can be used for anything from abstract specifications to concrete implementations that are runnable, even with structures and commands that are more specific to low level

[1] `http://www.gkc.org.uk/fermat.html`

languages. Early versions of the language were developed as "The Maintainer's assistant" [Ward, 1989] written in LISP. It included a large number of transformations and was relatively successful at restructuring assembly modules into equivalent high-level languages, but had more of an academic approach with not much attention given to spacial and temporal efficiency.

The next major version was a complete reimplementation of the transformation engine under the name *GREET – Generic REverse Engineering Tool*, with the first appearance of *MetaWSL*, an extension of the language for writing transformations directly in the language. It also featured abstract data types that represented programs as trees and had constructs for iterating over those trees. The system contained a translator from MetaWSL to LISP for execution and libraries for some high level constructs. There were parsers built for IBM 370 assembly language and JOVIAL[2] [Yang and Ward, 2003, Chapter 5.2].

The language is currently implemented in the FermaT transformation engine. The main system is fully written in MetaWSL and represents programs as abstract syntax trees. For sake of execution speed, WSL is translated into Scheme and executed with the help of *Hobbit* (an optimised translator to the C language). Some parts of the system are bonded together with *Perl* and *sh* scripts. The version used in most of the experiments in this thesis was internally labelled 18c, as the third major version used in 2018.

There is also work to introduce new concepts to the language to make it more accessible to other problems at hand. There was an expansion to support object oriented programming [Chen, Yang, Qiao, and Chu, 2006]. Another one was to make new approaches to concurrent programming [Younger, Bennett, and Luo, 1997].

Another big project was the development of a full type system that would make a base for future expansions, allow new ways to handle object oriented code and to improve the current transformations by giving them additional semantic information to work with [Ladkau, 2009].

3.2 WSL Core

The theoretical kernel language of WSL is based on infinitary first order logic with a mathematical model in which the semantics of a program are functions that switch

[2] JOVIAL is a language similar to ALGOL, but more focused on embedded systems, mainly military air crafts. It was developed in 1959 by a team at System Development Corporation (SDC) headed by Jules Schwartz. The acronym stands for "Jules' Own Version of the International Algebraic Language"[J. I. Schwartz, 1978]

from a state to a set of states. The states themselves are a mapping of variables to their values. Then, a program can be viewed as a function f that maps some initial state s to a set of potential final states $f(s)$. There is a special state $\perp$ that represents an error state or a non-terminating program – in the context of these definitions these are equivalent since they lead to no result. If two programs have the same semantic function they are considered equivalent, with no consideration of how they reach the same results.

With this model, a relation of *refinement* between programs (that is, *state functions*) can be defined and written as $\leq$, such that f_1 is refined by f_2:

$$f_1 \leq f_2 \iff \forall s f_2(s) \subseteq f_1(s)$$

In other words, a program is more refined if its set of final states is smaller than the other program's. If two programs refine each other then they are equivalent.

The lowest level of the WSL language consists of the following primitives:

- $\{P\}$ **Assertion** – if formula P is false, the program aborts, otherwise the statement terminates in a normal fashion.
- $[Q]$ **Guard** – enforces Q to be true at this point, but with no changes to the values of the variables. The main usage of this is to restrict previous non-determinism in the program, since the model allows a *set* of final states. In the case that Q can not be made true, then the set of final states becomes empty.
- $add(X)$ **Add variable** – adds all the variables from set X to the state space, if they are not already present, and assigns them arbitrary values. This can be combined with the guard statement to have desired values.
- $remove(X)$ **Remove variable** – removes the variables from set X if they are present in the current state space.

These primitives can then be composed together with the following rules:

- $(S1; S2)$ **Sequence** – execute $S1$, then $S2$.
- $(S1 \sqcap S2)$ **Non-deterministic choice** – execute one of $S1$ or $S2$.
- $\mu X.S$ **Recursion** – S is a sequence of statements which may contain occurrences of X. These are the recursive calls, and occurrences of X are replaced with the whole body of S.

These constructs may seem quite strange as a basis for representing computer programs, but when combined they produce all of the regular structures that are expected. As hinted in the descriptions, an assignment is a combination of adding a variable and restricting its value. A classical if statement is made of blocks that will execute just one branch, while a conditional loop can be made using the recursion rule, as shown in Table 3.1. Using such rules further, other structures such as procedures and functions in the classical sense of programming languages can be defined.

Table 3.1: Examples of WSL statements and equivalent kernel statements

WSL code	Kernel Language
`z := 5`	$add(z); [z = 5]$
IF c **THEN** S1 **ELSE** S2 **FI**	$([c]; S1) \sqcap ([\neg c]; S2)$
WHILE c **DO** S **OD**	$(\mu X.(([c]; S; X) \sqcap [\neg c])$

The consequences of infinitary logic is that the kernel language itself is not implementable for the most part. For this reason these are sometimes described as *the Quarks of Programming* [Ward, 2004]. Just like quarks, the elements of the kernel are not observable in isolation, yet they combine into items which were previously considered as fundamental in programming. The advantage of starting from this level is that structures can be *proven* to be right and any new type of statement only needs to be proven from the existing ones, with no need for a full re-evaluation of the system. The implementation in FermaT therefore starts from a level higher than the kernel, while still maintaining most of the advantages of the definitions.

3.3 Action Systems

An *action system* is a specific structure developed in WSL with the goal of representing messy ("spaghetti") code with a lot of jumps and go-to style commands in such a way that can be mixed with the higher level structures and eventually converted into them. This is the type of programs that is typically found in legacy libraries and low level code in general. An action system consists of a number of actions, each defined by its name and containing a block of code. In general it is similar to a collection of procedures with no parameters, but the execution of the whole system can be interrupted. A basic example is shown in Figure 3.1. The

action whose name is used at the beginning of the system is the one from which the execution starts, while CALL commands are used to jump to other actions. Once that action ends, execution will be returned to the caller. The whole action system stops its execution once the start action finishes, or if there is a call to a special action named "Z" which terminates the system.

```
ACTIONS start:
  sidequest ==
    PRINT("sidequest")
  END
  start ==
    PRINT("start");
    CALL sidequest;
    PRINT("we are back");
    CALL final;
    PRINT("the end")
  END
  final ==
    PRINT("final")
  END
ENDACTIONS
```

Figure 3.1: An action system

Depending on how the jumps and returns are used there are three types of action systems. The first one is the *recursive* in which all the calls are returned normally, like the first example shown (Figure 3.1). This is what is commonly expected out of recursive procedures. The second one is the *regular* system in which none of the calls ever return – the system is terminated by a CALL Z (Figure 3.2). Regular action systems have some advantages in transformations. For instance, since the calls are not returning, then any code that comes after a CALL command can be ignored, and CALL commands are therefore just simple jumps. In fact there are a lot of transformations in FermaT that can only be applied to regular systems and in turn often produce regular systems (with obvious exceptions such as converting an action system into "normal" code). The third type is the *hybrid* action system which is any combination of both returning and non-returning calls.

Tools that translate legacy code to WSL tend to use action systems for most or all of the structures and control flow commands. This includes both of the tools developed in this thesis, as well as the tools used and developed by Martin Ward and Software Technology Research Laboratory [Ward, 2004]. FermaT supports the generation and visualisation of a call graph in an action system, which can be helpful

```
ACTIONS start:
  middle ==
    PRINT("middle");
    CALL final
  END
  start ==
    PRINT("start");
    CALL middle;
  END
  final ==
    PRINT("final");
    CALL Z
  END
ENDACTIONS
```

Figure 3.2: A regular action system

for a better understanding of the underlying system. These can also be generated and visualised using FME (FermaT Maintenance Environment) [Ladkau, 2007].

3.4 Working With Meta-WSL

WSL has the ability to work with programs written in WSL itself. This part of the language is called *MetaWSL*. A piece of code under inspection is represented by an abstract syntax tree (AST), or more precisely in the current implementation by a list that can have lists as its elements. Meta-WSL procedures know how to handle these lists and what to expect in them when they represent a valid program. Items in the lists have associated general and specific types. General types are statement, expression, condition, value, definition and list versions of some of these. Specific types are, as the name implies, more specific, and can be number, string, "less", "for statement", etc. For instance, a while loop is of type `T_While` and contains in its list a `T_Condition` and a `T_Statements` item. An assignment is of type `T_Assignment` and holds one or more items of type `T_Assign` (it is a separated type because it is used in other places as well, and there can be multiple simultaneous assignments) which in turn holds an `T_Lvalue` and a `T_Expression`. Tables with these relations can be found in the reference manual for WSL [Ward, Hardcastle, and Natelberg, 2008], or can be found by directly extracting the information from the source code with a command like shown in Figure 3.3.

```
> grep "\[T_Assign\]" $Fermat/src/adt/WSL-init.wsl

Syntax_Name[T_Assign]              := "Assign";
Syntax_Comps[T_Assign]              := <T_Lvalue, T_Expression>;
```

Figure 3.3: Grepping component types in WSL

To distinguish MetaWSL from "regular" WSL, there are definitions for special meta-procedures (`MW_PROC`), meta-functions (`MW_FUNCT`) and meta-boolean functions (`MW_BFUNCT`). The names for all of these should start with an `@`.

These lists that represent code can of course be stored in regular variables, passed as arguments, or be returned by certain functions. To assist many operations (especially transformations) there is also a globally defined "current program" that is being worked on. The current position in it (represented by a list of indexes in the lists starting from the top one) and the current items are also globally available. There are many built-in procedures that enable the user to move around in the program, inspect it and to change it, some of which are:

- `@Program` returns the whole current program
- `@Posn` returns the current position (list of indexes in the AST)
- `@I` returns the current item; this can be anything from the whole program, down to individual variable names
- `@New_Program` accepts a single parameter, an AST, which will be set as the current program
- `@Print_WSL` displays the AST of the of the received item as shown in Figure 3.4; it can be very useful to understand what are the components of particular statements
- `@PP_Item` is a pretty print procedure, that accepts three parameters: the item, the width of the maximum line, and the file to print out the code to; if the filename is an empty string it will be printed out to the standard output
- `@Checkpoint` is a convenient shorthand when working with the whole program, since it always prints the whole program in 80 characters width to the file name given as a parameter

FermaT's pretty printer is somewhat different from the wide spread C-like style and more in line with LISP. The indenting in most of the figures that feature WSL is done in this style.

There are many more procedures to inspect the properties of the passed item, such as `@GT`, `@ST`, `@Components`, and `@Value` which return the general type, specific type, components, and value, respectively.

Program:

```
@New_Program(FILL Statements
a := 5;
b := 10;
PRINT(a + b)
ENDFILL);

@Print_WSL(@Program, "")
```

Output:

```
Statements
   Assignment
   :  Assign
   :     Var_Lvalue a
   :     Number 5
   Assignment
   :  Assign
   :     Var_Lvalue b
   :     Number 10
   Print
   :  Expressions
   :     Plus
   :        Variable a
   :        Variable b
```

Figure 3.4: Creating a new program and viewing its structure

3.4.1 Item Simplification

One of the common activities in transforming programs is to detect specific patterns and simplify them. WSL has matching constructs for this type of work. To demonstrate a simple example we shall make a matcher for calling absolute values on negated values and replace them with the value itself (of course this is already present in FermaT). To quickly test our matcher, we can create a small program that will define an entry to test on, use the checkpoint command to display the code before and after the changes, and use the `FOREACH` construct to apply our matcher to all expressions. This is demonstrated in Figure 3.5. Alternatively, we could use commands to move the current position (such as `@Down`, `@Right`, etc) in the program to an appropriate expression and apply it there.

Displaying the changed program or saving it in a file, gives a good indication whether the changes were successful, but additional checks can and should be applied especially when prototyping. This can be done with `@Syntax_OK?`, as

```
@New_Program(FILL Statements
a:=5; b:=-5; PRINT(ABS(-a));PRINT(ABS(-b))
ENDFILL);

@Checkpoint("");
PRINT("transforming");

FOREACH Expression DO
  IFMATCH Expression ABS(- ~?x)
    THEN @Paste_Over(FILL Expression ABS(~?x) ENDFILL) ENDMATCH; OD;

@Checkpoint("");
IF @Syntax_OK?(@Program)
  THEN  PRINT("Syntax OK")
  ELSE  ERROR("Bad syntax") FI
```

Figure 3.5: Absolute value expression matching

shown in the end of Figure 3.5. This can catch some subtle errors. For instance, if in `@Paste_Over` we used "Expressions" instead of "Expression", the output would look appropriate and the saved file would actually be valid, but the syntax tree would have a problem. If this was part of a larger program, attempting further transformations would likely fail. Calling `@Syntax_OK` on this version of the code would result in a report shown in Figure 3.6.

```
Expressions
   Expressions
      Abs
         Variable a
Bad type at (3 1 1)
Gen type is: Expressions(10)  Should be: Expression(2)
```

Figure 3.6: Example of a bad syntax report

FermaT has a built in maths simplifier and several procedures that rely on it. Procedures such as `Simplify`, `Simplify_Cond` and `Simplify_Expn` return a new simplified item, while others like `@Or`, `@And` will first combine conditions and try to simplify them together. More information about the simplifier and adding patterns to it can be found in the manual and the documentation that comes with FermaT (see `doc/adding-patterns.txt`).

3.4.2 Example: Converting Numeric Codes to Strings

As a working example we will work on a relatively simple and understandable problem. One of the tools that generates WSL code works on bytecode that has numeric codes for characters and handles that by creating `@List_To_String` calls with the appropriate numbers [Pracner and Budimac, 2017b]. To make the code more understandable to humans and more compact in general, we will transform those into strings.

The first version (Figure 3.7) assumes that everything in a `@List_To_String` is a number literal and will not do anything if this is not the case. If all the items in the list are numbers, the `MAP` function can be used on the list to apply `@V` on each item, which converts the abstract items into their values. The resulting list can then be converted to a string with `@List_To_String` and finally a valid string item is created with `@Make`. This is then pasted over the initial expression. As can be seen from the output shown in Figure 3.7 this means that if a numeric variable is in the list, the code will not transform it at all. In this example a constant propagation transformation executed before this code would replace the variable with the value and actually solve the problem. In a general case, the variable "a" could be a user input, or a procedure parameter, and might not be replaceable.

To have a more fine grained handling of individual items, we need to go through them one by one and store the changed versions in a list (called "res" in the code given in Figure 3.8). If a numeric code is found it should be converted to a character and added to the list. If the previous item in the list was a string, it should be added to it, otherwise it is added as a new item in the list. In this version we will also include anything that is not a number as a separate entry in the resulting list. The final step (when the list is completed) is to paste it over the original expression, but it is important to consider that multiple entries should be concatenated, while a single entry is directly pasted over.

It is important to note that while this insistence on converting numbers to characters increases the readability of the code, it can also result in a more complex program with more expressions. This is often not desirable, and is definitely not something to be included in the general simplifier. The actual code that is included in FermaT's simplifier is shown in Figure 3.9 and is much more similar to the first version of the code – just much more compact. It also takes care to not convert number 34, which is the code for double quotes and can not be included in a regular string as such, since it is the string delimiter. Further effort could be made to handle this case as well, but it is not very common and it would increase the code complexity.

Program:

```
@New_Program(FILL Statements a := 92;
   PRINT(@List_To_String(<65>));
   PRINT(@List_To_String(<65, 66>));
   PRINT(-a);
   PRINT(@List_To_String(<a>));
   PRINT(@List_To_String(<a, 67>))
              ENDFILL);
@Checkpoint("");
PRINT("transforming ----------");
FOREACH Expression DO
  VAR < IS_OK := 1, x := < >, str := "" >:
  IFMATCH Expression @List_To_String(<~*x>)
    THEN FOR elt IN x DO
      IF @ST(elt) <> T_Number
         THEN IS_OK := 0 FI OD;
    IF IS_OK = 1
      THEN str := @Make(T_String,
                @List_To_String(MAP("@V", x
                    )), < >);
           @Paste_Over(str) FI
    ELSE SKIP ENDMATCH ENDVAR OD;
PRINT("result ----------");
@Checkpoint("")
```

Output:

```
....
result ----------
a := 92;
PRINT("A");
PRINT("AB");
PRINT(-a);
PRINT(@List_To_String
    (<a>));
PRINT(@List_To_String
    (<a, 67>))
```

Figure 3.7: Program that converts lists with only numbers to strings

```
FOREACH Expression DO
  VAR < x := < >, res := < > >:
  IFMATCH Expression @List_To_String(< ~*x >)
    THEN
      FOR elt IN x DO
       IF @ST(elt) = T_Number
         THEN IF NOT EMPTY?(res) AND @ST(HEAD(res)) = T_String
            THEN
                t := HEAD(res);
                t := @Make(T_String,
                           @V(t) ++ @List_To_String(< @V(elt) >), < >);
                res := TAIL(res);
                res := < t > ++ res
            ELSE
                s := @Make(T_String,
                           @List_To_String(< @V(elt) >), < >);
                res := < s > ++ res  FI
         ELSE
            res := < FILL Expression
                      @List_To_String(< ~?elt >) ENDFILL > ++ res
         FI OD;

        IF LENGTH(res)>1
           THEN @Paste_Over(@Make(T_Concat, < >, REVERSE(res)))
           ELSE @Paste_Over(HEAD(res)) FI

    ENDMATCH ENDVAR OD;
```

Figure 3.8: Second version of the string converter

```
IFMATCH Expression @List_To_String(<~*x>)
    THEN VAR < OK := 1 >:
         FOR elt IN x DO
           IF @ST(elt) <> T_Number OR @V(elt) = 34 THEN OK := 0 FI OD;
         IF OK = 1
           THEN @Paste_Over(@Make(T_String,
                                  @List_To_String(MAP("@V", x)),
                                  < >)) FI ENDVAR ENDMATCH OD;
```

Figure 3.9: String simplification included in FermaT

There are also a few other improvements that will be discussed in the next section when the code from Figure 3.8 will be converted into a full transformation.

3.4.3 Writing A Transformation

Transformations in WSL can be anything that changes the current program while keeping the semantics of the original (with a few potential exceptional cases). For example, there are transformations that will reverse an IF/ELSE statement (for instance to make the condition evaluation simpler) or unroll the first iteration of a FOR statement. Additionally, there are applicability tests that will check if the semantics would be affected by the specific transformation. For instance, there is a transformation that deleted the current item, but only if it is redundant. Therefore the test for this transformation is checking if the current item is redundant.

The main exception to the preservation of full semantics are the slicing transformations, as these by definition preserve only a part of the original behaviour of the program that is relevant to the slicing criterion. These will not be covered in this thesis.

Transformations themselves are represented by two WSL files. The first one holds the code of the transformation. There are two main entry points that need to be defined in this file. The first one is a test procedure that has no parameters and checks whether the transformation can be applied to the current item in the program. It should raise errors with `@Fail` if the transformation is inapplicable, or call `@Pass` otherwise. The other procedure is the actual transformation that receives a single parameter with any potential additional data. For example, a rename transformation will receive the old and the new names. Other than these, there can be any number of helper procedures defined. To comply with the definition of a WSL program the file also needs to contain a body for the main program which can be a single `SKIP` instruction.

The second file should be named the same as the first one with a "`_d`" suffix and it holds the description of the transformation and some meta information as well as the names of the actual procedures to test and to apply the transformation from the first file. Figure 3.10 shows how this file should look for a new transformation that is based on the code shown in the previous section.

The transformations that are built into FermaT are kept in the `src/trans` folder and new ones can be added there and the whole system recompiled. Alternatively, it can be added "on the fly" in the working directory using a `patch.tr` file. More details about this can be found in the manual and the documentation that comes with FermaT.

```
IF EMPTY?(TR_SimplifyChar ) THEN TR_SimplifyChar := @New_TR_Number()
    FI;

TRs_Proc_Name[TR_SimplifyChar] := "SimplifyChar" ;
TRs_Test[TR_SimplifyChar]:=!XF funct(@SimplifyChar_Test);
TRs_Code[TR_SimplifyChar]:=!XF funct(@SimplifyChar_Code);
TRs_Name[TR_SimplifyChar] := "Simplify Char";
TRs_Keywords[TR_SimplifyChar] := < "Simplify" > ;
TRs_Help[TR_SimplifyChar] := "Simplify Char will find expressions like
    '@List_To_String(<97>)' and replace them with chars.";
TRs_Prompt[TR_SimplifyChar] := "";
TRs_Data_Gen_Type[TR_SimplifyChar] := ""
```

Figure 3.10: Transformation description file

Figure 3.11 shows how the main file with the new transformation should look like. This transformation has no additional procedures. The test procedure just calls `@Pass` always, since there are no specific pre conditions needed for the transformation to be applied and there are no semantics changes. At worst it will not find anything to change and leave the original program as it is. A more zealous version of the test procedure could check if there are any `@List_To_String` items and even analyse their content to see if the transformations will change anything. On the other hand, a general approach to programs that call transformations is to first call the test and then to call the main transformation, which would result in duplicate checks and loss of efficiency.

The main procedure is very similar to the earlier developed version shown in Figure 3.8 with some commands being less verbose and less temporary variables used, which in turn can make it harder to read. There are several functional improvements beside that. In a proper program `@List_To_String` should receive a list of numbers and numeric variables. Anything other in the list results in undefined behaviour, therefore it is probably best if our transformation leaves any problematic call as it was. This means that the new version only handles numbers and variables, and if the current item is anything else it sets the error flag which will result in no changes being applied. The other thing that is changed is that the new version correctly handles an empty list being passed to the procedure and replaces it with an empty string (the previous version would crash if this was the case).

```
MW_PROC @SimplifyChar_Test() ==
  @Pass END;

MW_PROC @SimplifyChar_Code(Data) ==
  FOREACH Expression DO
  IFMATCH Expression @List_To_String(<~*x>)
      THEN VAR < res := < >, IS_OK := 1 >:
      FOR elt IN x DO
        IF @ST(elt) = T_Number
          THEN IF NOT (EMPTY?(res)) AND @ST(HEAD(res)) = T_String
                 THEN res := <@Make(T_String, @V(HEAD(res))
                      ++ @List_To_String(<@V(elt)>),
                      < >)>  ++ TAIL(res)
                 ELSE res := <@Make(T_String,
                      @List_To_String(<@V(elt)>), < >)> ++ res FI
        ELSIF @ST(elt) = T_Variable
          THEN res := <FILL Expression @List_To_String(<~?elt>) ENDFILL
              >
                      ++ res
          ELSE IS_OK := 0 FI OD;
      IF IS_OK = 1
        THEN IF EMPTY?(res)
               THEN @Paste_Over(@Make(T_String, "", < >))
             ELSIF LENGTH(res) > 1
               THEN @Paste_Over(@Make(T_Concat, < >, REVERSE(res)))
               ELSE @Paste_Over(HEAD(res)) FI FI ENDVAR
      ELSE SKIP ENDMATCH OD END;

SKIP
```

Figure 3.11: Transformation main file

3.5 Built-in Metrics

FermaT comes with a number of built-in software metrics, that will work on the item passed as a parameter to the appropriate MetaWSL procedure.

Statements count (`@Stat_Count`) returns a number of all statements in the passed item. Since comments are technically statements in WSL, there is also a `@Stat_Count_NC` variant.

General type count (`@Gen_Type_Count`) returns the number of items of the general type specified by the first parameter in the Item passed as the second parameter. For instance this metric can be used to count the number of expressions in a program by passing `T_Expression` as a parameter. Other general types are statement, condition, value and definition.

Specific type count (`@Spec_Type_Count`) is analogous to the general type count, and can be used for items with the passed specific type, such as assignments, mathematical operations, logical operations, loops, procedure calls, etc. Some of the types that represent lists of items (such as statements, conditions, expressions) have the same specific and general type.

McCabe cyclomatic complexity (`@McCabe`) implements the "classical" definition of the number of "basic paths" that can be taken through the program represented as a control flow graph.

McCabe essential complexity (`@Essential`) was defined in the same paper as the cyclomatic complexity, and it refers to the cyclomatic complexity of control flow graph that has all of the structured control structures reduced. In other words all of the loops, branches, and similar single-entry-single-exit structures are replaced with placeholder statements. A value of one represents an "ideally" structured program.

CFDF (Control Flow, Data Flow) (`@CFDF_Metric`) gives a good estimate of the data exchange in the program, by counting the assignments of variables, as well as all types of procedure calls in a program.

Number of nodes (`@Total_Size`) returns the size of the abstract tree that represents the passed item.

Branch loop metrics (`@BL_Metric`) returns the number of loops (while, for, endless loops) combined with the number of procedure calls of any type.

Structure (`@Struct_Metric`) returns a custom weighted metric of the various parts of the given item. In the implementation used it gives the least weight of 1 to simple operations such as addition and multiplication, operands such as division and exponent have weight 2, other expressions are weighted at 4, `IF` statements are five times the number of branches they have, while all types of calls and jumps have even higher values.

Statement types (`@Stat_Types`) is not a metric in the classical sense of returning a value. Instead it returns all the statement types in the specified item.

Chapter 4

Assembly Language

This chapter deals with the basic ideas of assembly language and specifically the x86 architecture with the MASM dialect that was used as the input for one of the translator developed during the work on this thesis. For the purposes of this work most of the other variants of assembly would lead to similar results, and similar tools could be constructed. The main reason for this choice was the author's basic familiarity of with these particular concepts at the time.

At the lowest and most direct level, programs are actually machine code instructions, sequences of numbers that represent various instruction codes and their optional parameters. However, this is a layer that humans almost never use directly and have not been doing since the very earliest of computers. What is used are assembly languages which still do not offer almost any abstractions to the internal operations of a processor, but present these in a human readable way. Instructions are represented by mnemonics instead of numbers, and similar is mostly available for the registers, flags and other internal structures of a processor. Being so closely related to direct architecture makes these languages very specific and not portable in a general case.

The first assembly language was probably the one created in 1947 by Kathleen Booth (née Britten) for the ARC computer that was developed at Princeton University under the leadership of John von Neumann [Booth and Britten, 1947, 1949].

With the appearance of high-level languages and abstractions, the usage of assembly significantly reduced through the years. It is still used for writing device drivers and other low-level interfaces that are presented to "regular" applications by the operating system (or a virtual machine). It is also still used for many embedded

devices. For a long time it was also used for critical pieces of software that required speed and optimisations, with many compilers offering options to directly integrate it with high-level languages. Over time with the advancements in hardware speed and compiler optimisations, these needs have reduced significantly.

A *disassembler* is a reversing tool that takes in machine code and produces a version that is more readable to humans. Standard assemblers will not include any superfluous data in the binaries, so it is impossible to reconstruct any original comments, variable names, etc. On the other hand, some disassemblers can enrich the code with comments about the calls. A *decompiler* similarly takes machine code as its input, but is supposed to translate it into a high-level language. This is most often done on executables that were made with a compiler, and generally the goal is to get close to the initial source code. This is however a very difficult task, due to the levels of optimisation introduced, and the general ambiguity of the translation to assembly. For instance, a tool developed to produce C language code from SPARC assembly, *asm2c*, was able to reduce the size of programs by 66% on non-optimised assembly and by about 5% on optimised versions [Cifuentes, Simon, and Fraboulet, 1998]. An important aspect of the end results is the correctness, since some abstractions can be wrongly interpreted. *Phoenix*, a GNU C decompiler, was proven on a set of programs from *coreutils* to outperform the (as they stated) *de facto* industry standard tool *IDA Hex-Rays*[1] [E. J. Schwartz, Lee, Woo, and Brumley, 2013]

4.1 x86 Architecture

x86 refers to a family of architectures based on the Intel 8086 and 8088 central processor units, with a long line of backward compatibility. The term comes from the common suffix of several generations of successor processors from Intel: 80186, 80286, 80386 and 80486, before switching to the "Pentium" brand name. The original processors had 16-bit registers and words, but starting from the 386 they were expanded to 32-bit to be able to access more memory, among other things. Later expansions to 64-bit had different names, so since then, this term is mostly used to refer to the 32-bit instruction set. The tool presented in Section 6.1 works with a subset of the architecture and just the basic concepts of it. For those purposes it mostly considers the 286 processor, which is what this chapter will mostly focus on [*80286 Programmer's Reference Manual* 1987].

The x86 assembly language has two main syntax dialects – the one used originally by Intel in their documentation and the other one used by AT&T. The main

[1] `https://www.hex-rays.com/products/decompiler/index.shtml`

differences are the order of the operands, how the literals (also called "immediates") are marked and similar details. Intel syntax is also used by Microsoft's *Macro Assembly (MASM)*, and Borland's *Turbo Assembler (TASM)* [*Turbo Assembler 2.0 User's Guide* 1990]. On the other hand, the most prominent user of AT&T syntax is *GNU Assembler* or *gas*, which is actually a cross platform assembler and not just for the x86 architecture. Due to these programs being more popular with different groups, the first syntax is more dominant in the MS-DOS and Windows world, while the other one is more prominent in Unix related systems – especially since Unix originated from Bell Laboratories, which were part of AT&T.

A single program executed by an x86 processor has several memory segments attached to it. Typically, based on the original specifications, the instructions themselves are stored in the *code* segment, variables and other data is stored in the *data* segment, the *stack* segment refers to the active stack, and finally there is an *extra* segment which can be used as needed. Starting with 80386 there are two more segments, *FS* and *GS*, with no specific predefined uses.

When the program is loaded into memory it is literally made out of the instruction codes and numbers that represent addresses, offsets and similar. Usually a programmer will work with some sort of an assembler program, which will give a layer of conveniences and abstractions. For instance, labels can be defined in the code to "point" at certain instructions, and then these can be used as jump targets, instead of actually counting the relative offsets of the instruction. Similarly there are variable definitions for offsets in the *data* segment. In both MASM and TASM there are macro definitions, which are parts of the code that can be called in a way similar to procedures in higher level languages, but these actually get inlined during the assembly stage (the translation to instruction codes).

4.1.1 x86 Registers

In a processor registers are special internal memory locations that are quickly accessable and can have special functions for different instructions. The registers in the x86 architecture are not symmetrical, in the sense that most have some special features for some instructions and are not interchangeable. Some of the registers are referred to as "general use", but even those usually have specialities. The following are the main registers in an 80286 processor, all of which are 16-bit.

General use registers: AX, BX, CX, and DX. CX is the only one that can be used with the internal `loop` instruction. AX and DX have special meaning for some string instructions, as well as for division and some others. BX can specially be used for address offsets. All of these can also be accessed as two

8-bit registers containing the corresponding high-and low-parts of the original register, for instance AH and AL are parts of AX.

Segmentation registers: CS, DS, SS, and ES contain the starting addresses of the memory segments currently in use, respectively: code, data, stack and extra.

Address registers: SI, DI, SP, and BP are mostly used for special purposes, although they can also be used for general operations. SP is the *stack pointer* and generally holds the address of the current top of the stack in the stack segment, while BP is the *base pointer* and in most situations points to another address in the stack segment. For instance, pointing at the return address in a procedure stack frame, and separating the local variables from the parameters given in the call. Finally, SI and DI are *source and destination indexes*, and have special meaning for string operations, but can be used otherwise as offsets in data structures. Specially, register BX can also be used for addressing purposes and is also referred to as a "base" register together with BP.

Special registers. *Flags* is a register with special meaning for the bits in it, such as last operation had an overflow, meant to be read by some instructions (more detail later). IP, or *instruction pointer*, holds the position of the operation that is about to be executed by the processor, and it is not meant to be accessed or changed by the program directly, but only implicitly by special instructions such as jumps.

Later on, with the introduction of the 80386, most of these registers would become 32-bit, the general ones renamed with an "E" prefix and still being accessable by the old names as the lower 16 bits. With the *AMD64* architecture there is another prefix, "R", for the 64-bit versions.

The FLAGS register consists of several flag fields, which are often just referenced by their first letter, and are mapped to individual bits as shown in Table 4.1. They are set as a result of some instructions, while other instructions (mostly jumps) are dependant on some of these fields. The flags are:

Carry flag indicates if there is a carry or borrow out of the most significant bits in the previous arithmetic operation, that is whether the result is too big for the number of bits used. It is used by some instructions, such as "addition with carry".

Auxiliary Carry (or Adjust) Flag is set to show if there was a carry from the least significant 4-bits in the last operation. The main usage is for *binary*

coded decimal arithmetic in which each decimal digit is represented by a fixed number of binary digits – most often 4.

Zero flag indicates that the result of the last instruction was zero (commonly used to check if two number are equal after subtracting them).

Sign flag indicates whether the result was negative (when SF=1) or positive.

Overflow flag is set to show that an overflow has occurred in the most significant bit, or in other words when signed two's complement representation is used. In some ways similar to the carry bit, usage of one or the other is affected by what types of numbers are used.

Parity flag indicates whether the sum of the lower 8 bits of the operation is even or odd, or in other words, if there was even or odd ones in the lowest byte of the result.

Direction flag is used specially in string instructions to control the forward or backward direction of the operation, i.e., whether the address registers are incremented or decremented automatically.

Interrupt enable flag is used to allow external interrupts, otherwise only non-maskable ones (such as unrecoverable hardware errors) are allowed. This is usually only accessible to the kernel of the operating system to prevent non-privileged programs to interrupt the CPU.

Trap flag can put the processor into a special single-step mode used primarily for debugging.

Table 4.1: FLAG register bits in an x86 processor

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
flag	-	-	-	-	O	D	I	T	S	Z	-	A	-	P	-	C

4.1.2 x86 Instructions

Instruction in assembly are written with mnemonics (usually three letters). They are most often followed by one or two operands, but can also have no operands. In MASM syntax the first operand for most instructions is the destination, while

the second one is the source. Operands can be registers, memory locations (presented as variables or as actual memory offsets) or literals, referred to as "immediate" operands in the manuals. Some instructions have specific limitations on what operand combinations they can work with. Operand sizes on an 80286 can be either 8 or 16 bits, which is sometimes infered from the context (for example the size of the register used), but can also be stated explicitly when referring to memory locations.

Memory manipulation instructions. One of the most common instructions is `mov`, short for *move*, which copies a value from the source operand to the destination operand. There is also the `xchg` (*exchange*) instruction which swaps the operands, with no need for a temporary memory location and is also an atomic operation. These instructions can be used for any segment, including the stack segment. There are also special instructions, `push` and `pop` that work with the stack directly and adjust the stack pointer as needed. There are also `pusha` and `popa` variants which copy all of the register values to and from the stack, and `pushf` and `popf` which work with just the flags register.

Arithmetic and logic instructions The processor supports a number of common operations on numbers and most of them also affect the flag register as appropriate. They can work directly with registers, memory locations, and immediate fields, but not simultaneously with two memory locations. Instruction `add` increases the destination operand by the value of the source operand. `adc` does the same, but also takes into consideration the value of the *c* flag, and is therefore often used to work with numbers longer than 16-bit. `inc` increases the destination operand by one. Analogous are the `sub`, `sbb` (*subtract with borrow*), and `dec` instructions. Multiplication of unsigned integer values is done with `mul`, and it takes a single operand which is then multiplied with `AX`. The result is stored in `AX` if the source was a byte, and in the pair `DX:AX` if the source was a word (two bytes). There is a similar `imul` instruction which works with signed integers. Division is done with `div`, which takes the current value from `AX` and divides it with the operand received, and stores the quotient and remainder pair in `AH:AL` or `DX:AX` depending on the size of the operand.

There are also the usual logic operations implemented, which act in the same way as the arithmetic ones, taking one or two operands and storing the result in the target operator. These are: `and`, `not`, `or`, and `xor`. Additionally there is `neg` which replaces the value with its two's complement.

A somewhat special variant are `test` and `cmp` which do not change the target operand, just set the flag registers like a logical "and" or a subtraction, respectively.

Control transfer instructions. Most of these instructions have a single operand which is the address in the *code* segment of the instruction that is to be executed next. There are unconditional jump instructions, such as `jmp` and its variants. Conditional jumps mostly depend on the values of flag registers, and for convenience sake some have multiple mnemonic tied to them. For instance, `ja` (*jump if above*) is the same as `jnbe` (*jump if not below nor equal*). The jumps and their conditions are listed in Table 4.2.

Table 4.2: Conditional Jump Instructions on an 80286 Processor

Mnemonic	Condition Tested	"Jump If..."
	Unsigned Conditional Transfers	
JA/JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNP/JPO	PF = 0	not parity/parity odd
JP/JPE	PF = 1	parity/parity even
	Signed Conditional Transfers	
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 0	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNO	OF = 0	not overflow
JNS	SF = 0	not sign (positive, including 0)
JO	OF = 1	overflow
JS	SF = 1	sign (negative)

Adapted from [*80286 Programmer's Reference Manual* 1987]

A special type of conditional jump is the `loop` instruction, which reduces the `CX` register by one and if it is not zero jumps to the give address. There are also variants of this instruction that will only jump if `CX` is zero and some other flag is not set. One example is `loopne` (*while not equal*) and `loopnz` (*while not zero*) which are the same instruction and additionally check the zero flag.

There are also unconditional jump instructions meant to be used in the contexts of procedures. The first one is `call`, which acts as a regular jump, except it first stores the current instruction pointer on the stack. The other one is `ret` which uses the value stored on the stack to return the execution to the original point. It has an optional single operand which is the number of elements that will additionally be removed from the stack.

The actual conventions of calls and returns can vary a lot between different operating systems and compilers. The procedure parameters can be passed on through the usage of registers, or stack, or a combination of both, where the first few parameters are passed in the registers and the remaining ones are on the stack. Such schemes are used by some Borland and Microsoft compilers. Cleaning up the passed parameters from the stack is also a matter of convention: for instance the `ret` instruction can be used by the procedure itself to do this, while in other conventions the callee is the one responsible for this after the procedure has ended. Similarly, the actual return value(s) are also subject to conventions. For instance many compilers, such as Borland Pascal, used different registers for different types of return values, while some others may use the stack for return values as well.

Interrupts. A somewhat different type of control transfer are interrupts. They are meant to handle exceptional situations, such as hardware problems, or special requests from the user. One use is with the `int` instruction which is meant primarily for system calls, such as interaction with hardware devices for input or output. The instruction has a single parameter with the number of the interrupt service routine to be activated. Often the value of `AX` (or a part of it, potentially other registers as well) is used to be more specific about the function being invoked.

Interrupt service routines are generally provided (at least partly) by the operating system. For instance `int 20h` is commonly used to finish the execution of a program and return control to the operating system. Interrupt service `21h` was commonly used to access DOS (*disk operating system*, common for IBM PCs) system calls. If `AH` was set to 1, a character would be read from the keyboard and returned in `AL`, if it was 2, a character with the code currently stored in `DL` would be written on the standard output. A whole string (starting at the address given in `DX` and ending with "$") can be output by setting 9 in `AH`. Reading a string could be performed

with `AH` set to `0Ah`. It would be stored into a *dos buffer*, which starts with two extra bytes, the first one representing the length of the buffer, and the other one the actual number of characters read, with no special char at the end. The starting address of this buffer is given to the interrupt in `DX`.

Other instructions. There is a number of direct string manipulation instructions, and a few that are meant to operate with addresses, and some that shift or rotate the bits in the operands. None of these will be covered here, since they were not the focus of the translation tool at this point.

Chapter 5

Bytecode and MicroJava Language

This chapter explains the benefits of virtual machines, and more specifically the Java Virtual Machine and the associated bytecode. It is then followed by an introduction to MicroJava, a subset language with its own virtual machine, that is later on used in this thesis as a proof of concept language for the translation and transformation tools presented.

5.1 Bytecode

In modern complex information systems there is a strong need for portability, between current systems and for future operating systems. One of the major solutions for these problems is to work with virtual machines that will always behave the same way if there is an implementation for the target system. Today the *Java Virtual Machine* is practically omnipresent – it is hard to find a system that is not able to run Java bytecode.

These advantages are not used only for the Java programming languages. Scala, Groovy, and Closure were developed with the intent to compile into bytecode. There are bytecode compilers for Python, Ruby, C, Lisp, Scheme, Pascal, Prolog, and many others.

Due to its popularity there is a lot of research that looks at various aspects of bytecode and how to improve it. Often there are direct bytecode changes that either add new functions to the existing program or try to make it work faster.

Some of these are industry "standards", such as Oracle Hibernate, and some will be discussed later in this section.

Bytecode generated by a Java compiler from "normal" Java programs can often be decompiled into code nearly equivalent to the original. There are several freely available tools for Java bytecode decompilation that can restore the original Java source code to varying degrees, such as CFR[1], Fernflower (which is built into IntelliJ Idea[2]), JD-GUI[3], Krakatau[4] and Procyon[5]. Of these listed, Krakatau has a slightly different approach since it just tries to construct a valid Java program based on the bytecode without presuming that the bytecode was compiled from Java. However, the success of any of these decompilers is not a given – they can have issues with various optimisations that are performed, or there might be additional code inserted by an instrumentation tool that adds new features seamlessly (persistence, network support, job management, monitoring, . . .). There is also the possibility that the bytecode was intentionally obfuscated to prevent decompilation, by introducing random names, dead code etc. This is mostly done with commercial products to protect intellectual property. More importantly, since many languages these days support the usage of JVM, it is quite possible that the bytecode was not generated by a Java compiler at all.

ASM is a general purpose framework for working with bytecode [Kuleshov, 2007]. It provides tools for end users, but the libraries are used by a large number of tools, including Jython, JRuby, Eclipse and some of Oracle's persistence systems.

Soot is another framework that provides several tools that work with different presentations of bytecode to make it easier to navigate in, optimise, and inspect the code. It is used in a number of research tools, and students' courses [Lam, Bodden, Lhoták, and Hendren, 2011].

Another reason to work directly with bytecode and make code injections is to improve performance measuring. *DYPER* is a tool that monitors the dynamic behaviour of a program [Reiss, 2008]. What makes it interesting is that it dynamically manages the overhead of such monitoring by not allowing the inserted code to go over a certain threshold. It also allows giving priorities to *proflets* that manage different things depending on the user's needs. *J-RAF2* (short for Java Resource Accounting Framework) tries to build a reliable cross-platform way of monitoring resources in a JVM [Hulaas and Binder, 2008]. *ByCounter* is another tool that focuses on counting executed instructions in bytecode. It does so by instrument-

[1] `http://www.benf.org/other/cfr/`
[2] `https://github.com/JetBrains/intellij-community`
[3] `https://github.com/java-decompiler/jd-gui`
[4] `https://github.com/Storyyeller/Krakatau`
[5] `https://bitbucket.org/mstrobel/procyon/overview`

ing the bytecode itself instead of applying changes to the virtual machine, making it therefore independent of the used JVM [Kuperberg, Krogmann, and Reussner, 2008].

A formal framework with a focus on verifying direct bytecode transformations with the goal of eliminating errors that might be introduced by the transformation is presented in [Lounas, Mezghiche, and Lanet, 2013]. The approach is based around embedDSU which works with Java Card applications, but the framework may be applied to others systems as well. The framework defines formal semantics for static analysis of code that can be used to verify that no type errors were introduced by transformations, and using Hoare triples and predicate transformations to check if the behavioural aspects of programs were modified with the changes.

Another interesting research line in improving the performance of compiled bytecode by solving some optimisations with instrumentations. For instance, there is the problem of flag variables, ones that are set at significantly different points in the code compared to the actual conditional jump that uses them. These constructs can make automated testing and coverage harder than it needs to be. One approach is to replace the Boolean values with integers (which are the same size in Java) that will give more information about jumps, while preserving the original semantics and structure of programs [Li and Fraser, 2011]. The experiments have shown that there is indeed performance to be gained from this approach when there are flag variables. Unfortunately, this occurs relatively rarely and applying it to all of the bytecode can add non-negligible performance overheads, so it should be used selectively.

GenProg, a system that uses genetic programming for automated repair, described earlier in Section 2.1.2 is also applicable to Java bytecode [Schulte, Forrest, and Weimer, 2010].

5.2 MicroJava

MicroJava is a language developed by Hanspeter Mössenböck for usage in compiler construction courses [Mössenböck, 2018], based on the book by Niklaus Wirth [Wirth, 1996]. The name is given because it is (for most intents and purposes) a subset of the full Java language, and therefore has no relation to Java Micro Edition (Java ME).

The language features concepts that are expected in a typical programming language with the syntax very similar to Java and the C family of languages. The focus is on having a minimal language to demonstrate the basic features of a compiler, without going into too much detail that can arise from a full production language.

It features two primitive types for integers and characters and supports defining custom types as inner classes. There are also reference types for arrays and objects. There is no inheritance and variables are static. An example of a MicroJava program can be seen in Figure 5.1. The complete syntax and definitions are given in appendix B.1.

```
program P
{
  void main()
  int i;
  {
    i = 0;
    while (i < 5) {
      print(i);
      i = i + 1;
    }
  }
}
```

Figure 5.1: MicroJava code ("while-print" program)

5.2.1 MicroJava Virtual Machine Specification

The compiler construction course developed around MicroJava includes a complete virtual machine specification. In essence it is very similar to Java Virtual Machine [Lindholm, Yellin, Bracha, and Buckley, 2015], but leaves out some of the details so that it can focus more on the concepts that students need to learn. This, in many ways, makes it perfect as a target for a proof of concept tool.

There are several differences compared to JVM. The operand addresses are fixed in the MicroJava Virtual Machine (MJVM), unlike JVM which uses a loader to resolve the names in the constant pool. There is a far lesser emphasis on encoding types in the instructions, the goal being to reduce the number of instructions. This change makes it harder to verify the consistency of the generated object file.

MicroJava virtual machine defines several memory areas. First is the *code* area, which contains the actual code of the methods. The *program counter* is therefore an index into this area that represents the currently executed instruction.

There is a *data* area that stores the variables of the main program. There is also a *heap* area that is used for dynamic allocation of objects and arrays. There are no commands for manual deallocation of the objects. The specification does

not include a garbage collector, so this version of the VM does no deallocations of the memory, although such a collector could be added.

There are two stacks specified – the *procedure stack* and the *expression stack* (*pstack* and *estack* respectively). The procedure stack is used for storing activation frames of the invoked methods – their local variables and return addresses. The expression stack stores the operands for the instruction. The parameters for method activation and return values are also passed through this stack.

The instructions of the MJVM can be divided into several groups, which will be shortly introduced with a few examples. A detailed specification of the instructions available is given in Appendix B.2,

In total there are 57 instructions defined for the virtual machine in the original 1999 specification. A later revised specification reduced this number to 54 with the removal of `inc`, `dup` and `dup2` instructions which were not used by the reference compiler. Internally the operands for the instructions can be a byte, a short int (16 bit) or a word (32 bit).

Constants and variables instructions. These are responsible for putting constants and variable values on the expression stack, or taking them from that stack. The first group of these instructions are `const_0`, `const_1`, ... `const_5` that put 0 to 5 on the stack, `const_m1` that puts -1 on the stack, as well as the general `const` instruction that has an operand. Then there are `load` and `store` commands that put local variables on the stack, or assign values to them from the top of the stack. Similar instructions exist for global variables, object fields and array elements.

Arithmetic instructions. All of these pop the required number of operands from the expression stack and push the result on the same stack. For instance `add` takes two numbers, and pushes the result to the top of the stack. There are other instructions for subtraction, multiplication, division, and remainder. Additionally, there is negation and byte shifting (both will return the result on top of the stack), and the increment operator which will change the value of a variable in place.

Jump instructions. There are six conditional jump instructions (equal, not equal, less than, less or equal, greater than, greater or equal), all of which take two operands from the expression stack, compare them and then perform the jump to the relative address given as an operand if the condition is satisfied. Additionally there is an unconditional jump that has no interaction with the expression stack.

Object creation instructions. Instructions `new` and `newarray` create new objects and new arrays, both of which return the address of the result on the top of the stack. The size of the object is taken as an operand, while for the arrays, the length is taken from the stack and the element size is given as an operand. Both of these structures are stored in the *heap* area of the memory.

Input and output instructions. Built directly into the virtual machine are instructions for reading from the standard input stream and writing to the standard output stream. There is `read` that will read an `int` value and store it on top of the expression stack. There is `print` which will take two values of the top of the stack, the second one being the actual integer that will be output, and the first one is the width in characters that is used for formating the output. Additionally there are `bread` and `bprint` that work with a single byte and interpret it as an ASCII character.

Stack manipulation instructions. Instruction `pop` removes the top element from the expression stack and discards it. Instruction `dup` duplicates the value at the top of the stack, while `dup2` duplicates the top two values on the stack (i.e. if it was "xy" before it will end up as "xyxy").

Method invocation instructions. There are four instructions related to method invocation. The first two are used for switching the execution to a method – these are `call` and `enter`. The first one stores the address that will be executed after the method returns and then changes the program counter to the start of the method. The second one, `enter`, creates the *activation frame* on the procedure stack based on the two operands it needs, which are respectively the number of parameters the method receives; and the number of local variables the method uses. The address of the previously active frame is stored on the procedure stack (for later restoration), before the space for variables is allocated. This instruction is also responsible for initialising the local variables to the default values and for copying the parameters from the expression stack. The second pair of instructions, `return` and `exit`, are responsible for restoring the execution flow after the method has finished. The first one, `return` is the opposite of `call` – it takes the address previously stored on the procedure stack and switches the program counter to it. The second one, `exit`, is responsible for dismantling the activation frame. In essence, this can be done with just the restoration of the frame pointer to its previous value. There is no need for deleting the actual values of local variables, since any new frame that can later on occupy the same space will overwrite these on initialisation anyway.

Trap instruction. Finally, there is the `trap` instruction, which is meant for run time error reporting. It has a single operand which is the error code, and it should stop the execution of the program.

Part II

Translation

Low-Level Code Translations

The general process presented in this dissertation has two separate steps: one is translation of the original code to *WSL*, while the other is transformation of that code using the built in features of WSL.

This part describes two tools that were created to translate two low level languages to WSL. Since the end result is made with the intent to be transformed, very little effort is expended on optimising the translations at this step in either of the tools. The main focus is in ensuring the correctness of the output, while being as verbose as needed, which in general results in long pieces of code. This verbosity will be eliminated in the second step of the process, during the transformations, as shown in Part III.

Modelling other low-level architectures is discussed in Section 9.2.

Chapter 6

Translating Assembly Code

As already explained, the process presented here consists of two basic steps: translation to the language WSL and then transformation of the obtained code. In the case of assembly language, a new tool was developed as part of this work, called *asm2wsl* which translates a subset of assembly code to *WSL*, trying to capture all of the aspects of the original with a complete focus on correctness and without much effort to reduce the verbosity of the results at this step. For the second stage, transformations can be manually applied, or we can use various transformation scripts written in *MetaWSL*, or the combination of the two approaches. This chapter will focus on the translation stage, while the transformations are explained in more detail in Part III, Chapter 8.

6.1 Translation Tool *asm2wsl*

Asm2wsl is a program that translates a subset of x86 assembly language in the *MASM* dialect (see Chapter 4) to *WSL* with the purpose of transforming these programs into more readable and maintainable versions of themselves [Pracner and Budimac, 2011a]. It is available on the project's web site[1] under the terms of GNU Public Licence. The main goal with this tool is to test the plausibility of translating all the aspects of the original code into high-level structures.

Earlier known tools developed to work with *FermaT* and *WSL* presented in a number of papers [Ward, 1999, 2000, 2004; Ward, Zedan, and Hardcastle, 2004], create additional files during the translation to WSL, which contain data about the

[1] `https://perun.pmf.uns.ac.rs/pracner/transformations`

variables and their mapping in the memory. These files are then used when the transformed and improved code is translated into (for example) *C* code and the appropriate pointer types are created.

An important practical advantage with the approach in *asm2wsl* is the possibility to run the translated programs with the WSL interpreter at any point in the transformation process and have run time verification of the correctness of these programs, while earlier it mostly needed to be translated back into some other language to achieve this. Thus, there is less reliance on a second translation tool, and at the same time the resulting WSL programs could be translated into any other output language without any need to know of the original assembly mappings. The crucial downside is that this is almost impossible to do in a general case and therefore limits the current version of *asm2wsl* to a smaller subset of assembly code.

To keep things simpler for these initial steps, the tool was made so that it presumes to work with a 16-bit 80286 type processor. As explained earlier in Section 4.1, the basics of the architecture did not change much in later processors, at least not in the sense of the concepts tested here.

The tool has been implemented in Java, making it platform independent. At its core, this is a line by line, single pass translator, with the focus on translating all the aspects and side effects of the original code, without considering size optimisation at this stage of the process. This generally results in programs that are significantly larger than the original assembly, but the goal anyway is to run automated transformations on this code, which can then also reduce the size of the code. The same principle was successfully used in already mentioned earlier translators which use WSL for transformations.

The translator features a few optional command line switches that influence the way the programs are handled, which can be seen when it is run with no parameters. One option is to include the original comments from the assembly code as comments in the translated version (which is turned on by default, but can be switched off). Another option is to additionally include the original assembly lines in comments next to their translated part, which can help understanding the workings of the code, and the translator itself. The program can be expanded with a dump of the internal state of the virtual processor at the very end of the code. This can be helpful to understand potential problems, but was mainly developed to be able to work with code segments that only worked with the processor without any results being sent to the "outside" world. Furthermore, there is a switch to change the way stack access operations are translated, which will be covered in more details later.

Assembly instructions are directly linked with the processor and its internal states, which is not common for high-level languages in general or WSL specifically. The most common program flow control operations depend on the previously

set flags in the processor, so one option to capture all the aspects of these commands is to create structures that will make a "virtual" processor. The registers from the processor are then represented as global variables, while bits from the flag register are all defined as separate variables (which they already are for most intents and purposes in the processor and are mainly combined into a single register to reduce space).

Assignments and arithmetic. Assignments are commonly done with the `mov` instruction, which copies a value from one address or register to another. There are some limitations on what combinations are allowed, but they are not checked by *asm2wsl*, since it assumes that the given code was checked and compiled earlier with an external assembler program. Since `mov` does not affect the flags, the translation is just an assignment as can be seen in Figure 6.1. WSL also provides a construct for simultaneous assignments, which means the `xchg` instruction can be translated without using an additional temporary variable.

The processor operates with numbers of different sizes, in the case of an 80286 these can be 8 or 16 bits. This is important in many situations, but one of the most important is with any of the instructions that can cause an overflow. To deal with this an additional variable, `overflow`, is introduced in the internal state of the virtual processor that the translations use. The value of this variable is set depending on the detected size of the target operand, although this is not always reliable. The adequate flag variable is then set as would be in the original processor, and any additional adjustments of the target value. Figure 6.1 shows this with examples of `add` translations. Later architectures added 32 and 64 bit sized numbers, which can be handled in the same manner, but would require translations to have more details. This is one of the reasons the tool presumes an older processor for the early versions.

Additionally, an *x86* processor allows independent access to *Low* and *High* parts of 16 bit registers (the 8 bit parts of them). To handle this, *asm2wsl* adds operations that get or set the adequate parts of the register, instead of some other means to directly access parts of the variable as memory. This also has to be taken into consideration for a lot of the instruction translations.

Other arithmetic instructions (such as `sub`, `inc`, `dec`) are translated with all of these details. Bit shifting instructions, `shr` and `shl` are translated as multiplications and division by 2. Most of the other instructions that work directly with bits are currently not supported since they were not featured in the initial samples. There are a few exceptions, for instance, a common practice is to use `xor ax,ax` as a faster alternative to `mov ax,0`, which is recognised by the translator.

```
mov ax, dx    →  ax := dx
xchg ax, dx   →  <ax := dx, dx := ax >
mov  ah, n    →  t_e_m_p := n;
                 ax := (ax MOD 256) + t_e_m_p * 256;

                 overflow := 65536;
                 dx := dx + ax;
                 IF dx >= overflow THEN
add  dx, ax   →      dx := dx MOD overflow;
                     flag_o :=1; flag_c := 1;
                 ELSE flag_o :=0; flag_c := 0;
                 FI;

                 overflow := 256;
                 t_e_m_p := (ax MOD 255) + 12;
                 IF t_e_m_p >= overflow THEN
add  al, 12   →     t_e_m_p := t_e_m_p MOD overflow;
                    flag_o :=1; flag_c := 1;
                 ELSE flag_o :=0; flag_c := 0; FI;
                 ax := (ax DIV 256)*256 + t_e_m_p;
```

Figure 6.1: *asm2wsl* translations – assignments and overflow handling

Labels and jumps. The main usage of labels in assembly code is as targets for jump commands instead of memory offsets. The *Action system* structure was designed exactly with these types of jumps in mind (see Section 3.3). Blocks of assembly between two labels are translated as a single action, and the name is taken from the label at the start. Thus any jump command can just use the same name as it would in the assembly code. Regular jumps in assembly are one direction, that is there is no return address to remember. Actions are designed to return to the call site by default, but also provide the option to instantly stop the execution of the whole system with a call to the predefined `Z` action, and therefore abandon any remaining returns. If the system is generated so that no action actually returns then it is called *regular*, which is exactly what *asm2wsl* generates. These types of systems are supported by transformations so they can commonly be replaced by high-level structured code. A call to `Z` is generated at the end of the program, and also at other places that are recognised to be special exits in the assembly (for instance, interrupts to return control to the operating system). To support the "normal" flow of the program to the next statement after a label, the end of each action includes a call to the next one.

Jumps are therefore translated with their internal semantics. In most cases, these are tests of one or several processor flags which were set earlier by some other

instruction. The target labels are unchanged in translation. Unconditional jump is just a single call. The translation of the `loop` instruction includes the `cx` register being decremented before the tests being performed and also needs to create a new "dummy" label to be able to continue the "normal" flow of execution once the loop is done (Figure 6.2).

```
je    exit       → IF flag_z = 1 THEN CALL exit FI;
ja    greater    → IF flag_z = 0 AND flag_c = 0 THEN CALL greater FI;
jmp   compare    → CALL compare;

                        ....
                   CALL theloop
                    END
                   theloop ==
    ....           POP(ax, stack);
theloop:           overflow := 65536; dx := dx + ax;
    pop ax       → IF dx >= overflow
    add dx, ax     THEN dx := dx MOD overflow; flag_o :=1; flag_c := 1;
    loop theloop   ELSE flag_o :=0; flag_c := 0; FI;
    ....           cx := cx - 1;
                   IF cx>0 THEN CALL theloop ELSE CALL dummy21 FI
                   END
                   dummy21 ==
                        ....
```

Figure 6.2: *asm2wsl* translations – examples of jumps and label handling

Arrays. In assembly there are no real explicit arrays, just memory offsets. In other words, everything can be an array of memory locations. However, programs are often written with the idea of a "variable" in the data segment marking the start of the array, with the appropriate number of consecutive addresses occupied either with explicit numbers (or letters), or by a construct such as `dup`. Then the more or less standard syntax of accessing the elements with `arr[i]` is used. *asm2wsl* supports some forms of declaring arrays, mainly the one with the predefined values. It also does automatic adjustments to the indexes when accessing offsets, since arrays start from 1 in *WSL*, while offset 0 is the first element in assembly.

Stack. The processor's main stack is implemented as a global list. The `pop` and `push` commands take and put elements on the start of this list directly. No additional checks (such as element size and compatibility, presence of elements on the stack, ...) are performed at this point, with the presumption that the original

code used the stack in a correct way. There is a command line switch to change the translation of these in WSL as either `POP/PUSH` (the default), or as `HEAD/TAIL` operations with the list.

Macros. In the current version of *asm2wsl* if macro definitions are encountered they get copied as comments in the resulting program, with appropriate warnings given to the user. This is mainly because the actual definition can have a number of complex features with details that can be dependant on the assembler at hand. On the other hand, the macro call feature is used as a convenient abstraction for some actions. More specifically, some special names are recognised and translated directly to WSL equivalents. These are mainly input and output operations that would otherwise use interrupts (which are hard to translate). The main examples of these names and translations are given in Table 6.1. In general, this feature can be used to separate whatever hardware specific operations would be hard to translate to WSL, while still giving executable code and placeholders for eventual translation to some third language if needed.

Table 6.1: Translation of some special macro names

Assembly	**WSL**
`print_str x`	`PRINFLUSH(x);`
`print_num x`	`PRINFLUSH(x);`
`print_char x`	`PRINFLUSH(x);`
`print_new_line`	`PRINT("");`
`read_str x`	`@Read_Line_Proc(VAR x, Standard_Input_Port)`
`read_num x`	`@Read_Line_Proc(VAR t_e_m_p, Standard_Input_Port);` `x := @String_To_Num(t_e_m_p);`
`end_execution`	`CALL Z;`

Procedures. The tool has support for translating procedures from assembly. They are translated as a nested action system inside a single action that corresponds to the label with the procedure name. This makes all of the labels inside a procedure contained in the inner action system and makes it possible to end the execution of the procedure with `CALL Z`. It also has the convenient consequence that the

names can be the same as some of the ones in the outer system. The return to caller location is handled within the outer action system – the action presenting the procedure (unlike the regular ones) does not end with a call to the next action, but returns. This also removes the need to decode the return address from the stack.

The existence of actions that return to the call site does change the type of the main action system to *hybrid*, instead of *regular*, when none of the calls returned. However, this can be handled with transformations later on. The inner action system is still regular and should get simplified and potentially replaced with a procedure or function if needed. That will make the main system regular again and enable all of the transformations as usual.

An example of a whole translated program is provided later on in Section 8.1.1 together with the appropriate transformations.

Further discussions about modelling other instructions and architectures are covered in Section 9.2, as well as how these can then be transformed by this process.

Chapter 7

Translating MicroJava Bytecode

This section shows the main details about how bytecode can be translated to WSL. The main ideas is to get WSL code that will behave exactly the same as the original bytecode, on the same level of abstraction with all the intermediate internal steps (such as stack states, etc). This version of the program can then be automatically or manually transformed into higher level human readable code by using formal transformations and restructuring in *MetaWSL*, shown later in Part III. This type of approach has already been proven to be successful with restructuring industrial legacy assembly code [Ward, Zedan, and Hardcastle, 2004], and in the development of another assembly tool in this dissertation (Section 6.1). The results presented here confirm that the same can be used for bytecode.

7.1 Translation Tool *mjc2wsl*

The translation tool developed as part of this thesis is called *mjc2wsl* and is available on the project's web site[1] under the terms of GNU Public Licence. The tool was developed in the Java programming language and can be run on any platform that has Java support. The basic structures used in the translator will be explained first, followed by the main groups of instructions and how they are translated.

[1] `https://perun.pmf.uns.ac.rs/pracner/transformations`

The input file for this tool is the binary object file generated by a MicroJava compiler adhering to the format given in the specification (Appendix B.3). The headers at the start of the file specify the code size, the data segment size, and the starting address for the program, followed by the actual code instructions and operands encoded as bytes. A binary version of such a compiler is distributed with the tool itself and is used in the automated testing.

To translate the operations on the same level of abstraction, with the same operational semantics, the translator creates a "virtual" MJVM which contains all variables, lists, and arrays the real one would. In MJVM there are no operations that access individual memory addresses, there is only access to individual variables or objects or array members, so there is no need to simulate the memory as a byte array. Instead, there are several structures in place that emulate the storage of objects and arrays. Some of these have more than one way that the tool can translate them, influenced by command line switches.

The *expression stack* is simulated straightforwardly, with a list that has elements added or removed from its start, called `mjvm_estack`. Most instructions work with the expression stack, to get their input, to store their results, or both.

For all of these stacks, the translation of the interactions with the list can be translated in two ways, depending on switches selected. One option is to use `HEAD` and `TAIL` commands, and the other one is to use the more direct `POP` and `PUSH` commands.

The *data* segment that holds the global (static) variables is an array of words in the specification, and is simulated as such – a straightforward array, whose size is known directly from the object file.

The *procedure (method) stack* is simulated with a list called `mjvm_mstack` and is used for *activation frames*, which are local variables for the procedures. The local variables in the MJVM are therefore indexes in this stack starting from the current frame. Since there are no instructions to access this stack directly, but only to access single local variables, they are simulated on a higher level of abstraction. There are two options, depending on the switches selected when running the translator. The first one is a to use an array for the current local variables, basically the currently active frame. On entering a new procedure, the current array as a whole is stored in the list that emulates the procedure stack and a new array is created for the new frame. On exiting a procedure the process is reversed. The second option is to create automatically named variables, one for each of the local variables. On entering a procedure the current local variables are stored one by one on the procedure stack, while on exiting the procedure, the variables are restored from the said stack. Both of these approaches never actually use the method stack for the currently running procedure, but only for the suspended ones.

In MJVM arrays and objects are both dynamically allocated in the *heap*, with the start address being the reference for commands that work with the fields of objects or elements of arrays. According to the specification, blocks should be just sequentially allocated and no garbage collection is done (although one could be added without changes to the program semantics). In the environment simulated by *mjc2wsl*, there are separate lists made to store the dynamically allocated arrays and objects. References to them are then the indices into the list where the individual objects are, instead of the start address in the heap. The appropriate commands that work with these structures (i.e. getting an array element) are translated accordingly. In the current version of the translator the size of the heap is not tracked, meaning that it is virtually limitless. This means that programs that would originally throw heap overflow errors would not do the same in the translated version, at least not at the same point. On the other hand, overflow errors and such abnormal termination of programs are not currently the focus of this research.

The translator relies on the *action system* special structure in WSL that is made exactly to cope with low-level code and jumps (Section 3.3). Bytecode instructions are translated into their own actions with names that represent the original address of the instruction. This enables the handling of jumps in the code that are characteristic to low-level programming and are expressed as address changes in the compiled bytecode. The last statement in a typical action will therefore be a call to the next action, as would happen in the virtual machine. The main exception to this are jump instructions which of course change the "normal" flow of the instruction pointer to the next line. The first instruction will always be `a14`, since the binary file will have the first 14 bytes are the headers, and the addresses are 0 indexed. The starting address of the program is directly translated as the start action name in the action system.

Instructions were already described in Section 5.2.1, and a detailed specification is available in Appendix B.2. The following text will show examples of translations for these groups of instructions, with some reminders of how they work.

The first group of instructions are for loading constants, and for loading and storing variables (both global and local) and object fields. For instance, `const_0` puts a "0" on top of the expression stack. The translation for this operation is straightforward: a single push to the appropriate stack with the appropriate value (shown as part of Figure 7.1). Operations for storing values work the same, except that there is a pop instead of the push. Some of these translations can be influenced by command line parameters, specifically the treatment of local variables.

The next group are the arithmetic instructions. All of them pop the required number of operands from the expression stack, and push the result on the stack. These are all translated in a straightforward manner. For instance, `add` takes two

numbers and pushes the result to the stack, as shown in Figure 7.1. The other instructions are the usual mathematical operations that work in the same way: subtraction, multiplication, division and remainder. There are also negation and byte shifting (which will return the result on top of the stack), as well as the increment operator which will change the value of a variable in place.

There are six conditional jump instructions (equal, not equal, less than, less or equal, greater than, greater or equal) which all take two operands from the expression stack. Depending on their relation a flag is set that determines if the jump will occur with a call to a specified action, otherwise the next action is called (shown in Figure 7.1 for `jge`). It is vital that the jump to the next action happens only if the condition is not fulfilled due to the recursive nature of the action system. Otherwise if the condition held true, the first (true) jump would happen, eventually there would be a recursive return and a fall through to the second (false) jump which should not happen. Additionally there is an unconditional jump that has no interaction with the expression stack. Translation of this jump also needs to take care to not include the unconditional jump to the next action after the "proper" jump.

```
C:"#15 (const_0)";
PUSH(mjvm_estack,0);

C:"#23 (add)";
VAR < tempa := 0, tempb := 0, tempres := 0 > :
POP(tempa, mjvm_estack);
POP(tempb, mjvm_estack);
tempres := tempb + tempa;
PUSH(mjvm_estack,tempres);
ENDVAR;

C:"#48 (jge) -- complete action; a24 is the next action";
 a21 ==
VAR < tempa := 0, tempb := 0 > :
POP(tempa, mjvm_estack);
POP(tempb, mjvm_estack);
IF tempb >= tempa THEN mjvm_flag_jump := 1 ELSE mjvm_flag_jump := 0 FI;
ENDVAR;
IF mjvm_flag_jump = 1 THEN CALL a34 ELSE CALL a24 FI;
SKIP
 END
```

Figure 7.1: Examples of translation of bytecode instructions (given in the comments) to WSL

Method calls in MicroJava bytecode consist of two instructions – `call` and `enter`. They are translated in a relatively straightforward way. The first command stores the caller address and jumps to the appropriate address. This will be translated with a call to the right action, without the need to explicitly store the caller address. This is possible because the presented tool generates *recursive action systems*, and therefore the WSL run time will take care of the addresses (unlike some of the assembly approaches which generated *regular action systems* that never return and need additional jumps for this). The second instruction (`enter`) creates the activation frame on the *procedure* stack and picks up the appropriate parameters from the *expression* stack to store them in the local variables. All of these steps are generated as the translation, with some variation depending on the currently selected mode for local variables.

Method returns also consist of two instructions – `exit` and `return`. The first one removes the activation frame, i.e., reverts what the adequate `enter` did. The second one returns to the caller address, which will be translated to an empty statement, since the action system itself will take care of returning to the appropriate place.

The MicroJava Virtual Machine also specifies instructions for direct input and output from the standard ports. They are translated with the normal print and read procedures, with some additional attention to the conversions of types as needed. Printing commands can also be influenced with a command line switch to specify whether all the printing boiler plate is done inline, or if it is separated into a procedure. Since version 0.2 of the tool, this is just two statements, so the impact on the end code is now not as significant, but this is due to improvements in FermaT. Earlier versions had more code to deal with the specific of the formating.

As already stated, at this stage the main goals are just translating the low-level behaviour of the original code. The size of the result is of little concern, since the application of transformations will remove the redundancies. A single bytecode operation will practically never be translated to a single WSL command (on average it is 3-4 statements). Additionally there are commands for the action system and helper procedures, making the result up to 10 times as long as the original. For instance the program in Figure 7.2 has 12 lines of code in its original Micro Java version, only 7 of those are actual statements and declarations, it gets compiled into 16 bytecode operations, and is then translated into around 65 WSL statements spread over around 130 lines of code (the precise size depends on the parameters used).

The current version of the translator supports all of the instructions in MJVM with almost all of their low-level internal logic. This Virtual Machine was (intentionally) designed with limited options for direct manipulation of individual memory

```
program P
{
  void main()
  int i;
  {
    i = 0;
    while (i < 5) {
      print(i);
      i = i + 1;
    }
  }
}
```

```
14: enter 0 1
17: const_0
18: store_0
19: load_0
20: const_5
21: jge 13 (=34)
24: load_0
25: const_0
26: print
27: load_0
28: const_1
29: add
30: store_0
31: jmp -12 (=19)
34: exit
35: return
```

Figure 7.2: MicroJava code and the translated bytecode ("while-print" program)

addresses. This means that most of the abstractions introduced by the translator should never be noticed. Valid bytecode outputs created by the compiler distributed with the tool should always be translated into correct programs, with the minor differences noted before (mainly the unlimited heap). Similar can be said for any valid bytecode programs, generated with different compilers or written by hand. In an extreme case, an example could be constructed to misuse the array access (or some other appropriate operation) in such a way to work against the introduced abstractions and therefore create potentially invalid programs. On the other hand, such an example would be opposing the design principles of the virtual machine itself.

7.1.1 Overview of Translation Variants

As was already mentioned, the translation results can be influenced by command line switches to a greater or smaller degree. The effects of some of these switches were covered at the appropriate structure and instruction translations, but are gathered here for a practical overview.

Local variables `--genLocalsAsArray` or `--genLocalsSeparate`: the first option generates an array of local variables that are accessed by their index; these are stored as a whole on the procedure stack. The second option generates separate names for all the local variables; 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.

Print procedure `--genProcedurePrint` or `--genInlinePrint` will either create a single procedure for printing to the screen, or make the code for printing inlined directly at the print sites. Differences in these two options since version 0.2 are very minimal, due to improvements in how printing is handled in WSL made during the writing of this thesis. The generated print procedure in the current version consists of only two statements. Earlier versions needed more code to handle some formating specifics.

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.

Code execution tracking. Several additional options can be used to get a much more detailed output of the internal workings of the virtual machine. One is printing all changes of the expression stack, another is to print the original addresses of the instruction being executed. There is also a simulated "trap" instruction that pauses after each instruction. These were mostly intended for testing the translator, but also for testing the code transformations in later steps of the process. More details can be found in the documentation of the tool.

7.1.2 Influence of Some Switches on the Metrics of Translated Programs

As already stated, the translation process can be influenced by switches. In this section we will give an overview of how some of these influence the metrics (introduced in more detail in Section 3.5). The experiments were run using *mjc2wsl* version 1.0.0, FermaT 18c (internal version number), and sample set *alpha-mj*. The focus was on three binary switches, meaning 8 versions of programs. For simpler references in future text, variants of programs will be marked with two letter shorthands, all of which are shown in Table 7.1. Versions of programs are therefore marked with three two letter codes, in the order given in the table. All of these

translated versions together will be referred to as *alpha-wsl-v8*. Later on, in the transformation part (Section 8.4.1) the influences of these switches on the end results and the whole transformation process will be discussed.

Comparison will use percentage differences to the best results, i.e., the difference between the value and the best value for that sample and metric, divided by the best value. This is done to normalise the differences and make them more comparable, since they can vary greatly between samples, but also between metrics.

Table 7.1: Abbreviations for various parameters used

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

McCabe's Cyclomatic Complexity is affected only by the switch for the temporary variables, with the local versions often having higher values for the metric. This is due to additional jump flags that need to be set and checked later on, outside of the `VAR` block when the variables are no longer available. On average the increase is 27.25%, yet it can go as high as 66.67%. In rare cases of samples with none of these characteristic jumps, there are no differences. This results in a high standard deviation of 23 percentage points. It is natural that the different types of access to the stacks or the different storage of local variables are not affecting this metric.

McCabe's Essential Complexity is the same for all combinations of the switches. This is to be expected – the additional `IF` statements are not counted, as they will be grouped into the same single-entry-single-exit block.

Number of statements reports lowest results with the combination of *pp-gl* switches (Table 7.2). The final switch (*ar* or *sp*) does not influence the results much, mostly just a few percent between them. Which one of these is better is dependant on the sample. 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 differences in the first switch is because `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 before mentioned jump flags. Finally, the third switch depends on the sample at hand – sometimes extra handling of the individual local variables at the start and end of procedures will be more expensive than the array operations need for the other versions.

Table 7.2: *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 is affected in similar ways as the number of statements, but with greater increases of the values in some cases. 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%. Compared to the number of statements, *CFDF* is more sensitive to the `HEAD/TAIL` variation since it introduces more list operations.

Size metric (of the abstract syntax tree) has a clear winner on all of the samples: *pp-gl-sp*, as shown in Table 7.4. Next up is *pp-gl-ar*, which is on average 8% worse ($\pm$ 2.5 percentage points). As was the case with *statements* and *cfdf*, the other groups based on the first two switches follow it, but the margins are bigger and the differences between the *ar* and *sp* versions are also bigger. In short, *pp-lo* is about 30% worse, *ht-gl* 50–60%, while *ht-lo-sp* is at 77% and *ht-lo-ar* at 85%. The differences are more pronounced because there are more generated nodes in the AST that earlier metrics would ignore.

Structure metric has very similar numbers as the *size* metric (mostly just a few percentage points higher), which means that again all of the samples have their best results in *pp-gl-sp*, as can be seen in Table 7.5. Similarities are due to this

Table 7.3: *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

Table 7.4: *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

metric being a weighted sum of the components of the program, and naturally higher weights generally mean more nodes in the abstract syntax tree.

Table 7.5: *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 there is a trend that metric numbers are strongly grouped by the switches used, with the highest difference in the *ht/pp* switch, followed by the *gl/lo*. The *sp/ar* also shows differences, but sometimes they are not very significant. For most metrics the lowest values are achieved with *pp-gl-sp*, usually closely followed or sometimes beaten by the *pp-gl-ar* variant. However it should be noted that the numbers themselves do not always represent the actual readability of these translations. For instance, although the *gl* variants usually have better 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 (Section 8.4), it is more important how these switches will influence that process and the transformed end results.

7.1.3 Verification of The Translations

Additional verification of the correctness of translations is done by comparing run time behaviours with the MicroJava programs, for which an automated testing facility is included with the source files. All of the samples are run as the original MicroJava Bytecode and as the translated WSL, with a preset collection of inputs, and the outputs of both versions are recorded. These are then compared, and the differences, if any, are highlighted.

Translations made using *mjc2wsl* version *1.0.0* with all the variations listed above are almost always a perfect match. The only differences in the outputs of

all the tests on sample set *alpha-mj* and *alpha-wsl-v8* using *FermaT 18c* (internal version number), were related to details in exception outputs and buffering handling when reading single characters – the implementation of MJVM stores the line breaks in the buffer and the next read will pick that up.

Part III

Transformation

Chapter 8

Code Transformations

Program transformation can be defined as any operation that takes a computer program and generates another program [Ward, 1989]. The WSL language enables users to work directly with loaded programs and change them in various ways using the built-in capabilities of *MetaWSL* (Section 3.4) and the vast catalogue of semantics-preserving transformations that are supplied with FermaT. In particular, this work is interested in WSL programs that present low level programs and trying to rework them into high-level code. The main examples of this is code that is obtained by the presented tools, *asm2wsl* and *mjc2wsl*, which will take all the little details and inner workings of an x86 processor or a MicroJava Virtual Machine and present it as WSL. It should be noted that the presented work is not limited nor made specifically to handle this code, but should work on general WSL programs.

Transformations can be selected by hand and applied manually, using command line tools, or by using the freely available visual tool *Fermat Maintenance Environment (FME)* [Ladkau, 2009]. In FME, all the transformations available are listed in a sidebar and the user can easily select the piece of code to apply them to. This can be of use both to perform the whole process of transforming code or to experiment with the possibilities. Another option that should still be considered manual is to write a specific MetaWSL program that will transform a determined input program. Manual approaches are further shown in Section 8.1.

Transformations can also be automated by writing various scripts that will take input programs and use different heuristics to apply individual transformations to the program as a whole, or to specific parts of it. These can vary wildly in the complexities and can range from some basic pre-processing, running scripts for specific types of programs, or fully automated processes that should work on any

input. Basics are shown in Section 8.2, with the following sections containing specific experiments using a hill climbing algorithm for the automations.

Described methods can be freely combined. For instance, an automated script can be run on the code and then the result can be examined in FME, with some additional modifications made by hand (such as specific cases that the script did not detect). The result of that could then be given to the same script for further improvements, or to a different script that adapts the program to some other needs. This is especially useful when new types of programs are being processed and from which the script itself can be improved.

8.1 Manual Transformation of Translated Code

Manual transformations, in this context, are those which are chosen by a human user. These can be applied manually, using command line tools, or by using the freely available visual tool Fermat Maintenance Environment [Ladkau, 2009]. This tool lists all the transformations available in a sidebar and enables the user to easily select the piece of code to apply them to. This can be of use to both perform the whole process of transforming code, or to experiment with the possibilities and get more familiar with the system. The obvious advantages of hand picking transformations is that (with enough experience) practically any system can be simplified to a satisfactory size. The obvious problems are that it takes a lot of time to work on large programs, or large numbers of programs, and the aforementioned need for experience with the system to use it.

The next section will demonstrate an example how a relatively small program can be transformed by hand.

8.1.1 Manual Transformation Example – GCD

This section will examine a greatest common divisor program written in assembly – a direct implementation of the Euclidean algorithm without input or output to the screen (Figure 8.1). The algorithm itself is considered to be among the first ones ever described. The original version takes two numbers and simply subtracts the smaller from the larger one until they are equal, which marks the end of the process and either one of these can now be returned as the result. Later improvements to the algorithm use remainders to speed up the process in cases where the numbers are significantly different in size.

Translation into WSL using *asm2wsl* (described in detail in Section 6) produces a program with about 40 statements (Figure 8.2) out of the original 10 statements

```
model small
.code
            mov ax,12
            mov bx,8
compare:
            cmp ax,bx
            je  theend
            ja  greater
            sub bx,ax
            jmp compare
greater:
            sub ax,bx
            jmp compare
theend:
            nop
end
```

Figure 8.1: GCD example assembly code

– which is an expected increase, since all the aspects of the assembly commands are translated with a virtual processor in mind, with all the registers and states emulated. As previously explained, the main structure that enables the translation of "go to" type of jumps common in assembly are action systems, which are sets of parameter less procedures (Section 3.3). Action system execute until all the calls return, or a special predefined action `z` is called (which ends the execution of the whole system). The translator mostly relies on this second way and generation of *regular* systems.

While trying to figure out the inner workings of an action systems it can be very useful to generate a call graph, an option that is built into FermaT and is also easily accessible from FME. For this sample it will produce a diagram like in Figure 8.3. This makes is easier to visualise the connected parts of the program and the "busy" parts of it, with many inputs or outputs. On larger diagrams, this gets progressively more useful.

On a program like this, with a lot of variables, a good choice is to apply the *Remove Redundant Vars* transformation, which in this case will take care of unused processor registers and produce the program given in Figure 8.4, with 36 statements, but also some structures removed.

As explained earlier, translation introduces a lot of flag variables, and some have remained in the program. These can be removed quite efficiently with the *Flag Removal* transformation which seeks to replace the variable tests with the

```
VAR < ax := 0, bx := 0, cx := 0, dx := 0,
        si := 0, di := 0, bp := 0, sp := 0,
        ss := 0, ds := 0, cs := 0, es := 0,
        flag_o := 0, flag_d := 0, flag_i := 0, flag_t := 0,
        flag_z := 0, flag_s := 0, flag_p := 0, flag_a := 0,
        flag_c := 0, overflow := 256, stack := < >, t_e_m_p := 0 > :
VAR< skipvar := 0 >:
SKIP;
ACTIONS A_S_start:
  A_S_start == ax := 12; bx := 8; CALL compare END
  compare ==
    IF ax = bx THEN flag_z := 1 ELSE flag_z := 0 FI;
    IF ax < bx THEN flag_c := 1 ELSE flag_c := 0 FI;
    IF flag_z = 1 THEN CALL theend FI;
    IF flag_z = 0 AND flag_c = 0
      THEN CALL greater FI;
    IF bx = ax THEN flag_z := 1 ELSE flag_z := 0 FI;
    IF bx < ax THEN flag_c := 1 ELSE flag_c := 0 FI;
    bx := bx - ax;
    CALL compare;
    CALL greater END
  greater ==
    IF ax = bx THEN flag_z := 1 ELSE flag_z := 0 FI;
    IF ax < bx THEN flag_c := 1 ELSE flag_c := 0 FI;
    ax := ax - bx;
    CALL compare;
    CALL theend END
  theend ==
    SKIP;
    CALL Z;
    SKIP END ENDACTIONS ENDVAR ENDVAR
```

Figure 8.2: GCD program translated to WSL

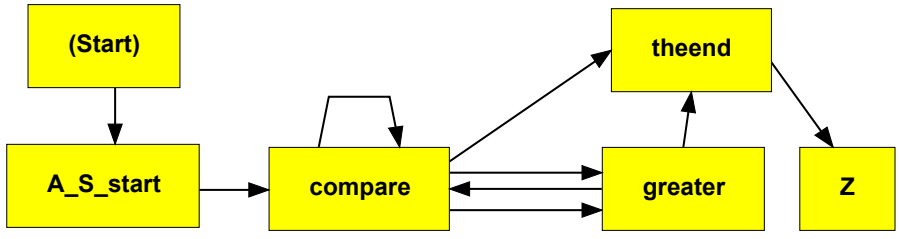


Figure 8.3: Generated call graph for the GCD program

```
VAR < ax := 0, bx := 0, flag_z := 0, flag_c := 0 >:
ACTIONS A_S_start:
  A_S_start == ax := 12; bx := 8; CALL compare END
  compare ==
    IF ax = bx THEN flag_z := 1 ELSE flag_z := 0 FI;
    IF ax < bx THEN flag_c := 1 ELSE flag_c := 0 FI;
    IF flag_z = 1 THEN CALL theend FI;
    IF flag_c = 0 AND flag_z = 0
      THEN CALL greater FI;
    IF ax = bx THEN flag_z := 1 ELSE flag_z := 0 FI;
    IF ax > bx THEN flag_c := 1 ELSE flag_c := 0 FI;
    bx := bx - ax;
    CALL compare;
    CALL greater END
  greater ==
    IF ax = bx THEN flag_z := 1 ELSE flag_z := 0 FI;
    IF ax < bx THEN flag_c := 1 ELSE flag_c := 0 FI;
    ax := ax - bx;
    CALL compare;
    CALL theend END
  theend == CALL Z END ENDACTIONS ENDVAR
```

Figure 8.4: GCD – *Remove All Redundant Variables* transformation applied

expressions that were assigned to those variables. Application of this transformation reduces the size of the code to 19 statements (Figure 8.5).

Action systems of appropriate forms can be transformed into endless loops, with the calls to the special `Z` action that would end the system being replaced with exits from the loop (Figure 8.6). These exits can be used by the *Floop to while* transform to convert the endless (*forever*) loop into a `WHILE`. *Constant propagation* can be used to initialise the variables to the proper values from the start. In the case of this example, the end result (Figure 8.7) is a very clear high-level version of the Euclidean algorithm in only 5 statements.

Since this example used predefined values for the input, unlike a more "normal" program, the output is also predefined. The program can then be further simplified by combining *Unroll loop* and *Constant propagation* (the number of applications for different start values could be higher) and produce the following code, with no calculations in it:

```
VAR < ax := 12, bx := 8 >: ax := 4; bx := 4 ENDVAR
```

Going a step further, this code has no interactions with the "outer" world, so additional simplifications can lead to a valid replacement of the whole program with

```
VAR < ax := 0, bx := 0 >:
ACTIONS A_S_start:
  A_S_start == ax := 12; bx := 8; CALL compare END
  compare ==
    IF ax = bx
      THEN IF ax < bx THEN CALL theend ELSE CALL theend FI
      ELSE IF ax >= bx THEN CALL greater FI FI;
    bx := bx - ax;
    CALL compare;
    CALL greater END
  greater ==
    ax := ax - bx; CALL compare; CALL theend END
  theend == CALL Z END ENDACTIONS ENDVAR
```

Figure 8.5: GCD – *Remove flags* applied

```
VAR < ax := 0, bx := 0 >:
ax := 12;
bx := 8;
DO IF ax = bx
     THEN IF ax < bx THEN EXIT(1) ELSE EXIT(1) FI
     ELSE IF ax >= bx THEN ax := ax - bx ELSE bx := bx - ax FI
   FI OD ENDVAR
```

Figure 8.6: GCD – *Collapse Action System* applied

```
VAR < ax := 12, bx := 8 >:
WHILE ax <> bx DO
  IF ax >= bx THEN ax := ax - bx ELSE bx := bx - ax FI
OD ENDVAR
```

Figure 8.7: GCD – *FLoop to WHILE* and *Constant propagation* applied

a single SKIP statement. Of course this is all a consequence of the artificial example cut out of context that did not do anything with the results of its calculation. Adding any sorts of prints or passing the result in a different way to the "outer world" would stop the process at the previous step.

The end result of this process that started from a translation of assembly code is a clear and concise version of the original algorithm. In a more extreme case, the final version is just an output of the end result, since the start values were hard coded. The number of transformations applied was also low, but this was the "optimal" path based on experience from a lot of trial and error on different programs. While an application like FME can help a novice to find all the applicable transformations, it can still be daunting to pick the right ones.

8.2 Automated Transformation of Code

Manual selection and application of transformations can give excellent results, but is time consuming and requires a decent amount of knowledge and experience for more complex systems. The more interesting option for restructuring is to automate the whole process as much as possible, by writing scripts in *MetaWSL*.

For instance, a script can be written around the idea of using just transformations that will simplify the code or move it to a higher level of abstraction (there are transformations that go the other way). An example of a script like this is distributed with *mjc2wsl*. The main transformations in this process are the ones that simplify the action system and then try to remove it, which typically results in a straight block of statements or a DO/OD loop that has exits in the middle. The next step is to transform such a loop, if it exists, into a more structured WHILE loop. Other transformations that this script tries to apply include constant propagation and removal of redundant commands and variables. The end result after this script will usually not be a fully restructured program, but it does remove some of the manual efforts.

A natural expansion of this idea is to try to use the same type of transformations, but to apply some of them in loops while there are any changes to the program at hand. This enables the combination of the applicable transformations in various orders. The selection of these transformations is more important now, since there is a possibility of infinite loops if transformations that undo each other are used. The version that is distributed with *mjc2wsl* uses mostly the same transformations like the previous script and can lead to slightly better end results.

The main approach used in the experiments in this thesis is slightly more "intelligent" and is based around the definition of a *fitness function* that will evaluate

the programs and make them comparable. In a simple case the fitness could be the number of statements, on the other hand it could be any combination of weighted metrics. Using such a function, the script tries to apply transformations to the program (or parts of the program) and checks whether the result is more desirable and continues the process as long as possible. This approach is generally known as *hill climbing*. It is, of course, much more expensive to run than the simple scripts, and it is quite dependent on the selected fitness function. It can give very good results, as will be shown in the following sections. The implementation used, developed in cooperation with Dr. Martin Ward, is simply called `hill_climbing.wsl`, and is given in whole in Appendix D.1. It uses a set of hand picked transformations to try and improve the program. Once all of them are exhausted, the script will test combinations of two transformations to check for improvements, and then again revert to single ones. This version uses the custom *structure* metric as the fitness function, which has proven to be good in practice. However, it is an open question what types of functions could lead to potentially better results. The script saves intermediate versions of programs in a temporary folder, while also generating a detailed log of all the transformations tested, which ones were successful and how many there were in total. All of these features can be used for detailed analysis of the process.

8.3 Hill Climbing Approach on Assembly Samples

This section will first present a few examples of transformations of single samples, followed by an overview of the metrics changes in the whole process for sample set *asm-a* (Section 8.3.2).

8.3.1 Examples of Automatically Transformed Samples

Using the same GCD sample program that was manually transformed in Section 8.1.1, the hill climbing process will automatically come up with an identical end result on the same level of abstraction – that is, a single `SKIP` statement. Once a `PRINT(ax)` was added to the code, the end result is a `WHILE` with an `IF`, the same as in Figure 8.7. This was achieved with 24 selected transformations, listed in Figure 8.8, which is more than in the manual approach, but these are mainly small transformations, cheap to test and execute. The whole process applied 2109 transformations, about half of which were tested after the last successful one.

Since the input is predefined, the simplest possible version of this program would be a single statement (`PRINT(4)`). This is not achievable with the current script

```
1001: Success:Absorb Right: at <3,2,2,2,2,2,2,2,5>
1002: Success:Add_Loop_To_Action: at <3,2,2,2,2>
1003: Success:Add Assertion: at <3,2,2,2,2,2,2,2,1,1,9>
1004: Success:Collapse Action System: at <3,2,2,2,2>
1005: Success:Add Assertion: at <3,2,2,2,3,1,1,1,8>
1006: Success:Constant Propagation: at <>
1007: Success:Delete Item: at <1>
1008: Success:Delete Item: at <1>
1009: Success:Delete Item: at <1,2,1>
1010: Success:Constant Propagation: at <>
1011: Success:Constant Propagation: at <>
1012: Success:Delete Item: at <1,2,1,1,3,1,2,1>
1013: Success:Delete Item: at <1,2,1,1,3,1,2,2>
1014: Success:Delete Item: at <1,2,1,1,4>
1015: Success:Absorb Right: at <1,2,1,1,3>
1016: Success:Constant Propagation: at <>
1017: Success:Align Nested Statements: at <1,2,1,1,3,2>
1018: Success:Align Nested Statements: at <1,2,1,1,3,2>
1019: Success:Delete Redundant Statement: at <1,2,1,1,1>
1020: Success:Delete Redundant Statement: at <1,2,1,1,1>
1021: Success:Delete Redundant Statement: at <1,2,1,1,1,3,2,1>
1022: Success:Delete Redundant Statement: at <1,2,1,1,1,3,2,1>
1023: Success:Floop To While: at <1,2,1>
1024: Success:Remove Redundant Vars: at <>
```

Figure 8.8: Assembly GCD hill climbing successful transformations log

and fitness function, since the metrics in the intermediate versions of the programs are not improved. Unrolling the loop increases most metrics, and the propagation of the values just returns everything to previous sizes – the metrics do not care if the values of the variables are now lower. In this example it takes about 4 steps to reach better metrics. Looking ahead several steps would significantly increase the search space, and for general programs would rarely lead to improvements. With other start values the look ahead could be in the hundreds before there is an improvement. In a more general case with other input programs there is no guarantee that the process would ever end, since this is a variation of the halting problem, which is impossible to solve [Turing, 1937].

Other variations of the same algorithm were also automatically transformed. A version of the same program with user input for the starting numbers was transformed into the same core statements surrounded with the inputs and prints. For this process 37 transformations were selected out of the about 6K tested. Another version wrapped up the main part of the algorithm in a procedure. The result was again the same program, since the procedure is used only once and it got inlined into the main program. A somewhat different implementation uses recursion to solve the problem, always ensuring *a* is the larger number, subtracting *b* from it, and calling itself until they are the same. Interestingly, this was again transformed into a single `WHILE` loop by the automated transformations, since the recursion could be simplified to this form. Two variants of this implementation were transformed: one with predefined values, and another with user input, both of which ended with the core code shown in Figure 8.9. The number of transformations were 38 and 44, respectively, out of the about 6K and 12K tested. This shows that the increase for handling input and the extra code is not big in terms of selected transformations, but can be quite significant with the number of tested ones.

Several other samples are part of the *asm-a* set, including several versions of summing up arrays or user input stored in the stack, and an intentionally overly complicated factorial calculation. Transforming all of these results in some variants of loops and other high-level structures, with much improved metrics (see next section).

8.3.2 Overview of the Changes in the Whole Process

A selection of assembly programs (*asm-a*) was automatically transformed using the hill climbing program. To compare programs in different stages of the process, their sizes were expressed as the number of statements in them. The assembly code used was automatically striped of the macro definitions, to get a better picture of the "core" functionalities. The statement count was then defined as all the lines

```
gcd proc
    ; get params
    pop ax
    pop bx
    cmp ax,bx
    je endequal
    ja greatera
    ; ensure ax is greater
    xchg ax,bx
greatera:
    sub ax,bx
    push bx
    push ax
    call gcd
    pop ax ; result

endequal:
    push ax ; result
    ret
gcd endp
```

→

```
WHILE ax <> bx DO
  IF ax <= bx
    THEN < ax := bx, bx := ax > FI;
  ax := ax - bx OD;
```

Figure 8.9: Recursive GCD, assembly and automatically transformed WSL

that were not directives, empty, or comment only. This does count labels that are alone on a line as statements, but gives a good estimate of the sizes. For the WSL versions it was much simpler, since there is a number of statements metric built in. All of these for the set *asm-a* are given in Figure 8.10, with the boxes for the majority of the values, the lines in them being the actual median, and additional lines to show the scope of the remaining samples. The single dot in the *wsl* column shows an outlier value.

There is an increase in size with the translation to WSL (this is expected, as explained earlier with the translation process), and then a significant decrease with the transformations, in line with how the process is designed to work. Sizes per sample in different stages were divided to get more precise coefficients. The average result is an increase of 2.09 ± 0.29 times in the translation to WSL, i.e., for each of assembly statement a bit over two WSL statements were made. Similarly, the decrease from translated to transformed versions was on average 5.15 ± 2.96 times, or about 5 statements get replaced by a single one on average. Comparing directly, the original assembly samples are on average 2.41 ± 1.21 times larger then the final transformed ones. Again, this is just the core parts of the assembly programs, with most of the low-level operations for conversions of input and output removed.

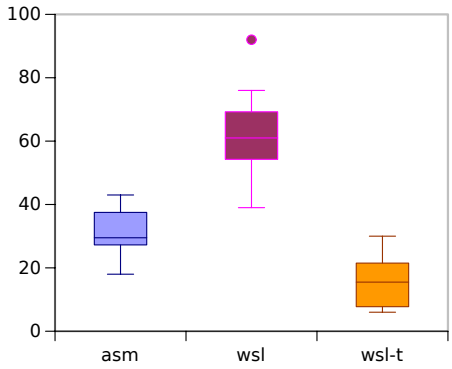


Figure 8.10: Program sizes in different stages (*asm-a*)

The end results of the transformation process can be compared to the originally translated versions using several built-in metrics (Section 3.5). None of these were readily available for assembly, so the first stage was not compared here. The average values, standard deviations, and the percentages of changes are presented in Table 8.1 for the following metrics: *McCabe's cyclomatic* and *essential* complexity; number of statements; number of expressions; *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.

Table 8.1: *asm-a* transformation metrics

Metric	WSL	WSL-t	% diff
McCabe Cyclo	7.40 ± 3	3.60 ± 1	41.60 ± 30
McCabe Essential	1.00 ± 0	1.10 ± 0	-10.00 ± 32
Statements	62.30 ± 15	16.00 ± 9	73.90 ± 14
Expressions	84.20 ± 28	31.50 ± 20	63.80 ± 14
CFDF	88.60 ± 16	22.80 ± 9	73.90 ± 8
Size	327.40 ± 71	95.30 ± 51	70.90 ± 13
Structure	947.70 ± 199	206.90 ± 119	78.20 ± 10

Included standard deviations for both the original programs and the transformed versions are high, which is a consequence of the variations between the samples. For this reason, the main focus is on the percentage of improvements, which is

less susceptible to these variations. Most metrics, with two exceptions, show 63 to 78% improvements, with deviations of less than 15 percentage points, indicating a relatively stable end result. McCabe's cyclomatic complexity had somewhat weaker results of about 42 percent, with a high deviation of 30 percentage points. This is partly due to the smaller numbers of the metric.

The outlier in the table is McCabe's essential complexity, which on average increased. Looking at individual samples, the increase is present only in a single one, where some of the introduced loops were not properly reduced, and all the other samples had unchanged values for this metric. Manually applied additional transformations can further simplify this program and in the process reduce the number to the expected value. The problem for the hill climbing process was that there was a significant increase of the structure metric when it tried to go in the "correct" direction, which is something that should be improved in the future, with further evolution of the process.

Another aspect of the process are execution times for the transformations. These were analysed from a run on a single core of a 3.5GHz Intel processor. A group of samples which are in the 0.5 to 2 seconds range can be noted, and then there are several samples with times of 5 to almost 30 seconds (see Figure 8.11). However none of the presented metrics of the input programs is directly related to the execution times. For instance, the shortest time was never on a sample with the lowest metrics values. On the other hand, about half of the samples with longer execution times had metrics in the ranges of the fastest group mentioned earlier. The samples with the largest metrics values also had varied times and were never the slowest.

In general, the execution times are more dependent on the properties of the program that is being transformed and how fast some structures can be found. This implementation of hill climbing tries to apply transformations to any part of the program until there is an improvement. Therefore, an input sample that is significantly reduced in size earlier in the process will also have a significant reduction in execution time. Further discussions of this aspect of the process are given in the following sections with MicroJava samples and then in the overview of the inputs in general (Section 8.5).

Results confirmed that the increases in size of the translations can be easily handled by later transformations, which again means that any sort of translator to WSL can be made with little effort spent on size optimisations. Automated transformations were able to reduce the size well below the original programs. Further manual transformations can be used when needed to achieve better end results. Even then, time spent by a human would be significantly reduced since a (potentially large) part of the process was already done and (maybe more importantly)

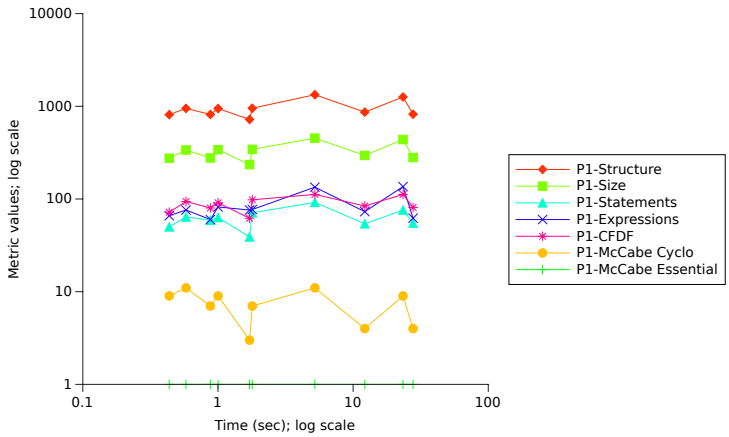


Figure 8.11: Transformation execution times for assembly samples

the program should also be much more understandable for a human user than what was originally translated.

8.4 Hill Climbing Approach on MicroJava

This section will first examine the effects of different translations on the process of automated transformations. Then the best of these translations and the results of all stages of the process is analysed in more detail in Section 8.4.2. This is followed by a few examples of characteristic programs being transformed in Section 8.4.3.

8.4.1 Comparison of Variations of Translation

Translation tool *mjc2wsl* has a number of switches that changes how some structures and operations are handled. Influences of some of these switches on the metrics of the translated programs were examined in Section 7.1.2. This section will focus on comparing the final outputs of the process on *alpha-wsl-v8* (all versions of the programs from the *alpha-mj*) sample set that were automatically transformed with the hill climbing program. Shorthands for the version names are listed in Table 7.1. Part of the process was automated using GNU Parallel [Tange, 2011]. Since all of these translations came from the same bytecode, the comparison will only take into

account the sizes of the final programs and try to 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.

The comparison will use percentage differences to the best results, that is the difference between the value and the best value for that sample and metric, divided by the best value. This is done to normalise the differences and make them more comparable, since they can vary greatly between samples, but also between metrics. In very rare cases, the best value of CFDF (*Control Flow and Data Flow*) can reach 0, which would lead to a "division by zero" error. To solve this problem that particular example has the values offset by one, which maintains the raw difference and gives an acceptable approximation of the percentage difference.

For *McCabe's Cyclomatic Complexity* *pp-lo* (*push/pop* for stack operations, with local `VAR` blocks) variants always have the best results. The *pp-gl* variants (with global temporary variables) have slightly worse results (up to 16%) on a few of the samples. The *ht* versions (*head/tail* for stack operations) vary – sometimes they are the same, but can be twice as big as the best results on some samples, mainly due to procedure parameter transformations which work with `POP/PUSH` exclusively.

McCabe's Essential Complexity is mostly the same across all of the variants. There are a few more complex examples for which the *gl* variants have their complexities several percentage points higher.

Number of statements and *Control Flow/Data Flow* are two metrics that have similar trends in their results. On average, the best results are with *pp-gl-ar* (local variables stored in an array), with *pp-lo-sp* being close, and in the case of sample `Rek1` being the best. Tables 8.2 and 8.3 illustrate this in more detail. The worst cases for *ht* versions are very consistent. In all variants there are samples that end up being the same as the best sizes (visible in the *min* column).

Size (of the AST) and *structure* (weighted sum of elements) metrics are showing similar trends between themselves. Variant *pp-gl-sp* is on average the best, with a few samples where *pp-gl-ar* is better. The *ht* variants are generally significantly worse, as can be seen from Tables 8.4 and 8.5.

Overall, there is a high variation of the differences in results within a single group across the different samples, which is obvious from the high standard deviation numbers in the given tables. This is partly because for every metric there were samples that would be transformed into the same form no matter what variant was given. On the other hand there would be a few samples with extremely bad results, shown in the *max* columns in the tables.

Table 8.2: Transformed *alpha-wsl-v8*, statements metric

	avg	stdev	max	min
ht-gl-ar	80.19	119.08	360.00	0.00
ht-gl-sp	134.48	120.73	340.00	0.00
ht-lo-ar	73.13	117.46	360.00	0.00
ht-lo-sp	108.17	112.14	340.00	0.00
pp-gl-ar	15.18	20.68	50.00	0.00
pp-gl-sp	36.26	48.35	136.36	0.00
pp-lo-ar	2.78	8.61	33.33	0.00
pp-lo-sp	10.78	18.03	55.93	0.00

percentage difference to lowest results

Table 8.3: Transformed *alpha-wsl-v8*, CFDF metric

	avg	stdev	max	min
ht-gl-ar	137.55	219.51	720.00	0.00
ht-gl-sp	277.05	247.39	700.00	0.00
ht-lo-ar	133.10	218.03	720.00	0.00
ht-lo-sp	242.23	254.28	700.00	0.00
pp-gl-ar	19.38	29.15	100.00	0.00
pp-gl-sp	39.31	48.75	120.00	0.00
pp-lo-ar	4.39	14.93	60.00	0.00
pp-lo-sp	12.61	22.00	65.06	0.00

percentage difference to lowest results

Table 8.4: Transformed *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 8.5: Transformed *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

Observing the results across metrics, *Essential* is an outlier for which only the globalness of the variables made a difference in a sparse few cases. For other metrics, much clearer conclusions can be made.

A general observation is that *ht* variants are worse for the process than the *pp* ones, 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.

Finally, the differences in storing local variables in a procedure are non-existent for McCabe's Cyclomatic Complexity. For others, storing the variables in a single array shows better results than handling them separately. The main exception to this is, however, an important one – the best results overall for *size* and *structure* are exactly with *pp-lo-sp*.

Compared to the initial metrics of the translated programs, a lot of the trends from there are reversed. 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.

Another issue to be considered when choosing the translation variant to work with is the execution times of the transformations, which are shown in Table 8.6, as well as the number of total transformations tried, and the number of transformations

that were selected. These numbers are taken from experiments run on a Intel Xeon E5-2420, clocked at 2.2 GHz, with all the times taken as totals for transforming all the samples in a single run. Looking at the switches individually, there is a clear 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 depending on the first two: in one extreme *ht-gl-ar* is twice as fast, with the *lo* versions it is about 20% slower, while *pp-gl-ar* is about 60% slower. The number of transformations tried also rises with the times, although not by the same factors. For example, the worst case has 162 times longer execution and "only" 22 times more transformations tried. 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 8.6: 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

In general, the variants with better metrics took significantly less time to execute. This makes sense from the point of the hill climbing process – on longer programs there are more transformations to try, therefore when the transformations are successful the process finds the local minimums faster.

The start points of the transformations also differ, as discussed earlier in Section 7.1.2, which influences the process as well. Overall the pop-push handling of the stack proves to be better than using head-tail. The initial translation has lower (and better) metrics for the global (*gl*) versions of programs, but this proves to be worse for the transformation process, since the end results with local (*lo*) variables have lower metrics, and the process itself takes significantly less time. The final switch for the storage of procedure local variables as an array (*ar*) or separate items (*sp*) has the least influence and which one is better varies between metrics on the

final transformed programs. On the initial translations, *sp* would usually be slightly better, while on the transformed ones *ar* shows better results for most metrics, except *size* and *structure*. The variant used in the main experiments was *pp-lo-sp*, since that one showed the best results for the *structure* metric, which is used for the fitness evaluation in the hill climbing process.

With all this in mind, considering the best results and execution times, the best candidates for transformations are *pp-lo-ar* and *pp-lo-sp* variants, with slight advantages going one way or the other, depending on the metrics and samples. The latter one will be used for further analysis of the process, due to it being slightly faster. Appendix C.1 contains more tables with metric changes for all of these remaining variants.

8.4.2 Overview of Changes in the Whole Process

This section focuses on the relations between the various versions of the programs during the whole process:

- the original manually written MicroJava code (marked with *mj* in Figure 8.12);
- the compiled bytecode (*mjc*);
- translations to WSL with *mjc2wsl* from the compiled bytecode;
- the fully transformed versions (*wsl-t*) made by the *hill climbing* program.

For this analysis the *pp-lo-sp* variant of the translated programs in sample set *alpha-mj* is used, for the reasons explained in the previous section.

One way to compare the programs is by the raw number of statements in them. By the nature of the process this will vary across the stages, with big increases in stages two and three, and a big decrease in the final stage, as shown in Figure 8.12. The thick parts of the bars display the majority of samples and the horizontal lines on them show the actual average. The thin extensions show the additional range of the minimal and maximal size, with an optional dot that represents an outlier, such as an unusually long program among the samples.

The average increase in size after compiling to bytecode in this experiment was by a factor of 2.55 with a standard deviation of 0.51, or in other words the MicroJava compiler produces 2.55 statements of bytecode for each original statement, which is a natural consequence of going to lower level structures. The translation to WSL multiplied the size by an additional 3.78 ± 0.38, mostly due to the additional statements need to emulate the inner workings of an MJVM. The increase in size for the WSL translation is somewhat dependent on the options used in *mjc2wsl*,

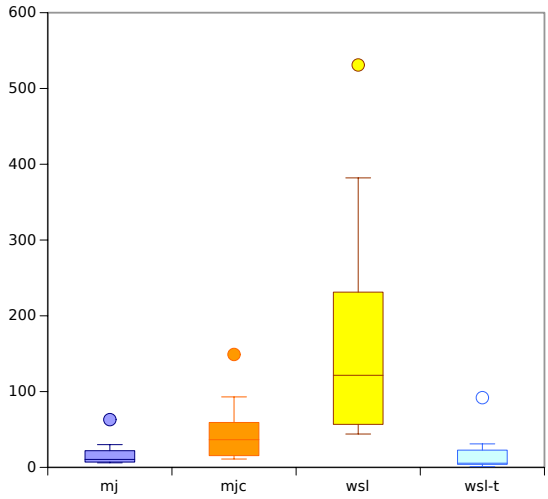


Figure 8.12: Number of statements in programs in different stages of the process (*alpha-mj*)

discussed in more detail earlier in Section 7.1.2. When comparing directly, the translated WSL had on average 9.55 ± 1.62 statements for a single original MicroJava statement.

The final stage reduces the sizes of the programs to similar levels as the originals, using the automated transformations. On average the reduction from translated WSL is by a deceivingly high factor of 16.89, with an equally high standard deviation of 11.55. The number suggests that the new programs should be overall much smaller that the starting ones, but the standard deviation correctly suggests there is a lot of variation. Some programs were indeed much smaller in the end, while a few remain bigger.

To evaluate the whole process better, the differences for each sample were expressed as a percentage of increase or decrease in the final transformed programs compared to the original MicroJava versions. The average for this on all samples was $24.43\% \pm 36.56$. The high deviation confirms again that there is a lot of difference in results, as can be seen in Figure 8.13, which shows all of the percentages for all of the samples, sorted in increasing order. On one side there are a few samples that are 60 to 80% less statements in the end, while on the other there are a few with 40% more. In some ways this is to be expected, because some structures are

very likely to be reduced to a single statement, while others must remain relatively complicated, or they will not get recognised properly and stay in long forms. The best percentages may seem a bit flattering, but this is due to two main reasons: some constructs are more compact in WSL; the original samples were written by humans, so they are potentially more verbose than they could be when a machine tries to reduces them as much as it can. On the other hand the worst percentages were on programs with a lot of array references and pointers which are currently not fully recognised by the transformations.

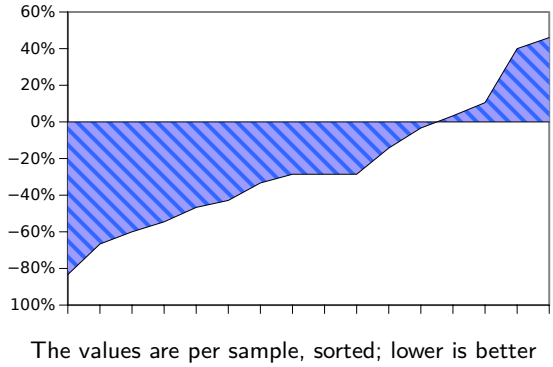


Figure 8.13: Percentage difference in statement count of original MicroJava and transformed WSL

Other metrics for Micro Java are unavailable, at least as of this writing, and as far as the author is aware. Tools tested that work with Java were not applicable, mostly since they tend to work directly on the bytecode which is significantly different for these purposes. The statement counting presented before was done with a relatively simple custom script that relied on samples being properly formated. It was not as simple to make scripts for other metrics, and therefore the comparison of other metrics throughout the whole process was not possible at this point.

On the other hand the metrics that are built into WSL (see Section 3.5 for a list and descriptions) can be used to give more of an insight of the quality of improvements in the last step of the process – the actual transformations. Table 8.7 shows the average values of the original code, the transformed code, and the average percent of improvement per program in *alpha-wsl-pp-lo-sp* sample set, all for the following metrics: *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.

Table 8.7: *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. The least improved is McCabe Essential complexity at 57.69%, partially because it can not go lower than 1 in a normal program, which it was in many of the transformed results, while the starting values were not very high. McCabe Cyclomatic complexity had a better improvement level of 66.38%, while all of the other metrics had much better results with improvements in the range of $87 - 94\%$. The standard deviations on the "raw" values of the metrics are pretty high, showing again the variety of the samples themselves. On the other hand all of percentage results had low standard deviations, showing that the process gives stable results in terms of the improvements for a single program.

Another important aspect is the time complexity of the process. All of the samples were transformed on a single core of an 3.5GHz Intel processor. The times were then compared to all of the available metrics for correlation. The data from *alpha-mj* shows a trend of execution time growth with the increase in most metrics, with a few deviations from some samples (Figure 8.14).

However, the data from the assembly samples (Section 8.3.2) showed that the growth in general was not directly related to the metrics values, and that there were much greater differences to the trends. To get more information about the process and the trends, alpha-mj was expanded with a few longer input programs that were not previously processed in detail (Figure 8.15). The new graph shows similar behaviour as was seen with the assembly samples, since some of the added programs were significantly outside of the trends.

In general, similarly to the assembly samples (Section 8.3.2), execution times do not seem to be a direct function of any of the basic metrics of the input pro-

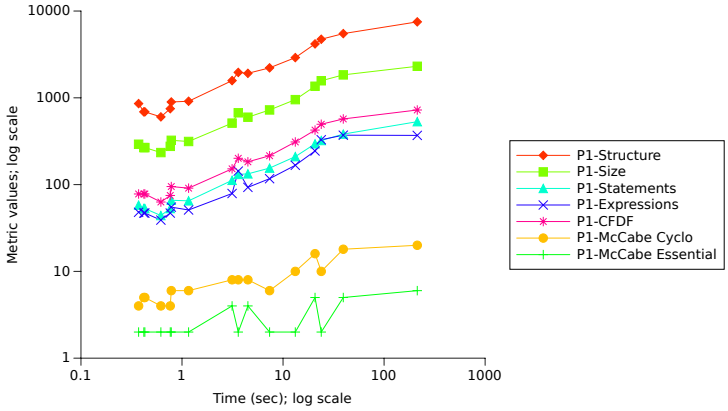


Figure 8.14: Transformation execution times for the *alpha-mj* set

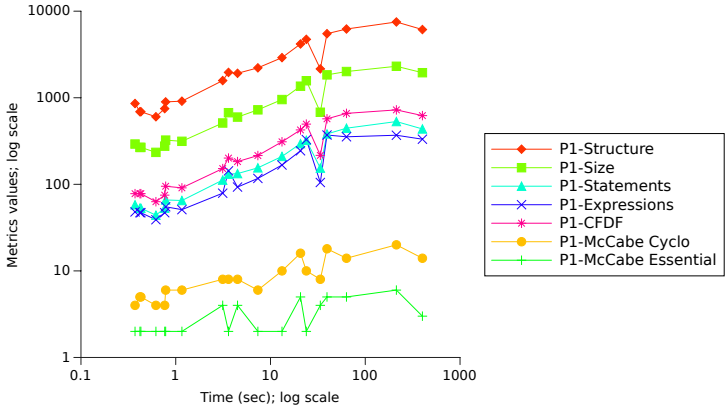


Figure 8.15: Transformation execution times for the expanded *alpha-mj* set

grams, although here we can observe stronger correlation. What is really important is what properties these programs have, and how long it takes for some of their basic features to get restructured. How these properties and susceptibility to restructuring correlate with different metrics is a topic for further study, since the program sets used in this thesis are not large and representative enough to support general conclusions on this issue.

Overall, this again showed that a translator to WSL can focus primarily on being correct without focusing on being efficient with the amount of code it produces since this can be reduced in automatic ways to sizes comparable to the original *high-level logic*, even though the translator was working with compiled low-level bytecode.

8.4.3 Examples of Automatically Transformed Samples

One of the examples for MicroJava syntax was a loop example named "while-print" presented in Figure 5.1 (page 50). Figure 8.16 shows this sample after the application of 51 transformations that were automatically picked by the hill climbing program using the *structure* metric as its guide. During the process more than 3400 transformations were tried. The end result is very clear, and practically identical to the starting program, with a declaration of a single local variable being used, and a while loop with increments and prints. What is important to note is that the transformations were applied to a bytecode translation and that the result was a high-level version of the program with no hints given to the hill climbing process where the input was obtained from.

```
VAR < mjvm_locals_0 := 0 >:
WHILE mjvm_locals_0 < 5 DO
  PRINFLUSH(mjvm_locals_0);
  mjvm_locals_0 := mjvm_locals_0 + 1 OD ENDVAR
```

Figure 8.16: Automatically transformed "while-print" example

An important aspect of the transformations is the recognition of recursion. An example with a recursive procedure with a single parameter is shown in Figure 8.17 both as the original source code, and the compiled bytecode. The recursive nature is fully maintained in the final automatically transformed version shown in Figure 8.18. The same script was used and it applied more than 50 transformations to this code, while more than 6000 were tried during the process. WSL has a syntax with a separate `WHERE` clause which contains the procedure and function definitions. The

procedure name remains from the original address that it started at, while the parameters are sequentially named as needed.

```
                                14: enter 1 1
                                17: load_0
                                18: const_0
                                19: print
                                20: load_0
                                21: const_0
                                22: jle 9 (=31)
 program Rek1{                  25: load_0
     void func(int i){          26: const_1
         print(i);              27: sub
         if (i>0)               28: call -14 (=14)
             func(i-1);         31: exit
     }                          32: return
                                33: enter 0 0
     void main() {              36: const_5
         func(5);               37: call -23 (=14)
     }                          40: exit
 }                              41: return
```

Figure 8.17: Recursion example – MicroJava and bytecode

```
BEGIN
  a14(5)
WHERE
  PROC a14(par1) ==
    PRINFLUSH(par1); IF par1 > 0 THEN a14(par1 - 1) FI
  END
END
```

Figure 8.18: Recursion example – transformed WSL code

To demonstrate the handling of functions that have return values, the next example is a recursive Fibonacci method. Figure 8.19 shows the original MicroJava program and the final transformed version. This sample compiles into 40 operations in bytecode, which is then translated to 133 statements in WSL. The final WSL program had more than 90 transformations applied to it by the automated script, out of the almost 16K tried. The syntax of a `FUNCT` requires that the block ends with an expression in parenthesis that is the actual return result. The `@Format`

command is added to get the same output formatting as the original program in MJVM does.

```
program RekFib{
  int fib(int f)
  {
     if (f==0)
       return 0;
     if (f==1)
       return 1;
     return fib(f-2)+fib(f-1);
  }

  void main()
  {
     print(fib(0),3);
     print(fib(2),3);
     print(fib(7),3);
  }
}
```

```
BEGIN
  PRINFLUSH(@Format(3, a14(0)));
  PRINFLUSH(@Format(3, a14(2)));
  PRINFLUSH(@Format(3, a14(7)))
WHERE
  FUNCT a14(par1) ==
    VAR <  >:
    SKIP;
    (IF par1 = 1
      THEN 1
      ELSE IF par1 <> 0
        THEN a14(par1 - 2)
             + a14(par1 - 1)
        ELSE 0 FI FI) END
END
```

Figure 8.19: Recursive Fibonacci example – MicroJava and transformed WSL code

8.4.4 Verification of the Transformed Programs

As an additional verification of the outputs of the transformation process, especially during regression tests, an automated testing facility is integrated with the source build tools. The layout is very similar to that presented in Section 7.1.3, except that here the original translated program and the transformed version are run on the same inputs and any differences in the outputs are reported for manual inspection.

All of the presented programs and versions in *alpha-wsl-v8* show exactly the same outputs after transformations.

8.5 Overview of Hill Climbing Inputs

The transformation script is designed so that it works with any general WSL program, not just with the ones obtained from the mentioned translation tools.

The translated programs that are the input for the transformations had a lot of differences between them. With *asm2wsl* the resulting program is a *regular* action system, where none of the calls return, and the execution ends when the predefined

z action is called. On the other hand, *mjc2wsl* creates the opposite, a *recursive* action system, in which all of the calls return and the execution ends when the first called action is done. With *asm2wsl* individual actions represent labels in the original program, and can consist of any number of instructions, while *mjc2wsl* creates a single action for each of the instructions in the original bytecode.

Both tools feature switches that can change some aspects of the output code and the way it is organised. There are versions with mostly global variables that are reused a lot, and others with a lot of small scope local variables. Local variables for a procedure (*stack frames*) were also treated in different ways. There were different ways in which the stacks were accessed. However all of these types of outputs were automatically transformed with the same transformation script, admittedly with some variation in the quality of end results.

The same script was also used successfully on some hand written WSL programs, including the transformer itself.

Execution times of the transformation process were compared with the metrics of the input programs on both the assembly (Section 8.3.2) and MicroJava (Section 8.4.2) samples. The growth of execution time was not directly proportional to any of the metrics. That is, there were significant outliers to any of the trends, and programs with similar metrics can have wildly different transformation times. The properties of the presented process are such that it is more important what the samples have in them than how long they are. The hill climbing algorithm will try to apply transformations to any part of the programs to find an improvement. If the input is such that large improvements are found earlier, there will be less places to test transformations later on, significantly reducing the execution time.

On the other hand it is early to make generalised statements about input programs in general. The sample size here is relatively small. Also, while the growth is not directly proportional to any of the metrics, there is an expectation that larger and more complex programs will take longer to transform, it is just not clear how much longer.

The quality of the end result of automated transformations is heavily reliant on the used fitness function that guides the process. In these experiments this was the custom *structure* metric. It was successful in restructuring and reducing the metrics of the programs generated with the tools developed in this thesis. However, with other input programs it could prove to be much less successful. In general, it is impossible to find a universally good fitness function. The "no free lunch" theorem for search and optimisation states that for any search algorithm that is well suited for one class of problems, there is a separate class that will offset this advantage [Wolpert and Macready, 1997]. Of course, this holds when all possible inputs are taken into consideration, and there is no prior knowledge of the input

space at hand. For this process in particular, the worst possible input would be one where no transformation can lead to the reduction of the fitness function. In that case the end result would be the same as the original input. This also means that the process can not "corrupt" a program and lead it to a worse state, at least not according to the fitness function.

When considering the input programs that were the main target of this research, there are characteristics that can be taken into account. They do not feature high-level structures and in general have a large amount of labels and jumps that can be restructured. Since these start programs will have higher values for the *structure* metric than even a modestly structured equivalent program, it is likely that, on average, this fitness function is adequate for this class of problem. Of course, it is always an open question whether there is a better function and algorithm.

The process is also guaranteed to end on any possible input. All the transformations are tried on all possible points in the program, and the only possible step forward is to reduce the value of the structure metric. If no transformation improved the metric, the process is finished. It is not possible to get into a loop between two states, since the metric can not go up.

In Section 8.3.1, an example was shown in which further improvements could be made, but it would need at least 4 steps of look-ahead before the metric would be reduced. However, as discussed there, it is impossible to know for a given program how many steps are needed for an improvement, or if there is indeed such a number, since this is a variation on the halting problem [Turing, 1937].

Chapter 9

Thesis Conclusions

This dissertation presents approaches to automated software translation and transformation with the main goal of extracting high level concepts from low level original programs. One of the main problems that developers face when working with such low level programs is to first understand how they exactly work. Quite often the only artefact to work with are the executables themselves, with the original source code and documentation unavailable, out of date, or possibly have not existed to start with. As such, the process presented here can be used in various software maintenance applications, since it enables easier understanding of programs and at least partially automates the restructuring process.

The presented process consists of several stages and is designed in a way that allows for a flexible combination of tools. One of the core elements is the usage of semantics preserving transformations available in *FermaT*, a system built around *WSL (Wide Spectrum Language)*. To be able to use the transformations in a general case the first basic step is the translation from the original language to WSL. In general the transformations can be manually selected, or done in an automated way, or a combination of both. The main interest of the experiments done in this thesis was the automation of the process. This enables users with no domain knowledge to use the tools effectively. At the same time, due to the flexibility of the process, an experienced user can adapt parts of it as needed, or integrate it into another restructuring or development process.

Two tools developed for translation from x86 assembly and MicroJava bytecode were presented in Sections 6.1 and 7.1. Both of these tools try to encapsulate the interaction of the software with the underlying machine, in one case a part of an x86 processor, in the other a complete stack based virtual machine. This

is achieved via emulating the behaviour of these systems, with additional global variables that represent their internal states, and as a consequence the resulting code has a significant increase in size and verbosity.

The automated transformation part of the process uses a *hill climbing* algorithm. Transformations from a predefined set are applied to the program, and a *fitness function* is used to evaluate whether the new version is improved and should be used as the next step in the process. This is repeated until no transformation shows an improvement. The results of transformations of translated assembly and MicroJava samples were given in Sections 8.3 and 8.4. The metrics of the variations of translation of *mjc2wsl* were presented in Section 7.1.2, and their influence on the transformation process was analysed in Section 8.4.1. The transformation script was designed to work with general WSL programs, i.e., it is not dependant on knowing that they were translated from these tools. The main variations of input programs that were handled in the experiments are listed in Section 8.5. The same section also shows that for these experiments the execution times are not directly related to the available metrics of the input programs. The times are much more dependent on the ability of the transformations to reduce the size of the programs faster. There is of course a trend that longer programs will take longer to process, but it is hard to determine how much longer. Still, since restructuring is mostly an offline activity the duration of the process is secondary to the end results. The section finishes with a discussion on the applicability of the process to other possible inputs.

The additional commands that were introduced by the translator can mostly or completely be removed and simplified during the transformation stage. This makes the creation of these translators simpler and faster, since almost all of the efforts can be invested in the correctness of the process, instead of simultaneously making size optimisations.

During the development of this thesis, FermaT itself was also extended by adding transformations to its library, as well as some improvements and expansions to existing ones. These include transformation of pushes to the stack before procedure calls into formal parameters, as well as pops inside the body of a procedure into assignments from those parameters. Similarly, a push at the end of a procedure can be recognised as a return parameter, and transform the procedure into a function. All of these were written to be as general as possible in the spirit of other transformations in Fermat. For instance, the aforementioned transformation that recognises stack operations as formal parameters of a procedure, works for any general stack. No assumption of a particular name specific to a translator is made. The expression simplifier in FermaT was extended with more optimisations of character operations, such as conversion of numeric codes into actual chars, and combining lists of individual chars into strings where applicable.

An explicit list of contributions was given in Section 1.1, page 4.

9.1 Comparison to Other Approaches

In principle, the main steps of this approach are similar to the earlier assembly migration done with FermaT [Ward, 2013; Ward, Zedan, and Hardcastle, 2004]. These would also start with a translator from assembly to WSL, appropriate transformations of these and finally a translator from WSL to a high level language, such as C or COBOL, and were successfully used in large scale industrial migrations, sometimes with upwards of half a million lines of code of assembly. However, these processes were mostly crafted and customised for the programs at hand, and the low level memory management was in general handled with additional tables and mappings that would be "hidden" from the WSL translations and be reintroduced once the code was translated to the target language.

The main difference that the tools in this thesis introduce is the insistence on having runable, self-contained resulting code. This provides full flexibility to use the resulting code in whatever process that can handle WSL and can decouple the final translation tools as well. Another important advantage is the ability to do run time testing at any point in the process. This was used to verify that the translations worked in the same manner as the original programs, and that the final transformed versions work in the same way as the original translations. It was also very useful to hunt down any problems that were introduced during the transformation process itself, for instance while testing a new individual transformation, or a whole new transformation process. This can be useful in a larger process of migrating old code. If a fault in the original code is found during the process it can be fixed and tested right there. If it is more convenient the fix could be introduced at any point of the process, and the restructuring could be continued from there. The down side is that this made significant limitations to the inputs that the current version of *asm2wsl* can handle, which is one of the reasons that less effort was put into its development. On the other hand, *mjc2wsl* is capable of handling practically any valid bytecode, due to the different, more strict ways that MJVM handles the memory. Therefore, the down side is much less prominent with MicroJava.

The goals of the thesis are also very similar to "classic" decompilers, some of which were listed in the chapters about assembly and bytecode. The general approach involves the analysis of control flows and structures, as well as pattern matching to uncover higher level versions of the programs. In case of assembly programs, this is usually much more difficult, due to intense compiler optimisations and ambiguity of translations. With bytecode many of these problems are avoided,

since a lot of data, such as types, methods and sometimes even comments, are encoded in the binary form. This is due to the just-in-time nature of the optimisations in the virtual machines. The approach presented in this thesis splits up the process in stages and makes the automated transformation program independent of the input language and architecture. Those specifics are handled by the translators to WSL, with a focus on correct translation and no need to optimise the size of their outputs, as explained in the previous section. The transformations work on general WSL input programs, with a focus on removing low-level structures. Therefore, this approach is more general and adaptable, but on the other hand decompilers can show excellent results for their particular languages and specific types of programs they were designed for. No direct comparison of outputs of decompilers with this work was possible, due to the differences in MicroJava and regular Java bytecode. In general the outputs of decompilers and the approach in this thesis can be excellent, high-level versions of programs, or can be only partially restructured, depending on the input programs.

Comparison to other systems is much less direct, largely due to the specific nature of FermaT. For instance, GenProg (Section 2.1.2) uses fitness in its automated process, but there are many differences past this initial likeness. The goal of software repair is very different to the one here. The fitness is also based on testing the behaviour of programs, not on the qualities of the code. Changes to the programs are done with mutation operators, not with formal semantics preserving transformations, which in turn would not be really applicable for a process that actually tries to change the semantics of a program.

The hill climbing algorithm was not successfully used before for this type of application, at least as far as we are aware of. Fatiregun [2006] compared several approaches to solve a similar problem of automated restructuring, but with a significantly different starting point and route to the solution. The transformation sequence used on a program is the *instance* that is being optimised by mutation operators. The comparison showed genetic algorithms to be better than either hill climbing or random search [Fatiregun, Harman, and Hierons, 2004]. It should be noted, however, that the approach itself was much more in line with general genetic algorithm approaches. Hill climbing was also discussed as a potential solution for bug fixing. For instance, in [Arcuri and Yao, 2008], the authors used genetic algorithm for bug fixing. They do mention hill climbing, but were sceptical about it and its tendency towards local optimums.

9.2 Future Work

There are many directions to continue the research presented in this thesis, especially taking into consideration the flexibilities and independence of the tools at different stages of the processes presented.

Translation Tools

The assembly translation tool originally started this research, but was somewhat left behind compared to the later work on bytecode. This was partly because of the complexities of behaviours of an x86 processor and the very different ways that assembly programs in general go about similar tasks. Because the author is not a "native" assembly programmer, it was much harder to anticipate the various "normal" behaviours and therefore it was hard to find good real world inputs that could be handled in a reasonable amount of time. Still, many of the lessons learned from the other tools could be used to improve this one.

The MicroJava bytecode translator does handle almost all of the behaviours of the virtual machine as it was designed to be used. Further development could be made to support some irregularities and fringe cases. Even more different translations of same inputs could be added for further research of the impact this can have on transformations.

More importantly, tools for handling regular Java bytecode could be developed, potentially as a gradual expansion of MicroJava and the associate virtual machine and introduction of new features into it, or as a completely new tool from scratch. Handling of many calls to the API and classes outside of the ones translated could be handled with wrappers and external calls built into WSL.

Other low level languages and architectures could also be translated into WSL using similar ideas as presented here. In some cases it might be more efficient to expand WSL itself instead of modelling all the behaviours with some sorts of "virtual" processors. The larger part of such efforts is to expand the existing transformations that come with FermaT to be able to recognise and handle these new structures.

Many advanced instructions, such as vector multiplication are straightforward to implement in a step by step manner. Conditional execution instructions, such as those seen on some ARM processors, are those that take into consideration some of the processor flags to decide if they are going to be executed. These could be modelled analogous to the conditional jump instructions, with explicit checks of the important flags. Some cases could prove to be more challenging, like the memory management in *asm2wsl*, but even these could be further developed given enough time, and might be a requirement for some types of programs. For instance, *volatile*

variables require that they are read from the memory on every access, since they could have been changed by another process. For this to work, the translation either need to have direct access to the memory, or special labels for an external tool that will synchronise the values as needed.

The current translators were not made to work with multi-threaded programs. Such support in the future could be made in two ways, most likely a combination of both. One is to have a much deeper virtual processor with a simulation of how the actual process handling is done, with the translation of appropriate instructions that facilitate this. The other is to have a deeper analysis of the threads and to translate them to adequate structures in WSL for concurrent programs. These would include any types of atomic operations (such as value increments and swaps), as well as any synchronisation or locking instructions. In some types of multi threaded applications, the `nop` instruction, which has no effect, is used to "pad" the program and ensure a waiting time. With the current tools, these would be translated to a `SKIP` and deleted at some point. This could be easily changed in the translation stage to have a non-deletable variant of the statement. These approaches to threads would result in much slower translated programs, but in a restructuring scenario the accuracy of the end models is more important. Additional optimisations could be performed on the final result in WSL, or during the translation to the target language.

Most of the modern high-performance central processors feature out-of-order execution, which means that they can adapt the order of instructions based on the availability of input data, or organise them in other logical groups. For most cases, the out-of-order scenario is just a CPU feature that speeds up the execution and is not something that changes the logic of the program. If such strategies would be important for the code at hand, they could also be simulated on the level of the virtual processor. This would, however, require a lot of work to get all the details implemented and could have serious impacts on the abilities of transformations to restructure the code without expanding them as well.

There is also a need for open source translators from WSL to other languages. The migration projects that use FermaT for industrial purposes used custom translators. These are usually tailored for the job at hand and sometimes even feature specific transformations that would adapt WSL code for easier translation to the target language [Ward, 2013]. Closest to this project would be to create a translator to MicroJava or directly to the appropriate bytecode. It would also be interesting to have a translator to Java. WSL is translated into Scheme for execution in the current version of FermaT, and then with the Hobbit compiler into C code for an

even faster execution. Both of these could also be potential starting points for some types of translators.

Developments to FermaT

Expanding and improving FermaT, WSL, and the transformation catalogue is also integral to most future efforts. Some of the existing transformations could be expanded to recognise even more special cases. For instance, this thesis explored some of the differences between using pop and push for stack-like structures and direct list manipulations. Some transformations were more successful on one variant. The causes for this should be further explored and the transformations themselves potentially expanded to handle this better. Alternatively further transformations could be added that recognise these different access types and convert between them when needed. Similar analysis could be done with the differences between how local variables are handled, and the balance between local and global variables.

There are also ideas for transformations that would not influence any metrics, but would increase readability. One of such ideas is to automatically rename local variables with generic names (like `mjvm_locals_1`) to more associative names. For instance if it is detected that the variable is used as a counter it could be named `count_1`, or the "classic" `i, j, k` letters for counters. If it is mainly used for tests it could be named as a flag, etc. Others could be renamed from a set of neutral words that are easier to follow for a human then "var1", "var2", etc.

Improvements to The Automated Transformation Process

By the very definition of the hill climbing process, it is very dependant on the choice of its fitness function. In this thesis, some initial experiments showed that the custom *Structure* metric was giving good results, while also being computationally very efficient. However there is a lot of room to explore other fitness functions and potentially combine metrics in them depending on some attributes of the sample under hand.

The hill climbing process itself could also be further improved with changes to the transformations used. There is a large amount of data about the transformations tried during these experiments which could be analysed to change these sets. The current version tries a single transformation and then tries to pair them up to see if there will be any success after the second transformation. It also features groups of transformations that are useful when paired up, so these are tested earlier that others. The current sets were made based on the domain knowledge of Dr. Martin Ward and some early experiments, but some further tweaks and experiments based

on the collected data could also be used. There are transformations that are always or almost always used after another transformation, so they could be grouped up, or potentially the later ones could be separated to not be used early on. The process could also be expanded to test more steps ahead, although this would lead to an exponential increase in time needed.

A variation on the current script could try all the transformations on a program, and then choose the one with the best fitness for the next step, instead of just progressing as soon as an improvement is made. This could also be optimised to use multiple cores for the parallel transformations of versions. The hill climbing algorithm by design has a tendency to find local optimums. A common approach to solve this is to try the process from several points in the space of possibilities and check for the best result after several attempts. In the current implementation this could be achieved with shuffling of the transformation application order. Logs from such experiments could again be used for better "default" orderings. Of course with any of these adaptations based on the data from the initial sample sets, one needs to be careful of over-fitting the process to these programs and potentially make it worse for new samples.

Another improvement could be done with the evaluation of metrics in the whole process, by potentially using external tools, such as those in SSQSA *(Set of Software Quality Static Analyzers)* [Rakić, 2015]. Metric values of translated and transformed WSL are currently used to evaluate the improvements in the transformation process, but if unified metrics were available, then the initial source code could also be compared. This would be possible with SMIILE *(Software Metrics Independent on Input LanguagE)*, which is a part of SSQSA. Additionally, the system has other analysers which could be applied for deeper understanding of the process itself, but also as an expansion of the process itself. For instance, control flow graphs could be generated from the different versions of the code and used in its understanding.

New Transformation Processes

The current process relies on the hill climbing algorithm. Some variations to the algorithm were already suggested in the previous section, but other searching approaches could also be tested in the future. One of these is the *tabu* algorithm, that is very similar, but allows for steps towards worse solutions where no better ones can be found [Glover, 1989]. Variations of evolutionary algorithms could also be tested. The transformations could be used as mutation operators, while the metrics would remain to be the fitness evaluation.

On the other hand, the approach with hill climbing could also be used with other transformation systems, not just FermaT. It is important to note that any system that does not feature semantic preserving transformations would also need additional verification that the behaviour of the program has not changed. For instance, this could be done with sets of test inputs and outputs ran after each change.

Part IV

Appendices

Appendix A

FermaT Transformations Catalogue

Following is a list of available transformations distributed with FermaT. It was generated directly from the source files, based on the scripts that FME uses for its list of transformations.

Abort Processing Type: Simplify

Nodes: `T_Abort`

Simplify statement sequences containing an ABORT.

Absorb Left Type: Join

Nodes: `T_Assign, T_Assignment, T_Cond, T_D_If, T_Floop, T_For, T_Statement, T_Var, T_Where, T_While`

This transformation will absorb into the selected statement the one that precedes it.

Absorb Right Type: Join

Nodes: `T_Assert, T_Assign, T_Assignment, T_Exit, T_For, T_Statement, T_Var, T_Where, T_While`

This transformation will absorb into the selected statement the one that follows it.

Actions to Procs Type: Rewrite Simplify

Nodes: `T_A_S`

Search for actions which call one other action and make them into procs.

Actions to Where Type: Rewrite Simplify

Nodes: `T_A_S`

Convert an Action System to a Where clause.

Add Assertion Type: Insert

Nodes: `T_Abort, T_Assert, T_Assign, T_Assignment, T_Cond, T_Condition, T_False, T_Guarded, T_Statement, T_True, T_While`

This transformation will add an assertion after the current item, if some suitable information can be ascertained.

Add Left Type: Join

Nodes: `T_Assign, T_Statement`

This transformation will add the selected statement (or sequence of statements) into the statement that precedes it without doing further simplification.

Add Loop To Action Type: Simplify

Nodes: `T_A_S`

If an action is only called by one other action, in a regular system we can merge the calls to the first action by introducing a Floop, replacing the calls by EXITs and adding a single call after the loop.

Align Nested Statements Type: Rewrite

Nodes: `T_Comment, T_Cond, T_Guarded`

This transformation takes a guarded clause whose first statement is a If and integrates it with the outer condition by absorbing the other guarded statements into the inner If, and then modifying its conditions appropriately. This is the converse of Partially Join Cases.

Align Nested Vars Type: Simplify

Nodes: `T_Var`

Aligns nested VAR blocks into a single block if the variables are all different.

All Proc Stacks To Pars Type: Simplify

Nodes: `T_Proc, T_Where`

Convert stack references to a procedure parameter for all procedures in a program.

All Push Pop Type: Rewrite

Nodes: `T_Push`

Apply Push_Pop wherever possible.

Apply To Right Type: Use/Apply

Nodes: `T_Assert, T_Assign, T_Cond, T_D_Do, T_D_If, T_Statement, T_While`

This transformation will apply the current program item to the one to its immediate right. For example, if the current item is an assertion and the next item is an IF statement, then the transformation will attempt to simplify the conditions using the assertions.

Array To Vars Type: Rewrite

Nodes: `T_Array, T_Number, T_Proc_Call, T_Var, T_Var_Lvalue`

Convert a local variable array to a set of local variables.

Collapse Action System Type: Rewrite

Nodes: `T_A_S`

Collapse action system will use simplifications and substitution to transform an action system into a sequence of statements, possibly inside a DO loop.

Collapse All Action Systems Type: Rewrite

Nodes: `T_Where`

Collapse All Action Systems will attempt to collapse the action systems within a program which is a WHERE structure.

Combine Where Structures Type: Rewrite

Nodes: `T_Where`

Combine Where Structures will combine two nested WHERE structures into one structure which will contain the definitions from each of the original WHERE structures. The selected WHERE structure will be merged into an enclosing one if there is one or, failing that, into an enclosed WHERE structure.

Constant Propagation Type: Simplify

Nodes: `T_Condition, T_Expression`

Constant Propagation finds assignments of constants to variables in the selected item and propagates the values through the selected item (replacing variables in expressions by the appropriate values).

D Do To Floop Type: Rewrite

Nodes: `T_D_Do`

Convert any D_Do loop to a DO...OD loop.

Decrement Statement Type: Rewrite

Nodes: `T_Statement`

Decrement a statement, provided it is enclosed in a suitable double Floop.

Delete All Assertions Type: Simplify

Nodes: `T_Assert`

This transformation will delete all the ASSERT statements within the selected code. If the resulting code is not syntactically correct, the program will be tidied up which may well result in the re-instatement of ASSERT or SKIP statements.

Delete All Comments Type: Delete

Nodes: `T_Comment`

This transformation will delete all the COMMENT statements within the selected code. If the resulting code is not syntactically correct, the program will be tidied up which may well result in the insertion of SKIP statements.

Delete All Redundant Type: Delete

Nodes: `T_Assign`

Delete All Redundant searches for redundant statements and deletes all the ones it finds. A statement is Redundant if it calls nothing external and the variables it modifies will all be assigned again before their values are accessed.

Delete All Skips Type: Delete Simplify

Nodes: `T_Skip`

This transformation will delete all the SKIP statements within the selected code. If the resulting code is not syntactically correct, the program will be tidied up which may well result in the re-instatement of SKIP statements.

Delete Item Type: Delete

Nodes: `T_Action, T_And, T_Assert, T_Assign, T_Assignment, T_BFunct, T_Comment, T_Cond, T_Condition, T_Divide, T_Expression, T_False, T_For, T_Funct, T_Guarded, T_Minus, T_Or, T_Plus, T_Proc, T_Skip, T_Statement, T_Statements, T_Times, T_True, T_While`

This transformation will delete a program item that is redundant or unreachable.

Delete Redundant Statement Type: Delete

Nodes: `T_Abort, T_Assert, T_Assign, T_Assignment, T_Comment, T_Error, T_Prinflush, T_Print, T_Rel_Seg_Lvalue, T_Statement, T_Sub_Seg_Lvalue`

Delete Redundant Statement checks whether the current statement is Redundant (because it calls nothing external and the variables it modifies will all be assigned again before their values are accessed). If so, it deletes the Statement.

Delete Unreachable Code Type: Simplify

Nodes: `T_A_S, T_Cond, T_D_Do, T_D_If, T_False`

Delete Unreachable Code will remove unreachable statements in the selected object. It will also remove unreachable cases in an IF statement, e.g those which follow a TRUE guard.

Delete What Follows Type: Use/Apply

Nodes: `T_Cond, T_Guarded, T_Statement, T_True`

Delete What Follows will delete the code which follows the selected item if it can never be executed.

Double to Single Loop Type: Rewrite

Nodes: `T_Cond, T_Exit, T_Floop`

Double to Single Loop will convert a double nested loop to a single loop, if this can be done without significantly increasing the size of the program.

Else If To Elsif Type: Rewrite

Nodes: `T_Comment, T_Cond, T_Guarded`

This transformation will replace an Else clause which contains an If statement with an Elsif clause. The transformation can be selected with either the outer If statement, or the Else clause selected.

Elsif To Else If Type: Rewrite

Nodes: `T_Cond, T_Guarded`

This transformation will replace an Elsif clause in an If statement with an Else clause which itself contains an If statement. The transformation can be selected with either the If statement, or the Elsif clause selected.

Expand And Separate All Type: Simplify

Nodes: `T_A_S`

Expand And Separate All will attempt to apply the transformation Expand and Separate to the first statement in each action in an action system. This will be useful for dealing with the skip_flag in WSL derived from Herma assembly.

Expand And Separate Type: Reorder

Nodes:

Expand And Separate will expand the selected IF statement to include all the following statements, then separate all possible statements from the resulting IF. This is probably only useful if the IF includes a CALL, EXIT etc. which is duplicated in the following statements, otherwise it will probably achieve nothing.

Expand Call Type: Rewrite

Nodes: `T_Actions, T_Call, T_Funct, T_Funct_Call, T_Proc, T_Proc_Call, T_Skip`

Expand_Call will replace a call to an action, procedure or function with the corresponding definition.

Expand Forward Type: Join

Nodes: `T_Cond, T_D_If`

Expand_Forward will copy the following statement into the end of each branch of the selected IF or D_IF statement. It differs from Absorb Right in that the statement is only absorbed into the top level of the selected IF.

Find Terminals Type: Rewrite

Nodes:

Find and mark the terminal statements in the selected statement. If a terminal statement is a local proc call, apply recursively to the proc body.

Flag Removal Type: Simplify

Nodes: `T_Var`

Attempt to remove references to flag variables

Floop To While Type: Rewrite

Nodes: `T_Call, T_Floop`

Convert a suitable DO...OD loop to a While loop

For In To Reduce Type: Simplify
Nodes: `T_Assert, T_For_In`

Replace a FOR x IN y loop with an equivalent REDUCE.

For To While Type: Rewrite
Nodes: `T_For`

Convert any FOR loop to a VAR plus WHILE loop

Force Double - Single Loop Type: Rewrite
Nodes: `T_Floop`

Force Double - Single Loop will convert a double nested loop to a single loop, regardless of any increase in program size which this causes.

Fully Absorb Right Type: Join
Nodes: `T_Assert, T_Assign, T_Assignment, T_Exit, T_For, T_Statement, T_Statements, T_Var, T_Where, T_While`

This transformation will absorb into the selected statement all the statements that follow it.

Fully Expand Forward Type: Join
Nodes:

Apply Expand Forward as often as possible.

Globalise Procs Type: Rewrite
Nodes: `T_Statement, T_Where`

Move procs to an enclosing WHERE (opposite of Globalise_Procs).

Globals To Pars Type: Rewrite
Nodes: `T_Where`

Convert global variables in procs to extra VAR parameters.

Ifmatch Processing Type: Simplify
Nodes:

Convert an IFMATCH statement to a @New_Match call.

Increment Statement Type: Rewrite
Nodes: `T_Assign, T_Floop, T_Statement, T_Statements`

Increment a statement, provided it is enclosed in a suitable double Floop.

Insert Assertion(s) Type: Insert

Nodes: `T_Cond, T_D_Do, T_D_If, T_Guarded, T_While`

This transformation will add an assertion inside the current item, if some suitable information can be ascertained.

Join All Cases Type: Rewrite Join

Nodes: `T_Comment, T_Cond`

This transformation will join any guards in an If statement which contain the same sequence of statements (thus reducing their number) by changing the conditions of all the guards as appropriate.

Join Cases Left Type: Join

Nodes: `T_Guarded`

This transformation will merge the selected Guarded clause with the one before it.

Join Cases Right Type: Join

Nodes: `T_Funct_Call, T_Guarded, T_MW_Funct_Call, T_MW_Proc_Call, T_Proc_Call, T_Skip, T_X_Funct_Call, T_X_Proc_Call`

This transformation will merge the selected Guarded clause with the one after it.

Localise Procs Type: Rewrite

Nodes: `T_Statement, T_Statements`

Create a local WHERE for procs which are only called in the selected item.

Loop Doubling Type: Insert

Nodes: `T_Floop`

Loop doubling will duplicate the body of an Floop.

Loop Inversion Type: Move

Nodes: `T_Assign, T_Floop, T_Statement`

Loop inversion will move the selected statement to the top of the loop body.

Loop To Move Type: Rewrite

Nodes: `T_Floop, T_Greater_Eq, T_Number, T_While`

Convert a suitable DO...OD or WHILE loop to assignments.

Make Loop Type: Insert

Nodes: `T_Call, T_Statement, T_Statements`

Make loop will increment a statement or statement sequence and put it in a (false) loop.

Make Procedure Type: Rewrite

Nodes: `T_Action, T_Call, T_Statement, T_Statements`

Make Procedure will make a procedure from the body of an action or from a list of statements.

Make Reducible Type: Rewrite

Nodes: `T_Statement, T_Statements`

Use absorption if necessary to make the selected item reducible (ie all terminal statements with terminal value 1 are in terminal positions in the given item).

Merge Calls in Action Type: Simplify

Nodes: `T_Action, T_Call, T_Cond`

Merge Calls in Action will attempt to merge calls which call the same action, in the selected action.

Merge Calls Type: Simplify

Nodes: `T_A_S`

Use absorption to reduce the number of calls in an action system.

Merge Cond Right Type: Simplify

Nodes: `T_Comment, T_Cond, T_Skip`

Merge a binary Cond with a subsequent Cond which uses the same (or the opposite) test

Merge Left Type: Join

Nodes: `T_Assign, T_Assignment, T_Statement`

This transformation will merge the selected statement (or sequence of statements) into the statement that precedes it.

Merge Right Type: Join

Nodes: `T_Assign, T_Assignment, T_Statement`

This transformation will merge the selected statement into the statement that precedes it.

Meta Trans Type: Simplify

Nodes: `T_Foreach_Cond, T_Foreach_Expn, T_Ifmatch_Cond, T_Ifmatch_Expn`

Convert a FOREACH with a long sequence of IFMATCH commands to a more efficient form

Move Comment Left Type: Move

Nodes: `T_Comment, T_Statement`

Moves the selected Comment Left.

Move Comment Right Type: Move

Nodes: `T_Comment, T_Statement`

Moves the selected Comment Right.

Move Comments Type: Rewrite

Nodes: `T_A_S`

Move Comments will move any comments which appear at the end of actions within an action system and which follow a call. The comments will be moved in front of the call. This will help tidy up the output of the Herma assembly translator.

Move To Left Type: Move

Nodes: `T_Assign`

This transformation will move the selected item to the left so that it is exchanged with the item that precedes it.

Move To Right Type: Move

Nodes: `T_Action, T_Assign, T_Assignment, T_Call, T_Comment, T_Condition, T_Definition, T_Expression, T_Guarded, T_MW_Proc_Call, T_Proc_Call, T_Push, T_Skip, T_Statement, T_Var_Lvalue, T_Where, T_X_Proc_Call`

This transformation will move the selected item to the right so that it is exchanged with the item that follows it.

Partially Join Cases Type: Rewrite Join

Nodes: `T_Comment, T_Cond`

This transformation will join any guards in an If statement which contain almost the same sequence of statements (thus reducing their number) by introducing a nested If and changing the conditions of all the guards as appropriate.

Proc To Funct Type: Rewrite

Nodes: `T_Proc, T_Var_Lvalue`

Convert a procedure with a single return parameter to a function.

Prog To Spec Type: Abstraction

Nodes: `T_Abort, T_Assert, T_Assignment, T_Comment, T_Cond, T_D_If, T_Prinflush, T_Print, T_Skip, T_Spec, T_Statement, T_Statements, T_Var`

Convert given program to an equivalent specification statement.

Prune Dispatch Type: Simplify

Nodes: `T_A_S, T_Action, T_Cond, T_Negate, T_Number, T_Or, T_Statements`

Simplify the dispatch action by removing references to dest values which do not appear in the rest of the program.

Push Pop Type: Rewrite

Nodes: `T_Pop, T_Push, T_Var`

Look for a statement sequence with a PUSH of a var followed by a POP of the same var. Put the sequence inside a VAR to show that the variable is unchanged.

Recursion To Loop Type: Rewrite

Nodes: `T_Action, T_Call, T_Proc, T_Proc_Call, T_Var`

Recursion To Loop will replace the body of a recursive action if possible by an equivalent loop structure.

Reduce Loop Type: Simplify

Nodes: `T_Floop`

Automatically make the body of a DO...OD reducible (by introducing new procedures as necessary) and either remove the loop (if it is a dummy loop) or convert the loop to a WHILE loop (if the loop is a proper sequence).

Reduce Multiple Loops Type: Simplify

Nodes: `T_Floop`

This transformation will reduce the number of multiply nested loops to a minimum.

Refine Spec Type: Refinement

Nodes: `T_Spec`

Refine a specification statement into something closer to an implementation.

Remove All Redundant Vars Type: Delete

Nodes: `T_MW_BFunct, T_MW_Funct, T_Statement, T_Statements, T_Var`

Remove All Redundant Vars applys Remove Redundant Vars to every VAR structure in the statement or sequence

Remove Dummy Loop Type: Simplify

Nodes: `T_Floop`

Remove Dummy Loop will remove a DO loop which is redundant.

Remove Redundant Vars Type: Delete

Nodes: `T_MW_BFunct, T_MW_Funct, T_Statements, T_Var, T_Where, T_X_Proc_Call`

Remove Redundant Vars takes out as many local variables as possible from the selected VAR structure. If they can all be taken out, the VAR is replaced by its (possibly modified) body.

Rename Defns Type: Rewrite

Nodes: `T_Where`

Rename PROC definitions to avoid name clashes. This allows us to move all the definitions to a single outer WHERE clause.

Rename Proc Type: Rewrite

Nodes: `T_Proc`

Rename a PROC to given new name.

Replace Accs With Value Type: Rewrite

Nodes:

This transformation will apply Replace With Value to all variables with the names a0, a1, a2 and a3 in the selected item.

Replace With Value Type: Rewrite

Nodes: `T_Aref, T_Aref_Lvalue, T_Number, T_Rel_Seg, T_Rel_Seg_Lvalue, T_Struct, T_Sub_Seg, T_Sub_Seg_Lvalue, T_Var_Lvalue, T_Variable`

This transformation will replace a variable (in an expression) by its value – provided that that value can be uniquely determined at that point in the program.

Replace With Variable Type: Rewrite

Nodes: `T_Expression`

This transformation will search for a variable which is assigned to the selected expression. If found, it will replace the expression with the variable.

Reverse Order Type: Reorder

Nodes: `T_And, T_Assignment, T_Cond, T_Equal, T_Greater, T_Greater_Eq, T_If, T_Less, T_Less_Eq, T_Max, T_Min, T_Not_Equal, T_Or, T_Plus, T_Times`

This transformation will reverse the order of most two-component items; in particular expressions, conditions and Ifs which have two branches.

Roll Loop Type: Rewrite

Nodes: `T_Cond`

Roll the first step of a WHILE loop.

Semantic Slice Type: Simplify

Nodes: `T_A_Proc_Call, T_Exit, T_Floop, T_For_In, T_Statement, T_Statements, T_Var_Lvalue, T_While`

Perform Semantic Slicing on a subset of WSL. Enter the list of variables to slice on as the data parameter.

Separate Both Type: Reorder

Nodes: `T_Assignment, T_Cond`

Separate Both will take code out to the right and the left of the selected structure.

Separate Exit Code Type: Reorder

Nodes: `T_Exit, T_Floop`

Separate Exit Code will take exit code (code which must lead to termination of the loop) out of the loop, using a flag if necessary that indicates which exit from the loop was taken.

Separate Left Type: Reorder

Nodes: `T_Assignment, T_Cond, T_D_If, T_For, T_Funct_Call, T_MW_Funct_Call, T_MW_Proc_Call, T_Proc_Call, T_Skip, T_Var, T_Where, T_While, T_X_Funct_Call, T_X_Proc_Call`

Separate_Left will take code out to the left of the selected structure. As much code as possible will be taken out; if all the statements are taken out then the original containing structure will be removed.

Separate Right Type: Reorder

Nodes: `T_Assignment, T_Cond, T_D_If, T_Floop, T_For, T_MW_Proc_Call, T_Proc_Call, T_Skip, T_Var, T_Where, T_While`

Separate_Right will take code out to the right of the selected structure.

Simple Slice Type: Simplify

Nodes: `T_Abort, T_Assert, T_Assignment, T_Comment, T_Cond, T_D_If, T_Skip, T_Spec, T_Statement, T_Statements, T_Var, T_Var_Lvalue, T_While`

Perform Simple Slicing on a subset of WSL. Enter the list of variables to slice on as the data parameter.

Simplify Action System Type: Simplify

Nodes: `T_A_S`

Simplify action system will attempt to remove actions and calls from an action system by successively applying simplifying transformations. As many of the actions as possible will be eliminated without making the program significantly larger.

Simplify Type: Simplify

Nodes:

This transformation will simplify any component as fully as possible.

Simplify If Type: Simplify

Nodes: `T_Comment, T_Cond, T_False, T_Not_Equal, T_True`

Simplify If will remove false cases from an IF statement, and any cases whose conditions imply earlier conditions. Any repeated statements which can be taken outside the if will be, and the conditions will be simplified if possible.

Simplify Item Type: Simplify

Nodes: `T_A_S, T_Assert, T_Assign, T_Assignment, T_Cond, T_Condition, T_D_If, T_Expression, T_False, T_Floop, T_Funct, T_Guarded, T_Skip, T_True, T_Var, T_Var_Lvalue, T_Where, T_While`

This transformation will simplify an item, but not recursively simplify the components inside it. In particular, the transformation will simplify expressions, conditions and degenerate conditional, local variable and loop statements.

Sort Procs Type: Rewrite

Nodes: `T_Where`

Sort the order of procs in a WHERE so that: (a) A proc appears after _all_ the procs which call it, and (b) Secondary ordering is via a depth-first search of the call graph: ie via the order in which proc calls are encountered in a depth-first scan.

Stack To Par Type: Simplify

Nodes: `T_Pop, T_Proc, T_Proc_Call, T_Push, T_Var, T_Where`

Convert stack references to a procedure parameter.

Stack To Return Type: Simplify

Nodes: `T_Proc`

Convert stack references to a procedure return parameter.

Stack To Var Type: Simplify

Nodes: `T_Pop, T_Push, T_Statements, T_Variable`

Convert a stack PUSH/POP pair to a local variable.

Static Single Assignment Type: Rewrite

Nodes: `T_For, T_Var, T_Where, T_X_Funct_Call`

Convert WSL code to Static Single Assignment form by renaming variables and adding phi function assignments.

Substitute and Delete Type: Rewrite

Nodes: `T_Action, T_Funct, T_Proc, T_Skip`

Substitute and Delete will replace all calls to an action, procedure or function with the corresponding definition, and delete the definition.

Syntactic Slice Type: Simplify

Nodes: `T_Statements`

Perform Syntactic Slicing using SSA and control dependencies. Enter the list of variables to slice on as the data parameter.

Take Out Left Type: Move

Nodes: `T_Assign, T_D_Do, T_Floop, T_For, T_Guarded, T_Statement, T_Var, T_Where, T_While`

This transformation will take the selected item out of the enclosing structure towards the left.

Take Out Of Loop Type: Move

Nodes: `T_Assign, T_Statement`

This transformation will take the selected item out of an appropriate enclosing loop towards the right.

Take Out Right Type: Move

Nodes: `T_Assign, T_D_Do, T_For, T_Guarded, T_Statement, T_Var, T_Where, T_While`

This transformation will take the selected item out of the enclosing structure towards the right.

Unfold Proc Call Type: Rewrite

Nodes: `T_Proc_Call`

Unfold the selected procedure call, replacing it with a copy of the procedure body.

Unfold Proc Calls Type: Simplify

Nodes: `T_Proc, T_Where`

Unfold Proc Calls searches for procedures which are only called once, unfolds the call and removes the procedure.

Unroll Loop Type: Rewrite

Nodes: `T_Floop, T_While`

Unroll the first step of a WHILE loop.

Use Assertion Type: Use/Apply

Nodes: `T_Assert`

Use the currently selected assertion to simplify code.

Var Pars To Val Pars Type: Rewrite
Nodes: `T_Where`

Add all VAR pars as extra value pars where needed. This is needed by the SSA transformation so that the input and output parameters can get different names.

While To Abort Type: Simplify
Nodes: `T_While`

This transformation replaces a non-terminating while loop with a conditional abort.

While To Floop Type: Rewrite
Nodes: `T_While`

Convert any WHILE loop to a DO...OD loop

While To For In Type: Simplify
Nodes: `T_Statements, T_While`

Replace a WHILE loop with an equivalent FOR x IN y loop.

While To Reduce Type: Simplify
Nodes: `T_Assert, T_While`

Replace a WHILE loop with an equivalent REDUCE or MAP.

Appendix B

MicroJava Specifics

These detailed specifications of various parts of MicroJava and MicroJava Virtual Machine are taken and adapted from [Mössenböck, 2018].

If not otherwise noted, all of these are for the 1999 version of the language.

B.1 Syntax

```
Program         = "program" ident
                  {ConstDecl | VarDecl | ClassDecl}
                  "{" {MethodDecl} "}".

ConstDecl       = "final" Type ident "="
                    (number | charConst) ";".
VarDecl         = Type ident {"," ident } ";".
ClassDecl       = "class" ident "{" {VarDecl} "}".
MethodDecl      = (Type | "void") ident "(" [FormPars] ")"
                  {VarDecl} Block.
FormPars        = Type ident {"," Type ident}.
Type            = ident ["[" "]"].

Block           = "{" {Statement} "}".
Statement       = Designator ("=" Expr | "(" [ActPars] ")"
                | "++" | "--") ";"
                | "if" "(" Condition ")" Statement ["else"
                    Statement]
```

```
                | "while" "(" Condition ")" Statement
                | "break" ";"
                | "return" [Expr] ";"
                | "read" "(" Designator ")" ";"
                | "print" "(" Expr ["," number] ")" ";"
                | Block
                | ";".
ActPars         = Expr {"," Expr}.

Condition       = CondTerm {"||" CondTerm}.
CondTerm        = CondFact {"&&" CondFact}.
CondFact        = Expr Relop Expr.
Relop           = "==" | "!=" | ">" | ">=" | "<" | "<=".
Expr            = ["-"] Term {Addop Term}.
Term            = Factor {Mulop Factor}.
Factor          = Designator ["(" [ActPars] ")"]
                | number
                | charConst
                | "new" ident ["[" Expr "]"]
                | "(" Expr ")".
Designator      = ident {"." ident | "[" Expr "]"}.
Addop           = "+" | "-".
Mulop =          "*" | "/" | "%".
```

Lexical Structure

```
Terminal classes:
        ident     = letter {letter | digit | "_"}.
        number    = digit {digit}.
        charConst = "'" char "'".  // including '\r' and '\n'
Keywords:
        program class
        if      else    while   read    print   return  break
        void    final   new
Operators:
        +       -       *       /       %       ++      --
        ==      !=      >       >=      <       <=
        &&      ||
        (       )       [       ]       {       }
        =       ;       ,       .
Comments:       // to the end of line
```

B.2 MJVM instructions specification

The following tables show the instructions of the MicroJava VM together with their encoding and their behaviour. The third column of the tables show the contents of estack before and after every instruction, for example

```
..., val, val
..., val
```

means that this instruction removes two words from estack and pushes a new word onto it. The operands of the instructions have the following meaning:

```
b      a byte
s      a short int (16 bits)
w      a word (32 bits)
```

Variables of type char are stored in the lowest byte of a word and are manipulated with word instructions (e.g. load, store). Array elements of type char are stored in a byte array and are loaded and stored with special instructions.

Loading and storing of local variables

```
1          load b          ...              Load
                           ..., val           push(local[b]);
2..5       load_n          ...              Load (n = 0..3)
                           ..., val           push(local[n]);
6          store b         ..., val         Store
                           ...                 local[b] = pop();
7..10 store_n              ..., val         Store (n = 0..3)
                           ...                 local[n] = pop();
```

Loading and storing of global variables

```
11         getstatic s     ...              Load static variable
                           ..., val           push(data[s]);
12         putstatic s     ..., val         Store static variable
                           ...                 data[s] = pop();
```

Loading and storing of object fields

```
13         getfield s      ..., adr         Load object field
                           ..., val           adr = pop()/4; push(heap[adr+s]);
14         putfield s      ..., adr, val    Store object field
                           ...                 val = pop(); adr = pop()/4;
                                               heap[adr+s] = val;
```

Loading of constants

```
15..20 const_n          ...                 Load constant (n = 0..5)
                        ..., val              push(n);
21     const_m1         ...                 Load minus one
                        ..., -1               push(-1);
22     const w          ...                 Load constant
                        ..., val              push(w);
```

Arithmetic

```
23     add              ..., val1, val2     Add
                        ..., val1+val2        push(pop() + pop());
24     sub              ..., val1, val2     Subtract
                        ..., val1-val2        push(-pop() + pop());
25     mul              ..., val1, val2     Multiply
                        ..., val1*val2        push(pop() * pop());
26     div              ..., val1, val2     Divide
                        ..., val1/val2        x = pop(); push(pop() / x);
27     rem              ..., val1, val2     Remainder
                        ..., val1%val2        x = pop(); push(pop() % x);
28     neg              ..., val            Negate
                        ..., - val            push(-pop());
29     shl              ..., val, x         Shift left
                        ..., val1             x = pop(); push(pop() << x);
30     shr              ..., val, x         Shift right (arithmetically)
                        ..., val1             x = pop(); push(pop() >> x);
31     inc b1, b2       ...                 Increment variable
                        ...                    local[b1] = local[b1] + b2;
```

Object creation

```
32     new s            ...                 New object
                        ..., adr              allocate area of s bytes;
                                              initialize area to all 0;
                                              push(adr(area));
33     newarray b       ..., n              New array
                        ..., adr              n = pop();
                                              if (b==0)
                                                alloc. array, elems of byte size;
                                              else if (b==1)
                                                alloc. array, elems of word size;
                                              initialize array to all 0;
                                              push(adr(array))
```

Array access

```
34     aload            ..., adr, i          Load array element
                        ..., val               i = pop(); adr = pop()/4+1;
                                               push(heap[adr+i]);
35     astore           ..., adr, i, val     Store array element
                        ...                    val = pop(); i = pop();
                                               adr = pop()/4+1;
                                               heap[adr+i] = val;
36     baload           ..., adr, i          Load byte array element
                        ..., val               i = pop(); adr = pop()/4+1;
                                               x = heap[adr+i/4];
```

```
                                          push(byte i%4 of x);
37    bastore          ..., adr, i, val   Store byte array element
                       ...                  val = pop(); i = pop();
                                            adr = pop()/4+1;
                                            x = heap[adr+i/4];
                                            set byte i%4 in x;
                                            heap[adr+i/4] = x;
38    arraylength      ..., adr           Get array length
                       ..., len             adr = pop();
                                            push(heap[adr]);
```

Stack manipulation

```
39    pop              ..., val           Remove topmost stack element
                       ...                  dummy = pop();
40    dup              ..., val           Duplicate topmost stack element
                       ..., val, val        x = pop(); push(x); push(x);
41    dup2             ..., v1, v2         Duplicate top two stack elements
                       ..., v1, v2, v1, v2   y = pop(); x = pop();
                                             push(x); push(y); push(x); push(y);
```

Jumps

Jump distances are relative to the beginning of the jump instruction.

```
42    jmp s                           Jump unconditionally
                                              pc = pc + s;
43..48 j<cond> s       ..., x, y      Jump conditionally (eq, ne, lt, le, gt, ge)
                       ...                  y = pop(); x = pop();
                                            if (x cond y) pc = pc + s;
```

Method call

PUSH and POP work on pstack.

```
49    call s                  Call method
                                PUSH(pc+3); pc := pc + s;
50    return                  Return
                                pc = POP();
51    enter b1, b2            Enter method
                                psize = b1; lsize = b2; // in words
                                PUSH(fp); fp = sp; sp = sp + lsize;
                                initialize frame to 0;
                                for (i=psize-1;i>=0;i--) local[i] = pop();
52    exit                    Exit method
                                sp = fp; fp = POP();
```

Input/Output

```
53    read       ...                  Read
                 ..., val               readInt(x); push(x);
54    print      ..., val, width      Print
```

```
                ...                    width = pop(); writeInt(pop(), width);
55     bread    ...                  Read byte
                ..., val               readChar(ch); push(ch);
56     bprint   ..., val, width      Print byte
                ...                    width = pop(); writeChar(pop(), width);
```

Miscellaneous

```
57     trap b          Generate run time error
                         print error message depending on b;
                         stop execution;
```

B.3 MicroJava Compiled Object File Format

- 2 bytes: "MJ"
- 4 bytes: code size in bytes
- 4 bytes: number of words for the global data
- 4 bytes: mainPC: the address of main() relative to the beginning of the code area
- n bytes: the code area (n = code size specified in the header)

The first 14 bytes are taken by the metadata, and the first actuall instruction is then at address 14 (since the addresses start from 0).

The code area contains all the instructions and their operands encoded directly as bytes.

Appendix C

Additional Data

C.1 MicroJava Transformation Metrics Tables

The transformations of *pp-lo-sp* variant of the translations of the *alpha-mj* sample set were analysed in Section 8.4.2, with an overview of the numbers shown in Table 8.7 (page 102). The following tables contain the data for the other variations used in the experiments.

Table C.1: *alpha-wsl-ht-gl-ar* transformation metrics

Metric	WSL	WSL-t	% diff
McCabe Cyclo	6.38 ± 2	3.62 ± 3	49.94 ± 24
McCabe Essential	2.88 ± 1	1.19 ± 1	55.81 ± 14
Statements	166.00 ± 138	23.62 ± 30	87.94 ± 6
CFDF	283.38 ± 236	35.50 ± 47	90.12 ± 8
Size	957.88 ± 772	182.19 ± 214	84.25 ± 9
Structure	2840.50 ± 2389	434.06 ± 515	87.44 ± 8

Table C.2: *alpha-wsl-ht-gl-sp* transformation metrics

Metric	WSL	WSL-t	% diff
McCabe Cyclo	6.38 ± 2	3.81 ± 3	48.25 ± 26
McCabe Essential	2.88 ± 1	1.62 ± 1	47.06 ± 18
Statements	166.38 ± 141	30.62 ± 41	85.00 ± 7
CFDF	285.38 ± 245	50.94 ± 71	86.12 ± 8
Size	921.81 ± 758	204.12 ± 254	82.06 ± 8
Structure	2747.75 ± 2364	493.19 ± 625	85.38 ± 6

Table C.3: *alpha-wsl-ht-lo-ar* transformation metrics

Metric	WSL	WSL-t	% diff
McCabe Cyclo	8.62 ± 5	3.44 ± 3	63.25 ± 12
McCabe Essential	2.88 ± 1	1.00 ± 0	58.94 ± 14
Statements	193.44 ± 166	20.81 ± 24	89.88 ± 5
CFDF	320.12 ± 273	32.31 ± 40	91.38 ± 7
Size	1148.19 ± 962	174.19 ± 193	86.62 ± 8
Structure	3446.06 ± 2987	413.50 ± 463	89.25 ± 6

Table C.4: *alpha-wsl-ht-lo-sp* transformation metrics

Metric	WSL	WSL-t	% diff
McCabe Cyclo	8.62 ± 5	3.50 ± 3	62.94 ± 12
McCabe Essential	2.88 ± 1	1.00 ± 0	58.94 ± 14
Statements	193.81 ± 169	22.00 ± 24	88.56 ± 4
CFDF	322.12 ± 282	36.19 ± 43	89.38 ± 5
Size	1112.12 ± 948	161.25 ± 166	86.25 ± 5
Structure	3353.31 ± 2961	382.75 ± 398	88.81 ± 4

Table C.5: *alpha-wsl-pp-gl-ar* transformation metrics

Metric	WSL	WSL-t	% diff
McCabe Cyclo	6.38 ± 2	3.38 ± 3	53.94 ± 23
McCabe Essential	2.88 ± 1	1.38 ± 1	52.06 ± 15
Statements	139.31 ± 115	16.31 ± 22	89.94 ± 5
CFDF	202.94 ± 166	20.25 ± 29	92.50 ± 5
Size	635.75 ± 490	119.50 ± 131	84.44 ± 7
Structure	1901.56 ± 1571	262.38 ± 299	88.38 ± 5

Table C.6: *alpha-wsl-pp-gl-sp* transformation metrics

Metric	WSL	WSL-t	% diff
McCabe Cyclo	6.38 ± 2	3.38 ± 3	53.94 ± 23
McCabe Essential	2.88 ± 1	1.50 ± 1	49.56 ± 17
Statements	139.00 ± 117	22.44 ± 31	87.50 ± 7
CFDF	202.94 ± 170	29.75 ± 45	90.25 ± 9
Size	591.75 ± 459	134.44 ± 159	82.38 ± 9
Structure	1785.56 ± 1496	297.31 ± 360	87.00 ± 7

Table C.7: *alpha-wsl-pp-lo-ar* 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

Appendix D

Source Code

D.1 The Hill Climbing program

```
C:"
================================================================================

Hill Climbing Automated Transformation Selection program
Copyright (C) 2018 Martin Ward (martin@gkc.org.uk)
                   Doni Pracner (doni.pracner@dmi.uns.ac.rs)

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
================================================================================
    ";

MW_PROC @HC_Main() ==
  VAR < trs := ARRAY(200, 0), i := 0, wanted := < >, whole := < >,
        pre_trans := < >,  tr := 0,
        prog := "simple5.wsl", base := "", result := "",
        count := 1000, done := 0, Argv := ARGV,
```

```
        hc_version_string := "hc-18-jan-1",
        log_failure := 1,
        minimal_output := 1,
        total_transformations_tried := 0, writeout_mod := 100,
        temp_folder := "hc-temp-versions/", use_temp_folder := 1, last
            := 0 >:

  C:" Transformations to use: (not sure about TR_Collapse_Action_System
      !) ";

  wanted := @Make_Set(<

    TR_Absorb_Left, TR_Absorb_Right,
    TR_Add_Assertion, TR_Add_Left,
    TR_Add_Loop_To_Action, TR_Align_Nested_Statements,
    TR_Collapse_Action_System, TR_Combine_Wheres,
        TR_Constant_Propagation,
    TR_D_Do_To_Floop, TR_Decrement_Statement, TR_Delete_All_Assertions,
    TR_Delete_All_Skips, TR_Delete_Item,
    TR_Delete_Redundant_Statement, TR_Delete_Unreachable_Code,
    TR_Delete_What_Follows, TR_Double_To_Single_Loop,
    TR_Else_If_To_Elsif, TR_Elsif_To_Else_If,
    TR_Expand_And_Separate, TR_Expand_Call,
    TR_Expand_Forward, TR_Floop_To_While,
    TR_For_In_To_Reduce, TR_For_To_While,
        TR_Force_Double_To_Single_Loop,
    TR_Fully_Absorb_Right, TR_Fully_Expand_Forward,
        TR_Increment_Statement,
    TR_Insert_Assertion, TR_Join_All_Cases, TR_Join_Cases_Left,
    TR_Join_Cases_Right, TR_Make_Loop, TR_Make_Reducible,
    TR_Merge_Calls_In_Action, TR_Merge_Calls_In_System,
        TR_Merge_Cond_Right,
    TR_Merge_Left, TR_Merge_Right, TR_Move_Comment_Left,
    TR_Move_Comment_Right, TR_Move_Comments, TR_Move_To_Left,
    TR_Move_To_Right, TR_Partially_Join_Cases, TR_Push_Pop,
    TR_Recursion_To_Loop, TR_Reduce_Loop, TR_Reduce_Multiple_Loops,
    TR_Remove_Dummy_Loop, TR_Remove_Redundant_Vars, TR_Reverse_Order,
    TR_Roll_Loop, TR_Separate_Both, TR_Separate_Exit_Code,
        TR_Separate_Left,
    TR_Separate_Right, TR_Simplify, TR_Simplify_Action_System,
    TR_Simplify_If, TR_Simplify_Item, TR_Substitute_And_Delete,
    TR_Take_Out_Left, TR_Take_Out_Of_Loop, TR_Take_Out_Right,
    TR_Unroll_Loop, TR_Use_Assertion,
    TR_Stack_To_Var, TR_Stack_To_Par,
    TR_Proc_To_Funct, TR_Stack_To_Return, TR_Array_To_Vars,
    TR_While_To_Abort, TR_While_To_Floop, TR_While_To_For_In,
        TR_While_To_Reduce

  >);
```

```
C:" Transformations which only make sense when applied to the whole
    program: ";

whole := @Make_Set(< TR_Constant_Propagation, TR_Simplify,
    TR_Delete_All_Redundant >);

C:" Transformations which are potentially useful preparation for ";
C:" another transformation: ";

trans1 := @Make_Set(<
  TR_Absorb_Left, TR_Absorb_Right, TR_Constant_Propagation,
  TR_Make_Loop, TR_Move_To_Left, TR_Remove_Redundant_Vars,
  TR_Separate_Both, TR_Separate_Right, TR_Substitute_And_Delete,
  TR_While_To_Floop
>);

trans2 := @Make_Set(<
  TR_Absorb_Left, TR_Absorb_Right, TR_Constant_Propagation,
  TR_Delete_Redundant_Statement,
  TR_Remove_Redundant_Vars, TR_Substitute_And_Delete
>);

IF minimal_output = 0 THEN PRINT("Found ", LENGTH(wanted), "
    transformations.") FI;

Argv := TAIL(Argv);
IF Argv = < > THEN Argv := <prog> FI;

FOR prog IN Argv DO

  base := prog;
  IF SLENGTH(base) > 4 AND SUBSTR(base, SLENGTH(base) - 4, 4) = ".wsl
      "
    THEN base := SUBSTR(base, 0, SLENGTH(base) - 4) FI;
  result := base ++ "_tr.wsl";
  @Write_To(base ++ ".log");

  IF use_temp_folder = 1 THEN
     C:"other versions should go to the temp folder";

     C:"check for folders in path, set temp folder";
     last := @Last_Index(base,"/");
     IF last > 0 THEN
              temp_folder := SUBSTR(base,0,last+1) ++ temp_folder;
              base := temp_folder ++ SUBSTR(base,last+1);
      ELSE
              base := temp_folder ++ base; FI;
      @Create_Folder(temp_folder) FI;
```

```
    @WL("Hill Climbing");
    @WL("version:" ++ hc_version_string);
    @WL("Input file: " ++ prog);
    @WL("Output file: " ++ result);

    @New_Program(@Parse_File(prog, T_Statements));


    @WL("Found " ++ @String(LENGTH(wanted)) ++ " transformations.");
    @WL("");

    done := 0;
    WHILE done >= 0 DO
      WHILE done >= 0 DO
        PRINT("----------------- Level 1 ---------------");
        @WL  ("----------------- Level 1 ---------------");
        WHILE done >= 0 DO
          @HC_Test_All(1, whole, @Program, < > VAR done, count) OD;
        done := 0;
        PRINT("----------------- Level 2 ---------------");
        @WL  ("----------------- Level 2 ---------------");
        @HC_Test_All(2, whole, @Program, < > VAR done, count) OD;
      done := 0;
      VAR < trans1 := wanted, trans2 := wanted >:
      PRINT("----------------- Level 3 ---------------");
      @WL  ("----------------- Level 3 ---------------");
      @HC_Test_All(2, whole, @Program, < > VAR done, count) ENDVAR OD;

    @Checkpoint(result);
    @WL("transformations tried:" ++ @String(total_transformations_tried
        ));
    PRINT("total transformations tried:", total_transformations_tried);
    @WL("");
    @End_Write();

    SKIP OD ENDVAR .;


C:" Return TRUE if I1 is better than I2 according to the defined
    metrics ";

MW_BFUNCT @HC_Better?(I1, I2) == :
  SKIP;
  (@Struct_Metric(I1) < @Struct_Metric(I2)) .;


MW_PROC @HC_Checkpoint(tr, prev VAR count) ==
  C:" Keep this version ";
```

```
  count := count + 1;
  @Checkpoint(base ++ "-" ++ @String(count) ++ ".wsl");
  IF minimal_output =0
    THEN PRINT(TRs_Name[tr], " at ", posn) FI;
  @WS(@String(count) ++ ": Success:");
  VAR < L :=  REVERSE(<<tr, posn>> ++ prev) >:
  WHILE NOT EMPTY?(L) DO
    pair := HEAD(L); L := TAIL(L);
    @WS(TRs_Name[pair[1]] ++ ": at <");
    @WL(@Join(",", MAP("@String", pair[2])) ++ ">");
    IF NOT EMPTY?(L) THEN @WS("   +: Success:") FI OD ENDVAR;
  IF minimal_output = 0 THEN
          @HC_Metrics("old:  ", old);
          @HC_Metrics("orig: ", orig);
          @HC_Metrics("new:  ", @Program) FI .;


MW_PROC @HC_Metrics(str, I) ==
  PRINFLUSH(str);
  FOR n IN < @Struct_Metric(I),
             @Spec_Type_Count(T_Action, I), @Spec_Type_Count(T_Call, I)
                 ,
             @Spec_Type_Count(T_A_Proc_Call, I), @McCabe(I),
             @Stat_Count(I), @Gen_Type_Count(T_Expression, I) > DO
    PRINFLUSH(n, " ") OD;
  PRINT("") .;


MW_PROC @HC_Test_All(depth, whole, orig, prev VAR done, count) ==
  VAR < tr := 0, trs := < >, old := @Program, posn := < > >:
  @Goto(< >);
  IF depth = 1 AND EMPTY?(prev)
    THEN trs := wanted
  ELSIF depth = 2 AND EMPTY?(prev)
    THEN trs := trans1
  ELSIF depth = 1 AND NOT EMPTY?(prev)
    THEN trs := trans2
    ELSE PRINT("depth = ", depth, " prev = ", LENGTH(prev));
         ERROR("-- not yet implemented") FI;
  IF EMPTY?(trs) THEN ERROR("Empty transformation list!") FI;
  tr := HEAD(trs);
  DO IF FALSE
       THEN PRINT(depth, " testing ", TRs_Name[tr], " at ", @Posn) FI;
     IF tr IN whole AND @Up?
       THEN SKIP
     ELSIF @GT(@I) IN <T_Expression, T_Condition, T_Lvalue,
                       T_Expressions, T_Lvalues>
       THEN SKIP
     ELSIF @Trans?(tr)
```

```
       THEN posn := @Posn;
            @Trans(tr, "");
            total_transformations_tried := total_transformations_tried
                + 1;
            IF total_transformations_tried MOD writeout_mod = 0
              THEN PRINT("*** transformations tried:",
                  total_transformations_tried) FI;
            IF @ST(@I) = T_Assignment AND @Size(@I) > 1
              THEN C:" don't generate parallel assignments "
            ELSIF @HC_Better?(@Program, orig)
              THEN done := tr;
                   @HC_Checkpoint(tr, prev VAR count);
                   EXIT(1)
            ELSIF depth > 1 AND NOT @Equal?(@Program, orig)
              THEN IF minimal_output = 0 THEN PRINT("== Sub-test after
                  ", TRs_Name[tr], " at ", posn) FI;
                   @HC_Test_All(depth - 1, whole, orig, <<tr, posn>> ++
                        prev
                                VAR done, count);
                   IF done > 0
                     THEN EXIT(1) FI FI;
            IF log_failure = 1
              THEN @WL("- Depth:" ++ @String(depth) ++ " Tried:" ++
                  TRs_Name[tr]) FI;
            C:" Revert program ";
            @New_Program(old);
            @Goto(posn) FI;
     C:" Move to new position. ";
     C:" Do not descend into these types: ";
     IF @Down? AND @GT(@I) NOTIN <T_Expression, T_Condition, T_Lvalue,
                                  T_Expressions, T_Lvalues>
       THEN @Down
     ELSIF @Right?
       THEN @Right
       ELSE WHILE @Up? AND NOT @Right? DO @Up OD;
            IF @Right?
              THEN @Right
              ELSE C:" Next transformation ";
                   trs := TAIL(trs);
                   IF EMPTY?(trs) THEN done := -1; EXIT(1) FI;
                   tr := HEAD(trs);
                   @Goto(< >) FI FI OD ENDVAR .;

MW_FUNCT @Last_Index(string, sep) ==
VAR< last := SLENGTH(string) - 1 >:

        WHILE last > 0 AND SUBSTR(string,last,1) <> sep DO
                last := last - 1; OD;
```

```
(last) END;

C:" Call the main routine after all other routines have been defined: "
    ;

@HC_Main;

SKIP
```

Bibliography

80286 Programmer's Reference Manual (1987). Available at `http://datasheets.chipdb.org/Intel/x86/286/manuals/286INTEL.zip`. Intel Corporation (cit. on pp. 38, 43).

Ab. Rahim, Lukman and Jon Whittle (2015). "A survey of approaches for verifying model transformations". English. In: *Software & Systems Modeling* 14.2, pp. 1003–1028. ISSN: 1619-1366. DOI: `10.1007/s10270-013-0358-0`. URL: `http://dx.doi.org/10.1007/s10270-013-0358-0` (cit. on p. 15).

Ammann, Paul and Jeff Offutt (2008). *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK. ISBN: 0-52188-038-6 (cit. on p. 14).

Arcuri, Andrea and Xin Yao (2008). "A novel co-evolutionary approach to automatic software bug fixing". In: *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE, pp. 162–168 (cit. on pp. 5, 112, 178).

Belady, Laszlo A. and Meir M. Lehman (1976). "A model of large program development". In: *IBM Syst. J.* 15.3, pp. 225–252. ISSN: 0018-8670. DOI: `dx.doi.org/10.1147/sj.153.0225` (cit. on p. 11).

Bennett, Keith H (1995). "Legacy systems: coping with success". In: *IEEE Software* 12.1, pp. 19–23. ISSN: 0740-7459. DOI: `10.1109/52.363157`. URL: `doi.ieeecomputersociety.org/10.1109/52.363157` (cit. on p. 11).

Boehm, Barry (2006). "A View of 20th and 21st Century Software Engineering". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, pp. 12–29. ISBN: 1-59593-375-1. DOI: `10.1145/1134285.1134288`. URL: `http://doi.acm.org/10.1145/1134285.1134288` (cit. on p. 10).

Booth, Andrew D and Kathleen HV Britten (1947). *General considerations in the design of an all purpose electronic digital computer*. Tech. rep. Institute for Advanced Studies, Princeton, pp. 1–28 (cit. on p. 37).

— (1949). "Principles and Progress in the Construction of High-Speed Digital Computers". In: *The Quarterly Journal of Mechanics and Applied Mathematics* 2.2, pp. 182–197 (cit. on p. 37).

Brant, John, Jason Lecerf, Thierry Goubier, and Stéphane Ducasse (2017). *Smacc: a Compiler-Compiler*. The Pharo Booklet Collection. URL: http://books.pharo.org/booklet-Smacc/ (cit. on p. 16).

Brant, John and Don Roberts (2009). "The SmaCC Transformation Engine: How to Convert Your Entire Code Base into a Different Programming Language". In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: ACM, pp. 809–810. ISBN: 978-1-60558-768-4. DOI: 10.1145/1639950.1640026. URL: http://doi.acm.org/10.1145/1639950.1640026 (cit. on p. 16).

Brooks, Frederick (1995). *The mythical man-month : essays on software engineering*. Reading, Massachusetts: Addison-Wesley Publishing Company. ISBN: 0201835959 (cit. on p. 10).

Cánovas Izquierdo, Javier Luis and Jesús García Molina (2014). "Extracting models from source code in software modernization". English. In: *Software & Systems Modeling* 13.2, pp. 713–734. ISSN: 1619-1366. DOI: 10.1007/s10270-012-0270-z. URL: http://dx.doi.org/10.1007/s10270-012-0270-z (cit. on p. 15).

Chen, Feng, Hongji Yang, Bing Qiao, and William Cheng-Chung Chu (2006). "A Formal Model Driven Approach to Dependable Software Evolution". In: *Computer Software and Applications Conference, Annual International* 1, pp. 205–214. ISSN: 0730-3157. DOI: doi.ieeecomputersociety.org/10.1109/COMPSAC.2006.10 (cit. on p. 20).

Cifuentes, Cristina and K. John Gough (1995). "Decompilation of binary programs". In: *Software: Practice and Experience* 25.7, pp. 811–829. ISSN: 1097-024X. DOI: 10.1002/spe.4380250706. URL: http://dx.doi.org/10.1002/spe.4380250706 (cit. on p. 13).

Cifuentes, Cristina and Doug Simon (2000). "Procedure Abstraction Recovery from Binary Code". In: *Proceedings of the Conference on Software Maintenance and Reengineering*. CSMR '00. Washington, DC, USA: IEEE Computer Society, pp. 55–64. ISBN: 0-7695-0546-5. URL:

http://dl.acm.org/citation.cfm?id=518900.795261 (cit. on p. 13).

Cifuentes, Cristina, Doug Simon, and Antoine Fraboulet (1998). "Assembly to High-Level Language Translation". In: *Proceedings of the International Conference on Software Maintenance*. ICSM '98. Washington, DC, USA: IEEE Computer Society, pp. 228–. ISBN: 0-8186-8779-7 (cit. on p. 38).

Cifuentes, Cristina, Mike Van Emmerik, and Trent Waddington (2001). "Computer Security Analysis through Decompilation and High-Level Debugging". In: *Proceedings Eighth Working Conference on Reverse Engineering(WCRE)*, pp. 375–380. DOI: 10.1109/WCRE.2001.957846. URL: doi.ieeecomputersociety.org/10.1109/WCRE.2001.957846 (cit. on p. 13).

DeMarco, Tom (1986). *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall. ISBN: 978-0131717114 (cit. on p. 16).

Demeyer, Serge, Stéphane Ducasse, and Oscar Nierstrasz (2008). *Object-Oriented Reengineering Patterns*. Square Bracket Associates. ISBN: 978-3-9523341-2-6. URL: http://scg.unibe.ch/download/oorp/ (cit. on p. 14).

Deming, W. Edwards (1991). *The New Economics for Industry, Government, Education*. Mit Press. ISBN: 978-0262541169 (cit. on p. 16).

Dijkstra, Edsger W. (1972). "The Humble Programmer". In: *Commun. ACM* 15.10, pp. 859–866. ISSN: 0001-0782. DOI: 10.1145/355604.361591. URL: http://doi.acm.org/10.1145/355604.361591 (cit. on p. 9).

Ducasse, Stéphane, Michele Lanza, and Sander Tichelaar (2000). "Moose: an extensible language-independent environment for reengineering object-oriented systems". In: *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*. Vol. 4, pp. 24–30 (cit. on p. 14).

Fatiregun, Deji (2006). "Search-based Program Transformation For Amorphous Slicing and Program Comprehension". Author name on the cover is "Ayọdèji"; papers use just "Deji". PhD thesis. King's College, London (cit. on pp. 112, 178).

Fatiregun, Deji, Mark Harman, and Robert M. Hierons (2004). "Evolving transformation sequences using genetic algorithms". In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*, pp. 65–74. DOI: 10.1109/SCAM.2004.11 (cit. on pp. 5, 112, 178).

Fernandez-Ramil, Juan, Angela Lozano, Michel Wermelinger, and Andrea Capiluppi (2008). "Empirical studies of open source evolution". In: *Software Evolution*. Ed. by Tom Mens and Serge Demeyer. .III. Springer Berlin

Heidelberg, pp. 263–288. DOI: dx.doi.org/doi:10.1007/978-3-540-76440-3_11 (cit. on p. 13).

Glover, Fred (1989). "Tabu Search – Part 1". In: *ORSA Journal on Computing* 1.2, pp. 190–206. DOI: 10.1287/ijoc.1.3.190 (cit. on pp. 116, 179).

Godfrey, M.W. and Q. Tu (2000). "Evolution in open source software: A case study". In: *Int'l Conf. Software Maintenance (ICSM), Los Alamitos, California*. IEEE Computer Society Press, pp. 131–142 (cit. on p. 13).

Hemel, Zef, Lennart C.L. Kats, Danny M. Groenewegen, and Eelco Visser (2010). "Code generation by model transformation: a case study in transformation modularity". English. In: *Software & Systems Modeling* 9.3, pp. 375–402. ISSN: 1619-1366. DOI: 10.1007/s10270-009-0136-1. URL: http://dx.doi.org/10.1007/s10270-009-0136-1 (cit. on p. 14).

Herraiz, Israel, Daniel Rodriguez, Gregorio Robles, and Jesus M. Gonzalez-Barahona (2013). "The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review". In: *ACM Comput. Surv.* 46.2, 28:1–28:28. ISSN: 0360-0300. DOI: 10.1145/2543581.2543595. URL: http://doi.acm.org/10.1145/2543581.2543595 (cit. on p. 12).

Hills, Mark and Paul Klint (2014). "PHP air: Analyzing PHP systems with rascal". In: *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, pp. 454–457 (cit. on p. 15).

Hulaas, Jarle and Walter Binder (2008). "Program transformations for light-weight cpu accounting and control in the java virtual machine". In: *Higher-Order and Symbolic Computation* 21.1-2, pp. 119–146 (cit. on p. 48).

Israeli, Ayelet and Dror G. Feitelson (2010). "The Linux Kernel As a Case Study in Software Evolution". In: *J. Syst. Softw.* 83.3, pp. 485–501. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.09.042. URL: http://dx.doi.org/10.1016/j.jss.2009.09.042 (cit. on p. 13).

Kaner, Cem, Senior Member, and Walter P. Bond (2004). "Software Engineering Metrics: What Do They Measure and How Do We Know?" In: *In METRICS 2004. IEEE CS*. Press (cit. on p. 16).

Kats, Lennart C.L. and Eelco Visser (2010). "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs". In: *SIGPLAN Not.* 45.10, pp. 444–463. ISSN: 0362-1340. DOI: 10.1145/1932682.1869497. URL: http://doi.acm.org/10.1145/1932682.1869497 (cit. on p. 15).

Klint, Paul, Tijs van der Storm, and Jurgen Vinju (2011). "EASY Meta-programming with Rascal". In: *Proceedings of the 3rd International*

Summer School Conference on Generative and Transformational Techniques in Software Engineering III. GTTSE'09. Braga, Portugal: Springer-Verlag, pp. 222–289. ISBN: 978-3-642-18022-4 (cit. on p. 15).

Kuleshov, Eugene (2007). "Using the ASM framework to implement common Java bytecode transformation patterns". In: *Aspect-Oriented Software Development 2007*. Vancouver, Canada (cit. on p. 48).

Kuperberg, Michael, Martin Krogmann, and Ralf Reussner (2008). "ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations". In: *Proceedings of the 3rd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation , Budapest, Hungary, 5th April 2008 (ETAPS 2008, 11th European Joint Conferences on Theory and Practice of Software)* (cit. on p. 49).

Ladkau, Matthias (2007). *FermaT Maintenance Environment Tutorial*. Tech. rep. Software Technology Research Laboratory, De Montfort University, Leicester (cit. on pp. 19, 24).

— (2009). "A Wide Spectrum Type System for Transformation Theory". PhD thesis. De Montfort University, Leicester (cit. on pp. 20, 81, 82).

Lam, Patrick, Eric Bodden, Ondřej Lhoták, and Laurie Hendren (2011). "The Soot framework for Java program analysis: a retrospective". In: *Cetus Users and Compiler Infrastructure Workshop (in conjuction with PACT 2011)*. Galveston Island, TX (cit. on p. 48).

Le Goues, Claire, Stephanie Forrest, and Westley Weimer (2013). "Current challenges in automatic software repair". In: *Software Quality Journal* 21.3, pp. 421–443. DOI: 10.1007/s11219-013-9208-0 (cit. on p. 14).

Le Goues, Claire, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer (2012). "GenProg: A Generic Method for Automatic Software Repair". In: *IEEE Trans. Software Eng.* 38.1, pp. 54–72. DOI: 10.1109/TSE.2011.104 (cit. on pp. 14, 178).

Lehman, Meir M. (1997). "Laws of software evolution revisited". In: *Lecture Notes in Computer Science 1149*. Springer-Verlag, pp. 108–124 (cit. on p. 11).

— (2001). *FEAST/2 Final Report*. Tech. rep. Dept. of Comp., Imp. Col., London, (cit. on p. 12).

Lehman, Meir M. and Laszlo A. Belady, eds. (1985). *Program Evolution: Processes of Software Change*. San Diego, CA, USA: Academic Press Professional, Inc. ISBN: 0-12-442440-6 (cit. on p. 11).

Li, Yanchuan and Gordon Fraser (2011). "Bytecode testability transformation". In: *Search Based Software Engineering*. Springer, pp. 237–251 (cit. on p. 49).

Lincke, Rüdiger, Jonas Lundberg, and Welf Löwe (2008). "Comparing Software Metrics Tools". In: *Proceedings of the 2008 International Symposium on*

Software Testing and Analysis. ISSTA '08. Seattle, WA, USA: ACM, pp. 131–142. ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390648. URL: http://doi.acm.org/10.1145/1390630.1390648 (cit. on p. 16).

Lindholm, Tim, Frank Yellin, Gilad Bracha, and Alex Buckley (2015). *The Java Virtual Machine Specification, Java SE 8 Edition*. Oracle Ameria Inc. (cit. on p. 50).

Lounas, Razika, Mohamed Mezghiche, and Jean-Louis Lanet (2013). "Towards a General Framework for Formal Reasoning about Java Bytecode Transformation." In: *Fourth International Symposium on Symbolic Computation in Software Science – SCSS*, pp. 63–73 (cit. on p. 49).

Mössenböck, Hanspeter (2018). "Language Specification and Compiler Construction". Handouts for the course. URL: http://ssw.jku.at/Misc/CC/ (cit. on pp. 49, 139).

Naur, Peter and Brian Randell, eds. (1969). *Software Engineering: Report on a Conference sponsored by the NATO Science Committee*. Garmisch, Germany, 7th to 11th October 1968. Brussels: Scientific Affairs Division, NATO. 231 pp. (cit. on p. 9).

Novak, Jernej and Gordana Rakić (2010). "Comparison of software metrics tools for .NET". In: *Proc. of 13th International Multiconference Information Society-IS, Vol A*, pp. 231–234 (cit. on p. 16).

Parnas, David Lorge (1994). "Software Aging". In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE '94. Sorrento, Italy: IEEE Computer Society Press, pp. 279–287. ISBN: 0-8186-5855-X. URL: http://dl.acm.org/citation.cfm?id=257734.257788 (cit. on pp. 9, 10).

Pirzada, Shamim S. (1988). "An statistical examination of the evolution of the Unix system". PhD thesis. Dept. of Computing, Imperial College, London (cit. on p. 13).

Pracner, Doni and Zoran Budimac (2011a). "Restructuring Assembly Code Using Formal Transformations". In: *Proc. of Symposium on Computer Languages, Implementations and Tools (SCLIT 2011) held within International Conference on Numerical Analysis and Applied Mathematics ICNAAM 2011*. Ed. by Theodore E. Simos. Vol. 1389. AIP proceedings. Kassandra, Halkidiki, Greece, pp. 845–848. ISBN: 978-0-7354-0956-9 (cit. on pp. 5, 7, 59).

— (2011b). "Understanding Old Assembly Code Using WSL". In: *Proc. of the 14th International Multiconference on Information Society (IS 2011)*. Ed. by Marko Bohanec et al. Vol. A. Ljubljana, Slovenia: Institute "Jožef Stefan",

Ljubljana, pp. 171–174. ISBN: 978-961-264-035-4. URL: http://is.ijs.si (cit. on pp. 5, 7).

— (2013). "Transforming Low-level Languages Using FermaT and WSL". In: *Proceedings of the 2nd Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications*. Ed. by Zoran Budimac. Vol. 1053. CEUR-WS.org, pp. 71–78. URL: http://ceur-ws.org/Vol-1053/ (cit. on pp. 5, 6).

— (2017a). "A Practical Tutorial for FermaT and WSL Transformations". In: *Proceedings of the 6th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications*. Ed. by Zoran Budimac. Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Serbia, 11:01–11:08. URL: http://ceur-ws.org/Vol-1938/ (cit. on p. 6).

— (2017b). "Enabling code transformations with FermaT on simplified bytecode". In: *Journal of Software: Evolution and Process* 29.5, e1857–n/a. ISSN: 2047-7481. DOI: 10.1002/smr.1857. URL: http://dx.doi.org/10.1002/smr.1857 (cit. on pp. 5, 6, 28).

Rakić, Gordana (2015). "Extendable and Adaptable Framework for Input Language Independent Static Analysis". PhD thesis. University of Novi Sad, Faculty of Sciences (cit. on pp. 17, 116, 179).

Reiss, Steven P (2008). "Controlled dynamic performance analysis". In: *Proceedings of the 7th international workshop on Software and performance*. ACM, pp. 43–54 (cit. on p. 48).

Schulte, Eric M., Stephanie Forrest, and Westley Weimer (2010). "Automated program repair through the evolution of assembly code". In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE '10)*. Ed. by Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto. ACM, pp. 313–316. DOI: 10.1145/1858996.1859059 (cit. on pp. 14, 49).

Schwartz, Edward J., JongHyup Lee, Maverick Woo, and David Brumley (2013). "Native x86 Decompilation using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring". In: *Proceedings of the USENIX Security '13 Symposium*, pp. 353–368 (cit. on p. 38).

Schwartz, Jules I. (1978). "The Development of JOVIAL". In: *SIGPLAN Not.* 13.8, pp. 203–214. ISSN: 0362-1340. DOI: 10.1145/960118.808385. URL: http://doi.acm.org/10.1145/960118.808385 (cit. on p. 20).

Sneed, Harry (2000). "Encapsulation of legacy software: A technique for reusing legacy software components". In: *Annals of Software Engineering* 9 (1). 10.1023/A:1018989111417, pp. 293–313. ISSN: 1022-7091. URL: http://dx.doi.org/10.1023/A:1018989111417 (cit. on p. 13).

Tange, Ole (2011). "GNU Parallel - The Command-Line Power Tool". In: *;login: The USENIX Magazine* 36.1, pp. 42–47. DOI: http://dx.doi.org/10.5281/zenodo.16303. URL: http://www.gnu.org/s/parallel (cit. on p. 94).

Tripathy, Priyadarshi and Kshirasagar Naik (2014). *Software Evolution and Maintenance*. A Practitioner's Approach. John Wiley & Sons. 416 pp. ISBN: 978-0-470-60341-3 (cit. on pp. 10, 11, 13).

Turbo Assembler 2.0 User's Guide (1990). Borland International (cit. on p. 39).

Turing, Alan Mathison (1937). "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1, pp. 230–265. DOI: 10.1112/plms/s2-42.1.230. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230 (cit. on pp. 90, 108).

Visser, Eelco (2004). "Program Transformation with Stratego/XT". In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Ed. by Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 216–238. ISBN: 978-3-540-25935-0. DOI: 10.1007/978-3-540-25935-0_13. URL: https://doi.org/10.1007/978-3-540-25935-0_13 (cit. on p. 15).

— (2005). "A Survey of Strategies in Rule-Based Program Transformation Systems". In: *Journal of Symbolic Computation* 40.1. Ed. by Bernhard Gramlich and Salvador Lucas. Special issue on Reduction Strategies in Rewriting and Programming, pp. 831–873 (cit. on p. 15).

Wagner, Christian (2014). *Model-Driven Software Migration: A Methodology*. Reengineering, Recovery and Modernization of Legacy Systems. Springer Vieweg, Wiesbaden. 304 pp. ISBN: 978-3-658-05269-0. DOI: 10.1007/978-3-658-05270-6 (cit. on pp. 10, 11, 13).

Ward, Martin (1989). "Proving Program Refinements and Transformations". PhD thesis. Oxford University (cit. on pp. 15, 20, 81).

— (1999). "Assembler to C Migration using the FermaT Transformation System". In: *IEEE International Conference on Software Maintenance (ICSM'99)*. IEEE Computer Society Press, pp. 67–76 (cit. on pp. 19, 59, 168).

— (2000). "Reverse Engineering from Assembler to Formal Specifications via Program Transformations". In: *7th Working Conference on Reverse Engineering, Brisbane, Queensland, Australia*. IEEE Computer Society, pp. 11–20 (cit. on p. 59).

— (2004). "Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations". In: *Science of Computer Programming, Special Issue on*

Program Transformation 52/1-3, pp. 213–255. DOI: dx.doi.org/10.1016/j.scico.2004.03.007 (cit. on pp. 3, 19, 22, 23, 59, 168).

— (2013). "Assembler restructuring in FermaT". In: *SCAM*. IEEE, pp. 147–156. DOI: http://dx.doi.org/10.1109/SCAM.2013.6648196 (cit. on pp. 19, 111, 114, 168, 177).

Ward, Martin and Keith H Bennett (1995a). "Formal methods for legacy systems". In: *Journal of Software: Evolution and Process* 7.3, pp. 203–219. DOI: 10.1002/smr.4360070305 (cit. on p. 15).

— (1995b). "Formal methods to aid the evolution of software". In: *International Journal of Software Engineering and Knowledge Engineering* 05.01, pp. 25–47. DOI: 10.1142/S0218194095000034 (cit. on p. 19).

Ward, Martin, Tim Hardcastle, and Stefan Natelberg (2008). *WSL Programmer's Reference Manual* (cit. on pp. 19, 24).

Ward, Martin and Hussein Zedan (2014). "Provably correct derivation of algorithms using FermaT". English. In: *Formal Aspects of Computing* 26.5, pp. 993–1031. ISSN: 0934-5043. DOI: 10.1007/s00165-013-0287-2. URL: http://dx.doi.org/10.1007/s00165-013-0287-2 (cit. on p. 19).

— (2017). "The formal semantics of program slicing for nonterminating computations". In: *Journal of Software: Evolution and Process* 29.1. e1803 smr.1803, e1803–n/a. ISSN: 2047-7481. DOI: 10.1002/smr.1803. URL: http://dx.doi.org/10.1002/smr.1803 (cit. on p. 19).

Ward, Martin, Hussein Zedan, and Tim Hardcastle (2004). "Legacy Assembler Reengineering and Migration". In: *ICSM2004, The 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society, pp. 157–166. DOI: 10.1109/ICSM.2004.1357800 (cit. on pp. 19, 59, 67, 111, 168, 177).

Wirth, Niklaus (1995). "A plea for lean software". In: *Computer* 28.2, pp. 64–68. ISSN: 0018-9162. DOI: 10.1109/2.348001 (cit. on p. 9).

— (1996). *Compiler Construction*. Addison-Wesley. ISBN: 0201403536 (cit. on p. 49).

Wolpert, David H. and William G. Macready (1997). "No free lunch theorems for optimization". In: *IEEE Transactions on Evolutionary Computation* 1.1, pp. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893 (cit. on pp. 5, 107).

Yang, Hongji and Martin Ward (2003). *Succesful Evolution of Software Systems*. Norwood, MA, USA: Artech House. ISBN: 1-58053-349-3 (cit. on pp. 10, 14, 20).

Younger, E.J., Keith H Bennett, and Z. Luo (1997). "A Formal Transformation and Refinement Method for Concurrent Programs". In: *Software Maintenance, IEEE International Conference on*, pp. 287–295. ISSN: 1063-6773. DOI: 10.1109/ICSM.1997.624256 (cit. on p. 20).

Prošireni izvod

Sa razvojem i pojeftinjenjem računara dolazi do njihove sve veće rasprostranjenosti. Danas su oni integrisani u skoro svaki aspekt naših života. I na najmanjim današnjim računarima se nekad pokreću veoma kompleksni softveri. Zbog ove kompleksnosti, a i zbog toga što softver nema fizička svojstva i lako se kopira, danas se sve ređe razvija potpuno nov i nezavistan softver. Samim tim postoji sve veća potreba da se stari softver menja i unapređuje kako bi se mogao integrisati u nova okruženja. Veoma važan deo svakog takvog procesa je razumevanje kako originalni softver radi. Često su dostupne samo konačne, izvršne verzije programa, dok originalni izvorni kôd i dokumentacija možda nisu dostupni, možda su zastareli, a možda nikad nisu ni postojali.

Softver u svojoj izvršnoj formi je tipično na nekom niskom nivou apstrakcije. Jedan tip ovoga su mašinski programi specifični za hardver na kome se pokreću i u skladu sa tim se direktno i izvršavaju na njemu. Drugi tip su uglavnom virtuelne mašine ili interpreteri na niski nivo, kao što su kompajlirane Java klase u odgovarajući bajtkôd, p-code za jezik Pascal ili .NET kôd. Izvršne verzije se dobijaju ili kompajliranjem iz neke reprezentacije visokog nivoa, ili se pišu direktno.

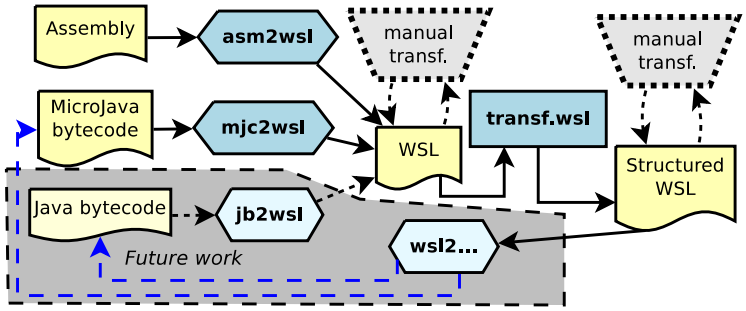
U okviru ove teze se predstavlja pristup radu sa kôdom niskog nivoa koji omogućava automatsko restrukturiranje i podizanje na više nivoe. Samim tim postaje mnogo lakše razumeti logiku programa što smanjuje vreme razvoja. Krajnji rezultat procesa je u najboljem slučaju jasan kôd visokog nivoa. U najgorim slučajevima je urađen bar deo procesa reinženjeringa.

Proces

Proces predstavljen u tezi se dobrim delom oslanja na jezik *WSL (eng. Wide Spectrum Language – jezik širokog spektra)*, koji je upravo dizajniran tako da omogućuje rad sa različitim nivoima programa, od apstraktnih specifikacija do izvršnog

kôda i kôda niskog nivoa. Jezik je kreirao dr Martin Ward, a trenutna implementacija je deo sistema *FermaT*. U okviru ovog sistema je ugrađen veliki broj transformacija programa koje očuvavaju semantiku i u skladu sa tim su veoma primenljive na restrukturiranje kôda. Sam sistem je već uspešno bio primenjivan u restrukturiranju industrijskih asemblerskih biblioteka u održive C i COBOL programe [Ward, 1999, 2004, 2013; Ward, Zedan, and Hardcastle, 2004].

Proces je dizajniran tako da bude fleksibilan i sastoji se od više nezavisnih alata (Slika 1.1). Samim tim je lako menjati proces po potrebi, ali i upotrebiti razvijene alate u drugim procesima. Tipično se mogu razlikovati dva glavna koraka. Prvi je prevođenje u WSL, a drugi su transformacije u samom WSL-u. Za potrebe prevođenja su razvijena dva alata, jedan koji radi sa podskupom x86 asemblera i drugi koji radi sa MikroJava bajtkôdom. Rezultat prevođenja je program niskog nivoa u WSL jeziku, dok je krajnji proizvod procesa strukturirana verzija programa.



Ponovljena Slika 1.1: Prikaz toka rada sa kôdom niskog nivoa

Prevodioci

U okviru ovog rada razvijena su dva prevodioca iz jezika niskog nivoa u WSL. Kod oba je fokus bio na što direktnijem prevođenju sa simuliranjem svih propratnih efekata instrukcija. U ovoj fazi rada nema pokušaja da se optimizuje veličina izlaznog programa, što je značajno smanjivalo vreme razvoja alata, kao i potrebe za detaljnim testiranjem takvih optimizacija. Kasnije transformacije su se pokazale sposobnim da uklone višak kôda i opravdale ovaj pristup razvoju.

Grupa dr Martina Ward-a se već bavila prevođenjem iz asemblera u WSL. Njihovi alati prevode delove asemblera, a neke detalje, naročito pristup memoriji mapiraju u specijalnim fajlovima. To omogućava da se glavni delovi kôda transformišu u

FermaT-u, a da se potom pri prevođenju u ciljni jezik iskoriste pomoćna mapiranja za dobijanje kompletnog programa. Mana ovog pristupa je što se WSL verzija ne može zapravo pokrenuti i testirati, nego je to tek moguće sa prevedenim verzijama kad je ceo proces završen.

Prevodilac za asembler

Prvi alat koji je razvijen je *asm2wsl*. Njegov ulaz su asemblerski programi, pisani za x86 procesore, u dijalektu za *Microsoft Macro Assembler (MASM)*, koji takođe koristi i *Borlandov Turbo Assembler (TASM)*. Ideja ovog alata je bila da u potpunosti predstavi ponašanje asemblerskog programa u WSL-u, tako da on može da se pokrene, testira i modifikuje u samom okruženju. Pošto to istovremeno povlači sa sobom i probleme u prepoznavanju nekih struktura, dosta je smanjen broj mogućih ulaznih programa. Od početka je alat koncipiran tako da pretpostavlja da radi sa 80286 procesorom, pri čemu je glavna prednost što na njemu postoje promenljive u samo dve veličine, 8 i 16 bita. Kod kasnijih procesora je uvedeno i 32, a kasnije i 64 bita, ali su osnovni koncepti ostali isti. Za ovaj prototip koji je trebao da pokaže izvodljivost koncepta, dovoljan je bio i jednostavniji procesor.

Za potrebe funkcionisanja programa se u prevedene programe uvodi koncept „virtuelnog" procesora. U suštini je to skup promenljivih koje predstavljaju stanje procesora, odnosno njegovih različitih registara. Sam prevodilac funkcioniše tako što prevodi pojedinačne instrukcije i kako one zavise od i utiču na stanja procesora. Slične strukture postoje i za emulaciju ponašanja procesorskog steka. Promenljive u programu se tretiraju kao nezavisne, a ne samo labele na adrese u memoriji. Sve ovo naravno dodatno ograničava skup ulaznih programa koji će biti ispravno tretirani, međutim sa druge strane omogućava da se ispravno prepoznati programi mogu dovesti do visokog nivoa apstrakcije.

Za predstavljanje celog asemblerskog programa se koristi specijalna struktura *action system* koja je ugrađena u WSL. Ona je upravo i dizajnirana da efikasno predstavi takozvani „špageti" kôd koji se sastoji od mnogo skokova i labela. Sistem se sastoji od kolekcije procedura bez parametara koje mogu da zovu jedna drugu. Kad se procedura završi, vraća kontrolu pozivaocu. Početna akcija se precizira pri definiciji samog sistema. Izvršavanje sistema akcija se završava kad se ta početna akcija završi, ili specijalno ako se pozove predefinisano ime `z` koje momentalno prekida izvršavanje. U zavisnosti od toga kakvi odnosi među akcijama, može se definisati nekoliko tipova akcionih sistema. Prvi je *rekurzivan* sistem akcija u kome sve pozvane akcije normalno završavaju svoj rad i vraćaju kontrolu pozivaocu. Drugi su *regularni* sistemi u kojima nijedna akcija ne vraća kontrolu pozivaocu, nego se

kompletan sistem sastoji od serije poziva, sve do konačnog poziva akciji z. Svi ostali sistemi se nazivaju *hibridnima*.

Alat *asm2wsl*, kao i prethodni asemblerski prevodioci, proizvodi *regularne* akcione sisteme. Postoji više transformacija, naročito onih koje pojednostavljuju akcione sisteme, koje prepoznaju ovu osobinu. U skladu sa tim mogu da veoma efikasno pretvore sistem u neku strukturu višeg nivoa, kao što je grananje ili petlja. U osnovi se labele u asembleru prevode kao počeci akcija u WSL. Ovo omogućava da se kasniji skokovi na te labele prevode jednostavnim pozivima tih imena, bez potrebe da se pamte adrese i slično.

Makro definicije se u ovoj verziji alata uopšte ne prevode, već se samo postavljaju kao komentari u programu, a korisnik dobija poruke upozorenja o ovome. Glavni razlog je što one mogu biti veoma kompleksne i dosta zavise od konkretnog asemblera koji se koristi. Međutim ovaj mehanizam je iskorišćen za prepoznavanje nekih specijalnih imena koji se onda direktno prevode u WSL ekvivalente. Primarno je iskorišćeno da se omogući rad sa ulazom i izlazom podataka koji inače funkcioniše preko komplikovanih sistemskih prekida (*eng. interrupt*).

Prevodilac za MikroJava bajtkôd

Drugi alat koji je razvijen u okviru ove teze je *mjc2wsl*, čiji ulaz je kompajlirani bajtkôd iz jezika MikroJava. Ovaj jezik je podskup jezika Java, i primarno je predviđen za učenje konstrukcije kompajlera. U skladu sa tim je i odgovarajući bajtkôd nešto jednostavniji u *MicroJava virtuelnoj mašini* (*MJVM*), nego što je u *Java virtuelnoj mašini* (*JVM*). Ovaj podskup je odabran za prototipiziranje pristupa bajtkôdu, pošto se mogu demonstrirati glavni koncepti, a istovremeno kreirati alat koji u potpunosti pokriva mogućnosti jedne virtuelne mašine. Nakon uspešnih eksperimenata se ovaj pristup polako može proširivati i na celu Javu ili druge virtuelne mašine.

Prevodilac principijelno radi slično kao i prethodni. Program se prevodi naredbu po naredbu, a u generisanom kôdu se definišu strukture koje predstavljaju stanja virtuelne mašine. Glavne strukture su stek za izraze i stek za procedure, koje se simuliraju direktno kao liste. Osim toga postoje strukture za skladištenje globalnih promenljivih, kao i nizova i objekata.

Program u celini je opet predstavljen strukturom *action system*. Za razliku od asemblerskog alata, ovde se akcije definišu za svaku pojedinačnu instrukciju bajtkôda, a imena im se generišu na osnovu adresa u izvršnom fajlu. Ovim se omogućava simuliranje skokova na adrese koje su zapisane u fajlu kod odgovarajućih naredbi. Sistem koji se generiše ovim alatom je *rekurzivan*, u kome se svi pozivi uredno vraćaju, za razliku od onog kod *asm2wsl*. Prednost rekurzivnog sistema

je što se pozivi procedura mogu simulirati bez dodatnog pamćenja adresa poziva, odnosno sam sistem će nastaviti izvršavanje od odgovarajućeg mesta.

Alat *mjc2wsl* podržava sve komande koje postoje u MJVM. Simuliranje struktura je takvo da se podržavaju dokumentovane operacije instrukcija. Samim tim je alat u stanju da prepozna i ispravno prevede bilo kakve programe koji su generisani kompajlerom koji se distribuira uz alat. Osim toga podržava i bilo kakve druge bajtkôd programe koji se pridržavaju instrukcija kako je namenjeno. Naravno, moguće je konstruisati program takav da zloupotrebljava provere opsega pristupa memoriji i u skladu sa tim dovede program u nepoznato stanje zbog nekih apstrakcija koje su uvedene u prevod. Diskutabilno je da li bi takve programe i trebalo podržavati, ali se u budućnosti može uložiti dodatno napora da se simuliraju svi detalji pristupa memoriji.

U okviru alata postoje i prekidači koji menjaju kako se prevode neke instrukcije. One su kasnije korišćene da se dodatno testiraju transformacije na različitim prevodima istih programa. Jedna od promena je da li se za privremene promenljive koriste ista globalno definisana imena, ili se uvek generišu mali lokalni blokovi za promenljive. Postoji i prekidač za različito skladištenje lokalnih promenljivih na steku za procedure: mogu se skladištiti zajedno kao niz, ili svaka odvojeno. Operacije sa svim stekovima se isto mogu prevoditi na dva načina, korišćenjem uobičajenih `POP` i `PUSH` naredbi, ili direktnim operacijama nad listama sa `HEAD` i `TAIL`.

Transformacije

Transformacije se mogu primenjivati na različite načine, ali primarni cilj ovog istraživanja je bila potpuna automatizacija odabira, tako da i korisnici bez iskustva u radu sa sistemom mogu efikasno da primene ovaj proces za svoje potrebe. Sa druge strane zbog fleksibilnosti procesa, iskusni korisnici mogu lako da ga prošire ili da ga integrišu u neki drugi već postojeći proces.

Automatizacija je postignuta *pretraživanjem usponom (eng. hill climbing)*. Algoritam se sastoji od primene transformacija iz prethodno odabranog skupa na program i provere da li je rezultat bolji na osnovu neke *funkcije pogodnosti (eng. fitness function)*. Ako je novi program bolji, uzima se za novu osnovu za primenu daljih transformacija. Inače se odbacuje i nastavlja sa primenom transformacija na trenutni program. Proces se nastavlja dokle god je moguće naći bolji program.

Uspeh procesa uveliko zavisi i od odabira funkcije pogodnosti, pošto ona treba da uspešno navodi prema većoj strukturiranosti programa. Očigledni kandidati su različite metrike programa, kao što je broj naredbi, što u opštem slučaju želimo da umanjimo. U okviru ovog rada za funkciju pogodnosti je korišćena metrika

Structure ugrađena u WSL, koja daje sumu „težina" struktura u programu i time vodi ka boljim rezultatima nego obično prebrojavanje naredbi.

Same transformacije u procesu ne pretpostavljaju ništa u vezi ulaznih programa, odnosno mogu se primenjivati na bilo kakve ulaze i ne zavise od korišćenih prevodilaca. Kvalitet konačnih rezultata će varirati u zavisnosti od tipa ulaza, pošto su primarno odabrane transformacije koje dižu nivo apstrakcije programa. Za neke specifične ulaze ovo može značiti da će promene biti minimalne, ali čak i onda će programi zadržati svoju kompletnu semantiku, odnosno neće dovesti do „kvarenja" programa.

Eksperimenti vršeni na nekoliko tipova ulaznih programa niskog nivoa su pokazali da rezultati mogu biti izuzetni. Transformator može da postepeno od kôda u kome se nalaze uslovni skokovi izvede petlje sa jasnim uslovima za izlazak iz njih. U mnogim situacijama je moguće da se dobije čak i kompaktnija verzija programa. Na Slici 8.9 su prikazani originalni asemblerski kôd rekurzivne procedure koja traži najmanji zajednički sadržalac, kao i dobijeni automatski transformisani WSL. Iako je original rekurzivan, krajnji rezultat se sastoji od jedne `WHILE` petlje i naredbe grananja, pošto rekurzija ovde nije bila neophodna i mogla se uprostiti. Sa druge strane je sistem pokazao dobre rezultate i na rekurzivnim programima. Na Slici 8.19 je prikazan originalni MicroJava kôd za određivanje Fibonačijevog broja i finalna transformisana verzija koja je dobijena od prevedenog kôda niskog nivoa.

```
gcd proc
    ; get params
    pop ax
    pop bx
    cmp ax,bx
    je endequal
    ja greatera
    ; ensure ax is greater
    xchg ax,bx
greatera:
    sub ax,bx
    push bx
    push ax
    call gcd
    pop ax ; result

endequal:
    push ax ; result
    ret
gcd endp
```

→

```
WHILE ax <> bx DO
  IF ax <= bx
    THEN < ax := bx, bx := ax > FI;
  ax := ax - bx OD;
```

Ponovljena Slika 8.9: Rekurzivni NZD, asembler i automatski transformisani WSL

```
program RekFib{
  int fib(int f)
  {
     if (f==0)
       return 0;
     if (f==1)
       return 1;
     return fib(f-2)+fib(f-1);
  }

  void main()
  {
     print(fib(0),3);
     print(fib(2),3);
     print(fib(7),3);
  }
}
```

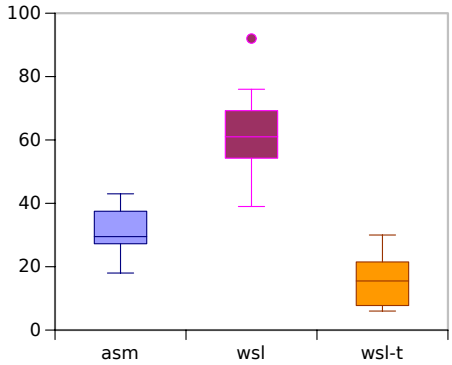
```
BEGIN
  PRINFLUSH(@Format(3, a14(0)));
  PRINFLUSH(@Format(3, a14(2)));
  PRINFLUSH(@Format(3, a14(7)))
WHERE
  FUNCT a14(par1) ==
    VAR <  >:
    SKIP;
    (IF par1 = 1
      THEN 1
      ELSE IF par1 <> 0
        THEN a14(par1 - 2)
             + a14(par1 - 1)
        ELSE 0 FI FI) END
END
```

Ponovljena Slika 8.19: Rekurzivni Fibonači – MicroJava i transformisani WSL kôd

Na primerima koji su prevedeni iz asemblera (skup programa *asm-a*), krajnji automatski transformisani programi su značajno kraći od originalnog kôda (Slika 8.10). Treba napomenuti da su u brojanju naredbi asemblera uglavnom izostavljeni komplikovani delovi koji se bave prekidima za učitavanje i ispis podataka. U proseku su se programi povećavali za 2.09 ± 0.29 puta pri prevođenju u WSL, odnosno za svaku naredbu u asembleru se generiše nešto više od dve naredbe u WSL. Umanjenje pri transformacijama je veće, u proseku 5.15 ± 2.96 puta, odnosno otprilike 5 naredbi se zameni sa jednom. U direktnom poređenju, originalni ulazi u asembleru su 2.41 ± 1.21 puta veći od krajnjih rezultata. Tabela 8.1 prikazuje unapređenja drugih metrika pri automatskim transformacijama prevedenih programa. Jedino pogoršanje koje postoji je kod Mekejbove esencijalne kompleksnosti. Čak i ono je rezultat samo jednog ulaza kod koga automatski proces nije uspeo da nađe transformaciju kojom bi obrisao uvedenu petlju. Naime, da bi se stiglo do tog boljeg rezultata prvo se mora proći kroz jedno povećanje metrike koja je korišćena za funkciju pogodnosti. Nakon primene procesa je bilo moguće ručno primeniti ove transformacije i dovesti kôd do adekvatnih nivoa metrika.

Kod ulaza u jeziku MikroJava postoji originalni izvorni kôd, pa postoji još jedna prilika za poređenje faza procesa i mogućnost direktnog poređenja originalnih programa visokog nivoa sa onima dobijenim automatski od njihovih kompajliranih verzija.

Automatski transformator je primenjen na sve raspoložive tipove prevoda, korišćenjem različitih prekidača u prevodiocu. Različiti prevodi istih programa su trans-



Ponovljena Slika 8.10: Veličine programa u različitim fazama procesa (*asm-a*)

Ponovljena Tabela 8.1: *asm-a* metrike transformacija

Metric	WSL	WSL-t	% diff
McCabe Cyclo	7.40 ± 3	3.60 ± 1	41.60 ± 30
McCabe Essential	1.00 ± 0	1.10 ± 0	-10.00 ± 32
Statements	62.30 ± 15	16.00 ± 9	73.90 ± 14
Expressions	84.20 ± 28	31.50 ± 20	63.80 ± 14
CFDF	88.60 ± 16	22.80 ± 9	73.90 ± 8
Size	327.40 ± 71	95.30 ± 51	70.90 ± 13
Structure	947.70 ± 199	206.90 ± 119	78.20 ± 10

formisani sa različitim procentima unapređenja metrika. Istovremeno je važilo da se do boljih rezultata skoro uvek dolazilo u značajno kraćem vremenu. Ovo je posledica samog algoritma koji primenjuje sve transformacije na sve delove programa. Ukoliko su one ranije uspešne, dalji proces se ubrzava jer ima manje mesta za primene. Na osnovu analiza različitih ulaza se dodatno može unaprediti sam proces, ali i pomoći korisniku da odabere ulaze koji su najadekvatniji za ovaj proces.

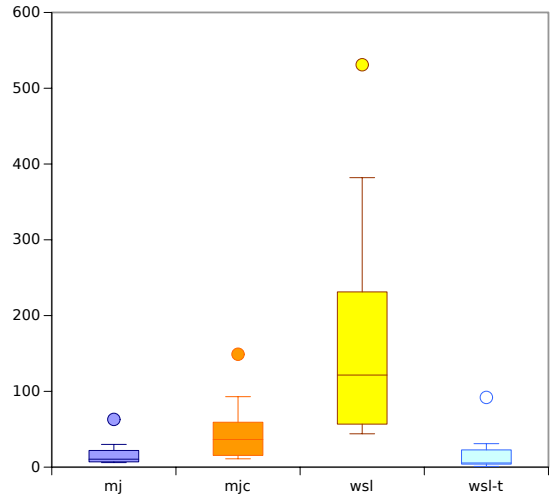
Za dalje analize procesa je korišćena varijanta prevoda koja je dala najbolje rezultate na kraju procesa (`PUSH/POP`, lokalne privremene promenljive, odvojene lokalne promenljive na steku za procedure; skraćeno *pp-lo-sp*). U toku procesa dolazi do očekivanog povećavanja broja naredbi pri kompajliranju i prevođenju u WSL, dok se na kraju ovaj broj značajno smanji automatskim transformacijama (Slika 8.12). Kompajliranje povećava broj naredbi u proseku 2.55 puta, dok prevođenje u WSL to dodatno množi sa 3.78 ± 0.38. Ukupno je to u proseku 9.55 ± 1.62 naredbi u WSL za svaku originalnu naredbu iz MikroJave, međutim ovde imamo i dosta dodatnog kôda koji simulira virtuelnu mašinu. Automatske transformacije ovaj broj umanjuju u proseku čak 16.89 puta, sa proporcionalno visokom standardnom devijacijom od 11.55. Devijacija ukazuje na velike razlike između različitih ulaznih programa, odnosno na sposobnost transformatora da ih popravlja. Pri direktnom poređenju se vidi da su neki programi stvarno dosta umanjeni, dok su drugi nešto veći od originala. Umanjenje je takođe donekle varljivo veliko jer se poredi sa originalnim programima koji su ručno pisani i imaju možda više detalja nego što je neophodno. Takođe, neke strukture se u jeziku WSL mogu pisati kompaktnije nego u MikroJavi.

Za dodatno poređenje razlika originala i krajnjih transformisanih programa su uzeti pojedinačni primeri i razlike predstavljene kao procentualno povećanje ili umanjenje (Slika 8.13). Prosečno poboljšanje kod ovako predstavljenih brojeva je 24.43% ± 36.56. Devijacija opet ukazuje na velike razlike, koje su očigledne i na slici. Sa jedne strane su primeri kod kojih je poboljšanje 60% do 80%, dok su na drugoj nekoliko primera kod kojih je pogoršanje oko 40%.

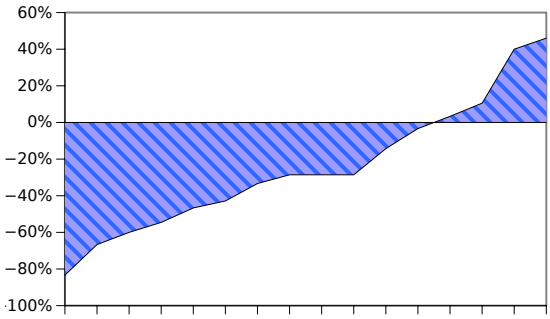
Kod svih testiranih metrika su primećena značajna poboljšanja nakon automatskih transformacija programa (Tabela 8.7). Mekejbova ciklomatska i esencijalna kompleksnost se smanjuju za oko 60%. Ostale metrike se umanjuju za 87% do 94%.

Zaključak i budući rad

U okviru disertacije je predstavljen proces za automatsko restrukturiranje kôda iz niskog u visoki nivo. Ovakav postupak se može primeniti u različitim scenarijima reinženjerstva kôda. Sa jedne strane može značajno olakšati razumevanje programa



Ponovljena Slika 8.12: Broj naredbi u programima u različitim fazama procesa (*alpha-mj*)



Vrednosti su po ulaznim programima, sortirane; niže vrednosti su bolje

Ponovljena Slika 8.13: Procentualna razlika u broju naredbi između originalnih MicroJava programa i transformisanog WSL-a

Ponovljena Tabela 8.7: promene metrika pri transformacijama za *alpha-wsl-pp-lo-sp*

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

i time skratiti potrebno vreme razvoja. U mnogim primenama je upravo restrukturiranje glavni cilj što se ovde može postići automatski – ponekad će rezultat već biti maksimalno pojednostavljen program, a u drugim situacijama će biti barem delimično bolje strukturiran.

Automatski proces transformisanja se pokazao efikasnim na nekoliko različitih tipova ulaznih programa niskog nivoa, te se može očekivati da će se moći uspešno primenjivati i na nove tipove ulaza.

Glavni koraci procesa su donekle slični sa onima koji su već predstavljani u ranijim radovima migracije kôda korišćenjem sistema FermaT [Ward, 2013; Ward, Zedan, and Hardcastle, 2004]. Koriste se prevodioci u WSL, taj kôd se restrukturira transformacijama, a potom se prevodi u održivi C ili COBOL. Proces je bio uspešno primenjivan i na industrijske stare biblioteke Međutim kod ovih pristupa su alati uglavnom prilagođavani konkretnim problemima. Neki delovi programa, naročito direktan rad sa memorijskim strukturama, bi bili odvojeno skladišteni u tabelama koje se ponovo koriste tek na kraju procesa, odnosno praktično su sakriveni od WSL implementacije. U okviru ove teze alati proizvode programe koji u potpunosti predstavljaju sve aspekte originalnog kôda. Jedna od prednosti ovakvog pristupa je što se oni onda mogu i izvršavati u bilo kojoj od faza procesa. Značajnije, ne postoji međuzavisnost ulaznog i izlaznog prevodioca od dodatnih fajlova i tabela. Mana pristupa je što se umanjuje skup mogućih ulaznih programa, pošto je teže predstaviti sve ove detalje. Ovo je naročito izraženo kod *asm2wsl* alata zbog velikih varijacija u mogućnostima asemblera. Sa druge strane, *mjc2wsl* je u stanju da predstavi skoro sve moguće validne programe, najviše zbog toga što MikroJava virtuelna mašina ima mnogo strože definisano ponašanje. Samim tim ova mana skoro da nije ni izražena kod ovog drugog alata.

Poređenje sa drugim sistemima je teže, uglavnom zbog različitih specifičnosti sistema FermaT. Na primer, sistem GenProg [Le Goues, Nguyen, Forrest, and Weimer, 2012], koristi funkciju podobnosti za automatizaciju procesa transformacije programa. Međutim već sam cilj je drugačiji, pošto se koristi za popravljanje grešaka u programu na osnovu testova. Sama funkcija podobnosti je bazirana upravo na tim testovima i njihovoj uspešnosti, a ne na statičkom kvalitetu samog programa. Za promene programa se koriste mutacioni operatori, koji ne očuvavaju semantiku programa, kao što je slučaj kod transformacija u ovom procesu. Sa druge strane, očuvanje semantike i nije pogodno za popravljanje programa u kome znamo da postoje greške.

Algoritam pretraživanja uspinjanjem (*eng. hill climbing*) dosada nije bio uspešno primenjivan na ovaj tip problema, bar koliko je autoru poznato. Fatiregun i saradnici su poredili nekoliko pristupa za rešavanja sličnog problema automatskog restrukturiranja, ali sa dosta različitim početnim tačkama i putem do rešenja [Fatiregun, 2006]. U tim radovima se sekvenca transformacija koristi kao *instanca* sa kojom se radi i na koju se primenjuju mutacijski operatori. Njihovi rezultati su pokazali da je genetsko programiranje bolje nego pretraživanje uspinjanjem i nasumično pretraživanje [Fatiregun, Harman, and Hierons, 2004]. Treba napomenuti da je ipak i samo predstavljanje problema u ovim radovima mnogo više u skladu sa normama za genetsko pretraživanje. I drugi radovi su pominjali pretraživanje uspinjanjem kao moguću opciju za popravljanje grešaka u programima. Na primer u [Arcuri and Yao, 2008], autori koriste genetsko programiranje za popravljanje programa i ukratko diskutuju mogućnost pretraživanja uspinjanjem, ali izražavaju skepticizam u vezi mogućih problema sa lokalnim optimumima.

Postoji dosta mogućih daljih smerova za razvoj napravljenih alata i samog procesa. Kao prvo je moguće razvijati nove prevodioce u WSL jezik, i time omogućiti da se programi iz novih jezika restrukturiraju. Jedan od primarnih ciljeva za budućnost je razvoj prevodioca iz Java bajtkôda, no i mnogi drugi mogu biti zanimljivi. Sa druge strane postoji potreba za otvorenim i dostupnim prevodiocima iz WSL-a u druge jezike. Prevodioci koji su korišćeni za pretvaranje asemblera u C i COBOL nisu javno dostupni.

Sam sistem FermaT se takođe može dalje unapređivati i proširivati. U okviru ove teze su već dodate neke nove transformacije, a neke postojeće su proširivane. Neke od uočenih razlika u različitim tipovima ulaznih programa bi se mogle negirati uvođenjem novih transformacija koje ih prepoznaju.

Proces automatske transformacije se isto može unapređivati. Kao prvo treba dodatno analizirati različite funkcije podobnosti i njihov uticaj na proces. Redosled odabira transformacija se isto može unaprediti na osnovu dosada prikupljenih po-

dataka. Sa jedne strane se možda može doći do još kvalitetnijih rezultata drugim redosledom transformacija, a sa druge se možda može postići veća vremenska efikasnost ukoliko se neke „korisnije" transformacije primenjuju ranije od drugih. Jedan od problema algoritma pretraživanja uspinjanjem je što ima tendenciju da nalazi lokalne ekstreme. Ovo se tipično rešava tako što se proces pokrene više puta sa različitim početnim tačkama i odabere se najbolji od svih rezultata. U ovom slučaju bi se to moglo postići mešanjem redosleda transformacija.

Trenutna implementacija isprobava transformacije dok ne nađe neki napredak. Alternativna implementacija bi mogla da isproba sve moguće transformacije na trenutnom programu i onda od dobijenih rezultata odabere najbolji za sledeći korak algoritma. Kod takve implementacije postoji i prilika za paralelizaciju testiranja transformacija.

Dalja unapređenja bi se mogla postići primenom detaljnijih metrika kroz ceo proces. Trenutno se koristi veći broj metrika samo za programe koji su pisani u WSL jeziku, dok se za originalne verzije programa može naći manji broj metrika. Istovremeno postoji i problem različitih implementacija i tih postojećih metrika. Problem bi se mogao rešiti nekim sistemom koji izračunava univerzalne metrike za različite jezike, kao što je SSQSA (*Set of Software Quality Static Analyzers*) [Rakić, 2015].

Trenutni proces se oslanja na algoritam pretraživanja uspinjanjem. Moguće bi bilo implementirati i druge strategije za pretrage. Jedan od takvih je *tabu* algoritam, koji dozvoljava i da se prave koraci ka gorim rešenjima ako ne postoje bolja, čime se izbegava problem lokalnih minimuma [Glover, 1989]. Takođe bi bilo moguće kreirati varijacije evolucionih algoritama u kojima bi se transformacije koristile kao mutacioni operatori, a metrike bi se i dalje mogle koristiti kao funkcije podobnosti.

Sa druge strane bi bilo moguće implementirati sve ove algoritme i pristupe koristeći i druge sisteme za transformisanje programa, a ne samo FermaT. Bilo koji sistem koji ne garantuje očuvanje semantike programa bi zahtevao da se dodatno testira da li generisani programi i dalje rade isto kao originalni. Tipično se ovako nešto postiže kolekcijama očekivanih ulaza i izlaza.

Short Biography

Doni Pracner was born on 4.12.1984 in Novi Sad.

He finished "Miroslav Antić" elementary school in Futog in 1999, followed by high school "Jovan Jovanović Zmaj", Novi Sad, natural sciences orientation, in 2003. The same year he enrolled into the Faculty of Sciences in Novi Sad to study computer science. He obtained his diploma in July of 2007, and a masters degree in November of 2010.

He has been involved in teaching at the Department of Mathematics and Informatics, Faculty of Sciences in Novi Sad, first as a research associate and later as a teaching and research assistant since 2007. He has conducted theoretical and practical exercises in several computer science courses for undergraduate and master students, including Introduction to Programming, Data Structures and Algorithms 1, 2 and 3; Operating Systems 2; Software Testing and Validation.

He coauthored more than ten papers as conference proceedings and journal articles. He was an organising committee member for several international computer science conferences and workshops. He was a researcher on several national, international and bilateral projects.

During his undergraduate studies he won an exceptional award for a student paper in the area of data mining. Based on other papers in this area, he received the "Aleksandar Saša Popović" faculty award for research work in 2008.

Novi Sad, 2018 Doni Pracner

Kratka biografija

Doni (Aleksandar) Pracner je rođen 4.12.1984. godine u Novom Sadu.

Završio je osnovnu školu „Miroslav Antić" u Futogu, 1999. godine, a nakon toga gimnaziju „Jovan Jovanović Zmaj" u Novom Sadu, prirodno-matematički smer, 2003. godine. Iste godine upisuje Prirodno-matematički Fakultet u Novom Sadu, smer diplomirani informatičar. Redovne, diplomske, studije završava u julu 2007. godine, a master studije u novembru 2010.

Od 2007 je kao istraživač-pripravnik, kasnije asistent i saradnik u nastavi, na Departmanu za matematiku i informatiku Prirodno-matematičkog Fakulteta u Novom Sadu uključen u održavanje nastave iz više predmeta, uključujući „Uvod u programiranje", „Strukture podataka i algoritmi 1, 2 i 3", „Operativni sistemi 2" i „Testiranje softvera".

Koautor je više od deset radova objavljenih na međunarodnim konferencijama i u naučnim časopisima. Bio je član organizacionog odbora nekoliko međunarnodnih konferencija i radionica. Učesnik je više aktuelnih i završenih naučnih projekta državnih ministarstava, kao i međunarodnih projekata.

Za vreme studiranja je osvojio izuzetnu nagradu univerziteta za temat iz oblasti „Data Mining". Na osnovu radova iz iste oblasti mu je 2008. godine dodeljena fakultetska nagrada „Aleksandar Saša Popović" za istraživački rad na polju računarskih nauka.

Novi Sad, 2018.

Doni Pracner

.....

Univerzitet u Novom Sadu
Prirodno-matematički fakultet
Ključna dokumentacijska informacija

Redni broj: RBR	
Identifikacioni broj: IBR	
Tip dokumentacije: TD	Monografska dokumentacija
Tip zapisa: TZ	Tekstualni štampani materijal
Vrsta rada: VR	Doktorska disertacija
Autor: AU	Doni Pracner
Mentor: MN	dr Zoran Budimac
Naslov rada: NR	Prevođenje i transformisanje programa niskog nivoa
Jezik publikacije: JP	engleski
Jezik izvoda: JI	srpski/engleski
Zemlja publikovanja: ZP	Srbija
Uže geografsko područje: UGP	Vojvodina

Godina: 2018
GO

Izdavač: autorski reprint
IZ
Mesto i adresa: Novi Sad, Trg D. Obradovića 4
MA

Fizički opis rada: 9/211 (xx + 191)/89/23/36/0/4
(broj poglavlja/strana/lit. citata/tabela/slika/grafika/priloga)
FO
Naučna oblast: Računarske nauke
NO
Naučna disciplina: Evolucija softvera
ND
Predmetna odrednica
/Ključne reči: Evolucija softvera, transformacije kôda, automatizacija transformacija, formalne transformacije, asembler, bajtkôd, Wide Spectrum Language, MicroJava
PO
UDK
Čuva se:
ČU
Važna napomena:
VN

Izvod: U okviru ove teze se predstavlja pristup radu sa programima niskog nivoa koji omogućava automatsko restrukturiranje i podizanje na više nivoe. Samim tim postaje mnogo lakše razumeti logiku programa što smanjuje vreme razvoja.

Proces je dizajniran tako da bude fleksibilan i sastoji se od više nezavisnih alata. Samim tim je lako menjati proces po potrebi, ali i upotrebiti razvijene alate u drugim procesima. Tipično se mogu razlikovati dva glavna koraka. Prvi je prevođenje u jezik WSL, za koji postoji veliki broj transformacija programa koje očuvavaju semantiku. Drugi su transformacije u samom WSL-u. Za potrebe prevođenja su razvijena dva alata, jedan koji radi sa podskupom x86 asemblera i drugi koji radi sa MikroJava bajtkôdom. Rezultat prevođenja je program niskog nivoa u WSL jeziku.

Primarni cilj ovog istraživanja je bila potpuna automatizacija odabira transformacija, tako da i korisnici bez iskustva u radu sa sistemom mogu efikasno da primene ovaj proces za svoje potrebe. Sa druge strane zbog fleksibilnosti procesa, iskusni korisnici mogu lako da ga prošire ili da ga integrišu u neki drugi već postojeći proces. Automatizacija je postignuta *pretraživanjem usponom (eng. hill climbing)*.
Eksperimenti vršeni na nekoliko tipova ulaznih programa niskog nivoa su pokazali da rezultati mogu biti izuzetni. Za funkciju pogodnosti je korišćena ugrađena metrika koja daje "težinu" struktura u programu. Kod ulaza za koje je originalni izvorni kôd bio dostupan, krajnje metrike najboljih varijanti prevedenih i transformisanih programa su bile na sličnom nivou. Neki primeri su bolji od originala, dok su drugi bili nešto kompleksniji. Rezultati su uvek pokazivali značajna unapređenja u odnosu na originalni kôd niskog nivoa.

IZ
Datum prihvatanja teme od strane
Senata: 14.09.2017.
DP
Datum odbrane:
DO
Članovi komisije:
(Naučni stepen/ime i prezime/zvanje/fakultet)
KO
Predsednik: dr Miloš Radovanović, vanredni profesor,
Univerzitet u Novom Sadu, Prirodno-matematički fakultet

Mentor: dr Zoran Budimac, redovni profesor,
Univerzitet u Novom Sadu, Prirodno-matematički fakultet

Član: dr Gordana Rakić, docent,
Univerzitet u Novom Sadu, Prirodno-matematički fakultet

Član: dr Zaharije Radivojević, vanredni profesor,
Univerzitet u Beogradu, Elektrotehnički fakultet

University of Novi Sad
Faculty of Science
Key Words Documentation

Accession number: NO	
Identification number: INO	
Document type: DT	Monograph documentation
Type of record: TR	Textual printed material
Contents code: CC	Doctoral dissertation
Author: AU	Doni Pracner
Mentor: MN	Dr. Zoran Budimac
Title: TI	Translation and Transformation of Low Level Programs
Language of text: LT	English
Language of abstract LA	Serbian/English
Country of publication: CP	Serbia
Locality of publication: LP	Vojvodina

Publication year: 2018
PY

Publisher: Author's reprint
PU
Publ. place: Novi Sad, Trg D. Obradovića 4
PP

Physical description: 9/211 (xx + 191)/89/23/36/0/4
(no. chapters/pages/bib. refs/tables/figures/graphs/appendices)
PO
Scientific field: Computer Science
SF
Scientific discipline: Software Evolution
SD
Subject/Key words: Software Evolution, Code transformations, Automated Transformations, Formal transformations, assembly, bytecode, Wide Spectrum Language, MicroJava
SKW
UC
Holding data:
HD
Note:
N

Abstract: This thesis presents an approach for working with low level source code that enables automatic restructuring and raising the abstraction level of the programs. This makes it easier to understand the logic of the program, which in turn reduces the development time. The process in this thesis was designed to be flexible and consists of several independent tools. This makes the process easy to adapt as needed, while at the same time the developed tools can be used for other processes. There are usually two basic steps. First is the translation to WSL language, which has a great number of semantic preserving program transformations. The second step are the transformations of the translated WSL. Two tools were developed for translation: one that works with a subset of x86 assembly, and another that works with MicroJava bytecode. The result of the translation is a low level program in WSL.

The primary goal of this thesis was to fully automate the selection of the transformations. This enables users with no domain knowledge to efficiently use this process as needed. At the same time, the flexibility of the process enables experienced users to adapt it as needed or integrate it into other processes. The automation was achieved with a *hill climbing* algorithm.
Experiments that were run on several types of input programs showed that the results can be excellent. The fitness function used was a built-in metric that gives the "weight" of structures in a program. On input samples that had original high level source codes, the end result metrics of the translated and transformed programs were comparable. On some samples the result was even better than the originals, on some others they were somewhat more complex. When comparing with low level original source code, the end results was always significantly improved.

AB
Accepted on Senate: 14.09.2017
AS
Defended:
DE
Thesis Defend Board:
(Degree/first and last name/title/faculty)
DB

President:	Dr. Miloš Radovanović, associate professor, University of Novi Sad, Faculty of Sciences
Mentor:	Dr. Zoran Budimac, full professor, University of Novi Sad, Faculty of Sciences
Member:	Dr. Gordana Rakić, assistant professor, University of Novi Sad, Faculty of Sciences
Member:	Dr. Zaharije Radivojević, associate professor, University of Belgrade, School of Electrical Engineering