Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Migrating IBM HLASM to C

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Jakob Wilhelm BSc
Registration Number 0226237

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, 9th September, 2015

Jakob Wilhelm   Andreas Krall

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Migration von IBM HLASM nach C

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Jakob Wilhelm BSc
Matrikelnummer 0226237

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 9. September 2015

Jakob Wilhelm	Andreas Krall

Erklärung zur Verfassung der Arbeit

Jakob Wilhelm BSc
Neulengbach

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. September 2015

Jakob Wilhelm

Danksagung

Zuerst möchte ich meinen aufrichtigen Dank an meinen Betreuer, Prof. Andreas Krall, richten, der mich fortlaufend während des gesamten Projekts unterstützt hat. Seine Ratschläge haben mir geholfen, neue Einblicke und Erkenntnisse zu gewinnen und mit großer Zuversicht am Fortschritt zu arbeiten.

Ich möchte meinem Chef, Mag. Rainer Frömmel, für die Chance danken, Forschungsarbeit im Bereich von Reverse Engineering und Übersetzerbau durchzuführen, auch wenn die Einträglichkeit der Ergebnisse zu Beginn oft ungewiss ist.

Ich möchte auch meinen Dank an die Open-Source Community zum Ausdruck bringen, ganz besonders die unzähligen Freiwilligen, die ihre wertvolle Zeit in die Entwicklung von freier Software stecken.

Nicht zuletzt möchte ich meiner Familie danken, die mich die letzten Jahre während meines Studiums großartig unterstützt hat.

Acknowledgements

First, I would like to express my sincere gratitude to my advisor, Prof. Andreas Krall, who has supported me continuously throughout the whole project. His guidance helped me to gain new insights and confidently pursue progresses.

I would like to thank my boss, Mag. Rainer Frömmel, for the chance of doing research in the areas of reverse and compiler engineering, even if the usefulness of the outcome is often uncertain in the beginning.

I would also like to express my gratitude to the open source community, especially to numberless volunteers who put their valuable time into engineering free software.

Last but not least, I would like to thank my family who has supported me within the last few years during my studies.

Kurzfassung

Großrechner sind weiterhin im Betrieb, um das tägliche Geschäft von Organisationen mit Hilfe von bestehenden Softwarelösungen zu bewältigen. Sobald sie allerdings abgelöst werden sollen, muss eine Möglichkeit gefunden werden, die bestehenden Geschäftsanwendungen auf eine neue Plattform zu migrieren. Eine Neuentwicklung erlaubt den Einsatz von modernen Programmiersprachen, Frameworks und Techniken, ist allerdings in der Praxis oftmals nur schwer durchführbar bedingt durch das mögliche Fehlen von Dokumentation der Programme und Spezifikationen der Anforderungen sowie die enormen notwendigen Ressourcen zum Entwickeln und Testen der Anwendungen.

Diese Arbeit beschäftigt sich mit der Migration von Geschäftsanwendungen, die für IBM Großrechner in Assembler geschrieben wurden (HLASM), durch automatisierte Übersetzung in portablen C/C++ Code. Es wird gezeigt, wie verschiedene Compiler- und Analysetechniken eingesetzt werden können, um aus linearem Assemblercode eine strukturierte Darstellung der Programmlogik zu erreichen. Dadurch ist es nicht nur möglich, den Großrechner zu verlassen, sondern auch die Lesbarkeit der Programme und daraus folgend die Wartbarkeit eben dieser zu verbessern.

Die beschriebene Implementierung wurde mit Hilfe von produktiven Assembler-Programmen von einer größeren Organisation evaluiert.

Abstract

Mainframe computers are still running legacy applications to handle day-to-day business for organizations. At the time when the mainframe should be replaced, it has to be discussed how business applications could be transferred to the new platform. Redeveloping from scratch allows the usage of modern languages, frameworks and techniques but is difficult due to the possible lack of documentation and requirement specifications and the vast efforts required for developing, testing and integrating the application.

This work will discuss the migration of business applications written in assembler for IBM mainframes (HLASM) using automatic translation to portable C/C++ code. We will show the benefit of different compiler and analysis techniques transforming the linear assembler code to a structured representation that will not only help us leaving the mainframe but also increasing readability and thus maintainability of the applications.

The implementation has been evaluated using a set of legacy assembler programs used in production from a larger organization.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

List of Figures **xvi**

List of Tables **xvii**

List of Algorithms **xix**

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Problem Definition . . . 2
- 1.3 Expected Outcome . . . 3
- 1.4 Outline . . . 4

2 Foundations **5**
- 2.1 Compiler Techniques of Avail . . . 5
- 2.2 Previous Work . . . 6

3 Implementation **13**
- 3.1 Preview . . . 13
- 3.2 Intermediate Representation . . . 13
- 3.3 Parser . . . 16
- 3.4 Analyses . . . 17
- 3.5 Code generation . . . 30

4 Evaluation **37**
- 4.1 Current State of Implementation . . . 37
- 4.2 Rating of Implementation State . . . 40

5 Conclusion and Outlook **43**
- 5.1 Future Work . . . 43

A Sample Program Conversions **51**
A.1 Arithmetic Operations 51
A.2 Restructuring 54

Bibliography **59**

List of Figures

1.1 Sample usage of struct definitions 2

3.1 Compiler architecture - an overview 14
3.2 Class hierarchy of instruction types 15
3.3 first code sample 16
3.4 Parser processing a sample instruction 17
3.5 Branch with two different branch targets 18
3.6 Sample usage of condition codes 23
3.7 Sample generated code hiding condition code fields 23
3.8 Sample code with basic blocks 25
3.9 sample usage of runtime library 31
3.10 Sample field definitions in HLASM 32
3.11 Sample field definitions in C++ 32
3.12 Conditional code in HLASM 33
3.13 Conditional code in C 34
3.14 Sample code to be structured in HLASM 35
3.15 Structured code in C 36

4.1 HLASM code as comments 39

5.1 Limitations of control flow analysis 45
5.2 Sample: self-modifying code 48
5.3 Sample HLASM macro code 49

A.1 Sample arithmetic operations: ASM code 51
A.2 Sample arithmetic operations: generated header file 52
A.3 Sample arithmetic operations: generated code file 52
A.4 Restructuring: ASM code 54
A.5 Restructuring: generated header file 55

A.6 Restructuring: generated code file 56

List of Tables

4.1 Status of program conversion 38
4.2 Branch instructions in programs 39

List of Algorithms

1 Control- and Dataflow Analysis using Value-Sets 20
2 Control- and Dataflow Analysis using Value-Sets (cntd.) 21
3 Detecting potential entries of separate functions 26
4 Checking separate function validity 27
5 Restructuring of Intermediate Representation 28
5 Restructuring of Intermediate Representation (ctnd.) 29
6 Replacing gotos 47

CHAPTER 1

Introduction

1.1 Background

1.1.1 IBM System/390

IBM's Enterprise System Architecture/390 (ESA/390)[IBM03] has been introduced in 1997 with the the announcement of the S/390 G4 system which has been updated by S/390 G5 less than a year later[SAIC$^+$99]. S/390 is a CISC architecture based on and still compatible with S/360 introduced in 1964.

Application programers can make use of instructions including arithmetic operations, tests and comparisons, different branches, string operations and which is especially notable compared to architectures wide-spread nowadays - decimal operations.

Instructions are 2, 4 or 6 bytes long depending on the operand types the instruction requires. Operands are mostly registers or memory addresses. For the latter, indirect addressing in the form D(X,B)[1] is applied. Index and base require four bits each (there are 16 registers that have to be addressable) and displacement consumes 12 bits, resulting in a maximum offset of 4095 bytes. The index field is not available for all instructions taking address parameters, either because the field is not available at all or it is interpreted as length (e.g. for string operations).

Some instructions also require immediate values as operands, which are passed as plain 8-bit immediate number or in indirect addressing format where only few bits are relevant and the rest is ignored. E.g. *SLA* (Shift Left Single) takes the number of bits to shift as immediate parameter. SLA has the following memory layout (in half-bytes): 8BRXBDDD which means that the instruction is identified by 8B, R defines the register which value should be shifted, X/B define index and base of the address parameter

[1] *Displacement*, *Index* and *Base* are three addends resulting in a memory address. While displacement is an immediate value assembled to the application binary, both index and base are read from a register whose number gets assembled to the binary.

(both will just be ignored) and DDD is the displacement factor. Only bits 26-31 of the displacement factor will be considered as immediate value, the others will be dismissed.

In general, using immediate values is restricted to small values (8 bit). To use larger values, they have to be assembled separately and referenced by their address.

1.1.2 IBM High Level Assembler for z/OS, z/VM and z/VSE

HLASM stands for *High Level Assembler* which is available for z/OS, z/VM and z/VSE. Beside encoding instructions of the architecture, HLASM supports definitions of constants, typed structs and most notably macros. Structure definitions are a list of fields with incrementing offsets relative to a section the structure belongs to. It is then possible to link a register with that structure which results in the assembler translating subsequent field accesses to memory addresses using the register as base and the relative offset of the field as displacement. This is accomplished by the *USING* directive which is passed the base register and the section name to be linked. Figure 1.1 shows a sample usage of structures. In this example, both last and first name consume 20 characters and the year

```
PERSON    DSECT
LSTNAME   DS      CL20
FSTNAME   DS      CL20
BORN      DS      PL2
```

Figure 1.1: Sample usage of struct definitions

of birth is stored in packed (decimal) format consuming two bytes.

Macros allow definitions of code blocks to be inserted at different positions in the program including textual replacements adapting the code for specific applications. As they are replaced completely before the resulting assembler code is processed, they form an independent meta-language. Macros are outside the scope of this thesis and therefore will not be discussed further.

1.2 Problem Definition

This work concentrates on applications written in the assembler language implementing business processes. Concerned code has evolved over the last decades and would not be written in assembler nowadays. But although it is technically superseded, it still contains lots of internal knowledge that is very valuable to an organization. Writing those applications from scratch would require enormous efforts to define specifications, implement them and assure the quality of the new hand-written code to prohibit even minimal deviations from the existing implementation. Therefore, one is seeking for automated transformation processes that transfer a long-term grown infrastructure to a newer, sustainable platform fit for the future.

There are a number of solutions for replatforming such software, e.g.:

- rewriting the code manually for the new platform
- keeping the code and use an interpreter, simulator or emulator
- translating the code automatically

As already mentioned, the first solution might not always be feasible. Additionally, the transformation process should be completed as fast as possible as it will interfere with day-to-day business. While the code is rewritten from scratch, there will be lots of changes to the legacy code, too. This makes it hard to keep track of all the modifications and keep a consistent view.

The last two choices have different strengths. While the former facilitates to reproduce all characteristics of the legacy system, the latter gives the chance to transform the system to the new platform and leave behind all legacies, resulting in a more modern environment which most probably will be strived for.

In the domain of emulation, there are a few solutions targeting S/390 like Hercules[her] (which is specific for S/390) or qemu[Bel05] (which is a multi-guest-platform emulator supporting S/390). In the domain of translation, there are no well-established solutions available we know of. Translation is rather proposed as consulting service than provided as (commercially) available software.

It is required that the new system is sustainable which implies maintainability. This requirement is best solved with code that is state-of-the-art of software development, thus, (as far as possible) sophisticated generated code is favored over keeping legacy assembler sources. Therefore, an (at least mostly) automated transformation for legacy IBM HLASM code has to be implemented.

1.3 Expected Outcome

The goal is to implement a source-to-source compiler that is able to translate all given HLASM code to the desired target language on the future platform. The desired target language is C, a compromise between near-to-the-system programming typical of assembler and higher-level programming facilitating maintainability, which is available on a huge range of different platforms among which x86_64 has been chosen.

Realistically, we are expecting some restrictions concerning the generated code and therefore lower our expectations. The actual aim is to automatically translate almost the entire amount of sources and only skip few problematic cases. These cases mostly concern dynamic code, i.e. dynamically computed branches or self-modifying code.

The implementation is still valuable in case automatic translation is not successful for all input sources provided that detailed information about the complications is given in the case of abnormal situations. The translator should not only be able to translate the sources but also to give detailed information about how much of the source code could be translated and which kind of problems have to be expected within the rest of the input sources.

Summarized, the goal is to implement an automatic translation process from HLASM programs to C and to evaluate its benefit in practice. The proposed work should help making the decision on how to handle legacy assembler sources and estimate the inevitable amount of manual intervention needed. The evaluation should also include information about how the implementation could be improved and how further work on the project could bring forward the transformation process for future utilization.

1.4 Outline

Chapter 2 will present previous work, both specific for migrating assembler code for IBM S/390 and general techniques approved for source-to-source translation and decompilers (or reverse engineering in general).

The implementation of the actual work will be discussed in detail in the following chapter 3.

Subsequently, chapter 4 will evaluate the outcome of the work. It will be discussed whether we benefit from the implemented code generation yet and whether our expectations have been satisfied.

Finally, chapter 5 will conclude this work and point out how development could proceed.

Sample program conversions can be found in appendix A

CHAPTER 2

Foundations

There are two main parts that shall be discussed in this chapter:

- General techniques in the domain of compiler engineering, reverse engineering, interpreters, etc. that could be of avail for implementing the proposed project.
- Previous work that has been done in the domain of translating HLASM code from IBM mainframes and translating (reverse engineer) ASM code in general. This part should focus only on comparable projects.

2.1 Compiler Techniques of Avail

Very basic technologies used in different kinds of compilers (including source-to-source and reverse compilers) are described in detail in the literature, cf. [AP02][LSUA06]. Hence, aspects being relevant for this work will only be mentioned in brief in this section.

First of all, a *parser* is needed as frontend to read the assembler source files and present the program as *abstract syntax tree* (AST) suitable for further processing. For developing the parser, one can make use of a *parser generator* like ANTLR [PQ95]. Depending on the syntax, it might be preferred to write the parser by hand (especially if it is rather simple and context-sensitive) using regular expressions.

Different analyses are used for various purposes in different kinds of compilers. A *liveness analysis* is used for register allocation in compilers to determine the extent of a register's liveness (i.e. to determine when the register is freely available again to be used for a different purpose). But this analysis is also valuable for reverse engineering as it is able to locate dead-stores (stores to fields which values will never be read) or the liveness range can be used to determine the intention of a variable and that information can be used to identify the type of a field or even find meaningful names (e.g. some generated variable name could contain 'ptr' or 'idx' if it is used as pointer to a field or index to some array).

Control and dataflow analyses allow to depicture the program flow (i.e. all possible edges between instructions) including the possible values of fields or registers. This is useful for restructuring the program while keeping the semantics. In many (or even most) cases, an instruction will not assign a constant value known at compile-time but some value that is calculated at runtime and thus might also change between repeating executions of the instruction. Therefore, it is not sufficient for the dataflow analysis to store a single value but it has to keep track of multiple possible values. This is accomplished by storing a value-set [BJSW13][BR04][BR10] or a value-range [BML+13][Pry07].

2.2 Previous Work

2.2.1 Bogart

Feldman and Friedman present *Bogart* [FF99], a tool for automatic translation of assembler programs written for S/390 to C using tools and techniques in the domain of Artificial Intelligence. Bogart has been used to translate a database system and an application generator, both being parts of a real existing, large commercial application.

Bogart succeeds a previously developed brute-force approach implemented for translating the same application and shows many improvements. Both approaches require manual code changes prior to automatic translation necessitated by untranslatable constructs like self-modifying code. The authors claim that manual code preparation has advanced at about 3600 lines of code per person-month and this amount of work has been decreased drastically (no explicit figures available) by switching to the more sophisticated translation provided by Bogart.

Bogart implements different methods for improving the generated code. Gotos are replaced by loops and conditionals whenever possible. Instructions which results are not used are removed from the generated code; redundant code is only generated once when detected. Several instructions can be consolidated to one single C statement when possible, e.g. subsequent arithmetic instructions writing to the same register could be translated as one single assignment having a longer arithmetic expression on the right side.

The performance is mainly discussed compared to the precedent brute-force approach claiming a performance gain of factor 1.74. The performance hit compared to handcrafted C is only 10%. However, compared to the original program the execution time has tripled.

2.2.2 Relogix

MicroAPL Ltd. has developed the *Relogix Assembler to C Translator* [Mic09][Mar10] and made it available in different versions for different architectures, among which one has been designed for translating IBM mainframe assembler, called *Relogix/MF*. Conversion of code is provided as service, thus the translator is only used internally and rather presented as black-box, i.e. features are discussed rather than internals.

Relogix demands manual code rewrites prior to generation for some constructs that cannot be translated automatically „because of fundamental architectural differences between assembler and C“. Most probably, this addresses common problems like self-modifying code and dynamically computed branch targets.

Relogix tries to generate code that is as readable and maintainable as possible. Therefore, apart from replacing gotos, more higher-level reverse engineering techniques are used. Multiple instructions can be merged to one single statement, e.g. combinations of compare/branch instructions consolidated to one single loop or *if*-condition.

The liveness of registers is evaluated to find distinct (temporary) variables, i.e. if a function uses a register at two (or more) different locations and the register is never *live* at each other location, the register has two different purposes and therefore can be translated using two different variables. The usage of a register is analyzed further and heuristics are used to guess the type of variable it represents, e.g. using it in arithmetic operations like multiplications might be a hint it is an integer field whereas using it to access data could lead to the assumption that it represents a pointer variable. Based on these observations, variables of different types are generated to replace the register. Even meaningful variable names can be discovered in many cases, e.g. if a variable is used as loop iterator, it could be called *i*, if a variable is used as pointer to access a field it could be named like that field with a suffix like *_ptr* appended.

2.2.3 FermaT

Martin Ward presents the *FermaT Transformation System*[War99], a code transformation framework for assembler and Cobol programs developed at Durham University and Software Migrations Ltd. It is based on multiple transformation rules that are guaranteed to be correct by formal methods and applied subsequently on the code. Therefore, the original source code's listing files[1] are transfered to the *Wide Spectrum Language* (WSL), an intermediate representation combining low-level and high-level code constructs. Using this language, it is possible to depicture both the input program as well as the (functionally equivalent) transformed program which will be written to C from the WSL representation.

The *Transformation Engine* is the heart of the FermaT workbench. It contains a library of transformation rules that can be applied. A single rule can be implemented to solve specific problems, e.g. restructure a particular code construct. But rules cannot only be written for translating ASM code to another programming language, they can also be used to analyze or modify anything else: as example, the Year 2000-problem is given.

The implementation seems to be fully developed as shown by an evaluation translating 1,925 assembler listing files containing nearly 6 million lines, among them about 3 millions of source lines (dismissing headers, etc.). The author claims that all files could be

[1]The listing files contain additional information that facilitates further processing, like instruction addresses. Therefore, the listing files are favored over plain source files. Indeed, there is no information in listing files that could not be extracted from source files. So the approach in general should not be dependent on listing files but using them is the more convenient way.

translated and the generated code is compilable. Although, the code has to be checked for *FIXME*-comments generated in case the code could contain errors, e.g. when branch addresses or *EX* target instructions cannot be determined statically. Unfortunately, there are no figures available indicating which percentage of programs is affected by such issues, but they rather seem to be the exception.

2.2.4 dcc

Cifuentes presents the decompiler *dcc* [CG95] for the DOS operating system on the Intel 80286 architecture as part of her work on reverse compilation techniques [Cif94]. Although it is not handling IBM HLASM, this compiler shares many techniques with a source-to-source compiler translating assembler code. The project consists of three main parts: a machine-dependent front-end reading the input source code, a machine- and language-independent analyzing module, the *universal decompiling machine* (UDM) and finally a back-end dependent on the target language. Thinking of our proposed project, we would only have to replace the front-end to read assembler source code instead of binary code. As assembler code is very low-level and assembler statements represent single machine instructions, it should be possible to replace the front-end without difficulty. One huge advantage of translating assembler code is the availability of symbolic names which can be used to deduce variable or procedure names. On the other hand, there is nothing missing compared to binary code, thus, everything described for binary analyses is likewise true for analysing assembler code.

The UDM provides the ability to restructure code, i.e. replace goto statements and partition code into multiple procedures. This is accomplished by detailed dissections (mostly in the domain of graph analyzes targeting the control flow graph) described in detail in different works from the author [Cif93][Cif96].

An example shows a program in binary representation (as hexdump) together with the automatically generated C code from the decompiler. As expected, the code is well structured and well readable.

Indeed, it has to be mentioned that dcc has been designed as *decompiler* for programs that have been *compiled* beforehand, i.e. programs that are not written in the assembler language but have been compiled to binary from a higher-level language. Therefore, handling of unstructured code is not the key issue as structured code will compile to binary code that can be structured easily. Throughout the different works of the author, it is mentioned that algorithms and techniques described can be used for all kind of programs, even those not possible to restructure. If restructuring fails, it could still replace at least some of the gotos or find some procedures to extract and stick to goto statements otherwise. Of course, there is again no solution available for dynamic code constructs like self-modifying code or dynamically calculated branch targets.

2.2.5 Mimic

Mimic, titled *fast System/370 simulator* by the author[May87], is a simulator running S/370 applications on IBM's RT PC (*RISC Technology Personal Computer*). It is based on

the assumption that performance can be improved compared to established simulators by grouping instructions and translating them as semantic units, e.g. associated comparison and branch instructions can often be translated together.

If such instructions were not handled together, it would be necessary to store condition codes (which is done differently on S/370 and RT PC and arises new problems) that occupy additional memory. In general, processing only single instructions could lead to the necessity of additional registers or memory allocations. Especially registers are a very limited resource. If there are no more free registers available (which is rather probable), register values have to be moved to main memory which again introduces additional instructions and overhead.

Mimic uses control- and dataflow analyses to retrieve the program flow. Self-modifying code is not allowed. The binary program code is partitioned into multiple *code block*s, more or less a set of instructions allocated consecutively in memory. Code is translated at runtime once a code block is executed for the first time. The generated code is then stored for the case it is executed again.

It has been shown that this approach significantly decreases the amount of host instructions needed to implement a given program. Thus, it might be worthwhile to analyze a program not only instruction-by-instruction but from a more distant view to respect the context.

2.2.6 BAP: A Binary Analysis Platform

BAP[BJSW13], abbreviated for *Binary Analysis Platform* is a publicly available infrastructure for program analysis and verification operating on binary executable code. It has been developed by Carnegie Mellon University in Pittsburgh, PA, USA and is available for x86 and ARM, although only a subset of each instruction set is supported.

BAP is built up by three main parts: a frontend reading the binary program code, an intermediate representation and a backend for implementing analyses and verifications.

Side effects are preserved by explicit statements, e.g. condition codes set by instructions are explicitly assigned to a special variable. The code is later transformed to SSA form and code optimizations are performed: dead stores are eliminated.

BAP also provides the ability to present a program in different forms, e.g. as control flow graph.

As it is publicly available in source form, it is a valuable resource to gain an impression on the whole process of reverse engineering a program in compiled binary form.

2.2.7 Dynamic Liveness Analysis

Probst et al.[PKS02] researched on liveness analysis for dynamic translators. The analysis is used to eliminate dead stores, i.e. stores to registers which values will never be read because the register will be overwritten before it is read again.

The optimal solution to this problem is to find all occurrences of stores that are not necessary. Having said that, any other solution that reduces the amount of such stores is an improvement to the translated program as well. Finding the optimal solution is very

expensive, thus it might be more efficient to eliminate only some of the dead stores and keep those hard to detect. At some point, gaining speed by not executing dead stores will not pay off time needed for detailed analysis.

The solution presented is to use dynamic liveness analysis which is improved by every run of a code block. The results are stored on program termination and might get reused and even improved on future runs of the program. It has been shown that the results of the cheaper dynamic liveness analysis get very close to the optimal solution after only few program runs.

2.2.8 Compiled Simulation

Brandner et al.[BHK13] discuss different approaches for instruction set simulators. While they focus on digital signal processors (DSPs), their work is just as well valueable for all kinds of instruction set simulators. The comparison covers simple to optimized interpreters, compiled simulation and simulations using dedicated hardware (both FPGA and ASIC) for peripheral devices as well as the simulated processor itself.

Most interesting in the scope of this thesis is the topic *compiled simulation*. There are two main ways to go: static and dynamic compiled simulation. While the first requires the program to be translated before it is executed, the latter compiles the program on-the-fly when it is about being ran. Compiled simulation is said to generate a simulator of a target architecture for one specific application program. This approach is very close to reverse engineering and source-to-source translation from assembler to higher-level language code as the application-specific simulator is generated as higher-level code (e.g. C or C++), too. Optimizations are then applied by an established compiler that translates the generated C or C++ to binary code.

Comparing static and dynamic compiled simulation, some possible problems with static code translations are addressed, mainly in the domain of dynamic branch targets and self-modifying code. These issues can be solved using the dynamic approach but pursueing the static approach, there is no other solution than switching back to interpretation or some dynamic translation model.

Choices for various aspects of the design of code generation are discussed, e.g. how to handle status registers and flags or how to represent program flow. One option for the latter is to generate one single function with a large select-statement in a loop and use select's fall-through to execute instructions sequentially. If a branch should occur, one sets the address of the branch target to the field tested by select and break out of the select statement. As enclosed by a loop, select will just be executed again and continue program flow at the desired branch target address.

Farfeleder et al.[FKH07] have implemented a compiled emulator for inorder pipelined architectures. They focus on VLIW processors for digital signal processing but again, many aspects of their work are likewise valid for general-purpose processors or processors in general.

Each basic block of the program is generated as separate function in C, returning the index of the next basic block to be executed. Basic blocks are organized in an array storing their addresses. Performance metrics presented by the authors show that programs built

of larger basic blocks are simulated more efficiently. This implies that handling branches or control flow in general is rather expensive and summarizing instructions to blocks rather than processing them one-by-one leads to significant speed-up.

Again, branch targets not statically determinable as well as self-modifying code are an unpassable problem for compiled emulation. In such cases, it is necessary to switch back to interpretation which is much slower but can handle all dynamic situations. As soon as possible, in the best case at the end of the current basic block, otherwise at the end of some following basic block, execution switches back to compiled emulation.

CHAPTER 3

Implementation

3.1 Preview

The code translation consists of three main stages, executed consecutively:

- parser
- analyses and restructuring
- code generation

It is assumed that the code provided as input is already precompiled, i.e. there is no macro code left as a macro processor is not part of this project.

The input assembler code has to be parsed first to be available in memory in an intermediate representation suitable for further processing.

Program semantics are then examined in the analysis step to determine control and data flow. After that, instructions are reordered to achieve clean program structures (i.e. loops and conditionals will replace *goto*s and labels where possible).

Finally, appropriate statements in the C programming language have to be found to implement the semantics described by the intermediate representation. This code will be inherently structured as restructuring has already taken place in the preceding step.

The composition is shown in figure 3.1.

3.2 Intermediate Representation

As the input assembler sources are not constructed by structured code but single instructions, the intermediate representation (*IR*) is nothing more than a depiction of that list of instructions.

Every single instruction of the IR consists of a mnemonic, represented by an enum value, and a list of parameters, represented by objects of different classes according

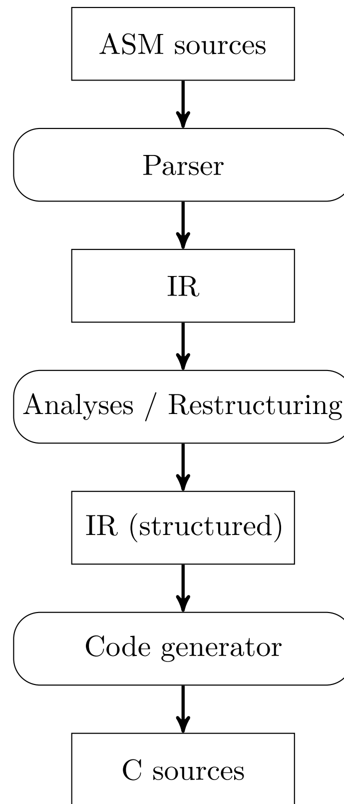


Figure 3.1: Compiler architecture - an overview

to the different types of possible parameters. E.g. the parameter "4" could depict the intermediate integer value 4, the register *R4* or the address 4. Correspondingly, the parameter for the instruction would be of type *ImmediateNumeric*, *Register* or *AddressLiteral*. All those types have in common their parent class *Expression*. An expression can be evaluated to get the scalar value it is representing. This is definitely possible at runtime, constant expressions like immediates are allowed to be evaluated at compile-time, too.

There are two main types of instructions which will be described following: *HLASM* directives and *S390* instructions.

S390 instructions represent the instruction set of the processor. They form the program logic and have an offset and length which is defined by the instruction set architecture (*ISA*) of the S390 architecture. Therefore, they are *MemoryConsumingInstruction*s which means that they are adressable and written to the assembled code in binary form.

There are two main types of S390 instructions: arithmetic/logic operations and branches. The former include calculations that produce results and comparisons which set condition codes. Some instructions might also do both, performing calculations and setting condition codes. The latter transfer control flow, either unconditionally or

conditionally, commonly evaluating condition codes set previously. Those two classes are adequate to model the behaviour of most instructions, i.e. it is not necessary to have any more subclasses but it is sufficient to distinguish them using the mnemonic set. Instructions not fitting this scheme have to be implemented separately, which is the case for *SupervisorCall*.

HLASM instructions provide additional means for application developers and do not correspond to instructions of the processor. They might be assembled as part of the compiled program, i.e. they are *MemoryConsumingInstruction*s in our model, or they might just change the behaviour of the assembler itself. Likewise they are not part of the assembled program on the host, the latter will not be translated to generated C code themselves but might change the generation of code in general.

There are two HLASM instructions that consume memory, both for declaring data structures: *DC* and *DS*. Instructions not consuming memory[1] include *DSECT*, *USING*,... (cf. [IBM08]).

The hierarchy of instruction types used for the IR is shown in figure 3.2.

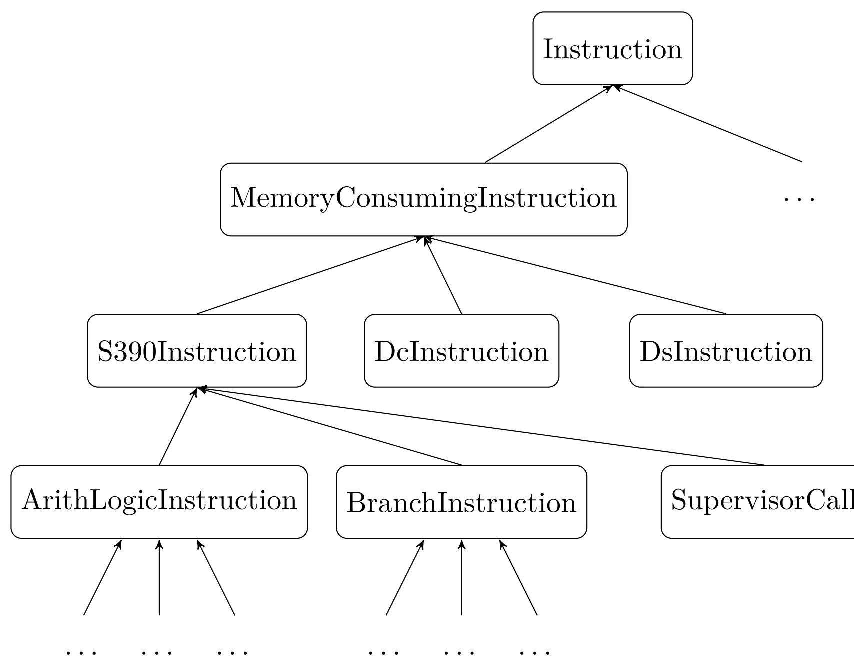


Figure 3.2: Class hierarchy of instruction types

Analyzing steps and code generation rely on additional information that is not yet depictured by the core intermediate representation. Section 3.4 will describe how instructions are wrapped to nodes suitable for analyzing the control- and data-flow. For describing program behaviour and even target language code generation, this core intermediate representation is satisfactory.

[1]More exactly: not being adressable as DC/DS instructions are allowed to consume 0 bytes.

3.3 Parser

As the assembler code is not structured but consists of nothing more than a list of individual instructions, it lends itself to be processed line-by-line. The only possible relationship between two instructions is one referencing the other. Therefore, the referencing instruction has to know the offset of the referenced one. Forward references are allowed, hence it is necessary to process the input sources in two passes:

- First pass processes all instructions to get their offsets (by summarizing their operation code sizes on the host).
- Second pass processes the instruction operands which includes references to other instructions (by label names).

Figure 3.3 gives a first basic sample of ASM code as it is read by the parser. The

```
         SR    R1,R1        subtract R1 from R1 (set R1 to 0)
         CR    R1,R2        compare R1,R2
         BE    TC1_TRU      if equal, branch to TC1_TRU
         A     R15,DUMMY    add DUMMY to R15
         B     TC1_END      branch to TC1_END
TC1_TRU  S     R15,DUMMY    subtract DUMMY from R15
TC1_END  SR    R15,R15      subtract R15 from R15
```

Figure 3.3: first code sample

presented code comprises conditional code execution, known as *if-else*-construct in structured programming languages. There are two separate code paths among which only one is chosen to be executed. This is accomplished by branches to labels *below*[2] the branch instruction itself. Therefore, it is necessary to know all labels including their instructions before any operands can be resolved.

As already mentioned, the code is processed line-by-line. This is accomplished by simple string operations only, it is not necessary to implement a complex parser using a grammar. Parsing is even facilitated by the fact it is possible to respect the actual context of the program or one single instruction that is instantly processed. E.g., if an instruction requires a register as parameter, the parser can look for a register and interpret "R1" or "1" as register. Without context information it would not be clear whether "1" was a register or something else, like an immediate number or memory address.

Contextual parsing of expressions is handled by an *expression parser*. It is fed by single expressions (i.e. instructions are split before and the expression parser is called

[2]Typically, one can distinguish between branches forming conditional code execution and branches forming loops simply by regarding the direction of the branch: if it is going downwards, code execution is conditional, if it is going upwards, code is executed repeatedly.

with every single parameter expression) and called depending on which type of expression is expected.

The expressions returned might be *absolute* or *relocatable*. A field address (including labeled instructions) is always relocatable as its relative offset within the program section can be determined at compile-time but its final location will be determined by the operating system when the program is loaded into main memory before execution. Relocatable expressions are always handled with a reference to the section they belong to. Sections are introduced in the program by the *CSECT* or *DSECT* command. The difference between two relocatable expressions within the same section is absolute: the relocation-offset will be added to both expression values so it will be distinguished when calculating the difference. Examples for absolute expressions are constant values or lengths, which are not manipulated when the program is copied to main memory.

An example for parsing an *Add*-instruction is given in figure 3.4.

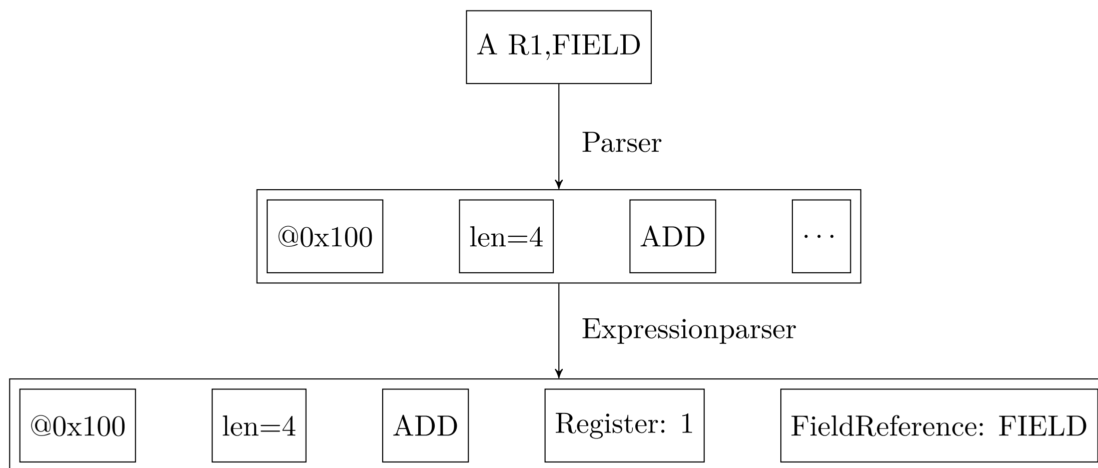


Figure 3.4: Parser processing a sample instruction

3.4 Analyses

The main purpose of the analyses described below is restructuring the program. The intermediate representation returned by the parser is completely unstructured. In order to find control structures (conditionals and loops), it is necessary to collect information about the program flow. The program is then restructured in several steps that will be discussed now.

3.4.1 Control- and Dataflow

The first step of restructuring an unstructured program is to determine the control flow. At this stage, a list of all subsequent instructions (including their addresses and their operands) is available. All instructions are wrapped to *CFGNode*s which may have paths

to *to*- and *from*-nodes wrapping instructions the control flow goes to or comes from. These nodes are stored to a table that enables lookups using the memory location (i.e. section and relative offset) to retrieve them. Once this table is complete, the actual control flow detection is started.

Beginning with the first (entry) node, all reachable nodes are linked and the analyses continues recursively with those nodes. There are several different cases which nodes have to be considered as reachable:

- For non-branching instructions, only the next instruction in memory is reachable.
- For unconditional branches, only the instruction at the target address is reachable.
- For conditional branches, both the next instruction in memory and the instruction at the target address are reachable.

While it is trivial to determine the next instruction in memory, it might be challenging to evaluate the target address. In case the target is a label, it is obvious which instruction is reached by the branch. But what happens if the target is determined by the value of a register or memory location? In this case, the target is depending on a value available only at runtime. In principle, it might get impossible to solve the problem of translating such branches to proper structures. In practice, there are good chances to success, though. In some situations, the address of a label might get loaded to a register just before a branch to that register. That means that it is possible to determine the control flow path at compile time if we trace the register's values. This is where data flow analyses comes in.

As described, it might be necessary to know register or memory values to determine the control flow. Therefore, it is needed to trace values as they get set. Of course, not all values are available at compile time resulting in undefined values. This analysis concentrates on values defined at compile time and acts on the assumption that undefined values mostly come from data processed by the application and values needed for determining the control flow are defined predominantly.

In some situations, it might be desirable to track more than one single value. An example is given in figure 3.5. In this case it is still possible to generate useful C code.

```
         L    R3,FOO      R3 points to FOO
         CR   R1,R2       compare some registers
         BE   LBLA        skip next instruction if equal
         L    R3,BAR      R3 points to BAR
LBLA     BR   R3          R3 points to FOO or BAR
```

Figure 3.5: Branch with two different branch targets

Although it might not be possible to generate structured code, the possible targets of

goto statements are known at compile time, thus it is possible to generate labels for these statements.

Branches to arbitrary addresses not known at compile time cannot be translated to valid C code. When the translated C program is compiled to binary code, instructions will reside at different memory locations than their corresponding counterparts on the host.

The only chance to solve that problem would be to maintain a mapping from addresses of assembler instructions to addresses of C statements. Obviously, that is not appropriate with respect to maintainability. Application programmers would have to adapt this mapping whenever they modify an application which obviously is very error-prone.

Summarized, this implementation does not support the translation of branches whose targets cannot be determined statically. In such cases, the translation fails and manual intervention is required. Application programmers would have to rewrite the concerned source code to allow all branch targets to be resolved statically.

The evaluation of branch targets is done by the *value-set-analysis* used for determining the program- and data-flow described below. Branch targets are identified by values stored in registers or memory locations and therefore do not need to be handled differently than any other value.

*CFGNode*s have two sets of values, representing the set of incoming values which is taken over by the incoming nodes and the set of outgoing values which is calculated by applying the node's instruction on the incoming value set and then passed to subsequent instruction nodes. Whenever a value cannot be determined, it is said to be *undefined* which is represented by an empty value set. The set of values is limited because too comprehensive value sets are of no more use than handling a value as undefined in terms of detecting the program control flow.

Control- and dataflow-analysis are performed iteratively. Backward branches (loops) might extend the value set of a previously visited node. Keeping in mind that branch targets are depending on data sets, enlarged value sets can introduce new code paths. Additional code paths, on the other hand, might lead to additional possible values. Both analyses influence each other. Therefore, the algorithm has to be applied until no more changes occur.

Every time the data set for a node is (re)calculated, it is compared with the previously set values. As long as changes are observed, the algorithm continues to process all successor nodes. As already mentioned, the size of the value sets is limited, this will prohibit too long evaluation times that would only lead to huge value sets that are of no use.

This described behaviour is also presented in brief in algorithms 1 (detecting control flow paths and handling instruction side-effects) and 2 (merging data values from incoming instruction nodes).

Algorithm 1 Control- and Dataflow Analysis using Value-Sets

```
 1: nodesToProcess ← {entryNode}
 2: while nodesToProcess ≠ ∅ do
 3:     n = pop next from nodesToProcess
 4:     reprocessNeeded ← n.mergeState()
 5:     if reprocessNeeded then
 6:         copy all values from the incoming value-set to the outgoing value-set
 7:
 8:                                                   ▷ handle data manipulation first
 9:         switch n's instruction's mnemonic do
10:             case AR                                              ▷ Add Register
11:                 use R1, R2 as instruction's operands
12:                 VS_out(R1) ← {r1 + r2 | r1 ∈ VS_in(R1), r2 ∈ VS_in(R2)}
13:             case SR                                         ▷ Subtract Register
14:                 use R1, R2 as instruction's operands
15:                 VS_out(R1) ← {r1 − r2 | r1 ∈ VS_in(R1), r2 ∈ VS_in(R2)}
16:             case BR                                           ▷ Branch Register
17:                                       ▷ do nothing, only affects control flow
18:             [...]
19:         end switch
20:
21:                                            ▷ prevent exorbitant value set sizes
22:         for all vs ← VS_out do
23:             if |vs| > max_vs_size
24:                 vs ← ∅
25:         end for
26:
27:                                                       ▷ handle control flow
28:         switch n's instruction type do
29:             case branch instruction
30:                 if branch instruction ≠ NOP
31:                     n.to ← n.to ∪ {branch target(s)}
32:                 if branch is not unconditional     ▷ conditional branch or NOP
33:                     n.to ← n.to ∪ {n's successor in memory}
34:             case arithmetic/logic instruction
35:                 n.to ← {n's successor in memory}
36:         end switch
37:
38:         nodesToProcess ← nodesToProcess ∪ n.to        ▷ process successor nodes
39:     end if
40: end while
```

Whenever a node is handled, new program paths can be detected or its instruction's

semantics can lead to updated value sets. If this is the case, subsequent nodes have to be reprocessed. Therefore, the function *mergeState* is called for every succeeding node. It will recalculate the value sets for both registers and memory locations and determine whether this recalculation has led to different results and thus nodes have to be reprocessed further to detect possible new program paths.

To diagnose whether a value set has been updated, it is not necessary to compare all values but sufficient to compare the size of the value set only. Values are not appended to the set before it has been shown that they are possible values for a register or memory location. Therefore, they will not be removed under any circumstances later. In case the value set grows too large or it turns out that it is not possible to reasonably delimit the set of possible values, the set will be set to *undefined* which means that any value is considered as possible. Summarized, we have to check whether the value set has grown or been set to *undefined*, otherwise, we can assume that it has not been altered.

Memory locations are handled like registers. Instead of an array of registers, they are organized by a map using memory addresses as keys. Unaligned memory accesses are supported but in case overlapping value sets are detected, they are set to *undefined*.

Algorithm 2 Control- and Dataflow Analysis using Value-Sets (cntd.)

```
 1: static revision = 0                  ▷ use node revision to track iteration information
 2:
Require: node's revision has been initialized to 0 prior to first run
 3: procedure mergeState(node)
 4:     needMerge ← false                         ▷ required to merge incoming nodes
 5:     triggersUpdate ← false              ▷ successor nodes will have to be reprocessed
 6:     if this.revision = 0 then
 7:         needMerge ← true
 8:         triggersUpdate ← true
 9:     end if
10:
11:     if needMerge then
12:         for all reg ← registers do
13:             numValues ← vs_in_reg
14:             for all from ← node.from do
15:                 vs_in_reg ← vs_in_reg ∪ node.from.vs_out_reg
16:             end for
17:             if |vs_in_reg| ≠ numValues then
18:                 triggersUpdate ← true
19:             end if
20:         end for
21:     end if
22: end procedure
                                          ▷ handle memory locations likewise registers
```

In favor of clarity, the presented algorithm describes only the basic behaviour of the processes. Many details that have been implemented – mostly concerning error handling – are hidden. Some more details have to be handled carefully, e.g handling values is a bit more complicated because values can be absolute but also relative to some section.

Of course, whenever an operation is executed for different value sets, those values have to be checked whether they are both absolute or part of the same section. Binary operations have different restrictions on absoluteness or code section membership of operands.

E.g., it is not possible to evaluate an addition of two values part of the same or different sections. As the section offset will be determined not before the program is run, adding those offsets will lead to senseless values. On the other hand, adding an absolute value to a relative makes perfect sense as it will point to some field relative to the section independent of the actual location the section will be placed at runtime. Of course, the rules are not the same for different instructions: e.g. subtracting a relative value from some other relative value, both referring to the same section, is useful as it will return the offset of some field relative to some other field (or calculate the length of a field / set of fields).

3.4.2 Elimination of Condition Codes

While it seems to be trivial to translate machine instructions to corresponding C statements at first glance, it is a bit more complicated to imitate the exact behaviour including side-effects. This involves condition codes that are set by the processor. E.g. a simple *Compare* instruction sets flags to indicate whether the result is equal, the first operand is lower or the first operand is higher than the second. Arithmetic instructions might set whether a result is zero, positive, negative or an overflow has occurred. Depending on the branch instruction following, a different set of those condition codes is evaluated while the others might not get used at all. In many situations, even none of the flags will be read. While comparisons are often executed to determine further control flow (i.e. they are followed by branch instructions), arithmetic operations will often be executed and their condition codes ignored.

Calculating these condition codes is not a complex task - but comparing with the processor doing that calculation in the same cycle as side-effect while executing the instruction, there will not be any comparable C code without performace hits. This step concentrates on finding condition codes that are actually read and marking all others to be hidden. This will reduce the size of generated code (assignments to the condition code field will just be removed) with the positive side-effect of improving performance.

Finding condition codes is a liveness analysis[AP02] and therefore executed backwards. The four condition codes available on S/390 are handled as separate variables whose liveness will be determined. If an instruction reads one or more of those flags, they will be marked as *used*. On the other hand, when an instruction sets condition codes, the whole set will be marked as *defined* which means that previously set values are not *live* anymore and if not yet read previously, those values will never be needed. Thus, wherever

those condition codes have been set, the code for calculating the codes has not to be generated.

Even if a condition code is used, it will only rarely be necessary to generate it's assignment. The code in figure 3.6 shall be given as example.

```
CR  R1,R2       compare some registers
BE  LBLA        branch to LBLA in case they are equal
```

Figure 3.6: Sample usage of condition codes

Obviously, the condition code indicating whether the operands of the comparison are equal or not is needed. It is read by the subsequent instruction to evaluate whether the operands have been equal. But in this case, instead of storing the comparison results to flags and evaluating those, the comparison can be evaluated as needed resulting in code as the following (figure 3.7; pseudo code for demonstration purposes).

```
if r1 == r2:
    // code from LBLA
```

Figure 3.7: Sample generated code hiding condition code fields

The condition can be generated together with the branch instruction whenever there is only one possible comparison instruction that might set the condition codes read by the branch instruction. If all branch instructions that read a comparison's condition code results can only read those condition codes set by this single comparison, the comparison's condition codes are not used at all as all comparisons are generated as part of the branch instruction. This is the general case as most comparison/branch instructions occur pairwise.

3.4.3 Restructuring Control Flow

At the time the control flow graph has been determined, the program can be restructured to gain an intermediate representation suitable for generating structured code. This is done in three main phases: at first, basic blocks are derived from the control flow graph. Basic blocks lead directly to loops. Secondly, dominator and follow nodes are determined to find conditionals (if/else). At last, as the concerned nodes are marked as forming the beginning or end of a loop or conditional structure, the nodes are assigned to a structured graph.

Intervals

Intervals form partitions of the control flow graph. In graph theory, an interval is the maximum subgraph that has only one single entry node. The algorithm starts with the

entry node of the program which forms the header node of the first interval. Directly reachable nodes are added to this interval as long as all their predecessors are already part of the interval. This assures that the whole interval has only one entry node. When no more nodes can be added to the interval, the next reachable node forms the header of the next interval.

As some nodes are linked with nodes of other intervals, those intervals are connected and form a graph themselves. Hence, the algorithm can be applied on the graph of intervals again, resulting in a derived graph. This procedure can be repeated as long as a derived graph differs from the graph it is derived from and the graph consists of more than one single node.

As a loop header always has a predecessor (the latching node) pointing "back" to the header, this header node has a predecessor that is not yet part of the interval. Therefore, loop header nodes always form the header of a new interval. The sequence of derived graphs depictures the nesting depth of loops in the program analyzed. A sample is given in figure 3.8.

The algorithm used is described in [Cif96] as part of the *dcc* decompiler[Cif94].

Dominators and Follow Nodes

Node A *dominates* node B when all paths to B include A. If we refer to the previous example given in figure 3.8: node 1 dominates all other nodes as it is not possible to reach any other node not passing node 1. As there are no forward jumps for the first few nodes (until node 9), all these nodes dominate their subsequent nodes. Nodes 11/12 do not dominate any other node as their successor is node 13 which can be reached by both of them, thus, no single node is required to be passed. The last dominator node for 11/12 but also 13 is node 9, which has to be passed in any case, no matter which path is taken.

A node has a list of dominators which is the set of all nodes that have to be traversed to reach it. The last node traversed to reach the node is called *immediate dominator*. The set of dominators always consists of the union of the immediate dominator's dominators and the immediate dominator itself.

A *common follow node* is the next node that will be reached by two different branches of the control flow. Again refering to figure 3.8: node 9 has edges to both 10 and 12 but in any case node 13 will be reached and merge the control flow branches. Node 13 is therefore the common follow node for 10/12.

Algorithms for determining dominators and follow nodes are again taken from [Cif96] where more detailed information can be obtained.

Building a Structured Representation

As soon as basic blocks, dominators and follow nodes have been identified, it is possible to build a structured representation of the program. As mentioned, loops can be detected by using (derived) interval graphs. Conditionals are detected by branches merging in common follow nodes but it is not obvious whether a branch forms a loop or not. Therefore, structuring starts with loops and continues to detect conditionals afterwards.

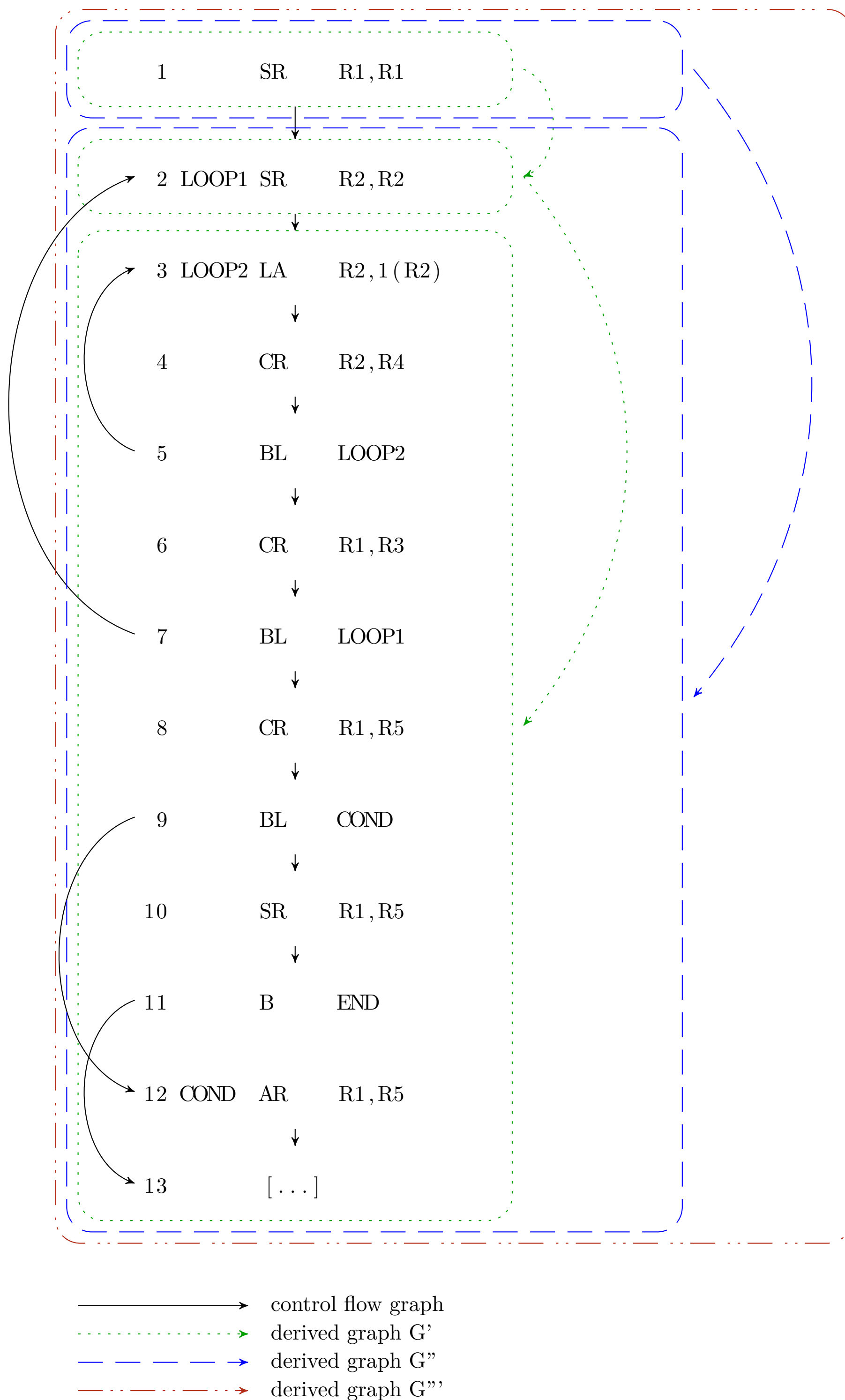


Figure 3.8: Sample code with basic blocks

Apart from simple control flow constructs, there might also exist code forming functions, i.e. code that is only called by *branch-and-link* (BAL/BALR) instructions and always returns to the code where it has been called from. Therefore, the control flow graph is searched for code that can be extracted to separate functions. Detection of possible entries to separate functions is described in algorithm 3.

Algorithm 3 Detecting potential entries of separate functions

1: $functionEntries \leftarrow n \in nodes$
2: $\quad | \; |n.from| > 0 \wedge$
3: $\quad \forall f = n.from : f$ encapsulates a BAL-instruction

Those possible entries are then inspected whether all code paths from the entry lead to some branch back to the callee and the resulting code section has no other entries than the one entry node itself. The complete algorithm for these checks is shown in algorithm 4.

If it is not possible to prove that a designated entry forms a sound function, branch-and-link instructions will not be replaced by function calls but will remain. Thus, it will be necessary to generate labels and corresponding *goto* statements as part of the translated program.

Hence, the control flow graph will be transformed to a structured representation. This is a recursive approach starting with the entry node of the program. Whenever a control structure (e.g. loop or conditional) is detected, it will be handled until its end. The basic procedure to do so is shown in algorithm 5.

Algorithm 4 Checking separate function validity

Require: List of preselected possible entries: they all have one or more incoming edges pointing to no other than encapsulated BAL instructions, i.e. they might not be accessed other than by BAL.

```
for all possible entries e do
    returnRegister ← return register from BAL instruction e.from_0

    ▷ All calls have to use same return register
    if ¬∀rr = return reg from BAL instruction e.from : rr == returnRegister then
        break                                         ▷ no valid function entry
    end if

    processNodes ← {e}
    memberNodes ← ∅
    unresolvedFrom ← ∅

    repeat
        n ← pop next (any) node from processNodes
        memberNodes ← memberNodes ∪ {n}
        unresolvedFrom ← unresolvedFrom ∪ {f ∈ n.from | f ∉ memberNodes}
        processNodes ← processNodes∪
            {t ∈ n.to | t ∉ memberNodes ∧ t does not branch to returnRegister}
    until processNodes == ∅

    ▷ By now, all previously unresolved nodes of incoming edges should have been
added as members to the function. If so, a valid function has been detected that can
be generated separately.

    if ∀u = unresolvedFrom : u ∈ memberNodes then
        e is a valid entry of a separate function
    end if

end for
```

Algorithm 5 Restructuring of Intermediate Representation

```
1:  scopes ← empty stack
2:
3:  procedure restruct(entry)
4:      unprocessed ← {entry}
5:      while ¬(unprocessed == ∅) do
6:          restructCodeBlock(unprocessed_0)
7:          unprocessed ← unprocessed \ unprocessed_0
8:      end while
9:  end procedure
10:
11: procedure restructCodeBlock(node)
12:     repeat
13:         node ← restructNode(node)
14:     until node == null ∨ node.to == null ∨ node.to == ∅
15: end procedure
16:
17: procedure restructNode(node)
18:     if node already visited then return null
19:     end if
20:     visited ← visited ∪ {node}
21:     if node is BAL/BALR ∧ branch target is valid function entry then
22:         create call statement to branch target return next node in memory
23:     else if node is loop header then
24:         restructLoop(node) return node following loop
25:     else if node is branch to register then
26:         if |node.to| == 1 then
27:             create goto statement to referenced target instruction
28:             mark label of referenced target instruction used
29:         else
30:             create goto statement to register value
31:         end if
32:     else if |node.to| == 1 then
33:         if node represents branch then
34:             if ∃loop : node ∈ {loop.header, loop.latch} ∨ ∃f : (f == follow(node)∧
    |f.size| == 1) then      ▷ Branch implicitly handled by control structure, no explicit
    code generation needed
35:             end if
36:         else
37:             if node represents BAL/BALR then
38:                 add LA to store next instruction address to return register
39:             end if
```

Algorithm 5 Restructuring of Intermediate Representation (ctnd.)

```
40:             create goto statement to referenced target instruction
41:             mark label of referenced target instruction used
42:             unprocessed ← unprocessed ∪ node.to
43:         end if
44:     else if |node.to| == 2 then
45:         if node has follow node then
46:             ifStmt ← newIfStmt
47:             for all to ← node.to do
48:                 if to is follow node then
49:                                                         ▷ do not process here
50:                 else
51:                     if to is subsequent node in memory after node then
52:                         handle code block as if-branch
53:                     else
54:                         handle code block as else-branch
55:                     end if
56:                 end if
57:             end for
58:         else
59:             assert(node has to represent a branch instruction)
60:             if ∃node.to : node.to is succeeding node in memory then
61:                 conditional branch
62:                 create If-Stmt
63:                 get branch target, create goto statement, add to if-block
64:             else
65:                 create goto statement
66:             end if
67:         end if
68:     else
69:         error handling
70:     end if
71: end procedure
72:
73: procedure restructLoop(node)
74:     ▷ similar to restructCodeBlock, abort processing nodes as soon as latching node
    of loop is detected
75: end procedure
```

3.5 Code generation

By now, the assembler program to be translated has been parsed and analyzed completely and is present in a structured intermediate representation that is perfectly suited to be processed using the *visitor pattern*[GHJV94].

There are two main parts for writing a C(++) program: writing the header file with all field definitions and function prototypes and writing the actual program code to the program file. Before those two parts will be discussed following, we will have a look on design desicions regarding the code generation process in general and which further prerequisites are necessary to write the final code.

3.5.1 General C(++) Code Generation

Beside processor instructions, HLASM programs consist of data declarations. Fields are of different types, e.g. binary numbers, decimal numbers, character strings, etc. While some types, like binary numbers, are supported in both C(++) and our target platform x86_64, others, like decimal floating points, are not available. There are two main approaches to solve this problem:

- Fields could be generated untyped, e.g. as *char[]*-fields and used by functions of a runtime-library implementing specific operations that will then interpret the values as the desired data types.
- The runtime-library could implement the needed types and support needed operations to be executed on fields of those types, e.g. fields could support the basic arithmetic operators (+, -, ...).

The second solution is not possible using plain C as operator overloading is possible in C++ only. Further, some of those types require parameters. It is possible to have numbers with specific lengths in packed decimal format. It therefore has to be possible to have types that can have varying dimensions. This is perfectly possible using C++ templates. Concerning the first approach, this is no issue at all: lengths of data types could just be passed to the operations together with a pointer to the untyped field. Of course, it then was necessary to pass this length information every time a field is used.

This implementation uses a runtime-library implementing native S390 types not available on the target platform. Using those types in the generated code even does not really interfere with the requirement that the target code should mostly be in plain C, not C++. The library implementation, involving details making intensive usage of C++ template techniques is hidden in the runtime-library that will not be visible to application programmers (as long as they focus on application programming as on the host and do not intend to touch the underlying base). Using the operators and templated types in the program code is of course C++-specific but this usage does not differ from the usage of native types, like integers. From the application programmer's view, this generated code should be easy to maintain even without any knowledge of C++ (template) programming.

The library used has been made available from preceding projects and is not publicly available. It will therefore not be discussed as part of this thesis. Few examples of its usage should be given in figure 3.9 to get a first impression:

```
1 fixed_decimal<7> fd = 0;
2 pic<PIC_9<5>> pic = 10;
3 fd += 100;
4 while (pic < fd) {
5     pic = pic + 3;
6 }
```

Figure 3.9: sample usage of runtime library

As mentioned, parameters like length information for types can be passed as template parameters. E.g., the field *fd* stores a signed decimal value with 7 digits and therefore occupies 4 bytes[3]. Operators (assignments, arithmetic operations, comparisons) are overloaded to provide the exact same behaviour as the corresponding types on the host architecture.

If fields are passed as operands to instructions that require different types than those provided, appropriate casts are added.

3.5.2 Generation of Header Files

As discussed in previous section 3.5.1, a runtime-library provides implementations for the host types. Hence, fields used in assembler programs will be generated as typed fields in the translated program. Fields part of the assembler program are unstructured, i.e. they are all directly part of the program and not subordinated to some structure, function or even tighter scope.

However, fields are organized in sections introduced by the *DSECT* instruction. A section is a contiguous, relocatable area of memory where instructions or data can be placed to. On execution, sections will be established in main memory and thus implicitly receive their addresses. The offsets of fields or instructions inside a section will not be touched as a section will always be handled at once. Certainly, different sections might get arbitrary memory locations assigned and might or might not be aligned consecutively. Those sections are translated as separate C-*struct*s.

It is common to use a sequence of fields like a structure or object of a class in an object-oriented language. This is accomplished by a field of 0-byte length just before the associated fields. The header field will then not occupy any storage to hold data but share the address with the first field being part of the "structure". Technically, there is no necessity to have this field but it is a common convention to express that a sequence of fields belonging together is addressed.

[3]The length of decimal fields is calculated by the formula (<number of digits> + 1) / 2 as every digit and the sign occupy a half-byte each

There even have been considerations to use those constructs to partition sections to even finer-grained blocks and generate separate structures for every sequence of fields introduced by one 0-byte-field. However, it turned out that while it is easy to find the beginning of a designated structure, it is nearly impossible to find its end. Most structures are not followed by the next clearly separated structure but often single fields are in between. It might also occur that structures consist of a set of fields but depending on the current usage the whole structure or only part of it is used. After all, it seemed to be more appropriate to generate a list of fields on the same level and translate 0-byte-fields as additional pointers pointing to the first memory-consuming field of the 'structure'.

Figure 3.10 gives a small example of definitions for few fields forming a logical record collecting some person data. The automatically[4] generated code is shown in figure 3.11.

```
1 SECT1     DSECT
2 ID        DS     AL4
3 NAME      DS     0C
4 FNAME     DS     CL20
5 LNAME     DS     CL20
6 BORN      DS     PL5
7 BALANCE   DS     AL4
8 VALUES    DS     4AL2
```

Figure 3.10: Sample field definitions in HLASM

```
1 struct _ASM_T_SECT1 {
2     int32_t ID;
3     fstring<20> FNAME;
4     fstring<20> LNAME;
5     fixed_decimal<9> BORN;
6     int32_t BALANCE;
7     int16_t VALUES;
8 } *SECT1;
9 fstring<0> *NAME = (fstring<0>*) ((char*) SECT1 + 4);
```

Figure 3.11: Sample field definitions in C++

While it seems obvious for the reader that the field *NAME* will be used to address the two name fields *FNAME* and *LNAME* belonging together, it is hard to find an algorithm capable of recognizing this relationship.

[4] In general, samples of generated code are produced automatically and do not contain any manual changes with few exceptions: in favor of clarity, namespaces of the runtime-library as well as comments containing the original assembler source-code are omitted.

3.5.3 Generation of Source Files

The most challenging part of generating program sources has already been accomplished by restructuring the program. We now have a structured intermediate representation that can be processed by a *visitor*.

Most instructions are translated individually, i.e. there is one C-statement replacing the original instruction.

Arithmetic operations are implemented by operators that are overloaded for all host-types provided by the runtime library. Therefore, there are no different operations for different operand types but different operand types passed to the same operator symbol. This is happening implicitly in case typed fields are passed to the operations. In case memory locations or field of divergent types are passed, casts are inserted to obtain the desired behaviour.

Branch instructions are not generated at all in the optimal case - when their semantics are preserved implicitly by control structures introduced. In case this has not been possible, they are replaced by *goto* statements.

Most control structures rely on conditions determining under which circumstances (if at all or how long) their encapsulated code is executed. Generally, condition code flags are not necessary and have already been removed. Conditions of control structures are generated by regarding a pair consisting of a comparison and a branch instruction.

E.g., having conditional code (i.e. a simple *if*-block in a structured language): a *comparison* instructions is used to receive a condition code by comparing two registers. A *branch* instruction is then used to *conditionally* skip some lines of code if the last comparison has shown that the both registers are equal. The branch condition is inversed for generating C-code. While the branch instruction in assembler determines whether to skip some lines of code, in C the condition determines whether the code containing this some lines of code should be executed or not. This behaviour is demonstrated in a very simple first example by figures 3.12 and 3.13.

```
1          SR      R1,R1
2          CR      R1,R2
3          BE      TC0_TRU
4          A       R15,DUMMY
5 TC0_TRU  S       R15,DUMMY
```

Figure 3.12: Conditional code in HLASM

The instruction *BE* (*branch-if-equal*) evolved to a comparison whether two registers are not equal.

More sample code, including loops, is provided by figures 3.14 and 3.15.

In case the generation of a *goto*-statement is necessary, the target is always tried to be expressed as literal label name. This is also true for branches to register values in case those value could have been determined beforehand. If this value is the address of an instruction with a label, this label name is used as *goto* target. Otherwise, a label name

```
R[0] = 0;
if (R[0] != R[1]) {
    R[14] += *(DUMMY);
}
R[14] -= *(DUMMY);
```

Figure 3.13: Conditional code in C

has to be generated: therefore, the name of the previous label in memory is suffixed by the offset of the target to this previous label. In case there is no previous label, a label name using the hash value of the instruction is generated. Of course, this name is not meaningful then.

```
1  * test case 1: if/else
2           SR      R1,R1
3           CR      R1,R2
4           BE      TC1_TRU
5           A       R15,DUMMY
6           B       TC1_END
7  TC1_TRU  S       R15,DUMMY
8  TC1_END  SR      R15,R15
9
10 * test case 10: pre-conditional loop
11          SR      R1,R1
12 TC10_H   CR      R1,R2
13          BNL     TC10_A
14          LA      R1,1(R1)
15          B       TC10_H
16 TC10_A   SR      R15,R15
17
18 * test case 12: post-conditional with branch at header
19          SR      R1,R1
20 TC12_H   CR      R1,R1
21          BE      TC12_A
22          A       R1,10(R15)
23 TC12_A   A       R1,14(R15)
24          CR      R1,R5
25          BL      TC12_H
26          SR      R15,R15
27
28          A       R15,DUMMY
29 FOO      DS      0H
30          A       R15,DUMMY
31          CR      R15,R1
32          BL      FOO
```

Figure 3.14: Sample code to be structured in HLASM

```
1  /* test case 1: if/else */
2      R[0] = 0;
3      if (R[0] == R[1]) {
4  // TC1_TRU
5          R[14] -= *(DUMMY);
6      } else {
7          R[14] += *(DUMMY);
8      }
9  // TC1_END:
10     R[14] = 0;
11 
12 /* test case 10: pre-conditional loop */
13     R[0] = 0;
14     while (R[0] < R[1]) {
15         R[0] = (ptrdiff_t) (ptrdiff_t) (1) + (ptrdiff_t)
             (R[0]);
16     }
17 // TC10_A
18     R[14] = 0;
19 
20 /* test case 12: post-conditional with branch at header */
21     R[0] = 0;
22     do {
23         if (R[0] != R[0]) {
24             R[0] += *((int32_t*) ((ptrdiff_t) (10) +
                 (ptrdiff_t) (R[14])));
25         }
26 // TC12_A
27         R[0] += *((int32_t*) ((ptrdiff_t) (14) + (ptrdiff_t)
             (R[14])));
28     } while (R[0] < R[4]);
29     R[14] = 0;
```

Figure 3.15: Structured code in C

CHAPTER 4

Evaluation

4.1 Current State of Implementation

The implementation has been tested by translating an enterprise real-life application consisting of 542 assembler programs containing about 270.000 lines of code (including comments, empty lines, etc.). The source has been made available as-is, i.e., there are no guarantees that there are no deprecated programs, dead code or on the other hand missing code. As the implementation advanced, it turned out that the provided code is rather incomplete.

Most notably, lots of macros used by the programs are not available. Even though a macro processor is outside the scope of this work, a primitive processor has been implemented that can handle simple code replacements and ignore missing macros. Of course, if macros are just skipped the generated code is of no use for the application user. But still, the code is valuable as test case for code generation.

E.g., input/output operations or user interactions are often handled by macros. While these are inevitable for the program logic, they only play a minor role for code translation. IO operations only read or write data strings and do not interfere with program control flow. They would be generated as simple function calls to a runtime-library. Hence, the implementation will ignore macros not provided (or too complex). It will still be possible to evaluate whether it is possible to parse, analyze and generate those programs. It is not expected that generation fails because of missing macros. Likewise, it is assumed that programs will not fail caused by macros that could have been translated without them as long they have been missing.

Apart from simple macros for e.g. IO, there are others including field definitions. While the first are just skipped, the latter are causing troubles with the current implementation. Many programs rely on fields that are missing their definitions. While it would be possible to implicitly declare those fields as necessary, it has been chosen to interpret these situations as errors.

Table 4.1 summarizes the state of translation for all programs of the test case. Less than half of the programs is considered therefore as the other part does not contain any instructions apart from field definitions at all. After ignoring programs that even could not have been parsed, there remain 119 programs of interest.

Total programs	542
Programs failed (missing fields)	134
Empty programs	289
Remaining programs	119
CFG ok	100
Generation ok	74

Table 4.1: Status of program conversion

There are 19 programs whose control flow could not have been deduced. Some programs receive a function address as parameter and call this function. As the address is passed from outside at runtime, its value cannot be determined by the static control flow analysis. Therefore, those calls will never be possible to be resolved. The same holds for dynamic address calculations which cause that branch targets cannot be detected.

Few other cases can be deduced to missing implementation, e.g. handling of *EQU*-instructions as operands might fail in some situations. Those problems are supposed to be fixed in the near future.

Generation is working for 74 out of 100 programs. Errors are mainly caused by instructions that are not yet implemented. This concerns the *EX*-instruction which has not yet been implemented as it is part of the domain of self-modifying code which has been left open completely. Some other instructions, e.g. for text processing, are still lacking their implementation. In general, those are rarely-used instructions that do not have an influence on program control flow and therefore have not been rated as really important for a first evaluation implementation. They are expected to be provided soon, although.

It has to be noted that the current implementation is at a very early stage and has to evolve further to be really useful. Smaller generated programs have been tested and the behaviour seemed to be right. But in general, code generation is not to be considered as tested carefully and generated code might contain bugs or even not be able to be compiled.

The aim of this project is to build a tool that is able to translate code automatically to C(++) under the constraint that the generated code should be as readable and maintainable as possible. It has yet been shown that the tool, despite the fact it is a first draft only that has to evolve further, is capable of translating most of the valid source code yet. But what about code quality and maintainability?

A sample has been provided previously (cf. figure 3.14, 3.15). The generated code does not require great knowledge in C or C++ to be read by an application programmer. By reason of limited space, comments have been omitted in this sample. Figure 4.1 shows

```
1     R[0] = 0;                  // xx:8:     SR      R1,R1
2     if (R[0] != R[1]) {        // xx:9:     CR      R1,R2
3         R[14] += *(DUMMY);     // xx:11:    A       R15,DUMMY
4     }
```

Figure 4.1: HLASM code as comments

how assembler code can remain adjacent to the generated code[1].

The relationship between assembler and generated C statement should be obvious to the reader.

One important requirement has been to restructure the source code. While it is possible to have *goto*-statements in C(++), they are considered as harmful and should be avoided. But primarily, code is expected to be better readable and maintainable if *goto*s are replaced by control structures.

The current implementation is yet able to replace about 81 %[2] of all *goto* statements. Of course, this is extremely depending on the actual assembler sources. If application programmers back off from using disordered branches but try to write structured code that is readable, the percentage of replaceded *goto*s is approaching 100. Otherwise, the results might get disappointing.

Table 4.2 lists the portion of branches in analyzed programs. Nearly one third of the

Total reachable instructions	7836	100%
Branch instructions	2434	31.06%
Unresolved branch targets	3	0.04%
Branches to 3+ possible targets	128	1.63%

Table 4.2: Branch instructions in programs

assembler instructions are different kind of branches. Only three targets of a total number of 2434 branch instructions could not have been resolved. 1.63% of branch instructions seem to have three or more distinct possible targets. Those instructions are translated using *goto* statements.

Although it might not be clear which control structures are replacing branch instructions, it should be simple to understand the semantics of the generated program. Application programmers should be able to work on the generated program sources and identify their control flow without the need of understanding the exact link between the generated and the original sources.

Of course, there are still some other traces indicating that the program has not been written in C but has been generated from some assembler sources. The registers have been translated as a simple array of binary values. Hence, they are used as raw, untyped

[1] Generated code, spaces and file names in comments removed.

[2] Altogether, the programs that could have been analyzed contained 4824 *goto* statements. 893 (19 %) have not been able to be replaced by control structures.

buffer frequently. Actually, register accesses should be replaced by local variables that are typed respectively their usage and are residing in a local scope with limited validity and only one single purpose, i.e. a variable should hold a specific value and not be reused to hold some completely different data in case it is not needed anymore for the old purpose inside some function or other local scope. Nevertheless, readability is not limited compared to the original sources which suffer the same problem.

The use as untyped buffer also results in massive usage of casts. Especially the very common use-case of registers used for indirect addressing requires multiple casts to *ptrdiff_t* to use the declared *integer* value as pointer supporting arithmetic operations. It has to be admitted that not all casts generated are really needed. Removal of redundant casts is to be considered as enhancement to be implemented in the future.

It has been expected that in most cases explicit storage of condition codes is not required and flags therefore can be hidden. The evaluation showed that 3158 times a condition code has been set - but in no single case it has been necessary to explicitly generate code for setting it.

I.e., in every single case it has been possible to identify the comparison or arithmetic instruction that sets the condition code used by a branch and therefore generate conditional code including a comparison expression without querying condition flags.

Performance of the translator has not been an issue during the implementation of this work. The requirements in this regard are very low, thus easy to meet: it should be possible to finish a migration project within few days, usually a weekend. The conversion of assembler projects should only take a reasonable amount of time within the whole project.

While there have not been any regular detailed measurements, performance always seemed promising. A complete conversion of the mentioned applications used for evaluation takes 33 s on the test machine (Microsoft Windows 7 (64 bit) on Intel Core i7-3960X @3.30 GHz, Java 1.8 b123 (64 bit) with default VM settings, single-threaded run).

On one hand, future more detailed and sophisticated analyses are expected to lower the performance of the translation process. On the other hand, there is still room for performance improvements by utilizing multiple processor cores but probably also by optimizing the implementation (profiling has not been needful and therefore not been done at all so far).

Summarized, it cannot be hidden that the generated code has its origins in assembler programming but it is qualified to be read and maintained by application programmers.

4.2 Rating of Implementation State

While only 74 out of 542 programs have been translated successfully, the evaluation can be seen as successful. Most problems are caused by empty programs or missing include files or macros. It has been shown that most idioms of HLASM can be processed by the translator. Unfortunately, there remain some exceptions as not all problems could have been solved. However, no new problems could have been found by the evaluation as all remaining issues have been expected from the very first. Particularly to be mentioned:

- Dynamic calculated addresses used as branch targets that could not have been resolved.
- Self-modifying code.
- ASCII/EBCDIC conversion issues: these problems still remain but have not been addressed by this evaluation at all so far.

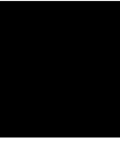
It is hard to determine whether the implementation is of any use for migrating a real-life application yet. Needless to say, it would require many adaptions for the actual assembler source code that has to be processed.

A macro processor is not absolutely required as it was likewise possible to use an existing compiler on the host to resolve macros and get a precompiled representation of the concerned source code. Another option would be to use listing files written by the compiler on the host at compile time as input for code translation. This would require changes in the front end but the amount of work required should remain arguable.

Usefulness is considerably depending on the programming style of the assembler program sources. If dynamic branch target calculations and self-modifying code are not used at all, there are good chances that the translator will produce valuable results. The application used for evaluation only slightly made usage of such constructs. At some point, it would require less work to manually adapt those few spots in the program and then translate it automatically than to manually rewrite the whole program. Additionally, it seems that automatic restructuring is doing a fairly good job yet and it might not be certain whether programmers would achieve comparable results without excessive efforts.

To give a final recommendation in brief: the implementation has evolved over the last few months and while it is in an early stage, the results produced for programs of the application used for evaluation look promising. Depending on the size of a potential project, usage of automatic translation using this implementation could be reasonable.

If a manual translation would require more than half a year, it should be more efficient to put resources into advancing the compiler than rewriting the code manually. Of course, the translator could also be valuable for possible future projects which would make it interesting for even smaller projects. It has to be mentioned that this estimation is only very rough. As said before, it all depends on different aspects of the actual source code. So while some constructs might be easy to translate manually, they could be a pain for automatic processing and vice versa. In-depth assessment of the objective assembler sources is the key to success.



CHAPTER 5

Conclusion and Outlook

This work has described the implementation of a tool for automatic code translation from HLASM code written for IBM hosts to platform-independant C++ code for evaluation purposes. Although the state of development is still far from production quality, the project can be seen as success.

It has been proven that automatic translation is possible for the majority of programs used for evaluating. Still, there remain some topics that can not be seen as completely solved, most notably handling of dynamic calculated branch targets and self-modifying code. But those topics have been addressed and turned out to restrain only a small part of source code from being translated completely automatically.

Code readability and maintainability is given with comments containing the original assembler sources beside the generated code. Semantics of converted instructions are clear and easy to understand even for programmers not familiar with C, C++ or this type of syntax in general. Program flow and structure is even more obvious as structured programming and code indentation are a huge improvement compared to massive usage of branch instructions.

5.1 Future Work

While this first version can be seen as success evaluating the possibility of automatic code translation of HLASM programs, there is still much room for improvements. Foremost, it should be mentioned that this is a *proof-of-concept* implementation leading to the opinion that it might be worthwhile to pursue this work. Therefore, there are still some cases not yet implemented, not well-tested or not tested at all. Efforts should be put into hardening the implementation.

Having said that, there are some more specific improvements as well that will be discussed in the following.

5.1.1 General Code Style Discussions

There have been some discussions regarding the style of the generated code. Some of the aspects thereof shall be named below.

Replace Operator Overloads by Library Method Calls

Operator overloads hide details of arithmetic operations from the application programmer. This might be strived for when the produced code should be easy to read and understand. On the other hand, assembler programmers are already used to selecting instructions depending on the type of data that should be processed. Even for advanced C++-programmers, it might not always be obvious at first sight which semantic details are hidden behind a specific overload. If different operand types are mixed in an operation, the rules for casting and method dispatching might get even more complicated.

Additionally, it might not be desired to get C++ code but rather stay with plain C. The solution was to replace assignments having arithmetic operations on the right side in generated code by method calls to functions that have to be implemented in a runtime library. Those functions would have to be provided for all possible operand types and used accordingly to the corresponding assembler instruction (there would be a 1:1 mapping between assembler instructions and library routines).

The amount of work for the proposed changes is manageable. The current implementation is already situated in a runtime library. We would need to implement wrapper functions in plain C-style that use the current implementation in the background. We assume for now that there are no objections against using a C++-library in the background. Adaption of code generation was performed straightforward, too. Only local changes of a very limited dimension of code were necessary.

Replacement of operator overloads has been discussed but is not scheduled at the moment. It is just an option that should be kept in mind in case it is favored by customers.

Method Calls to hide generated Code Constructs

At the moment, some assembler instructions are not mapped as one single corresponding C(++)-instruction. As an example, this is the case for *MVO* (move with offset): it requires the generation of a loop. Instructions requiring more than one single statement to be translated could be implemented in a runtime library and referenced by the generated code. The generated code would then be less redundant and thus more readable.

This change is strongly recommended as soon as readability and maintainability is of importance – especially, as it should be without difficulty to implement.

5.1.2 Improvements for Control- and Dataflow-Analyses

The implementation has always been constructed to be safe, i.e. identified value sets might contain more values than there can be at runtime but not the other way round. If we find too many possible values, we might also find to many branch targets or program paths and the generation might fail. On the other hand, if we omit possible values, we

might also omit program paths and the generation might not fail while it better should. This way, generated code might run but skip code parts as they are considered to be *dead* by the control flow analyses.

Of course, generation should not fail at all. There are some rare cases that have not been implemented properly. Dataflow analysis is mainly used for determining the control flow, thus it is mostly useful for operations handling addresses. Usually, string operations are not used to transfer a code address and therefore value sets assigned by such operations are not calculated at all but just set to be *undefined*.

Adresses might also be part of larger memory blocks that are transferred at once by operations like *MVC*. In case overlapping memory regions are detected, both are set to be *undefined*. A more fine-grained solution was possible here, possible code addresses part of one region would not be overwritten and therefore could be preserved.

In general, there is only one value set for each instruction node. Figure 5.1 gives a small sample where this approach might not be sufficient.

The sample code starts with a comparison instruction followed by a related conditional branch. In case the condition is met, instruction *LA* in line 3 is skipped. Otherwise, the address of label *FOO* is loaded to R12. Line 5 (LBLA) represents some arbitrary code not relevant for this sample except that it will not modify the contents of R12.

The same combination of comparison/branch is used again to conditionally skip a line of code (line 8, BR). If the instruction loading the address has been skipped, the branch instruction will also be skipped as their execution is depending on the same condition. This situation is not handled by the analyses so far. Control flow graphs are global and exist in only one version: it is not possible to define different paths in the control flow depending on various conditions.

```
1          CR       R1,R2
2          BC       <COND>,LBLA
3          LA       R12,FOO
4  * instructions at LBLA will not modify R12
5  LBLA    [...]
6          CR       R1,R2
7          BC       <COND>,LBLB
8          BR       R12
9  LBLB    [...]
10
11 FOO     [...]
```

Figure 5.1: Limitations of control flow analysis

It is still too early to discuss the impact of this limitation. Nevertheless, there have been observations that the problem discussed appears multiple times. It will be necessary to discuss whether different paths in the control flow analysis should be supported (i.e. whether it is necessary to support it and the amount of time required to implement this

improved technique pays off). The above sample indicates that the generation will fail in case it does not recognize that the branch will only be executed when the address has been loaded before. Potentially, the register is used for handling code addresses in general and does not contain anything other than program addresses during the entire program run. This way, the generation would not fail although it is not clear that the branch will only target this single address. Of course, these assumptions are highly speculative and rely on the code style of the applications.

5.1.3 Removal of Goto-Statements

The implemented algorithm is beneficial and the reduction of unstructured branches a significant advance towards a maintainable code base. Nevertheless, there are still remaining *goto*s and options for improvements.

At the moment, *goto*s are only replaced when the control flow is structured. Erosa et al.[EH94] present an algorithm for eliminating *goto* statements that eliminates all of them.

First, both *break* and *continue* statements are still allowed. The translation of more complex control flow graphs requires transformation processes, that move the *goto* instruction inside the graph. Therefore, additional flag variables have to be generated that forward control flow information between different levels of the scope.

Algorithm 6 gives an example for a *goto* statement leaving two loops and skipping some additional code afterwards. The *goto* statement itself is replaced by an assignment to the flag variable *flag_foo* which indicates whether the referred label should be accessed or code should continue normal operation. In case the label should be targeted, every underlying scope has to query the flag variable and handle it appropriately, i.e. forward it by breaking out of the current scope or transferring control to the right address if the label is on the same scope.

A complete removal of all *goto* statements is reachable using the described algorithms. Whether this is a goal that has to be strived for, still has to be discussed. The introduction of flag variables, that have to be queried along different levels, is a drawback that cannot be neglected.

5.1.4 Type-inference Techniques

Alan Mycroft presents *Type-Based Decompilation*[Myc99], an approach to infer a field's or register's type by analyzing its usage.

Starting basis as described is the compiled program which is first translated to SSA form. Doing so, live ranges for assigned values can be found and appropriate type information for the variables in SSA-form can be deduced. This is accomplished by defining type constraints to each assigned SSA variable. Assignment chains are then evaluated to determine the final types (single assignments could lead to ambiguous results, e.g. assigning 0 to a field is equally legitimate for pointers and integer values, further usage could resolve that issue).

Algorithm 6 Replacing gotos

```
1:  for i ← 1 → 10 do
2:      for j ← 1 → 5 do
3:          ...
4:          if some condition then
5:              flag_foo ← true
6:              break
7:          end if
8:          ...
9:      end for
10:     if flag_foo then
11:         break
12:     end if
13:     ...
14: end for
15: if ¬flag_foo then
16:     [...]
17: end if
18: // foo:
19: ...
```

Even the generation of arrays and structures is discussed. Loops processing linked lists are detected and appropriate structures can be generated to depicture the linked list that is implicitly used by the type of loop implementation in assembler code.

The fact that HLASM allows typed field definitions saves already a significant part of the work. If memory is not addressed by raw addresses but the code makes use of such field definitions, types of fields are available already. The algorithms presented are still valuable for register accesses. Splitting the live ranges of registers to multiple variables that are even typed correctly would significantly increase program readability. Using local variables instead of global registers enables application programmers to ascertain the impact of local changes and overlook the global impact.

5.1.5 Handling of Self-Modifying Code

At the moment, self-modifying code is not handled at all. In case an instruction is writing to a memory location storing executable instructions, code translation just fails.

In theory, it is possible to write whole programs dynamically at runtime without any restrictions compared to hand-written assembler code that has to be compiled. From the translator's view, we would have to dynamically interpret, analyze and translate the code whenever it changes. That means, we cannot only translate code statically and ship the generated code but have to incorporate an *interpreter* to be shipped with the application. Whenever code is generated that obsoletes the generated version, dynamic retranslation is needed. From today's perspective, this functionality is not projected. But, there are

some cases that might be possible to be solved by less substantial means that shall be dicussed in the following.

As an example, let us have a look on the code provided by figure 5.2. To understand

```
1 AAA     BC       0,AUFRUF
2         [...]
3         MVI      AAA+1,X'F0'
4         [...]
```

Figure 5.2: Sample: self-modifying code

the intention of the move operation, we have to explain the binary representation of the branch instruction in memory. The instruction is four bytes long and encoded using the following scheme: *47MXBDDD*. The first byte, which is always set to the hexadecimal value '47' identifies the instruction as branch. The third half-byte, 'M' defines the mask for querying condition codes. The last five halfbytes address the target using indirect addressing (index register, base register, displacement immediate value).

The move instruction changes the mask of the branch instruction (we ignore the fact that it modifies the index used for branch target calculation, too[1]). In case the values '0' and 'F' are written as mask, it is possible to toggle between a *NOP* (no operation) and an unconditional branch.

Generation of the provided code would not require much more than a flag variable in C++. The move instruction would than be translated to modify its value and the branch instruction to query it before branching.

First analyzes have shown that such idioms are covering by far most of the cases of self-modifying code. Hence, limited support for self-modifying code seems reasonable and is projected for the near future.

5.1.6 Macro Processing

Still, macro processing is out of the scope of this work. Nevertheless, it should be mentioned as possible future extension.

HLASM macros are written in a meta-language that incorporates assembler code. Macro processing has to take place before parsing the assembler code using the present parser.

It is not only possible to define simple code substitutions but also to define small pieces of even structured code generating assembler code. Therefore, it is not sufficient to simply replace pieces of code but necessary to *interpret* the macro code. Figure 5.3 gives a small sample of macro source including conditional code production. A macro is

[1] Of course, we have to check automatically whether the branch target changes. Regarding this sample, we assume that the value will not be altered. In general, this pattern is only used to modify the mask of the branch instruction, not anything else. But this is nothing more than a convention and therefore cannot be expected without being checked.

```
 1            MACRO
 2            MACRO1    &PARAM
 3            LCLA      &CNT
 4  &CNT      SETA      0
 5  .LOOP     A         R5,&PARAM
 6  &CNT      SETA      &CNT+1
 7            AIF       ('&CNT' NE '9').LOOP
 8            MEND
 9            [...]
10            MACRO1    10(R1)
```

Figure 5.3: Sample HLASM macro code

introduced by the *MACRO* keyword, followed by a line containing its name (*MACRO1*) and parameters({*PARAM*}). *LCLA* introduces a **local a**dress macro variable - the address is just a binary value and can be used for whatever purpose. The variable defined in line 3 is then initialized to 0. Labels are defined similar as in plain assembler code - but having a dot at the beginning. Code starting at the macro label *.LOOP* adds an address, passed as parameter *PARAM* to the macro, to register *R5*. Thereafter, the loop counter variable *CNT* is incremented by 1. *AIF* tests a macro condition and conditionally jumps to the named label. In this case, a jump to *.LOOP* is executed until *CNT* reaches the value 9. *MEND* marks the end of the macro. Line 10 shows the usage of the macro.

It has to be mentioned that this code is executed at compile-time by an *interpreter* that has to be part of the compiler. This simple code forms a loop that is repeatedly executed eight times by the compiler - each time it produces one single line of assembler code – the *ADD*-instruction of line 5. The assembler parser will later receive the product of the macro parser, which will include those eight *ADD*-instructions instead of a call to the macro.

5.1.7 Supervisor-Calls

SVC-instructions allow to access system functions like input/output operations or hardware timers, the clock, etc. Different supervisor calls are identified by an 8-bit number which is passed as immediate parameter to the instruction.

The realization is depending on the target architecture and operating system. From today's perspective, supervisor calls will be implemented as library functions using system facilities. It has not been evaluated yet, whether all system calls are needed or can be presented on any arbitrary target operating system.

Implementation is not accurately scheduled so far, but it is expected that frequently used operations (e.g. for IO) are provided in the near future.

5.1.8 Pattern Recognition

Recurring patterns (i.e. common sequences of instructions) are often implemented using macros that allow to copy a piece of source code wherever needed. In some situations, there might be recurring sequences of code in the program code itself, that are not swapped out to library functions or macros. Such code patterns could be described in a machine-readable form to be detected when translating source code. It then was possible to generate manually prepared code for a specific pattern that is assumed to be more readable and maintainable as normal generated code which ignores its meaning.

At the moment, there are no plans to implement pattern recognition. Defining patterns requires manual work and there is no evidence yet that this approach will lead to better results. This topic will not be addressed before the implementation has reached a very advanced state.

APPENDIX A

Sample Program Conversions

A.1 Arithmetic Operations

Figure A.1: Sample arithmetic operations: ASM code

```
1  * sample arithmetic operations
2
3  SAMPLE  CSECT
4          USING   SAMPLE,R13
5
6          LM      R1,R3,10(R6)
7          LM      R10,R2,DEC1
8
9          LPR R1,R2
10         LNR R1,R2
11
12         ST      R1,BIN1
13         ST      R1,DEC1
14         ST      R1,5(10,R2)
15         STC     R1,BIN1
16         STC     R1,5(10,R2)
17         STM     R1,R3,10(R5,R6)
18         STM     R10,R2,DEC1
19
20         AR      R1,R2
21         A       R1,BIN1
22         A       R1,DEC1
23         A       R1,10(R2)
24         AH      R1,BIN1
```

```
25          AH      R1,BIN2
26          AH      R1,10(R2)
27          AP      DEC1,DEC2
28          AP      DEC1,10(5,R2)
29          AP      5(8,R2),DEC2
30          ZAP     5(8,R2),DEC2
31
32
33 DEC1     DC   P'0'
34 DEC2     DC   P'100'
35 BIN1     DC   F'100'
36 BIN2     DC   H'10'
37 BIN3     DC   F'42'
```

Figure A.2: Sample arithmetic operations: generated header file

```
1  #ifndef __sample_arith_H
2  #define __sample_arith_H
3  #pragma pack(push)
4  #pragma pack(1)
5
6  #include "hlasm_types.h"
7  #include <cstdint>
8
9  unsigned char asm_data[] = {0x0C, 0x10, 0x0C, 0x64, 0x00,
     0x00, 0x00, 0x0A, 0x00, 0x2A, 0x00, 0x00, 0x00};
10
11 ptrdiff_t R[16];
12
13 hlasm::fixed_decimal<1> *DEC1 = (hlasm::fixed_decimal<1>*)
     ((char*) asm_data + 0);
14 hlasm::fixed_decimal<3> *DEC2 = (hlasm::fixed_decimal<3>*)
     ((char*) asm_data + 1);
15 int32_t *BIN1 = (int32_t*) ((char*) asm_data + 3);
16 int16_t *BIN2 = (int16_t*) ((char*) asm_data + 7);
17 int32_t *BIN3 = (int32_t*) ((char*) asm_data + 9);
18 #pragma pack(pop)
19 #endif /* __sample_arith_H */
```

Figure A.3: Sample arithmetic operations: generated code file

```
1  #include "sample_arith.asm.h"
2
3  int cc = 0;
```

```
4
5 int main() {
6     memcpy(R + sizeof(*R) * 0, (void*) ((ptrdiff_t) (10) +
        (ptrdiff_t) (R[5])), sizeof(*R) * 3);
7     memcpy(R + sizeof(*R) * 9, (void*) (DEC1), sizeof(*R) * 7);
8     memcpy(R, (void*) ((ptrdiff_t) ((DEC1)) + sizeof(*R) * 7),
        sizeof(*R) * 2);
9     R[0] = abs(R[1]);
10    R[0] = - abs(R[1]);
11
12    *(BIN1) = R[0];
13    *((int32_t*) (DEC1)) = R[0];
14    *((int32_t*) ((ptrdiff_t) (5) + (ptrdiff_t) ((ptrdiff_t)
        (R[9]) + (ptrdiff_t) (
15          R[1])))) = R[0];
16    R[0] = *((int8_t*) (BIN1));
17    R[0] = *((int8_t*) ((ptrdiff_t) (5) + (ptrdiff_t)
        ((ptrdiff_t) (R[9]) + (ptrdiff_t) (
18          R[1]))));
19    memcpy((void*) ((ptrdiff_t) (10) + (ptrdiff_t)
        ((ptrdiff_t) (R[4]) + (ptrdiff_t) (
20          R[5]))), R + sizeof(*R) * 0, sizeof(*R) * 3);
21    memcpy((void*) (DEC1), R + sizeof(*R) * 9, sizeof(*R) * 7);
22    memcpy((void*) ((ptrdiff_t) ((DEC1)) + sizeof(*R) * 7), R,
        sizeof(*R) * 2);
23
24    R[0] += R[1];
25    R[0] += *(BIN1);
26    R[0] += *((int32_t*) (DEC1));
27    R[0] += *((int32_t*) ((ptrdiff_t) (10) + (ptrdiff_t)
        (R[1])));
28    R[0] += *((int16_t*) (BIN1));
29    R[0] += *(BIN2);
30    R[0] += *((int16_t*) ((ptrdiff_t) (10) + (ptrdiff_t)
        (R[1])));
31    *(DEC1) += *(DEC2);
32    *(DEC1) += *((fixed_decimal<(5 + 1) / 2>*) ((ptrdiff_t)
        (10) + (ptrdiff_t) (R[1]
33          )));
34    *((fixed_decimal<(8 + 1) / 2>*) ((ptrdiff_t) (5) +
        (ptrdiff_t) (R[1]))) += *(
35          DEC2);
36    *((fixed_decimal<(8 + 1) / 2>*) ((ptrdiff_t) (5) +
```

```
        (ptrdiff_t) (R[1]))) = 0;
37      *((fixed_decimal<(8 + 1) / 2>*) ((ptrdiff_t) (5) +
        (ptrdiff_t) (R[1]))) += *(
38          DEC2);
39  }
```

A.2 Restructuring

Figure A.4: Restructuring: ASM code

```
1   SAMPLE   CSECT
2            USING    SAMPLE,R13
3
4   * test case 0: simple if
5            SR       R1,R1
6            CR       R1,R2
7            BE       TC0_TRU
8            A        R15,DUMMY
9   TC0_TRU  S        R15,DUMMY
10
11  * test case 1: if/else
12           SR       R1,R1
13           CR       R1,R2
14           BE       TC1_TRU
15           A        R15,DUMMY
16           B        TC1_END
17  TC1_TRU  S        R15,DUMMY
18  TC1_END  SR       R15,R15
19
20  * test case 10: pre-conditional loop
21           SR       R1,R1
22  TC10_H   CR       R1,R2
23           BNL      TC10_A
24           LA       R1,1(R1)
25           B        TC10_H
26  TC10_A   SR       R15,R15
27
28  * test case 11: post-conditional loop
29           SR       R1,R1
30  TC11_H   A        R15,DUMMY
31           LA       R1,1(R1)
32           CR       R1,R2
```

```
33          BL      TC11_H
34          SR      R15,R15
35
36  * test case 12: post-conditional with branch at header
37          SR      R1,R1
38  TC12_H  CR      R1,R1
39          BE      TC12_A
40          A       R1,10(R15)
41  TC12_A  A       R1,14(R15)
42          CR      R1,R5
43          BL      TC12_H
44          SR      R15,R15
45
46          A       R15,DUMMY
47  FOO     DS      0H
48          A       R15,DUMMY
49          CR      R15,R1
50          BL      FOO
51
52  * test case 30: function call
53          B       TC30
54
55  FUNCA   AR      R5,R6
56          BR      R13
57
58  TC30    BAL     R13,FUNCA
59          SR      R15,R15
60
61  PGM_END A       R15,DUMMY
62
63  DUMMY   DC    F'100' * used as operand for dummy instructions
```

Figure A.5: Restructuring: generated header file

```
1  #ifndef __sample_asm_H
2  #define __sample_asm_H
3  #pragma pack(push)
4  #pragma pack(1)
5
6  #include "hlasm_types.h"
7  #include <cstdint>
8
9  unsigned char asm_data[] = {0x64, 0x00, 0x00, 0x00};
10
```

```
11 ptrdiff_t R[16];
12
13 int32_t *DUMMY = (int32_t*) ((char*) asm_data + 0);
14 #pragma pack(pop)
15 #endif /* __sample_asm_H */
```

Figure A.6: Restructuring: generated code file

```
1  #include "sample_asm.asm.h"
2
3  int cc = 0;
4
5  int main() {
6      R[0] = 0;
7      if (R[0] != R[1]) {
8          R[14] += *(DUMMY);
9      }
10 // TC0_TRU
11     R[14] -= *(DUMMY);
12
13 /* test case 1: if/else */
14     R[0] = 0;
15     if (R[0] == R[1]) {
16 // TC1_TRU
17         R[14] -= *(DUMMY);
18     } else {
19         R[14] += *(DUMMY);
20     }
21 // TC1_END
22     R[14] = 0;
23
24 /* test case 10: pre-conditional loop */
25     R[0] = 0;
26     while (R[0] < R[1]) {
27         R[0] = (ptrdiff_t) (ptrdiff_t) (1) + (ptrdiff_t)
             (R[0]);
28     }
29 // TC10_A
30     R[14] = 0;
31
32 /* test case 11: post-conditional loop */
33     R[0] = 0;
34     do {
```

```
35          R[0] = (ptrdiff_t) (ptrdiff_t) (1) + (ptrdiff_t)
              (R[0]);
36      } while (R[0] < R[1]);
37      R[14] = 0;
38
39  /* test case 12: post-conditional with branch at header */
40      R[0] = 0;
41      do {
42          if (R[0] != R[0]) {
43              R[0] += *((int32_t*) ((ptrdiff_t) (10) +
                  (ptrdiff_t) (R[14])));
44          }
45  // TC12_A
46          R[0] += *((int32_t*) ((ptrdiff_t) (14) + (ptrdiff_t)
              (R[14])));
47      } while (R[0] < R[4]);
48      R[14] = 0;
49
50      R[14] += *(DUMMY);
51      do {
52      } while (R[14] < R[0]);
53      goto TC30;
54  TC30:
55      R[13] = (ptrdiff_t) (&&FUNCA);
56      FUNCA();
57      R[14] = 0;
58
59  // PGM_END
60      R[14] += *(DUMMY);
61
62  }
63
64  int FUNCA() {
65  // FUNCA
66      R[4] += R[5];
67  }
```

Bibliography

[AP02] Andrew W Appel and Jens Palsberg. *Modern compiler implementation in Java.* Cambridge, 2002.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[BHK13] Florian Brandner, Nigel Horspool, and Andreas Krall. DSP instruction set simulation. In *Handbook of Signal Processing Systems*, pages 945–974. Springer, 2013.

[BJSW13] David Brumley, Ivan Jager, Edward J Schwartz, and Spencer Whitman. The BAP handbook, 2013.

[BML+13] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Hao Lin, Chirag Dave, Rudolf Eigenmann, Samuel P Midkiff, et al. The cetus source-to-source compiler infrastructure: overview and evaluation. *International Journal of Parallel Programming*, 41(6):753–767, 2013.

[BR04] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 5–23. Springer, 2004.

[BR10] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):23, 2010.

[CG95] Cristina Cifuentes and K John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[Cif93] Cristina Cifuentes. A structuring algorithm for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana de Informatica*, pages 267–276. Citeseer, 1993.

[Cif94] Cristina Cifuentes. *Reverse compilation techniques.* PhD thesis, Queensland University of Technology, 1994.

[Cif96] Cristina Cifuentes. Structuring decompiled graphs. In *Compiler Construction*, pages 91–105. Springer, 1996.

[EH94] Ana M Erosa and Laurie J Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Computer Languages, 1994., Proceedings of the 1994 International Conference on*, pages 229–240. IEEE, 1994.

[FF99] Yishai A Feldman and Doron A Friedman. Portability by automatic translation: A large-scale case study. *Artificial Intelligence*, 107(1):1–28, 1999.

[FKH07] Stefan Farfeleder, Andreas Krall, and Nigel Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *Journal of Systems Architecture*, 53(8):501–510, 2007.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[her] hercules-390.eu. The Hercules System/370, ESA/390 and z/Architecture Emulator. http://www.hercules-390.eu. Last retrieved on 12/03/2014.

[IBM03] IBM. *Enterprise System Architecture/390 Principles of Operation*. IBM, 9 edition, june 2003.

[IBM08] IBM. *High Level Assembler for z/OS & z/VM & z/VSE Language Reference*, 6 edition, 7 2008.

[LSUA06] Monica Lam, Ravi Sethi, JD Ullman, and AV Aho. *Compilers: Principles, Techniques and Tools*. Pearson Education, 2nd edition, 2006.

[Mar10] Simon Marsden. Relogix: Converting assembler to high quality C, 9 2010.

[May87] Cathy May. *Mimic: a fast system/370 simulator*, volume 22. ACM, 1987.

[Mic09] MicroAPL. An Introduction to the Relogix Assembler to C Translator, 6 2009.

[Myc99] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Programming Languages and Systems*, pages 208–223. Springer, 1999.

[PKS02] Mark Probst, Andreas Krall, and Bernhard Scholz. Register liveness analysis for optimizing dynamic binary translation. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 35–44. IEEE, 2002.

[PQ95] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[Pry07] Ivan Pryanishnikov. *Static Program Analyses and Code Transformations for DSP Software*. PhD thesis, Vienna University of Technology, Austria, 2007.

[SAIC+99] Timothy J Slegel, Robert M Averill III, Mark A Check, Bruce C Giamei, Barry W Krumm, Christopher A Krygowski, Wen H Li, John S Liptay, John D MacDougall, Thomas J McPherson, et al. Ibm's s/390 g5 microprocessor design. *Micro, IEEE*, 19(2):12–23, 1999.

[War99] Martin P Ward. Assembler to c migration using the fermat transformation system. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 67–76. IEEE, 1999.