Diploma Thesis

A Pattern Matcher Generator for Retargetable Code Generation and Optimisation

Gernot Gebhard
19. October 2006

Prof. Dr. Reinhard Wilhelm
Chair for Programming Languages and Compiler Construction

Prof. Dr.-Ing. Philipp Slusallek
Chair for Computer Graphics

Tutor: Dipl.-Inform. Philipp Lucas
Compiler Design Lab

Department of Computer Science
Saarland University
D-66041 Saarbrücken

Acknowledgement

First of all, I wish to thank Prof. Reinhard Wilhelm for giving me the opportunity to conduct this work. Additionally, I thank Prof. Reinhard Wilhelm and Prof. Philipp Slusallek in advance for the survey of this work.

My special thank goes to Philipp Lucas, who provided the topic for this diploma thesis. His ongoing assistance and many fruitful discussions concerning various aspects of this work had a great influence on the style of this document and certain design decisions of the practical part of this work. So, the successful completion of this diploma thesis is to great extend due to him.

Additionally, I thank Nicolas Fritz for many refreshing discussions together with Philipp Lucas and Prof. Philipp Slusallek.

Last but not least, I thank my friends, my whole family and – most important of all – my fiancée Claudia for their never-ending support, without which I would not have been able to complete this work.

Extra credits go to all authors of free software for their excellent work. This diploma thesis has been realised entirely using free software.

About this Document

I have designed and written this paper with OpenOffice. The PDF version of this document features clickable hyperlinks that connect references to their origin. So the reader can easily jump to a footnote, a figure or an example, wherever referenced.

Although both figures and tables are self-explaining most of the time, to prevent misconceptions I advice the reader to always parse them in conjunction with the surrounding text.

Bold page numbers in the index at the end of this paper denote the most relevant page.

Declaration of Originality

I hereby declare on oath that this thesis is my own work and that, to the best of my knowledge, it contains no material previously published, or substantially overlapping with material submitted for the award of any other degree at any institution, except where due acknowledgement is made in the text.

Sankt Ingbert, 19. October 2006.

Table of Contents

Abstract

Depending on its internal structure, adding support for an all-new target platform or introducing a new kind of optimisation to a compiler that already supports multiple architectures can become a complex and confusing affair. Especially when new architectures with different features are released at a very fast rate, as it is currently experienced in the area of graphics processing units, this becomes even more problematic. Thus, modifying the compiler turns out to be error-prone and time-consuming. Employing generated rule-based pattern matchers, which are capable of generating and optimising code for different target architectures at once, will help to overcome these problems. Instead of having to cope with low-level code, a developer can finally attend to the essential thing, increasing his productivity, by specifying rules that describe how certain code patterns should be treated.

Exemplified on cgisc, a compiler for the GPU programming language CGiS, the present work demonstrates the advantages and disadvantages of using these generated pattern matchers in the compilation process.

I. Introduction

1. History of Computation

Tools have been a constant companion of the human civilisation. The ability to create and use complex tools definitely ensured survival of mankind since the beginning of history, and thus there exists almost no aspect of human live where tools are not being employed. Hence, it is not very surprising that the applied mathematics has benefited much from the invention of certain computational devices.

The first available device was either a Chinese or Babylonian invention of the antiquity and is nowadays known as the *abacus*. The abacus is a board with free-moving beads that are aligned in rows of bars or chinks. The value of a bead depends on its position within these rows, as Figure I.1 shows. In addition to the four fundamental arithmetic operations, the abacus can also be used to compute square and even cubic root [1].

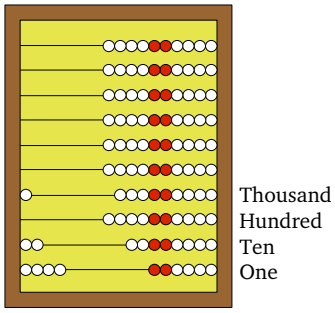

Figure I.1: Russian abacus showing the number 1024.

In 1623, Wilhelm Schickard [2] invented the first mechanical calculator, the *calculating clock*, which supported addition and subtraction of six-digit numbers. Multiplication, division and square roots were computable using a set of Napier's rods (or bones), which are a special kind of abacus, that was mounted on the machine. Other mechanical calculators followed, as e.g., the *Pascaline* in 1645, named after its inventor Blaise Pascal, and the *Stepped Reckoner* that was introduced by Gottfried Wilhelm Leibniz in 1671 developing Pascal's ideas. Like all other mechanical devices that have been constructed thereafter, none of them could be programmed, because all supported operations were part of their design. So to speak, all these machines had fixed programs. Reprogramming such a machine thus meant either restructuring, rewiring or redesigning, which is obviously not reasonable for an arbitrary computation.

Publishing the design of a mechanical general-purpose computer, the *analytical engine* [3], in 1831, Charles Babbage took an important step in the history of computers. The idea came up during the development of the *differential engine*, which is a mechanical computing device for tabulating logarithms and trigonometric functions by evaluating approximating polynomials, as he realised that a much more sophisticated design was possible. Although the design was ahead

of the times by at least one hundred years, the machine was actually never built due to financial, legal and political reasons. Until the midst of the 20th century, there was no comparable approach.

Between 1934 and 1938 Konrad Zuse has developed the Z1, the first mechanical program-controlled computer. Unfortunately, the machine was working unreliable due to mechanical problems that rendered the machine unusable in practice. Unsatisfied with the first attempt, Zuse finally managed to create the Z3 using telephone relays in 1941 [5]. It was the first fully functional, programmable computer, on which an arbitrary computation could be implemented[1]. Not knowing Zuse's invention, John von Neumann published the *von Neumann architecture*[2] three years after the construction of the Z3. The first working *von Neumann machine* was the Manchester Small-Scale Experimental Machine (SSEM) (also nicknamed *Baby*) that was built at the University of Manchester in 1948 [6]. In the years to come, several other implementations of this architecture followed, which quickly made it the most widely spread architecture of the world. Thus, it is unsurprising that the von Neumann architecture was taken as a basis for almost every modern hardware architecture, such as e.g., IA-32, IA-64, AMD64 or RISC.

The computers of the first generation were very energy-hungry machines that were built with relays or vacuum tubes. A power consumption of at least ten kilowatts was not uncommon at these times. In the late 1950s transistorised computers, normally referred as computers of the second generation, replaced the bulky machines of the first generation substituting relays and vacuum tubes with the much smaller and more reliable transistor, which has been invented in 1947. However, these computers were still very expensive and were thus mainly used by governments, universities or large companies. The invention of the integrated circuit, independently achieved by Jack St. Clair Kilby and Robert Noyce in 1958, started the miniaturisation process resulting in a dramatically increasing production of the even smaller computers of the third generation. All computers of the fourth generation, to which today's machines belong, feature the microprocessor that has been developed at Intel in 1971. Since then, the computing power has been steadily increasing following Moore's Law[3] accompanied by the ongoing miniaturisation process, which resulted in the highly efficient computers of the modernity.

In 1982 the Japanese Ministry of International Trade and Industry started an ambitious project to realise a *fifth generation computer* that should benefit from massive parallelism. Although a working *Parallel Inference Machine* (PIM) has been constructed in 1991, the whole project was stopped two years afterwards [8]. The PIM never met with commercial success, and the proprietary architecture was ultimately outstripped in computation speed by less specialised hardware, such as e.g., by Intel's x86 machines.

1 The Z3 was proven to be Turing-complete in 1998 [4].

2 Von Neumann's approach was not very different from Zuse's computer. The main difference was that von Neumann's design does not separate instructions and data from each other. Computers following this design are also called stored-program computers.

3 Gordon Moore predicted in 1965 that the amount of transistors on a chip, which is a rough indicator for its computing power, would double every two years [7].

2. Programming Languages

The very first computers featured a small set of machine instructions, which modified the memory or performed simple arithmetic operations. Although machine code is sufficient to implement an arbitrary algorithm, writing a program on this lowest level of hardware abstraction was – and still is – a fault-prone and time-intensive matter. To speed up the program development process, another way of telling a machine what it has to do had to be found.

The human language unfortunately seemed unsuitable for this procedure, as there was – and still is – no reasonable way to translate it into machine code[4]. Thus, there was need of an intermediate language that can be easily translated into a sequence of machine instructions and that provides a quickly-understandable abstraction of the underlying hardware. Naturally, a program written in such a language must first be converted into machine code, before the computer is able to execute the program. The translation is usually done by an auxiliary program called *compiler*, which Section 3 briefly introduces and Chapter II discusses in more detail.

The history of programming languages can be dated back to Charles Babbage and Ada Lovelace, who thought of some way to instruct the analytical engine. Because – as mentioned above – this engine was never built, this was only theoretical work. The Plankalkül is believed to be the world's first (non-von Neumann[5]) high-level programming language that Konrad Zuse has been designed. Although Zuse published his design lately in 1972 [9], it is generally accepted that Zuse has invented the Plankalkül between 1942 and 1945 [10]. Albeit the Plankalkül never awoke from its sleeping beauty slumber, as Zuse liked to say, a compiler for the Plankalkül has been eventually implemented in 2000 [11]; five years after his death. Between 1954 and 1957 an IBM team led by John Warner Backus – the co-inventor of the Backus-Naur form – introduced FORTRAN and the first compiler that has ever been available. Numerous other programming languages followed, resulting in more than a thousand known languages, most of which are domain-specific (tailored to a certain problem). Nowadays, about twenty of them are widely-used, such as e.g., C, Java, Perl or Python.

Depending on their hardware abstraction level, the known languages are classified as either low-level (assembler) or high-level, hardware-independent programming languages. The latter of them are further divided into five classes:

- In an *imperative or procedural language*, an algorithm is implemented by specifying *how* the computation has to take place. An imperative program is thus a sequence of operations processing the available data. Being close to the hardware, imperative programming languages are commonly used to program hardware (e.g., the Linux kernel that is implemented in C). However, as these language provide a rather low abstraction of the underlying hardware, developers of complex software systems nowadays make use of more sophisticated programming languages. Well-known imperative programming languages are Ada (named after Ada Lovelace), BASIC, C, COBOL or FORTRAN.
- Instead of specifying how to compute the result, *declarative languages* are designed for writing programs that describe *what* is to be computed. There are different approaches realising this programming paradigm. On the one hand, functional programming languages such as LISP or Standard ML aim at expressing algorithms as close as possible in

4 It is unclear whether this would remedy the complexity of writing computer programs, as stated in SIGPLAN notices, Vol. 2, No. 2: “Make it possible for programmers to write programs in English, and you will find that programmers cannot write in English.”

5 Every language abstracting the von Neumann hardware architecture is called a von Neumann programming language. Abstracting a different kind of hardware, the Plankalkül does obviously not belong to this class of programming languages.

form of mathematical functions. On the other hand, logic programming languages make use of mathematical logic. The most famous programming language of this kind is Prolog. Declarative programming languages are very popular in applied mathematics, but are seldom used to implement non-mathematical applications.

- *Distributed or parallel programming languages* are employed, if a program should be written for a parallel or distributed architecture. Specific synchronisation problems arise out of this kind of programming that can be handled using special language constructs. To satisfy the needs, new languages (occam or Parallaxis) have been designed, and even existing languages, such as e.g., FORTRAN or Pascal, have been extended by appropriate language constructs. Many of the parallel programming languages have been developed as research languages rather than as languages for production use. However, with the increasing number of processing cores in modern computers, these languages might become more and more interesting for software developers.
- *Object-oriented languages* comprehend each important information as some kind of object that is described through the data it stores (member variables) and the operations it may perform (member functions). These languages have revolutionised the art of computer programming, as they provide a different kind of abstraction of the ongoing processes. Instead of having functions calling each other, an algorithm is implemented by having several objects interact among one another. There exist pure object-oriented languages such as Smalltalk, Eiffel or Java as well as hybrid languages, such as e.g., C^{++} or Object Pascal, which were extended by object-oriented language constructs.
- *Aspect-oriented languages* try to assist the developer by pushing the level of abstraction even higher [12]. A language that belongs to this class is in general not a stand-alone language. This means that aspect-oriented languages are not used to implement a whole algorithm. Instead, these meta-programming languages are designed to aid organising the structure of an algorithm. The idea is based on the observation that whenever functional units with different internal structure (e.g., two different functions) share some kind of behaviour, code is very often duplicated. In the long run, this does not tend to be maintainable at all. To keep up the maintainability, these shared behaviours – called *aspects* in this context – are to be expressed in an appropriate aspect-oriented language. Although sounding revolutionary, these meta-languages are not well-established. However, there exists an aspect-oriented extension for Java, named aspectj.

It should be mentioned that the above language concepts are not mutually exclusive. As object-oriented language features are completely orthogonal to imperative language constructs, a language may combine both aspects. Thus, Java can be understood as an object-oriented as well as an imperative language. Thus, programming languages cannot be strictly divided into disjoint classes, which however does not mean that the above classification is wrong. It can still act as a guideline when trying to identify certain key features.

3. Compilers and Retargetable Pattern Matchers

As indicated above, compiling a program – translating it into machine code – is necessary before it can be executed; regardless of the employed programming language. Before the code generation takes place, the compiler first analyses the program's syntactical and semantic correctness and usually creates an abstract internal representation of the program. This is done by the compiler's *front end*. If an error has occurred, the compiler stops processing, showing the most reasonable cause. Otherwise, the compiler finishes with code generation and optimisation, which constitute the compiler's *back end*, by converting the internal program representation into a sequence of machine instructions for the target architecture.

On a first look, this method sounds so generic that it appears feasible to implement an universal compiler capable of translating any source language for an arbitrary target architecture. However, besides the knowledge about the different programming languages, such a compiler must also have certain information about the target platform. Although nearly every widely-spread architecture, such as e.g., IA-32, IA-64, AMD64, Sparc, ARM, RISC or MIPS, is an instance of the von Neumann architecture, they all differ in certain key features, such as the amount of available registers and their types or the available instruction set. Disappointing as it is, no working universal compiler has ever been implemented up to the present time. Thus, state-of-the-art compilers that support n languages and m architectures consequently feature n different front ends and m different back ends.

To minimise the complexity of a compiler, the idea came up to generate portable byte code instead of native machine code, which would make only one back end necessary. Rather than writing multiple back ends, a byte code interpreter or *virtual machine* would have to be implemented on each desired platform. In spite of a probable performance decrease[6] when interpreting code instead of executing it, this idea has become quite popular and has already been realised for several languages, such as Java or .NET.

However, not every hardware architecture is suitable for implementing virtual machines, as, for instance, modern graphics processing units (GPUs). They are an interesting target for general purpose programming, because their highly parallel architecture enables them to perform certain computations much faster than most recent processors [13]. Because there are many differences between CPU and GPU on the architectural level, as there are e.g., no integer data types available, exploiting these computational resources only can be done by introducing new language constructs, which finally results in new languages that provide some kind of abstraction of the underlying GPU architecture. As hinted, making use of their computing power cannot be achieved by implementing a virtual machine: this would awfully diminish all their advantages. It is for this reason that a compiler for any of these languages must provide a different back end for each supported architecture. Because new GPU architectures are currently released at a fast rate, this becomes even more problematic.

Exactly at this point, compiler developers would benefit greatly from a pattern matcher generator creating retargetable pattern matchers that replace the previous code generation and optimisation mechanism. Supporting new hardware architectures can then be realised more quickly by adding new rules describing how to handle code patterns for the new target. Additionally, it is no longer necessary to cope with the code generation and optimisation algorithms, because everything is handled automatically by generated pattern matchers. This enables a developer to focus earlier on more important matters, which will finally increase their productivity.

6 This problem no longer exists, because modern virtual machines make use of just-in-time-compilers that translate the byte code into native machine code once, whenever required. After this translation process is done, a performance decrease is no longer measurable.

The work at hand presents the theoretical background and the implementation of a pattern matcher generator that creates retargetable pattern matchers, which can be employed in multiple environments. On the basis of cgisc, a compiler for the programming language CGiS that abstracts recent GPU architectures, I will demonstrate that it is feasible to replace the code generation and optimisation process of a compiler using the generated pattern matchers as a multi-target back end.

This document comprises eight chapters that are based on each other. Chapter II will provide all necessary information about the environment in which the pattern matcher generator is being employed. The complete theoretical background is covered in Chapter III, whereas details about the implementation are discussed in Chapter IV. Discussing the requirements that are necessary to make a compiler work with the generated pattern matchers and demonstrating a code generation and a code optimisation pattern matcher by means of the CGiS compiler, Chapter V is followed by a comparison between the presented approach with related work in Chapter VI. The document finally concludes with an outlook on future extensions to the pattern matcher generator in Chapter VII and with a list of all made achievements presented in Chapter VIII.

II. Background

1. General-Purpose Programming on the GPU

1.1. History

In the 1970s, the first computers capable of drawing images were available, which was a major step towards all-purpose machines in the development of computers. This feature enabled them to be employed in applications in which text output is not sufficient, such as computer games. Because the first computers with this capability were too slow to handle the graphical output in reasonable time, they had to rely on auxiliary hardware, which finally led to the development of the first graphics chips in the late 1970s.

These chips are not be understood as the first available graphics cards, because they had no shape-drawing support and served as some kind of coprocessor[7] to execute certain drawing operations, such as moving bitmaps around, the CPU was too weak for. An important chip of this time was the *blitter*[8], which made a noticeable performance increase possible. The blitter used the Bit BLT[9] algorithm, which David Ingalls invented for the Xerox Alto computer in 1975 [15], to speed up copying and moving of data in memory. Because the blitter was specialised on bitmap-data transfer, the chip could move bitmaps more quickly around than the CPU. The first mass-market computer featuring a blitter was the Amiga that has been introduced by Commodore in 1985.

As the chip manufacturing process improved, it was eventually possible to combine drawing operations and the advantages of the first graphics chips on a single board and later in a single chip, which resulted in the first available graphics cards in the late 1980s. Although not being as flexible as general-purpose coprocessors, this specialised graphics hardware finally surpassed them, such that these coprocessors soon vanished from the market. As 1991 S3 published the first single-chip two-dimensional accelerator named S3 86C911, the graphics chip development entered its hot phase. Alongside with improvements to manufacturing methods came a steady increase in performance and capabilities of graphics cards, which soon enabled more advanced graphic chips to handle video playback in real-time.

A few years after the first single-chip graphics accelerators have been released, 3dfx Interactive began, as its name suggests, to develop three-dimensional graphics controllers. In 1996, 3dfx released the first graphics card that accelerated the rendering of three-dimensional images. Consisting of 1 million transistors and running at core speed of 50 MHz, the Voodoo Graphics was capable of rendering 1 million vertices per second. Only nine years after 3dfx released its card, NVIDIA, which bought up 3dfx in December 2000, introduced the GeForce 7800 GTX that comprised 302 million transistors and rendered about 1.1 billion vertices per second, while running at 550 MHz.

Within less than a decade, the performance of three-dimensional graphics accelerators has been increasing at a much faster rate than that of general-purpose processors [16]. The given numbers show that the amount of transistors on graphics chips has been increasing by a factor of 32 every two years and has thus been 16 times faster than the foretold processor transistor count growth in Moore's Law. Until now, this trend held on, and there is still no evidence indicating a

7 As the development of display processors began, it soon turned out that simple chips did not satisfy the needs of computer graphics [14]. Thus, the first graphics chips were in fact general-purpose processors, which made their construction a costly affair.

8 Blitter is an acronym for **bl**ock **i**mage **t**ransf**er**.

9 Bit BLT abbreviates **bit bl**ock **t**ransfer. It is also known as blitting in computer graphics.

slowdown in this evolution. Instead, the converse seems to happen. There is no other area of computer development that has ever experienced a boom like this.

After general-purpose graphics coprocessors have been abandoned, the chip industry concentrated on specialised hardware, which made the manufacturing process much cheaper. Thus, graphics cards unfortunately offered no way of implementing any other graphical effects besides the built-in ones, such as transform and lighting. But since the progressing manufacturing technology allowed more complex chips that performed more complex operations, this drawback was soon going to be removed. In 1999 NVIDIA introduced the register combiners as an OpenGL 1.2 extension [18], which was the first mechanism that allowed a variety of computations executed on the GPU on a per-pixel level, including signed addition, signed multiplication, and dot product. The NVIDIA GeForce 256 (NV10) was the first graphics cards that supported this extension. A vendor-independent standard was eventually available, as vertex and pixel shaders have been published with DirectX 8 in November 2000. Vertex and pixel shaders are (small) assembly programs that are executed on the graphics chip. As their name suggests, vertex shaders operate on vertices, which enables them to manipulate the geometry of the scene, whereas pixel shaders are employed to compute the textures that are put on the scene's polygons. A corresponding approach was officially introduced in the OpenGL 2.0 core specification that has been released in October 2004 [19], whereas vertex and pixel shaders are called vertex and fragment programs[10]. To keep it simple, I will stick to the terms vertex and pixel shaders for the rest of this document.

This new standard enabled developers to implement graphical effects by writing shaders that are guaranteed to run on any graphics card that supports the standard without the need for any prior modification. NVIDIA's GeForce 3 (NV20) and ATI's Radeon 8500 (R200) were the first graphics cards available that implemented this mechanism (see Table II.1 for supported standards of most recent graphics cards). Due to their programmability, graphics cards that support vertex and pixel shaders are also referred as graphics processing units (GPUs).

Graphics Card	*Chip*	*OpenGL*	*DirectX*	*Pixel Shader*
ATI X1900 XTX	R580	2.0	9.0	3.0
ATI X800 XT PE	R480	2.0	9.0	2.0
ATI 9100 IGP	RC350	2.0	8.1	1.4
NVIDIA 7800 GTX	G70	2.0	9.0	3.0
NVIDIA 6800	NV40	2.0	9.0	3.0
NVIDIA FX 5800	NV30	2.0	9.0	2.0

Table II.1: Supported standards of most recent graphics cards.

It soon turned out that – besides graphical effects – it is possible to use the GPU for general-purpose computations [20], as both vertex and pixel shaders have access to a rich instruction set that includes many vector operations and that just recently supports branching. However, implementing an arbitrary algorithm on the GPU is not as easy as it sounds, because there are many differences between CPUs and GPUs that are to be considered. Besides going into the pe-

10 Although not being part of the OpenGL core specification, the vertex and fragment program extensions were already supported by some vendors after they have been officially approved in September 2002. NVIDIA has introduced its own extensions in 2000 (NV_vertex_program) and 2001 (NV_fragment_program), followed by ATI's extensions that were published in 2001 (EXT_vertex_shader) and 2002 (ATI_fragment_shader). See [17] for their documentation.

culiarities of the GPU architecture, the following section identifies the reasons why the GPU is an interesting target for general-purpose programming.

1.2. Architecture

Modern graphics cards feature 256 to 512 MB DDR3 RAM accessed through a 256 bit wide data path with a bandwidth of more than 50 GB/s. Their cores are running at 550 to 650 MHz driving up to eight vertex and at most 48 pixel shader processors in their computations. As it appears to the host system, exactly one shader of the appropriate type is executed in parallel on these processing units, while operating independently on the input data. With this enormous computing power, modern graphics cards are capable of rendering photo-realistic images in real-time. Figure II.1 demonstrates the conversion of a rather simple three-dimensional scene into a two-dimensional picture. Before the final image can be displayed on the screen, two stages of the rendering process have to be passed. During the rasterisation stage the three-dimensional scene (vertex data) is projected onto a two-dimensional plain, which is a two-dimensional bitmap in this context. This step takes place in the graphics card's rasteriser (as shown in Figure II.2). Afterwards, the pixels of the bitmap are coloured according to light sources, assigned textures and the geometry of the scene. The resulting image is eventually put into the frame buffer, from where it can be displayed on screen.

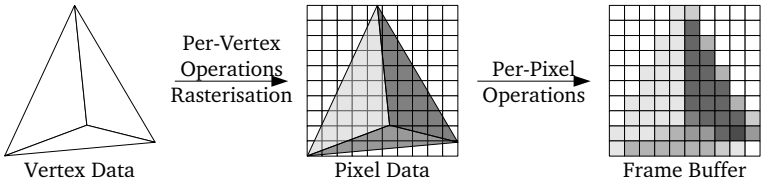

Figure II.1: Abstract view on the GPU rendering process.

The high parallelism enables modern graphics cards to perform certain computations much faster than recent CPUs. Although these vast computational resources are required to render images in real-time, they are primarily lying dormant, as graphics cards deal most of the time with drawing the graphical user interface of an operating system. But, as hinted previously, it is possible to exploit this raw computing power for general-purpose programming by means of shader programs. Because modern graphics cards feature in general much more pixel than vertex processors, vertex shaders are uncommonly used when an algorithm is ported to the GPU. Additionally, vertex processors had until recently no access to the texture memory, which is usually misused to store input data. Without access to the texture memory, it was infeasible to provide a vertex shader with arbitrary data. Thus, pixel shaders are the tool of choice in general-purpose programming on the GPU.

Besides using the texture memory to receive its input data, a shader program disposes of several temporary registers to compute its results. Instead of colouring pixels to draw a nice image, a general-purpose pixel shader misuses them to store the computation results that are finally put into the frame buffer, from where it can be downloaded into the computer's main memory afterwards. As images are rendered into the frame buffer, it also valid to speak of rendering the results. However, there are certain peculiarities of the GPU that have to be considered, when implementing an algorithm.

The first difference to observe is that the GPU architecture is not a von Neumann architecture, because program storage and data storage are – if not physically – logically separated, which makes it impossible to implement self-modifying programs on the GPU. This is furthermore reflected in the fact that memory can either be used for reading or writing, but not for both at the

same time. It is additionally not possible to realise algorithms that strongly depend on integer arithmetic, such as DES [22], because the GPU totally lacks integer data types and operations. In fact, every memory cell, be that one of the texture memory or of the frame buffer, or even a temporary register, is a colour with four components of float type. Moreover, arbitrarily large chunks of data cannot be uploaded into the texture memory, as textures may currently only have a maximum width and height of 4096 pixels. Too large data blocks must be arranged such that they fit into a single texture by making use of other colour components – if this is possible at all – or by dividing them into multiple textures. The latter is unfortunately not always an option, because pixels shaders cannot read from arbitrarily many textures, and some architectures do not support rendering into multiple textures, such as the NVIDIA NV30. Besides the limited amount of accessible textures, pixel shaders must not be arbitrarily long. Thus, a pixel shader containing too many instructions or using too many textures must be split up into a couple of interdependent pixel shaders, each of which computes intermediate results for the next one until the final results can be calculated. Obviously, it is not possible to convert each program in this fashion. To prevent a program from looping endlessly and thus blocking the GPU, recent architectures that support loops and subroutine calls stop the program after a specific number of instructions have been executed[11]. Recent GPUs terminate a shader after the 65536th executed instruction.

Although these restrictions appear massive, they are largely compensated by the highly parallel hardware that enables modern GPUs to perform certain computations much faster than recent CPUs. Additionally, the set of implementable algorithms is still large enough, which finally makes the GPU a promising target for general-purpose programming.

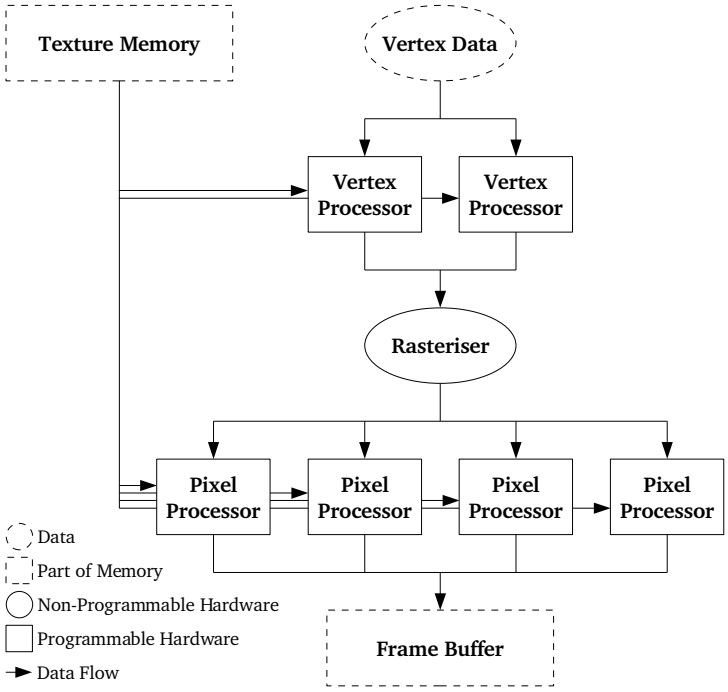

Figure II.2: The GPU rendering process in more detail.

11 This might soon change, as Direct3D 10 shaders will have no instruction count limit [21]. To maintain the systems stability, the operating system will then have to monitor the GPU to determine whether the graphics card has hung up and requires resetting.

1.3. Languages

As mentioned in the previous section, pixel shaders are just assembly programs. Because writing a program in an assembler language has always been both unintuitive and complex at the same time, several high-level (shading) languages that provide different levels of abstractions of the GPU architecture have emerged to unburden developers from the harsh reality of assembler programming. Being only designed to aid developers in coding visual-effect shaders, most of these languages still provide a rather low abstraction and thus cannot be used to fully exploit the high parallelism of the GPU architecture. To simplify the development of data-parallel algorithms a higher level of abstraction is required, as provided by Brook for GPUs and CGiS.

The following sections briefly introduce the most interesting languages available, whereas an emphasis is put on CGiS, as it is of further interest within this work. Not being discussed in this chapter – but worth mentioning – are the Stanford real-time shading language (RTSL) [23] on the one hand, and the Accelerator library [24] on the other hand.

1.3.1. RenderMan Shading Language

Shading languages have been invented long before the first graphics cards supported vertex and pixel shaders. In 1984 Robert L. Cook introduced shade trees [25], which offer a more sophisticated method of describing the shading properties of a surface. Instead of specifying the shading behaviour through a small set of variables or fixed models, such as the reflection model (e.g., Gouraud [26] or Phong [27]), a user could design a variety surface properties in this tree-structured shading model that only supported a handful arithmetical operations.

The invention of shade trees eventually led to the first available shading language that Pixar has published together with the RenderMan Interface Specification in 1989 [28]. This interface has been designed to simplify the communication between modelling and rendering programs capable of generating photo-realistic images. Describing three-dimensional scenes with three-dimensional primitives, this concept bears a strong resemblance to PostScript, which is used to design two-dimensional page layouts. Alongside with the RenderMan Interface, Pixar published the RenderMan Shading Language that describes surface properties in special procedures (shaders). In contrast to the other (shading) languages being discussed in the following sections, the RenderMan Shading Language is employed in offline rendering that aims at the best image quality possible.

Although not being executed on any GPU, the RenderMan Shading Language is still worth mentioning, because it can be understood as the predecessor of all shading languages. The language further extends the idea of shade trees, by allowing the user to write arbitrarily complex descriptions of surface properties. With the PhotoRealistic RenderMan, Pixar managed to create the first implementation of the RenderMan Shading Language, which was in fact the first shading language that has ever been implemented.

In contrast to the two available shader types on recent graphics cards, the language distinguishes six different shader types:

- The optical properties of illuminated objects are being modelled by *surface shaders*. They compute the output colour and opacity of a point on the surface by taking the incoming light and the physical properties into account.
- Any kind of light source can be implemented using *light source shaders*.
- Not being associated with surfaces, *volume shaders* modify the colour of light rays that pass through some part of space. Effects like fog can be realised with this shader type.

- *Displacement shaders* are related to surface shaders, as they can move surfaces around and manipulate the surface normal, resulting in a different illumination behaviour.
- In addition to displacement shaders, *transformation shaders* can additionally be used to modify the geometry of the scene. They describe a non-linear transformation of all points in space to new points.
- Comparable to image filters, *imager shaders* perform a transformation on the final pixels of the resulting image.

On a first look it appears that surface, light source, volume and imager shaders correspond to pixel shaders, whereas displacement and transformation shaders are some kind of vertex shader. However, as vertex and pixel shaders are strongly tied to the GPU architecture, it is unclear whether RenderMan shaders can be easily translated into vertex and pixel shaders.

1.3.2. GLSL

In conjunction with the OpenGL ARB[12], 3Dlabs developed the OpenGL Shading Language (GLSL, also known as glslang), which is the latest shading language available. Originally being introduced as an extension to OpenGL 1.5 in 2003, the OpenGL ARB has formally included GLSL in the OpenGL 2.0 specification that has been released in 2004 [19].

The GLSL syntax is a mixture of C and C^{++} with a strong emphasis on C. GLSL omits most C^{++} language features except for the concept of overloading functions and constructors that are used to initialise variables and to convert types. Because there are no implicit type conversions and no explicit casts, calling a constructor is the only way to perform a type conversion. Enums and unions are not supported, whereas GLSL provides all basic arithmetic types alongside with vector, matrix and structure types. Additionally, texture handles have been introduced to abstract the texture access. GLSL contains all operators of C and C^{++} except for bitwise and pointer operators. Bitwise operators are not provided, because there are, as mentioned in Chapter I, no integer data types available on the GPU. Albeit arrays and the array subscript operator are supported, pointer operations cannot be realised, as the GPU only has limited indirect addressing capabilities that restricts the GPU to only allow reading from registers and textures. Thus, there exists no GPU (shading) language that provides pointer data types. GLSL supports function calls, whereas neither direct nor indirect recursion is permitted. Additionally, labels as well as switch and goto statements are not supported due to the limitations of the GPU architecture.

```
void main (void)
{
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Figure II.3: Simple GLSL pixel shader that colours the resulting pixels red.

Figure II.3 shows a simple GLSL shader. As the main function misleadingly suggests, shaders written in GLSL are not stand-alone applications. They always require an auxiliary program that makes use of the OpenGL API. To run a GLSL shader, it has to be passed as string via the OpenGL API to the vendor's driver, where it will be compiled before it is finally executed on the GPU (see Figure II.4). The main advantages of this approach are that an external compiler is not necessary, and that new shaders can easily be created on the fly. However, results and performance of the generated shaders might vary from driver to driver, as the quality of the

12 Excerpt from [29]: “The OpenGL Architecture Review Board is an independent consortium formed in 1992 that governs the OpenGL specification and evolution.”

shaders heavily depends on the vendor's GLSL compiler implementation. Naturally, this is not really a problem, as the OpenGL API allows uploading of shaders written in assembler to the graphics card. Thus, a developer, who wants to avoid driver-specific differences, may still employ a GLSL compiler from an external, vendor independent library.

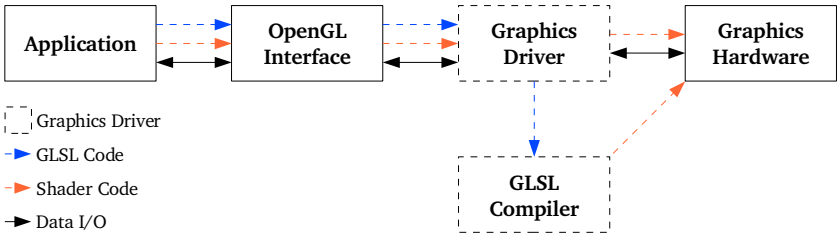

Figure II.4: Involved software and hardware layers when using GLSL.

1.3.3. HLSL

The DirectX pendant to GLSL is the High Level Shading Language (HLSL) that has been available with DirectX 9, which was released in December 2002. Being designed after similar design concepts, HLSL does not differ greatly from GLSL on a syntactical as well as on a semantic level. Additionally, HLSL shaders are – like GLSL shaders – no stand-alone programs, and thus require an auxiliary program that uses DirectX or, to be more precise, Direct3D. Besides the distinct runtime environments in which HLSL and GLSL shaders are employed, the main difference between the two languages is that, instead of being located in the graphics card driver, the HLSL compiler is part of the Direct3D library. As a consequence, driver-dependent differences in the generated shader programs cannot occur. Similar to GLSL, it is possible to compile new shaders during runtime without the need to recompile the main application. In contrast to GLSL, the Direct3D API also allows to only compile a HLSL shader and upload it manually to the GPU later on (see Figure II.5), whereas the HLSL compiler only generates DirectX pixel and vertex shaders. So, if a developer does not want to rely on the Direct3D compiler, the developer can also make use of an external compiler. However, this will soon no longer be possible, because the upcoming Direct3D 10 API will only accept shaders written in HLSL [21]. Nevertheless, HLSL is nowadays well established and mainly used in computer games to implement visual effects.

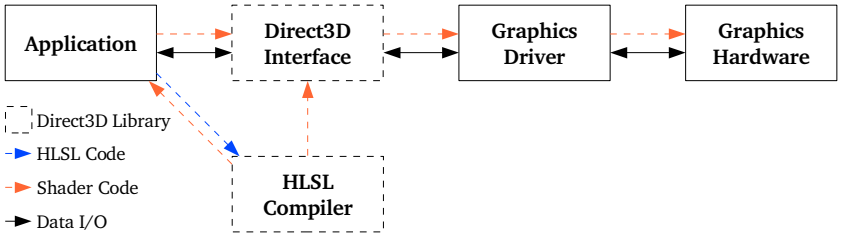

Figure II.5: Interaction between the host application and the HLSL compiler.

1.3.4. Cg

In June 2002, NVIDIA has officially released C for graphics (Cg) that has been developed concurrently to GLSL and HLSL. The key design decision was to create a general-purpose language rather than a domain-specific language like the RenderMan Shading Language [30]. One reason for designing a general-purpose language was to make the runtime costs of the operations understandable. This design concept would be infeasible with operations that abstract some kind of mechanism on a high abstraction level, as it would have been introduced in a domain-specific language. Furthermore, the Cg developers wanted to create a language that makes it possible to use the GPU for non-shading computations. As C manages to achieve performance and portability at the same time, the Cg development team took syntax, semantic and philosophy of C as basis for the Cg language specification. Besides extensions and modifications to C that are necessary to fully exploit the underlying GPU architecture, Cg introduces certain language constructs of C^{++} or Java (e.g., constructors), just as GLSL.

Thus, Cg is not very different from GLSL or HLSL from a syntactical and semantic point of view. Like GLSL and HLSL shaders, Cg shaders can be compiled during the runtime of an auxiliary application, which is a prerequisite before any Cg shader can be executed at all. It is additionally possible to only compile a Cg shader and upload the compiled shader manually to the graphics card, as Figure II.6 demonstrates.

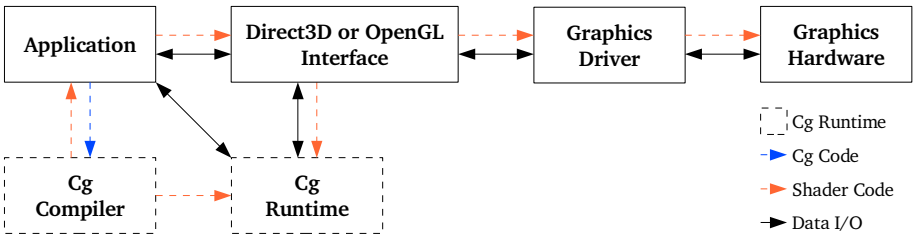

Figure II.6: Interplay between the application and the Cg runtime.

It turns out that Cg is one of the first – if not the first – graphics API independent shading programming languages, because it provides a Direct3D and an OpenGL interface as well. To support both graphics libraries, the Cg compiler is able to generate both DirectX and OpenGL pixel and vertex shaders. The Cg compiler supports the OpenGL pixel shader types that are defined in ARB_fragment_program, NV_fragment_program and NV_fragment_program2 and OpenGL vertex shaders that are defined in ARB_vertex_program and NV_vertex_program. This versatility makes Cg the language of choice for game developers who want to implement graphical effects that are guaranteed to run on every platform. Unsurprisingly, Cg is nowadays widely spread.

1.3.5. Sh

Sh was part of the research project SMASH that was dedicated to investigate the hardware features as well as the programmability of GPUs. The project comprised two parts, which was on the one hand Sh, and the GPU simulator Sm on the other hand. Instead of designing a new graphics programming language, the researchers invented a metalanguage that is completely embedded in C^{++}. By means of the GPU simulator Sm, the team around Michael D. McCool demonstrated the feasibility of this approach and claimed that it can be easily adopted to Direct3D and OpenGL [31]. Because the name SMASH already belonged to a related project, this name was no longer used and the project was eventually split up into its two parts Sh and Sm, of which Sm has not been developed any further. The first official Sh version that supported Direct3D and OpenGL was eventually available in July 2003.

When making use of Sh, a developer is able to program the graphics hardware using a single programming language (C++ in this context), instead of writing GPU programs in a language different to the application language. As Sh shaders can share variables with the main program, parameter binding code that associates program variables with shader variables is no longer necessary. Thus, less code is necessary to drive a shader program. In contrast to other graphics programming languages, Sh additionally unburdens the developer from texture handling, which further decreases the amount of necessary code. Comparable to GLSL, HLSL and Cg, shaders can be easily created on the fly in the main application, whereas it is not necessary to create some kind of textual representation. Unfortunately, this also implies that is not possible to load Sh shader code from a file or some other kind of medium during runtime, as the shader code is part of the main program. However, because less code is required, and a developer no longer has to care about several languages and the interaction between them, writing C++ embedded shaders finally reduces the overall complexity of an application and might additionally ease its debugging.

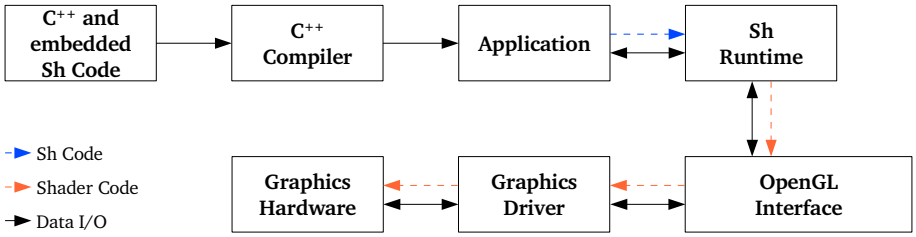

Figure II.7: Compilation and interaction of an Sh shader program.

Sh is a cross-platform graphics programming language that currently comes with both an OpenGL and a CPU back end (see Figure II.7), besides being available on Windows, Linux and Mac OS X. When using the GPU, Sh internally generates either OpenGL pixel and vertex shaders (only ARB_fragment_program and ARB_vertex_program) or GLSL shaders. As GPUs have become a promising target for general-purpose computation, the Sh development team aims for Sh to be a programming language that is suitable for any kind of computation on the GPU, as seen in Figure II.8 that shows an example Sh shader that adds two vectors. However, Sh is mainly used in shader programming.

```
// main code
int main (int argc, char **argv)
{
  // embedded shader code
  ShProgram program = SH_BEGIN_PROGRAM ()
  {
    ShInputAttrib1f u, v;
    ShOutputAttrib1f w;
    w = u+v;
  } SH_END;

  // init shader
  shCompile (program);
  ShChannel<ShAttrib1f> u(256), v(256), w(256);
  ...

  // execute shader
  w = program << u << v;
}
```

Figure II.8: Sh vector addition implementation.

1.3.6. Brook for GPUs

Brook for GPUs has emerged from the Brook language specification that has been designed as a programming language for the Merrimac streaming supercomputer [32]. A version of Brook for GPUs was already available in December 2003, before it has been presented to the public in August 2004 [33]. In the following, I will simply use Brook to stand representatively for Brook for GPUs.

Implementing a subset from the original language specification, Brook completely hides the underlying graphics hardware from the user, abstracting the GPU as a streaming coprocessor. This approach is feasible, because the GPU architecture offers certain mechanisms that are also found in stream programming. In contrast to other programming techniques, the stream programming model abstracts data as *streams*, which are sequences of records that require similar processing, whereas *kernels* represent functions that are applied to each element of a stream. The resemblance between streaming processors and vertex or pixel processors on the GPU is striking. Like a pixel processor that applies a pixel shader on each pixel of a texture, writing the output pixel to the frame buffer, a streaming processor executes a kernel over all elements of an input stream, while storing the results into an appropriate output stream.

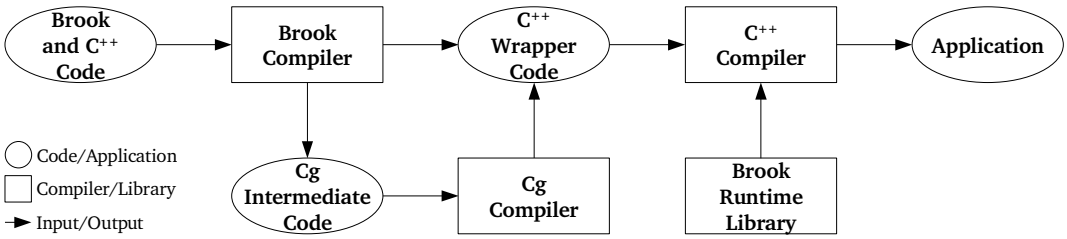

Figure II.9: Brook compilation process.

Like Sh, Brook is a platform independent programming language that provides a Direct3D, OpenGL and a pure CPU back end and offers a seamless integration of shader and C++ code. In contrast to Sh, Brook introduces stream programming language constructs, such as kernels and streams, to the C++ programming language. As these language extensions are normally not understood by a C++ compiler, a Brook program has first to be compiled with a special compiler, such as the Brook compiler brcc. Requiring yet another compiler, brcc is a source to source compiler that translates Brook code into pure C++ code. However, before generating the final C++ wrapper code, brcc produces intermediate Cg code that is compiled with the Cg compiler (see Figure II.9) that happens to create inefficient code under certain circumstances. For instance, Purcell et al. have experienced that a Cg implementation of an algorithm used 52 instructions, although it was possible to implement it using only 19 instructions [34]. However, optimising the generated assembler code heavily, the graphics driver might be able to diminish this drawback.

It turns out that Brook does not share all advantages of Sh. As the vector addition example in Figure II.10 clearly shows, Brook does not completely hide the texture handling from the user. Besides the fact that textures have to be declared explicitly, it is additionally not possible to access a once declared texture directly. Instead the user has to initialise textures using auxiliary variables that contain the initial data, which might confuse the user on the first look.

In contrast to other graphics programming languages, Brook is intended only to be employed in general-purpose programming. Being currently under active development, Brook has been used to realise the fast Fourier transform algorithm or a ray tracer among several other possible applications.

```
// shader code
kernel void add (float u<>, float v<>, out float w<>)
{
  w = u+v;
}

// main code
int main (int argc, char **argv)
{
  // init vectors
  float U[256], V[256], W[256];
  ...

  // init shader
  float u<256>, v<256>, w<256>;
  streamRead (u, U);
  streamRead (v, V);

  // execute shader
  add (u, v, w);

  // download data
  streamWrite (w, W);
  ...
}
```

Figure II.10: Vector addition in Brook.

1.3.7. CGiS

CGiS[13] is one of the most recent high-level graphics programming language that is currently being developed at the University of the Saarland. A description of the CGiS programming language has been officially introduced in November 2004 [35].

Like Brook, CGiS completely hides the underlying hardware and additionally does not require the user to know about the hardware's details. Yet, it might help to understand the capabilities and restrictions of CGiS more easily, if a developer knows what the GPU can and what it cannot do. Because there are no e.g., inverse trigonometrical functions available, or it is not possible at all to realise pointer operations (see Section 1.3.2), a user might become confused at first, when the user finds these features missing.

Similar to most languages that this chapter presents, C forms the basis of the CGiS language specification, whereas certain aspects of Pascal were also adopted. In contrast to Brook that abstracts the GPU as a streaming coprocessor and consequently requires a developer to think in terms of stream programming, CGiS abstracts the GPU as data-parallel hardware in a more general fashion. Furthermore, CGiS differs from Sh or Brook, as code that will be executed on the GPU and code that controls the execution on the GPU is strictly separated from the main application code. Just as pure shading languages (e.g., HLSL) are designed only to express the properties of some kind of material, CGiS is only used to describe the algorithm instead of the code that supplies the algorithm with data. This design concept was preferred over all others, because it helps the user to realise what exactly is going to be executed on the GPU.

Besides the three base types *bool*, *int* and *float* that are also part of C, CGiS introduces the new base type *half*, which is a floating point type with half the precision of a *float*, vectors of base types with a maximum width of four elements (e.g., *bool2*, *int3* and *float4*) and one- or two-dimensional streams of structures with predefined or arbitrary[14] size. Custom data types can addi-

13 CGiS abbreviates **C**omputer **G**raphics **i**n **S**cientific programming.
14 In this context, arbitrary means being defined at runtime.

tionally be defined with the *typedef* statement. Analogously to GLSL, CGiS keeps all C operators, except bitwise and pointer operators of C for the same reasons mentioned in Section 1.3.2, and introduces new operators that are specific for the GPU instruction set, such as cross and dot product, texture lookup, mask and swizzle. For efficiency reasons, CGiS additionally offers special syntactical constructs for primitive vector and matrix arithmetic.

```
PROGRAM add;

INTERFACE

extern in float U<_>;
extern in float V<_>;
extern out float W<_>;

CODE

function add (in float u, in float v, out float w)
{
  w = u+v;
}

CONTROL

forall (float u in U; float v in V; float w in W)
{
  add (u, v, w);
}
```

Figure II.11: CGiS program that adds two vectors.

```
#include "add.h"

int main (int argc, char **argv)
{
  // init vectors
  float *U, *V, *W;
  ...

  // upload data to gpu
  setup_sizes_add (256, 256, 256);
  setup_init_add ();
  set_data_add (STREAM_U, U);
  set_data_add (STREAM_V, V);
  setup_data_add ();

  // execute shader
  execute_add ();

  // download data from gpu
  get_data_add (STREAM_W, W);
  ...
}
```

Figure II.12: Directing the CGiS code.

As depicted by Figure II.11, to implement an algorithm in CGiS, it has to be split up into three sections:

- Data types, streams and variables are declared in the INTERFACE section. A variable can be marked as either *external* (accessible by the user) or *internal* (only accessible within the algorithm). When declaring an *external* variable, the user must additionally specify its flow direction that can either be *in*, to indicate that the variable can only be read, *out*, to only allow the variable to be written, or *inout*, to enable the variable to be read and written as well. The flow direction of *internal* variables is implicitly *inout*. The same rules apply to streams, whereas the user also has to specify the stream's dimensionality and the width of each dimension. Any dimension of a stream may also have an arbitrary width, which is indicated with an underscore (see Figure II.11). However, before any computation can occur, the sizes of each stream must be defined at runtime.
- The CODE section is a sequence of functions that may operate in parallel on single data elements. In contrast to C, functions do not have a return value. Instead, each function parameter has a defined flow direction that can be either *in*, *out* or *inout*, which enables functions to have multiple return values at once[15]. As neither indirect nor direct recursion is permitted, CGiS does not support function forward declarations. However, a function may still call another function as long as the callee has been declared previ-

15 The number of available function parameters with an *out* flow is determined by the number of components of the function parameters and is indirectly restricted by the maximum amount of textures a shader may render into. As a texture consists of pixels that are quadripartite float vectors, a GPU that supports only one output texture would restrict the available output parameters to either four *float*, two *float2* or one *float* and one *float3* parameter and so on. Recent GPUs support rendering into four textures.

ously. Each function can be understood as a shader template that will be used to create the final shader code. If e.g., a function is only called from another function, it will automatically be inlined, which finally results in a single function containing the code of the involved functions.

- The CONTROL section specifies, how the parallel computation has to take places. This section comprises at least one forall loop that may call any function that is contained in the CODE section. The sequence of the forall loops, specified in the CONTROL section, initiates the computation on the GPU, whereas each forall loop stands representatively for what is to be computed. Besides specifying the computation, the forall loops determine constraints involving the declared streams.

  In Figure II.11, the forall loop takes one element out of the three declared streams. To ensure that no element is selected twice, the arrays *U*, *V* and *W* must have the same sizes. Thus, setting distinct sizes for these arrays, will result either in a compile time or runtime exception, depending on whether the sizes are defined at compile time or during runtime.

As CGiS programs are not stand-alone applications, an auxiliary program is required to direct a CGiS program. Figure II.12 shows how the generated shader is initialised, before it can be executed and the resulting data can finally be downloaded from the frame buffer into the computer's main memory.

The CGiS compiler cgisc is a source to source compiler that translates CGiS code to C^{++} wrapper code, which encapsulates the generated shader code. In contrast to the Brook compiler, cgisc does not rely on an external compiler. Instead, the CGiS compiler generates shader code directly, which gives cgisc full control over the complete algorithm. Figure II.13 shows, how the compilation process actually takes place. Currently, the CGiS runtime library only supports an OpenGL back end, which is sufficient, because OpenGL is available for multiple platforms. Until now, there are no plans to integrate a Direct3D interface in the near future. It is unsure whether there will be such an attempt at all, because the upcoming Direct3D API will only accept shaders written in HLSL (see Section 1.3.3). Generating high-level shader code contradicts the CGiS team's design decision not to rely on external compilers, because this would diminish the control over the generated GPU assembler code.

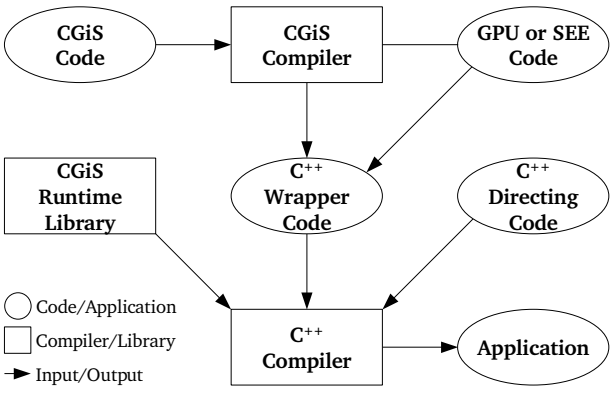

Figure II.13: CGiS compilation process.

Altogether, CGiS is an interesting language that supports general-purpose programming on a high level of abstraction that enables a developer to easily exploit the vast computational resources of recent GPUs. CGiS even allows a developer to exploit the power of SSE capable processors, because the CGiS compiler is also capable of generating SSE code. The compiler has unfortunately not yet been published, so CGiS is more or less unknown to the general public.

2. Compilers

Since the beginning of computer programming, compilers have been indispensable tools that ease and speed up the development of any software project. Although implementing certain algorithms for efficiency reasons, realising whole applications or operating systems completely in an assembly language would be nowadays unthinkable.

The following two sections describe the general design of a compiler, according to the design presented in [36], and compare this design to the structure of the CGiS compiler. Section 2.2 further reveals the internals of cgisc, showing how programs are internally represented, how the code generation takes place, and where exactly retargetable pattern matchers can be employed to improve the code generation and optimisation process.

2.1. General Design

In general, two disjoint modules constitute a compiler, as seen in Figure II.14. On the one hand, the *front end* reads the input data, creates an internal representation out of the input program and performs several analyses as well as platform-independent optimisations, whereas on the other hand the *back end* is responsible for the target code generation and optimisation using the input program representation that has been previously created in the compiler's front end. Both modules themselves further comprise several interdependent submodules.

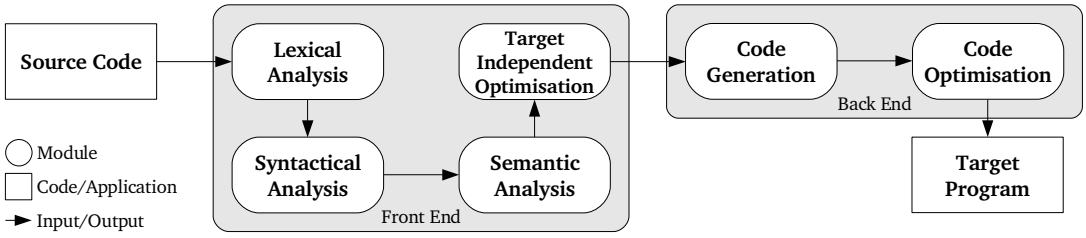

Figure II.14: A very general compiler structure.

Before the internal representation of the input program is passed to the back end, three analyses have to take place in the front end:

- The *lexical analysis* receives the input program as a sequence of characters and transform it into a sequence of terminal symbols of the source language. Considering the C programming language, typical terminal symbols would be separators (e.g., `;`, `{` or `}`), operators (e.g., == or +=) or reserved keywords (`typedef`, `struct`, etc.) amongst others.
- The *syntactical analysis* checks whether the input program is syntactically well-formed while converting the sequence of terminal characters into an abstract representation of the input program. The input program is commonly represented as a syntax tree or a call graph.
- Processing the input program representation, the *semantic analysis* determines whether certain language constraints have been violated that cannot be verified without context sensitive information. While introducing context sensitive data to the internal program representation, this analysis usually checks for type violations (e.g., Java does not allow the assignment of an integer to a boolean variable), whether a used variable has been declared beforehand or whether a variable has been declared multiple times.

- Additionally, certain platform-independent optimisations can be performed before the final internal representation is passed to the back end. These optimisations include e.g., the computation of expressions, whose values are known at compile time (*constant propagation*), or the removal of dead code (*dead code elimination*). Although not being an analysis, this step is understood as a part of the front end, as the applied optimisations are independent from the target architecture and the target language as well.

  Occasionally, submodules that cannot be clearly assigned to either front end or back end are summed up in an intermediate module called *middle end*.

If an error occurs during any of the three analyses, the compiler stops, reporting the most reasonable cause(s). As stated in [36], the front end, and thus any error that may occur during this processing phase, is ideally not related to the target architecture or target language. This strict separation is however not always possible, especially when the compiler comprises multiple back ends. Due to the differences between the supported GPUs, this problem inevitably applies to the CGiS compiler, as discussed in Section 2.2.

If the front end finishes otherwise without errors, the processing continues in the back end, where two final steps have to be performed. First, the code generator translates the abstract representation of the input program into the target language. During that process the generator might have to cope with the *register allocation*[16] and the *code selection*[17] that both depend on the target language. It is e.g., very unlikely that a source to source compiler will have to handle register allocation. Second, the code optimiser tries to improve the generated code by replacing certain instruction sequences with more efficient sequences or by reordering instructions (*instruction scheduling*). To perform these optimisations, the optimiser generally takes only a look at parts of the program and is thus only capable of computing local optima. Whenever an optimisation is applied, certain computations that were executed during the code generation usually have to be run again. If e.g., some instructions have been removed, the previously computed information about required registers might be outdated and the target program could be implement using fewer registers. As a matter of fact, code generation and optimisation are often interconnected submodules, because they normally cannot process the input program in a single pass. This problem is known as the *phase ordering problem*.

When back end has eventually finished processing, the compilation process ends, and the target program is finally available.

16 The register allocation tries to minimise the usage of memory cells by assigning registers to variables such that the available registers are used efficiently, whereas as few memory accesses as possible have to be performed.

17 Most architectures offer multiple equivalent instruction sequences for the same computation. As these instruction sequences may however differ in their execution time, depending on the actual context, the code selection has to select the most reasonable sequences.

2.2. CGiS Compiler Design

The following two sections discuss how the CGiS compiler internally represents any input program and how the compilation process actually takes place, with respect to code generation and optimisation. As described in Section 1.3.7, a CGiS source file is separated into the three sections INTERFACE, CODE and CONTROL. Although cgisc is also capable of generating SSE code, only the generation of GPU code is of further interest for the remainder of this chapter.

2.2.1. Internal Representation

The CGiS compiler internally represents each function as a *basic block* control flow graph. A basic block is a sequential, directed, branch-free graph of maximum length, whereas each node represents an instruction of the program. To ensure that the basic block can only be entered through the first node and only be exited through the last node, every node except the first and the last must have an in-degree and an out-degree of 1. Common instructions that begin a basic block are procedure entry points or targets of jumps or branches, whereas branching instructions or return statements usually end a basic block.

Each instruction in cgisc has at least one target and either one, two or three operands. To be compatible with the program analysis toolkit PAG [37], each control flow graph contains two mandatory basic blocks, of which the first one initialises all function parameters that have an in flow, whereas the second one writes the final results to all function parameters with an out flow. Figure II.15 shows how the CGiS compiler internally represents the *add* function of the vector addition example (see Figure II.11).

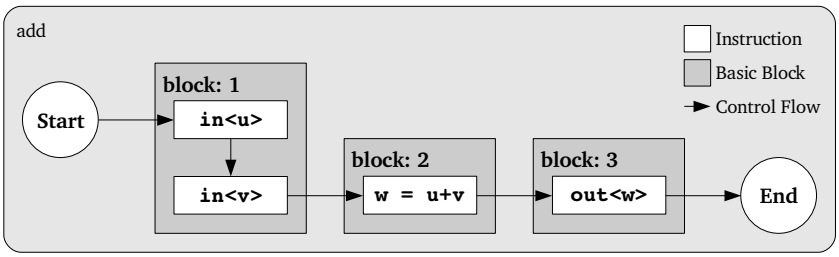

Figure II.15: Internal representation of a CGiS function.

2.2.2. Compiler Structure

As depicted by Figure II.16, the CGiS compiler implements the general compiler design, whereas it extends both front and back end with additional intermediate phases that are required for the GPU code generation.

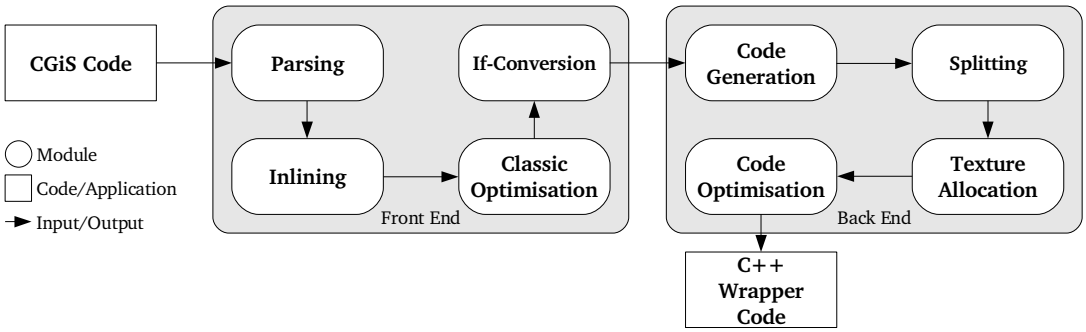

Figure II.16: Internal structure of the CGiS compiler.

Before the code generation commences, the front end processes an input program as follows:

- At first, the parser generates an internal representation of the program, while performing the lexical, the syntactical as well as the semantic analysis. At this point, it is already possible that the compiler rejects an input program due to incapabilities of the target architecture. For instance, if the CGiS compiler detects a loop, cgisc rejects the input program, if the target GPU does not support loops.
- In the next phase, the CGiS compiler inlines all function calls, if the target architecture does not support subroutines (e.g., NV30).
- Afterwards, cgisc employs classic optimisations that correspond to the target independent optimisations presented in Section 2.1.
- Although certain GPUs do not support branching (e.g., NV30), it is still possible to implement branching on these GPUs, if conditional assignments can be realised. This *if-conversion* additionally requires the CGiS compiler to transform the control flow graph of every function, such that no branches occur. This is done by executing the code of each branch, whereas the final results are first assigned to temporary variables. The conditions decide afterwards which values have to be assigned to the target variables. By definition, this transformation will result in a large basic block that contains the whole function body. See Figure II.17 for an example.

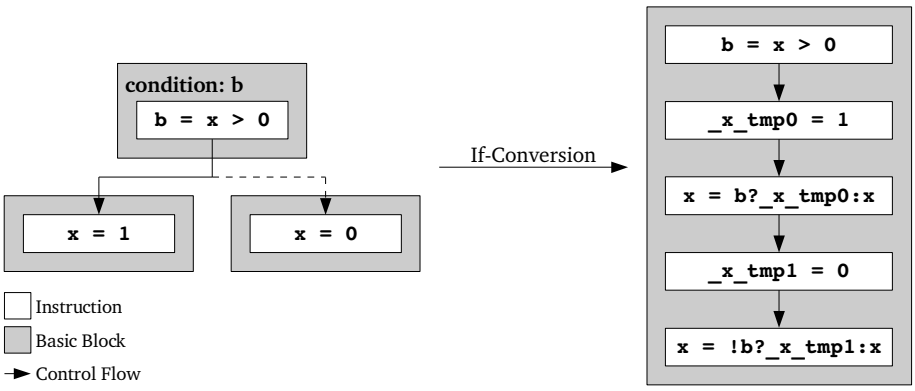

Figure II.17: Applied if-conversion.

If the front end has finished its work on the internal representation of the input program, the back end can finally begin generating the target code. Besides having to determine the number of required registers, the compiler also has to cope with two other problems. On the one hand, if a generated shader comprises too many instructions, cgisc must split it up into several interdependent shaders (*splitting*), because shaders may not be arbitrarily long, as mentioned previously in Section 1.2. The splitting phase might additionally have to introduce new intermediate streams to cache the results of the interdependent shaders. On the other hand, the compiler has to determine how many textures are necessary to store the declared and intermediate streams (*texture allocation*). It turns out that the phase ordering problem is also encountered in the CGiS compiler, as both phases depend on each other. In fact both phases are executed at the same time. Additionally, the compiler might have to rerun the previous phases, as the code optimisation possibly reduces the number of instructions in a shader. However that may be, a detailed discussion of the phase ordering problem is beyond the scope of this work.

After the back end has terminated, the compiler wraps the generated GPU code in a C^{++} header and source file that provide the infrastructure for executing the compiled program on the GPU.

2.2.3. Code Generation

As described in Section 2.2.1, every function is internally represented as a control flow graph that consists of basic blocks, whereas a basic block is a branch-free sequence of instructions. An instruction is an instance of a class that represents e.g., an unary or a binary operation. Instead of standing representatively for actual operations, the instruction objects are also responsible for the code generation, so they know how to generate corresponding code for any target architecture. Independent from the target GPU, the compiler processes the function's basic blocks and the contained instructions one after another by invoking the code generation function to generate the target code, while storing the results in the corresponding target basic block. This code selection method is completely deterministic, as the invoked instruction objects have no knowledge about their surroundings, and thus, there is always exactly one fitting instruction target code sequence. Although this implementation keeps the code generation as simple as possible, it unfortunately leads to suboptimal code and further has the disadvantage that the differences between the supported GPUs are mixed inside the code, which makes it the more unmaintainable the more different target architectures are supported.

Some GPU architectures (e.g., NV30 or A300) do not feature a division operator. Instead they provide the *RCP* (reciprocal) instruction that computes the multiplicative inverse of its operand. Thus, other than multiplying the nominator with the multiplicative inverse of the denominator, there is no other way of dividing two numbers on these architectures. Additionally, it is not possible to compute the square root directly, because recent GPUs only offer the *RSQ* (reciprocal square root) instruction that calculates the multiplicative inverse of the square root of a non-negative number. These peculiarities together with the unawareness of the instruction objects about their surroundings cause the CGiS compiler to generate less efficient code under certain circumstances, as Figure II.18 demonstrates.

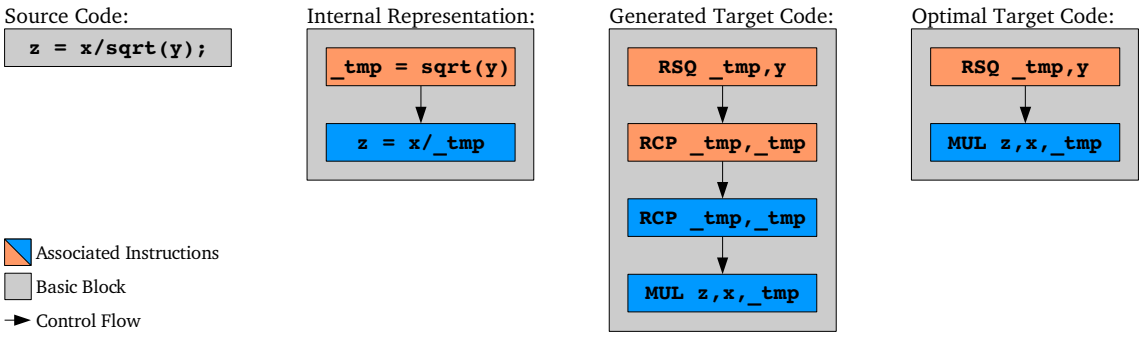

Figure II.18: Generating inefficient code.

To implement the expression `z = x/sqrt(y)`, cgisc generates four instructions, although it is possible to realise this expression using only two instructions. The compiler internally represents the given instruction using an unary operation that stores the square root of y in some temporary variable *_tmp* and an binary operation that assigns the value of x divided by *_tmp* to *z*. During the code generation, the unary operation does not know that its result is used as denominator in the next division operation. Thus, it generates two instructions (marked orange) to compute the square root of *y* to store it in the temporary variable *_tmp*. Due to the lack of a division operator, the binary operation has to generate code (marked blue) that first computes the multiplicative inverse of *_tmp* using the *RCP* instruction and that afterwards assigns the value of *x* multiplied with *_tmp* to *z*. However, as the second *RCP* instruction reverses the effect of the previous one, both *RCP* instructions can be omitted, which results in the optimal implementation of the given expression.

Instead of optimising the inefficient code later on, the CGiS compiler could be improved by replacing the code generation process with a more sophisticated approach that, on the one hand, allows the developer to specify the code generator's behaviour depending on the target architecture, and that takes the surroundings of an instruction into account to avoid problems like the one described above, on the other hand.

As a solution, the present work proposes the pattern matcher generator tpmg (see Chapter IV) that is capable of generating retargetable pattern matchers that operate on basic blocks, so that they can replace the existing code generator and are even able to perform certain optimisations. The generated pattern matchers can be employed in any compiler that internally represents programs as basic block control flow graphs.

III. Theory

In the first section, this chapter introduces the general idea behind the pattern matchers the pattern matcher generator tpmg creates (see Chapter IV). The second section introduces the used mathematical notation and formally describes the theoretical concepts that realise retargetable pattern matchers. Discussing the formal description of a pattern matcher, demonstrating the functioning of a pattern matcher and analysing the (worst-case) runtime of the pattern matching process, third section concludes this chapter.

1. General Idea

A retargetable pattern matcher combines two disjoint notions, as a closer look at the term *retargetable pattern matcher* reveals. On the one hand, a *pattern matcher* identifies certain patterns and replaces them appropriately. In this context, a *rule* defines a pattern to identify and how it should be replaced. On the other hand, the pattern matcher is *retargetable*. So, a pattern matcher is able to process the same input differently, depending on the rules to use; the target platform to process the input for. Rules that are specific to a certain target constitute a *profile*.

It turns out that – at least in this context – retargetable pattern matchers have a quite simple structure. A retargetable pattern matcher comprises several profiles appropriate to the targets the pattern matcher should support, whereas a profile contains several rules that specify how to process the input. However, apart from the structure of retargetable pattern matchers, the most interesting question is how to formally describe profiles and rules.

The formal description of a profile is not challenging, because profiles are just a finite sets of rules (see Section 3.3). So, the rules are the most interesting aspect of pattern matchers. Roughly speaking, a rule comprises a *search pattern* and a *replace pattern* that depends on the search pattern in general (see Section 3.1 and 3.2 for more details about patterns and rules). The rule's search pattern is a compact representation of a finite state automaton that matches certain input words. However, the standard finite state automaton model and its functioning are not quite sufficient to realise the desired matching behaviour, as Section 2.2 shows.

In the following, I will use the term *basic block* as synonym for the term *input word*, because the input words of a pattern matcher are basic blocks in this context.

Discussing why the common finite state automaton model is insufficient for this approach, Section 2.3 introduces a special kind of finite state automaton that differs from the standard finite state automaton in the following points (amongst other things):

- To enable an automaton to decide as early as possible whether it should reject an input word, transitions are guarded by a side condition. This modification of the automaton model has a positive side effect on the overall runtime of the pattern matcher.
- Because a pattern matcher is supposed to process an input word using several automata, it makes sense to redefine the acceptance behaviour such that an automaton accepts an input word, even if the automaton only has consumed a prefix of the input word and the automaton's current state is a final state. This different acceptance behaviour enables the pattern matcher to combine multiple automata.
- Because the different acceptance behaviour may cause an automaton to reach several accepting states for the same input word, matching an input word requires some kind of memory to keep track of all those accepting states, so that the pattern matcher is able to choose the most interesting of them (e.g., the state that consumed most of the input symbols).

2. Theoretical Background

2.1. Basics

The following notation is used throughout this chapter:

- For the remainder of this chapter, Σ denotes an arbitrary **alphabet**, which is a finite, non-empty set of symbols, whereas $\mathbb{N}$ denotes the set of natural numbers, including 0.
- Let $n \in \mathbb{N}$. Iff $w = w_1 w_2 ... w_n$, whereas $\forall i \in \mathbb{N}$ with $1 \leq i \leq n$ then $w_i \in \Sigma$, w is a **word** over Σ of length n. The term $w[i]$ denotes the i-th symbol w_i.
- The symbol ε denotes the **empty word**, whereas $\forall \Sigma.\ \varepsilon \notin \Sigma$.
- If $1 \leq i < j \leq n$, $w[i{:}j] \coloneqq w_i ... w_j$ is a **subword** of w. For any other i and j, $w[i{:}j] \coloneqq \varepsilon$.
- The **length** $n \in \mathbb{N}$ of a word w over Σ is written as $|w| \coloneqq n$.
- The set of all words over Σ is $\Sigma^* \coloneqq \{w \mid w \text{ is a word over } \Sigma \wedge |w| \geq 0\}$.
- The set of all non-empty words over Σ is $\Sigma^+ \coloneqq \{\mathrm{w} \mid \mathrm{w} \text{ is a word over } \Sigma \wedge |\mathrm{w}| \geq 1\}$. It obviously holds that $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.
- Let $v, w \in \Sigma^*$. If $v = v_1 ... v_n$ and $w = w_1 ... w_m$, the **concatenation** of the words v and w is $v.w \coloneqq vw = v_1 ... v_n w_1 ... w_m$. The empty word is the identity element with respect to the concatenation operator, so $\varepsilon.w = w.\varepsilon = w$.
- *Let* $w \in \Sigma^*$. Iff $\exists x, y \in \Sigma^*$, so that $w = xy$, then x is a **prefix** and y is a **suffix** of w.

Any subset of Σ^* is a **formal language** over Σ with the following operations:

- $L_1 \cup L_2 \coloneqq \{w \mid w \in L_1 \vee w \in L_2\}$ is the **union** of the formal languages L_1 and L_2.
- $L_1 L_2 \coloneqq \{vw \mid v \in L_1 \wedge w \in L_2\}$ denotes the **concatenation** of the formal languages L_1 and L_2.
- The **closure** of a formal language L is defined as $L^* \coloneqq \bigcup_{n \geqslant 0} \{w_1 ... w_n \mid w_i \in L,\ 1 \leq i \leq n\}$.

Additionally, the following abbreviations are used:

- The set $Y \coloneqq \{\, x.y \mid x, y \in \Sigma^*\}$. For any $w = xy \in \Sigma^*$, the word $x.y \in Y$ represents the progress of an automaton that has already processed the prefix x and that has not yet handled the suffix y. The next input symbol to consume is then $y[0]$, if $y \neq \varepsilon$.

2.2. Finite State Automaton

Processing a basic block, a pattern matcher has to identify instructions patterns that are to be replaced with appropriate sequences of instructions for the target architecture. These instruction patterns constitute nothing else but a regular language. Because finite state automata recognise regular languages, finite state automata form the basis of every pattern matcher. The following definitions introduce regular languages and finite state automata.

As regular expressions are commonly used for describing regular languages, they could also be used to describe the instruction patterns that a pattern matcher should replace. However, instead of regular expressions, I will introduce a domain-specific method to describe instruction patterns in Section 3.1.

Definition 2.2.1 (Regular Language)
The **regular languages** over Σ are inductively defined as follows:

- The empty set is a regular language over Σ.
- $\forall a \in \Sigma$. $\{a\}$ is a regular language over Σ.
- If L_1 and L_2 are regular languages over Σ, then $L_1 \cup L_2$, L_1L_2 and L_1^* are also regular languages over Σ.

Example 2.2.2
Table III.1 shows some regular languages and words they contain.

Regular Language	*Contained Words*
$\{a, b, c\}$	a, b, c
$\{a, b\}^*$	$\varepsilon, a, b, aa, ab, ba, bb, ...$
$\{a\}^* \cup \{b\}^*$	$\varepsilon, a, b, aa, bb, aaa, bbb, ...$
$\{ab\}^*\{a\}$	$a, aba, ababa, ...$

Table III.1: Example regular languages.

Definition 2.2.3 (Finite State Automaton)
A **(non-deterministic) finite state automaton** (NFA) is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

- Σ is the **input alphabet**,
- S is the non-empty set of **states**,
- $s_0 \in S$ is the **initial state**,
- $\delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S$ is the **transition relation**,
- $F \subseteq S$ is the set of **final states**.

A finite state automaton is **deterministic**, iff there exist no $s, t \in$ S, so that $(s, \varepsilon, t) \in \delta$ and for any $s \in S$ there exist no $t_1, t_2 \in S$ and $a \in \Sigma$, so that $(s, a, t_1) \in \delta \wedge (s, a, t_2) \in \delta \wedge t_1 \neq t_2$. The same effect can be gained, if δ is required to be a **transition function** of the type $S \times \Sigma \to S$.

To determine whether an NFA accepts an input word, the notions *configuration* and *step* are required. When processing an input word, the configuration represents the current situation, in which the automaton resides, whereas a step transfers a configuration into another when the automaton consumes the next input symbol.

Definition 2.2.4 (Configuration, Step, Acceptance)
Let $N = (\Sigma, S, s_0, \delta, F)$ be an NFA. A pair (s, w) with $s \in S$ and $w \in \Sigma^*$ is a **configuration** of N, whereas (s_0, w) is an **initial configuration** and (s_f, ε) with $s_f \in F$, is an **final configuration**.

A configuration is transferred into another using the **step** relation $\Rightarrow_N \subseteq (S \times \Sigma^*) \times (S \times \Sigma^*)$. The transition $(s, aw) \Rightarrow_N (t, w)$ is valid for any $s, t \in S$ and $a \in \Sigma \cup \{\varepsilon\}$, iff $(s, a, t) \in \delta$.

Iff there exists a series of configurations $(s_0, w) \Rightarrow_N ... \Rightarrow_N (s_f, \varepsilon)$ and $s_f \in F$, then N **accepts** the word $w \in \Sigma^*$.

The set $L_N = \{w \in \Sigma^* \mid w \text{ is accepted by } N\}$ is the **accepted language** of N.

Example 2.2.5

State	*Symbol*	*Target State*
s	a	t
s	a	u
t	b	s

Table III.2: Transition relation of an NFA that accepts the language $\{ab\}^*\{a\}$.

Let $\Sigma = \{a, b\}$ and $N = (\Sigma, \{s, t, u\}, s, \delta, \{u\})$ be an NFA. The transition relation δ is defined such that N accepts the regular language $\{ab\}^*\{a\}$. Showing the input symbol that is required to reach a state from another, Table III.2 displays δ. Transitions other than those specified in the table above are not allowed.

Because there exists the series of configurations $(s, aba) \Rightarrow_N (t, ba) \Rightarrow_N (s, a) \Rightarrow_N (u, \varepsilon)$, N accepts the input word *aba*. The automaton ejects the input word *abb,* because no configuration series beginning at (*s, abb*) ends with a final configuration. Only the following two configuration series of configurations are possible:

- $(s, abb) \Rightarrow_N (t, bb) \Rightarrow_N (s, b)$. At this point, the automaton cannot proceed any further, because there is no transition that originates in the state s under the symbol b.
- $(s, abb) \Rightarrow_N (u, bb)$. Although N has reached the final state u, the automaton rejects the input word, because the last element of the configuration series is not a final configuration.

The above automaton representation turns out to be unintuitive and confusing, especially if the automaton to represent is pretty complex. Instead, finite state automata are commonly represented as *state transition diagrams*, which are finite, directed, edge-labelled graphs. To receive the corresponding state transition diagram for any finite state automaton, simply interpret the input alphabet Σ as available edge labels, the states of the automaton as vertices, every triple $(s, a, t) \in \delta$ as an edge labelled with a between the vertices s and t and specially mark the initial vertex and the final vertices, so that they can be discerned from the others. Analogously, the corresponding finite state automaton can be derived for any state transition diagram.

Example 2.2.6

Figure III.1 shows the corresponding state transition diagram for the NFA N of Example .5.

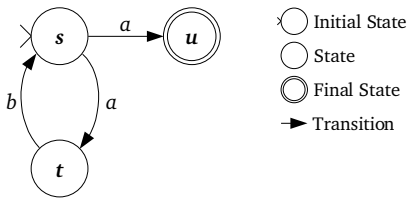

Figure III.1: State transition diagram for the NFA of Example .5.

2.3. Predicate Object Automaton

To process a basic block, it is preferable to use several small automata, each of which is specialised to a certain kind of problem, instead of a a single, monolithic automaton. By dividing the matching problem into disjoint subproblems – a very common procedure in computer programming – changes and additions can be more easily performed and it is likely that errors can be tracked down faster than in the monolithic approach. However, finite state automata and their method of processing an input word, as presented in Section 2.2, appear to be unsuitable for this application for several reasons.

A closer look at an input word of a patter matcher reveals that every input symbol is in fact an instance of the abstract representation of an instruction. The consequence is that – besides its type – each symbol has additional properties, such as the target register and the operands of the instructions. So, input symbols that have the same type, may still differ from each other. This observation furthermore leads to the fact that a pattern matcher's input alphabet comprises (instruction) classes, of which the input symbols are instantiated. Because a class may be derived from another one, certain input symbols belongs to multiple classes at once, as the following example shows.

For the remainder of this chapter, Σ denotes an alphabet, of which each symbol denotes an instruction class.

Example 2.3.1
Let $\Sigma = \{Unary, Sqrt\}$, where the class *Unary* denotes a generic unary operation and the class *Sqrt* represents a square root operation. The class *Sqrt* derives from the class *Unary*, because the square root is an unary operator.

If the current input symbol a is an instance of the instruction class *Sqrt*, the symbol a obviously belongs to the class *Sqrt* and to the the class *Unary* as well, because *Sqrt* derives from *Unary*. Otherwise, if a is only an instance of *Unary*, a can only be assigned to the class *Unary*.

Note that this notion introduces a little type clash. The concepts introduced in Section 2.1 require that each symbol of a word over Σ is contained in Σ. But how can an instance of a class be contained in a set of classes? To resolve this conflict, I redefine the $\in$-operator such that – in this context – $a \in \Sigma$ is equivalent to $class(a) \in \Sigma$, where $class(a)$ denotes the class of which a is an instance.

For this reason, the notion of a transition from state s to state t under the class a – remember that the input alphabet now consists of instruction classes – has to be redefined such that the automaton may take the transition whenever the automaton encounters an input symbol that is an instance of the class a.

Additionally, it makes sense to guard transitions by side conditions to enable the automaton to rule out false positives that the automaton would generated very likely, if it would not verify certain properties of the input symbols. A typical side condition would check whether the target register of an instruction is the operand of another instruction. These transition guards enable the automaton to stop the matching as early as possible, which finally improves the processing speed and reduces the amount required memory.

Besides the limited transition relation, there are yet other reasons that render the finite state automaton model unsuitable to implement a pattern matcher. To enable a pattern matcher to process an input basic block using multiple automata, the automata must have a different acceptance behaviour that causes the automata to accept an input word, even if only a prefix of

the input word has been consumed. This less strict acceptance behaviour might cause such an automaton to accept the same input would differently, having consumed different prefixes of the input word. This behaviour is desired, as any of these prefixes can be taken taken as vantage point to generate the most desirable (in general most efficient) target instruction sequence. To compute all possible prefixes, the automata must explore all possible matches (in parallel). Besides the prefixes, the automata additionally must provide information under which input symbol a state has been reached, because the properties of the input symbol – in combination with the target state – may have an extra influence on the code that the pattern matcher generates. If e.g., a pattern matcher compiles an addition instruction, the pattern matcher has to know about its target register and its operands. Thus, the automation that matches the addition instruction has to keep the instance of this instruction, so that the pattern matcher can extract the necessary information.

The following definition introduces the predicate object automaton that features the properties mentioned above.

Definition 2.3.2 (Predicate Object Automaton)

Let $B = \{true, false\}$ be the set of boolean constants, and *Predicate* be a finite set of boolean predicate functions of the type $\mathbb{N} \times (\mathbb{N} \times S \to \Sigma^*) \to B$[18].

A **predicate object automaton** (POA)[19] is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

- Σ is the finite, non-empty **input alphabet**,
- S is the non-empty set of **states**,
- $s_0 \in S$ is the **initial state**,
- $\delta \subseteq S \times ((\Sigma \times Predicate) \cup \{\varepsilon, skip\}) \times S$ is the **transition relation**,
- $F \subseteq S$ is the set of **final states**.

In a POA, boolean predicate functions guard the transitions the automaton takes. So, before a POA may consume the input symbol a and take the transition $(s, (b, q), t)$, the automaton has to check whether a is an instance of the class b and whether the function q evaluates to *true*. Additionally, the new automaton model introduces *skip*-transitions, whose purpose is to omit the current input symbol such that the automaton does not consume it. Just like ε-transitions, *skip*-transitions are not guarded by boolean predicate functions.

To enable a POA to compute all possible alternatives and to report the matched symbols to the pattern matcher, some kind of memory is required. The following definition introduces both state and memory functions that realise the desired functionality.

Definition 2.3.3 (State Function, Memory Function)

Let $P = (\Sigma, S, s_0, \delta, F)$ be a POA. Any function $\mathbb{N} \to S \times Y$ is a **state function** of P. The state function $\lambda n.(s_0, \varepsilon.w)$ is the **initial state function** for the word $w \in \Sigma^*$. In the following, *State* denotes the set of state functions.

Any function f: $\mathbb{N} \times S \to \Sigma^*$ is a **memory function** of P, iff $\forall n \in \mathbb{N}.\ f(n, s_0) = \varepsilon$. The memory function $\lambda(n, s).\varepsilon$ is the **initial memory function**. From now on, *Memory* denotes the set of all memory functions.

18 The function type $\mathbb{N} \times S \to \Sigma^*$ denotes some kind of memory that is introduced in Definition .3.

19 I have chosen the name predicate object automaton for this special kind of finite state automaton, because on the one hand transitions of the POA are guarded by boolean *predicate* functions, and on the other hand the automaton consumes instances (*objects*) of the classes in Σ.

A state function memorises the progress of a POA. For every alternative[20] the POA explores, the state function stores the state in which the POA currently resides and the remainder of the input word. For each alternative, a memory function remembers under which input symbol a state has been reached.

To describe how a POA processes an input word, the following functions that operate on state and memory functions are required.

Definition 2.3.4 (Skip, Consume, Copy, Erase)
Let $P = (\Sigma, S, s_0, \delta, F)$ be a POA, f a state function of P and g a memory function of P.

The function *skip*: $\mathbb{N} \times S \times State \times Memory \to State \times Memory$ omits the first symbol of the unprocessed suffix and associates that symbol with the given target state. Let $n \in \mathbb{N}$, $s, t \in S$, $a \in \Sigma$ and $x, y, z \in \Sigma^*$, so that $f(n) = (s, x.ay)$ and $g(n, t) = z$.

The result $(f', g') = skip(n, t, f, g)$ is then defined as:

$$f' = \lambda k.\begin{cases}(t, xa.y) & k=n \\ f(k) & \text{else.}\end{cases} \qquad g' = \lambda(k, s).\begin{cases}za & k=n \wedge s=t \\ g(k, s) & \text{else.}\end{cases}$$

The function *consume*: $\mathbb{N} \times S \times State \times Memory \to State \times Memory$ consumes the first symbol of the unprocessed suffix and associates that symbol with the given state. Let $n \in \mathbb{N}$, $s, t \in S$, $a \in \Sigma$ and x, y, z in Σ^*, so that $f(n) = (s, x.ay)$ and $g(n, t) = z$.

The result $(f', g') = consume(n, t, f, g)$ is then defined as:

$$f' = \lambda k.\begin{cases}(t, x.y) & k=n \\ f(k) & \text{else.}\end{cases} \qquad g' = \lambda(k, s).\begin{cases}za & k=n \wedge s=t \\ g(k, s) & \text{else.}\end{cases}$$

The function *copy*: $\mathbb{N} \times \mathbb{N} \times State \times Memory \to State \times Memory$ copies an entry of the given state and memory function. Let $n, m \in \mathbb{N}$.

The result $(f', g') = copy(n, m, f, g)$ is then defined as:

$$f' = \lambda k.\begin{cases}f(n) & k=m \\ f(k) & k \neq m.\end{cases} \qquad g' = \lambda(k, s).\begin{cases}g(n, s) & k=m \\ g(k, s) & k \neq m.\end{cases}$$

The function *erase*: $\mathbb{N} \times State \times Memory \to State \times Memory$ removes an entry from the given state and memory function. Let $n \in \mathbb{N}$.

The result $(f', g') = erase(n, f, g)$ is then defined as:

$$f' = \lambda k.\begin{cases}f(k) & k<n \\ f(k+1) & k \geq n.\end{cases} \qquad g' = \lambda(k, s).\begin{cases}g(k, s) & k<n \\ g(k+1, s) & k \geq n.\end{cases}$$

By means of state and memory functions and the operators *skip*, *copy*, *consume* and *erase*, the following definition formally describes how a POA processes an input word. Note that memory functions represent the context in which the side condition of a transition is being evaluated.

To describe more complex algorithms in the following, I will use pseudo-code, which is a mixture of mathematical notation and the C programming language. From the point on where a pseudo-code function has been defined, this function may be reused in any other algorithm

20 Each alternative is uniquely identified by a natural number.

thereafter. Additionally, functions may be overloaded, just as it possible in the C^{++} programming language. Furthermore, any function may *throw* an exception to indicate a fatal error and to stop the computation immediately. Finally, it is possible to assign multiple variables at once, whereas the right-hand side expression is evaluated first, before the assignment is performed.

Definition 2.3.5 (Configuration, Step, Acceptance)

Let $P = (\Sigma, S, s_0, \delta, F)$ be a POA. A triple $(f, g, n) \in State \times Memory \times \mathbb{N}$ is a **configuration** of P. Iff f is the initial state function of P for a word $w \in \Sigma^*$ and g is the initial memory function of P, then $(f, g, 1)$ is the **initial configuration** for the word w. The triple (f, g, n) is a **final configuration**, iff $\forall i \in \mathbb{N}$ with $1 \leq i \leq n.\ f(i) = (s_i, v_i.w_i)$ and $s_i \in F$.

The **step** binary relation $\rhd_P \subseteq (State \times Memory \times \mathbb{N}) \times (State \times Memory \times \mathbb{N})$ transfers a configuration into another. The transition $(f, g, n) \rhd_P (f', g', n')$ is computed according to the pseudo-code function *step* that is defined as follows:

```
State × Memory × ℕ step (State × Memory × ℕ (f, g, n)) {
    let i, j ∈ ℕ with i = 1 and j = n;
    while (i ≤ j) {
        let (s, x.y) ∈ S × Y with (s, x.y) = f(i);
        if (s ∈ F) i = i + 1;
        else {
            let M = Ø ∈ ℘(δ);
            if (y ≠ ε)
                M = {(s, (a, q), t) ∈ δ | y[0] is an instance of the class a ∧ q(i, g) = true}
                    ∪ {(s, skip, t) ∈ δ}
                    ∪ {(s, ε, t) ∈ δ};

            if (M ≠ Ø) {
                let k, m ∈ ℕ with k = i and m = 1;
                foreach (s, c, t) ∈ M {
                    if (m++ < |M|) {
                        (n, k) = (n + 1, n + 1);
                        (f, g) = copy(i, k, f, g);
                    }
                    if (c = (a, q)) (f, g) = consume(k, t, f, g);
                    else if (c = skip) (f, g) = skip(k, t, f, g);
                    else f = λn.{ (t, x.y)   i = n
                                { f(n)       else.
                }
                i = i + 1;
            }
            else {
                (j, n) = (j – 1, n – 1);
                (f, g) = erase(i, f, g);
            }
    }
    }
    return (f, g, n);
}
```

The function *step* iteratively modifies the current alternatives. As long as an alternative does not reside in a final state of the POA P, the function determines all possible target states that can be reached under the current input symbol. If no state can be reached, *step* removes that alternative. Otherwise, the function creates as many copies of that alternative as necessary and modifies the state and memory function according to the taken transitions.

Iff f is the initial state function for a word $w \in \Sigma^*$ and g is the initial memory function such that there exists a series of configurations $(f, g, 1) \vartriangleright_P \ldots \vartriangleright_P (f', g', n)$, where (f', g', n) is a final configuration and $n > 0$, the automaton P **matches** w with (f', g', n). Iff x_i is a prefix of w and $\exists i \in \mathbb{N}$ with $1 \leq i \leq n.\ f'(i) = (s_i, z_i.y_i)$ such that $w = x_i y_i$, the automaton P **accepts** the prefix x_i of w.

The set $L_P = \{w \mid P \text{ accepts } w\}$ is the **accepted language** of P.

If a pattern matcher's predicate object automaton matches a word $w \in \Sigma^*$ with (f, g, n), the set $\{u_i v_i \mid f(i) = (s_i, u_i.v_i) \wedge 1 \leq i \leq n\}$ contains the residues of all possible alternatives. After having chosen one of the alternatives, the pattern matcher continues, processing the corresponding residue word, if that word is not the empty word.

Example 2.3.6

Let $\Sigma = \{a, b\}$, whereas the classes a and b are not correlated in any way. Additionally, let the POA $P = (\Sigma, \{s, t, u, v\}, s, \delta, \{v\})$ with δ as Figure III.2 shows. Note that the used transition guards always return *true*. The POA P matches any input word that begins with the symbol a and that ends with the symbol b, whereas P does not consume the inner symbols.

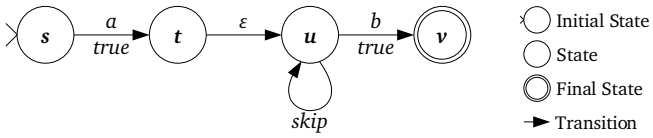

Figure III.2: Predicate object automaton that accepts the language $\{a\}\Sigma^*\{b\}$.

The automaton P matches the input word *aaabbb* with $(f, g, 3)$. Table III.3 shows the final state function f and memory function g, whereas the red marked symbols denote the symbols that P has skipped.

i	*f(i)*	*g(i, t)*	*g(i, u)*	*g(i, v)*	*Accepted Prefix*
1	(*v*, **aa**.*bb*)	*a*	**aa**	*b*	*a**aa**b*
2	(*v*, **aab**.*b*)	*a*	**aab**	*b*	*a**aab**b*
3	(*v*, **aabb**.)	*a*	**aabb**	*b*	*a**aabb**b*

Table III.3: Final state and memory function after matching aaabbb.

3. Pattern Matcher Theory

Besides the theoretic fundamentals that were discussed in Section 2, this section first covers additional necessary mechanisms, before it formally describes the pattern matcher.

3.1. Pattern

To spare the user to manually construct every POA of a pattern matcher, the following definition introduces three different pattern types, with which the user can easily specify the instruction patterns a POA should identify, and how the pattern matcher should replace the matched instructions.

Definition 3.1.1 (Item Pattern, Wildcard Pattern, Sequence Pattern)

A tuple $(a, q) \in \Sigma \times Predicate$ is an **item pattern** over Σ that describes an input symbol that is an instance of the class a and that satisfies the side condition q.

The symbol $*$ represents a **wildcard pattern** over Σ that denotes an unlimited number of input symbols. As a wildcard pattern does not consume input symbols, this pattern type can be used to skip uninteresting input symbols the pattern matcher should process later on.

A set $s = \{p_i \mid p_i \text{ is a pattern over } \Sigma \wedge 1 \leq i \leq n\}$ is a **sequence pattern** of length $n \in \mathbb{N}$ over Σ. A sequence pattern is either **ordered** (the patterns must be processed in the given order, first p_1, then p_2, etc.) or **unordered** (the patterns may be processed in any order). It immediately follows that an unordered sequence pattern of length n stands representatively for $n!$ different ordered sequence patterns. In the following, square brackets (i.e., $[p_1, ..., p_n]$) denote ordered sequence patterns, whereas curly brackets (i.e., $\{p_1, ..., p_n\}$) represent unordered sequences.

A pattern p is a **subpattern** of a sequence pattern s, iff $p \in s$ or iff p is a subpattern of any sequence pattern t with $t \in s$.

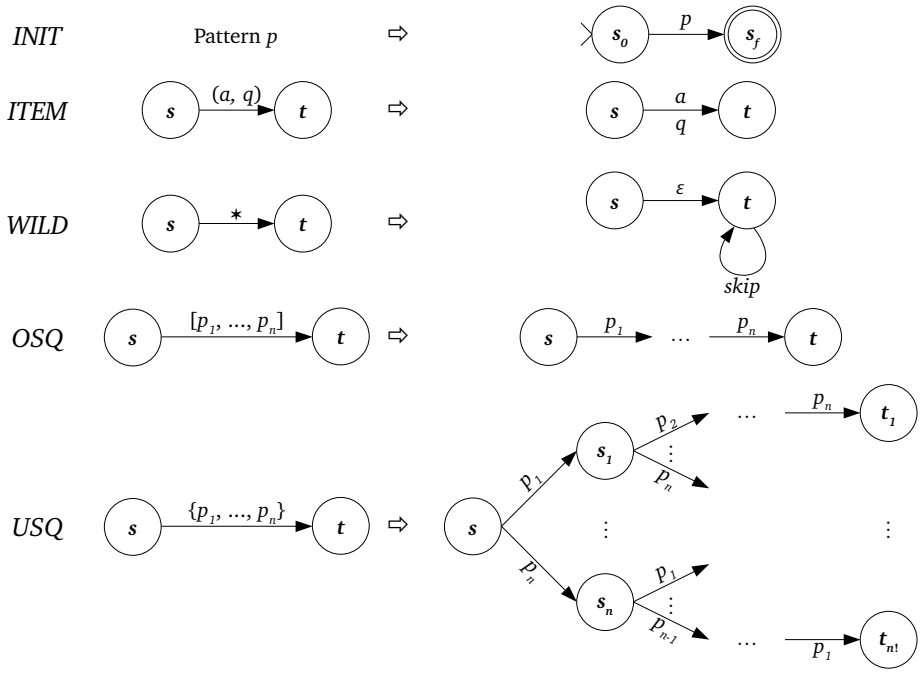

Figure III.3: Rules to generate a predicate object automaton for a pattern.

Each pattern type can be understood as a compact representation of a predicate object automaton. It turns out that there exists a POA for every pattern, but obviously not the other way round. Figure III.3 shows how to generate the corresponding automaton for a pattern p. To receive the final automaton, the rules must be applied as long as there exists an edge that is labelled with a pattern.

The construction rule *USQ* demonstrates how compact the pattern representation is. Because the unordered sequence on the left side abbreviates $n!$ different ordered sequences, the final predicate object automaton on the right side has to contain $n!$ mutually exclusive paths, not to match the same subpattern p_i twice. Because every subpattern has to appear on each path, the generated automaton consists of multiple states that are reached under exactly the same side condition. So, when the automaton processes an input word, the used memory function resembles a sparse matrix, because the automaton can only process the input word along a single path for every alternative. Each entry that does belong to a state that is not part of that path will simply remain unused (see Example .9).

In the worst case, using the generated POA thus results in an extensive waste of memory. To remedy this drawback, it is feasible to simulate the corresponding automaton for any pattern. However, the simulation requires additional mechanisms that are introduced in the following.

Example 3.1.2

Figure III.4 shows the construction of the corresponding predicate object automaton for the pattern [{(*a*, *true*), ∗}, (*b*, *true*)]. The automaton accepts those words whose prefix begins with *a* and ends with *b*, allowing an arbitrary number of symbols between the two symbols, or whose prefix starts with an arbitrary sequence of symbols and ends with *ab*.

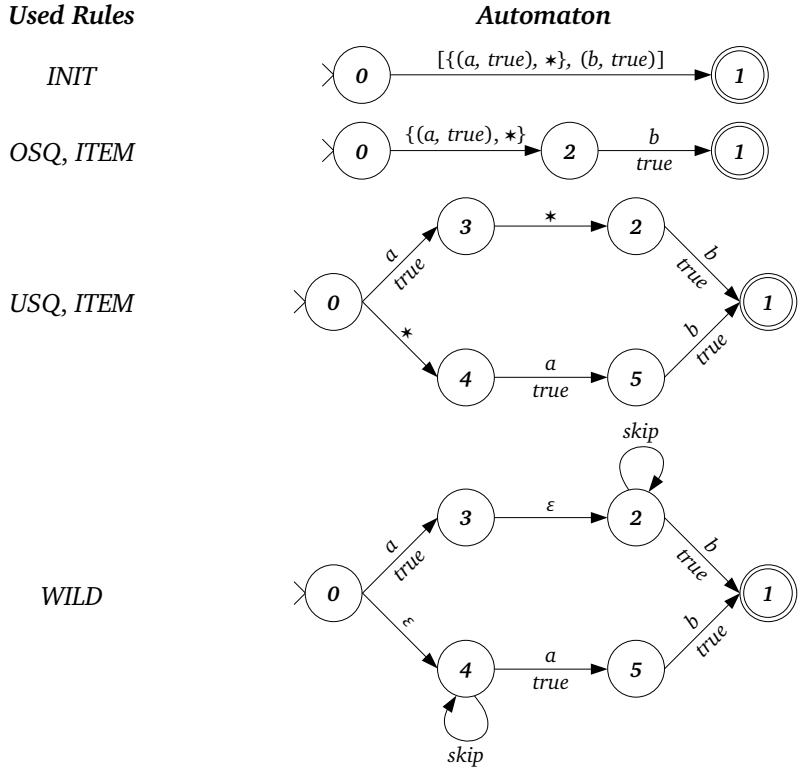

Figure III.4: Predicate object automaton for [{(*a*, *true*), ∗}, (*b*, *true*)].

To simulate the corresponding POA for an arbitrary pattern, the simulation has to select the patterns that should be matched next. Because sequence patterns only are pattern containers, the simulation only has to care about item and wildcard patterns. The following definition introduces the function *SUB* that computes the set of all subpatterns of a pattern, omitting all sequence patterns. Additionally, the definition introduces a method to access and to uniquely identify the subpatterns of a pattern. This mechanism is not required in the simulation, but of use to the pattern matcher, as demonstrated later on.

For the remainder of this chapter, the set *Pattern* abbreviates the set of all patterns, whereas the set *Sequence Pattern* denotes the set of all sequence patterns.

Definition 3.1.3 (SUB, Size, Index)

For any pattern p the function SUB: $Pattern \rightarrow \wp(Pattern)$ recursively computes the set of non-sequence subpatterns of p.

$$SUB(p) := \begin{cases} \bigcup_{q \in p} q & p \text{ is a sequence pattern} \\ \{p\} & \text{else.} \end{cases}$$

The number of non-sequence subpatterns a pattern p comprises determines the **size** of p. So, the size of the pattern p is defined as follows:

$$size(p) := |SUB(p)|.$$

Let s be a sequence pattern of length $n \in \mathbb{N}$ with $p_1, ..., p_n \in s$. For $1 \leq i \leq size(s)$, the function $pattern$: $\mathbb{N} \times Sequence\ Pattern \rightarrow Pattern$ determines the i-th pattern $p \in SUB(s)$. Because sequence pattern may be subpatterns of sequence patterns, the equation $pattern(i, s) = p_i$ does not hold necessarily. The function $pattern$ is defined as follows:

$$pattern(i, s) := \begin{cases} p_l \in s & \sum_{k=1}^{l-1} size(p_k) = i-1 \ \wedge \ p_l \text{ is not a sequence pattern} \\ pattern(i-m, p_l) & \text{else, with } m = \sum_{k=1}^{l-1} size(p_k) \ \wedge \ m \leq i \leq m+size(p_l). \end{cases}$$

The function $pattern$ assigns a unique number to any subpattern of a pattern, so that they can be discerned from each other. Iff $i \in \mathbb{N}$ with $pattern(i, s) = p$, then i is the **index** of p[21]. The index of a sequence pattern, is the index of its first non-sequence subpattern.

In the following, $s[i]$ abbreviates $pattern(i, s)$.

Example 3.1.4

Let $s = \{(b, true), *\}$ be an unordered sequence pattern and $t = [(a, true), s]$ be an ordered sequence pattern. Then, $SUB(t) = \{(a, true)\} \cup SUB(s) = \{(a, true), (b, true), *\}$, whereas $size(t) = 1+size(s) = 3$.

By definition of t and s, there are three different non-sequence patterns that the $pattern$ function makes accessible. So, $t[1] = (a, true)$, $t[2] = s[1] = (b, true)$ and $t[3] = t[2] = *$.

21 The indexes of the subpatterns of a pattern resembles the order in which they have been specified, as Example .4 shows. I have chosen this type of numbering to make the description of a pattern matcher as easy as possible (see Section 2 of Chapter IV). Nonetheless, a tree-like numbering would also have been possible, where e.g., 1.3 would denote the third subpattern of the first subpattern.

Similar to the *step* binary relation (see Definition .5), the simulation of the corresponding POA for an arbitrary pattern operates on a state and a memory function. However, in contrast to the automaton, the set of states used in the simulation will be the set of all non-sequence subpatterns of the pattern, whose corresponding POA is to be simulated. Memory functions (see Definition .3) are unfortunately not sufficient for this approach, because when processing wildcard patterns, the simulation requires an extended memory function that knows whether a wildcard pattern has finished to skip input symbols. This information is necessary, so that wildcard patterns only skip input symbols that are adjacent to each other.

Additionally, the simulation requires the auxiliary function *finish* that operates on extended memory functions. How the functions *skip*, *copy*, *consume* and *erase* (see Definition .4) operate on extended memory functions is implicitly clear.

Definition 3.1.5 (Extended Memory Function, Finish)

Let p be an arbitrary pattern, $S = SUB(p)$ the set of subpatterns of p. An **extended memory function** of p is a function of the type $\mathbb{N} \times S \to \Sigma^* \times B$. The function $\lambda(n, s) = (\varepsilon, false)$ is the **initial extended memory function**. In the following, *ExMemory* denotes the set of all extended memory functions.

Let g be an extended memory function of p, q an arbitrary subpattern of p – this includes sequence patterns that are not contained in S – and $i \in \mathbb{N}$. The pattern q is **finished** for the i-th alternative, iff $g(i, q') \in \Sigma^* \times \{true\}$ for every $q' \in SUB(q)$. So, a pattern is finished, iff itself and all of its subpatterns are finished.

The function $finish: \mathbb{N} \times S \times ExMemory \to ExMemory$ marks a pattern s for the alternative n as finished. The result $g' = finish(n, s, g)$ is then defined as:

$$g' = \lambda(k, s).\begin{cases}(w, true) & k{=}n \wedge s{=}t \\ g(k, s) & \text{else.}\end{cases}$$

The following definitions introduces three functions that are essential to the simulation of a POA. The function *ITEM* determines the amount of item patterns that have not yet consumed an input symbol. The simulation uses this function to determine whether a wildcard pattern may continue skipping input symbols. If e.g., three input symbols are left and three item patterns still have to match an input symbol, it would not make sense to skip any more symbols. Additionally, functions *FIRST* and *FOLLOW* enable the simulation to determine the patterns that should be matched next.

Definition 3.1.6 (FIRST, FOLLOW, ITEM)

Let p be an arbitrary pattern, q be an arbitrary subpattern of p, $S = SUB(p)$ be the set of non-sequence subpatterns of p and g be the used extended memory function of p.

The function $FIRST_g: \mathbb{N} \times Pattern \to \wp(S)$ determines for any alternative $i \in \mathbb{N}$ the first non-sequence subpatterns of q that have to be processed next:

- If q is a non-sequence pattern, there are two possibilities:
  - Iff q is finished for the i-th alternative, $FIRST_g(i, q) := \emptyset$.
  - Otherwise, $FIRST_g(i, q) := \{q\}$.
- Iff $q = [q_1, ..., q_n]$ is an ordered sequence pattern, $FIRST_g(i, q) := FIRST_g(i, q_1)$.
- Iff $q = \{q_1, ..., q_n\}$ is an unordered sequence pattern, $FIRST_g(i, q) := \bigcup_{q_j \in q} FIRST_g(i, q_j)$.

The function $FOLLOW_g$: $\mathbb{N} \times Pattern \to \wp(S)$ determines for any alternative $i \in \mathbb{N}$ all unfinished subpatterns of p that follow the subpattern q:

Iff q is a wildcard pattern that is not finished for the i-th alternative, let $Q = \{q\}$. Otherwise, let $Q = \emptyset$.

- If $q \in s$ with $s = [q_1, ..., q_n]$ an ordered sequence subpattern of p, there are two possibilities:
  - Iff $\exists j \in \mathbb{N}$ with $1 \leq j < n$ such that $q = q_j$, $FOLLOW_g(i, q) \coloneqq FIRST_g(i, q_{j+1}) \cup Q$.
  - Otherwise, $q = q_n$, so $FOLLOW_g(i, q) \coloneqq FOLLOW_g(i, s) \cup Q$.
- If $q \in s$, where s is an unordered sequence subpattern of p, there are again two possibilities:
  - Iff s is finished for the i-th alternative, $FOLLOW_g(i, q) \coloneqq FOLLOW_g(i, s)$.
  - Otherwise, some subpatterns of the sequence pattern s are still unfinished for the i-th alternative, so $FOLLOW_g(i, q) \coloneqq \bigcup_{\substack{q_j \in s \\ q_j \neq q}} FIRST_g(i, q_j) \cup Q$.
- In any other case $FOLLOW_g(i, q) \coloneqq Q$.

The function $ITEM_g$: $\mathbb{N} \times Pattern \to \wp(S)$ determines those item patterns in S that are not yet finished, so for any $i \in \mathbb{N}$, $ITEM_g(i, p) \coloneqq \{s \in S \mid s \text{ is an item pattern} \wedge s \text{ is not finished for } i\}$.

Example 3.1.7

Let $p = \{(a, true), \star, (b, true)\}$ be an unordered sequence pattern, g be an extended memory function of p, where $g(2, p[1]) = (a, true)$, $g(2, p[2]) = (aa, false)$ and $g(2, p[3]) = (\varepsilon, false)$, and $w = b$ the remaining suffix of the original input word. The current active pattern is the wildcard pattern $p[2]$. Using the previously introduced functions, the simulation decides how to proceed in the second alternative as follows:

- The algorithm first has to determine the subpatterns of p to continue with. It follows that $FOLLOW_g(2, p[2]) = FIRST_g(2, p[1]) \cup FIRST_g(2, p[3]) \cup \{p[2]\} = \{p[2], p[3]\}$. The wildcard pattern $p[2]$ appears in that set, because the pattern is not yet finished, as well as the item pattern $p[3]$.
- For each pattern in $FOLLOW_g(2, p[2])$, the algorithm would create a new alternative. However, a closer look at the remaining input symbols – there is only one – and the number of unfinished item patterns $ITEM_g(2, p) = \{p[3]\}$ reveals that it is senseless to let the wildcard pattern $p[2]$ to skip any more items, because there would then be no more input symbol left for $p[3]$.
- Finally, the algorithm decides to finish the wildcard pattern and modifies the memory function such that $g(2, p[2]) = (aa, true)$ and $g(2, p[3]) = (b, true)$. Thus, the second alternative represents a successful match.

The following definition introduces the notion a pattern configuration that comprises a state function, an extended memory function and the number of currently active alternatives. With the above introduced mechanisms a pattern configuration can be transferred into another. As extended memory functions represent the context in which the boolean predicate function of an item pattern is being executed, the boolean predicate function of any item pattern must be of the type $\mathbb{N} \times ExMemory \to B$ from now on.

Definition 3.1.8 (Pattern Configuration, Pattern Step, Pattern Acceptance)
Let p be an arbitrary pattern and $S = SUB(p) \cup \{s_0\}$ the set of states such that $s_0 \notin SUB(p)$. A triple $(f, g, n) \in State \times ExMemory \times \mathbb{N}$ is a **pattern configuration**. Iff f is the initial state function for an arbitrary input word $w \in \Sigma^*$ and g is the initial extended memory function for p, the triple $(f, g, 1)$ is the **initial pattern configuration** for w. A triple (f, g, n) is a **final pattern configuration**, iff $\forall i \in \mathbb{N}$ with $1 \leq i \leq n, f(i) = (s_i, u_i.v_i)$ such that $FOLLOW_g(i, s_i) = \emptyset$.

The **pattern step** binary relation $\rhd_p \subseteq (State \times ExMemory \times \mathbb{N}) \times (State \times ExMemory \times \mathbb{N})$ transfers a pattern configuration into another. The transition $(f, g, n) \rhd_p (f', g', n')$ is computed according to the pseudo-code function *step* that is defined as follows:

```
State × ExMemory × ℕ step (State × ExMemory × ℕ (f, g, n)) {
    let i, j ∈ ℕ with i = 1 and j = n;
    while (i ≤ j) {
        let (s, x.y) ∈ S × Y with (s, x.y) = f(i);
        let F ∈ ℘(S) with F = FIRST_g(i, p) if s = s_0 and F = FOLLOW_g(i, s) else;

        if (F = ∅) i = i + 1;
        else {
            let M = ∅ ∈ ℘(S);
            if (y ≠ ε)
                M = {(a, q) ∈ F | y[0] is an instance of a ∧ q(i, g) = true} ∪ {∗ ∈ F};

            if (M ≠ ∅) {
                let k, m ∈ ℕ with k = i and m = 1;
                foreach t ∈ M {
                    if (m++ < |M|) {
                        (n, k) = (n + 1, n + 1);
                        (f, g) = copy(i, k, f, g);
                    }
                    if (t = (a, q)) (f, g) = consume(k, t, f, g);
                    else if (|y| > ITEM_g(i, p)) {
                        n = n + 1;
                        (f, g) = copy(k, n, f, g);
                        (f, g) = skip(n, t, f, g);
                    }
                    g = finish(k, t, g);
                }
                i = i + 1;
            }
            else {
                (j, n) = (j – 1, n – 1);
                (f, g) = erase(i, f, g);
            }
        }
    }
    return (f, g, n);
}
```

The function *step* iteratively processes the current active alternatives. Similar to the transition function introduced in Definition .5, the above function determines the set of target states that can be reached with the current input symbol. If there are no target states for an alternative, the alternative is finished. Otherwise, the *step* function selects all those states, whose side condition is satisfied. If there are no such states, the algorithm erases the corresponding alternative. Otherwise, the *step* function copies the current alternative with respect to the number of available target states.

The main difference to the POA step relation is, that the states, on which the pattern step relation operates, are the subpatterns of the pattern p, whose corresponding POA is to be simulated. Additionally, the above algorithm behaves more intelligent than the POA *step* function, because the above function only allows wildcard patterns to consume any more input symbols, if there are enough symbols left for the remaining item patterns.

Iff f is the initial state function for a word $w \in \Sigma^*$ and g is the initial extended memory function of p such that there exists a series of pattern configurations $(f, g, 1) \vartriangleright_p \ldots \vartriangleright_p (f', g', n)$, where (f', g', n) is a final pattern configuration and $n > 0$, the pattern p **matches** the input word w with (f', g', n). Iff x_i is a prefix of w and $\exists i \in \mathbb{N}$ with $1 \leq i \leq n$ and $f'(i) = (s_i, z_i.y_i)$ such that $w = x_i y_i$, the pattern p **accepts** the prefix x_i of w.

The set $L_p = \{w \mid p \text{ accepts } w\}$ is the **accepted language** of p.

Example 3.1.9

Let $\Sigma = \{a, b, c\}$ be an alphabet, where the classes a, b, and c are not derived from one another, and $p = \{(a, true), (b, true), (c, true)\}$ be an unordered sequence pattern. Because every item pattern can only consume exactly one symbol of Σ, the accepted language of the pattern p is $L_p = \{abc, acb, bac, bca, cab, cba\}$. Figure III.5 shows the corresponding POA P for the pattern p.

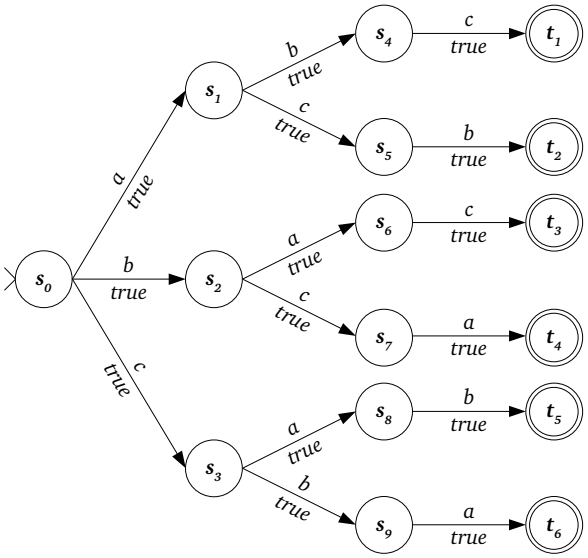

Figure III.5: Corresponding POA for {(a, true), (b, true), (c, true)}.

When processing any input word, the POA P can only take one path for every alternative. Thus, the used memory function g resembles a sparse matrix of which only 3 of 16 cells are used for every alternative (there is always only one alternative in this example), as Table III.4 demonstrates. The table shows the function values of the memory function g after the POA P has matched the input word *cba*.

$g(1, s_0)$	$g(1, s_1)$	$g(1, s_2)$	$g(1, s_3)$	$g(1, s_4)$	$g(1, s_5)$	$g(1, s_6)$	$g(1, s_7)$
ε	ε	ε	**c**	ε	ε	ε	ε
$g(1, s_8)$	$g(1, s_9)$	$g(1, t_1)$	$g(1, t_2)$	$g(1, t_3)$	$g(1, t_4)$	$g(1, t_5)$	$g(1, t_6)$
ε	**b**	ε	ε	ε	ε	ε	**a**

Table III.4: Memory function g after matching the word cba.

So, the major disadvantages of the corresponding POA *P* are the extreme waste of memory of the memory function (only 18.75% are used) and the different memory function entries that correspond to the same item pattern. In this example, $g(1, s_a)$ with $s_a \in \{s_1, s_6, s_8, t_4, t_6\}$ might store the matched input symbol that belongs to the item pattern $p[1]$.

Both disadvantages can be overcome, by simulating the POA *P* according to Definition .8. Matching any valid input word, the simulation of *P* will always produce an extended memory function *g'* that resembles the matrix that Table III.5 shows.

$g'(1, s_0)$	$g'(1, p[1])$	$g'(1, p[2])$	$g'(1, p[3])$
(ε, *false*)	(*a*, *true*)	(*b*, *true*)	(*c*, *true*)

Table III.5: Extended memory function g' after matching a valid input word.

Although the simulation means an increase in runtime (the $FOLLOW_g$ set must be recomputed after every step), the achieved advantages, such as minimal memory consumption, certainly outweigh the (minimal) loss in speed.

3.2. Rule

As hinted at the beginning of this chapter, a pattern matcher comprises profiles, which are finite sets of rules. A rule is dedicated to a certain kind of problem, such as the compilation of a binary operation. In this approach, four properties define the behaviour of a rule. The search pattern describes a sequence of instructions the rule should replace, whereas the replace pattern determines how the rule should substitute the matched instruction sequence. Additionally, the rule is defined through a cost function, which assigns a cost to each alternative of a successful match, and a global condition function, which checks global properties of the matched instruction sequence.

In the following, Σ_{in} denotes an arbitrary alphabet that comprises instruction classes that the symbols of any input word instantiate, which the search pattern of a rule matches, are instances of, whereas Σ_{out} denotes an arbitrary alphabet of instruction classes of which the output symbols of a rule's replace pattern are instances.

Definition 3.2.1 (Search Pattern, Replace Pattern)

Any sequence pattern s over Σ_{in}, where $SUB(s)$ comprises at least one item pattern over Σ_{in}, is a **search pattern** over Σ_{in}. Search patterns match instruction sequences a rule replaces later on. In the following, *Search Pattern* denotes the set of all search patterns.

Any function r: $\mathbb{N} \times ExMemory \rightarrow \Sigma_{out}^{*}$ is a **replace pattern** over Σ_{out} that produces a sequence of instructions for an alternative and an extended memory function. In the following, the set *Replace Pattern* denotes the set of all replace patterns.

Search patterns must contain at least one item pattern, to guarantee that the pattern matcher eventually terminates. If a pattern matcher would always chose rules, whose search pattern only consists of wildcard patterns, the input word would not be modified (wildcard patterns do not consume input symbols), which causes the pattern matcher to loop forever. Because item patterns consume an input symbol – under the assumption that their side condition is satisfied – there will be no more input symbols left after a finite number of iterations.

Search patterns differ additionally from usual patterns with respect to their way of processing an input word. When a search pattern matches an input word, the first pattern to match must always be an item pattern. If there is only a wildcard pattern to begin with, that pattern will be directly marked as finished. This procedure continues, until the search pattern encounters an item pattern. Thus, the search patterns $[*, (a, true), (b, true)]$ and $[(a, true), (b, true)]$ are equivalent. The main reason for this matching behaviour is to clearly define at which location a rule will insert its replace pattern. This is especially important, when a pattern matcher optimises a basic block.

Consider the instruction alphabet $\Sigma_{in} = \Sigma_{out} = \{SIN, COS, SCS\}$, where *SIN* is the class of sine instructions, *COS* denotes the class of cosine instructions and *SCS* represents the class of instructions that computes sine and cosine in parallel. Let $s = \{(SIN, true), *, (COS, true)\}$ be a search pattern and $r = \lambda(n, g).SCS$ be a replace pattern that substitutes the instruction sequence that s matches. The basic idea is that the replace pattern r will be inserted at the position where the input symbol resides that the first item subpattern of s matches. So, virtually, the instruction that the second item pattern matches will be pushed upwards past the wildcard pattern $s[2]$, because the pattern matcher assumes that the replace pattern r logically combines the instructions that the item patterns $s[1]$ and $s[3]$ match. Naturally, this behaviour requires the pattern matcher to verify whether it is valid push instructions upwards, as Example .6 demonstrates.

For efficiency reasons, each search pattern s automatically marks all wildcard patterns as finished, if every item pattern has matched an input symbol (i.e., $ITEM_g(i, s) = \emptyset$, for any $i \in \mathbb{N}$ and the used extended memory function g). Thus, the search pattern $[(a, true), (b, true), *]$ is equivalent to $[(a, true), (b, true)]$.

Example 3.2.2

Let $\Sigma = \{a, b, c\}$ be an alphabet and $s = \{(a, true), *, (b, true)\}$ be a search pattern over Σ. Although s is an unordered sequence pattern, the above restrictions force s to only accept input words that either begin with a and end with b or the other way round. So, s abbreviates the ordered sequence patterns $[(a, true), *, (b, true)]$ and $[(b, true), *, (a, true)]$. Table III.6 shows the resulting alternatives, after the search pattern s has matched the input word *babac* with the configuration $(f, g, 2)$, whereas the red marked symbols denote the symbols that the wildcard pattern $s[2]$ has skipped.

i	*f(i)*	*g(i, s[1])*	*g(i, s[2])*	*g(i, s[3])*	*Accepted Prefix*
1	(s[1], *.bac*)	*(a, true)*	(**ε**, true)	*(b, true)*	*ba*
2	(s[1], ***ab***.*c*)	*(a, true)*	(***ab***, *true*)	*(b, true)*	*b**ab**a*

Table III.6: Pattern configuration after matching babac.

After the search pattern of a rule has matched an input word, the rule first verifies whether each generated alternative satisfies the rule's global condition and assigns a cost to each valid alternative. Based on this information, the rule selects one of the alternatives that determines the outcome of the replace pattern. After the rule has inserted the replace pattern's value, the pattern matcher determines the residue ofthe input word to process next.

The following definition introduces mechanisms that are required to formally describe a rule and its functioning.

Definition 3.2.3 (Condition Function, Cost Function, Residue, Check)

Let s be any search pattern over Σ_{in}, $S = SUB(s)$ be the set of non-sequence subpatterns of s, $B = \{true, false\}$ be the set of boolean constants and (f, g, n) be a final pattern configuration for an arbitrary input word.

Any function $\mathbb{N} \times ExMemory \rightarrow B$ is a **condition function** that verifies whether an alternative satisfies a global side condition. In the following, *Condition* abbreviates the set of condition functions.

Any function $\mathbb{N} \times ExMemory \rightarrow \mathbb{Z}$ is a **cost function** that assigns an integer cost to an alternative depending on the matched input symbols. The cost of an alternative may be negative to indicate that an optimisation is better than another (the lower the cost the better). In the following, *Cost* denotes the set of all cost functions.

The function *residue*: $\mathbb{N} \times State \rightarrow \Sigma_{in}^*$ determines the residue word of an arbitrary alternative. For any $i \in \mathbb{N}$, $residue(i, f) := x_i y_i$, iff $f(i) = (s, x_i.y_i)$. The residue word is a remainder of the original input word the pattern matcher must process next.

The function *check*: $(State \times ExMemory \times \mathbb{N}) \times Condition \rightarrow State \times ExMemory \times \mathbb{N}$ operates on a pattern configuration by removing all alternatives that do not satisfy the given condition. The function behaves as the following pseudo-code function:

$State \times ExMemory \times \mathbb{N}$ *check* $((State \times ExMemory \times \mathbb{N})\ (f, g, n), Condition\ c)$ {
  let $i \in \mathbb{N}$ with $i = 1$;
  while $(i \leq n)$
    if $(c(i, g) = false)$ {
      $(f, g) = erase(i, f, g)$;
      $n = n - 1$;
    }
    else
      $i = i + 1$:
  return (f, g, n);
}

Example 3.2.4

Let $\Sigma = \{a, b\}$ be an alphabet, where the classes a and b are not correlated with each other, $s = [(a, true), \star\ (b, true)]$ be a search pattern over Σ, $w = abb$ be an input word and $(f, g, 2)$ be the final pattern configuration for w, as Table III.7 shows. The red marked symbols denote the input symbols the wildcard pattern $s[2]$ has skipped.

i	*f(i)*	*g(i, s[1])*	*g(i, s[2])*	*g(i, s[3])*	*Accepted Prefix*
1	(s[3], .b)	*(a, true)*	(**ε**, true)	*(b, true)*	*ab*
2	(s[3], **b**.)	*(a, true)*	(**b**, *true)*	*(b, true)*	*a**b**b*

Table III.7: Final pattern configuration after matching abb.

Additionally, let $c = \lambda(n, g).g(n, s[2]) = (b, true)$ be a condition function. The function *check* will remove the first alternative, because $g(1, s[2]) = (\varepsilon, true) \neq (b, true)$, so only the second alternative remains. Thus, $check((f, g, n), c) = (f', g', 1)$, where $f'(1) = f(2)$ and $g'(1) = g(2)$.

The residue word to continue with is then $residue(1, f') = b$, where b is the input symbol that $s[2]$ has skipped before.

Using the above introduced mechanism, the following definition formally describes rules and their functioning.

Definition 3.2.5 (Rule)

A quadruple $r = (sp, q, c, rp) \in Search\ Pattern \times Condition \times Cost \times Replace\ Pattern$, where sp is a search pattern over Σ_{in} and rp is a replace pattern over Σ_{out}, is a **rule** for the input alphabet Σ_{in} and the output alphabet Σ_{out}.

Both the search pattern sp and the condition function q determine the requirements before the rule may be applied. So, iff sp matches an input w over Σ_{in} with (f, g, n) and there exists at least one alternative that satisfies the rule's side condition – $check((f, g, n), q) = (f', g', m)$, where $m \neq 0$ – the rule r matches the input word w with (f', g', m).

Both cost function c and replace pattern r determine the behaviour of the rule, after the rule has matched an input word. A rule generally selects an alternative with the cheapest cost, before generating an output word over Σ_{out} with respect to the replace pattern and the previously selected alternative.

In the following *Rule* denotes the set of all rules. For any $r \in Rule$, $search(r)$ stands represent-

atively for the search pattern of the rule r, whereas $replace(r)$ denotes for the rule's replace pattern. Additionally, $condition(r)$ and $cost(r)$ represent the condition and cost function of the rule r.

The function $cheapest$: $Rule \times (State \times ExMemory \times \mathbb{N}) \rightarrow \mathbb{N}$ makes use of the cost function of a rule to determine a **cheapest alternative** of a final pattern configuration. If more than one alternative has the cheapest cost, the function *cheapest* selects the alternative with the lowest number.

$$cheapest(r, (f', g', m)) \coloneqq min\ \{i \mid cost(r)(i, g') = min\ \{cost(r)(j, g') \mid 1 \leq j \leq m\}\}.$$

If $i = cheapest(r, (f', g', m))$, then $c(i, g')$ is the **cost** of the rule r, whereas $residue(i, f')$ is the **residue** of the rule r.

Example 3.2.6

Let $\Sigma_{in} = \{+, *\}$ and $\Sigma_{out} = \{ADD, MUL, MAD\}$ be instruction alphabets, whereas each instruction of both alphabets has the three properties *target*, *first* and *second*, and *MAD* has the additional property *third*. Any instruction can be instantiated with a constructor, that receives the properties of the instruction as arguments. The expression $ADD(a, b, c)$ creates an *ADD*-instruction, where a is the target, b and c are the first and second operand of the instruction. The other instructions can be instantiated similarly. Any property of an instruction can be accessed with the dot-operator. For example, if $a \in \Sigma_{in}$, then $a.target$ denotes the target of the instruction instance a. The other properties can be accessed similarly.

An instance of $+ \in \Sigma_{in}$ is written as `a = b+d`, where `a` denotes the *target* variable and `b` and `c` are the *first* and *second* operand of the instruction. Instances of $* \in \Sigma_{in}$ are represented similarly. An *ADD*-instruction is written as `ADD a,b,c`, whereas `a` denotes the target register, `b` and `c` represent the *first* and *second* operand. Instances of *MUL* and *MAD* instructions are written analogously, whereas an *MAD* instruction is represented as `MAD a,b,c,d`, where `d` denotes the *third* operand. The value of the target register of an *MAD* instruction is then `a = (b*c)+d`.

Two rules are sufficient to translate any basic block over Σ_{in} into a basic block over Σ_{out}. The rule *add* translates a +-instruction into its corresponding counterpart, whereas search pattern, condition function, cost function and replace pattern of the rule *add* are defined as follows:

$$\begin{aligned}
search(add) &\coloneqq s_{add} = [(+, true)] \\
condition(add) &\coloneqq true \\
cost(add) &\coloneqq 1 \\
replace(add) &\coloneqq \lambda(n, g).ADD(+.target, +.first, +.second) \text{ where } g(n, s_{add}[1]) = (+, true).
\end{aligned}$$

Translating a $*$-instruction into a *MUL*-instruction, the rule *mul* functions similarly.

Let $v = *{+}{+}{+}$ be a source basic block over Σ_{in} and $w = \varepsilon$ a target basic block over Σ_{out}. The different colours denote different instruction instances. Both rules *add* and *mul* iteratively compile the source basic block v. Because $v[0]$ is an instance of the $*$-instruction, only the rule *mul* matches v with $(f_{mul}, g_{mul}, 1)$, where 1 is the cost and $+{+}{+}$ is the residue of *mul*.

So, *mul* modifies the target basic block such that $w = replace(mul)(1, g_{mul}) = MUL$. Only the rule *add* is able to match the remainder of the source basic block w, because the remainder only comprises instances of $+ \in \Sigma_{in}$. Analogously, the rule *add* then compiles the residue of the source basic block, which results in the target basic block, as Figure III.6 shows.

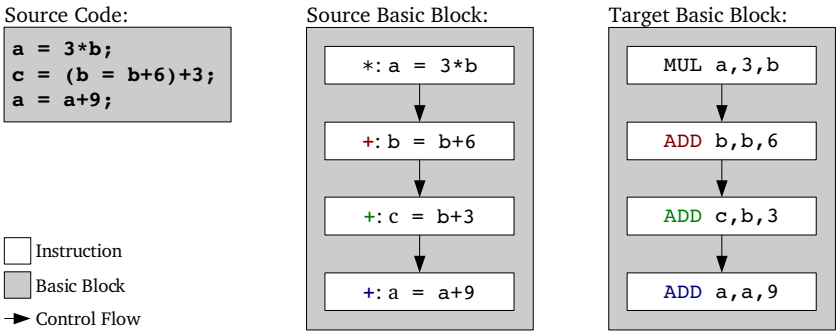

Figure III.6: Code generation with rules.

A closer look at the target basic block reveals that the generated code is not optimal. The first and the last instruction can be combined in the rule *mad* that is defined as follows:

$search(mad) \quad := s_{mad} = [(MUL, true), *, (ADD, q)]$

$condition(mad) \quad := true$

$cost(mad) \quad := -1$

$replace(mad) \quad := \lambda(n, g).MAD(MUL.target, MUL.first, MUL.second, ADD.second)$, whereas $g(n, s_{mad}[1]) = (MUL, true)$ and $g(n, s_{mad}[3]) = (ADD, true)$.

The side condition q of the item pattern $s_{mad}[3]$ checks whether the target of the matched *MUL*-instruction is both the target and the operand of the matched *ADD*-instruction. It is necessary to check whether the target of *MUL*-instruction and the *ADD*-instruction are the same, so that the rule *mad* may remove both matched instructions and replace them by an *MAD*-instruction.

The rule *mad* matches the previously generated target basic block $w = MULADDADDADD$ with (f_{mad}, g_{mad}, 1), where 1 is the cost and is $ADDADD$ the residue of *mad*. Figure III.7 shows the optimised basic block $u = replace(mad)(1, g_{mad}).ADDADD = MADADDADD$.

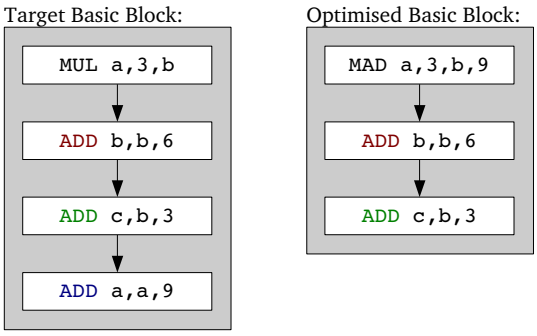

Figure III.7: Code optimisation with rules.

However, it is not valid to apply the above optimisation rule in every case. If the register `a` is either read or written between the two matched instructions, the rule mad would wrongly modify the basic block. To prevent this mistake, the condition of the rule *mad* should actually be the following:

$condition(mad) \quad := \lambda(n, g)$.the inner instructions x do neither read nor modify $ADD.target$, where $g(n, s_{mad}[2]) = (x, true)$ and $g(n, s_{mad}[3]) = (ADD, true)$.

This problem always occurs whenever a wildcard pattern appears in a search pattern. So, the pattern matcher generator generates rules that implicitly check these side conditions, which prevents the user from making errors and the pattern matcher from generating invalid basic blocks. Additionally, the pattern matcher does thus not force the user to repeat the same condition over and over again.

The following section introduces a formal description of a pattern matcher and describes its functioning in detail.

3.3. Pattern Matcher

The above introduced concepts make it finally possible to formally describe the functioning of a pattern matcher. To generate or optimise an input basic block, a pattern matcher operates a multitude of rules that are separated into profiles – one per supported target architecture.

Making use of a pattern matcher, which processes input basic blocks automatically, a user no longer has to cope with low-level algorithms that search and replace instruction sequences. Instead, the user describes the code generation and optimisation process on a higher level of abstraction in terms of rules and profiles that determine the pattern matcher's behaviour.

Definition 3.3.1 (Profile)

A finite set of rules $p_{in,out} = \{r_1, ..., r_n\}$, where $\forall i \in \mathbb{N}$ with $1 \leq i \leq n$, r_i is a rule for the input alphabet Σ_{in} and the output alphabet Σ_{out}, is a **profile** that translates basic blocks over Σ_{in} into basic blocks over Σ_{out}. Iff $\Sigma_{in} = \Sigma_{out}$, $p_{in,out}$ is an **optimisation profile**.

A pattern matcher comprises several profiles, each of which is dedicated to a target architecture. In general, a compiler makes use of two pattern matchers, whereas the first one handles the code generation, and the second one optimises the generated code afterwards.

Definition 3.3.2 (Pattern Matcher)

Let Σ_{in} be an input alphabet and $\Sigma_1, ..., \Sigma_n$ be different output alphabets. Any finite set of profiles $PM = \{p_{in,i(1)}, ..., p_{in,i(m)}\}$, where i: $\{1, ..., m\} \rightarrow \{1, ..., n\}$ is a surjective index function, is a **(retargetable) pattern matcher** that compiles a basic block over Σ_{in} into a basic block over any Σ_j with $j \in \{1, ..., n\}$.

There are basically two different methods how a pattern matcher processes a basic block. On the one hand, the pattern matcher can compile an input basic block in a single pass. This way of processing an input basic block requires that after a finite number of iterations the rules of the selected profile consume every instruction of the input basic block, so that the pattern matcher eventually has processed the whole basic block. If there exists an instruction that the used rules cannot consume, the pattern matcher is unable to compile the corresponding input basic block. Obviously, this *single-pass* processing method realises the code generation. On the other hand, the pattern matcher can make use of an optimisation profile to optimise an input basic block. Because not every instruction sequence can be optimised, the rules of the selected profile are not required to consume every instruction. As long as at least one rule matches (partitions of) the basic block, the pattern matcher continues to optimise the input basic block. Because the performed optimisations may result in new instruction sequences the pattern matcher could optimise, the pattern matcher generally processes the input basic block in *multiple* passes.

Both modes of operation make use of the costs of rules that match the input basic block. If several rule match the input basic block with multiple alternatives, the pattern matcher might always choose the cheapest rule (i.e., the rule with the cheapest alternative of all rules that match the basic block). However, just like every greedy algorithm, the pattern matcher might apply a sequence of rules whose total sum of costs is higher than the actual possible cost minimum. Thus, if the pattern matcher always takes the *local cost minimum*, it might miss the *global cost minimum*. So, the user has to determine whether the pattern matcher should determine the local cost minimum or compute the global cost minimum. However, as Chapter V demonstrates, the pattern matcher might not always generate more optimal basic blocks, while determining the global cost minimum.

Besides the cost minimum the pattern matcher should aim at, the user may also determine whether the pattern matcher should always take the first matching rule and omit the other rules. This *first-match* policy obviously has a positive impact on the runtime, but might also cause the pattern matcher to produce less optimal results. In the following, the first-match policy is not taken into account, and it is assumed that the pattern matcher always attempts to match the input basic block with all profile-specific rules.

The following two sections introduce the single- and multi-pass processing mode and point out the modifications in their implementation that are required to enable the pattern matcher to determine the local and the global cost minimum.

For the remainder of this chapter, let $PM = \{p_1, \ldots p_m\}$ be a pattern matcher for the input alphabet Σ_{in} and the output alphabets $\Sigma_1, \ldots, \Sigma_n$.

3.3.1. Single-Pass Matching Mode

To successfully compile an input basic block in a single pass, the rules of the used profile have to consume the input basic block completely. If that is not possible, the pattern matcher has encountered a sequence of instructions that none of the used rules accepts. However, in this case, it might still be possible to successfully process the whole basic block, if the pattern matcher previously had to choose from multiple alternatives. So, the pattern matcher stores the alternatives that a rule produces in a *match*, to enable the pattern matcher to backtrack.

Definition 3.3.3 (Match)
A tuple $(r, (f, g, n)) \in Rule \times (State \times ExMemory \times \mathbb{N})$ is a **match** of the rule r that matched an input basic block with (f, g, n). In the following *Match* denotes the set of all matches.

The **cost** of the match m is the cost of the rule r.

$$cost(m) \coloneqq cost(r)(cheapest(r, (f, g, n))).$$

The **residue** of the match m is the residue of the rule r.

$$residue(m) \coloneqq residue(cheapest(r, (f, g, n)), f).$$

When aiming at the local cost minimum, the pattern matcher has to apply a rule that belongs to a match with the cheapest cost. The function $cheapest\colon \wp(Match) \to Match$ determines a cheapest match out of a set of matches. If several matches are possible, the function always chooses the first available match. Let $M \in \wp(Match)$ be a set of matches.

$$cheapest(M) \coloneqq m \in M \text{ with } cost(m) = min\ \{cost(m') \mid m' \in M\}.$$

As hinted above, the pattern matcher might not be able to process the input basic block any further. In this case, the pattern matcher uses the function $next\colon \wp(Match) \to \wp(Match)$ that removes the cheapest alternative of the cheapest match. If this match only comprises a single alternative, the function completely removes that match from the given set of matches.

Let $M \in \wp(Match)$ be a set of matches, and $m = (r, (f, g, n)) = cheapest(M)$, if $M \neq \emptyset$. The function *next* is then defined as follows:

$$next(M) \coloneqq \begin{cases} \emptyset & M = \emptyset \\ M \setminus \{m\} & n = 1 \\ M \setminus \{m\} \cup (r, (f', g', n-1)) & \text{else, with } (f', g') = erase(n, f, g). \end{cases}$$

Algorithm: Compile basic block with respect to the local cost minimum (*processSingleLocal*).
Input: Basic block over Σ_{in}.
Output: Basic block over desired target alphabet Σ_{out}.

```
let p_{in,out} ∈ PM;

℘(Match) matchRules (Σ*_in u) {
    return {(r, (f, g, n)) | r ∈ p_{in, out} matches u with (f, g, n)};
}

Σ*_out processSingleLocal (Σ*_in u) {
    if (u = ε)
        return ε;

    let Q = ε ∈ ℘(Match)*;
    let q = Ø ∈ ℘(Match);
    let v ∈ Σ*_in with v = u;
    do {
        q = matchRules(v);
        if (q ≠ Ø)
            Q = Qq;
        else {
            while (q = Ø ∧ Q ≠ ε) {
                q = next(Q[|Q|]);
                Q = Q[1:|Q|-1];
                if (q ≠ Ø)
                    Q = Qq;
            }
            if (Q = ε)
                throw exception;
        }
        v = residue(cheapest(q));
    } while (v ≠ ε);

    let w = ε ∈ Σ*_out;
    for i = 1 to |Q|
        w = w.replace(r_i)(cheapest(r_i, (f_i, g_i, n_i)), g_i) with (r_i, (f_i, g_i, n_i)) = cheapest(Q[i]);
    return w;
}
```

The above implementation of single-pass processing mode that processes an input basic block with respect to the local cost minimum is pretty straightforward. If the basic block is empty, there is obviously nothing to do. Otherwise, the algorithm tries to match the remainder of the basic block. If at least one rule of the used profile matches the remainder, the implementation continues to process the residue of the cheapest match. Otherwise, the algorithm has reached a dead end and has to backtrack, whereas the computation stops with an exception, if there is no match left to restart the processing with. After having consumed the whole basic block, the algorithm constructs the output basic block.

Example 3.3.4

Let $\Sigma_{in} = \{a, b\}$ be the input alphabet, $\Sigma_{out} = \{c, d\}$ be the output alphabet, $pm = \{p_{in,\,out}\}$ be a pattern matcher, where $p_{in,\,out} = \{r_1, r_2, r_3\}$ is a profile for the input alphabet Σ_{in} and the output alphabet Σ_{out}. The rules r_1, r_2 and r_3 are defined as follows:

$r_1 \coloneqq ([(a, true), *, (a, true)], true, -4, cc)$,

$r_2 \coloneqq ([(a, true), *, (b, true)], true, \lambda(n, g).\text{-}|w| \text{ with } g(n, search(r_2)[2]) = (w, true), cd)$,

$r_3 \coloneqq ([(b, true), (a, true)], true, 4, dc)$.

Let $Q = \varepsilon \in \wp(Match)^*$ be the sequence of matches and $u = abab$ be the input basic block, whereas the colours denote different object instances. The above rules are designed such that the result of their replace patterns correspond to the matched input symbols by means of type (c represents a, d represents b) and colour.

The pattern matcher pm processes the input basic block u as follows:

- The rules match u with $q_1 = \{m_1 = (r_1, (f_1, g_1, 1)), m_2 = (r_2, (f_2, g_2, 2))\}$, whereas the rule r_1 produces one alternative with the cost -4 and the residue bb, and the rule r_2 generates two alternatives, of which the first one has the cost -2 and the residue ba, and the second one has the cost 0 and the residue ab. The algorithm sets $Q = q_1$ and continues to process the residue bb of the cheapest match (with cost -4).
- Unfortunately, there is no rule that matches the residue bb. Thus, the pattern matcher has to backtrack and removes the cheapest alternative of the cheapest match. The algorithm sets $Q = q_2$ with $q_2 = next(q_1) = \{m_2\}$ and continues to process the residue ba of the cheapest match (with cost -2).
- Only the rule r_3 matches ba, producing one alternative with cost 4 and residue ε. So, the algorithm sets $Q = q_2q_3$ with $q_3 = \{m_3 = (r_3, (f_3, g_3, 1))\}$. Because all input symbols have been consumed, the pattern matcher then constructs the final basic block w over Σ_{out} that is defined as follows:

  $w = replace(r_2)(1, g_2).replace(r_3)(1, g_3) = cd.dc = cddc$.

  The total sum of costs is then $cost(r_2, 1) + cost(r_3, 1) = -2 + 4 = 2$.

Table III.8 shows which parts of the original basic block $u = abab$ the rules r_1, r_2 and r_3 have matched while processing u.

Match	*Rule*	*Alternative*	*a*	*b*	*a*	*b*	*Cost*
m_1	r_1	1	*(a, true)*	$*$	*(a, true)*	-	*-4*
m_2	r_2	1	*(a, true)*	*(b, true)*	-		*0*
		2	*(a, true)*	$*$		*(b, true)*	*-2*
m_3	r_3	1	-	*(b, true)*	*(a, true)*	-	*4*

Table III.8: Generated alternatives while compiling abab (local cost minimum).

Although the pattern matcher had always chosen the cheapest alternative possible, the total sum of costs is not optimal. By means of the following pseudo-code algorithm, Example .5 demonstrates how the achieve the global cost minimum while compiling a basic block.

Algorithm: Compile basic block determining global cost minimum (*processSingleGlobal*).
Input: Basic block over Σ_{in}.
Output: Basic block over desired target alphabet Σ_{out}.

let $p_{in,\,out} \in PM$;

$\mathbb{Z}$ *cost* (*Match** Q) {
  let $c = 0 \in \mathbb{Z}$;
  for $i = 1$ **to** $|Q|$
    $c = c + cost(Q[i])$;
  return c;
}

Match extract (*Match* $(r, (f, g, n))$, $\mathbb{N}$ i) {
  return $(r, (f', g', 1))$ **with** $f'(1) = f(i) \wedge g'(1) = g(i)$;
}

*Match** *processSingleRecursive* (Σ_{in}^{*} v) {
  let $q \in \wp(Match)$ **with** $q = matchRules(v)$;
  if $(q = \emptyset)$
    return ε;

  let $Q = \varepsilon \in Match^{*}$;
  foreach $m = (r, (f, g, n)) \in q$
    for $i = 1$ **to** n {
      let $P = \varepsilon \in Match^{*}$;
      if $(residue(i, f) \neq \varepsilon)$
        $P = processSingleRecursive(residue(i, f))$;

      if $((P \neq \varepsilon \vee residue(i, f) = \varepsilon) \wedge (Q = \varepsilon \vee cost(P) + cost(r)(i, g) < cost(Q)))$
        $Q = pP$ **with** $p = extract(m, i)$;
    }
  return Q;
}

Σ_{out}^{*} *processSingleGlobal* (Σ_{in}^{*} u) {
  if $(u = \varepsilon)$
    return ε;

  let $Q \in Match^{*}$ **with** $Q = processSingleRecursive(u)$;
  if $(Q = \varepsilon)$
    throw *exception*;

  let $w = \varepsilon \in \Sigma_{out}^{*}$;
  for $i = 1$ **to** $|Q|$
    $w = w.replace(r_i)(1, g_i)$ **with** $(r_i, (f_i, g_i, 1)) = Q[i]$;
  return w;
}

The core procedure of the above implementation is the function *processSingleRecursive*, which recursively computes a cheapest sequence of matches that consume the given basic block v. First, the function tries to match v with the rules of the used profile. As it is not an error, if none of the rules match the current basic block at this point, the function simply returns the empty sequence of matches in that case. Otherwise, the function recursively computes a sequence of matches that consume the residue of each alternative of the previously generated matches. To determine the global cost minimum, *processSingleRecursive* then chooses the cheapest of these sequences. If several sequences have the cheapest cost, the function automatically selects the first sequence it encounters.

The function *processSingleGlobal* calls the function *processSingleRecursive*, to process the input basic block u. If *processSingleRecursive* is not able to compute a sequence of matches that consumes the input basic block, the pattern matcher cannot compile that basic block. Otherwise, the main function generates the output basic block according to the sequence of matches that the function *processSingleRecursive* has produced.

Example 3.3.5

Let the pattern matcher *pm* and the basic block $u = abab$ be defined as in Example .4. In contrast to the previous processing mode, the pattern matcher investigates every alternative, so that the pattern matcher eventually detects a cheapest sequence of matches.

While processing the input basic block u, the pattern matcher has to choose between the two sequences of matches m_1m_2 and m_3m_4 that are displayed in Table III.9. Note that although the rule r_1 matches u in the beginning (see Example .4), there is no sequence of matches that begins with a match that originates from r_1 (no rule matches the residue bb). Both sequences of matches represent all possible ways to process the input basic block u with the given rules. The sequence m_1m_2 is cheapest with cost 0. Applying the replace pattern of the rule r_2 twice, the pattern matcher finally creates the output basic block $w = cdcd$.

Match	*Rule*	*Alternative*	*a*	*b*	*a*	*b*	*Cost*
m_1	r_2	1	*(a, true)*	*(b, true)*	-		*0*
m_2	r_2	1	-		*(a, true)*	*(b, true)*	*0*
m_3	r_2	1	*(a, true)*	∗		*(b, true)*	*-2*
m_4	r_3	1	-	*(b, true)*	*(a, true)*	-	*4*

Table III.9: Generated alternatives while compiling abab (global cost minimum).

3.3.2. Multi-Pass Matching Mode

To optimise a basic block the mechanisms presented in the previous section are not satisfying. Because not every instruction sequence can be optimised, the user would have to specify additional rules that consume and reinsert those sequences into the basic block, so that the pattern matcher does not abort the processing with an exception. Furthermore, the user would have to call the corresponding single-pass processing function manually as long as at least one rule can be applied. However, this lack in functionality awfully diminishes the usability of this approach. So, a pattern matcher offers two multi-pass processing modes that optimise any input basic block.

The following pseudo-code implementation demonstrates how a pattern matcher optimises a basic block with respect to the local cost minimum.

Algorithm: Optimise basic block with respect to the local cost minimum (*processMultiLocal*).
Input: Basic block over Σ.
Output: Optimised basic block over Σ.

let $p \in PM$;

Σ^* *processMultiLocal* (Σ^* u) {
  let $x, y, v \in \Sigma^*$ **with** $x = \varepsilon, y = u$ and $v = u$;
  while ($y \neq \varepsilon$) {
    let $q \in \wp(Match)$ **with** $q = matchRules(y)$;
    if ($q \neq \emptyset$) {
      let $m = (r, (f, g, n)) \in Match$ **with** $m = cheapest(q)$;
      $x = x.replace(r)(cheapest(r, (f, g, n)), g)$;
      $y = residue(m)$;
      $u = xy$;
    }
    else if ($y \neq \varepsilon$) {
      $x = xa$ **with** $a = y[0]$;
      $y = y[1{:}|y|]$;
    }

    if ($y = \varepsilon$)
      if ($v \neq u$) {
        $x = \varepsilon$;
        $y = u$;
        $v = u$;
      }
  }
  return u;
}

Always applying the rule that contributed the cheapest match, the above algorithm optimises a basic block in multiple passes. Because the pattern matcher does not need to consume each instruction and the pattern matcher does thus not have to backtrack if no rule matches, the algorithm does not remember any previous match.

The function *processMultiLocal* operates on the basic block suffix y that is initialised with the input basic block u in the beginning. Additionally, the algorithm stores the processed prefix of u in the variable x and remembers the original version of the given basic block in v. The function continues to optimise the basic block u until the suffix y is the empty word. To optimise the given basic block, the function *processMultiLocal* first tries to match the suffix y with every rule of the used optimisation profile p. If existent, the algorithm chooses the cheapest match, appends the replace pattern to the prefix x, assigns the residue of that match to y and finally updates the basic block u. Otherwise, the function skips the first symbol of the suffix y, as no rule is able to accept any prefix of y. By means of the optimised basic block u and its original version v, the algorithm determines whether it should continue. If u and v do not differ from each other, either no rule has matched or the replace patterns did not alter u. So, the function stops to optimise u in that case. Otherwise, the function repeats that procedure as described.

The following example demonstrates how this processing method optimises a basic block.

Example 3.3.6

Let $\Sigma = \{a, b, c, d\}$ be the instruction alphabet, $p = \{r_1, r_2, r_3\}$ be an optimisation profile and $pm = \{p\}$ be a pattern matcher. The rules r_1, r_2 and r_3 are defined as follows:

$r_1 \coloneqq ([(b, true), (b, true)], true, 2, a)$,

$r_2 \coloneqq ([(b, true), (b, true)], true, 1, cc)$,

$r_3 \coloneqq ([(c, true), (c, true)], true, 2, d)$.

Let $u = abbcc \in \Sigma^*$ be the basic block to optimise. As previously, the colours denote different instances, whereas the colours of the generated symbols resemble the matched instances.

The pattern matcher pm optimises the basic block u as follows:

1. The algorithm initialises x with ε, y with u and v with u.
2. As no rule matches y, the pattern matcher skips the first symbol of y, such that $x = a$ and $y = bbcc$.
3. Then, the rules r_1 and r_2 match y with the matches m_1 and m_2 respectively, whereas both rules only provide one alternative. Because m_2 is cheapest with cost 1, the pattern matcher chooses to apply r_2. Thus, $x = acc$, $y = cc$ and $u = accc$.
4. At this point, only the rule r_3 matches y with the match m_3 generating one alternative with cost 2. So, the pattern matcher assigns $x = accd$, $y = \varepsilon$ and $u = accd$. Because y has been consumed completely and $v = abbcc$ differs from u, the pattern matcher decides to restart and assigns $x = \varepsilon$, $y = u$ and $v = u$.
5. In the second iteration, only the rule r_3 eventually matches y with the match m_4. After this iteration, the optimised basic block is $u = add$. Again v and u differ from each other and the pattern matcher begins the third iteration. However, because no rule matches y in that iteration, the pattern matcher returns add, whereas the total sum of costs is 5.

Table III.10 displays which parts of the basic block the rules r_1, r_2 and r_3 have matched during the first two iterations of the optimisation process.

Match	*Rule*	*Alternative*	*a*	*b*	*b*	*c*	*c*	*Cost*
m_1	r_1	1	-	*(b, true)*	*(b, true)*	-	-	*2*
m_2	r_2	1	-	*(b, true)*	*(b, true)*	-	-	*1*
m_3	r_3	1	*(a, true)*	-	-	*(c, true)*	*(c, true)*	*2*
Match	***Rule***	***Alternative***	***a***	***c***	***c***	***d***		***Cost***
m_4	r_3	1	-	*(c, true)*	*(c, true)*	-		*2*

Table III.10: Generated alternatives in the first two passes while optimising abbcc.

However, the total sum of costs is not optimal. Example .7 demonstrates that, if the pattern matcher chooses the most expensive alternative in the first place (see 3.), the total cost sum is optimal.

So, using the following pseudo-code algorithm that determines the global cost minimum by investigating every alternative, a pattern matcher might achieve better results.

Algorithm: Optimise basic block determining the global cost minimum (*processMultiGlobal*).
Input: Basic block over Σ.
Output: Optimised basic block over Σ.

```
let p ∈ PM;

Σ* × ℤ processMultiRecursive (Σ* x, Σ* y, Σ* u, ℤ cost) {
    let q = Ø ∈ ℘(Match);
    while (true) {
        q = matchRules(y);
        if (q ≠ Ø)
            break;

        if (y ≠ ε) {
            x = xa with a = y[0];
            y = y[1:|y|];
        }

        if (y = ε)
            if (x ≠ u) {
                u = x;
                y = x;
                x = ε;
            }
            else
                return (u, cost);
    }

    let v = ε ∈ Σ*;
    let c = 0 ∈ ℤ;
    foreach m = (r, (f, g, n)) ∈ q
        for i = 1 to n {
            let x' ∈ Σ * with x' = x.replace(r)(i, g);
            let y' ∈ Σ * with y' = residue(i, f);
            let (w, d) ∈ Σ*×ℤ  with (w, d) = processMultiRecursive(x', y', u, cost + cost(r)(i, g));

            if (v = ε)
                (v, c) = (w, d);
            else if (d < c)
                (v, c) = (w, d);
        }
    return (v, c);
}

Σ* processMultiGlobal (Σ* u) {
    ℤ c = 0;
    (u, c) = processMultiRecursive(ε, u, u, 0);
    return u;
}
```

Investigating every successful match, the function *processMultiRecursive* recursively optimises a basic block. The function parameters have a similar meaning as in the multi-pass processing function *processMultiSingle*. The function parameter x is a prefix of the input basic block that has already been processed, y is a suffix of the basic block that is yet to be optimised, u is the initial version of the input basic block before an optimisation pass, and *cost* is the cost of the current iteration.

At first, the algorithm tries to match the suffix y. If no rule matches y, the algorithm skips the first symbol of y, if y is not the empty word. If the suffix y is the empty word, the algorithm decides whether to restart the matching. The function stops and simply returns u and *cost*, if the optimised basic block does not differ from the given basic block u (i.e., $x \neq u$).

If at lest one rule matches the suffix y, the algorithm calls itself for every alternative of each match to determine the cheapest optimisation. The algorithm eventually terminates, because it stops to process the given basic block, if the that basic block cannot be altered any further.

Example 3.3.7

Let the pattern matcher *pm* be defined as in Example .6 and $u = \mathbf{abbcc}$ be the basic block to optimise. According to the pseudo-code function *processMultiGlobal*, *pm* optimises u as follows:

1. The function *processMultiGlobal* calls *processMultiRecursive*$(\varepsilon, u, u, 0)$. Because none of the rules matches $y = u$, the algorithm skips the first symbol of y ($x = \mathbf{a}$) and matches $y = \mathbf{bbcc}$ with the rules r_1 and r_2, each of which provides only one alternative.
2. Assuming that the algorithm first investigates the alternative of r_2, the function *processMultiRecursive* calls itself with the parameters $x = \mathbf{acc}$, $y = \mathbf{cc}$, $u = \mathbf{abbcc}$ and the cost 1:
   a) At this point, only the rule r_3 is able to match y. Thus, *processMultiRecursive* calls itself with the parameters $x = \mathbf{accd}$, $y = \varepsilon$, $u = \mathbf{abbcc}$ and $cost = 3$.
   b) Because y is the empty word and $x = \mathbf{accd}$ differs from $u = \mathbf{abbcc}$, the algorithm sets $x = \varepsilon$, $y = \mathbf{accd}$ and $u = y$ to restart the matching. From now on, only r_2 can match, and the algorithm finally generates the basic block $\mathbf{add}$ with the cost 5.
3. Afterwards, the algorithm investigates the alternative of the rule r_1, and calls the function *processMultiRecursive* with $x = \mathbf{aa}$, $y = \mathbf{cc}$, $u = \mathbf{abbcc}$ and $cost = 2$:
   a) As previously, only the rule r_3 is able to match y. The algorithm then investigates the alternative that r_3 provides and calls the procedure *processMultiRecursive* with the parameters $x = \mathbf{aad}$, $y = \varepsilon$, $u = \mathbf{abbcc}$ and $cost = 4$.
   b) Again, the algorithm decides to restart the matching, as y is the empty word and x differs from u. However, in this case no rule matches any suffix y, so the function simply returns the basic block $\mathbf{aad}$ with the cost 4.
4. It turns out that the second alternative has provided a cheaper optimisation. Because there are no more alternatives left to investigate, the optimisation with the lowest possible cost of the input basic block $u = \mathbf{abbcc}$ is then $\mathbf{aad}$.

3.4. Complexity

At first, this section discusses the worst-case runtime of a rule to match a basic block, before the section estimates the runtime of a pattern matcher with respect to the four different processing methods that were introduced in Section 3.3.

3.4.1. Rule

The used search pattern of a rule determines the number steps a rule needs to match a basic block. The more alternatives a search pattern can produce, the more steps the search pattern has to take to match a basic block. So, for $n \in \mathbb{N}$ the runtime of the unordered sequence pattern $p_n = \{*_1, ..., *_n\}$ to match the basic block w of length $m \in \mathbb{N}$ represents an upper bound for the runtime of any search pattern.

Naturally, the sequence pattern p_n is not a valid search pattern, because it does not contain an item pattern (see Definition .1). Additionally, when matching a basic block, each wildcard pattern of p_n will be immediately marked as finished, because there is no item pattern to start with. However, in this case, I will simply disregard these restrictions.

Because the unordered sequence pattern p_n abbreviates $n!$ different ordered sequence patterns, the runtime of p_n can easily be derived from the runtime of the pattern $q_n = [*_1, ..., *_n]$. If q_n matches a basic block in k steps, then p_n consequently takes $n! * k$ steps to match the same basic block. To derive a function that computes the runtime of q_n, it is first necessary to understand how a wildcard pattern processes a basic block.

Basically, a wildcard pattern may choose between two operations to process a basic block. On the one hand, the pattern may *finish* and leave that basic block to a successor. On the other hand, the wildcard pattern may *skip* the first instruction of the basic block. To generate all possible alternatives, a wildcard pattern always performs both operations[22]. Thus, the runtime of a wildcard pattern is determined by how often the wildcard pattern has finished and by the number of input symbols the wildcard pattern has skipped.

A wildcard pattern processes the basic block w as follows:

- If there at least one input symbol left (i.e., $m > 0$), the wildcard pattern produces two alternatives. In the first one the wildcard pattern simply finishes to process the basic block, whereas in the second one the pattern skips the current input symbol.
- Otherwise, the wildcard pattern has no other choice but to finish.

So, the wildcard pattern takes $2m+1$ steps to generate all possible alternatives. A sequence of wildcard patterns matches the basic block similarly, whereas the successor of a wildcard pattern only continues to process the remaining input symbols as described, after that wildcard pattern has finished.

This observation leads to the function *steps*: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ that determines the number of steps, which the ordered sequence pattern q_n needs to process a basic block of length m.

$$steps(n, m) := \begin{cases} 2m+1 + \sum_{i=0}^{m} steps(n-1, i) & n > 0 \\ 0 & \text{else.} \end{cases}$$

22 In general, a wildcard pattern may only skip an input symbol, if there are still enough input symbols left for each unfinished item pattern. However, because there are no item patterns in this case, each wildcard pattern may safely skip arbitrarily many input symbols.

Experiments with the actual implementation show that the function *steps* is a perfect upper bound for the runtime of the ordered sequence pattern q_n. However, to provide a feeling for the complexity of the matching process, the above function is rather insufficient. So, the following lemma derives an appropriate complexity class.

Lemma 3.4.1

For any $n, m \in \mathbb{N}$, it holds that $steps(n, m) \in O((m+1)^n)$.

Proof (Induction over n)

The case $n = 0$ is uninteresting, because there is no wildcard pattern available. So, in the following it is assumed that $n \geq 1$.

Base case ($n = 1, n = 2$):

$$steps(1, m) = 2m+1 + \sum_{i=0}^{m} \underbrace{steps(0, i)}_{=0} = 2m+1 \in O(m+1).$$

$$\begin{aligned} steps(2, m) &= 2m+1 + \sum_{i=0}^{m} \underbrace{steps(1, i)}_{=2i+1} \\ &= 2m+1 + (m+1) + 2\sum_{i=0}^{m} i \\ &= 2m+1 + \underbrace{(m+1) + m(m+1)}_{=(m+1)^2} \in O((m+1)^2). \end{aligned}$$

Induction step: $n \rightarrow n+1$ (assuming that the claim holds for all $i \in \mathbb{N}$ with $i < n$):

$$\begin{aligned} steps(n+1, m) &= 2m+1 + \sum_{i=0}^{m} \underbrace{steps(n, i)}_{\substack{\leq g_i \in O((i+1)^n) \\ \leq g_{m+1} \in O((m+1)^n)}} \\ &\leq 2m+1 + \sum_{i=0}^{m} g_{m+1} = 2m+1 + \underbrace{(m+1)g_{m+1}}_{\in O((m+1)^{n+1})} \in O((m+1)^{n+1}) \end{aligned}$$

□

So, the upper bound of the runtime of the ordered sequence pattern q_n to match a basic block of length m is $O((m+1)^n)$, whereas $O(n! * (m+1)^n)$ is the upper bound for the runtime of the unordered sequence pattern p_n.

It seems that due to this upper bound a pattern matcher compiles a basic block with an unacceptable runtime. However, the patterns p_n and q_n represent a class of patterns that will never occur in practice. In general, a rule matches a basic block much faster, because its search pattern comprises at most one wildcard pattern. A common search pattern has either the form $p_{i,n} = \{a_1, ..., a_i, *, a_{i+2}, ..., a_n\}$ or $q_{i,n} = [a_1, ..., a_i, *, a_{i+2}, ..., a_n]$, whereas $1 \leq i < n-1$ ($n \geq 3$) and the a_j denote item patterns. Similar as above, the runtime of the pattern $p_{i,n}$ can be directly derived from the runtime of $q_{i,n}$, as follows:

To estimate the runtime of the search patterns $p_{i,n}$ and $q_{i,n}$ it is assumed that the basic block w contains enough symbols (i.e., $m \geq n-1$), so that both patterns are able to match w. The pattern $q_{i,n}$ matches w in three phases. At first, the item patterns $a_1, ..., a_i$ have to consume an input symbol, before the inner wildcard pattern may start to match the remaining basic block.

However, the wildcard pattern may only continue to skip input symbols as long as there are at least $k := n - i - 1$ symbols left, so that there is an item pattern for each of the k remaining item patterns $a_{i+2}, ..., a_n$. So, the wildcard pattern only has $j := m - i - k = m - n + 1$ symbols at its disposal. Thus, the wildcard patterns needs $steps(1, j) = 2m - 2n + 3$ operations to process the j input symbols, whereas the wildcard pattern creates $j+1$ different alternatives. After the wildcard pattern is finished, the remaining k item patterns have to match a symbol in each of the generated alternatives. So, the number of steps the ordered sequence pattern $q_{i,n}$ needs to match the basic block w is determined by the function $steps$: $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ that is defined as follows:

$$\begin{aligned} steps(i, n, m) &:= i + \underbrace{(2m - 2n + 3)}_{= steps(1,\, j)} + \underbrace{(m - n + 2)}_{= j+1}\underbrace{(n - i - 1)}_{= k} \\ &= nm - im + m + in - n^2 - i + n + 1 \in O(nm) \end{aligned}$$

So, the upper bound for the runtime of $q_{i,n}$ to match the basic block w is $O(nm)$. To determine the runtime of $p_{i,n}$ all possible configurations of the subpatterns of $p_{i,n}$ have to be taken into account. There are $(n\text{-}1)!$ different permutations of the item patterns $a_1, ..., a_i, a_{i+1}, ..., a_n$. Additionally, the inner wildcard pattern may appear at n different locations. Based on this data, the following formula describes an upper bound for $p_{i,n}$:

$$\begin{aligned} (n-1)! * \sum_{i=1}^{n} \underbrace{steps(i, n, m)}_{\leq g \,\in\, O(nm)} &\leq (n-1)! * \sum_{i=1}^{n} g \\ &= (n-1)! * ng = n!*g \in O(n!*nm) \end{aligned}$$

As expected, the upper bound for the runtime of the unordered sequence pattern $p_{i,n}$ to match the basic block w is then $O(n!*nm)$. As long as n is small, the runtime of both pattern types is acceptable in practice, as Section 2.3 of Chapter V shows.

If an ordered sequence pattern only consists of n item patterns, the number of steps to match the basic block w is obviously bounded by $O(n)$, whereas $O(n!)$ is obviously an upper bound for a corresponding unordered sequence pattern.

So, the user should employ wildcard patterns sparsely to keep the worst-case processing time of a pattern matcher as low as possible.

3.4.2. Pattern Matcher

Let w be a basic block of length $m \in \mathbb{N}$ and p a profile that contains $n \in \mathbb{N}$ different rules. To estimate the worst-case runtime of a pattern matcher that uses the profile p to process w, the number of alternatives, which the rules of p can produce, play an important role. The number of alternatives that a rule can generate obviously depends on how many unordered sequence patterns and wildcard patterns its search pattern comprises. The general rule of thumb is: the less unordered sequence patterns a search pattern contains the more alternatives the search pattern is able to produce. The search patterns $s_1 = \{a_1, a_2, a_3\}$ and $s_2 = \{a_1, \{a_2, a_3\}\}$, where the a_i denote item patterns, demonstrate this effect. Although both search patterns have the same size (number of non-sequence subpatterns), s_1 is able to generate $3! = 6$ alternatives, whereas s_2 only can create $2!*2! = 4$ different alternatives, because the inner unordered sequence pattern prevents s_2 from matching every permutation of the a_i. In contrast to the unordered sequence patterns, the more wildcard patterns a search pattern comprises the more alternatives the search pattern is able to create, whereas the number of alternatives depends both on the basic

block length and the number of item patterns that have not yet consumed an input symbol, as the previous section shows. So, the number of alternatives, which a rule can generate, depends on the size of the search pattern and the length of the basic block. In the following, $k \in \mathbb{N}$ abbreviates the maximum number of alternatives, which the rules of the profile p are able to create when matching the basic block w.

By means of the size of the input basic block m, the number of available rules n, the maximum number of alternatives k and the function f that represents an upper bound for the runtime of each rule (see Section 3.4.1), the remainder of this section assesses the worst-case runtime of the four processing methods (see Section 3.3.1 and 3.3.2).

- **Single-pass, local cost minimum**

  Under the assumption that it is not necessary to backtrack, the pattern matcher compiles the basic block as follows. At first the n rules have to match the remainder of the basic block. Then, the pattern matcher determines the cheapest alternative by sorting the k maximum possible alternatives (the actual implementation uses of the heap sort algorithm that sorts the alternatives in $O(k\log(k))$). In the worst case, the cheapest alternative removes only one symbol from the basic block, so that the pattern matcher has to repeat the procedure until no more input symbols are left. So, the following formula represents an upper bound for this processing method.

$$\begin{aligned}\underbrace{nf+k\log(k) + nf+k\log(k) + \ldots}_{m \text{ times}} &= \sum_{i=1}^{m} (nf + k\log(k)) \\ &= mnf + mk\log(k) \in O(mnf + mk\log(k))\end{aligned}$$

  If the pattern matcher always has to backtrack, this processing methods becomes very expensive. In addition to the above procedure, the pattern matcher has to investigate each of the k alternatives, after the pattern matcher has sorted the alternatives. Under the assumption that each alternative only removes one input symbol from the basic block, the following formula describes an upper bound for this processing method.

$$\begin{aligned}\underbrace{nf+k\log(k) + k(\ldots)}_{m \text{ times (recursive)}} &= \sum_{i=1}^{m} \underbrace{k^{i-1}}_{\leq k^{m-1}} (nf + k\log(k)) \\ &\leq \sum_{i=1}^{m} k^{m-1}(nf + k\log(k)) \\ &= mnfk^{m-1} + mk^{m}\log(k)) \in O(mnfk^{m-1} + mk^{m}\log(k))\end{aligned}$$

- **Single-pass, global cost minimum**

  An upper bound for the single-pass processing method that determines the global cost minimum can be directly derived from the previous runtime analysis. This processing method compiles a basic block investigating all available alternatives to compute the global cost minimum. Apart from sorting the alternatives, the processing method behaves as the single-pass processing method that always has to backtrack. The following formula represents an upper bound for this processing method.

$$\underbrace{nf + k(nf + k(\ldots))}_{m \text{ times (recursive)}} = \sum_{i=1}^{m} nf\, \underbrace{k^{i-1}}_{\leq k^{m-1}} \leq \sum_{i=1}^{m} nfk^{m-1} = mnfk^{m-1} \in O(mnfk^{m-1})$$

- **Multi-pass, local cost minimum**

  Basically, this processing method acts like the single-pass processing mode, whereas it does not remember any previously generated alternative, because backtracking is not required, and it might process the basic block multiple times. Note that the algorithm first tries to match every symbol of the input basic block, before the algorithm decides to restart the matching. So, the runtime of one iteration corresponds to the runtime of the single-pass processing method (without backtracking).

  How often this processing method iterates, obviously depends on the given basic block and the used profile. Unfortunately, there exists no universal formula that estimates the maximum number of iterations. Even though it is possible to determine the maximum iteration count for special profiles[23], there is no applicable restriction that helps to estimate the number of iterations in the general case.

  The following formula represents an upper bound for the runtime of this processing method, whereas $l \in \mathbb{N}$ with $l \geq 1$ denotes the maximum number of iterations.

$$\begin{aligned} \underbrace{nf + k\log(k) + \ldots}_{lm \text{ times}} &= \sum_{i=1}^{lm} (nf + k\log(k)) \\ &= lmnf + lmk\log(k) \in O(lmnf + lmk\log(k)) \end{aligned}$$

- **Multi-pass, global cost minimum**

  This algorithm optimises a basic block like the multi-pass processing that determines the local cost minimum. Instead of sorting the generated alternatives, this algorithm investigates each alternative to determine the global cost minimum. Because this processing may restart the matching in each of the generated alternatives, this algorithm is the most expensive one. As previously, it is not possible to determine the maximum number of iterations for an arbitrary basic block and profile.

  The following formula represents an upper bound for this multi-pass optimisation algorithm, whereas $l \in \mathbb{N}$ with $l \geq 1$ denotes the maximum number iterations.

$$\begin{aligned} \underbrace{nf + k(nf + k(\ldots))}_{lm \text{ times (recursive)}} = \sum_{i=1}^{lm} nf \underbrace{k^{i-1}}_{\leq k^{lm-1}} &\leq \sum_{i=1}^{m} nfk^{lm-1} \\ &= lmnfk^{lm-1} \in O(lmnfk^{lm-1}) \end{aligned}$$

It turns out that the single-pass processing method that compiles a basic block with respect to the local cost minimum is – as expected – the cheapest pattern matcher processing method. If it is not necessary to backtrack, this processing method is linear in the basic block length and the number of rules and logarithmic in the number of alternatives. However, if the pattern matcher is forced to backtrack all the time, this single-pass processing method is exponential in the number of alternatives and is thus even more expensive than the single-pass processing method that determines the global cost minimum. This relationship does not apply to the two multi-pass processing methods, because the pattern matcher does not need to backtrack, if no rule matches the basic block while optimising a basic block. Being exponential in the number of alternatives, the multi-pass processing method is most expensive.

23 If e.g., the rules' replace patterns create instructions that none of the used search patterns are able to match, this processing method will iterate at most 2 times.

So, the user should select the processing mode carefully, so that the pattern matcher does not take too long to compile or optimise a basic block. As Chapter V demonstrates, it does not always pay off to determine the global cost minimum.

IV. Pattern Matcher Generator

Discussing the functioning of the pattern matcher generator tpmg and the generated pattern matchers, this chapter presents the practical main part of the present work. After the first section gives an insight into the tpmg implementation, the second section introduces the pattern matcher description language, in which the user can specify any kind of pattern matcher. The third section discusses the differences between the implementation of the tpmg-generated pattern matchers and the pseudo-code realisation (see Chapter III). Briefly introducing the tpmg debugger interface, the fourth section concludes this chapter.

1. Overview

The acronym tpmg means **t**ree **p**attern **m**atcher **g**enerator. Although the pattern matchers do not operate on tree-like structures, I have chosen this name, because every instruction can be understood as a minimal expression tree, whose root is the target register (or variable) and whose leafs are the operands. Using appropriate side conditions, a pattern matcher is able to match arbitrarily large expression trees, whereas it is not required that the instructions are stored in tree-like data structures.

Figure IV.1 shows the corresponding expression tree that the optimisation rule *mad* matches (see Example .6 in Chapter III). Remember that the rule searches two instructions – that not need to be adjacent to each other – where the operand and the target of the *ADD* instruction is the target of the *MUL* instruction.

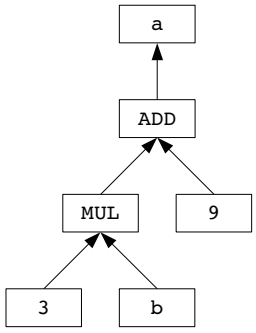

Figure IV.1: Expression tree.

I have implemented the pattern matcher generator tpmg in the C++ programming language, using the scanner generator flex [38] and the parser generator bison [39] to realise the tpmg front end. Besides making use of the standard template library (STL [40]), the application does not depend on any other external library, so that tpmg is available on almost every platform. To ease the understanding of the tpmg implementation, I have annotated the code, as well as the tpmg-generated code, with doxygen meta tags [41]. To automatise the build process, the project makes use of autoconf [42] and automake [43] that are known to work on various platforms. I have successfully compiled tpmg on Linux, NetBSD and Windows XP (using mingw).

The purpose of tpmg is to compile a pattern matcher description into an automated, retargetable pattern matcher. Currently, tpmg only generates C++ pattern matchers that are build on top of the tpmg template library that makes use of the STL[24]. The template library defines the necessary data types (e.g., *ItemPattern*, *SequencePattern* or *Rule*) and implements the basic pattern matcher functionality. Although this approach requires the used C++ compiler to support

24 Of course, any other data structure library with similar features suffices.

templates and unfortunately causes longer compile times, the major advantage is the type safety at compile time. So, this approach overcomes the ugliness of a pure C implementation, where instruction objects would have to be passed as function parameters of type `void*`. Independent from the C++ template support, the compiler is additionally required to provide runtime type information (RTTI) of the processed instruction objects. With RTTI, a pattern matcher does not require an external mechanism that makes the objects to match distinguishable from each other. So, the pattern matcher does not require that the objects to process implement an interface in advance. However, to enable the generated pattern matchers to make use of the STL, the objects to match have to be instances of a common base class. A pattern matcher receives the input basic block to compile or optimise in form of a `std::list` over that common base class (e.g., if the common base class is `Instruction`, `std::list<Instruction>` must be the type of the basic block). After having processed the basic block, the pattern matcher returns the compiled (or optimised) basic block or throws an exception to report an error, if e.g., a certain instruction object could not be matched. Because the tpmg-generated pattern matchers only depend on the STL, they can be compiled on almost every platform. Like tpmg, I have successfully tested tpmg-generated pattern matchers on Linux, NetBSD and Windows XP.

The tpmg back end can be adjusted to any other object orientated programming language, as e.g., Java, if the target language provides the following features:

- RTTI support to distinguish the processed objects during runtime,
- Template support (or a similar mechanism) for type safety at compile time,
- Exception handling[25],
- Available data structure library like STL.

Independent from the target language, creating and embedding a tpmg-generated pattern matcher in an application always takes place according to Figure IV.2. At first, the user has to describe the pattern matcher in a tpmg rule file. The user may chose to outsource parts of the rule file (e.g., a profile or just a single rule) in auxiliary rule files, to keep the pattern matcher description readable and – more important – maintainable. The format of a rule file is determined by the pattern matcher description language (see Section 2). Next, tpmg parses and validates the input rule file(s) to generate the pattern matcher. Before the application can be finally compiled, the user has to make sure that the application uses of the pattern matcher through the generated interface. Thus, the user obviously has to implement the application in the same language as the tpmg-generated pattern matcher.

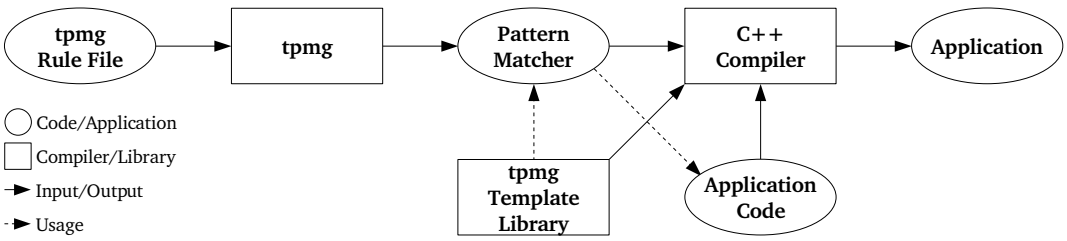

Figure IV.2: tpmg compilation diagram.

25 Exceptions are not truly required to implement the pattern matchers, but offer a much nicer way to report errors without having to encode an error in the return value of the affected functions.

2. Pattern Matcher Description Language

The pattern matcher description language orientates itself at the bison grammar and adopts language constructs of recent object oriented programming languages, such as Java and C++. Instead of creating a completely new syntax, I have chosen to combine well-known languages in this description language, so that a new user finds himself in a familiar environment and can start developing right away.

The following sections introduce the syntax and semantics of the pattern matcher description language. I will use an extended Backus-Naur form to introduce the syntax, whereas upper case alphanumerical strings denote non-terminals of the grammar. Lower case alphanumerical strings and non-alphanumerical strings enclosed in single quotation marks represent terminal symbols of the description language. The following four operators extend the standard Backus-Naur form to make the grammar more easier to read:

- (...) to identify a group of symbols,
- [...] to denote that the enclosed section is optional,
- * to mark that the preceding symbol (or group of symbols) may appear arbitrarily often or never,
- $^+$ to indicate that the preceding symbol (or symbol group) must appear at least once.

2.1. Outline

A tpmg rule file contains at least one rule set that may be preceded by a header section.

```
RULEFILE ::= ([HEADER] RULESET)+

HEADER   ::= '%{' ASCII* '%}'
```

In the *header* section, the user has to define the types of the instruction objects on which the pattern matcher operates. Because every language construct of the target language (i.e., C++ in this case) is allowed within the header section, the user can additionally specify own data types, functions or global variables, which the generated pattern matcher can access. The user may even implement the whole application that makes use of the pattern matcher in the rule file header. However, I strongly advice against doing so in practice, to keep the project maintainable in the long run. Figure IV.3 shows a typical header section.

```
%{
  // include header files to define data types
  #include <custom_types.h>
  ...

  // embedded function
  bool valid (UnaryInstruction *instruction)
  {
    ...
  }

  // global variable
  static int flag = 0;
%}
```

Figure IV.3: Typical rule file header section.

Note that tpmg does neither validate the contents of the header section nor evaluate the preprocessor statements inside the header section. This has to be done by the C++ compiler later on.

After the header section, follows the *rule set* section that contains the representation of the pattern matcher. In that section, the user specifies the common base class of the instruction objects, the profiles and the rules that constitute the pattern matcher. Section 2.2 further discusses the syntax and semantics of the rule set.

A rule file may contain multiple header and rule set sections. However, a header must always precede a rule set and there may be at most one header per rule set. After the tpmg front end has processed the rule file, tpmg merges all headers and rule sets, so that the user can outsource the pattern matcher description in multiple rule files.

Comments may appear everywhere within a rule file. A line starting with `//` denotes a single-line comment, whereas `/*` ... `*/` encloses a multi-line comment.

Additionally, the user may use certain preprocessor statements known from C and C++ respectively. The pattern matcher generator contains its own preprocessor that understands the following subset of preprocessor statements:

- The statement `#include "FILENAME"` commands tpmg to open and process the given file – with respect to the current working directory and the parser status – before the pattern matcher generator may continue with the current file. If the file could not be found, tpmg tries to locate that file in search paths that the user has provided. If the file could not be found at all, the pattern matcher generator stops immediately.
- In contrast to the above statement, the `#include <FILENAME>` statement commands tpmg to only locate the specified file in the user-provided search paths.
- The user may define a variable with `#define VARIABLE`. In contrast to other preprocessors, defined variables do not have a value. The main purpose of variables is to make include guards possible. With the `#undef VARIABLE` statement, the user can undefine a previously defined variable.
- To hide specific parts of the rule file from tpmg and the internal preprocessor, the user may use the `#ifdef VARIABLE`, `#else` and `#endif` statements. An `#ifdef` marks the beginning of a guarded section that must be terminated by an `#endif`. If the specified variable is not defined, the guarded section will be hidden and is visible otherwise. The user can achieve the opposite behaviour with the `#ifndef VARIABLE` statement. The `#else` statement, which may only appear between `#ifdef` (or `#ifndef`) and `#endif`, makes it additionally possible to make a partition of the guarded area visible while the other one is hidden and vice versa.

  To prevent a file from being included twice, the user must guard the file contents as follows:

```
#ifndef UNIQUE_IDENTIFIER
#define UNIQUE_IDENTIFIER

...

#endif
```

2.2. Rule Set

A rule set contains the complete specification of a pattern matcher, which is defined through the common base class of the input alphabet, the common base class of the output alphabet and the rules and profiles for each supported hardware architecture. Additionally, a rule set determines the implicit behaviour of the generated pattern matcher.

```
RULESET        ::= RULESET_HEADER RULESET_BODY
RULESET_HEADER ::= ruleset '<' CLASSNAME ',' CLASSNAME '>'
RULESET_BODY   ::= '{' [IMPLICIT] (RULE | PROFILE)* '}'
```

The first class name of the rule set header denotes the *input base class* that determines the input alphabet. The input base class is the common base class of all instruction objects that may appear in an input basic block. Thus, the input alphabet implicitly contains the input base class and all other classes that are derived from that input base class. The second class name determines the *output base class* and the pattern matcher's output alphabet analogously. If the two base classes differ from each other, the generated pattern matcher can only offer the single-pass basic block processing methods[26] (see Section 3.3.1 in Chapter III). Thus, the pattern matcher can only be employed in the code generation. Otherwise, the pattern matcher is additionally able to optimise basic blocks with the multi-pass processing methods.

The *main rule set* is the first rule set that tpmg processes. The pattern matcher generator will append any other rule set to main rule set, if the input and output class match the input and output class of the main rule set. As tpmg is not able to detect class relationships, the class names must be exactly the same. Otherwise, tpmg will stop, reporting that the rule set could not be appended to the main rule set.

It appears that there is a major difference between the formal pattern matcher description and the pattern matcher description language. In contrast to the profiles of a formal pattern matcher, every profile of a tpmg-generated pattern matcher must operate on the same input and output alphabet. However, this restriction does not diminish the expressive power of the description language. So, if the application uses different instruction classes for each supported hardware architecture, these classes simply have to share a common base class, so that the pattern matcher is able to generate instruction sequences for each target platform.

The rule set body comprises the rules and profiles of the pattern matcher. Besides the user-specified profiles, a rule set implicitly contains the *global* profile that combines all those rules that do not belong to a specific profile.

As hinted in Example .6 in Chapter III, tpmg creates rules that implicitly check for side effects that might cause the pattern matcher to wrongly modify the semantics of the processed basic block. However, because tpmg does not know the environment, in which the generated pattern matcher will be employed, the pattern matcher generator will only insert a function call. If those side effects can occur, the user has thus to specify the verification function in the *implicit* section that precedes the rule and profile definition.

26 This is not quite true. If the input base class is derived from the output class, the input alphabet Σ_{in} is a subset of the output alphabet Σ_{out}. In that case, the pattern matcher is able to optimise any input basic block. Because the pattern matcher simply skips any instruction object that cannot be optimised, the pattern matcher will not cease to function, if it ever encounters an instruction object whose class is contained in $\Sigma_{out} \setminus \Sigma_{in}$. However, due to restrictions of the C^{++} template mechanism and the pattern matcher generator's inability to detect such class relationships, tpmg cannot support this special kind of optimisation pattern matcher.

2.2.1. Profile

Profiles enable the user to combine rules that are dedicated to a certain kind of problem. Depending on the desired outcome, an application, which makes use of a pattern matcher, first has to select the appropriate profile, before the generated pattern matcher can start to process the input basic block. In general, a profile combines rules that are related to a specific target architecture.

```
PROFILE        ::= PROFILE_HEADER PROFILE_BODY
PROFILE_HEADER ::= profile PROFILENAME [PROFILE_EXTEND]
PROFILE_EXTEND ::= ':' extends PROFILENAME [',' PROFILENAME]*
PROFILE_BODY   ::= ';' | '{' (RULE | RULE_OMIT)+ '}'
RULE_OMIT      ::= omit RULE_ID [',' RULE_ID]* ';'
RULE_ID        ::= [[PROFILENAME] '::'] RULENAME
```

The name of a profile must be unique and must not be the empty word. In every profile – this includes the global profile as well – the name of a rule may only occur once. So, there may be a rule named *add* in the *NV30* profile as well as in the *NV40* profile, but there may not be a second rule called *add* in any of the two profiles.

Each profile automatically derives from the global profile and thus inherits all rules of the global profile. A profile will even inherit those rules of the global profile, that have been specified syntactically after that profile. The user may additionally derive a profile from multiple other profiles, so that the user does not have to implement a rule twice that two profiles have in common. Deriving from the same profile more than once is not an error and does not cause the new profile to contain the rules of the derived profile multiple times. Because the profile body may be empty, the user can easily create aliases of profile in that way. The pattern matcher generator does not support profile forward declarations. So, if the user attempts to derive a new profile from an unknown profile, or a profile that has not been processed, tpmg stops with an appropriate error message. Additionally, tpmg does not support nested profiles.

Whenever a profile inherits rules from other profiles, the user might want to prevent certain rules from being included in the new profile, if they are no longer required. At every position within the profile specification, the user can exclude any number of rules from the profile using the `omit` keyword. If the user attempts to omit a rule that does not exist, tpmg stops with an appropriate error. Because a rule name may only occur once per profile, each rule can be uniquely identified by means of the name of the profile that contains the rule and the name of the rule. However, depending on the current context, the user does not always have to specify the exact location of a rule he wants to omit. If e.g., a rule named *add o*nly occurs in the *NV30 p*rofile, and the user wants to exclude that rule in another profile that derives the *NV30 p*rofile, the name of rule suffices. Otherwise, if the global profile also contains a rule called *add,* the user has to provide the exact location of the rule (i.e., ::*add* for the rule in the global profile and *NV30*::*add fo*r the rule in the *NV30* profile). Note that whenever the user derives a profile from other profiles, the profile will only inherit the rules that have not been omitted previously.

2.2.2. Rule

A rule describes a single step of the code generation or the code optimisation process. To specify the behaviour of a rule, the user must provide a search pattern to identify the instruction sequence that has to be compiled (or optimised), together with certain side conditions that have to be satisfied before the pattern matcher may apply the rule. Additionally, the user has to specify the cost function that assigns an integer cost to each generated alternative and the replace pattern that substitutes the matched instruction sequence.

```
RULE        ::= RULE_HEADER RULE_BODY
RULE_HEADER ::= rule RULENAME [RULE_EXTEND] [RULE_MASK]
RULE_EXTEND ::= ':' RULE_ID
RULE_MASK   ::= ':' '(' INTEGER ')'
RULE_BODY   ::= '{' [SEARCH] [CONDITION] [COST] [REPLACE] '}'
```

The rule definition is split into the rule header and the rule body. In the rule header, the user has to specify the name of the rule, which may not occur twice in the profile the rule belongs to, as mentioned in Section 2.2.1. Additionally, the user can derive the new rule from an existing rule that must not necessarily be included in the current profile. In contrast to a profile, a rule may only derive from one rule at a time, because tpmg would not be capable to clearly define the outcome otherwise. The child rule inherits the search pattern, the global condition function, the cost function and the replace pattern from the parent rule. If the parent rule does not exist, tpmg stops with a corresponding error message. Using the inheritance mechanism, the user can easily create aliases of any previously defined rule. Respecifying a specific property in the rule body, the user can even override the inherited rule specification. However, when deriving a rule from another, there are certain constraints the user has to keep in mind. The end of this section further discusses potential problems that might arise.

In some cases, the user wants to disable a rule right from the start, to prevent the generated pattern matcher from matching a basic block with that rule, whereas the application should decide during runtime whether the pattern matcher may use this rule. To disable a rule, the user has to define a bit mask in the rule header. Note that the bit mask may be constant expression. To re-enable a disabled rule, the application has to provide the pattern matcher with a bit mask, with which the pattern matcher determines whether a disabled rule may be used[27]. Using this mechanism, the user can easily enable or disable whole subsets of a profile.

The rule body specifies the runtime behaviour of the rule, which is determined by the search pattern, the global condition function, the cost function and the replace pattern. Apart from minor modifications, the search pattern syntax corresponds to the formal search pattern description (see Definition .1 in Chapter III).

```
SEARCH    ::= SEQUENCE
SEQUENCE  ::= '[' SPATTERNS ']'
            | '{' SPATTERNS '}'
SPATTERNS ::= [SPATTERN [',' SPATTERN]*]
```

27 Any rule may be used, if its bit mask either undefined or matches the provided bit mask. If $a = 0x01$ is a rule's bit mask, and $b = 0x11$ is the provided bit mask, the pattern matcher will use the masked rule, because $a \,\&\, b = 0x01 \neq 0$ (& is the *binary and* operator). If otherwise $b = 0x10$, the pattern matcher may not use the rule, because $a \,\&\, b = 0$. Note that a rule will be permanently disabled, if its mask is zero.

```
SPATTERN    ::= SEQUENCE | ITEM | WILDCARD
ITEM        ::= ('.' | CLASSNAME) ['(' EXPRESSION ')']
EXPRESSION ::= ASCII*
WILDCARD    ::= '*'
```

Because search patterns must contain at least one item pattern, tpmg rejects any rule file that contains a search pattern that does not satisfy this condition. An item pattern is defined through the class name of the object instance to match and an optional side condition, which must be a boolean expression in the target language. To match any object that is an instance of the input base class, the user may simply provide a `.` as class name. To keep tpmg as language independent as possible, tpmg does not verify if the side condition is syntactically or semantically correct and leaves that to the target language compiler.

With the index of a pattern, the user can access any non-sequence subpattern of a search pattern. Similar to the notation of the bison grammar, `$$` represents the current pattern – if applicable – and `$i` represents the *i*-th subpattern of a search pattern. If the *i*-th subpattern is an item pattern, the expression `$i` returns the matched instruction object. Otherwise, `$i` returns the number of objects that the corresponding wildcard pattern has consumed. However, the user must not access a pattern, if tpmg detects that the rule might not have matched that pattern at that point. Additionally, the user must not call any member function that modifies the matched object, so that the rule does not modify the input basic block. Note that the pattern matcher generator can only prevent the modification of the input basic block, if the target language provides a mechanism that marks an object as unmodifiable like the `const` modifier in C++.

Example 2.2.1

Let $\Sigma = \{A, B\}$ be the input alphabet, where both classes provide the functions *isValid* and *setValid* (the function name represents their purpose). Additionally, let the search patterns s_1, s_2 and s_3 be defined as follows:

s_1 := `[ A,` `B ($1->isValid()) ]`

s_2 := `{ A,` `B ($1->isValid()) }`

s_3 := `[ A ($$->setValid(true), true), B ($1->isValid()) ]`

Each search pattern matches the same instances of the classes *A* and *B* (with respect to the member function *isValid*), whereas s_2 does not care about their order of appearance. However, according to the above constraint, s_2 is not a valid search pattern. Because s_2 is an unordered sequence pattern, the second item pattern might be the first subpattern of s_2 that matches an instruction object. In that case, the side condition of $s_2[2]$ cannot call any member function of the object that the first item pattern matchers, because $s_2[1]$ has not yet matched an instance of *A*. So, whenever the second item pattern of s_2 matches first, it is invalid to evaluate the side condition of that item pattern. To prevent this kind of error in advance, tpmg rejects any item pattern side condition that attempts an unsafe pattern access.

Because the side condition of $s_3[1]$ modifies the matched object, the search pattern s_3 is also invalid. However, the pattern matcher generator is not able to detect the error, because tpmg does not collect any data about the capabilities of the instruction objects. Instead, the target language compiler will reject the generated pattern matcher, because the side condition of $s_3[1]$ attempts to modify an unmodifiable object.

To minimise the memory usage, tpmg reduces each search pattern to its normal form. If e.g., a sequence subpattern of a search pattern only comprises one pattern, tpmg replaces the subsequence by the inner pattern. If tpmg encounters an ordered sequence pattern that comprises at least two consecutive wildcard patterns or an unordered sequence pattern that contains more that one wildcard pattern during the normalisation process, tpmg rejects the rule file. The pattern matcher generator strictly rejects those search patterns, because on the one hand the excessive wildcard patterns are redundant, and on the other hand, the more wildcard patterns a search pattern consists of, the longer it takes to match the pattern, as Section 3.4 in Chapter III discusses[28]. By means of several search patterns, the following example demonstrates the normalisation process.

Example 2.2.2

Let $\Sigma = \{A, B, C\}$ the input alphabet. The following search patterns are either invalid or valid and can be normalised or not:

- The following search patterns can be normalised, as they consist of sequences that only comprise a single pattern (which may even be a sequence pattern itself).

  $s \coloneqq$ `[ A, [B], C ]`

  $t \coloneqq$ `[ A, {B}, C ]`

  It turns out that the search patterns s and t represent the same normalised pattern.

  $normalise(s) = normalise(t) =$ `[ A, B, C ]`

- If an ordered sequence pattern contains another ordered sequence pattern, the inner pattern is redundant. However, in any other case, tpmg must not flatten any sequence pattern that comprises more than one subpattern.

  $s \coloneqq$ `[ A, [B, C] ]`

  $t \coloneqq$ `[ A, {B, C} ]`

  $u \coloneqq$ `{ A, [B, C] }`

  $v \coloneqq$ `{ A, {B, C} }`

  So, tpmg normalises the search pattern s and does not modify t, u and v.

  $normalise(s) =$ `[ A, B, C ]`

  $normalise(p) = p$ with $p \in \{t, u, v\}$

- The pattern matcher generator rejects the following search pattern, because the normalised pattern contains two consecutive wildcard patterns.

  $s \coloneqq$ `[ [A, *], *, B ]`

  $normalise(s) =$ `[ A, *, *, B ]`

28 Why does tpmg reject the rule file at all? Instead, tpmg could simply remove the excessive wildcard patterns. However, if tpmg would remove a pattern, tpmg would implicitly change the indices of the remaining patterns. Not to confuse the user with varying indices, tpmg just rejects the rule file.

As tpmg normalises every search pattern, the user does not need to care about the complexity of the used search patterns. However, tpmg forces the user to use wildcard patterns sparsely, so that the generated pattern matcher processes basic blocks in an acceptable time.

```
CONDITION ::= '{' ASCII* '}'
COST      ::= '{' ASCII* '}'
```

The runtime behaviour of the rule is further determined by the global condition function that must return either *true* or *false* and the cost function that has to return an integer value. In addition to the item pattern side conditions, both functions may access any subpattern of the search pattern to aid their computation using the above notation. Similar to the item pattern conditions, tpmg does not verify the syntactical and semantic correctness of these functions. Again, the target language compiler has to take over this part.

```
REPLACE      ::= '[' [RPATTERNS] ']'
RPATTERNS    ::= RPATTERN [',' RPATTERN]*
RPATTERN     ::= RGUARD | RITEM
RGUARD       ::= if '(' EXPRESSION ')' REPLACE [else (RGUARD | REPLACE)]
RITEM        ::= '$' INTEGER [INITIALISERS]
               |       ITEM [INITIALISERS]
INITIALISERS ::= (':' INITIALISER)+
INITIALISER  ::= IDENTIFIER '(' EXPRESSION ')'
```

The replace pattern determines the final outcome of a rule. In contrast to the formal description, replace patterns are a special kind of ordered sequence patterns that may only comprise ordered sequence patterns and item patterns. To enable a rule to create different instruction sequences, subsequences of a replace pattern may be guarded by side conditions, so that the user does not have to write a new rule for each possible outcome. So, although replace patterns are syntactically sequence patterns, they are implicitly functions that generate different sequences of instructions depending on the current context.

While optimising the input basic block, the user might want to conserve previously matched objects, so that the rule does not have to allocate a new object instance with the same properties and to dispose of the matched object. Using the above pattern access notation, the user may specify the instruction object the rule should preserve and insert in the final basic block. If the accessed pattern is a wildcard pattern, tpmg rejects the replace pattern and aborts.

To create a new object instance, the user has to specify the class name and the arguments the replace pattern should pass to the constructor of the class. If the user does not provide any arguments, the replace pattern calls the default constructor of the desired class. To create an instance of the output base class, the user may simply provide a `.` as the target class name.

Both conserved and newly created objects can further be initialised with a mechanism that syntactically corresponds to the C++ constructor initialisation. However, in contrast to specifying the member variables to initialise, the user has to provide the name of a member function and the arguments that should be passed to that function. In this way, both the rule file and the application code remain readable and maintainable, as the user does not have to realise constructors that can take every possible combination of arguments.

Example 2.2.3

Let $\Sigma = \{Object\}$ be the instruction alphabet, where the *Object* class is defined as follows:

```
class Object
{
  public:
    Object (std::string name);
    void setValue (int value);
    int value (void) const;

  private:
    std::string m_name;
    int m_value;
};
```

Using the above mechanism, the user can then allocate and initialise a new instance of the *Object* class with the following replace pattern:

```
[ Object ("a"): setValue (12) ]
```

The generated pattern matcher will then create the new instance as follows:

```
Object *instance = new Object ("a");
instance->setValue (12);
```

In addition to the subpattern access, the user may also access both conserved and newly allocated object instances from within a replace pattern using the same notation. The index of an object instance is determined by the position within the replace pattern – analogously to the search pattern indices – disregarding the subsequence side conditions. To distinguish accesses to the search pattern from access to the replace pattern and to be able to access the search pattern from within the replace pattern, the indices of the subpatterns of the replace pattern start with $j+1$, iff j is the largest index that occurs in the search pattern.

Example 2.2.4

Let the *Object* class be defined as in Example .3, and s be the search pattern and r be the replace pattern that are defined as follows:

```
s := [ Object ]

r := [ Object ("a"),
       Object ("b"): setValue ($1->value()+$2->value()) ]
```

After the rule has finished to match the basic block, the replace pattern creates two object instances. When the generated code allocates the second object, it initialises the value of that object with the sum of the value of the matched item (`$1`) and the value of the first allocated object (`$2`).

Because subsequences within an replace pattern may be guarded by side conditions, it is consequently not always safe to access an object from such a sequence. If a rule would use the following replace pattern r', the generated pattern matcher could terminate abnormally.

```
r' := [ if ($1->value() > 0)
          [ Object ("a") ],
        Object ("b"): setValue ($1->value()+$2->value()) ]
```

If the value of the matched object (`$1`) is not positive, the replace pattern does not create the first object of the replace pattern. In that case, the object `$2` is undefined and the call to the member function *value* causes an invalid memory access. Although such an access is generally unsafe, tpmg does not forbid unsafe object accesses, because tpmg is not able to determine whether an invalid access will occur or not. Instead, tpmg only warns about possibly unsafe accesses and generates code that detects those accesses during runtime and that throws an exception to enable a controllable program termination. Providing tpmg with a special command line parameter, the user can disable this runtime access verification.

Note that the four rule properties are optional. Except for the global condition function[29], the user must specify the search pattern, the cost function and the replace pattern. If at least one of these properties is undefined, the specified rule is *virtual* and cannot be used by the pattern matcher. Although such a rule appears to be useless on its own, the rule can still serve as as a template from which the user can derive new rules that implement the missing properties.

As hinted at the beginning of this section, the user has to keep certain constraints in mind, when deriving rules from another. The following example demonstrates the two common pitfalls that might occur.

Example 2.2.5

Let $\Sigma_{in} = \{A, B\}$ be the input alphabet and $\Sigma_{out} = \{C, D\}$ be the output alphabet, whereas A is the input base class and C the output base class. It is assumed that every class, except A, provides the member function *check* that returns a boolean value. Furthermore, it is assumed that the user has defined the rule *base* as follows:

```
rule base
{
   search:  [ A, B ]
   replace: [ C, D ]
}
```

The rule *base* matches the object sequence *AB* and translates that sequence into the target alphabet sequence *CD*. Because the cost function is not defined, the rule is virtual and will thus not be used during runtime. Instead of refining the rule *base*, the user specifies other rules:

- First, the user derives the rule *first*, that implements the global condition function and the cost function. In contrast to the rule *base*, the rule *first* only accepts those object sequences *AB*, where the member function *check* of the matched *B* returns true:

  ```
  rule first : extends base
  {
     condition: { return $2->check(); }
     cost:      { return 2; }
  }
  ```

  As every instance of the class *B* provides the member function *check*, it is valid to derive the rule *first* from the rule *base* in this way. Additionally, the pattern matcher can make use of the rule *first*, because the user has specified all required properties.

29 If the user does not provide a global condition function and the rule does not inherit the condition function from another rule, tpmg assumes that the global condition is always true.

- Furthermore, the user derives the rule *second* from the rule *first* and replaces the inherited search pattern as follows:

```
rule second : extends first
{
   search: [ A ]
}
```

  However, the derived rule is not valid, because the global condition function accesses the second pattern of the search pattern, which only contains one item pattern in this rule. So, tpmg rejects the rule file and reports that the global condition function of the rule *second* cannot access the pattern that `$2` denotes. Thus, the user also has to override the inherited global condition function to make the rule definition valid.

- Instead, the user modifies the rule second so that the search pattern matches two instances of the class *A*:

```
rule second : extends first
{
   search: [ A, A ]
}
```

  On the first look, this rule definition appears to be valid, because the global condition function is now able to access the second subpattern of the search pattern. However, because the class *A* does not provide the member function *check*, the target language compiler will not compile the generated pattern matcher. Unfortunately, tpmg is not able to detect this kind of error, because the pattern matcher generator lacks the necessary information. Instead, tpmg can only warn the user that the pattern denoted by `$2` is not of the expected type *B*.

- Finally, the user defines the rules *third* and *fourth*. The rule *third* derives from the rule *first*, and modifies the inherited replace pattern such that the allocated instance of the class *C* is passed to the constructor of the class *D*. The rule *fourth* inherits the properties from the rule *third* and overrides the search pattern such that it accepts the object sequence *ABB*:

```
rule third : extends first
{
   replace: [ C, D ($3) ]
}

rule fourth : extends third
{
   search: [ A, B, B ]
}
```

  Because the rule *third* derives from a valid rule and its replace pattern is also valid, there is nothing wrong with that rule. The interesting question is now, which object instance is passed to the constructor of the class *D*, when the pattern matcher applies the rule *fourth*. According to the semantics of the `$` operator, the user would expect that the second instance of the class *B*, which the search pattern of the rule *fourth* matches, is passed to the constructor. If so, the search pattern would implicitly modify the inherited

replace pattern, which contradicts the common notion of inheritance[30]. However, as tpmg knows that the pattern denoted by `$3` belongs to a different context, in which the search pattern only comprised two objects, tpmg associates the pattern access with the correct object instance. Thus, if the user overrides the search pattern, tpmg does not require the user to respecify the inherited replace pattern, if the replace pattern contains inter-pattern accesses.

The rule inheritance is a powerful mechanism that enables the user to define the behaviour of the generated pattern matcher on a very high level of abstraction. So, the user no longer has to cope with the actual matching. Instead, the user simply has to identify the instruction patterns the pattern matcher has to replace.

However, besides the rules and the profiles, the user has to implement certain functions the generated pattern matcher implicitly calls during the matching process. The following section discusses why these functions are required and what they are used for.

2.2.3. Implicit Functions

The implicit section of the rule set contains the definition of three functions, of which the pattern matcher makes use while processing a basic block. Depending on its mode of operation, the generated pattern matcher does not require the user to implement each implicit function.

```
IMPLICIT  ::= implicit '{' [CONDITION] [COPY] [POSTPASS] '}'
COPY      ::= copy '{' ASCII* '}'
CONDITION ::= condition '{' ASCII* '}'
POSTPASS  ::= postpass '{' ASCII* '}'
```

To ensure that the pattern matcher does not generate invalid code, the pattern matcher first calls the implicit condition function, before the generated pattern matcher may apply a rule, whose search pattern contains at least one wildcard pattern. Example .6 demonstrates the necessity for the implicit condition function.

Example 2.2.6

Let $\Sigma = \{ADD, MUL, MAD\}$ be the instruction alphabet, where the instructions *ADD*, *MUL* and *MAD* and the rule *mad* are defined according to Example .6 in Chapter III. Remember that each instruction has a *target* register, a *first* and a *second* operand. The *MAD* instruction additionally has a *third* operand. Under the assumption that the constructor of each instruction class takes these properties as arguments, the tpmg implementation of the rule *mad* looks like the following.

```
rule mad
{
  search:  [ MUL,
             *,
             ADD ($$->target == $1->target && $$->first == $1->target) ]
  cost:    { return 1; }
  replace: [ MAD ($1->target, $1->first, $1->second, $3->second) ]
}
```

30 This means that an inherited property remains unmodified unless it has been explicitly overridden.

The rule mad tries to detect a *MUL* and an *ADD* instruction, where the target and the first operand of the *ADD* instruction equals the target of the *MUL* instruction, so that both of them may be combined to a *MAD* instruction. However, as Figure IV.4 shows, this side condition is too weak and does not prevent an invalid optimisation.

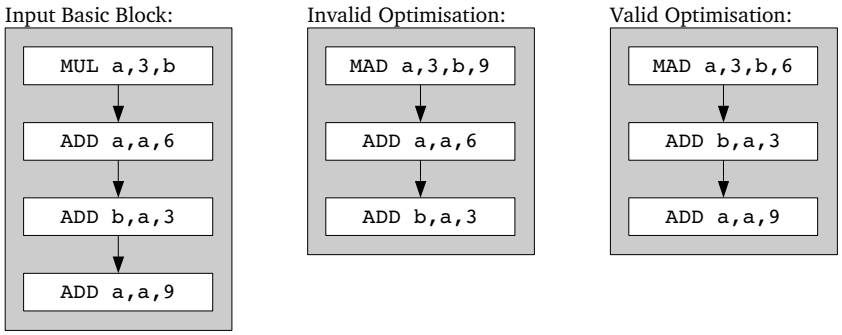

Figure IV.4: Invalid and valid optimisation of the input basic block.

Being applied to the input basic block, the search pattern produces two alternatives, whereas one alternative matches the first and the last instruction, and the other alternative matches the first two instructions. Thus, the pattern matcher can choose between two possible optimisations of the input basic block. However, the first optimisation is not valid, because the optimisation modifies the semantics of the input basic block. At the end of the optimised basic block, the value of the register b is $3*b + 18$. Instead, the value of that register should be $3*b + 9$. What went wrong?

After the pattern matcher had chosen the first alternative of the rule *mad*, the pattern matcher has virtually pushed the last instruction behind the first instruction, before the pattern matcher has applied the rule *mad*, which replaced both instructions with a *MAD* instruction. However, because the instructions on the path between the first and the last instruction both read and write the target register of the last instruction, the pattern matcher may not push that last instruction upwards. So, to rule out invalid alternatives, the rule *mad* also has to check, whether the instructions on the path between two matched instructions read or write the target register of the matched *ADD* instruction. Thus, only the second alternative is valid for the given input basic block.

Every rule that contains at least one wildcard pattern has to verify this special side condition. To prevent that the user forgets to specify this side condition and thus causes the generated pattern matcher to create invalid code sequences, the pattern matcher implicitly calls the condition function, whose purpose is to check whether instructions may be pushed upwards. The parameter of this function is a sequence of item and wildcard patterns in their order of occurrence in the current alternative. Before the pattern matcher inquires the global condition function of a rule, the pattern matcher first checks the implicit condition. If the user has not specified the implicit condition function, the pattern matcher assumes that this special side condition is always true.

In contrast to the implicit condition function, a pattern matcher only makes use of the copy and the postpass function, while optimising a basic block. The generated pattern matchers are designed to work hand in hand with a program analysis. To keep the program analysis data up to date, the program analysis has to be rerun after the pattern matcher has applied a rule while optimising a basic block. Thus, after applying a rule, the pattern matcher calls the postpass function, which receives a reference to the current basic block, so that the program analysis can be restarted.

To determine the global cost minimum while optimising a basic block, the pattern matcher has to investigate every possible alternative. Additionally, the pattern matcher has to create copies of the input basic block, so that the pattern matcher is able to compare the total cost of every alternative. Because a rule does usually not affect the whole basic block, the copies of the input basic block will share some instruction objects. To prevent that a rule disposes an object that several copies of the input basic block contain, the pattern matcher has to keep track of the number of basic blocks that share a specific object (reference count). Thus, a rule may only delete an object, if only one basic block contains that object. However, because the pattern matcher is supposed to work together with a program analysis, the reference count does not suffice, and the pattern matcher must additionally create copies of each instruction object, under the assumption that the used program analysis associates its data with the instruction objects. For that reason, the user has to specify the implicit copy function, if the pattern matcher should determine the global cost minimum while optimising a basic block. Example .7 shows why the pattern matcher has to copy the instruction objects as well.

Example 2.2.7

Let $\Sigma = \{ADD, MUL, MAD\}$ the instruction alphabet, where *ADD*, *MUL* and *MAD* are defined as in Example .6. The used pattern matcher optimises basic blocks over Σ and works hand in hand with a program analysis that computes the used definition sets for every instruction. To distinguish the instruction objects from each other, every instruction object has a unique identification number. For every instruction object $i \in \mathbb{N}$, $ud(i) \subseteq \mathbb{N}$ denotes the set of objects, whose target is an operand of the instruction object i.

Figure IV.5 shows the beginning of a basic block and two possible optimisations. Before the pattern matcher starts to optimise the given basic block, the *used definition* sets are defined as follows: $ud(1) = \emptyset$ (the definition of a is unknown), $ud(2) = \{1\}$ and $ud(3) = \{2\}$. There are two possible optimisations: The first optimisation modifies the first instruction and removes the second instruction, whereas the second optimisation combines the first and the second instruction to a *MAD* instruction. To compute the global minimum, the pattern matcher investigates both alternatives. At first, the pattern matcher copies the input basic block and removes the second instruction. The postpass function restarts the program analysis and sets $ud(3) = \{4\}$. Next, the pattern matcher applies the second optimisation and replaces the first two instructions with a *MAD* instruction. Then, the postpass function restarts the program analysis again and sets $ud(3) = \{5\}$.

So, the program analysis does not know that the third instruction is contained in two different basic blocks. For that reason, it is necessary that the pattern matcher creates copies of the instruction objects to synchronise the program analysis data with the generated basic blocks.

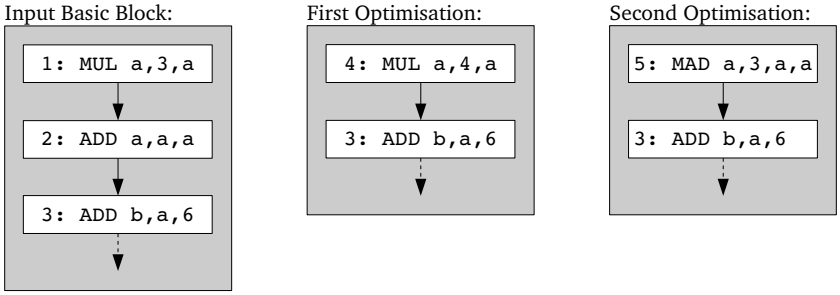

Figure IV.5: Different valid optimisations of the input basic block.

3. Generated Pattern Matcher

As hinted at the beginning of this Chapter in Section 1, the pattern matcher generator only creates C++ pattern matchers that depend on the STL and the tpmg template library. Thus, reading a rule file, tpmg compiles the specified pattern matcher into a C++ pattern matcher class. The pattern matcher generator puts the generated class into a custom namespace, to avoid name conflicts with the template library or the application, in which the pattern matcher will be embedded.

Creating a class for each rule, tpmg maps the rule inheritance on the class inheritance mechanism of C++, so that tpmg does not have to specify inherited properties of a rule twice. The pattern matcher generator does not explicitly create the predicate object automaton for each search pattern. Instead, the tpmg template library provides the mechanism to simulate the corresponding predicate object automaton (see Section 3.1 in Chapter III), so that tpmg only has to specify the search pattern. Using the tpmg template library, the user can even create custom pattern matchers without tpmg. However, implementing a pattern matcher in the description language is much more comfortable.

In contrast to the rules, tpmg does not represent profiles as classes. Instead, tpmg just creates a special initialisation function with which the application can specify the profile the pattern matcher should use. If the pattern matcher does not support the desired profile, the function throws an appropriate exception. Otherwise, the initialisation function allocates the required rules.

Depending on the input base class and output base class, the generated pattern matcher offers either two or four different basic block processing methods (see Section 3.3.1 and 3.3.2 in Chapter III). Independent of the used processing mode, the generated pattern matcher aborts processing the input basic block with an exception, if the pattern matcher cannot process the basic block any further. The thrown exception contains a reference to the object from which the processing could not be continued. If any rule tries to access a replace pattern that has not been allocated, the pattern matcher aborts the computation with another exception to prevent an invalid memory access (see Example .4 in Section 2.2.2). The exception reports the index of the pattern and the line of the rule file, from where the invalid access has been attempted.

The C++ implementation of the four basic block processing modes does not differ greatly from the pseudo-code implementation of Section 3.3.1 and Section 3.3.2 in Chapter III. In fact, the implementation of the two single-pass processing methods corresponds exactly to the pseudo-code realisation. On the contrary, the multi-pass processing functions differ from the theoretical implementation. While optimising a basic block, the generated pattern matcher does not compare the input basic block with the optimised basic block to determine whether to stop processing the basic block. Instead, the pattern matcher continues as long as the pattern matcher can apply at least one rule. However, this different behaviour might cause the pattern matcher to loop endlessly, if the pattern matcher is able to repeatedly apply a specific sequence of rules[31]. So, to prevent an endless loop, the user can additionally specify the maximum level of recursion that may occur while optimising a basic block.

Before embedding a generated pattern matcher in the application, the user has to ensure that the application satisfies certain prerequisites – discussed in Chapter V – that are required, so that the application works hand in hand with the generated pattern matcher.

31 In that case there must be flaw in the used profile. If the pattern matcher can apply a specific sequence of rules over and over again, the used rules do not seem to optimise the basic block at all.

4. Debugger Interface

To understand how the generated pattern matcher processes its input, the user can make use of the tpmg debugger interface that is also part of the tpmg template library. The user simply has to derive a new class from the tpmg debugger class and pass an instance of that class to the generated pattern matcher during runtime. Whenever something interesting happens, the pattern matcher calls the corresponding function of the given instance of the debugger class to inform the user about the recent event. Currently, the debugger interface can receive the following events:

- The event *currentRule* informs the debugger that a new rule starts to match the basic block. If the rule has finished, the pattern matcher emits the event *finishRule* and produces the event *applyRule*, when the rule is going to insert the replace pattern in the basic block.
- Whenever a rule produces a new alternative, the pattern matcher generates the event *newAlternative*. To indicate which alternative the active rule currently processes, the pattern matcher produces the event *currentAlternative*, whereas the pattern matcher emits the event *deleteAlternative* to report that the current rule cannot investigate an alternative any further.
- Before an item pattern may consume an item of the basic block, the current rule must first check the side condition of the item pattern. To inform the debugger whether the side condition is satisfied, the pattern matcher emits the event *checkItemPattern*. The event reports the current alternative, the index of the item pattern, the item, which the item pattern should match, and the result of the item pattern side condition.
- The events *matchItemPattern* and *matchWildcardPattern* indicate that an item pattern and a wildcard pattern respectively has matched an item of the basic block. Each of the two events reports which item has been matched, the index of the pattern and the alternative in which the item has been matched. Whenever, a rule marks a wildcard pattern as finished, the pattern matcher emits the event *finishWildcardPattern*.
- If the pattern matcher has finished processing a basic block, the event *finishBasicBlock* informs the debugger that the matching has finished and reports the total cost sum of the applied rules.

V. Compiler Integration

By means of the CGiS compiler, this chapter discusses the necessary steps to integrate a tpmg-generated pattern matcher into a compiler (or any other application). The first section introduces the prerequisites the compiler's internal architecture has to satisfy, before the user can start to embed the generated pattern matcher. Demonstrating the modifications to the CGiS compiler and comparing the performance of the generated pattern matchers with the original code generation and optimisation, the second section concludes this chapter.

1. Prerequisites

If the user wants to integrate a tpmg-generated pattern matcher into a compiler, the compiler implementation must satisfy four prerequisites. The main prerequisite concerns the programming language, in which the user has realised the compiler, whereas the other requirements are related to the internal architecture of the compiler.

- As hinted in Section 1 of Chapter IV, tpmg only generates C++ pattern matchers that depend on the tpmg template library. Thus, the user has to implement those parts of the compiler that communicates with the generated pattern matcher in C++. For the integration to be successful, the used compiler[32] must support templates and provide runtime type information (RTTI) as well. Additionally, the generated pattern matcher depends on an implementation of the STL.
- Because the generated pattern matcher expects a basic block as input, the compiler must represent the program code internally as a basic block control flow graph, or using a similar representation that makes use of basic blocks. Because the input and output basic blocks have the type `std::list`, the user either has to adjust the internal basic block representation or has to convert the used representation to and from the expected type.
- Furthermore, every processed and allocated object must be derived from a common base class, so that they can be combined in a `std::list`. If the class *A* is the base class of the input objects, and the class *B* is the base class of the output objects, the type of the input basic blocks is `std::list<A *>`, whereas the type of the output basic blocks is `std::list<B *>`. The classes *A* and *B* must not necessarily be different from each other.
- Finally, the rules of the generated pattern matcher must have access to the properties of the matched objects, so that the rules are able to query them while matching a basic block. To give the rules access to vital properties of the matched objects, the user can either declare the corresponding member variables as public or declare the rule classes as friends of the accessed instruction classes. However, the ideal solution to this problem is to give the rules access to these properties through special member functions.

  Additionally, the rules must be able to create new instruction object instances, because the pattern matcher would obviously not function otherwise.

If the implementation of the compiler already satisfies the above properties, the user only has to care about the pattern matcher specification and the integration of the generated pattern matcher into the compiler.

32 I have successfully tested tpmg-generated pattern matchers with the GNU C compiler (gcc) and the Microsoft Visual C++ compiler (msvc++).

2. Modifications to the CGiS Compiler

To demonstrate the benefit of tpmg-generated pattern matchers, I have extended the CGiS compiler by two pattern matchers. The first matcher compiles the abstract representation for the desired target platform, whereas the second matcher optimises the generated code, if possible.

Fortunately, cgisc satisfies most of the prerequisites (see Section 1) in advance, so I mainly had to cope with the specification of the two pattern matchers. The compiler has been implemented in the C++ programming language and internally represents the input program as a basic block control flow graph. A basic block is of the type `std::list<CirOperation>`, whereas the class `CirOperation`[33] is the common base class of all instruction objects. To further differentiate the abstract representation from the different target representations, cgisc introduces three classes that are derived from `CirOperation`:

- All those instruction objects that are used in the abstract representation of the input program are derived from the class `CirCGiSOperation`.
- The class `CirGPUOperation` is the common base class for each instruction object that represents an instruction of the target GPU architecture (e.g., A300 or NV30).
- Finally, each instruction object that represents an SSE operation derives from the class `CirSSEOperation`.

Besides specifying the code generation and code optimisation pattern matchers, I additionally had to adjust several instruction classes, so that the generated pattern matchers are able to access important properties of the matched instances of these instruction classes.

2.1. Code Generation

The compiler makes use of several classes that extend the class `CirCGiSOperation` to create an abstract representation of the input program. To generate code for the target architecture, cgisc compiles each instance of these instruction classes into a corresponding sequence of instruction objects. The compiler uses the following classes to represent an input program:

- The classes `CirCGiSUnOp`, `CirCGiSBinOp` and `CirCGiSTerOp` represent an unary, a binary and ternary operations respectively. Each instance of these classes has its own target register and a fixed number of operands. An instance additionally has a certain opcode that determines the type of the operation. If e.g., the opcode is `OP_UN_NEG` the instance represents an unary negation operation, whereas an instruction object with the opcode `OP_BIN_ADD` abstracts the addition of two operands.
- The CGiS compiler represents function parameters with the class `CirCGiSDataInOut`. Each instance owns the corresponding register and knows whether that register may only be read, only be written, or be both read and written. The class `CirGGiSIndex` is a special kind of function parameter that represents the index of a function parameter that originates from a stream. Instances of the class `CirCGiSDataComp` are necessary, if the output parameters of a function modify only certain parts of a stream.
- To abstract accesses to streams, the compiler makes use of the class `CirCGiSLookup`. An instance of this class knows which stream to access and in which register to write the result.

33 The prefix Cir abbreviates **C**GiS **i**nternal **r**epresentation.

- For architectures that do not support branching (e.g., NV30 or A300), the compiler makes use of the classes `CirCGiSGuardedAss` and `CirCGiSSetGuard` to realise the if-conversion (see Figure II.17). An instance of the class `CirCGiSGuardedAss` represents an assignment to a variable that depends on a certain side condition (guard), whereas an instance of the class `CirCGiSSetGuard` modifies the value of a guard depending on the value of a specific register.
- If the target architecture supports branching (e.g., NV40), the compiler makes use of the classes `CirCGiSIf`, `CirCGiSElse` and `CirCGiSEndIf`. Although cgisc internally represents branching with the control flow graph, the compiler needs these classes to generate the correct GPU code.
- If the target platform additionally supports loops (e.g., NV40), the CGiS compiler abstracts loops and breaks with the classes `CirCGiSLoopStart`, `CirCGiSBreak` and `CirCGiSLoopEnd`.

Depending on the desired target platform, cgisc compiles instances of the above instruction classes into sequences of objects that are either derived from the class `CirGPUOperation` or the class `CirSSEOperation`. Currently, cgisc supports the A300 (in development), NV30 and NV40 GPU architectures and is also capable of generating SSE code. So, the code generation pattern matcher comprises four profiles. Because the SSE profile is not part of my work, I will only discuss the GPU profiles in the following.

Each of the abstract representation classes are able to generate a corresponding sequence of GPU instructions, whereas the above classes allocate instances of the following derivatives of the class `CirGPUOperation` to represent the compiled program:

- The classes `CirGPUUnOp`, `CirGPUBinOp` and `CirGPUTerOp` stand representatively for an unary, a binary or a ternary operation. Each instance of these classes is determined through an opcode, a target register and a certain number of operands.
- Accesses to streams, which are realised using textures on the GPU, are implemented with instances of the class `CirGPUTexfetchOp`. An instance of this class knows from which texture and which coordinate to read the data from and in which register to write the data.
- The class `CirGPULoopOp` represents the start of a loop and thus corresponds to the class `CirCGiSLoopStart`. An instance of this class is defined by the maximum loop count.
- The class `CirGPUNullOp` corresponds to the branch and the remaining loop classes, whereas the opcode determines the type of an instance of this class. So, if the opcode is e.g., `OP_GPU_IF`, the instance corresponds to an instance of the class `CirCGiSIf`.

The specification of each profile orientates itself strictly at the cgisc code generation method, as presented in Section 2.2.3 in Chapter II. A GPU profile contains at least one rule for each instruction class that is able to generate a corresponding sequence of GPU instructions for the corresponding GPU architecture.

The advantages of the pattern matcher are quite obvious. At first, the instruction classes no longer need to know how to generate code for the desired target architecture, so that their only purpose is to represent an operation. This automatically leads to the second advantage. If the compiler should support a new hardware architecture, the user can easily introduce that architecture by adding a new profile to the pattern matcher[34]. Thus, the implementation stays maintainable over time, because it is no longer necessary to grindingly modify the code generation

34 Additionally, the user might have to add new instruction classes for the new platform.

function for each instruction object. Finally, the compiler is able to generate more efficient code, because the pattern matcher is able to take the surroundings of an instruction into account, before creating the corresponding code.

The following example displays an excerpt of the code generation pattern matcher and shows that the pattern matcher is able to create more efficient code than the original code generation method.

Example 2.1.1

Let NV30 be the desired target platform. Remember that this hardware architecture does not support a division operation, but provides the *RCP* operation that computes the multiplicative inverse of its operand. There is additionally no way to directly calculate the square root, as the NV30 only provides the *RSQ* operation that computes the multiplicative inverse square root of its operand (see Section 2.2.3 in Chapter II). Thus, the pattern matcher must treat division and square root specially, by creating an auxiliary *RCP* instruction. However, if the pattern matcher encounters a division through a square root, the auxiliary *RCP* instructions are not required. So, the NV30 profile contains the following rules[35], amongst others:

```
profile NV30
{
  rule binary_div
  {
    search:  [ CirCGiSBinOp ($$->opcode() == OP_BIN_DIV) ]
    cost:    { return 2; }
    replace: [ CirGPUUnOp  (OP_GPU_RCP, new CirSymReg (TYPE_FLOAT),
                            $1->second_operand()),
               CirGPUBinOp (OP_GPU_MUL, $1->target(),
                            $1->first_operand(), $2->target()) ]
  }
  rule binary_div_sqrt
  {
    search:  [ CirGPUUnOp (($$->opcode() == OP_UN_SQRT)),
               *,
               CirGPUBinOp (($$->opcode() == OP_BIN_DIV)
                            && ($$->second_operand() == $1->target())) ]
    cost:    { return 2; }
    replace: [ CirGPUUnOp  (OP_GPU_RSQ, new CirSymReg (TYPE_FLOAT),
                            $1->operand()),
               CirGPUBinOp (OP_GPU_MUL, $3->target(),
                            $3->first_operand(), $4->target()) ]
  }
  rule unary_sqrt
  {
    search:  [ CirGPUUnOp ($$->opcode() == OP_UN_SQRT) ]
    cost:    { return 2; }
    replace: [ CirGPUUnOp (OP_GPU_RSQ, new CirSymReg (TYPE_FLOAT),
                           $1->operand()),
               CirGPUUnOp (OP_GPU_RCP, $1->target(),
                           $2->target()) ]
  }
  ...
}
```

35 The rules have been simplified a bit to make them easier to read.

The rules *unary_sqrt* and *binary_div* implement the original cgisc code generation. Whenever the pattern matcher encounters an unary square root operation, the rule *unary_sqrt* first creates an unary operation that assigns the multiplicative inverse of the operand to a new symbolic register and then creates a auxiliary unary operation that inverts the result of the first one. The rule *binary_div* functions similarly. Additionally, the profile contains the rule *binary_div_sqrt* that handles divisions through square roots separately.

Let the function *divide* be defined as follows:

```
function divide (in float a, in float b, out float c)
{
  c = a/sqrt(b);
}
```

Not using the code generation pattern matcher, the compiler would generate the following shader program for the above function:

```
!!ARBfp1.0
TEMP _A_1_1,_A_1_2;
TEX _A_1_1.y,fragment.texcoord[0].xyxx,texture[0],RECT;
TEX _A_1_2.x,fragment.texcoord[0].xyxx,texture[0],RECT;
TEX result.color.xyzw,fragment.texcoord[0].xyxx,texture[0],RECT;
RSQ _A_1_1.x, _A_1_1.y;
RCP _A_1_1.y, _A_1_1.x;
RCP _A_1_1.x, _A_1_1.y;
MUL _A_1_1.y, _A_1_2.x, _A_1_1.x;
MOV result.color.z, _A_1_1.y;
END
```

A closer look at the generated code reveals that the two RCP instructions (marked red) are redundant. Thus, the pattern matcher contains the additional rule *binary_div_sqrt* to prevent the pattern matcher from generating inefficient code. So, the pattern matcher creates the following shader program:

```
!!ARBfp1.0
TEMP _A_1_1,_A_1_2;
TEX _A_1_1.y,fragment.texcoord[0].xyxx,texture[0],RECT;
TEX _A_1_2.x,fragment.texcoord[0].xyxx,texture[0],RECT;
TEX result.color.xyzw,fragment.texcoord[0].xyxx,texture[0],RECT;
RSQ _A_1_1.x, _A_1_1.y;
MUL _A_1_1.y, _A_1_2.x, _A_1_1.x;
MOV result.color.z, _A_1_1.y;
END
```

The remaining rules of the NV30 profile are strictly oriented at the code generation function of the abstract representation classes. As the A300 architecture does not differ from the NV30 with respect to the supported instructions, the A300 profile is simply an alias for the NV30 profile. However, because the NV40 architecture differs from the NV30 architecture, the corresponding NV40 profile overrides the NV30 profile, as the following example shows.

Example 2.1.2

On the one hand, the NV40 profile has to cope with instances of the branch and loop classes (see above), because the NV40 architecture supports both branches and loops. On the other

hand, the NV40 architecture provides a division operation (DIV), which makes it possible to divide two operands through each other with a single instruction. Thus, the NV40 profile inherits the rules of the NV30 profile and omits the rule *binary_div*. Additionally, the profile contains its own *binary_div* rule that extends the NV30 *binary_div* rule. So, the NV40 profile looks like following:

```
profile NV40 : extends NV30
{
  omit NV30::binary_div;

  rule binary_div : extends NV30::binary_div
  {
    cost:    { return 1; }
    replace: [ CirGPUBinOp (OP_GPU_DIV,
                            $1->target(),
                            $1->first_operand(), $1->second_operand()) ]
  }

  ...
}
```

Let the function *divide* be defined as follows:

```
function divide (in float a, in float b, out float c)
{
  c = a/b;
}
```

When generating code for the NV30 architecture, the pattern matcher generates the following shader code:

```
!!ARBfp1.0
TEMP _A_1_1,_A_1_2;
TEX _A_1_1.y,fragment.texcoord[0].xyxx,texture[0],RECT;
TEX _A_1_2.x,fragment.texcoord[0].xyxx,texture[0],RECT;
TEX result.color.xyzw,fragment.texcoord[0].xyxx,texture[0],RECT;
RCP _A_1_1.x, _A_1_1.y;
MUL _A_1_1.y, _A_1_2.x, _A_1_1.x;
MOV result.color.z, _A_1_1.y;
END
```

Making use of the NV40 profile, the pattern matcher replaces the red marked instructions with a single DIV operation and generates the folbwing code:

```
!!ARBfp1.0
OPTION NV_fragment_program2;
TEMP _A_1_1,_A_1_2;
TEX _A_1_1.y,fragment.texcoord[0].xyxx,texture[0],RECT;
TEX _A_1_2.x,fragment.texcoord[0].xyxx,texture[0],RECT;
TEX result.color[0].xyzw,fragment.texcoord[0].xyxx,texture[0],RECT;
DIV _A_1_1.y, _A_1_2.x, _A_1_1.y;
MOV result.color[0].z, _A_1_1.y;
END
```

2.2. Code Optimisation

The code optimisation pattern matcher combines several optimisations to optimise a basic block that contains instances of the class `CirGPUOperation`. Currently, the pattern matcher does not make use of a program analysis, because the integration of PAG generated analyses is not yet finished. So, the code optimisation pattern matcher is currently not as effective as it could be, because the pattern matcher cannot verify whether a register is still live[36]. However, using the following optimisations, the pattern matcher is still able to produce reasonable results:

- **Nop Detection**

  During the code generation, the compiler might generate instructions that assign the contents of a register to themselves. Because these instruction effectively do nothing, the pattern matcher may safely remove them. The following table demonstrates which instructions the optimisation pattern matcher may remove.

Input Instruction Sequence	*Output Instruction Sequence*
`MOV A.x, A.x;`	ε
`MOV A.xy, A.xyxx;`	ε
`MOV A.x, A.xyxx;`	ε *The pattern matcher may remove this instruction, because the y-component of the register A is unused.*
`MOV A.xy, A.xzxx;`	`MOV A.xy, A.xzxx;` *The instruction is not a nop, because the z-component is assigned to the y-component.*

- **Dead Code Elimination**

  Whenever the pattern matcher detects two adjacent instructions, where the second instruction overrides the first – i.e., if the second one overwrites the result of the first one – and the second one does not make use of the results of the first one, the pattern matcher may safely remove the first instruction.

Input Instruction Sequence	*Output Instruction Sequence*
`ADD A.xy, A.yxxx, B.yxxx;` `MOV A.xy, B.xyxx;`	`MOV A.xy, B.xyxx;`

`ADD A.xy, A.yxxx, B.yxxx;`
`MOV A.xy, A.yxxx;`
	`ADD A.xy, A.yxxx, B.yxxx;`
	`MOV A.xy, A.yxxx;`
	The result of the second instruction depends on the result of the first instruction, so the pattern matcher must not remove the first instruction.

`ADD A.xy, A.yxxx, B.yxxx;`
`MOV A.x, B.x;`
	`ADD A.xy, A.yxxx, B.yxxx;`
	`MOV A.x, B.x;`
	The first instruction may not be removed, as the second instruction does not override the first instruction.

36 That is because the pattern matcher processes the program representation on a basic block level and does not know about the control flow graph.

- **Copy Elimination**

  To reduce the number of instructions the pattern matcher tries to remove unnecessary copy operations. If the pattern matcher detects three adjacent instructions, where the second instruction copies the result of the first instruction and the third instruction overrides the first instruction without using the target of the first instruction, the pattern matcher may both remove the second instruction and replace the target of the first instruction with the target of the second instruction.

Input Instruction Sequence	*Output Instruction Sequence*

`MUL A.x, A.y, A.z;`
`MOV B.x, A.x;`
`ADD A.x, B.z, B.x;`
	`MUL B.x, A.y, A.z;`
	`ADD A.x, B.z, B.x;`

`MUL A.x, A.y, A.z;`
`MOV B.x, A.x;`
`ADD A.x, A.x, B.y;`
	`MUL A.x, A.y, A.z;`
	`MOV B.x, A.x;`
	`ADD A.x, A.x, B.y;`
	Because the ADD instruction depends on the result of the MUL instruction, the target of the MUL instruction must not be replaced.

- **Copy Propagation**

  To further increase the speed of the generated code, the pattern matcher detects all those operands that are actually copies of another register. Thus, the pattern matcher propagates copies as far as possible to the bottom of the basic block.

Input Instruction Sequence	*Output Instruction Sequence*

`MOV A.xy, B.xyxx;`
`ADD A.xy, A.yxxx, B.yxxx;`
	`MOV A.xy, B.xyxx;`
	`ADD A.xy, B.yxxx, B.yxxx;`
	The pattern matcher replaces the first operand with the actual content of the accessed register. Note that the first instruction is actually dead and can be removed in the next step (dead code elimination).

`MOV C.xy, B.yxxx;`
`MUL A.xy, C.yxxx, B.yxxx;`
	`MUL A.xy, B.xyxx, B.yxxx;`
	`MOV C.xy, B.yxxx;`
	After replacing C.yxxx with B.xyxx, the pattern matcher pushes the MOV instruction downwards.

`MOV C.xy, A.yxxx;`
`SUB A.xy, C.yxxx, B.yxxx;`
	`MOV C.xy, A.yxxx;`
	`SUB A.xy, A.xyxx, B.yxxx;`
	Because the second instruction modifies both the x- and y-component of A, the MOV instruction must not be pushed downwards.

`MOV C.xy, A.yx;`
`SUB A.xy, C.yz, B.yx;`
	`MOV C.xy, A.yx;`
	`SUB A.xy, C.yz, B.yx;`
	Similar to the previous example, the pattern matcher cannot propagate the contents of C, because the value of C.z is unknown.

- **Constant Propagation**

  Being a special case of the copy propagation, this optimisation tries to propagate constants throughout the basic block.

Input Instruction Sequence	*Output Instruction Sequence*

`MOV B.x, {2}.x;`
`MUL B.y, {3}.x, B.x;`
	`MUL B.y, {3}.x, {2}.x;`
	`MOV B.x, {2}.x;`
	After inserting the value of B.x, the pattern matcher pushes the MOV instruction downwards to further propagate the constant.

`MOV B.xy, {2, 3}.xyxx;`
`MUL B.y, A.x, B.x;`
	`MOV B.xy, {2, 3}.xyxx;`
	`MUL B.y, A.x, {2}.x;`
	The pattern matcher inserts the constant, but cannot push it downwards, because the MUL instruction partially overwrites the result of the MOV instruction.

- **Constant Vectorisation**

  Whenever the pattern matcher detects two *MOV* instructions that assign a constant to the same register, the pattern matcher combines them to a single *MOV* instruction.

Input Instruction Sequence	*Output Instruction Sequence*
`MOV B.x, {2}.x;` `MOV B.w, {3}.x;`	`MOV B.xw, {2, 3}.xxxy;`
`MOV B.xy, {2, 3}.xyxx;` `MOV B.yz, {4, 5}.xxyx;`	`MOV B.xyz, {2, 4, 5}.xyzx;`

- **Constant Folding**

  The optimisation pattern matcher folds each operation whose operands are constant.

Input Instruction Sequence	*Output Instruction Sequence*
`MUL B.y, {3}.x, {2}.x;`	`MOV B.y, {6}.x;`
`SLT B.x, {2}.x, {1}.x;`	`MOV B.x, {0}.x;`

- **MAD Combination**

  According to the rule *mad* of Example .6 in Chapter IV, the pattern matcher combines an *ADD* and a *MUL* instruction to an *MAD* instruction.

Input Instruction Sequence	*Output Instruction Sequence*
`MUL A.x, B.x, {2}.x;` `ADD A.x, {3}.x, A.x;`	`MAD A.x, B.x, {2}.x, {3}.x;`

`MUL A.x, B.x, {2}.x;`
`SUB B.x, A.x, {2}.x;`
`ADD A.x, {3}.x, A.x;`
	`MUL A.x, B.x, {2}.x;`
	`SUB B.x, A.x, {2}.x;`
	`ADD A.x, {3}.x, A.x;`
	The pattern matcher must not combine the two instructions, as the SUB instruction depends on the result of the MUL instruction.

- **Condition Optimisation**

  This optimisation only applies to the NV40 architecture. It reduces the required number of instructions that are related to a branch test. In contrast to other architectures, the GPU performs the branch test on a special condition register, whose contents must be determined beforehand – to do that, the character `C` must be added to the name of the corresponding instruction – as the following example shows:

```
SGT A.x, A.x, {2}.x;
MOVC A.x, A.x;
IF GT.x;
  MOV A.x, {0};
ELSE;
  MOV A.x, {1};
ENDIF;
```

  After executing the above code, the value of the *x*-component of the register *A* is 0, if the value of that component was greater than 2, and is 1 otherwise. However, as each instruction is able to modify the condition register, the above code is slightly inefficient. Pushing the modification of the condition register as far as possible to the top, the condition optimisation tries to remove unnecessary *MOVC* instructions.

Input Instruction Sequence	*Output Instruction Sequence*
`SGT A.x, A.x, {2}.x;` `MOVC A.x, A.x;`	`SGTC A.x, A.x, {2}.x;`

`SLT A.x, B.x, {2}.x;`
`MOVC B.x, B.x;`
`MOVC A.x, A.x;`
	`SLT A.x, B.x, {2}.x;`
	`MOVC B.x, B.x;`
	`MOVC A.x, A.x;`
	In this case, the pattern matcher must not perform the previous optimisation, because the condition register would then be B and not A. However, the first MOVC instruction is obsolete and could be removed by another optimisation.

Apart from the copy and constant propagation, the discussed optimisations can be easily realised. The following tpmg code sequence displays an excerpt of the NV30 optimisation profile. The rule *dead_code* demonstrates how simple the implementation of the dead code elimination optimisation actually is, whereas the functions *overrides* and *uses_target* verify whether the first matched instruction may be removed.

```
profile NV30
{
  rule dead_code
  {
    search:    [ CirGPUOperation ($$->target() != NULL),
                 CirGPUOperation ($$->target() != NULL) ]
    condition: { return overrides($2, $1) && !uses_target($2, $1); }
    cost:      { return -1; }
    replace:   [ $2 ]
  }

  ...
}
```

Unfortunately, the tpmg implementation of both the copy and constant propagation optimisation is not that straightforward. The main problem is to prevent the pattern matcher from reverting a step of the optimisation right after that step has been performed, which finally causes the pattern matcher to loop endlessly. This problem only occurs, if the pattern matcher detects two adjacent unary instructions that do not interfere with each other. Table V.1 shows an example input basic block that would cause the pattern matcher to loop forever.

Basic Block (initial)	***Basic Block (after first step)***	***Basic Block (after second step)***	...

`MOV A.x, {2}.x;`
`MOV B.x, {3}.x;`
`...`
	`MOV B.x, {3}.x;`
	`MOV A.x, {2}.x;`
	`...`
		`MOV A.x, {2}.x;`
		`MOV B.x, {3}.x;`
		`...`
			...

Table V.1: Endless loop during the constant propagation optimisation.

When the pattern matcher starts to optimise the initial version of the basic block, the pattern matcher will apply the constant propagation rule at first. Because there is no value analysis available at the time of writing, this rule tries to push constants as far as possible to the bottom of the basic block. So, the constant propagation optimisation pushes the first *MOV* instruction after the second one and leaves the residue of the basic block to the other optimisation rules. However, after the pattern matcher has finished the first optimisation pass, the constant propagation rule can be applied again. So, the pattern matcher automatically reverts the effect of the first iteration. To prevent these endless loops, every unary operation has to remember the last obstacle the operation has hit, so that the pattern matcher is able to detect whether a rule is going to revert the effect of a previous optimisation step.

Altogether, the advantages (high level of abstraction, higher maintainability), which the tpmg implementation of the above optimisations offers, certainly outweigh the small organisational effort that is necessary to prevent the pattern matcher from looping endlessly. Note that this behaviour of the generated pattern matcher arises from its design and is thus not a bug. With a working value analysis, the constant propagation rule could be refined such that it is not required to push constants downwards in the basic block.

2.3. Competitive Comparison

By means of 20 test cases, the following three sections compare the original code generation and code optimisation method of the CGiS compiler with the code generation and code optimisation pattern matcher with respect to their runtime and the efficiency of the generated code. All but two of the test cases originate from examples and regression tests of the CGiS compiler CVS repository. The following two test series – one for the NV30 and the other for the NV40 profile – have been executed on an AMD Athlon 64 3700+ and a NVIDIA GeForce 6800 Ultra. The A300 profile has been excluded, because it resembles the NV30 profile.

2.3.1. Code Generation

As hinted in Section 2.1, the code generation pattern matcher implements the original code generation method, apart from a minor exception. So, it is expected that the pattern matcher creates the same code most of the time, whereas the pattern matcher has been designed such that it works best when determining the local cost minimum and using the first-match policy. The following two tables show how much time the original code generation method and the pattern matcher take to compile abstract instructions into GPU instructions.

Test	*Abstract Instr.*	*Code Generation (original)*		*Code Generation (tpmg)*		
		GPU Instr.	*Time (µs)*	*GPU Instr.*	*Time (µs)*	*Factor*
1	11	8	37	8	181	4.9
2	12	18	86	18	160	2.1
3	12	10	79	10	173	2.2
4	12	9	47	9	174	3.7
5	12	23	44	23	203	4.6
6	13	15	74	15	177	2.4
7	14	10	46	10	228	5.0
8	16	13	78	13	260	3.3
9	19	11	57	11	340	6.0
10	20	16	76	16	280	3.7
11	25	20	82	20	382	4.7
12	26	18	73	18	329	4.5
13	28	18	69	16	497	7.2
14	32	24	237	24	1279	5.4
15	36	32	131	32	558	4.3
16	52	40	154	40	727	4.7
17	59	39	137	35	911	6.7
18	73	44	158	44	1885	11.9
19	85	75	242	75	1163	4.8
20	105	99	321	99	1338	4.2
Average	33.1	27.1	111	26.8	563	5.1

Table V.2: Comparison of the original and the tpmg NV30 code generation.

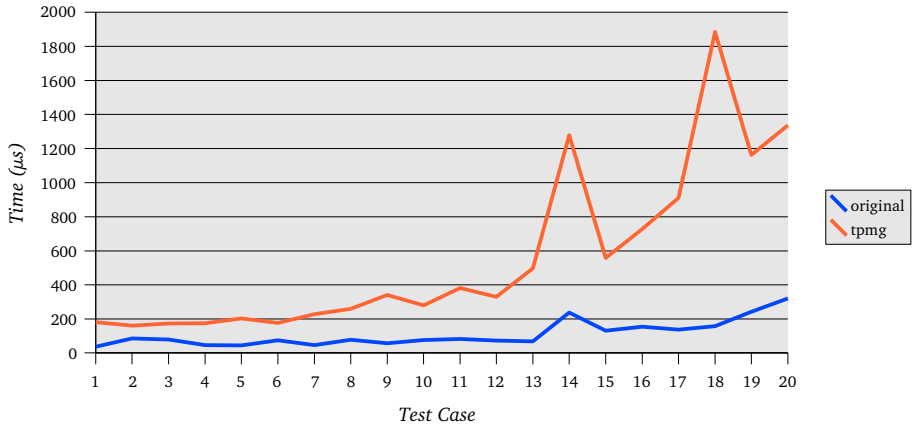

Figure V.1: Original and tpmg NV30 code generation time.

Table V.2 shows how the original code generation method compares to the pattern matcher when generating code for the NV30 hardware architecture. Apart from the test cases 13 and 17, the code generation pattern matcher has produced the same number of instructions. This difference arises from the pattern matcher's ability to create more optimal NV30 code when it comes to divisions of square roots (see Example .1). According to Figure V.1, which visualises the compile time of each test case, the execution time of the code generation pattern matcher scales linearly with the number of abstract instructions to compile. This corresponds to the runtime analysis (see Section 3.4 of Chapter III). Test case 18 represents the only exception of this observation, whereas the reason for the divergence is unclear[37].

Taking 17μs to compile an instruction on average, the code generation pattern matcher is about five times slower than the original code generator. However, because the runtime of the pattern matcher averages only 1.59% of the total processing time, this increase in runtime does not have a great impact on the overall runtime of the CGiS compiler.

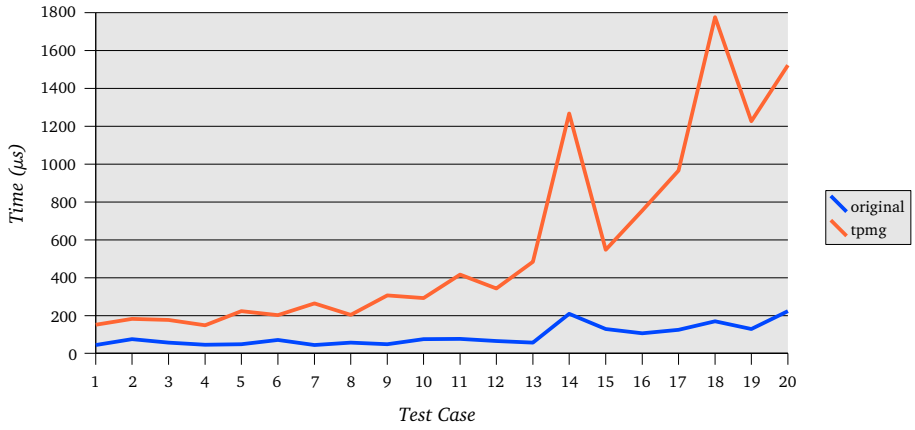

Figure V.2: Time to generate NV40 code (original and tpmg code generator).

Because the NV40 profile of the code generation pattern matcher comprises more rules, as the pattern matcher additionally has to cope with branches and loops, the pattern matcher needs

37 While compiling test case 18, the pattern matcher neither needs to backtrack nor makes use of rules whose search pattern contains a wildcard pattern.

more time to generate code for the NV40 architecture, as Table V.3 shows. As expected, the pattern matcher produces the same instruction for each test case, except for the test cases 13 and 17 for the same reason as above. A comparison of the results shown in Table V.2 and Table V.3 reveals that the CGiS compiler generates less instructions when compiling a program for the NV40 architecture (see test cases 11 to 14, 16, 17, 19 and 20). The main reason for this difference is that cgisc does not need to perform the if-conversion (as Figure II.17 in Chapter II demonstrates), because the NV40 architecture supports branches. When compiling test case 19, the compiler thus creates 75 NV30 instructions, but only 58 NV40 instructions.

According to Figure V.2, the execution time of the code pattern matcher scales linearly with the number of instructions to compile, whereas test case 18 diverges as above. As Table V.3 shows, the pattern matcher is about six times slower than the original code generator when generating NV40 code. On average, the code generation pattern matcher compiles an abstract instruction into an NV40 instruction in 18μs. However, taking less than 2% of the total processing time to compile the abstract representation, the pattern matcher virtually does not slow down the CGiS compiler at all.

Test	*Abstract Instr.*	*Code Generation (original)*		*Code Generation (tpmg)*		
		GPU Instr.	*Time (μs)*	*GPU Instr.*	*Time (μs)*	*Factor*
1	11	8	45	8	152	3.4
2	12	18	74	18	182	2.6
3	12	10	57	10	177	3.1
4	12	9	46	9	148	3.2
5	12	23	48	23	224	4.7
6	13	15	71	15	202	2.8
7	14	10	44	10	264	6.0
8	16	13	57	13	204	3.6
9	19	11	49	11	306	6.2
10	20	16	75	16	292	3.9
11	25	18	77	18	417	5.4
12	26	17	66	17	343	5.2
13	28	17	57	16	484	8.5
14	32	22	210	22	1268	6.0
15	36	32	129	32	547	4.2
16	50	36	107	36	754	7.0
17	59	36	125	34	966	7.7
18	73	44	170	44	1777	10.5
19	81	58	129	58	1227	9.5
20	105	93	223	93	1523	6.8
Average	32.8	25.3	93	25.2	573	6.2

Table V.3: Comparison of both NV40 code generation methods.

2.3.2. Code Optimisation

Displaying the number of generated instructions and the necessary time, the following two tables demonstrate the runtime of the code optimisation pattern matcher in comparison with the original code optimisation method.

Note that the results presented in Table V.4 and Table V.5 strongly depend on the preceding code generation phase. Thus, if the original code generator creates 18 instructions where the pattern matcher generates only 16 instructions, the original code optimiser has to optimise 18 instructions, whereas the code optimisation pattern matcher must only process 16 instructions. So, the effectiveness of the code optimisers depend on the outcome of the corresponding code generators.

In contrast to the code generation pattern matcher, the code optimisation pattern matcher is designed to determine either the local or the global cost minimum. However, it turns out that the pattern matcher works too inefficient when optimising a basic block with respect to the global cost minimum.

Test	*Code optimisation (original)*		*Code Optimisation (local cost minimum)*			*Code Optimisation (global cost minimum)*		
	Instr.	*Time (µs)*	*Instr.*	*Time (µs)*	*Factor*	*Instr.*	*Time (µs)*	*Factor*
1	6	52	7	178	3.4	7	241	4.6
2	16	98	18	328	3.3	18	302	3.1
3	8	93	10	107	1.2	10	99	1.1
4	5	59	9	138	2.3	9	161	2.7
5	21	91	22	1149	12.6	22	1188	13.1
6	13	93	15	282	3.0	15	290	3.1
7	7	61	8	224	3.7	8	826	13.5
8	11	111	13	432	3.8	13	509	4.6
9	10	55	4	2087	37.9	4	1233	22.4
10	13	106	13	457	4.3	13	994	9.4
11	17	105	14	2364	22.5	14	18007	171.5
12	14	74	18	250	3.4	18	276	3.7
13	16	67	7	2783	41.5	7	5708	85.2
14	19	191	19	8220	43.0	19	729044	3817.0
15	31	89	32	8157	91.7	32	7933	89.1
16	34	164	35	2116	12.9	35	2765	16.9
17	36	118	24	5085	43.1	24	47989	406.7
18	42	106	28	8862	83.6	28	9863	93.0
19	65	607	72	3579	5.9	72	1294389	2132.4
20	90	499	90	1988	4.0	90	2577	5.2
Average	23.7	142	23.0	2439	17.2	23.0	106219	748.0

Table V.4: Comparison of the original and the tpmg NV30 code optimisation.

Table V.4 shows how fast and how strong the original code optimiser and the pattern matcher optimise the previously generated NV30 code. A closer look at the results reveals that the code optimisation pattern matcher only optimises about half of the test cases (9 to 11, 13, 14, 17, 18 and 20) as good as or better than the original code optimiser, independent from the cost minimum. The main reason is that the pattern matcher processes the code on a basic block level and does not make use of a program analysis. So, the pattern matcher is currently not able to perform certain optimisations. However, reducing the code by about 14.2%, the code optimisation pattern matcher is still slightly more effective than the original code optimiser, which reduces the program code by about 12.5%.

Additionally, Table V.4 shows that, independent from the target minimum, the pattern matcher generates the same number of instructions. Because the pattern matcher generally takes too long to determine the global cost minimum (on average about 748 times longer than the original code optimiser), the local cost minimum processing method is the processing method of choice. So, I will only refer to the local cost minimum processing method in the following.

As Figure V.3 depicts, there is no linear correlation between the runtime of the code optimisation pattern matcher and the number of instructions to optimise. Although test case 16 comprises more instructions than test case 17 (see Table V.2), the pattern matcher takes longer to optimise test case 17. This observation correlates to the runtime analysis, which predicts that the runtime of the local cost minimum processing method depends on the processing passes, which cannot be easily determined beforehand (see Section 3.4 of Chapter III).

Being on average about 17 times slower than the original code optimiser, the code optimisation pattern matcher optimises a NV30 instruction in 91μs while determining the local cost minimum. Although this appears to be pretty slow, the runtime of the pattern matcher only accounts for 4.9% of the total processing time of the CGiS compiler.

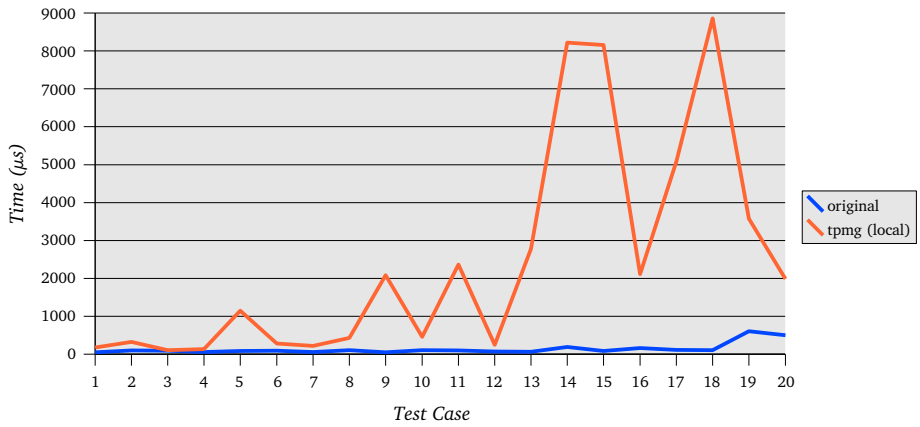

Figure V.3: Original and tpmg NV30 code optimisation time.

Table V.5 shows that the code optimisation pattern matcher generally optimises NV40 code slower but more effective than NV30 code. However, the pattern matcher is only able to optimise about half of the test cases as good as or better than the original code optimiser for the same reasons as above. However, determining the local cost minimum, the pattern matcher is able to reduce the program code by about 15.9% and is thus more effective than the original code optimiser, which reduces the code by about 13.0%.

A closer look at the test cases 14, 16 and 18 reveals that the pattern matcher produces more optimal results when determining the global cost minimum. However, because the effect is quite

minimal, and the pattern matcher takes too long to optimise the program code with respect to the global cost minimum (on average 596 times longer than the original code optimiser), the local cost minimum processing method is again the processing method of choice. So, I will only refer to the local cost minimum processing method in the following.

Test	*Code optimisation (original)*		*Code Optimisation (local cost minimum)*			*Code Optimisation (global cost minimum)*		
	Instr.	*Time (µs)*	*Instr.*	*Time (µs)*	*Factor*	*Instr.*	*Time (µs)*	*Factor*
1	6	53	7	177	3.3	7	258	4.9
2	16	98	18	335	3.4	18	314	3.2
3	8	94	10	122	1.3	10	122	1.3
4	5	61	9	173	2.8	9	187	3.1
5	21	85	22	1335	15.7	22	2238	26.3
6	13	92	15	352	3.5	15	319	3.5
7	7	63	8	227	3.6	8	973	15.4
8	11	92	13	510	5.5	13	576	6.3
9	10	52	4	1159	22.3	4	2537	48.8
10	13	104	13	477	4.6	13	1069	10.3
11	15	101	13	1830	18.1	13	11965	118.5
12	13	73	17	243	3.3	17	218	3.0
13	15	66	7	3436	52.1	7	6779	102.7
14	17	207	18	7694	37.2	17	863443	4171.2
15	31	86	32	9525	110.8	32	10026	116.6
16	30	133	32	2510	18.9	31	3519	26.5
17	33	113	21	5046	44.7	21	47658	421.8
18	42	104	31	8666	83.3	28	12298	118.3
19	48	337	49	3469	10.3	49	534982	1587.5
20	85	499	84	2528	5.1	84	2993	6.0
Average	22	126	21.2	2489	19.8	20.9	75124	596.2

Table V.5: Comparison of the NV40 code optimisers.

Figure V.4 depicts that the runtime of the code optimisation pattern matcher optimising NV40 code behaves similar to the pattern matcher's runtime when processing NV30 code. Although the preceding code generation phase produces less NV40 than NV30 instructions, the pattern matcher takes longer to optimise NV40 code, because the pattern matcher additionally checks whether the condition optimisation can be applied (see Section 2.2).

Independent from the architecture, the pattern matcher is most of the time able to apply the constant and copy propagation to optimise a basic block. Occasionally, the code optimisation pattern matcher makes use of the copy and dead code elimination and rarely applies constant folding and constant vectorisation. When optimising NV40 code, the pattern matcher is often able to make use of the condition optimisation, if the code contains branches.

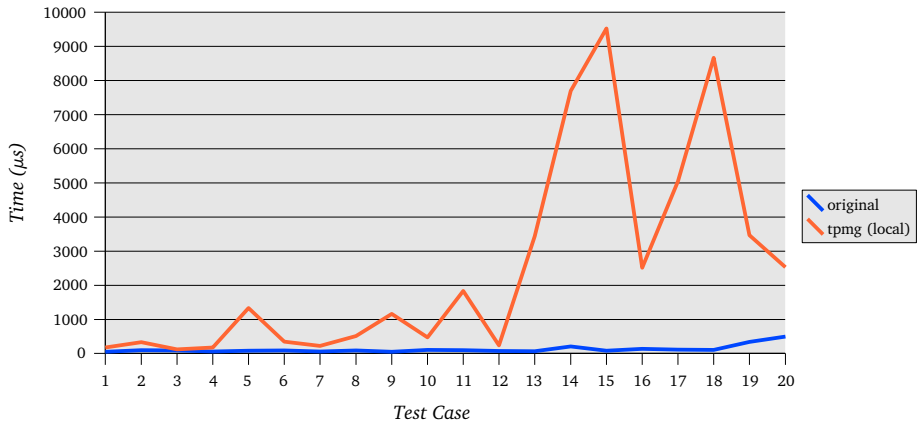

Figure V.4: Time to optimise NV40 code (original and tpmg optimiser).

On average, the pattern matcher optimises an NV40 instruction in 99μs and is about 19 times slower than the original code optimiser. However, as the optimisation only accounts for 5.0% of the total processing time of the CGiS compiler, this runtime increase is totally acceptable.

Although the code optimisation pattern matcher is on average slightly more effective than the original code optimiser, the pattern matcher is not as effective as it could be. The main reason is that the pattern matcher processes the basic blocks separately not knowing about their interconnections. So, the pattern matcher is not able to perform certain optimisations as the pattern matcher lacks the necessary information that a liveness or value analysis would provide. If the pattern matcher would possess the required data, its effectiveness could be raised to about 20% for NV30 code and about 25% for NV40 code.

As the following section demonstrates, the pattern-matcher generated and optimised code is nevertheless on average faster than the code, which the original code generation and optimisation procedure of the CGiS compiler produces.

2.3.3. Runtime

This section investigates the runtime of each test case to determine how efficient the generated code actually is. Because the original code optimiser and the code optimisation pattern matcher produce different results, there should be a measurable difference in their runtime.

Table V.6 displays how the execution times of the 20 test cases compare to each other with respect to the used optimiser and target architecture. All test cases have been compiled with gcc 3.3.6 (without any optimisation flag) and executed under Linux (Ubuntu 6.06) using the NVIDIA graphics card driver 1.0-8756.

Test	*NV30*			*NV40*		
	Time (orig., µs)	*Time (tpmg, µs)*	*% Diff.*	*Time (orig., µs)*	*Time (tpmg, µs)*	*% Diff.*
1	1965	1988	101.2	1921	1936	100.8
2	31703	24993	78.8	28534	24716	86.6
3	2594	2618	100.9	2590	2603	100.5
4	1835	2018	101.0	1957	2132	108.9
5	8398	10364	123.4	8331	10507	126.1
6	3766	3797	100.8	4025	3991	99.2
7	1892	1976	104.4	1893	1985	104.9
8	2462	2507	101.8	2542	2568	101.0
9	1654	1563	94.5	1676	1550	92.5
10	2897	2863	98.8	2881	2860	99.3
11	3299	3171	96.1	3312	3205	96.8
12	2153	2238	103.9	2148	2180	101.5
13	2011	1937	96.3	2103	2050	97.5
14	3343	3408	101.9	3733	3738	100.1
15	8739	8819	100.9	15975	15904	99.6
16	4477	4495	100.4	4157	4143	99.7
17	2556	2370	92.7	3039	2652	87.3
18	11858	11701	98.7	11985	11833	98.7
19	26441	26715	101.0	11978	12462	104.0
20	37204	38947	104.7	36355	34579	95.1
Average	8062	7924	98.3	7557	7380	97.7

Table V.6: Comparison of the runtime of the optimised NV30 and NV40 code.

On average, the pattern-matcher optimised NV30 code is about 1.7% faster, whereas the pattern-matcher optimised NV40 code is about 2.3% faster. The difference is clearly noticeable at the test cases that the code optimisation pattern matcher optimises best (e.g., 9 to 11, 13, 17 and 19).

However, some of the above results are quite confusing, especially the runtime of the test cases 2 and 5. Although the original code optimiser generates only 16 instructions, where the code

optimisation pattern matcher produces 18 instructions when optimising test case 2 for both the NV30 and NV40 architecture, the pattern-matcher optimised code is faster. This effect is too strong to be an error in measurement and must thus originate from the optimisations of the graphics card's driver. The same applies to test case 5, where the original code optimiser produces 21 instructions and the pattern matcher produces 22 instructions. Although the difference is only one instruction, the pattern-matcher optimised code is about 25% slower. It is very unlikely that a single operation causes this divergence. Note that, in both cases, the optimised code only differs in the number of *MOV* instructions.

Figure V.5 displays how the runtimes of the test cases compare to each other. A closer look reveals that the generated NV40 code of about half of the test cases is slower than the corresponding NV30 code. The runtime of the NV30 and NV40 code of test case 15 shows the most significant difference, although the code optimisers have produced the same instructions for both NV30 and the NV40 architecture. Again, this divergence seems originate from the driver of the graphics card (or this might even be a peculiarity of the used graphics card). In contrast to test case 15, the generated NV40 code of test case 19 runs significantly faster than the corresponding NV30 code. The reason for this difference appears to be the number of generated instructions (around 70 for NV30 and about 48 for NV40) and not to be a peculiarity of the graphics card's driver.

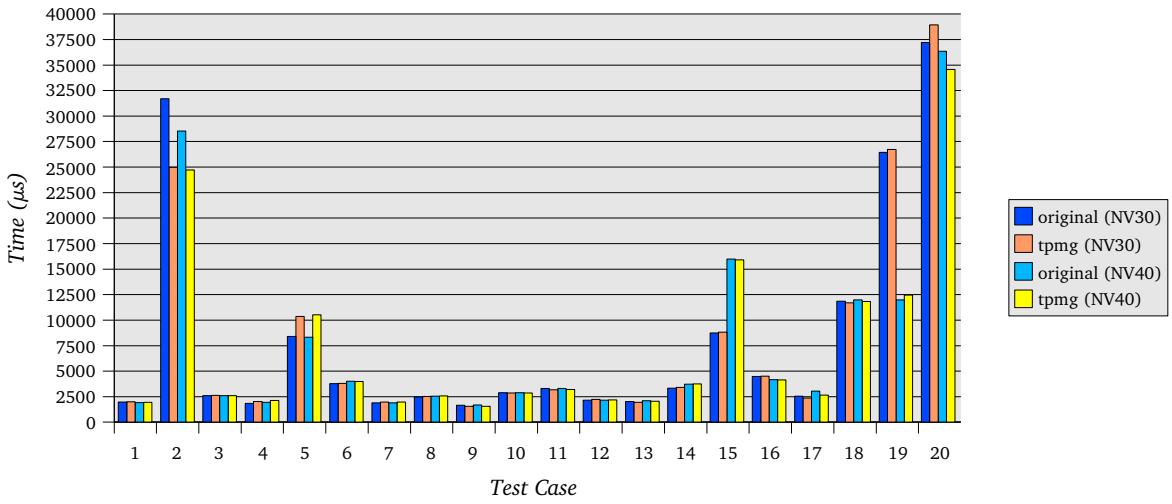

Figure V.5: Execution time of the optimised test cases.

So, although the code optimisation pattern matcher has not yet reached its full potential, the optimisations still have a measurable effect – even if little – despite the heavy optimisations the graphics card's driver additionally performs.

VI. Related Work

This chapter briefly introduces two systems that are related to the present work and discusses the main differences between these applications and the pattern matcher generator tpmg.

1. BURG

BURG is a program that compiles a tree grammar into a BURS[38] tree parser [44]. Similarly to the pattern matcher description language, the BURG tree grammar corresponds syntactically to the bison grammar. Comprising a header section, which configures the tree parser, and a rule section, which contains a sequence of cost-annotated rules, the BURG grammar furthermore bears resemblance to the tpmg grammar (see Section 2 in Chapter IV). However, at this point the similarities between the two applications stop.

In contrast to tpmg-generated pattern matchers, tree parsers operate – as their name already suggest – on trees. So, the application in which to embed a BURG-generated tree parser, has to represent the data in tree-like structures. To enable the tree parser to navigate within the internal tree structure, the user must additionally specify how to access the left and right child of each tree node. BURG-generated tree parsers differ furthermore from tpmg-generated pattern matchers, because the rules are not separated into different profiles. Instead, the user specifies a start rule with which the tree parser begins to match an input tree. The consequence is that a BURG-generated tree parser is dedicated to a single hardware architecture. Thus, the user has to implement a new tree parser for each architecture to support, whereas the application must select the appropriate tree parser manually. Additionally, BURG does not support wildcard patterns. The user may only specify patterns that describe either leafs or nodes with one or two child nodes. So, BURG tree parsers cannot skip past a specific portion of the input tree without processing it. In contrast to tpmg-generated pattern matchers, tree parsers are not able to match instructions that belong to different expression trees. Thus, a tree parser might not be able to optimise an *ADD* and a *MUL* instruction to a *MAD* instruction (see Example .6 in Chapter III), if the corresponding instructions belong to two different expression trees[39].

The main field of application of BURG-generated tree parsers is the code generation. Being designed to realise fast instruction selectors, BURG tree parsers discovers an optimal parse of an input tree in linear time. BURG is used by the ANSI C compiler lcc [45] that generates code for multiple target architectures, such as ALPHA, SPARC, MIPS R3000 and Intel x86.

2. Recognizer

Implementing a code optimiser is a complicated task, because it requires – amongst other things – detailed knowledge about the target architecture. So, whenever a compiler should support a new platform, the developer must invest a lot of time to realise good optimisations. To relieve the developer from this burden, João Dias and Norman Ramsey propose a *recognizer* [46], which realises machine-independent code selection and optimisation. A recognizer is generated automatically from a declarative machine description that clearly describes – independent from any compiler – properties of a target platform.

The generated recognizer requires the compiler to represent intermediate code as machine-independent register-transfer lists (RTLs) [47]. An RTL is some kind of intermediate code repres-

38 BURS abbreviates **B**ottom-**U**p **R**ewrite **S**ystem.

39 A tpmg-generated pattern matcher is not subject to this restriction, because the pattern matcher processes basic blocks. If the two instructions reside in the same basic block, the pattern matcher is able to optimise them.

entation that describes how data between the registers of an architecture is being transferred. Many compilers, such as the GNU C compiler gcc, make use of RTLs.

By means of a declarative machine description, the recognizer tries to generate more optimal RTLs. The recognizer will continue until no more optimisations can be applied. The recognizer omits a previously generated RTL, if the new RTL cannot be implemented on the target platform according to the machine description.

So this approach differs greatly from tpmg-generated pattern matchers, because the user does not have to explicitly implement the code optimiser, as the generated recognizer handles this part. Furthermore, the user does not have to care about the peculiarities of the target platform, because the used declarative machine description automatically covers all of them.

Dias and Ramsey have successfully generated and tested a recognizer for the x86 architecture in the Quick C-- compiler [48]. Unfortunately, the results were modest, because the rest of the compiler's x86 back end was still hand-written. So, to achieve better results, Dias and Ramsey ultimately plan to generate the whole back end.

VII. Future Work

Chapter V demonstrates that tpmg-generated pattern matchers can be used to replace a vital part of a compiler's back end. Although being able to improve the code generation and code optimisation, tpmg-generated pattern matchers can still be advanced to produce even better results.

In the future, I plan to implement the following features:

- **Java back end**

  A Java back end would definitely improve the versatility of tpmg. Apart from minor adjustments, the current C++ implementation could be easily ported to Java, because the current Java version supports both exceptions and – more important – templates.

- **Data structure interface**

  A data structure interface that enables the user to use custom data types with tpmg-generated pattern matchers would furthermore improve the versatility of tpmg. The main advantages of this improvement are that, on the one hand, the user is no longer forced to use the STL, and on the other hand, the user is able to specify which data structure to use, so that the runtime of a generated pattern matcher is not bound to implementation of the STL.

- **Keep matched items**

  The current implementation removes every object instance that has been matched by an item pattern. However, under certain circumstances, the user might want to keep a matched object instance. This feature would enable a rule to peek downwards in the basic block. A more sophisticated dead code elimination rule could then be specified as follows:

```
rule dead_code_elimination
{
  search:  [ Operation,
             *,
             keep: Operation ($$->target() == $1->target()) ]
  cost:    { return 1; }
  replace: []
}
```

  This rule detects two, not necessarily adjacent operations, where the second operation overrides the result of the first operation. If the rule matches, only the first operation may be removed from the basic block, because the second item pattern is marked specially (*keep* flag).

  If an object instance matched by an item pattern should be kept in the basic block, the implicit assumption that all object instances are pushed upwards past every wildcard pattern obviously no longer holds. Thus, the user must additionally be able to specify rule-specific implicit conditions (see below).

- **Specify an alternative insertion point**

  When optimising a basic block, every rule inserts the replace pattern before the first matched object instance. This behaviour results from the implicit assumption that all object instances are to be pushed upwards past every wildcard pattern in the basic block. However, to enable a rule to push object instances downwards, the user should be able to specify an alternative insertion point. So, a rule that pushes an instruction downwards past a wildcard pattern could be specified as follows:

```
rule push_downwards
{
  search:  [ Operation,
             *,
             after: Operation ]
  cost:    { return 1; }
  replace: [ $1,
             $2 ]
}
```

  If the rule matches, it will virtually push the first matched operation past the wildcard pattern in front of the second operation. So, the implicit condition must consider that the rule pushes the matched instruction downwards and not upwards. Thus, the user must be able to specify a rule-specific implicit condition (see next point).

- **Rule-specific implicit conditions**

  If the user wants certain rules to behave differently than others e.g., by specifying an alternative insertion point, a single global implicit condition function does not satisfy. Thus, the user must be able to override the global implicit condition function for these rules to ensure the correct behaviour of the pattern matcher.

- **Matching control-flow graphs**

  The ultimate goal is to generate pattern matchers that are able to process the whole control-flow graph instead just a single basic block. Coping with branches and loops automatically, a control-flow graph pattern matcher would then be able to achieve far better results than the current tpmg-generated pattern matchers.

  To keep the rule specifications as simple as possible, it appears to be feasible to hide the control-graph structure completely, so that the user does not have to cope with branches or loops within rules. Merely the implicit conditions would increase in complexity.

VIII. Conclusion

The present work discusses theoretical and practical aspects of the pattern matcher generator tpmg and demonstrates, by means of the CGiS compiler cgisc, the usefulness of the generated pattern matchers that can be used to realise retargetable code generators and optimisers. The following achievements have been accomplished:

- **Retargetability**

  Most important of all accomplishments, the profile and rule inheritance system, which corresponds to the class inheritance mechanism of object-oriented programming languages, enables a developer to quickly adjust a pattern matcher to a new target architecture. Besides specifying which profile to use, no more manual interaction with the generated pattern matcher is required.

- **Versatility**

  The integration of tpmg-generated pattern matchers into a compiler does not pose a problem at all, because the pattern matchers make few demands to the implementation of the compiler. Note that the field of application is not restricted to compilers, because tpmg-generated pattern matchers do not make any assumptions about the objects to process. So, the pattern matchers can by employed in any application that operates on lists of objects.

- **Extensibility**

  The tpmg template library has been designed such that it can be easily extended. So, most of the planned improvements presented in Chapter VII are easy to add. In fact, I have already implemented the *keep* flag that allows the developer to specify whether the matched object of an item pattern should not be removed from the basic block.

- **Cooperation with program analyses**

  To further pitch the tpmg-generated pattern matchers to a developer and ease the integration into a compiler, the pattern matchers are designed to work hand in hand with program analyses. Making use of implicit functions (condition and postpass), the developer can easily interact with any program analysis to influence the behaviour of a pattern matcher.

- **Increased productivity**

  Because the generated pattern matchers identify and replace instruction patterns automatically, the developer no longer has to cope with low-level algorithms. Thus, I expect that the developer is able to concentrate earlier on more important aspects of the compiler, which finally increases the developer's overall productivity.

- **Ease of use**

  Adopting language features of the bison, C^{++} and Java grammar, the pattern matcher description language provides a familiar development environment, in which the developer can start implementing the desired pattern matchers right away. Because both syntax and semantics are more or less self-explanatory, the developer will accustom with that new language in no time.

  Nicolas Fritz successfully made use of the pattern matcher description language to implement the SSE profile that handles the SSE code generation of the CGiS compiler.

- **Integration into an existing compiler**

  To confirm the usefulness of tpmg-generated pattern matchers, a code generation and a code optimisation pattern matcher have been introduced in the CGiS compiler cgisc. The test results show that, accompanied by an acceptable increase in runtime, the pattern matchers are able to produce better code than the original code generation and code optimisation procedure.

A. Pattern Matcher Description Language Grammar

Using an extended Backus-Naur form (see Section 2 of Chapter IV), this appendix chapter gives an overview of the grammar of the pattern matcher description language. Please note that the grammar does not include the preprocessor statements, because they are not visible to the parser.

```
RULEFILE       ::= ([HEADER] RULESET)+
HEADER         ::= '%{' ASCII* '%}'
RULESET        ::= RULESET_HEADER RULESET_BODY
RULESET_HEADER ::= ruleset '<' CLASSNAME ',' CLASSNAME '>'
RULESET_BODY   ::= '{' [IMPLICIT] (RULE | PROFILE)* '}'
PROFILE        ::= PROFILE_HEADER PROFILE_BODY
PROFILE_HEADER ::= profile PROFILENAME [PROFILE_EXTEND]
PROFILE_EXTEND ::= ':' extends PROFILENAME [',' PROFILENAME]*
PROFILE_BODY   ::= ';' | '{' (RULE | RULE_OMIT)+ '}'
RULE_OMIT      ::= omit RULE_ID [',' RULE_ID]* ';'
RULE_ID        ::= [[PROFILENAME] '::'] RULENAME
RULE           ::= RULE_HEADER RULE_BODY
RULE_HEADER    ::= rule RULENAME [RULE_EXTEND] [RULE_MASK]
RULE_EXTEND    ::= ':' RULE_ID
RULE_MASK      ::= ':' '(' INTEGER ')'
RULE_BODY      ::= '{' [SEARCH] [CONDITION] [COST] [REPLACE] '}'
SEARCH         ::= SEQUENCE
SEQUENCE       ::= '[' SPATTERNS ']' | '{' SPATTERNS '}'
SPATTERNS      ::= [SPATTERN [',' SPATTERN]*]
SPATTERN       ::= SEQUENCE | ITEM | WILDCARD
ITEM           ::= ('.' | CLASSNAME) ['(' EXPRESSION ')']
EXPRESSION     ::= ASCII*
WILDCARD       ::= '*'
CONDITION      ::= '{' ASCII* '}'
COST           ::= '{' ASCII* '}'
REPLACE        ::= '[' [RPATTERNS] ']'
RPATTENRS      ::= RPATTERN [',' RPATTERN]*
RPATTERN       ::= RGUARD | RITEM
RGUARD         ::= if '(' EXPRESSION ')' REPS [else (RGUARD | REPS)]
RITEM          ::= '$' INTEGER [INITIALISERS] | ITEM [INITIALISERS]
INITIALISERS   ::= (':' INITIALISER)+
INITIALISER    ::= IDENTIFIER '(' EXPRESSION ')'
```

B. Pattern Matcher Examples

In the first section, this appendix chapter demonstrates how to invoke tpmg. In the remaining sections, this chapter present some other applications tpmg-generated pattern matchers can be used for.

1. Invoking tpmg

The following command invokes tpmg:

```
tpmg [parameters] file...
```

The user must provide at least one file, otherwise the program stops immediately. Processing each specified file separately, tpmg creates a C++ header and code file depending on the input file's base name. If e.g., the name of an input file is `test.tpmg`, tpmg will create `test.h` and `test.cpp` – provided that the input file is valid.

The following command line parameters influence the behaviour of tpmg:

- `-D` *`VARIABLE`*

  This parameter defines a variable just as the tpmg preprocessor statement `#define`. It remains defined for each input file unless it is undefined.

- `-I` *`PATH`*

  Adds *`PATH`* to the list of include paths. The pattern matcher generator first searches in the specified include paths for files that are specified in include statements of the form:

  ```
  #include <FILE>
  ```

  If the pattern matcher generator cannot find the specified file, tpmg tries to look for the file in the current working directory.

- `-W`

  Enables the warnings, so that tpmg informs the user whenever it detects a potential runtime error (see Example .5 in Chapter IV). By default, warnings are disabled.

- `-P, --pedantic`

  This command parameter causes tpmg to stop processing, whenever tpmg encounters a warning.

- `-V, --verbose`

  After tpmg has processed an input file, tpmg display some information about the generated pattern matcher.

- `--funsafe`

  When specified, tpmg does not generated code that checks whether a rule accesses an unavailable pattern (see Example .4 in Chapter IV). This command line parameter may only be used, if the user manually checks for unsafe accesses (unavailable item patterns are NULL pointers).

2. List Sorting

This example shows how to realise a very simple list sorting algorithm, also known bubble sort, with a tpmg-generated pattern matcher. The pattern matcher sorts – optimises – an arbitrary list in either ascending or descending order with respect to the value of each item. The common base class of the list items could be implemented as follows:

```
class Item
{
  public:
    Item (int value)
      : m_value (value)
    {
    }

  public:
    inline int value (void) const { return m_value; }

  private:
    int m_value;
};
```

The following pattern matcher comprises two profiles, one to sort a list in ascending order and the other to sort a list in descending order. Both profiles contain a rule named *sort* that flips two adjacent items in the list depending on their value. The rule in the profile *Ascending* checks if the value of the first item is smaller than the value of the second item and flips both items to push the cheap item to left and the expensive item to the right. Note that it is not necessary to respecify the search pattern, the cost function or the replace pattern, if another sorting behaviour is desired.

```
ruleset<Item, Item>
{
  profile Ascending
  {
    rule sort
    {
      search:    [ ., . ]
      condition: { return $1->value() > $2->value(); }
      cost:      { return 1; }
      replace:   [ $2, $1 ]
    }
  }

  profile Descending
  {
    rule sort : extends Ascending::sort
    {
      condition: { return $1->value() < $2->value(); }
    }
  }
}
```

Independent from the used sort profile, the pattern matcher sorts the list in the desired order after a finite number of steps. As expected, the sorting method is quite inefficient and has a

worst runtime of $O(n^2)$, where n is the length of the list. Obviously, the sorting takes longest, if the items in input list are aligned in the opposite sorting order.

3. Calculator

This example demonstrates how to realise a Polish and Reverse Polish notation calculator with a tpmg-generated pattern matcher. The Polish notation is a special kind of notation for logic, arithmetic and algebra. Under the assumption that the arity of each operator is given, this notation is able to work without any kind of parenthesis. The Polish notation is also known as *prefix notation*, because it places the operators in front of their arguments. In contrast to the Polish notation, the Reverse Polish notation, also known as *postfix notation*, places the operators after their arguments.

Example 3.1

Given the expression e = (2+((2*4.5)/0.5))/(3-1.5). The expressions e_{PN} and e_{RPN} are equivalent expressions in Polish and Reverse Polish notation respectively:

e_{PN} = / + 2 / x 2 4.5 0.5 - 3 1.5

e_{RPN} = 2 2 4.5 x 0.5 / + 3 1.5 - /

Due to the simple structure of Polish notation expressions, a pattern matcher that evaluates these expressions can be easily realised. The pattern matcher "optimises" a list of instances of the `Object` class, from which the classes `Operator` and `Number` derive. Each number has a unique value that can be accessed with the `value` function. An operator implements the `eval` function that computes the result of the operation. To simplify this example, it is assumed that all operators are binary. So, the pattern matcher is specified as follows:

```
ruleset<Object, Object>
{
  profile Polish
  {
    rule Step
    {
      search:  [ Operator,
                 Number,
                 Number ]
      cost:    { return 1; }
      replace: [ Number ($1->eval($2->value(), $3->value())) ]
    }
  }

  profile ReversePolish
  {
    rule Step
    {
      search:  [ Number,
                 Number,
                 Operator ]
      cost:    { return 1; }
      replace: [ Number ($3->eval($1->value(), $2->value())) ]
    }
  }
}
```

Depending on the given profile, the generated pattern matcher evaluates the given expression by iteratively applying the rule `step` as long as possible. To detect an invalid expression, the user simply has to check whether the final expression (i.e., list of objects) only contains one instance of the class `Number`. The number of necessary steps increases linearly with the number of operators, so the overall runtime is $O(n)$, where n is the number of operators.

List of Figures

List of Tables

Bibliography

[1] L. Fernandes, The Abacus:
http://www.ee.ryerson.ca/~elf/abacus/

[2] Universität Tübingen, Short biography of Wilhelm Schickard (German):
http://www-ti.informatik.uni-tuebingen.de/deutsch/schickard/

[3] B. Randell, "From Analytical Engine to Electronic Digital Computer: The Contributions of Ludgate, Torres and Bush", *IEEE Annals of the History of Computing*, 4(4): 327-341, October 1982.

[4] R. Rojas, "How to make Konrad Zuse's Z3 a universal computer", *IEEE Annals of the History of Computing*, 20(3): 51-54, July 1998.

[5] R. Rojas, "Konrad Zuse's legacy: The Architecture of the Z1 and Z3", *IEEE Annals of the History of Computing*, 19(2): 5-16, 1997.

[6] M. R. Williams, "A History of Computing Technology", *IEEE Computer Society Press*, 1997. ISBN 0-8186-7739-2.

[7] G. Moore, "Cramming more components onto integrated circuits", *Eletronics*, 38(8): 114-117, April 1965.

[8] Institute for New Generation Computer Technology, "Fifth Generation Computer Systems '92: Proceedings of the International Conference on Fifth Generation Computer Systems", *IOS Press*, 1992. ISBN 90-5199-099-5.

[9] K. Zuse, "Der Plankalkül", *Gesellschaft für Mathematik und Datenverarbeitung*, Nr. 63, BMBW - GMD - 63, 1972.

[10] R. W. Sebesta, "Concepts of Programming Languages, 4th edition", *Addison-Wesley Longman Publishing Co., Inc.*, 1998. ISBN 02-0138-596-1.

[11] R. Rojas, C. Götekin, G. Friedland, M. Krüger, O. Langmack, and D. Kuniß, "Plankalkül: The First High-Level Programming Language and its Implementation", Technical Report B-3/2000, Freie Universität Berlin, Institut für Informatik, February 2000.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, pages 220-242, Jyväskylä, Finland, June 1997.

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware", *EUROGRAPHICS, State of The Art Report*, pages 21-51, Dublin, Ireland, 2005.

[14] T. H. Meyer and I.E. Sutherland, "On the Design of Display Processors", *Communications of the ACM*, 11(6): 410-414, June 1968.

[15] D. Ingalls, Bit BLT Inter-Office Memorandum, November 1975, Xerox PARC:
http://www.bitsavers.org/pdf/xerox/alto/BitBLT_Nov1975.pdf

[16] M. Breterniz Jr., H. Hum, and S. Kumar, "Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU", *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT 2003)*, pages 135-145, New Orleans, LA, USA, September 2003.

[17] OpenGL ARB, OpenGL Extension Registry:
http://www.opengl.org/registry/

[18] OpenGL ARB, NVIDIA register combiners OpenGL extension:
http://www.opengl.org/registry/specs/NV/register_combiners.txt

[19] M. Segal and K. Akeley, OpenGL 2.0 Specification:
http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf

[20] General-Purpose Computation using Graphics Hardware: http://www.gpgpu.org

[21] J. Hoxley, An Overview of Microsoft's Direct3D 10 API: http://www.gamedev.net/reference/programming/features/d3d10overview/page2.asp

[22] FIPS 46-3 Data Encryption Standard (DES), October 1999: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[23] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan, "A real-time procedural shading system for programmable graphics hardware", *Proceedings of the 28th annual conference on Computer graphics and interactive techniques (SIGGRAPH 2001)*, pages 159-170, Los Angeles, CA, USA, August 2001.

[24] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: simplified programming of graphics processing units for general-purpose uses via data-parallelism", Technical Report MSR-TR-2005-184, Microsoft Research, December 2005.

[25] R. J. Cook, "Shade Trees", *ACM SIGGRAPH Computer Graphics*, 18(3): 223-231, July 1984.

[26] H. Gouraud, "Computer Display of Curved Surfaces", *IEEE Transactions on Computers*, 22(6): 623-629, June 1971.

[27] B. T. Phong, "Illumination for Computer Generated Pictures", *Communications of the ACM*, 18(6): 311-317, June 1975.

[28] Pixar, The RenderMan Interface Specification: http://renderman.pixar.com/products/rispec/index.htm

[29] OpenGL Architecture Review Board: http://www.opengl.org/about/arb/

[30] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A System for Programming Graphics Hardware in a C-like Language", *ACM Transactions on Graphics (TOG)*, 22(3): 896-907, July 2003.

[31] M. D. McCool, Z. Qin, and Tiberiu S. Popa, "Shader Metaprogramming", *SIGGRAPH/EUROGRAPHICS Graphics Hardware Workshop*, pages 57-68, Saarbrücken, Germany, September 2002. Revised version.

[32] Merrimac - Stanford Streaming Supercomputer Project: http://merrimac.stanford.edu/

[33] I. Buck, T. Foley, D. Horn, J. Sugarman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware", *ACM Transactions on Graphics (TOG)*, 23(3): 777-786, August 2004.

[34] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon Mapping on Programmable Graphics Hardware", *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41-50, San Diego, CA, USA, July 2003.

[35] N. Fritz, P. Lucas, and P. Slusallek, "CGiS, a new Language for Data-Parallel GPU Programming", *Proceedings of the 9th International Workshop Vision, Modeling, and Visualization (VMV 2004)*, pages 241-248, Stanford, CA, USA, 2004.

[36] R. Wilhelm and D. Maurer, "Übersetzerbau - Theorie, Konstruktion, Generierung", *Springer-Verlag*, 1992. ISBN 3-540-55704-0 (German).

[37] M. Alt, F. Martin and R. Wilhelm, "Generating Analyzers with PAG", Technical Report A10/95, Universität des Saarlandes, FB 14 Informatik, 1995.

[38] Flex, a fast lexical analyser generator: http://www.gnu.org/software/flex/

[39] Bison, a general purpose parser generator: http://www.gnu.org/software/bison/

[40] SGI, The standard template library:
http://www.sgi.com/tech/stl/stl_introduction.html

[41] D. v. Heesch, Doxygen:
http://www.stack.nl/~dimitri/doxygen/

[42] Autoconf, an automated configuration script generator:
http://www.gnu.org/software/autoconf/

[43] Automake, an automated makefile generator:
http://www.gnu.org/software/automake/

[44] C. W. Fraser, R. R. Henry, and T. A. Proebsting, "BURG - Fast Optimal Instruction Selection and Tree Parsing", *SIGPLAN Notices*, 27(4): 68-76, April 1992.

[45] C. W. Fraser and D. Hanson, "A Retargetable C Compiler: Design and Implementation", *Addison-Wesley*, 1995. ISBN 0-8053-1670-1.

[46] J. Dias and N. Ramsey, "Converting Intermediate Code to Assembly Code Using Declarative Machine Descriptions", *Proceedings of the 15th International Conference on Compiler Construction (CC 2006)*, pages 217-231, Vienna, Austria, March 2006.

[47] J. W. Davidson and C. W. Fraser, "Register Allocation and Exhaustive Peephole Optimization", *Software - Practice and Experience*, 14(9): 857-865, September 1984.

[48] Quick C-- Compiler:
http://www.cminusminus.org/qc--.html

Index