Pattern Matching Strategies for Peephole Optimisation

Elif Aktolga
ea29@sussex.ac.uk

August 26, 2005

Supervisor: Dr. Des Watson

MRes Dissertation in
Computer Science and Artificial Intelligence
School of Science and Technology
University of Sussex

Abstract

Peephole optimisation is a simple and effective optimisation technique used in conventional compilers. In classical peephole optimisers, optimisation rules are commonly applied through string pattern matching by means of regular expressions. This string-based approach to matching has proven to be very effective, but it is just the syntax of the assembly code input that is processed and its meaning gets lost. This dissertation explores alternatives to this classical approach: pattern matching for peephole optimisation is viewed as a procedure that is composed of a rule application strategy and a specific pattern matching strategy for matching within a rule. Two 'generic' matching strategies are developed in an OOP context to perform qualitatively better and more intelligent matching by utilising the meaning of the code. The results show that generic matching can yield better and even faster results than matching with mere regular expressions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Peephole optimisation is a simple and effective optimisation technique used in conventional compilers. Commonly, optimisation rules are applied through string pattern matching by means of regular expressions. How can this task be performed in a more appropriate and meaningful manner? How would such pattern matching strategies have to be designed? These are the leading research questions of this work.

1.1 Motivation and Problem Statement

Compilers take a program as input, and typically produce semantically equivalent target machine instructions as output [18]. The programmer's responsibility lies in writing good source code, whereas the compiler's task is to provide the translation [26]. An 'optimal translation' would ideally require minimal CPU time and memory. In practice, this is usually not possible [2, 26]. But optimising compilers can approach this by improving generated code with various techniques. Therefore, code optimisation aims at producing more efficient, faster, and shorter code without changing its effects [26].

The problem with naïve code generation is that resulting code often contains redundancies when juxtaposed [9]. These can easily be reduced with an effective optimisation technique – *peephole optimisation*. It is typically applied to assembly code or to an equivalent low-level representation after code generation to make local improvements [2]: Small groups of instructions are analysed through a peephole and replaced where possible with shorter or faster instructions. The peephole optimiser is repeatedly passed over the modified assembly code, performing optimisations until no further changes are possible. Thus, many optimisations are achieved such as e.g. removal of multiple jumps, unreachable code, and various others [2, 34].

These optimisations are machine-dependent; i.e. the peephole optimiser must know about the target machine's instruction set in order to detect op-

timisations. This knowledge base is usually expressed with rules or so-called *patterns* [18]. Patterns contain variables that are instantiated with concrete values during application, which is known as *pattern matching*. Peephole optimisers traditionally use pattern matching in order to decide whether a rule is applicable, and to form the result of a successful application. In the 1980's, excellent techniques were developed for peephole optimisers to perform machine-dependent optimisations in a machine-independent way (see Chapter 2). The approach to pattern matching though has mostly been through regular expressions only. Although this technique is efficient and simple, its suitability is questionable.

With the emergence of new paradigms, such as Object-Oriented Programming (OOP), approaches to programming and the design of code have changed. In this regard different strategies such as 'generic pattern matching' [32] have been developed in the field of pattern matching. This work aims at exploring and applying such pattern matching strategies that suit the nature of peephole optimisation.

The implementation of this work in Java provides a simplistic peephole optimiser framework targeting the Intel x86 machine. The peephole optimiser can be used with three different strategies that were designed for it. Nevertheless, it shall be noted that performance is not the primary aim of this work, but rather researching the applicability of new design strategies to a classical, effective technique.

1.2 Overview

The structure of this dissertation is as follows: Chapter 2 contains a summary of relevant background work in the areas of peephole optimisation and pattern matching. In Chapter 3, the strategies designed for the system are formally described. Chapter 4 then focuses on implementation-specific aspects, such as the structure of the optimiser, language issues, and the parsers. Optimisation examples aid in comparing the strategies in Chapter 5, revealing the strengths and flaws of each. In Chapter 6, the success of the project is evaluated, and directions for future research are summarised.

Chapter 2

Prior Research

This chapter mainly outlines previous research in the field of peephole optimisation. Only relevant issues to the work described here are elaborated; therefore this is not meant to be a complete reference to prior peephole optimisation research: Section 2.1 describes the first peephole optimisers, whereas section 2.2 focuses on the retargetable approach of the 1980's. In section 2.3, techniques that attempt to integrate code generation and optimisation are discussed. The last section 2.4 refers to recent research concerning pattern matching strategies.

2.1 Classical Peephole Optimisers

McKeeman [25] introduced the notion of peephole optimisation in 1965. It was noted that code emitted from code generators contained many redundancies, particularly around borders of basic blocks (e.g. chains of jump instructions). Reducing these redundancies during the code generation phase would have resulted in complicated case analysis. So it was appropriate to introduce a separate phase that would deal with them. The idea was the following: the peephole, a small window comprising no more than two instructions of assembly code, is passed over the code. Whenever a sequence of redundant instructions is encountered, they are replaced by cheaper or shorter ones. The knowledge the peephole optimiser uses for this are simple hand-written pattern rules. These are first matched with assembly instructions for testing applicability. In case of a successful match, the instructions are replaced. Therefore, a typical pattern rule consists of a match part and a replacement part [23] (see below). The pattern set is usually small as this is sufficient for fast and efficient optimisation. Consequently, classical peephole optimisers can be characterised as being fast, rule-directed, machine-dependent, and hand-written.

Lamb [23] describes the implementation of a pattern-driven peephole optimiser. The optimiser consists of a collection of pattern rules, a translator that serves as a parser, and a pattern-matcher skeleton, which forms the

main part of the optimiser. The code generator produces assembly code in the form of a doubly-linked list of nodes so that the peephole optimiser can easily operate on it. The optimiser uses a compiled version of the patterns to analyse the code, which is then simplified where possible. As mentioned before, a pattern rule or an *optimisation rule* consists of a matching or a *preconditions part* (as in [36]) and a replacement part. The pattern is a generalisation over specific cases; therefore, it contains variables or abstract symbols that are instantiated during pattern matching. An optimisation rule takes the following form:

```
<input pattern line 1 [| condition(var)]>
          ...
<input pattern line n>
=
<replacement pattern line 1>
          ...
<replacement pattern line n>
```

`| condition(var)` is optional (therefore expressed with square brackets); it can only occur in the matching portion of the pattern. Lamb refers to this part as an *escape*, which is a condition over some variable(s) occurring in the pattern that must be met. A pattern matches successfully if and only if all the instructions in the preconditions part can be applied to adjacent assembly code instructions, including any specific conditions over variables, which must evaluate to true. Escapes can also be embedded in the replacement part of the pattern, where they rather represent an additional *action* to be taken than describing a condition over the variables as in the matching part. For instance, negating a variable might be such an action. It is difficult to represent this as assembly code, whereas calling a subroutine that does the job is preferable. Lamb performs matching *backwards*, not forwards as it was originally approached. In fact, this strategy is adopted in most peephole optimisers. Here 'backwards' refers to the testing part of the matching procedure, not to the order the instructions are dealt with (which is forwards always): the optimiser starts testing the last pattern instruction in the preconditions portion of the rule, finishing with the first one. This has the advantage that further optimisations can instantly be performed on newly inserted instructions and their predecessors. It is also very useful for interacting rules, since for those, new opportunities for optimisation arise after a replacement only. Thus, it is reasonable to utilise a general strategy that analyses the optimisability of the current instruction against its predecessors. In the other approach originally adopted by McKeeman [25], one would have to pass through the code several times until no more improvements can be made. The backwards strategy is surely faster and more efficient, since the code has to be scanned once only.

The work described in this dissertation adopts this idea for the regular expressions and the generic strategies (for details see sections 3.2.1 and 3.3.1), which is close to classical peephole optimiser implementation. Escapes are also handled similarly though they have not been explicitly implemented.

Fraser's peephole optimiser copt [12] also uses the above-mentioned style of optimisation rules and the backwards strategy — though being a retargetable optimiser (see section 2.2).

2.2 Retargetable Peephole Optimisers

As mentioned in the previous section, early peephole optimisers acted on assembly code directly and were machine-dependent. So they could only target a single machine. For use with other machines, they had to be completely rewritten. Retargetable optimisers distinguish themselves from classical ones in their flexibility, being machine-independent and therefore easily retargeted. They are usually slower in performance though than their classical counterparts.

Davidson and Fraser developed the first retargetable peephole optimiser, PO [9, 13]. This optimiser is brought into action before code generation, which allows a machine-independent analysis of the code. PO takes assembly code and the target machine's descriptions as input, proceeding as follows. First, it determines the effects of each assembly instruction that can be generated for the specified machine, which are expressed as *register transfer patterns*. This information forms a bi-directional translation grammar between the assembly code and the register transfers. PO then analyses each pair of adjacent instructions in the assembly code program, and converts them to equivalent instantiated register transfer patterns by means of the bi-directional grammar from the previous analysis. Afterwards, PO simplifies these patterns. Finally, it finds the best possible single assembly instruction that corresponds to the simplified pattern, by which the original instruction in the input assembly program is then replaced.

As this technique introduced a machine-independent approach to peephole optimisers, retargeting other machines was simply achieved by invoking PO on a different set of machine descriptions. Operating on register transfers rather than on assembly code also ensured that all possible optimisations were found without exhaustive case analysis. Retargetable peephole optimisation and PO in particular dominated research on peephole optimisation in the 1980's. A few papers have been published in connection with PO [4, 5, 6, 7, 8, 9, 10, 13]. The peephole optimiser of the GNU C compiler GCC is heavily inspired by PO.

Davidson and Fraser's idea of using register transfers as an intermediate code for compilers was appropriated later in the *RTL System* by Johnson

et al. [19]. Their work provides an object-oriented framework for optimising compilers.

Tanenbaum et al. [29] attempted a different approach by applying peephole optimisation to intermediate code instead of to object code: they required a peephole optimisation routine that was independent of the different compiler front and back ends involved. In order to maintain the portability of the compilers, optimisation had to be employed at the intermediate code level. Their design does not make use of any sophisticated strategies: Intermediate code is improved with a set of hand-written, common rules. The results indicate their system to be faster than Davidson and Fraser's retargetable approach. However, this is obvious since retargetable peephole optimisers are slower than their classical counterparts. Tanenbaum et al. certainly profited from optimisations at the intermediate code level being simpler because, for example, flags and register allocation issues do not need to be considered at that stage yet. But this is exactly the disadvantage of intermediate code optimisation: some optimisations do not occur until after code generation — so they cannot be dealt with at that phase.

In 1984, Davidson and Fraser built HOP, a fast compile-time peephole optimiser that uses PO [4, 7]. The aim was to automate the development of retargetable optimisers, i.e. automatically finding patterns for optimisation rules. To achieve this, PO is applied at compile-time, and a 'training set' of patterns is obtained. These aid HOP to infer its own patterns that approximately look like Lamb's optimisation rules (see section 2.1). HOP then utilises hashing to efficiently perform matching and replacement without string pattern matching or tree manipulation [15]. As opposed to the work described in this thesis, this is possible here because the optimisation rules use a fixed format for the specification of patterns. Thus, separating the skeleton of an instruction from its context-sensitive parts (i.e. operands) is straightforward. The patternised rules and input instructions are stored in different hash tables. In order to match a sequence of input instructions with rules, the addresses of the skeleton patterns are compared. If those 'match', the context sensitive information of the input is analysed for consistency with the rule. This way, HOP avoids dealing with strings and enhances matching speed to that of byte-to-byte comparison.

There are also other automatic generators for optimisation rules [20, 21]: Kessler's approach for instance does not require a training set. It simply utilises target machine descriptions in order to find optimisation opportunities. Warfield et al. [33] describe an expert system that 'learns' optimisation rules and thus simplifies the implementation of a retargetable peephole optimiser. Whitfield and Soffa present two tools, an optimisation specification language, and an optimiser generator, which automatically generate global optimisers [36]. These tools not only produce peephole optimisers, but they also compile code for parallel machines efficiently.

Later in 1984, Davidson and Fraser extended the original version of PO

and added common subexpression elimination (CSE) to it [5, 6]. The new version is divided into three phases that share the required tasks among them: the *Cacher* deals with CSEs, the *Combiner* simplifies register transfers, and the *Assigner* transfers these to assembly code.

Ganapathi et al. [1, 16, 17] took a completely different approach to retargetable code generation and optimisation. Instead of using register transfers, they employ attribute-grammar parsing techniques to describe target machine instructions. Pattern rules take the form of context-free productions, which have a set of attribute evaluation functions. The rules are similar to those described in section 2.1, having a preconditions and a replacement part. These rules improve the code while it is being parsed into a tree-like format. String pattern matching is typically applied here.

2.3 Integration of Code Generation and Peephole Optimisation

Although the retargetability of peephole optimisers had been successfully achieved in the early 1980's, speed always remained an issue. In order to overcome this, the overall execution time for the code generation and optimisation phases had to be reduced. As mentioned at the beginning of this chapter, considering optimisation in code generation itself would have resulted in complicated case analysis. So the goal here was to maintain the power of both phases without disturbing the individual algorithms too much. The solution lay in closely coupling both phases to a single phase.

In 1986, Fraser and Wendt [15] extended the initial implementation of HOP by combining its code generation and peephole optimisation phases to one phase, thus uniting PO and HOP. This new version of HOP is referred to as a 'general rule-based rewriting system', still operating by pattern matching through hashing. Lamb's idea of *escapes* (described in section 2.1) has also been added to HOP replacement pattern rules. The system saves time by improving code as it is being generated rather than in separate phases — a process which Fraser and Wendt call 'recycling'. Ancona [3] describes a similar rule-based rewriting system that optimises intermediate code (also see [29]).

Other work [5, 10] revealed that peephole optimisation can greatly improve code emitted by a naïve code generator. This idea, together with the integrated approach to code generation and peephole optimisation described above, can be found in Fraser and Hanson's retargetable C compiler lcc [11, 14]. lcc is not an 'optimising' compiler, however optimisation is included in (or rather 'hard-coded' into) code generation rules where necessary. In [11], lcc's code generator is described in detail.

Jacob Navia implemented lcc-win32 [27], which is lcc-based, but it targets the Windows platform only. He introduced an additional, separate

peephole optimiser for lcc-win32 that was not present in lcc.

The peephole optimiser described in this work utilises lcc-win32 as a tool only. Code emitted by lcc-win32's back end is improved by means of approximately 20 optimisation pattern rules (available on the CD). The optimiser rather follows the classical approach than the more modern retargetable one. This method was chosen as it is more efficient and suits better the purpose of this work. lcc-win32 merely provides a framework that this system can be tested and used with. Since the optimiser's rule set is incomplete in particular, the optimiser is not meant to replace Navia's optimiser in any form.

The next section gives a very general insight into the area of pattern matching strategies, providing background information to the techniques that have been explored in this work within the framework of peephole optimisation.

2.4 Pattern Matching Strategies

Because the field of pattern matching is relevant to many areas of computer science, much work has been done. The Knuth-Morris-Pratt (KMP) and the Boyer-Moore (BM) algorithms are classical in the context of string pattern matching. Basically, string pattern matching is concerned with locating *string patterns* in *text*. Both the pattern and the text consist of a specific sequence of characters and optionally variables. The aim here is to resolve, i.e. bind any variables so that the pattern and the text portions become equal. Traditionally, patterns are expressed with *regular expressions*, which provide a syntax for describing the sequence of characters to look for. This syntax allows abstraction over simple characters, so for example a single digit between 0-9 is `\d`, an n-digit number is `\d+` etc. Thus, we say that a pattern *matches* an input string if the string represents a valid instance of the pattern. The pattern fails to match a string if at any point during matching a contradiction occurs, i.e. the input string does not fulfil the pattern.

The LINUX command *grep* is a typical regular expression pattern matcher for text in files. The programming language Perl (**P**ractical **E**xtraction and **R**eport **L**anguage) specifically supports regular expressions-based pattern matching. Spinellis [28] describes a peephole optimiser written in Perl, aiming particularly at branch prediction. The optimisation rules are applied by string pattern matching, since they are specified with regular expressions only. This 'declarative' approach to pattern matching allows a simple specification and fast processing of the rules. However, the key advantage of pattern matching lies in being able to group regular expressions and to back reference them, simplifying the reuse of parts of the match. This saves the programmer from storing matched input in temporary variables,

which often results in unnecessarily complicated code. This approach is also used in the regular expressions-based strategy (see section 3.2.1).

This string-based approach to matching is surely very effective, but it merely processes the syntax of the assembly code input. The optimiser does not follow whether it is parsing and analysing a register, a constant, or a label definition — there is no semantic conception of the input at all. But pattern matching should not be understood as a mere 'mapping of strings'. In order to match a pattern of any kind, it has to be identified in the input. Hence, the input, which is some form of information, has to be processed. Thus above all, pattern matching is an *information processing problem*. It is a problem in which variables have to be bound and constraints have to be satisfied. Functional and logical programming languages are known to have in-built facilities for pattern matching. Often, functions or rules in these languages implicitly make use of this. In object-oriented languages like Java this is different: the smallest unit of information is an *object*, so everything is built around objects. Objects are characterised by their data fields and operations that can be performed on them. So by coding the object, the programmer also defines the object's *meaning*, its concept. In order to apply pattern matching here, one usually has to work on a lower level of abstraction, with strings and regular expressions as mentioned above. Visser and Lämmel [32] describe an implementation in Java that allows generic matching on object level. Their system makes intensive use of reflection for generic matching while also providing type-specific matching, which is usually faster. But it involves more programming effort. The generic matching strategy (section 3.3) uses many ideas from this. JMatch [24] is another attempt to introduce pattern matching in Java, though by language extension.

Mostly, as in the case of peephole optimisation, one not only wishes to identify patterns, but to replace them as well. So pattern matching is also a *rewriting problem*. Stratego [30, 31] is a term rewriting language that allows flexibility in strategy application by separating rule definitions from strategy specifications. Thus, according to the input, a suitable rewriting strategy can be chosen for application with a base set of rules. Peephole optimisers have always implicitly supported this design: Usually there is a separate rules file and a peephole optimiser skeleton in classical peephole optimisers. But as for strategical aspects of rule application, only the backwards strategy (see sections 2.1 and 3.2.2) has been applied so far. The next Chapter 3 discusses different pattern matching and rule application strategies for peephole optimisation.

Chapter 3

Strategies

In the context of peephole optimisation, this chapter presents a formal analysis of the pattern matching strategies that have been implemented. In section 3.1, first the framework of this dissertation is set up: Peephole optimisation is viewed from an information-processing perspective; the individual constituents that are involved in this process are introduced and formally defined. Then, in the sections 3.2 and 3.3 the strategical components are illustrated, which are the different pattern matching and rule application strategies that have been analysed in this work.

3.1 Introduction

This dissertation is about three issues:

- pattern matching in general – *the technique*,
- strategies for pattern matching – *the application* of this technique, and
- peephole optimisation – *the application area* for this technique.

What is pattern matching? As mentioned in chapter 2, the pattern matching problem (in this case within peephole optimisation), can be viewed as an information processing problem if generalised. Figure 3.1 shows the components that are involved in this process.

What exactly happens in this procedure; what kind of information is processed? The *input* to this system is some assembly code to be improved — this is the *incoming information*. Within the system, the peephole optimiser uses its *knowledge base* – the optimisation rules, to change parts of the code. The *interaction* between the assembly code and the rules is where the heart of the task is, where *information processing* occurs. The question here is: *how are these rules applied*; i.e., what kind of strategy is used? How exactly is this information processing performed? Possible answers to these

questions are presented in this dissertation (also see section 3.1.2). Upon a successful match, the code is altered, otherwise it remains unchanged. Once the application of rules is accomplished, the optimised assembly code is returned as the *output* of the system. This is basically what happens in a peephole optimiser.

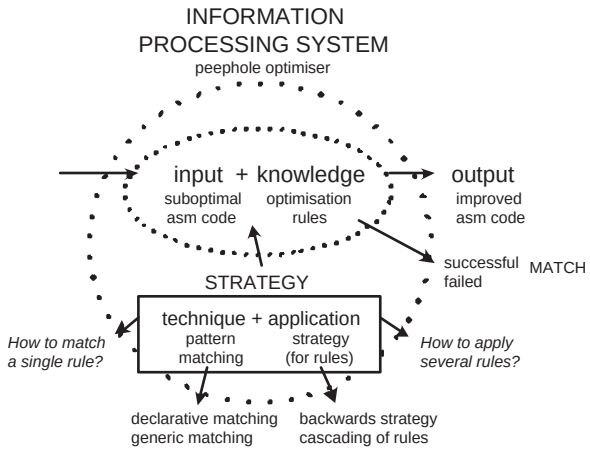

Figure 3.1: Peephole optimisation as an information processing problem

The next section contains formal definitions of the individual components and processes involved. The strategy part in particular is dealt with in the sections 3.2 and 3.3.

3.1.1 Basic Definitions

The definitions in this section strongly refer to the implementation. Let us start with the most general one, the *peephole optimiser*:

Definition 3.1 (Peephole Optimiser) *A peephole optimiser* $\mathcal{P} = (\mathcal{C}, \mathcal{O}, \mathcal{S})$ *accepts a sequence of machine code instructions* $\mathcal{C}$ *as its input. Its task is to replace suboptimal sequences in* $\mathcal{C}$ *with more efficient ones. Hence, a set of optimisation rules* $\mathcal{O}$ *are applied to* $\mathcal{C}$ *in a manner defined by the specified optimisation strategy* $\mathcal{S}$*. This is repeated until no more improvement is possible. The output of the peephole optimiser is the final improved version of the set of machine code instructions* $\mathcal{C}$*.*

This leads to the following definitions:

Definition 3.2 (Suboptimal instructions) *A sequence of machine code instructions* $\mathcal{C}_i = (c_1, \ldots, c_n)$ *is suboptimal, if there exists a sequence of machine code instructions* $\mathcal{C}_j = (c_1, \ldots, c_m)$ *with* $\mathcal{C}_i \neq \mathcal{C}_j$ *and* $\mathcal{C}_i \neq \emptyset$ *so that:*

- $\mathcal{C}_j$ *has the same effect as* $\mathcal{C}_i$
- *the sequence* $\mathcal{C}_j$ *is either faster than* $\mathcal{C}_i$ *as for execution time, or it is shorter, i.e.* $|\mathcal{C}_j| < |\mathcal{C}_i|$.

Such sequences $\mathcal{C}_i$ *and* $\mathcal{C}_j$ *are expressed by means of optimisation rules.*

The expressions *term* and *term pattern* ([32]) have to be clarified first before defining optimisation rules:

Definition 3.3 (Terms and Term Patterns) *Ground symbols are terms. Terms with variables are term patterns. A term pattern becomes a term if the variables in the pattern are instantiated with concrete terms (also called 'closed terms').*

Definition 3.4 (Optimisation Rule) *An optimisation rule* $\mathcal{O} = (\mathcal{C}_i, \mathcal{C}_j, \mathcal{E}_M, \mathcal{E}_R)$ *is defined over*

- *a sequence of suboptimal instructions* $\mathcal{C}_i \neq \emptyset$ *called the 'matching part'*
- *a sequence of 'improved' instructions* $\mathcal{C}_j$ *called 'replacements'*
- *a set of escapes* $\mathcal{E}_M$ *to be tested with* $\mathcal{C}_i$
- *a set of escapes* $\mathcal{E}_R$ *to be applied to* $\mathcal{C}_j$

$\mathcal{O}$ *typically contains term patterns in all parts. These are instantiated with closed terms during application of* $\mathcal{O}$*, which is also referred to as 'matching'.*

In this dissertation the expressions 'variable' and 'term pattern' are used synonymously. In Chapter 4 in particular, 'groups' and 'back-references' are also used in this context.

Definition 3.5 (Escape) *An escape consists of a function name and a set of variables* $v_1, \ldots, v_n$ *that the function shall be called with. An escape is a component of an optimisation rule. In the matching part of a rule the escape indicates a condition to be fulfilled in order for the rule to match, whereas in the replacement part it defines an additional action to be taken.*

Now the pattern matching problem can be defined as follows:

Definition 3.6 (Pattern Matching) *Given a problem* $\mathcal{P} = (\mathcal{C}, \mathcal{O}, \mathcal{S})$ *with* $\mathcal{C}$ *as the input,* $\mathcal{O}$ *being the knowledge base, and* $\mathcal{S}$ *a strategy, pattern matching is the process of finding distinct substitutions for the term patterns* $\mathcal{T} = \{t_1, \ldots, t_n\}$ *in the matching part of a rule* $o \in \mathcal{O}$ *so that* $\mathcal{T}$ *and the currently examined input sequence of machine instructions* $c_1, \ldots, c_n \in \mathcal{C}$

become equal. In case of a successful match, i.e. if such substitutions are found, the input sequence $c_1, \ldots, c_n$ is replaced with the instantiated version of the replacement part of r using the substitutions. Otherwise the match is considered to have failed.

For peephole optimisation, a specific case of pattern matching is particularly relevant:

Definition 3.7 (String Pattern Matching) *Given a pattern matching problem as defined in 3.6, in the context of string pattern matching the substitution s_p of a term pattern t_p and an input term i_q are considered equal if they denote the same sequence of characters. Thus for a successful match it holds that $s_p \equiv i_q$ and $|s_p| = |i_q|$.*

Definition 3.8 (Optimisation Strategy) *Given the pattern matching problem in 3.6, an optimisation strategy $\mathcal{S}$ is an algorithm that determines the order in which the rules $\{o_1, \ldots, o_n\} \in \mathcal{O}$ are applied to the input $\mathcal{C}$ so that the obtained result implements a specific aim.*

Such goals that the optimisation strategy implements can typically be optimisation for space or speed, or simply a preference of specific rules over others. Although the implementation of such aims mainly depends on the nature of the rules itself that are utilised, the application order is at least equally important – particularly for interacting rules.

3.1.2 Two concerns of this work

The main focus of this work is the strategy part shown in Figure 3.1, which is divided into two parts:

- ***Pattern matching strategy:*** This component particularly deals with the nature of the application of a *single* optimisation rule to the input (as in Definition 3.6). Matching strategies that have been implemented in this work are the more primitive declarative, regular expressions-based matching strategy, and a more abstract, object-oriented generic matching strategy.
- ***Rule application strategy:*** For this component, the *grouping* and *ordering* of the rules is analysed as for a certain aim (as in Definition 3.8). In the implementation, the backwards strategy has been employed for both the regular expressions strategy and the generic matching strategy. The generic matching strategy has been extended with the ability to cascade rules, i.e. to control several rule application sequences.

The rest of this chapter describes the algorithms and approaches to these strategies.

3.2 Declarative Pattern Matching

Pattern matching is a very powerful in-built programming feature in logical and functional programming languages (like e.g. PROLOG, LISP, Haskell, ML etc.). These languages allow conditions and relations between the involved data structures and variables to be expressed in a descriptive style. That is, the specifications are given in a declarative way, however the computation part of it is left to the machine, which in most cases is handled by an interpreter in these programming languages. This results in less code and fewer algorithms, and in easier maintenance ([28]), but also in a more compact representation of the logic of the program.

Imperative and procedural programming are counterparts to this paradigm, since in those languages (examples: C, Fortran, Pascal) the exact steps that shall be taken to obtain the result are expressed through methods or procedures. Consequently, more code is written. In Java, both programming styles can be adopted, although the procedural/imperative one is more common as objects are controlled by means of instructions in object methods. The regular expressions strategy illustrated in the next section uses the declarative approach, which is also encouraged by the `java.util.regex` package. Particularly the specification of the rules is expressed in a declarative style as in [28], using back-references and groups (see section 4.2.3 for more information).

3.2.1 Matching with Regular Expressions

The actual matching algorithm `testRule` (Algorithm 1) for the regular expressions strategy is straightforward in that it merely applies the matching pattern of the optimisation rule to the input, calling a few other subroutines for tests. The relevant parts of the input are extracted to a single-line string in `optimise` (Algorithm 2) before `testRule` is called. Thus in this latter method, only the rule has to be converted: although individual lines are already stored as strings – they are just concatenated with new line characters. The precise format of the rules is discussed in section 4.2.3, as this is a very implementation-specific feature. The 'matching power' lies in the declarative format of the rules, since those define the exact pattern to be matched by means of groups/variables. The other part of the task is performed by the matcher, which is handled internally in `java.util.regex`, saving a lot of programming effort here.

The subroutine `checkGroups` (not shown here) is required in `testRule` for testing whether all groups hold distinct matches. It checks all pairs of groups, causing the whole match to fail if identical matches for two groups are found. The matching classes in `java.util.regex` do not ensure this.

The methods `testEsc` and `applyEsc` that are used in `testRule` are not illustrated here, since their implementation is incomplete in the current ver-

Algorithm 1 Regular expressions-based matching (`testRule` in `RegExp`)

Arg 1: *s* - the unoptimised input sequence of machine code as a string
Arg 2: *r* - an optimisation rule
Returns: the replacement string for *s*

convert the match part of *r* to a single-line string ⇒ *pattern*
if *r* has replacements **then**
  convert them to a single-line string ⇒ *replace*
else
  replace = '@' for a successful match without replacements
end if
if *pattern* matches with *s* **then**
  apply *replace* to *s* ⇒ *result*
  // The following two instructions affect *result*
  call `checkGroups` to test for identical matches of several groups
  if *r* has escapes in the matching part **then**
    call `testEsc` to test these with *s*
  end if
end if
if *result* is empty (matching failed) **then**
  return *s*
else
  if *r* has escapes in the replacement **then**
    call `applyEsc` to apply them to *result*
  end if
  return *result*
end if

sion of the system.

The result of `testRule` is the following: if the replacement string is empty, i.e. matching has failed, the input string is returned to `optimise` as it was. Otherwise for a successful match, the replacement string is returned, which is then converted back to single assembly code lines. '@' is returned to indicate that the match was successful, but there are no replacements to be made (the matched lines shall simply be deleted).

The advantages of this algorithm are obvious: due to the declarative nature of the rules, only little programming effort is required in the matching routine. There is no need to handle variables and matches – it is all done by the regular expressions matching engine. One only has to carefully consider the format of the rules. However, since matching is performed on a string basis only, this approach is limited and rather 'dumb'. The matcher has no knowledge about what is being matched. This deficit becomes evident particularly when dealing with look-ahead issues in the next section.

3.2.2 Backwards Strategy

The backwards strategy has been introduced in chapter 2 in connection with Lamb's work [23]. In the implementation, it is used as a rule application strategy for both the declarative pattern matching approach and the 'basic' generic one. Algorithm 2 illustrates the logic of the algorithm independent of the pattern matching strategy utilised. Algorithm 3 and later Algorithm 4 show the declarative matching-specific and the generic-specific backwards strategy algorithms.

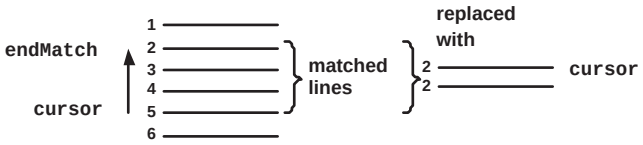

Figure 3.2: The backwards strategy.

Figure 3.2 illustrates a matching example with the backwards strategy. Initially, the cursor is set to the top of the code – line 1 – advancing forwards as rules are being tested. Optimisation rules though are tested backwards with the input. The 'window' size or peephole of the optimiser is always as big as the matching part of the current rule requires. Therefore, in order for a rule to be 'testable', its matching part must fit into the currently accessible part of the assembly code input, which is the intervall defined by $[1; cursor]$. This results in shorter rules being preferred over longer ones because they are chosen for testing beforehand (section 5.1 contains a concrete example).

For a rule with a matching part of 4 lines, `endMatch` indicates the last line of the rule to be matched – `cursor` the first line (in the implementation, this is performed vice versa from `endMatch` to `cursor`, which does not interfere with the backwards strategy). If the replacement consists of 2 lines as in Figure 3.2, the 4 lines are replaced and the cursor is set to the first line of the replacement. This approach is required to detect all optimisation opportunities further up the code that the replacement might have introduced. Rescanning the assembly input becomes redundant with this approach.

Algorithm 2 shows a formalised version of this strategy. Upon a successful match-replacement operation the index to the rules file is set back and the complete set of rules is tested again. This ensures that no optimisation opportunities are skipped.

Algorithm 3 demonstrates an adapted version of the backwards strategy to the declarative matching approach. A major difference between the two versions presented is the introduction of look-aheads. As explained later in section 4.2.3 in detail, an optimisation rule (in this work) can contain parts that match an arbitrary number of assembly code lines. Such lines are matched and only copied to the replacement without alteration. However

Algorithm 2 The backwards strategy

```
Require: asm - the set of assembly code instructions to optimise
Require: rules - the set of optimisation rules
Returns: the optimised version of asm

for all assembly lines in asm do
  for all rules do
    get current rule ⇒ r
    if size of current input allows matching with r then
      do pattern matching according to strategy
      if matching is successful then
        if there are replacements then
          insert them into asm
          set cursor to first line of replacement
        else
          set cursor back
        end if
        for all matched lines in asm do
          delete them
        end for
        set index to rules back to first rule
      end if
    end if
  end for
end for
return the optimised version of asm
```

in order to detect these, one has to analyse preceding lines in the input. For the regular expressions strategy, this means that the number of matching and replacement lines for a rule have to be 'virtually' incremented so that the optimiser considers one more line further up in the input. The rule then has to be reapplied. This is repeated until a match is found or until the maximum number of look-aheads allowed has been reached.

This linear methodology of looking ahead is not neat. For a rule that requires a look-ahead of n in order to match, this algorithm must apply the rule n times to discover the match. Mostly, if the maximum look-ahead is set to a lower limit than n, the rule does not match at all (for the options see Table A.4). The next generic strategy handles these 'look-ahead rules' cases better by using a labels table.

Algorithm 3 The backwards strategy in `optimise` in `RegExp`

Require: *asm* - the set of assembly code instructions to optimise
Require: *rules* - the set of optimisation rules
Require: *lookahead* - the number of assembly lines that is looked ahead
Require: *cursor* - the index to *asm* where matching starts
Require: *endMatch* - the index to *asm* where matching ends
Returns: the optimised version of *asm*

```
set lookahead= 0
for all assembly lines in asm do
  for all rules do
    get current rule ⇒ r
    if size of current input allows matching with r then
      convert the to be matched part in asm to a string ⇒ input
      call testRule with input and r ⇒ output
      if matching was successful then
        if output equals '@' then
          nothing to replace - set the cursor back
        else
          split up assembly code in output and insert them in asm
          set cursor to first line of replacement
        end if
        for all matched lines in asm do
          delete them at endMatch (newly replaced lines are not
          touched)
        end for
        set index to rules back to first rule
      end if
    else if r needs look-ahead then
      if no more look-aheads allowed then
        set lookahead to 0, quit this rule
      else
        increase lookahead and match-replace information about r
        update index to rules to retry matching r with higher look-ahead
      end if
    end if
  end for
end for
return the optimised version of asm
```

3.3 Generic Pattern Matching

Generic pattern matching denotes a more abstract form of pattern matching that is conducted on an object level (as in [32]). The power of this matching approach does not lie in the format of the optimisation rules as in the declarative approach, but in the generic way of handling the patterns: A distinction is made between abstract data types – the objects themselves, and the information they hold – the 'primitive' types. These are mostly simple strings. Since the abstract component here encapsulates the primitive one, assigning a conceptional meaning to it, this allows more 'intelligent' matching: First the type is analysed; only if it is promising, then the primitive information – the 'contents' – are compared. Hence, this higher level of abstraction allows an assembly code line to be further divided into ordinary instructions and label definitions; the former ones further consisting of the opcode part and the arguments, whereas the latter ones typically contain label numbers. This kind of information is particularly useful for rules that require a look-ahead. Instead of consuming one more line of input for every matching attempt, on demand the generic strategy uses recorded information about assembly instructions and their components for quick retrieval with just a few array accesses.

This generic approach requires a more procedural implementation style, which is also more expensive in terms of the programming effort. In particular since at the lower and primitive matching level regular expressions are not employed, variables have to be handled by hand. However, the complexity is contained here, since pattern matching is only carried out at depth 1.

The extended generic strategy, which is discussed in section 3.3.2, goes one step further by searching for several overlapping optimisation opportunities that might otherwise disqualify each other. It then chooses the most promising sequence of optimisations. Experimenting with interacting rules here becomes a very interesting issue.

3.3.1 Matching on Objects Level

In this subsection, the different aspects of generic matching are discussed separately in detail.

Backwards Strategy

Algorithm 4 shows the backwards strategy for the generic matching approach. Significant differences in comparison to the original backwards strategy (Algorithm 2) are the following:

- the use of exceptions by the subroutines in the crucial matching part between START EXCEPTION and END EXCEPTION supersedes

Algorithm 4 The backwards strategy in `optimise` in `Generic`

Require: *asm* - the set of assembly code instructions to optimise
Require: *rules* - the set of optimisation rules
Require: *cursor* - the index to *asm* where matching starts
Returns: the optimised version of *asm*

```
for all assembly lines in asm do
  for all rules do
    get current rule ⇒ r
    if size of current input allows matching with r then
      // START EXCEPTION
      call checkTypes with cursor and r to do type-specific matching
      call specMatchReplace with cursor and r to do string-based
      matching
      if there are no replacements then
        set cursor back
      else
        set cursor to first line of replacement
      end if
      set index to rules back to first rule
      reset variables table and clear hash table
      // END EXCEPTION
      if exception occurs then
        reset variables table and clear hash table
      end if
    end if
  end for
end for
return the optimised version of asm
```

the necessity for conditions as in Algorithm 2. An exception causes the remaining part of the block to be skipped until END EXCEPTION. This simplifies the code noticeably;

- matching is divided into two parts here: *abstract type matching* (`checkTypes`, Algorithm 5) and *specific, string-based matching* (`specMatchReplace`, Algorithms 6 and 7);
- matching, deletion, and replacements of assembly code lines are not handled here, but instead in succession in Algorithms 6 and 7;
- there are two helping tables, the *hash table* and the *variables table*, which aid in the handling of matched strings and in the detection of duplicate matches (`checkVars`, Algorithm 9); These are emptied here

after a successful/failed match;

- look-ahead issues are detected and dealt with separately in different subroutines (Algorithm 10).

Abstract Type Matching

The first matching subroutine (`checkTypes`, Algorithm 5) that is called in `optimise` performs abstract type matching. The types used within the generic strategy were introduced earlier: an assembly code line is classified either as an ordinary instruction or a label definition. Rules can also contain so-called look-ahead instructions. Ordinary instructions are further subdivided into the opcode and specific arguments. Label definitions typically contain a label number. Chapter 4 precisely describes these data structures (especially see Figure 4.5 for the components of an instruction).

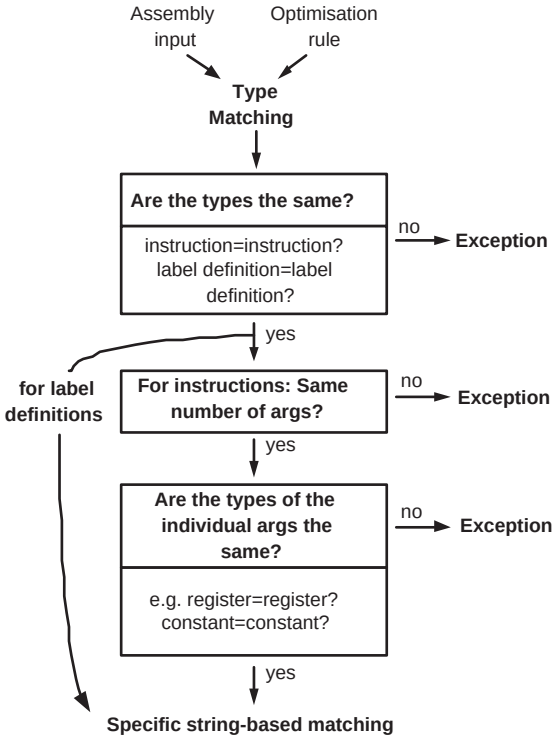

Figure 3.3: Abstract Type Matching in the generic strategy.

Figure 3.3 illustrates the abstract matching procedure that is formally specified in Algorithm 5. Matching is solely carried out by type comparison, which is realised by means of the Java reflection API in the implementation (similarly to [32]). The matching part of the current optimisation rule and the corresponding portion of assembly code input are tested line by line for

compatibility; i.e. only components of the same type match with each other. Instructions are then further analysed as to the number of arguments they possess, after which the compatibility of the individual instruction opcodes and arguments is tested. These tests can be interrupted by a mismatch any time, in which case an exception is thrown. The matching procedure finishes successfully if no exceptions occur. The rule and assembly code input are then handed on to the specific string-based matching procedure.

Additionally, in Algorithm 5 abstract type matching is stopped when a look-ahead instruction is encountered in a rule. It is not sensible to analyse the instructions further, since it is unknown how many lines will constitute the look-ahead.

Algorithm 5 Abstract Type Matching in `checkTypes` in `Generic`

Arg 1: *r* - the current rule that is being tested
Arg 2: *startMatch* - the index to *asm* where matching is started
Require: *endMatch* - the index to *asm* where matching is stopped
Require: *asm* - the set of assembly code instructions to optimise
Throws: an exception to indicate that the match failed

for all instructions in the matching part of $r \Rightarrow rLine$ and in $asm \Rightarrow asmLine$ between *startMatch* and *endMatch* **do**
  if *rLine* is a **look-ahead instruction then**
    stop and quit
  else
    if *rLine* and *asmLine* are instructions of different types **then**
      throw an exception and quit
    end if
    if *asmLine* is an **ordinary instruction then**
      if *asmLine* and *rLine* have a different number of arguments **then**
        throw an exception and quit
      end if
      for all arguments in $asmLine \Rightarrow asmArg$ and in $rLine \Rightarrow rArg$ **do**
        if *asmArg* and *rArg* are arguments of different types **then**
          throw an exception and quit
        end if
      end for
    end if
  end if
end for

Specific Matching and Replacements

Upon successful abstract type matching as shown in Algorithm 5, the specific matching procedure is invoked (`specMatchReplace`, Algorithm 6 and 7).

Algorithm 6 Specific Matching (part 1) in `specMatchReplace` in `Generic`

Arg 1: *r* - the current rule that is being tested
Arg 2: *startMatch* - the index to *asm* where matching is started
Require: *endMatch* - the index to *asm* where matching is stopped
Require: *asm* - the set of assembly code instructions to optimise
Require: the variables table where information about matches is stored
Require: the labels table with information about labels and jumps
Throws: an exception to indicate that the match failed

for all instructions in the matching part of *r* ⇒ *rLine* and in *asm* ⇒ *asmLine* between *startMatch* and *endMatch* **do**
  if *rLine* is a **look-ahead instruction then**
    // START LOOK-AHEAD EXCEPTION
    call `tryMatch` with *r*, the variables table, and the current index to *r* ⇒ index to end of look-ahead matching
    update index to *r* and *endMatch* according to this result
    // END LOOK-AHEAD EXCEPTION
    if a look-ahead exception occurs **then**
      call `lookaheadMatch` to perform line-by-line look-ahead matching ⇒ index to end of look-ahead matching
      update index to *r* and *endMatch* according to this result
    end if
  else if *rLine* is a **label definition then**
    call `match` with *rLine* and *asmLine* for string-based matching ⇒ *result*
    call `checkVars` with *result* to update the variables table
  else
    // *rLine* is an **ordinary instruction**
    for all arguments in *rLine* and *asmLine* **do**
      match each rule-assembly code pair by calling `match` ⇒ *result*
      call `checkVars` with *result* to update the variables table
    end for
  end if
end for
if *r* has escapes in the matching part **then**
  call `testEsc` to test these with *r* and the variables table
end if
call `delLTable` with *endMatch* and *startMatch* to delete information about the matched portion from the labels table

Algorithm 7 Specific Replacements (part 2) in `specMatchReplace` in `Generic`

Arg 1: *r* - the current rule that is being tested
Arg 2: *startMatch* - the index to *asm* where matching is started
Require: *endMatch* - the index to *asm* where matching is stopped
Require: *asm* - the set of assembly code instructions to optimise
Require: the variables table where information about matches is stored
Require: the labels table where information about labels and jumps is stored
Throws: an exception to indicate that the match failed

```
if there are replacements then
  for all instructions in the replacement part of r ⇒ replLine do
    if replLine is a look-ahead instruction then
      copy the matched part in asm to the correct position after
      endMatch
    else
      // replLine is an ordinary instruction or a label definition
      instantiate replLine in createReplacmnt and add it to the correct
      position in asm after endMatch
    end if
  end for
  call updLTable to update the labels table with the replacement infor-
  mation
  for all matched lines in asm do
    delete them
  end for
end if
```

This algorithm is divided into two parts here for a clearer layout. It treats the three different types of assembly lines in a separate manner. In case of look-ahead instructions, first the more 'intelligent' matching routine (`tryMatch`, Algorithm 10) is called. If that fails by throwing a special look-ahead exception, the method `lookaheadMatch` is invoked instead, which performs line-by-line look-ahead matching similarly to the regular expressions strategy.

Label definitions are handled by calling the string-based matching routine `match` (Algorithm 8), which is discussed in the next paragraph. The same procedure is applied to ordinary instructions; however it is repeated for each component of the instruction, i.e. the opcode and the arguments are matched individually. Since label definitions contain a label number only, one pass is sufficient for those. `checkVars` (Algorithm 9) updates the variables table after a match, where the matches are maintained. If the matching procedure fails at any time, the thrown exception is passed on to

`optimise`, where it is dealt with.

Escapes are invoked analogously to the regular expressions strategy; again the methods `testEsc` and `applyEsc` are not discussed here as they have not been implemented yet. In this strategy, the application of the escapes is handled with the replacements (see `createReplacmnt` in `GenASM` on the CD).

The second part of the algorithm shown in Algorithm 7, illustrates the application of the replacements. If a look-ahead instruction is encountered, the corresponding matched 'look-ahead' portion in the assembly input file is simply copied to its new position without alteration. Actually it is rather moved, since the 'source' lines are deleted later on (last `for` loop in this algorithm). Label definitions and ordinary instructions are simply replaced by retrieving the correct information from the variables table. Therefore the routine `createReplacmnt` is not explicitly described here.

Finally, the calls to `delLTable` and `updLTable` in Algorithms 6 and 7 manipulate the labels table, which is discussed later in this section.

String-based Matching and Replacements

As expressed in Definition 3.7, the problem of string pattern matching is concerned with finding a term substitution for a term pattern so that another term and the substitution become equal. Given a string $term_n$, the following typical 'patterns' can occur in a string matching problem within the context of peephole optimisation:

$pattern_1 := term_1$
$pattern_2 := variable_1$
$pattern_3 := term_1\ variable_1$
$pattern_4 := term_1\ variable_1\ term_2\ variable_2 \ldots$

Therefore, in order to match a $term_n$ with one of the patterns above, it has to be split in such a way as to 'fit' into the pattern. The first two cases are simple since they can be matched to the whole term immediately – resulting in a successful or failed match (the second pattern matches always). Patterns 3 and 4 though require the next pattern character to be found in the term (=search string) whenever a variable is encountered in the pattern. One has to look ahead in both the term and the pattern in order to determine the length of the string to be matched with the variable. This matching procedure is described in Algorithm 8. The generic expression 'item' signifies terms and variables (which occur in patterns). Matches for the variables are stored in a temporary variables table together with the type of the item that has been matched (register, label etc.). This information is later retrieved for the replacements. Figure 3.4 illustrates a typical matching and replacement example and the use of the variables table:

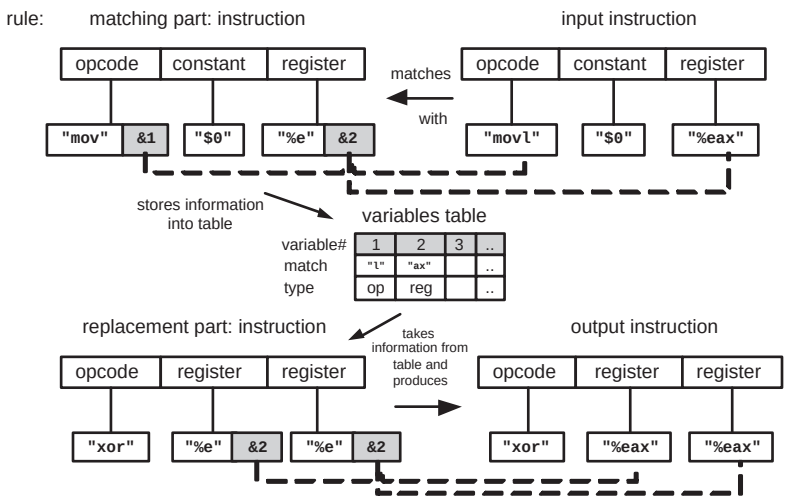

Figure 3.4: A matching and replacement example.

The input instruction here is the string `"movl $0,%eax"`, which matches with `"mov" &1 "$0" "%e" &2`. `&1` and `&2` denote variable numbers 1 and 2 here (please refer to Chapter 4 for a description of the syntax). The corresponding string portions are stored in the variables table. When creating the replacement, the variables (boxes with grey background) are replaced with the information in the variables table. The type information in the table is utilised if the exact type of the component is not evident in the replacement instruction. Here, the result is the string `"xor %eax,%eax"`.

The information in the temporary variables table has to be transferred to the actual one where all matches are kept for a single rule. This is sensible in particular for rules with several instructions in the matching part, since the initialisations must be consistent within a single rule (unlike the example in Figure 3.4). Algorithm 9 illustrates the `checkVars` function that implements this. Duplicate matches, i.e. different variables that denote the same string are detected by means of the *hash table*. This approach is more efficient than doing a linear search in the variables table to discover whether a match has already been associated with another variable. Both the variables and the hash table are emptied after a single matching attempt with a rule.

Algorithm 8 String-based matching in `match` in `Type` and extending classes

Arg: s - the input string/term that is matched to the pattern
Require: p - the pattern that consists of variables and terms
Returns: a variables table that contains initialisations for this match
Throws: an exception to indicate that the match failed

if p consists of a single term only and if $p \neq s$ **then**
  throw an exception and quit
else
  // p contains at least 1 variable
  repeat
    advance in p item by item
    if current item in p is a term and this item $\neq$ a portion of the remaining input in s **then**
      throw an exception and quit
    else
      // current item in p is a variable
      if there are no more items in p **then**
        fill the variables table at position [variable] with the remaining input in s and remember s' type
      else
        // more items in p
        find the next pattern character in $s \Rightarrow$ else throw an exception and quit
        fill the variables table at position [variable] with the remaining input in s up to the next pattern character and remember s' type
      end if
    end if
  until p is finished
  if s is not finished because $|s| \neq |p|$ (it is longer in this case) **then**
    throw an exception and quit
  end if
end if
return the variables table

So overall we can conclude that generic matching fails if

- the abstract types are incompatible (*generic matching*)
- two terms are not equal (*string-based matching*)
- the escapes are not fulfilled

- a variable that already denotes a match shall be instantiated with a different string (*variables table*)
- any two variables in the table denote the same string (no distinct matches; *hash table*)

Algorithm 9 Updating the variables table: `checkVars` in `Generic`

Arg: *invar* - the incoming variables table
Require: *var* - the actual variables table to which data shall be copied
Require: the hash table that keeps track of duplicate matches
Throws: an exception if an inconsistency is detected

for every variable ⇒ v in *invar* that contains a match **do**
  if *var* does not contain an initialisation for v **then**
    copy the info from *invar* to *var*
    if this match is associated with another variable according to the hash table **then**
      throw an exception and quit
    end if
  else
    // *var* contains a match for v
    if the matches in *var* and *invar* for v are different **then**
      throw an exception and quit
    end if
  end if
end for

Labels Table

The key advantageous characteristic of the generic strategy does not lie in the ability to perform matching on two levels – abstract and specific – but in the way rules with look-ahead instructions are handled. Instead of exhaustive line-by-line matching, as adopted in the regular expressions strategy (Algorithm 3), information about specific instructions is utilised to access possible matching candidates much quicker. I concluded that in rules, matching an arbitrary number of assembly instructions seems particularly sensible around jump instructions and label definitions. For this, a labels table that keeps track of all label definitions in the input assembly code together with any jumps to these definitions is necessary. This is achieved by remembering their line numbers (basic integers): the table is accessed by means of the label number, where the corresponding line number of the label definition or those of any jumps to this label can be retrieved. Figure 3.5 shows an example labels table.

label#	1	2	3	..
label line#	5	12	17	..
jump line#s	8 14	16 24	29	..

Figure 3.5: An example labels table.

The labels table is created and filled with line numbers during parsing of the assembly code input, as described in section 4.2.2. It is then updated in the generic matching strategy after every matching and replacement operation as line numbers change. This is realised by means of the routines `updLTable` and `delLTable`. The latter function resets or deletes any line numbers of instructions that are among the matched lines, i.e. between `startMatch` and `endMatch` in the assembly input. The former one is more complicated: it not only adds to the table newly replaced jump instructions and label definitions, but it also updates the rest of the labels table so that subsequent lines are shifted forwards properly. For this, the overall difference of matched and replaced lines is calculated, which then becomes the 'shift factor' for any such instructions.

Look-ahead Matching

Algorithm 10 shows the predictive look-ahead matching procedure `tryMatch` that is invoked for rules with look-ahead instructions. The algorithm currently covers two cases, which are illustrated in Figure 3.6 (concrete examples for these cases are available in section 5.1). The approach taken here is purely predictive: The aim is to guess the position of `endMatch` in the examples in Figure 3.2 by using the labels and variables tables only. Whether or not the prediction was correct is not known until the routine returns to `specMatchReplace`, where matching resumes with the first instruction before the look-ahead.

In the first example (case 1), the position of the jump instruction is guessed by using the `jumpTo` information to locate the closest jump instruction further up the code to label `jumpTo`. If label `jumpTo` is not initialised yet, a look-ahead exception is thrown to do line-by-line matching instead with this rule. This approach works fine for most simple rules. However, a possible future extension might be to try all available jump instructions in case the first one is incorrect.

The second example is more complex: first the position of the upper label definition is guessed. If the label number is not available yet, the definition's position is predicted by looking at the instruction after the look-ahead. If that is a label definition, it must contain variables that have been initialised already (due to the backwards strategy). The position of the upper label

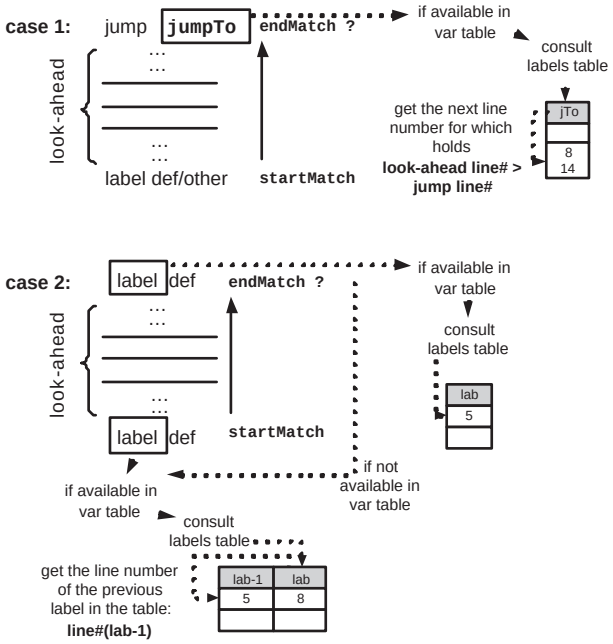

Figure 3.6: The two cases for predictive look-ahead matching.

definition is then guessed by trying the previous label definition in the table. Again, the functionality here could be extended in future.

The alternative to this method – the line-by-line look-ahead matching procedure `lookaheadMatch` combines abstract type matching and specific string-based matching for instructions to predict the size of the look-ahead. Label definitions are matched by type only, string-based matching for them is conducted in `specMatchReplace`. This approach is more accurate than the predictive matching procedure above on condition that the maximum number of look-ahead attempts is high enough to match the rule. However, this routine operates at the same level as the look-ahead matching procedure in the regular expressions strategy. Therefore, it is utilised as an alternative here to the predictive method.

To compare the two pattern matching strategies presented so far, it can be concluded that in the generic matching strategy rules that do not match are discarded much earlier than in the regular expressions strategy. This is due to abstract type matching, which is of advantage. However, for rules that actually do match, matching has to be performed twice — the abstract type matching and the string-based one. Still, the generic matching strategy is generally faster than the regular expressions one (as shown in Chapter 5). The next section presents an alternative or rather an extension to the backwards strategy, which is especially interesting for interacting rules.

Algorithm 10 Predictive look-ahead matching: `tryMatch` in `Special`

Arg 1: *var* - the variables table
Arg 2: *r* - the rule with the look-ahead that is being applied
Arg 3: *i* - the index to the instruction before the look-ahead in *r*'s matching part
Require: the labels table
Throws: a 'match exception' to indicate that the match failed; or a 'look-ahead exception' to invoke line-by line matching

look at instruction *i* in the matching part of *r* ⇒ *bLine*
if *bLine* is a **jump instruction then**
  if the 'jumpTo' label number is available (consult *var*) **then**
    try to get the next jump instruction to this label definition further up the code from the labels table at position [jumpTo]
    if it has been found **then**
      return its line number to `specMatchReplace` so that the size of the look-ahead is known
    else
      throw a match exception and quit
    end if
  else
    throw a look-ahead exception and quit
  end if
else if *bLine* is a **label definition then**
  if the label number is available (consult *var*) **then**
    return its line number to `specMatchReplace` so that the size of the look-ahead is known
  else
    look at the instruction after the look-ahead ⇒ *aLine*
    if *aLine* is a **label definition then**
      if the label number is available (consult *var*) ⇒ *labelNo* **then**
        return the line number of label *labelNo*-1 to `specMatchReplace`
      else
        throw a match exception and quit
      end if
    else
      throw a look-ahead exception and quit
    end if
  end if
else
  throw a look-ahead exception and quit
end if

3.3.2 Cascading of Rules

The advantage of the backwards strategy lies in the fact that the assembly input needs to be scanned once only, and that new optimisation opportunities are found instantly. However, there is an important flaw in this strategy: Due to the nature of the approach shorter rules are preferred over longer ones. This is not always desirable because the shorter rules mostly disqualify the longer ones that might be optimising more or better. More precisely, the relationship between rules can be specified as follows:

Definition 3.9 (Choice point) *Given a pattern matching problem as defined in 3.6, a choice point describes a state in the input $\mathcal{C}$ where at least two rules o_i and $o_j \in \mathcal{O}$ are applicable to the input sequence $c_1, \ldots, c_n \in \mathcal{C}$.*

Definition 3.10 (Interaction of Rules) *Given a choice point as defined in 3.9, if a matching rule o_i either disqualifies another matching rule o_j, or causes to qualify a rule o_j that did not match before, the rules o_i and o_j are described to interact.*

This relation is also valid for sets of rules $\mathcal{O}_k = \{o_1, \ldots, o_n\}$ and $\mathcal{O}_l = \{o_1, \ldots, o_m\}$.

According to the aim one wants the optimisation strategy to implement, editing the rules file is a possible but clumsy solution to prevent or enable the interaction of rules. A much better option is to use an optimisation strategy that searches for alternative optimisation sequences to apply the most promising one. The aim here is not to find the best optimisation sequence(s), which according to [18] is NP complete. The rules cascading strategy that has been employed here (described in Algorithm 11) as an extension to the generic strategy, uses the advantages of the backwards strategy, looking a few rules ahead to apply the best available optimisation sequence. The look-ahead limit can be manipulated by the user to experiment with the results. Within a single optimisation sequence, the maximum number of assembly lines analysed after each match is also limited so that scanning of the whole input is avoided. The criterion for choosing the 'best available optimisation sequence' is optimisation for space, i.e. the sequence of rules that eliminates most of the code is chosen. For this purpose, artificial choice points are created, which define the start and end points of alternative optimisation sequences. The results are maintained in so-called 'options':

Definition 3.11 (Option) *An Option saves the state of a pattern matching problem (3.6) between two choice points. Hence it holds the most recent information about the set of rules that have been applied, the state of the input, the labels table, and the cursor pointing to the input. Additionally, it also tracks the total sum of eliminated lines.*

Algorithm 11 The rules cascading strategy in `optimise` in `ExtGeneric`

```
// in addition to the requirements in Algorithm 4:
Require: options - the set of optimisation sequences currently explored
Require: o - the currently analysed optimisation sequence
Returns: the optimised version of asm

for all assembly lines in asm do
  for all rules do
    get current rule ⇒ r
    if rules quota has been reached for o or if enough assembly lines have
    been tested since the last match then
      if there is room in options for o then
        remember the state of asm, the labels table, and cursor in o
        add o to options and reset o
        set back asm, labels table, rules index, and cursor to their pre-
        vious states at last choice point
      end if
      if options is full then
        apply the best option in options ⇒ bestO: update asm, the labels
        table, rules index, and cursor according to bestO
        empty options
      end if
    else if size of current input allows matching with r then
      // START EXCEPTION
      call checkTypes with cursor and r
      call specMatchReplace with cursor and r
      remember r and cursor in this o
      if there are no replacements then
        set cursor back
      else
        set cursor to first line of replacement
      end if
      set index to rules back to first rule
      reset variables table and clear hash table
      // END EXCEPTION
      if exception occurs then
        reset variables table and clear hash table
      end if
    end if
  end for
end for
if options is not empty then
  apply the best option
end if
return the optimised version of asm
```

Chapter 4

Implementation and Design

In this chapter, implementation-specific aspects of the system formally described in Chapter 3 are explained. First, the packages and the Java class structure are illustrated. Then, the syntax of the lcc-win32 assembly language is briefly portrayed. In the remaining part of section 4.2, the two parsers are described.

4.1 Overview

The peephole optimiser described in this dissertation consists of two main parts – the *strategies* and the *parsers*. The former aspect was the main focus of Chapter 3. Here, amongst other things, the parsers shall be described. Their task is to supply the peephole optimiser with the required information, i.e. with the processed assembly code input and the rules set. The optimiser then improves the assembly code by utilising the chosen strategy, returning the optimised output.

The system has been implemented in Java, version 1.4.2. Since the main concern of this project is the exploration of different approaches to a single problem – here pattern matching – a programming language that allows *parts of the system to be designed differently for interchanging* suits best. The notion of object orientation fits here. The strategies have been designed so that common parts of the algorithms can be reused by inheritance, whereas specific parts are overwritten where necessary. This is of advantage to the design and conceptual aspects of this project. As for the individual strategies, the regular expressions strategy profits from the `java.util.regex` package as mentioned earlier; the generic strategies require an object-oriented environment in particular because of the use of more abstract data structures.

In the following subsections the structure of the implementation is represented diagrammatically; each class is briefly described with respect to its main function in the optimiser.

4.1.1 Class Structure

Figure 4.1 shows the package `mresproj` with the main entry class `POpt` and the sub-packages `strategies` and `parsers`. The latter sub-package further includes the packages `asm` and `rules`, which contain code for parsing assembly input code and optimisation rules.

As Figure 4.2 indicates, `POpt` is invoked with a rules set, (unoptimised) assembly code obtained by lcc-win32, a specified strategy, and with the options shown in Table A.4 in the appendix.

In the optimisation process first `POpt` hands on the rules set to `parsers.rules.RulesParser` and the assembly code to `parsers.asm.ASMParser` so that the required data structures can be created according to the specified strategy (also see 4.2.2 and 4.2.3). These are then passed on to one of the strategy classes in `strategies`, where the actual optimisation is performed: `RegExp`, `Generic`, or `ExtGeneric` (all descendants of `strategies.Strategy`). Finally, the optimised assembly code is printed to a file in `POpt`.

4.1.2 Organisation of Parsers

`mresproj.parsers` contains the two sub-packages `asm` and `rules` (not shown in Figure 4.3) with the JavaCC-created parsers for rules sets and assembly code. Besides, there are also the following classes in this package:

- `ASM:` This abstract class stands for a single line of assembly code. Strategy classes can thus define their own meaning of an assembly line by extending this class.

- `NonASM:` This class extends `ASM`, representing a line of assembly code that is irrelevant to the peephole optimiser: These are typically comments, assembler directives, and labels with method names. Such lines are not considered by the optimiser; so they are filtered from the relevant assembly code by the assembly parser, to then be reassembled again in `POpt` for the production of the optimised output after optimisation is completed.

- `Rule:` This class defines an optimisation rule that mainly consists of a matching and a replacement part (as in Definition 3.4). These contain specific `ASMs` according to the chosen strategy. Data structures for escapes are also available, although these have not been explicitly implemented yet in the system.

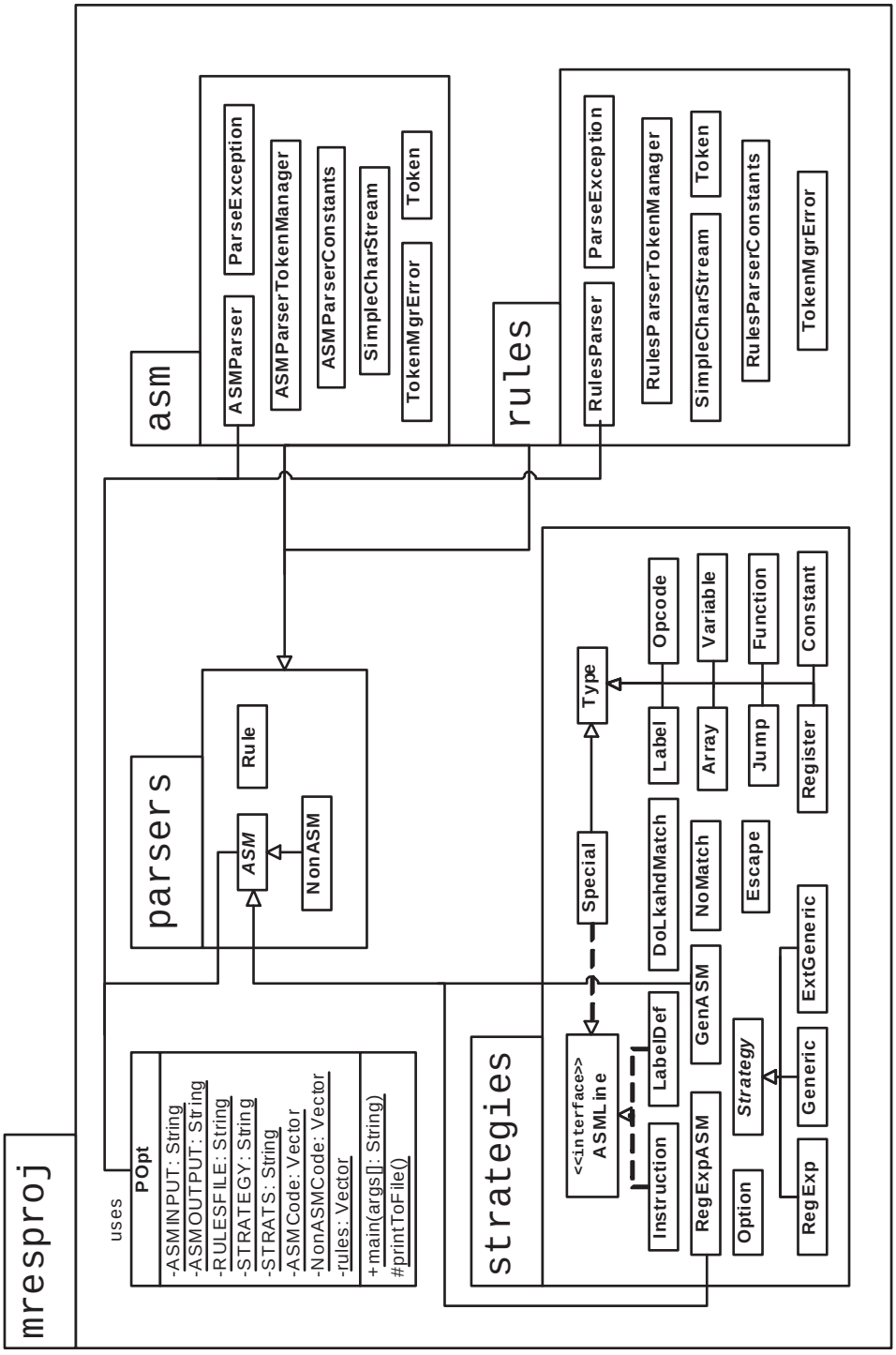

Figure 4.1: The package `mresproj` with its class and sub-packages.

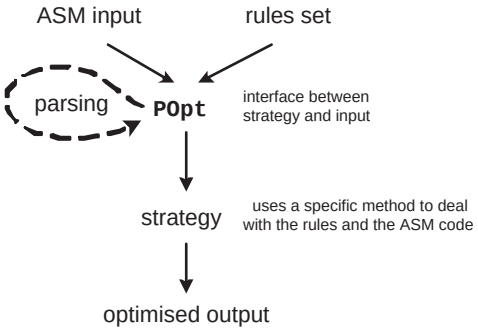

Figure 4.2: The optimisation process.

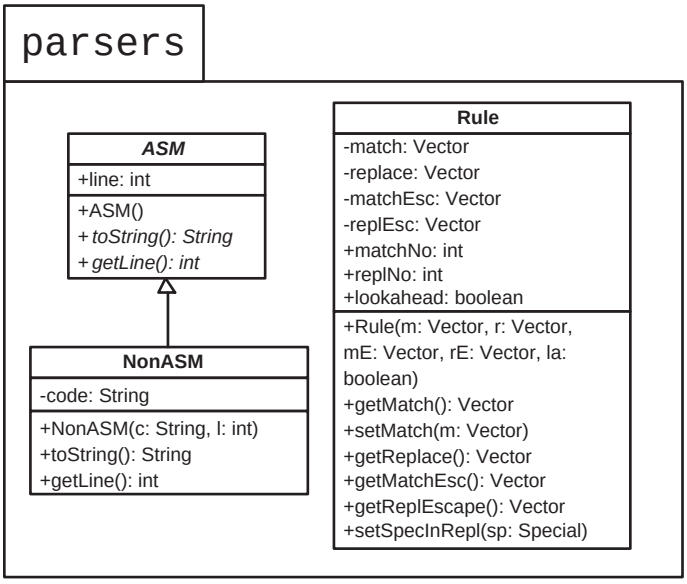

Figure 4.3: The package `mresproj.parsers` with its classes (sub-packages asm and rules are not shown).

4.1.3 Organisation of Strategies

`mresproj.strategies` contains the following strategy classes:

Elementary classes:

- `Strategy`: This abstract class describes the gist of a strategy so it must therefore be extended by implementing strategies. The `optimise` method is the main entry method that is invoked by `POpt` to initiate the optimisation.

- `NoMatch`: This is an exception class, which is used to indicate that a match attempt failed. This is a useful way of handling match failures at any point during optimisation, especially with regard to the generic matching strategies.

Classes for the regular expressions-based strategy:

- `RegExp`: This is the regular expressions strategy class. The method `prepMatch`, which prepares the format of the rules, is called before the optimisation is started (see section 4.2.3 for more information on the format of the rules). `testRule` (Algorithm 1, Chapter 3) is the actual method that performs the regular expressions-based matching and replacements.

- `RegExpASM`: This class defines a line of assembly code for the regular expressions strategy, which is simply stored as a `String`.

Classes for the generic strategy:

- `Generic`: This class contains the main algorithms for the generic strategy. The detailed functionality of this class is described in section 3.3.1. This class closely cooperates with `GenASM`, `Rule`, and `Special`.

- `GenASM`: Within the generic matching strategy, a line of assembly code can take different shapes. This is expressed by means of classes that implement the interface `ASMLine`, which is part of a `GenASM`.

- `ASMLine`: A generic assembly code line can either be an ordinary `Instruction`, a `LabelDef` (label definition) or a `Special`, which is the data structure for a look-ahead instruction, referring to an arbitrary number of assembly lines.

- `Instruction`: This class stands for an ordinary assembly instruction with one opcode and optionally up to four arguments. These individual components of an `Instruction` are descendants of `Type`; for `Instructions` that are part of a `GenASM` within a `Rule`, the components can be instances of `Type` itself.

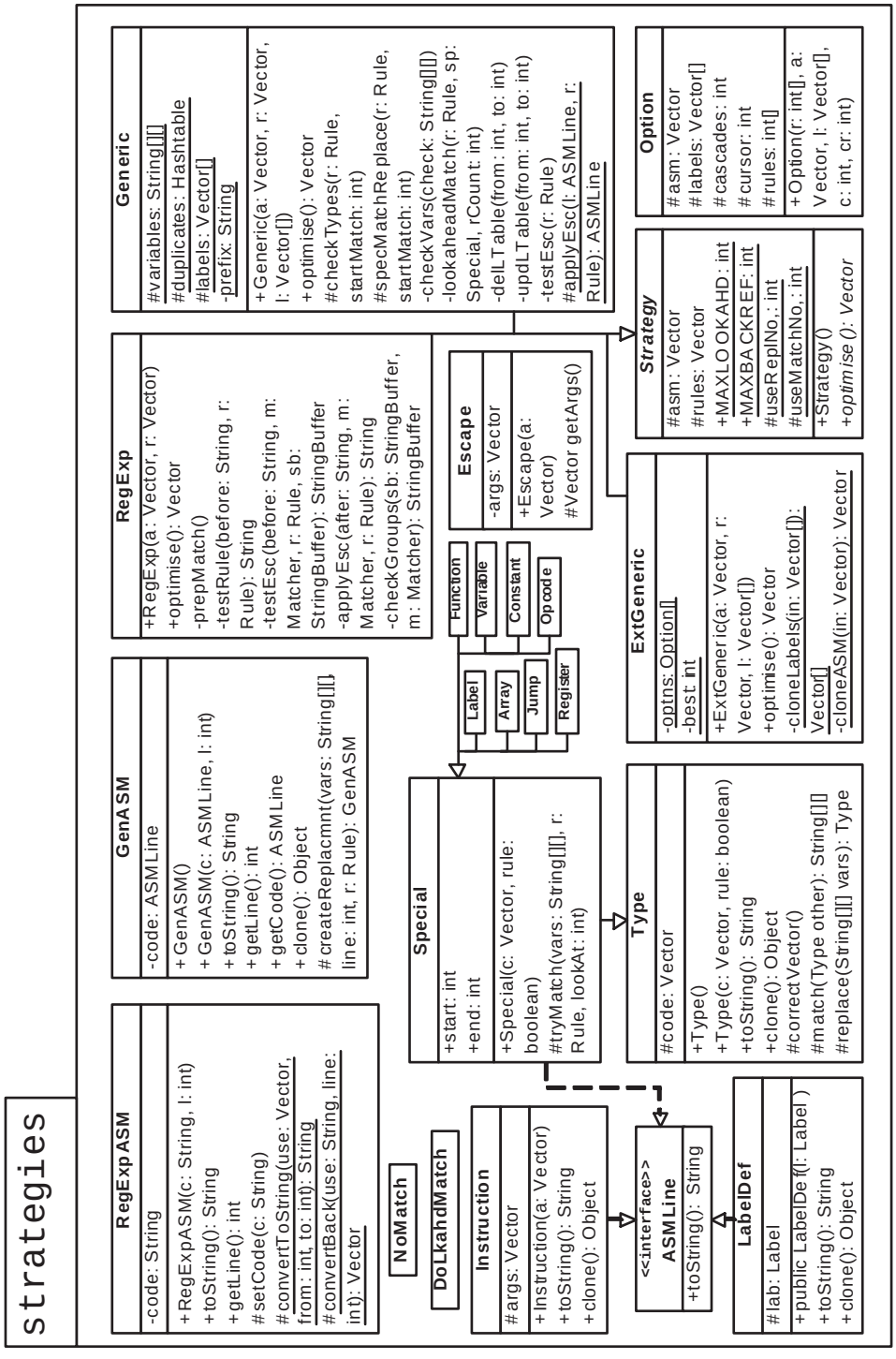

Figure 4.4: The package `mresproj.strategies` with its classes.

- `LabelDef:` Label definitions are expressed with this class. A `LabelDef` holds a `Label` which refers to a specific label number.

- `Type:` All basic elements of a `GenASM` are instances of this class (see Figure 4.5). Their implementations are almost identical. A `Type` and its subclasses store the information they hold as a `String` (or a collection of `Strings` in case of direct instances of `Type`). The crucial matching method is `match` (Algorithm 8), where matching is performed on a string basis. The method `replace` in which the string-based replacements are accomplished, is defined in `Type` only and hence it is directly used by the extending classes. In a `Rule`, a direct instance of this class refers to an uninstantiated element – a term pattern or variable as described in Definition 3.3. In this chapter, the expression 'group' is rather used. So assembly code never contains a direct instance of this class, since any existing `Types` are downcast to instances of extending classes during optimisation (i.e. all groups are resolved).

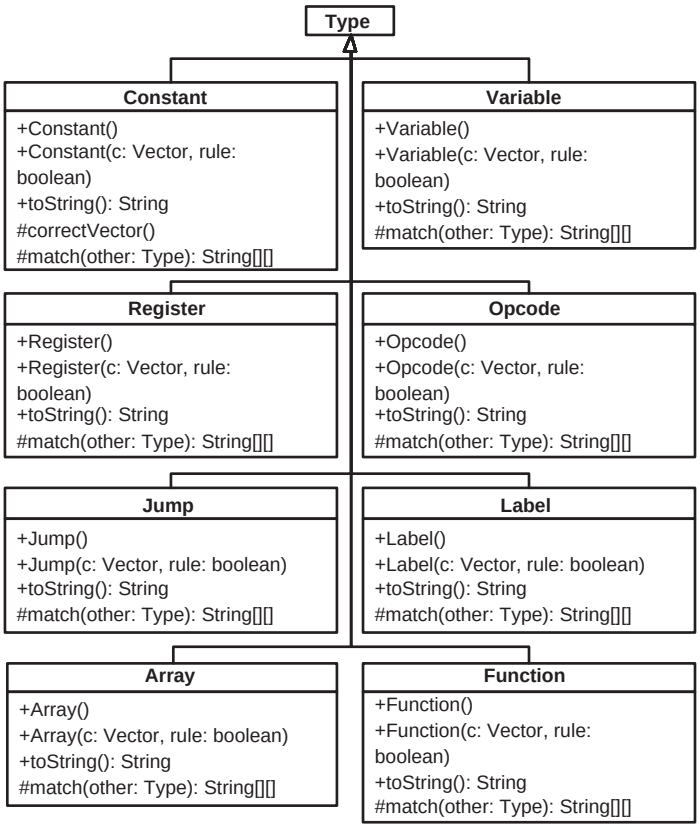

Figure 4.5: The class `mresproj.strategies.Type` with its subclasses (excluding `mresproj.strategies.Special`).

- `Special:` This is a special subclass of `Type` that can only occur in a `Rule`. It refers to a starred group/ look-ahead instruction, which is a group that can match an arbitrary number of assembly code lines.

- `Escape:` This class is not explicitly deployed in the current version of the system, since the example rules set utilised in Chapter 5 does not contain escapes. The idea is that an `Escape` holds a set of `Strings`, the first one referring to the name of the method (escape) to be called, and the rest to any arguments that this method shall be called with (formal definition on page 12).

- `DoLkahdMatch:` This exception is thrown if a `Rule` containing a `Special` cannot be matched by means of the labels table only (see section 3.3.1 for information on look-ahead matching), and a line-by-line matching procedure shall be attempted instead.

Additional classes for the extended generic strategy:

- `ExtGeneric:` This optimisation strategy extends `Generic`, overriding only the `optimise` method. Additional methods are necessary for cloning data structures only. `ExtGeneric` therefore works exactly as the simple generic strategy; the only difference lies in this strategy searching for alternative optimisation opportunities (remembered as `Options`) before actually applying a sequence of optimisations. This is done by looking ahead a predetermined number of assembly code lines, or until sufficient alternatives have been found (see section 3.3.2).

- `Option:` This class holds information about the state of the optimisation after a specific number of rule applications. This includes the state of the assembly code, the rules that have been applied, the number of lines eliminated, and the state of the labels table (as in Definition 3.11).

4.2 Parsers

The two parsers `ASMParser` and `RulesParser` have been created with JavaCC, version 3.2. The following section first clarifies a few aspects about the lcc-win32 assembly syntax that the parsers adhere to. Then, in sections 4.2.2 and 4.2.3, the two parsers are discussed in detail.

4.2.1 lcc-win32 Assembly Syntax

lcc-win32 uses the AT&T syntax for its assembly code – as opposed to the Intel syntax. Important features are:

- **Notation conventions:** Tables 4.1 and 4.2 show example notations for constants, labels, label definitions, variables, register, and arrays.

Notation	Meaning
`$4`	immediate operand; denotes the constant '4' here
`$_var`	static variable `var`
`_$1`	label number 1
`_$1:`	label definition for label number 1
`_functionName:`	start of the new function `functionName`
`;`	indicates the start of a comment
`.`	indicates the start of an assembler directive

Table 4.1: General notations

Notation	Meaning
`%exx` (32-bit) `%xx` (16-bit)	register operand; xx can be one of the following general purpose registers (GPR): `ax, bx, cx, dx, si, di, bp, sp`
`(%exx)`	value in address denoted by `%exx` *(indirect addressing mode)*
`8(%exx)` `(%exx,8)`	address of `%exx` plus an offset of 8 bytes *(indexed addressing mode)*
`_var(%exx)`	address of `%exx` plus an offset of `var`, which is a global variable
`(500)`	contents of address 500 *(direct addressing mode)*
`(%eax,%ecx)`	indirect addressing for `%eax + (%ecx)`
`(%eax,%ecx,2)`	indirect addressing for `%eax + (%ecx*2)`
`4(%eax,%ecx,2)`	indirect addressing for `%eax + (%ecx*2)` plus an offset of 4 bytes
`_ar(,%exx,4)`	position `%exx + 4` in array `ar`

Table 4.2: Registers and arrays

- **Mnemonics:** The instruction mnemonic or opcode is the first and compulsory element in an assembly code line. It consists of an operand and an operand-size attribute. This latter component is a suffix that refers to the size of the memory operand, which can be one of the following: `b` (=byte, 8-bit), `w` (=word, 16-bit), `l` (=long, 32-bit), and `q` (=quadruple word, 64-bit).
  Example: `movl %eax,%ecx` denotes the operation `mov` (move data) between the two registers `%eax` and `%ecx` of size `l` (=32-bit).

- **Instructions:** Expressed in simple EBNF style, the typical syntax of

an instruction is `mnemonic [arg1{,argn}]`. That is, the instruction mnemonic is followed by (optional) arguments that are separated by commas, which are to be read from left to right. Therefore, source operands appear before destination operands.
Example: `movl $4,%eax` assigns the value `4` to register `eax`.

- **Indentation:** In addition to the notation conventions mentioned above, lcc-win32 uses indentation to differentiate between various types and parts of assembly code (see Table 4.3). Obviously, each unit of assembly code is separated by a new line (`=\r\n`). The indentations below are inserted just *before* the corresponding element (except for mnemonics, as stated). The parsers introduced in the next two sections strictly follow these conventions; so this must be conformed to when editing or creating assembly files by hand.

Element	**Indentation**
comment	none
function definition	none
label definition	none
assembly instruction	`\t`
after a mnemonic	`\t`
assembler directive	`\t`

Table 4.3: Indentation in assembly code created by lcc-win32. *Abbreviations:* `\t`*=tab; none=no indentation*

This section only briefly touches the lcc-win32 syntax. For more detailed information, please refer to [35].

4.2.2 Assembly Code Input

`mresproj.parsers.asm.ASMParser` is the main parser class for the assembly input files. The JavaCC-created files are not dealt with here. The parser complies with the EBNF grammar in Figure 4.6. There are several remarks to make about this grammar:

- **Non-terminals and tokens:** All the symbols used on the left-hand side are non-terminals except for `Digit`, `Letter`, `Newline`, and `Anything`, which are 'tokens'. They are mentioned here for completeness. Other symbols, like `HexNum`, `Number`, or `Instr` are both non-terminals and terminals (tokens) in the implementation, as this is necessary for the parser to function correctly.
- **New lines:** A new line is considered as part of the grammar; it is a 'symbol' that terminates an assembly line because this simplifies

look-ahead issues (see below). Since this optimiser works with output obtained from lcc-win32, which only runs on Microsoft Windows machines, a new line character is equivalent to \r\n here.

- **Indirect addressing:** `IndReg` describes the types of indirect and indexed addressing modes for registers that the parser accepts. Typical examples are e.g.:

```
(%edi,2)            (,%esi)
8(%ebp)             _var(%eax)
(%ebp,%esi)         (%ebp,%esi,4)
2(%ebp,%esi,4)      _var(%ebp,%esi,4)
```

This parser is a top-down, one token look-ahead parser that switches to a higher look-ahead if required. For example, a look-ahead of 2 is required at the choice point in `ASMFile`: After a `Newline` either an `ASMLine` can start, or at least one more `Newline` can follow, which indicates that the file is being finished.

However, the parser is not quite complete. Particularly the grammar for indexed/indirect registers requires a more precise description. The current grammar has been inferred from simple examples and the documentation only.

Disregarded assembly code

As introduced in section 4.1.2, a `NonASM` is an irrelevant line of assembly code to the optimiser – being either a comment, an assembler directive, or a 'method label'. Fortunately, these can be recognised easily by looking at the first few characters in a line. It can be concluded from the information in Tables 4.1 and 4.3 that in `NonASM`

```
'\t.'                          denotes the start of an
                               assembler directive;
';'                            denotes the start of a comment;
'_' Anything {Anything} ':'    denotes a method label.
```

After all these elements `{Anything}` is parsed, which matches any character sequence except for the new line characters \r or \n. This way, parsing comments etc. does not become an issue. Such input is not discarded, but stored in string format (for all strategies) so that it can be reassembled after optimisation. A `NonASM` can also be a special case of assembly code that shall simply be ignored (e.g. lines starting with `_$M`).

How are `NonASMs` and `ASMs` reassembled after optimisation so that they are printed in the correct order? First of all, *the* correct order can never be guaranteed in the optimised output, as during optimisation the assembly

```
ASMFile = ASMLine {Newline ASMLine} Newline {Newline};
ASMLine = NonASM | ASM;
NonASM = ('\t.' | '\tpushl\t__imp___iob' | '_$M' | ';'
         | '_' Anything {Anything} ':') {Anything};
ASM = ('\t' Instr ['\t' Arg {',' Arg}] | LabelDef);
Instr = Letter {Letter | Digit | '_' | '+'};
Arg = Label | Address | Register | Array | FuncOrVar |
      '$' (Constant | Label);
Register = IndReg | NormReg;
Constant = NumOrHex | FuncOrVar;
LabelDef = Label ':';
Label= BegLabel Number;
BegLabel = '_$';
Address = '(' (Number | NormReg) ')';
Array = FuncOrVar '(,' NormReg ',' NumOrHex ')';
IndReg = '(,' NormReg ')' | '(' NormReg ',' NumOrHex ')' |
         (NumOrHex | FuncOrVar) '(' NormReg ')' |
         [NumOrHex | FuncOrVar] '(' NormReg ',' NormReg
         (')' | ',' NumOrHex ')');
NormReg = '%' Instr;
FuncOrVar = '_' Instr;
NumOrHex = Number | HexNum;
Number = ['-'] Digit {Digit};
HexNum = '0' ('x' |'X') (Digit | 'a' | 'b' | 'c' | 'd' | 'e' |
         'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F')
         {Digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' |
         'B' | 'C' | 'D' | 'E' | 'F'};
Letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' |
         'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' |
         's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' |
         'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' |
         'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' |
         'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z';
Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
        '9';
Newline = '\r\n';
Anything =  -('\r' | '\n');
```

Figure 4.6: EBNF Grammar for input assembly files.

code changes. Another reason for this is that the current version of the system does not update assembler directives; nor does it track the semantics of each `NonASM` so that consistency between those and the actual assembly code can be assured. Still, during parsing, a line number is assigned to every `ASM` and `NonASM` (see Figure 4.7, left-hand side). If `ASMs` are replaced during optimisation, the new assembly lines adopt the line number of the first matched assembly code line that is furthest up the code. Thus, the three `ASM` lines with line number 6 in Figure 4.7 (right-hand side) are all replaced lines — so are the four lines with line number 9. In `printToFile` (in the main class `POpt`), the optimised output is created by following the line numbers sequentially in ascending order. This is indicated with the arrow in Figure 4.7.

This approach to reassembling irrelevant assembly code with the optimised one does not create any inconsistencies with a simple rules set like the one that has been used for testing in this dissertation.

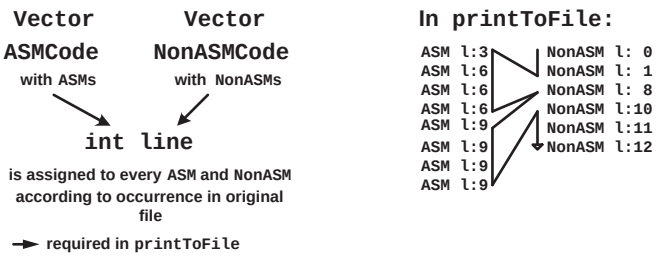

Figure 4.7: `POpt` reassembles `NonASMs` and `ASMs` by following their line numbers (originally assigned by `ASMParser`, but updated during optimisation in the strategy classes).

Production of the data structures

This parser not only accepts assembly code input, but it also creates suitable data structures, which depend on the chosen strategy. `ASMParser` differentiates between output for the regular expressions strategy and output for the generic strategies. Table 4.4 gives an overview of the data structures produced. There, it is evident that assembly code for the regular expressions strategy is kept in pure `String` format. For the generic strategies however, basic `String` elements are encapsulated in 'more meaningful' data structures like `Register`, `Constant` etc. Note that generic data structures are utilised on a very shallow level only; there are no further embedded object structures beyond the 'level' of mnemonics and arguments. Instead, rudimentary information is again stored in the form of `Strings`. This enables the generic strategies to perform type-level matching as well as more specific, string-based matching and replacements.

Non-Terminal	Regexp output	Generic output
NonASM	NonASM with a String	
ASM	RegExpASM: The whole assembly instruction as a String; line number	GenASM: Instruction or LabelDef; line number; *additional actions:* increment of asmCount; if the Instruction is a Jump → update labels table: labels[jumpTo]=add(asmCount)
Arg	String	Depending on what is parsed: Register, Label, Array, Function, or Constant
LabelDef	String	LabelDef with Label; *additional actions:* labels table is updated: labels[labelNo]=asmCount
Constant	String	Constant or Variable
All other	String	

Table 4.4: Production of data structures in ASMParser

The overall result which ASMParser returns to POpt is a collection of NonASMs and ASMs. In case of the generic strategies, a labels table is returned, too (introduced in section 3.3.1). It is initialised before the parser starts, and filled up during parsing. Whereas the line number information shown in Figure 4.7 keeps track of all the assembly code, asmCount mentioned in Table 4.4 only counts relevant assembly code to the optimiser. This is required for the labels table. Information in the labels table is updated in the generic strategies whenever a Jump or a LabelDef is encountered as described in section 3.3.1.

4.2.3 Optimisation Rules

mresproj.parsers.rules.RulesParser is the main parser class for rules set files. Its grammar (Figure 4.8) resembles that of ASMParser (Figure 4.6). This is also a top-down, one token look-ahead parser that switches to a higher look-ahead when required. The main differences between the two parsers lie in the following elements:

- Rule: This is the main sub-component in a rules file (analogous to ASMLine). Rules are separated from each other with at least one blank line (a new line). A '=>' acts as a separator between the matching and

```
RulesFile = Rule {Newline Rule};
Rule = {Newline} Match Newline '=>' Newline Replace;
Match = ASM {Newline ASM};
Replace = [ASM {Newline ASM}] Newline;
ASM = (Instr ['\t' Arg {',' Arg}] | LabelDef |
      Anything) ['|' Escape {Escape}];
Instr = Letter {Letter | Digit | Group };
Arg = Label | Register | Address | Group | Array | FuncOrVar |
      '$' (Constant | Label);
LabelDef = Label ':';
Label= BegLabel (Number | Group);
BegLabel = '_$';
Address = '(' (Number | Group | NormReg) ')';
Register = IndReg | NormReg;
Array = FuncOrVar '(,' NormReg ',' NumOrHex ')';
IndReg = '(,' NormReg ')' | '(' NormReg ',' NumOrHex ')' |
         (NumOrHex | FuncOrVar) '(' NormReg ')' |
         [NumOrHex | FuncOrVar] '(' NormReg ',' NormReg
         (')' | ',' NumOrHex ')');
NormReg = '%' Instr;
FuncOrVar = '_' (Instr | Group);
Constant = NumOrHex | FuncOrVar;
Escape = 'replc' '(' Arg ',' Arg ')';
HexNum = '0' ('x' |'X') (Digit | 'a' | 'b' | 'c' | 'd' | 'e' |
         'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F')
         {Digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' |
         'B' | 'C' | 'D' | 'E' | 'F'};
Letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' |
         'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' |
         's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' |
         'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' |
         'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' |
         'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z';
Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
        '9';
NumOrHex = Number | HexNum | Group;
Number = ['-'] Digit {Digit};
Newline = '\r\n';
Group = '&' Digit {Digit};
Anything = Group '*';
```

Figure 4.8: EBNF Grammar for rules.

the replacement parts in a `Rule`.

- `Match` and `Replace:` These components contain assembly code to be matched and replaced in the occurring order. As opposed to `Match`, `Replace` can be empty, which results in such a `Rule` causing all the matched assembly code lines to be deleted rather than replaced by others (as in Definition 3.4).

- `Escape:` This element can occur in both the matching part as well as in the replacement part of a `Rule`. It is basically a method call, consisting of the name of the subroutine to be called together with any variables. A '`|`' is used to indicate the start of an `Escape`.

- `Group:` This acts as a term pattern or variable within a `Rule`, which has to be instantiated with a concrete value during matching. `Groups` consist of a '`&`' followed by at least one digit. Digits should be chosen according to the occurrence of the `Group` in the rule starting from the matching part (first `Group=&1`, second `Group=&2` etc.). A `Group` matches the current input until the next pattern character in the rule, or until a new line is encountered. All `Groups` must have different instantiations, i.e. no two groups can match the same sequence of characters. Thus, a `Group` has to be used repeatedly in order to indicate the recurrence of a sequence of characters.

- `Anything:` This element exists in the assembly code grammar, too; but it has a completely different function here: An `Anything` is a special starred `Group` since it can be instantiated with several assembly lines. That is, it matches a string of arbitrary characters including new lines until the next pattern character. It is mainly used to copy a few lines of assembly code from the matching part to the replacement part without changes. An `Anything` can only occur as a single element in a line of a `Rule`, consisting of a '`&`' and digits, terminated by a '`*`'. The same conventions as in `Group` hold as to the distribution of digits. `Groups` and `Anythings` are treated identically in this regard. In this parser, `Special` and `Anything` are used synonymously; the only difference lying in the fact that `Anything` is the non-terminal that is being parsed, whereas `Special` is the data structure that is produced.

On completeness grounds, apart from pure non-terminals the grammar also contains symbols that are both non-terminals and terminals (`Group`, `Number`, `HexNum`, and `Escape`), and those that are terminals only (`BegLabel`, `Letter`, `Digit`, and `Newline`). Comments over one line are allowed anywhere in the rules set file, starting with a '`//`'.

Table 4.5 shows how groups should be employed to make elements match as intended.

Example	**matches** (group=match)
j&5 mov&1	the remaining part of the mnemonic; here, e.g.: &5=mp here, e.g.: &1=l, &1=b, &1=w, &1=q
%e&1	a register only in the direct addressing mode. The prefix '%e' restricts this item to match with registers. *Example matches:* &1=ax, &1=bx, &1=cx
_$&2: _$&2	a label definition since '_$' indicates the start of a label, however the succeeding ':' lets this group match with a label definition only (e.g.: &2=4). a label; here, e.g.: &2=3, &2=4
$&2	a constant, e.g. &2=5. Constants start with a '$' unlike labels.
&3	any element. This is produced as a Type in the parser so that it can be downcast to a specific Type (see Figure 4.5) during matching. However, this "pure" group can only match one single element, not multiple ones as e.g. '$4,%eax' (a constant and a register). Typical matches would be: &3=%ebx, &3=$2
| method(&4)	an escape with a variable. In this case, the escape method refers to an existing method in the Java implementation. The Group refers to a variable, e.g. &4=67.
&3(%e&4) _&5(,%e&6,&7)	a register in the indirect addressing mode; here, e.g.: &3=6, &4=cx an array; here, e.g.: &5=8, &6=si, &7=4
&1*	several assembly code lines separated by new line characters (=\r\n); e.g.: &1=subl $12,%esp\r\n push %ebx\r\npushl %esi\r\npushl %edi\r\n

Table 4.5: Example use of Groups with typical matches

The following rule demonstrates some of the issues above (the parts starting with '-->' do not belong to the rule; they indicate the structure only):

```
j&1     _$&2        -->  jump   label
&3*                 -->  special_group
_$&2:               -->  label_def
jmp     _$&4        -->  jump   label
=>
j&1     _$&4        -->  jump   label
&3*                 -->  special_group
_$&2:               -->  label_def
```

```
jmp     _$&4          -->  jump   label
```

Explanation: This rule consists of a matching and a replacement part of each 4 lines, separated by a '`=>`'. There are 4 groups/variables in this rule, of which number 3 is a special group (`Anything`). The purpose of this rule is to change the 'jump-to' of the first jump statement to avoid a multiple jump in the code (since the first instruction in the label definition `_$&2:` is another jump instruction). For this, the code after the first jump statement and that before the corresponding label definition is irrelevant – it just needs to be copied to the replacement part without changes, which is achieved by utilising `&3*`. All groups are reused in both parts of the rule, referring to the same sequence of matched characters.

The result of the parsing procedure in `RulesParser` is the whole set of parsed rules in an appropriate format. The next sub-section focuses on the format of these rules with respect to the different strategies.

Production of the data structures

Analogous to `ASMParser`, two different outputs can be generated in `RulesParser` — one for the regular expressions strategy and one for the generic strategies. Table 4.6 summarises the outputs of each non-terminal in the parser.

Despite the similarities between the production of outputs in both parsers of this work, the format of the rules in the lower levels is more distinctive and significant than that of the input assembly code, since this is eventually where pattern matching is employed with the appropriate strategy. The last row in Table 4.6 refers to the most basic elements of a `Rule`: In the *regular expressions strategy*, sensitive characters like '`$, (, ), *`' in the `Strings` have to be protected with backslashes for use with `java.util.regex`. `Groups` that have the format '`&1, &2`' etc. in the original rules file, are converted to '`\1, \2`' (or '`$1, $2`' for the replacement part) so that they are interpreted as *back-references* in the `java.util.regex` classes — referring to capturing groups. The following example shows how the rule for multiple jump eliminations (page 51) is converted for the regular expressions strategy in `RulesParser`:

```
j\1          _\$\2
\*3
_\$\2:
jmp          _\$\4
=>
j$1          _\$$4
$3
_\$$2:
jmp          _\$$4
```

Non-Terminal	Regexp output	Generic output
`ASM`	`RegExpASM:` The whole assembly instruction as a `String`	`GenASM: Instruction` or `LabelDef`
`Arg`	`String`	Depending on what is parsed: `Register`, `Label`, `Array`, `Function`, `Type`, or `Constant`
`LabelDef`	`String`	`LabelDef` with `Label`
`Constant`	`String`	`Constant` or `Variable`
`NumOrHex`, `Number`, `Anything`, `HexNum`, `Group`	`String`	
All other	`String`	`Vector:` Collection of 'pure' `Strings` and `Groups`; in `Arg` the `Vector` is embedded in the appropriate data structure

Table 4.6: Production of data structures in `RulesParser`

The use of '`$`' has to be considered carefully here: '`\$`' refers to the actual dollar symbol, whereas '`$`' in the replacement part indicates a back-reference to a group that occurred in the matching part. However, this syntax is not correct for use with the regular expressions strategy yet. In order not to complicate the parser, further modifications are dealt with in the regular expressions strategy itself. In `prepMatch`, a method that is called before the optimisation, the rules are processed further. The example above would look like the following after the final changes:

```
^j(.*?)      _\$(.*?)
((?s:.*?))
_\$\2:
jmp          _\$(.*?)$
=>
j$1          _\$$4
$3
_\$$2:
jmp          _\$$4
```

The `java.util.regex` classes require the first occurrence of a capturing group to be enclosed with parentheses. The regular expression '`.*?`' matches

any character zero or more times. However, being a reluctant quantifier, it tries to locate the first possible match in the input (until the next pattern character). New lines are not matched. '`(?s:.*?)`' though matches any character zero or more times including line terminators. Thus, the first occurrences of groups 1 to 4 are replaced with '`(.*?)`'in the matching part above; group 3 is replaced with `((?s:.*?))` (the second set of parentheses is for making `(?s:.*?)` a capturing group) as it is meant to match several assembly code lines. With this syntax, the regular expressions strategy can be applied straight away (see Chapter 3). The boundary matchers `^` and `$` enclose the matching part so that the exact match of the whole `String` is guaranteed.

For the *generic strategies*, the original syntax of the rules does not need to be altered. However, it is crucial how the individual chunks of strings are handled. As mentioned earlier, type-specific matching is achieved by means of the subclasses of `Type`. Here, string-based matching requires more effort than in the regular expressions strategy because it is done 'by hand' (as described in section 3.3.1). For every basic element of assembly code (which are: registers, arrays, constants, variables, labels, label definitions, mnemonics, and functions), the corresponding string representation is split around groups. This is demonstrated again with the example from above:

```
j&1     _$&2      -->  Jump: 'j' '&1'    Label: '_$' '&2'
&3*               -->  Special: '&3*'
_$&2:             -->  LabelDef: '_$' '&2' ':'
jmp     _$&4      -->  Jump: 'jmp'       Label: '_$' '&4'
=>
j&1     _$&4      -->  Jump: 'j' '&1'    Label: '_$' '&4'
&3*               -->  Special: '&3*'
_$&2:             -->  LabelDef: '_$' '&2' ':'
jmp     _$&4      -->  Jump: 'jmp'       Label: '_$' '&4'
```

The right-hand side shows how the left-hand side is divided up into smaller parts together with the data structure they are embedded in. This separation of 'data' and groups – assembled in `Vectors` – allows easier string-based matching in the generic strategies (see Algorithm 8 in Chapter 3).

Chapter 5

Comparison of Strategies

In this chapter, the features of the three strategies described in Chapter 3 are illustrated and compared to each other by means of concrete examples and test runs.

5.1 Efficiency

In this section, the regular expressions strategy (henceforth referred to as *'regexp'*), the generic strategy (*'generic'*), and the extended generic strategy (*'egeneric'*) are analysed with regard to the quality of their optimisations. Unless otherwise stated, the strategies are run with the default look-ahead of 4. The assembly code examples are also available on the CD attached to the dissertation. The rules utilised for these tests can be found in Appendix A (mostly taken from `rules.txt` on the CD), where also more examples are contained in addition to those presented in this section. Most of the rules only have a syntactical function – purely invented for demonstrating optimisations.

Let us begin with a simple example. Assuming our rules set consists of **Rule 1** only, which eliminates a line with an unnecessary comparison. The first column in Table 5.1 shows the assembly input (`ex1.asm` on the CD), whereas the second column shows optimisation with *regexp* and **Rule 1**. The result looks quite different if we add **Rule 2** to our rules set (column 3). As **Rule 2** is shorter than **Rule 1**, it is applied first, which disqualifies **Rule 1**. This is due to the backwards strategy. *Regexp* and *generic* yield the same result here, however *egeneric* applies **Rule 1** before **Rule 2**, optimising more (see `ex1_opt_eg.txt`).

Original Input	Regexp with rule 1	Regexp with rules 1 & 2
_$1:	_$1:	_$1:
movl $4,%ebx	movl $4,%ebx	movl $4,%ebx
movl $0,%eax	movl $0,%eax	xor %eax,%eax
cmp $0,%eax	jmp _$3	cmp $0,%eax
je _$3		je _$3
_$2:	_$2:	_$2:
movl %ecx,%ebx	movl %ecx,%ebx	movl %ecx,%ebx
_$3:	_$3:	_$3:
jmp _$1	jmp _$1	jmp _$1

Table 5.1: A simple optimisation example with two different rules sets

Let us consider a few look-ahead examples to demonstrate the advantage of the labels table for matching in *generic*. Table 5.2 shows a 'case 1' example as introduced in section 3.3.1 (`ex2.asm`). The semantics of the code in **Original Input** in between the first jump instruction and the corresponding label definition is insignificant. It is simply used to match with **Rule 3**, which is the only rule in the rules set for this test.

Original Input	Regexp	Generic
je _$2	je _$2	je _$3
movl %ecx,%ebx	movl %ecx,%ebx	movl %ecx,%ebx
idivl %ecx	idivl %ecx	idivl %ecx
movl %edx,%eax	movl %edx,%eax	movl %edx,%eax
movl %edx,%eax	movl %edx,%eax	movl %edx,%eax
movl $21,%ecx	movl $21,%ecx	movl $21,%ecx
_$2:	_$2:	_$2:
jmp _$3	jmp _$3	jmp _$3

Table 5.2: Look-ahead matching example 1

The look-ahead part is 5 lines long (between `je _$2` and `_$2:`, exclusively). This was chosen on purpose because *regexp* fails to optimise here with the default look-ahead. *Generic* however optimises regardless of the look-ahead chosen since it utilises the labels table only for matching. Needless to say that *regexp* yields the same result as *generic* with a higher look-ahead. *Egeneric* performs exactly like *generic* with such a small rule set.

Table 5.3 shows an example for 'case 2' (as defined in section 3.3.1) with another assembly code example (`ex3.asm`). We apply **Rule 4** to this. Again, *regexp* does not optimise at all if it is run with a look-ahead of 3 or smaller (as shown in Table 5.3), otherwise it yields the same result as *generic/egeneric*. For these latter two strategies, the chosen look-ahead is

Original Input	Regexp	Generic/Egeneric

`_$1:`
`je _$1`
`pushl -12(%ebp)`
`pushl -16(%ebp)`
`pushl $_$15`
`_$2:`
	`_$1:`
	`je _$1`
	`pushl -12(%ebp)`
	`pushl -16(%ebp)`
	`pushl $_$15`
	`_$2:`
		`_$1:`
		`jmp _$2`

Table 5.3: Look-ahead matching example 2

irrelevant because the labels table is consulted only.

The appendix (A.2) contains two more examples where the predictive look-ahead approach explicitly fails in *generic/egeneric*, and the line-by-line matching procedure is utilised instead. It is evident that if a change in the look-ahead settings affects the result when optimising with the generic strategies, it can be concluded that line-by-line matching took place, otherwise the predictive look-ahead matching procedure was applied.

Strategy (`-lo` for look-ahead)	**Reduction in code size in KB**
Generic/regexp egeneric `-lo 1` egeneric `-lo 7` until `-lo 11` egeneric `-lo 13` onwards	0.01
egeneric `-lo 2`, `-lo 6`	0.14
egeneric `-lo 3` until `-lo 5`	0.10
egeneric `-lo 12`	0.09

Table 5.4: Optimisation results for `switch.asm`

To demonstrate the strengths of *egeneric* over the other strategies, let us consider an example that originates from a 'real' piece of code with switch statements. As these are translated to jump instructions and label definitions during the compilation to assembly code, interesting optimisation opportunities arise. Concrete examples with excerpts of the code are shown in the appendix (A.2). The full code (`switch.asm`) is available on the CD, together with various optimisation outputs for different look-aheads. Table 5.4 summarises the results of these optimisations, grouping together identical or very similar outcomes. Whereas *generic* and *regexp* yield the poorest (identical) results, *egeneric* performs quite differently with varying look-aheads. As Table A.4 indicates, the look-ahead option controls three different issues in *egeneric*, which influence the maintenance of the `Option`

data structures and the choice points. Hence, different combinations of rules can arise, which lead to different results. Still, drawing general conclusions about look-ahead behaviour from the data above is difficult as the results vary according to the rules set and the assembly input chosen, too. However, it is evident that *egeneric* can perform far better than the other two strategies if run with the right look-ahead settings, which are 2 and 6 here. To achieve better results with the other strategies, one has to disable short interacting rules. Still, even with this approach the best performance of *egeneric* (code size reduction by 0.14 KB) cannot be reached, since this is a result of a combined application of interacting rules.

As mentioned before, a look-ahead of 1 causes *egeneric* to optimise like *generic*, and so do too high look-aheads (here from 13 onwards): as the distance between the artificial choice points grows, the 'rules cascading' feature of *egeneric* is suppressed. This does not affect rules with big matching parts as long as the look-ahead is sufficient for those. The appendix (A.2) contains an analysis of an excerpt of `switch.asm` with some of the look-aheads shown in Table 5.4.

5.2 Performance

In this section the strategies are analysed with regard to the quantity of their optimisations. That is, the amount of code reduced and the speed at which it is optimised is evaluated. Although the execution time is not a key concern in this work, it is interesting to observe how the different strategies perform with specific examples. The rules set `rules.txt` is utilised for all the test runs.

Reduction in code size is particularly interesting regarding *egeneric* with different look-ahead settings as this strategy has been designed for this purpose. Here, the examples have been tested with look-aheads between 1 and 12 for *egeneric*, and with the default look-ahead 4 for *generic* and *regexp*. It shall be noted that the rules set contains many short rules that do not affect the code size; i.e. these rules have match and replacement parts of the same sizes. Therefore in the results, optimisations with same code sizes are not necessarily identical. The following test files were used:

long This is a large piece of proprietary code from a commercial embedded application. Therefore it is not available on the CD.

switch This test file was introduced in section 5.1: it contains many jump instructions and label definitions, hence it provides a good example for demonstrating the strengths of the generic strategies.

All other test files were obtained from the latest distribution of lcc v4.2. They represent a good mixture of C source code using structures, various conditional statements, and different data structures such as arrays

(all files are available on the CD). Since the assembly parser is incomplete, lines that are currently not correctly recognised by `ASMParser` are commented out.

The test results are shown in Table A.5 in the appendix. The following can be interpreted from the data:

- Much to my surprise, *regexp* is always slower than the generic strategies. I expected the opposite due to the use of reflection for abstract type matching (as reported by [32]). *generic* performs up to 6 times better than *regexp* with the same look-ahead setting. Another test revealed that *regexp* spends about 70% of its time in the main matching routine `testRule`, in which the `java.util.regex` matching engine is used. So this is the main cause for the poor performance. *egeneric* is generally costlier than generic, which is an expected result.

- The more *egeneric* optimises, the more time it requires. The execution time with *egeneric* for huge assembly input such as `long` takes up to a minute. This indicates that *egeneric* is quite costly.

- The optimisation results with `switch` are very good (as discussed in 5.1), particularly for *egeneric* with the look-aheads 2 and 6. The rudimentary rules set had originally been designed to optimise opportunities in this file especially, so it is not applicable to all test files (positive examples are: `yacc`, `wf1`, `fields`, `front`, and `sort`). So this shows how important a good rules set is in addition to the correct choice of strategy and look-ahead.

- Due to the poor rules set, *egeneric* yields the best optimisation result (as for reduction in code size) for the files `switch`, `cf`, `stdarg`, and `wf1` only. A conclusion about the other strategies cannot be made in this point.

- Optimisation with `long` yields very interesting results: the generic strategies optimise about 21.6% regardless of the look-ahead setting. As mentioned in section 5.1, this indicates that the labels table was employed. Indeed, it is **Rule 10** in the appendix that achieves this: If one disables **Rule 10** in the rules set, the result is comparable to that of *regexp* as to the code size. This is a very positive result.

- Optimisation with `cf` yields a negative result concerning the code size reduction; i.e. it is increased and not reduced after optimisation. However, the line count does not change. *egeneric* yields better, 'neutral' results here with a look-ahead of 3 or greater by optimising less.

Chapter 6

Conclusions

This chapter concludes the results of this work. First, the achievements are summarised in section 6.1 . Then in section 6.2, the flaws of the implementation are described. Finally, section 6.3 suggests how the system can be improved and extended in future.

6.1 Achievements and Outcomes

The leading research question of this work formulated in Chapter 1 was: *how can pattern matching for peephole optimisation be performed in a more suitable and meaningful manner than by regular expressions? How would such strategies have to be designed?*

For this, I have implemented a classical peephole optimiser that takes as input assembly code produced by lcc-win32 and a rules set, returning as output the optimised code. To develop the strategies, I conceived the nature of pattern matching as strategies to rule application and pattern matching within a rule, which lead me to the following strategies:

- the *regular expressions strategy*: On comparison grounds, first this strategy has been implemented, following the classical approach with regular expressions (by means of the Java regexp engine in `java.util.regexp`) and the backwards strategy as in [23]. I also implemented a line-by-line look-ahead procedure for this strategy.

- the *generic strategy*: Apart from the backwards strategy as a rule application strategy, this strategy employs generic matching similarly to [32] by doing abstract type matching first, then moving on to more specific string-based matching by hand. I added a labels table that is particularly useful for look-ahead rules. The strategy can also switch to line-by-line matching as in the regular expressions strategy if required;

- the *extended generic strategy*: This strategy is identical to the generic

one, however it uses a cascading approach to the rules as a rule application strategy in order to implement optimisation for space.

The evaluation showed that the generic strategy generally optimises equally well but faster than the regular expressions strategy. It is also more efficient with look-ahead rules when the labels table can be applied. The regular expressions strategy is totally dependent on the look-ahead settings for rules that require one. So within this Java approach, the generic strategy is indeed a better alternative to the limited regular expressions one. However, the extended generic strategy performs best among all strategies provided that the correct look-ahead is discovered. It also became very obvious during the evaluation that the utilised rules set must be suitable to the assembly input to achieve good results with all strategies. Although the extended generic strategy is much costlier, its advantage pays off: editing the rules set as described in [10] becomes unnecessary. With the other strategies, the order of the rules has to be altered and rules have to be rewritten in order not to interact. This is a time-consuming and inaccurate approach as it must be done by hand, whereas experimenting with the extended generic strategy and different look-aheads is more effective.

6.2 Limitations

There were several difficulties in this project, which all trace back to my insufficient knowledge of lcc-win32's code generator and its full assembly syntax. As mentioned earlier, the rules set `rules.txt` used for the performance measurements (section 5.2) is semantically very insignificant and therefore results with compiler-produced assembly input files are generally not representative. It was very difficult to find such optimisation rules. I tackled this by inventing rules for specific examples to show the strategies work as intended. Due to this problem, I also omitted the implementation of rules with escapes. The assembly parser is also quite incomplete, as mentioned earlier. Unrecognised lines can be commented out though, which temporarily solves the problem.

6.3 Future work

This work has been a first attempt of its kind, mainly inspired by [32]. The issues mentioned in 6.2 should be the first to overcome in the future. Further suggestions for interesting extensions and improvements:

- **Labels table:** Since the labels table has proven to be useful, it should be utilised more. The current implementation of the look-ahead matching procedure using the labels table covers two cases only.

These should be extended and more should be added. Then, as generic matching is currently employed at level 1 only, 'deeper' generic matching by means of the labels table might help finding complex structures like chains of jump instructions and label definitions, or other constructions. Since this 'table' approach is a good way of accessing information quickly, another extension might consist of researching other constructs than jump instructions and label definitions for which a table might be useful. The aim here is to do more intelligent matching.

- **Extended generic strategy:** Basic improvements can be made concerning the maintenance of the options. For more challenging improvements first the look-ahead behaviour should be analysed closer with a good rules set and suitable test files. An extension might then attempt to automatically find the best look-ahead setting (or cascade) for the given assembly input before applying it. This would be a 'cascade of rule sequences'.

- **New strategies:** The analysis of pattern matching strategies does not end with the generic ones. Other rule application and pattern matching strategies might be explored: it would be particularly interesting to combine generic matching with AI methods as a next step.

Bibliography

[1] Aho, A. V., Ganapathi, M., Tjiang, S. W. K. (1989) *Code Generation Using Tree Matching and Dynamic Programming*. ACM TOPLAS 11(4):491-516.

[2] Aho, A. V., Sethi, R., Ullman, J. D. (1986) *Compilers: Principles, Techniques, and Tools*. Massachusetts: Addison-Wesley, pp. 554–558.

[3] Ancona, M. (1995) *An optimizing retargetable code generator*. Information and Software Technology 37(2):87–101.

[4] Davidson, J. W., Fraser, C. W. (1987) *Automatic Inference and Fast Interpretation of Peephole Optimization Rules*. Software - Practice & Experience 17(11):801-812.

[5] Davidson, J. W., Fraser, C. W. (1984) *Code selection through object code optimization*. ACM TOPLAS 6(4):505-526.

[6] Davidson, J. W., Fraser, C. W. (1984) *Register allocation and exhaustive peephole optimization*. Software - Practice & Experience 14(9):857-865.

[7] Davidson, J. W., Fraser, C. W. (1984) *Automatic generation of peephole optimizations*. CC84:111-116.

[8] Davidson, J. W., Fraser, C. W. (1982) *Eliminating redundant object code*. POPL82:128-132.

[9] Davidson, J. W., Fraser, C. W. (1980) *The design and application of a retargetable peephole optimizer*. ACM TOPLAS 2(2):191-202.

[10] Davidson, J. W., Whalley, D. B. (1989) *Quick compilers using peephole optimizations*. Software - Practice & Experience 19(1):195-203.

[11] Fraser, C. W. (1989) *A Language for Writing Code Generators*. Proceedings of the SIGPLAN '89 symposium on Compiler Construction, SIGPLAN Notices 24(7):238–245.

[12] Fraser, C. W. (1982) *copt, a simple, retargetable peephole optimizer*. Software, available at ftp://ftp.cs.princeton.edu/pub/lcc/contrib/copt.shar (up 24/02/2005).

[13] Fraser, C. W. (1979) *A compact, machine-independent peephole optimizer*. POPL'79:1-6.

[14] Fraser, C. W., Hanson, D. R. (1991) *A Retargetable Compiler for ANSI C*. SIGPLAN Notices 26(10):29–43.

[15] Fraser, C. W., Wendt, A. L. (1986) *Integrating Code Generation and Optimization*. Proceedings of the SIGPLAN'86 symposium on Compiler Construction, pp. 242–248.

[16] Ganapathi, M., Fischer, C. N. (1985) *Affix Grammar Driven Code Generation*. ACM TOPLAS 7(4):560–599.

[17] Ganapathi, M., Fischer, C. N. (1982) *Description-Driven Code Generation Using Attribute Grammars*. Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 108–119.

[18] Grune, D., Bal, H. E., Jacobs, C. J. H., Langendoen, K. H. (2000) *Modern Compiler Design*. NY: John Wiley & Sons, pp. 1, 371–375.

[19] Johnson, R. E., Mc Connell, C., Lake, J. M. (1991) *The RTL System: A Framework for Code Optimization*. In Code Generation – Concepts, Tools, Techniques, Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, ed. Robert Giegerich and Susan L. Graham, Springer-Verlag, 1992, pp. 255–274.

[20] Kessler, P. B. (1986) *Discovering machine-specific code improvements*. CC86:249-254.

[21] Kessler, R. R. (1984) *Peep - An architectural description driven peephole optimizer*. CC84:106-110.

[22] Knight, K. (1989) *Unification: a multidisciplinary survey*. ACM Comput. Surv., 21(1):93-124.

[23] Lamb, D. A. (1981) *Construction of a Peephole Optimizer*. Software - Practice & Experience 11(6):639–647.

[24] Liu, J., Myers, A. C. (2003) *JMatch: Iterable abstract pattern matching for Java*. Proceedings of the 5^{th} International Symposium on Practical Aspects of Declarative Languages.

[25] McKeeman, W. M. (1965) *Peephole optimization*. CACM 8(7):443-444.

[26] Morgan, R. (1998) *Building an Optimizing Compiler*. Digital Press, pp. 1–2.

[27] Navia, J. (1999-2005) *lcc-win32: A Compiler system for Windows*. Available at http://www.cs.virginia.edu/~lcc-win32/ (up 04/03/2005).

[28] Spinellis, D. (1999) *Declarative Peephole Optimization Using String Pattern Matching*. ACM SIGPLAN Notices 34(2):47–51.

[29] Tanenbaum, A. S., van Staveren, H., Stevenson, J. W. (1982) *Using Peephole Optimization on Intermediate Code*. ACM TOPLAS 4(1):21-36.

[30] Visser, E. et al. (2000-2005) *Stratego: Strategies for Program Transformation*. http://www.stratego-language.org/ (up 01/05/2005).

[31] Visser, E. (1999) *Strategic Pattern Matching*. RTA'99, Vol. 1631 of Lecture Notes in Computer Science, pp. 30–44.

[32] Visser, J., Lämmel, R. (2004) *Matching Objects*. Available at http://www.di.uminho.pt/~joost.visser/publications/MatchingObjects.pdf (up 12/02/2005).

[33] Warfield, J. W., Bauer, III, H. R. (1988) *An Expert System for a Retargetable Peephole Optimizer*. ACM SIGPLAN Notices 23(10):123–130.

[34] Watson, D. J. (1990) *High Level Languages and Their Compilers*. Boston: Addison-Wesley Longman Publishing Co., Inc, pp. 270–273.

[35] Webb, J. (2004) *Assembler with LCC-Win32*. Available at http://www.john.findlay1.btinternet.co.uk/asm/asm.htm (up 06/07/2005).

[36] Whitfield, D., Soffa, M. L. (1991) *Automatic Generation of Global Optimizers*. Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation.

Appendix A

Optimisations

This appendix contains the rules that are used in Chapter 5, more example optimisations, the options that `POpt` can be run with, and the results of the performance test which is commented on in section 5.2.

A.1 Rules

Some of the following rules that are utilised for the illustration of the strategies in Chapter 5 and in this appendix can be found in `rules.txt` on the CD.

Rule 1:

```
mov&1 $&2,&3
cmp $&2,&3
je _$&4
=>
mov&1 $&2,&3
jmp _$&4
```

Rule 2:

```
mov&1  $0,%e&2
=>
xor %e&2,%e&2
```

Rule 3:

```
j&1 _$&2
&3*
_$&2:
```

```
jmp _$&4
=>
j&1 _$&4
&3*
_$&2:
jmp _$&4
```

Rule 4:

```
_$&1:
&2*
_$&3:
=>
_$&1:
jmp _$&3
```

Rule 5:

```
j&1 _$&2
&3*
_$&4:
=>
jmp _$&4
```

Rule 6:

```
mov&1 %e&2,%e&3
&4*
mov&1 %e&2,%e&3
=>
mov&1 %e&2,%ebx
```

Rule 7:

```
movl $0,&1(%e&2)
=>
andl $0,&1(%e&2)
```

Rule 8:

```
je _$&1
jmp _$&2
_$&1:
```

```
=>
jne _$&2
_$&1:
```

Rule 9:

```
j&1 _$&2
movl $1,&3(%e&4)
jmp _$&5
_$&2:
movl $0,&3(%e&4)
_$&5:
movl &3(%e&4),%e&6
cmpl $0,%e&6
je _$&7
&8*
_$&9:
=>
j&1 _$&2
_$&5:
&8*
_$&2:
jmp _$&7
_$&9:
```

Rule 10:

```
j&1 _$&2
&3*
_$&4:
jmp _$&2
=>
jmp _$&4
```

A.2 More Examples for Section 5.1

In section 5.1 the two look-ahead cases that the generic strategies handle by means of the labels table were demonstrated with concrete examples. Here, two more cases are shown which the labels table fails to handle. The line-by-line look-ahead matching procedure optimises them instead.

Rule 5 addresses 'case 1' in section 3.3.1 where matching with the labels table fails because the match for `_$&2` is not available by the time the look-ahead part is being matched (in reversed order due to the backwards

Original Input	Regexp/Generic/Egeneric
`je _$1`	`jmp _$2`
`pushl -12(%ebp)`	
`pushl -16(%ebp)`	
`_$1:`	
`pushl $_$15`	
`_$2:`	
`jmp _$1`	`jmp _$1`

Table A.1: Look-ahead matching example 3

strategy). The example input shown in Table A.1 (`ex4.txt` on the CD) is therefore optimised with the non-predictive procedure instead (for all the strategies). This is evident because if the optimisation is run with a reduced look-ahead (≤ 3), no optimisation takes place.

If we append `jmp _$&2` to the end of the matching part of **Rule 5** to optimise the same example, the situation changes: since variable 2 is matched and therefore initialised before the look-ahead matching starts, the predictive look-ahead procedure can make use of the labels table here in the generic strategies. The whole input is replaced with `jmp _$2`. The result is always the same – regardless of the look-ahead settings chosen. *Regexp* of course requires a look-ahead of at least 4 as above to yield the same result.

To show a simple look-ahead example that does not involve jumps and labels definitions, Table A.2 demonstrates the result of applying **Rule 6** to the following input (`ex5.txt` on the CD):

Original Input	Regexp/Generic/Egeneric
`movl %eax,%ecx`	`movl %eax,%ebx`
`pushl -12(%ebp)`	
`pushl -16(%ebp)`	
`pushl $_$15`	
`pushl $_$15`	
`movl %eax,%ecx`	
`jmp _$1`	`jmp _$1`

Table A.2: Look-ahead matching example 4

Here, a look-ahead of 4 is required. The line-by-line matching procedure handles this for all strategies. The labels table would be useless here in *generic* and *egeneric* because there are no embracing label definitions or jump statements. As a future improvement, one might think about other constructions than labels and jump statements for which a more 'intelligent' form of matching would be useful.

In section 5.1, optimisation results for `switch.asm` with `rules.txt` for various look-ahead settings were discussed. Here it shall be demonstrated by means of an example why *generic* and *regexp* (also *egeneric* with a look-ahead of 1) fail to optimise well. Table A.3 shows a crucial excerpt where the long **Rule 9** can be applied. The applicability of this rule is not detected until the cursor reaches the last line of this excerpt (`_$10:`). However, in *generic* and *regexp* this rule is disqualified because **Rule 7** changes the input earlier (to `andl $0,-8(%ebp)` in line 5). In *egeneric* with look-aheads 2 and 4 this does not happen, since the total number of lines **Rule 7** eliminates (the 'match-replace difference' (MRD)) is 0, whereas for **Rule 9** it is 5. So **Rule 7** is not preferred.

egeneric run with a look-ahead of 2 applies two more rules to the result of that with look-ahead 4 – **Rule 3** and **Rule 10**. The intermediate steps are the following:

```
jle   _$8                jle   _$12                jmp   _$8
_$9:            Rule 3   _$9:           Rule 10
cmpl  $1,%ebx   ======>  cmpl  $1,%ebx  =======>
jne   _$5                jne   _$5
_$8:                     _$8:
jmp   _$12               jmp   _$12
_$10:                    _$10:                     _$10:
```

One cannot argue that the look-ahead of 4 is too big to detect the applicability of these two rules because optimisation with a look-ahead of 6 yields the same result as with look-ahead 2 – except that with look-ahead 6 a few opportunities for **Rule 2** are missed out. In fact, this has been observed for the look-aheads 4 and 5, too. Therefore, only the following can be concluded here: the higher the look-ahead is, the greater is the possibility that rules with a small MRD will be missed out. But remembering what *egeneric* has been designed for – optimisation for space – this is not an issue here.

Original Input	**Regexp/Generic**	**Egeneric** `-lo 4`	**Egeneric** `-lo 2`

`jle _$8`
`movl $1,-8(%ebp)`
`jmp _$9`
`_$8:`
`movl $0,-8(%ebp)`
`_$9:`
`movl -8(%ebp),%ebx`
`cmpl $0,%ebx`
`je _$12`
`cmpl $1,%ebx`
`je _$10`
`jmp _$5`
`_$10:`
	`jle _$8`
	`movl $1,-8(%ebp)`
	`jmp _$9`
	`_$8:`
	`andl $0,-8(%ebp)`
	`_$9:`
	`movl -8(%ebp),%ebx`
	`cmpl $0,%ebx`
	`je _$12`
	`cmpl $1,%ebx`
	
	`jne _$5`
	`_$10:`
		`jle _$8`
		`_$9:`
		`cmpl $1,%ebx`
		`jne _$5`
		`_$8:`
		`jmp _$12`
		
		`_$10:`
			`jmp _$8`
			
			`_$10:`

Table A.3: `switch.asm` and excerpts from different optimisations

Option	**Meaning**			
Syntax	General	RegExp	Generic	ExtGeneric
`-lo`	look-ahead	Maximum number of lines that is looked ahead for a rule that matches with input of variable size		In addition to the functionality in RegExp and Generic, this option determines the following: 1. the maximum number of rule application sequences that are maintained before applying the best one; 2. the maximum number of rules analysed for such an application sequence; 3. within a single optimisation sequence, the maximum range of assembly instructions analysed after every new matched rule
`-b`	back-references	Maximum number of variables/groups that occur in a rule		
`-la`	labels	–	Maximum number of labels that occur in the assembly input file; used for the size of the labels table	

Table A.4: The options in `POpt`

	8q	**array**	**cf**	**cvt**	**fields**	**front**	**incr**	**init**	**limits**	**long**	**sort**	**spill**	**stdarg**	**struct**	**switch**	**switch2**	**wf1**	**yacc**
r 4	132	172	122	296	164	154	146	148	108	8694	190	170	182	232	102	368	224	589
	0	0	-0.01	0	0.02	0.02	0	0	0	2	0.02	0	0	0	0.01	0	0.03	0.02
g 4	26	40	20	70	34	32	32	30	20	2429	40	46	40	52	20	82	48	148
	0	0	-0.01	0	0.02	0.02	0	0	0	141	0.02	0	0	0	0.01	0	0.03	0.05
e 1	46	74	34	90	54	42	44	44	26	51251	56	100	56	76	30	142	82	627
	0	0	-0.01	0	0.02	0.02	0	0	0	141	0.02	0	0	0	0.01	0	0.03	0.05
e 2	68	90	48	85	68	42	40	60	22	46063	60	110	72	92	42	192	120	741
	0	0	-0.01	0	0.02	0.02	0	0	0	141	0.02	0	0.01	0	0.14	0	0.03	0.05
e 3	88	128	50	86	62	40	42	80	20	45255	56	118	52	90	40	200	114	771
	0	0	0	0	0.02	0.02	0	0	0	141	0.02	0	0.01	0	0.10	0	0.04	0.05
e 4	90	160	50	90	52	44	42	62	22	51297	56	146	52	70	45	192	96	831
	0	0	0	0	0.02	0.02	0	0	0	141	0.02	0	0.01	0	0.10	0	0.04	0.05
e 5	90	169	38	82	56	40	40	40	26	43779	52	122	52	68	45	120	76	811
	0	0	0	0	0.02	0.02	0	0	0	141	0.02	0	0.01	0	0.10	0	0.04	0.05
e 6	78	154	36	84	50	40	40	44	24	45866	58	104	50	68	60	132	80	875
	0	0	0	0	0.02	0.02	0	0	0	141	0.02	0	0.01	0	0.14	0	0.04	0.05
e 7	46	140	34	86	50	48	40	40	26	40898	54	66	54	66	38	130	94	997
	0	0	0	0	0.02	0.02	0	0	0	140	0.02	0	0.01	0	0.01	0	0.04	0.05
e 8	48	116	38	86	52	44	44	46	24	41670	56	70	56	66	34	140	76	997
	0	0	0	0	0.02	0.02	0	0	0	141	0.02	0	0.01	0	0.01	0	0.02	0.05
e 9	50	70	40	88	50	42	40	46	24	41089	58	70	60	66	32	136	76	919
	0	0	0	0	0.02	0.02	0	0	0	141	0.02	0	0.01	0	0.01	0	0.02	0.05
e 10	52	64	42	84	50	44	42	42	28	46106	56	70	56	66	32	116	78	837
	0	0	0	0	0.02	0.02	0	0	0	141	0.02	0	0.01	0	0.01	0	0.02	0.04
e 11	50	68	44	82	58	46	42	46	26	46877	54	70	53	62	36	138	76	759
	0	0	0	0	0.02	0.02	0	0	0	141	0.02	0	0.01	0	0.01	0	0.02	0.04
e 12	44	72	32	81	54	44	40	50	24	59051	54	72	60	64	90	142	82	670
	0	0	0	0	0.02	0.02	0	0	0	141	0.02	0	0.01	0	0.09	0	0.04	0.04

Table A.5: Average performance measurements for the regular expression strategy with the default lookahead (**r 4**), the generic strategy (**g 4**), and the extended generic strategy for look-aheads between 1 and 12 (**e 1-12**) with different test files. The upper number in a cell indicates the run time of the optimisation in milliseconds, whereas the lower number refers to the reduction of code size in KB.