OORS: An Object-Oriented Rewrite System

Gernot Gebhard and Philipp Lucas

Dept. 6.2 Computer Science, Compiler Design Lab,
Campus E1 3, Saarland University,
D-66041 Saarbrücken, Germany
`[gebhard|phlucas]@cs.uni-sb.de`,
Website: http://rw4.cs.uni-sb.de/ gebhard/projects/oors/

Abstract. Retargeting a compiler's back end to a new architecture is a time-consuming process. This becomes an evident problem in the area of programmable graphics hardware (graphics processing units, *GPUs*) or embedded processors, where architectural changes are faster than elsewhere. We propose the *object-oriented rewrite system OORS* to overcome this problem. Using the OORS language, a compiler developer can express the code generation and optimization phase in terms of cost-annotated rewrite rules supporting complex non-linear matching and replacing patterns. *Retargetability* is achieved by organizing rules into profiles, one for each supported target architecture. Featuring a rule and profile inheritance mechanism, OORS makes the reuse of existing specifications possible. This is an improvement regarding traditional approaches. Altogether OORS increases the maintainability of the compiler's back end and thus both decreases the complexity and reduces the effort of the retargeting process. To show the potential of this approach, we have implemented a code generation and a code optimization pattern matcher supporting different target architectures using the OORS language and introduced them in a compiler of a programming language for CPUs and GPUs.

1. Introduction

As the number of different hardware architectures is steadily growing, easily retargetable compilers are most valuable. Amongst others, this applies to graphics processing units (GPUs). For the last few years the performance of GPUs has been increasing at a much faster rate than that of general-purpose processors and now exceeds the peak performance of high-end CPUs. The amount of transistors on graphics chips has been growing by a factor of 32 every two years [3]; 16 times faster than the CPU transistor count growth that Moore's Law predicts. Thus, GPUs have become more

Gernot Gebhard and Philipp Lucas

and more interesting for general-purpose programming [19]. Several high-level languages, such as Brook for GPUs [4] or CGIS [17, 14], have emerged to exploit the vast computational power that GPUs have to offer. Easily retargetable compilers for these languages are necessary, because new GPU architectures are released at a fast rate (e. g., NVIDIA's NV40 in 2004, G70 in 2005, and G80 in 2006). Naturally, the same applies to compilers supporting embedded systems, where a wide variety of different architectures with common heritage exists (e. g., Freescale's MPC555, MPC565, MPC755 and derivatives).

To decrease the complexity of the retargeting process and to keep the compiler maintainable in the long run, we propose the *object-oriented rewrite system OORS*. The idea of OORS originated from the fact that new architectures share many features with their predecessors, but offer an extended instruction set (apart from other new features). If a new instruction set architecture is released with minor differences to an already supported one, only small changes to an existing back end are required. So, the key feature of our approach is to be found in the *reusability* of existing specifications. We realized this by introducing object-oriented language features in the yacc-like OORS language. Easily making the reuse of existing specifications possible, the proposed language enables a compiler developer to implement an OORS code generator or code optimizer which is less complex than a hand-written one. Consequently, the OORS implementation is much easier to read and to maintain in the long run. Our experiences in introducing new GPU architectures (NV40/G80) in the CGIS compiler supports this claim.

Basically, an OORS specification describes a pattern matcher that translates attributed input strings into attributed output strings. Given a set of rewrite rules, the pattern matcher processes the input string as follows. First the pattern matcher tries to match the possibly non-contiguous pattern of each rewrite rule against the input string. If multiple rules are applicable, the pattern matcher computes the non-constant costs of each rule to determine the rule to apply. Finally, the selected rule emits an attributed string that is appended to the output string. After consuming the whole input string, the pattern matcher terminates.

The aim of this paper is to introduce the OORS language, its syntax and semantics and to demonstrate its applicability in real word applications. Additionally, this paper briefly introduces the pattern matcher generator OORG that compiles an OORS specification into a C^{++} *dynamically retargetable* pattern matcher. We show its applicability by means of a real-life compiler for GPUs and SIMD CPUs.

The remainder of this paper (an extended version of [15]) is structured as follows: Section 2 discusses related work. The object-oriented rewrite

system OORS is introduced in Section 3. This section covers both the OORS language and the matching process. Section 4 introduces the pattern matcher generator OORG and demonstrates the integration of an OORS pattern matcher in a compiler. Section 5 concludes this paper and discusses future work. Finally, the appendix found at the end of this paper presents some compiler-unrelated applications of OORS.

2. Related Work

Numerous other approaches have been suggested for code generator generators. Below we discuss related work.

Emmelmann et al. [9] propose BEG, a generator for efficient back ends. Using the description language BEGL, the developer implements tree-pattern matchers for code generation in terms of cost-annotated rewrite rules. In contrast to the OORS language, BEGL offers no rule-inheritance mechanism. The reuse of existing specifications is thus not possible. Additionally, a BEG pattern matcher is dedicated to a single target architecture only. The major difference is to be found in the processed input data. BEG code generators process *trees*, whereas OORS pattern matchers operate on instruction *sequences*.

In [13], Fraser et al. introduce the code generator BURG for the bottom-up rewrite system BURS which is similar to BEG. BURG is able to generate tree-pattern matchers for fast optimal instruction selection. A BURG-generated tree-parser is able to find an optimal parse of an input tree in linear time. As in BEGL, the BURG grammar does not feature mechanisms that make the reuse of existing specification possible. Additionally, BURS code generators are only able to generate code for a single architecture. BURG-generated code selectors are used in the ANSI C compiler lcc [12].

Ferdinand et al. [11] solve the code selection problem with deterministic finite tree automata that are generated automatically from regular tree grammars. In contrast to BURG, the left-hand and the right-hand side of rules are not limited to leafs or nodes with one or two child nodes. In contrast to OORS, the costs of a rule must be constant and thus cannot depend on the matched instructions. Similar to BURG, the developer cannot inherit rewrite rules from each other to easily copy reusable properties. As in the other approaches, it is not possible to target multiple architectures.

In [1], Alt et al. propose the CoSy model, which provides a framework for flexible combination and embedding of compiler phases to ease the construction of parallel and optimizing compilers. Using three different languages, the compiler developer can implement the different phases of a compiler on a high level of abstraction. Additionally, the developer is able to

specify the control flow and the interactions of the compiler phases. Existing implementations of compiler phases can be simply reused. However, if any modifications to the implementation are required, the framework requires copy-and-pasting of that implementation before the developer can change the code. Just as in the approaches discussed above, the code generator is based on tree-pattern matching. A comparison with OORS is hardly possible, because OORS is designed to only implement the code generation and code optimization phase.

Dias and Ramsey [8] propose a recognizer for machine-independent code selection and code optimization. A recognizer is generated automatically from a declarative machine description that describes properties of the target platform. The generated recognizer requires the compiler to represent intermediate code in the form of machine-independent register-transfer lists (RTLs) [7]. By means of a declarative machine description, the recognizer tries to generate better RTLs. The recognizer will continue until no more optimizations can be applied. The recognizer omits a previously generated RTL, if the new RTL cannot be implemented on the target platform according to the machine description. The authors have successfully generated and tested a recognizer for the x86 back end in the Quick C-- compiler [5]. This approach differs greatly from OORS pattern matchers, as the developer does not have to explicitly implement the code optimizer. The effort is shifted to implementing a complete machine description.

Farfeleder et. al [10] describe a similar approach. By means of a new architecture description language (ADL), the authors are able to derive an optimized tree-pattern matching instruction generator, a register allocator and an instruction scheduler. To demonstrate the applicability of the new ADL, the authors have implemented an ADL-generated compiler for the xD-SPcore digital signal processor. Again, the effort is shifted to implementing a complete machine description.

In [16], Lerner et al. introduce Rhodium, which is a new language for compiler optimizations, whose soundness can be proven automatically. The developer specifies optimizations in terms of transformation rules that are automatically proven to be semantics-preserving. Rhodium optimizations are not bound to a specific target architecture, because they process input programs transformed into a C-like intermediate language. In this way, the optimizations are automatically retargetable. However, the main goal of their approach is the automated soundness proofs of the compiler optimizations. Rhodium optimizations do not directly compete with OORS optimizations, because OORS operates on the instruction level. Apart from that, the approach is out of scope for this paper, because OORS was not designed for providing automated soundness proofs.

The following approaches provide methods to implement transformations on the source code level and are thus not directly comparable but nonetheless related to OORS.

The Turing eXtender Language TXL [6] is a powerful, special-purpose programming language originally designed for rapidly prototyping of language dialects. It is, however, a quite general source code transformation system. A transformational grammar is interpreted in functional style, consuming a source program and therewhile constructing a transformed source program (possibly in another language). This usage in source-to-source transformations and the basic consumption model form the fundamental differences to OORS. TXL features *grammar overrides*, which offer an extension mechanism similar to OORS' inheritances.

Larger program transformation frameworks such as the ASF+SDF Meta-Environment [22, 23] or Stratego/XT [2] offer a more general set of capabilities than OORS. OORS does not directly compete with these approaches, instead opting for providing a smaller and more specific transformational description.

In [24], Warth and Piumarta propose OMeta, a new object-oriented language for pattern matching. The main purpose of OMeta is to provide developers with a convenient way of implementing tokenizers, parsers, and tree transformers, all of which can be extended using object-oriented mechanisms. Apart from the object-oriented language aspects, OMeta also allows processing of arbitrary data and not just streams of characters. The main difference to OORS is that OMeta rules describe transformations of tree patterns instead of list patterns.

OORS differs in many ways from the approaches presented above. One difference concerns the way in which the *subjects of pattern matching* are represented: OORS operates on sequences of instructions, not on trees. This is because OORS is also employed in the code optimization phase. By representing the subject of matching as an instruction sequence, scheduling properties can be expressed alongside with other low-level optimizations (see Section 3.3). Another important point is that OORS features object-oriented language constructs that make the *reuse* of existing specifications easily possible. Apart from CoSy [1] and Stratego/XT [2], none of the presented approaches was designed with reusability in mind.

3. Object-Oriented Rewrite System

In this section, we introduce the key concepts of the OORS language and discuss the pattern matching process. For the sake of simplicity, we mainly

concentrate on code *generation*. Section 3.3 discusses the changes to the matching process required to realize code *optimization*.

3.1. Rules

As mentioned in Section 1, an OORS pattern matcher processes attributed strings, which are sequences of instructions. We assume that each instruction is an instance of a class of the compiler's internal representation (e. g., a binary instruction could be an instance of the *BinaryInstruction* class) with a common base class (e. g., *Instruction*). The available attributes of each instruction object (operands, modifiers, etc.) are then defined by the member functions of the corresponding class. So to speak, OORS rewrite rules determine transformations on sequences of instruction objects. The behavior of each rule is determined by the following four aspects:

search pattern: The search pattern determines constraints on the structure of the input that must be fulfilled before the pattern matcher may apply the rule. A search pattern is a non-empty, ordered or unordered, possibly discontinuous sequence over instruction classes (*item patterns*). By using *wildcard patterns*, the developer can specify search pattern with an arbitrary lookahead.

Each symbol of the search pattern may be guarded by a local side condition, which is simply a boolean expression over the instructions and their attributes. Using local side conditions, the developer is able to specify non-linear search patterns. For instance, a local side condition could check whether an operand of the currently matched instruction and the target of a previously matched instruction are of the same type (see Example 3).

condition: The condition corresponds to the local side conditions introduced above, but it can also check global properties. Syntactically, the main difference to the local conditions is that the developer implements a boolean function instead of a single boolean expression. An undefined condition function is assumed to return *true*. Condition functions come in handy when deriving rules from each other (see Section 3.2).

costs: The cost function associates a weight to each matched instance of the search pattern. The computed costs need not be constant and may depend on the matched instruction objects. If multiple rules match the input sequence, the pattern matcher determines the rule to apply according to the associated costs.

replace pattern: The replace pattern determines the generated instruction sequence that the pattern matcher appends to the output sequence

when applying the rule. Each element of the replace pattern corresponds to a constructor call of the instruction class with appropriate arguments. It is possible to access previously generated instructions when initializing a new instruction object.

Simple Rules Example 1 demonstrates how to specify a simple code generation rule that compiles a binary instruction for which the target architecture has a direct counterpart. The rule matches any binary instruction object. Thus, there is no need to implement a rule for every single binary instruction. This keeps the specification both readable and maintainable in the long run.

Example 1 (Rule that compiles any abstract binary instruction into its counterpart). Matched instructions objects can be accessed via $$ and $i-operators. The $i-operators enumerate both the matched and the generated operations uniformly and in their static order: The BinOp of the search pattern is $1 and the generated GPUBinOp is $2.

```
rule binary {
  search:  [ BinOp ]
  cost:    { return 1; }
  replace: [ GPUBinOp($1->opcode,$1->tgt,$1->op1,$1->op2) ]
}
```

Figure 1 illustrates the effect on the input and output instruction sequence after applying the rule binary. Note that the rule matches any binary instruction and not only a substraction instruction as shown in the figure.

Generic rules like the one shown in Example 1 are not always sufficient: Some instructions may require special rules. Example 2 shows how to compile the exponentiation operator for recent GPU architectures.

Example 2 (Code generation rule for a special unary operator). Instruction sets of recent GPU architectures do not feature an exponentiation operator. Instead, their instruction set contains the EX2 instruction, which computes powers of two. By using the identity $e^x = 2^{x/\ln(2)}$ (because $2^x = (e^{\ln(2)})^x$), we are able to compute e^x on the GPU using the rule exp.

```
rule exp {
  search:  [ UnOp($$->opcode == OP_EXP) ]
  cost:    { return 2; }
  replace: [ GPUBinOp(OP_MUL, SymReg(TYPE_FLOAT), $1->op,
                      Const(TYPE_FLOAT, 1/ln(2))),
             GPUUnOp(OP_EX2, $1->tgt, $2->tgt) ]
}
```

In contrast to the *absolute* $i references, $$ always refers to the *currently processed* instruction object, just like in yacc and related tools. Thus, the

search pattern specifies a local condition on the `UnOp`-object. Note that in the replacement pattern it is necessary to access both the matched instruction ($1) and the first generated instruction ($2). The target of $2, the new temporary value, is used as an operand to the final exponentiation operation.
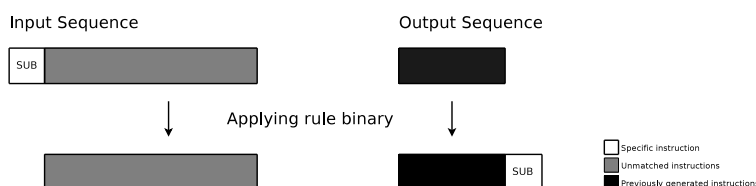
Input Sequence   Output Sequence

Applying rule binary

Specific instruction
Unmatched instructions
Previously generated instructions

Fig. 1: Input and output instruction sequence before and after applying the rule `binary`.

Complex Rules In some cases, the developer might want to match instructions objects that are not necessarily adjacent to each other in the input instruction sequence. Example 3 demonstrates a typical case.

Example 3 (Complex code generation rule). GPU architectures feature a combined sine-cosine instruction SCS [18]. From a single operand c, the instruction writes $\sin c$ and $\cos c$ into two register components. The following rule combines sine and cosine instructions in the intermediate representation into a single SCS operation. The *wildcard pattern* ($*$-pattern) denotes that the two instructions need not be adjacent to each other.[1] The curly braces in the search pattern indicate an *unordered* sequence: The instructions may be matched in any order.

```
rule scs {
   search:     { UnOp($$->opcode == OP_COS),
                 *,
                 UnOp($$->opcode == OP_SIN) }
   condition: { return $1->op == $3->op; }
   cost:      { return 1; }
   replace:    [ GPUUnOp(OP_SCS, SymReg($1->tgt, $3->tgt),
                         $1->op) ]
}
```

---

[1] In this case, $2 would refer not to a single instruction but to the *length* of the matched sequence; see Example 8 for an application for this.

Figure 2 illustrates the effect on the input and output instruction sequence after applying the rule `scs`. The figure demonstrates that the pattern matcher must not apply the rule in every case. For instance, if an instruction in-between the matched instructions modifies the operand or the target of the second matched instruction, the transformation will most likely modify the semantics of the input program and is thus invalid in general. Note that every rule with at least one wildcard pattern in its search pattern is subject to this negative side-effect. Thus, a special, semantics-preserving check is required.
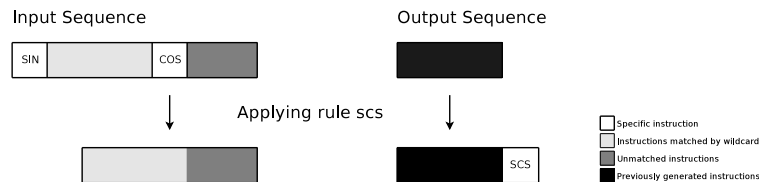

Fig. 2: Input and output instruction sequence before and after applying the rule `scs`.

The OORS language enables the developer to implement such a semantics-preserving check. The developer implements a *global implicit condition*, which decides whether the rule in question may be applied. This implicit condition is only checked for rules whose search pattern contains a wildcard pattern. This function checks for data dependencies between the matched instructions, which would prevent a reordering. For example, in rule `scs`, the function has to check whether the instruction sequence matched by the wild-card pattern writes the target or operand of the following instruction (write-write or write-read dependency) or uses its target (read-write dependency). Additionally, for architectures with guarded executions, it has to be checked that no instruction matched in the wildcard pattern writes to a guard register.

Under certain circumstances, it might not be sufficient to generate the same sequence of instructions all the time. Some instructions might have to be translated into different instruction sequences depending on the type of their operators or similar side conditions. For this reason, the OORS language allows the developer to guard any sequence of instructions to generate via `if-then-else` statements. This enables the developer to integrate all possible alternatives into a single replace pattern, which keeps the pattern matcher specification readable. Example 4 shows the usage of guards within replace patterns.

Gernot Gebhard and Philipp Lucas

Example 4 (Rule generating different instruction sequences for the same instruction type). On GPU architectures, each register is a float vector comprising four components (called r g b a). In the source language, and thus in the intermediate instructions, operations work on scalars or on vectors with a length of at most 4; in general, the native arithmetic instructions also support such vectorial operations. However, some instructions operate only on scalar operands. Thus, special treatment is required if an operand of the correlative abstract instruction is of vector type. To resolve this problem, a sequence of the same scalar instructions has to be generated for each vector component. The rule sin generates code for the SIN-instruction, which computes the sine of its operand. When generating the corresponding GPU code, the rule has to make sure to select the correct vector components (e. g., *cmp('r')* directs the instruction object to read from and write to the r-component).[2]

```
rule sin {
  search:  [ UnOp($$->opcode == OP_SIN) ]
  cost:    { return $1->tgt->components; }
  replace: [ if ($1->useComponent('r')) [
               GPUUnOp($1->opcode, $1->tgt, $1->op, cmp('r'))
             ],
             if ($1->useComponent('g')) [
               GPUUnOp($1->opcode, $1->tgt, $1->op, cmp('g'))
             ],
             if ($1->useComponent('b')) [
               GPUUnOp($1->opcode, $1->tgt, $1->op, cmp('b'))
             ],
             if ($1->useComponent('a')) [
               GPUUnOp($1->opcode, $1->tgt, $1->op, cmp('a'))
             ] ]
}
```

To further improve the maintainability and compositionality of the pattern matcher specifications, the OORS language introduces the notion of *intermediate replace patterns*. The developer specifies a replace pattern not on the instruction set of the target architecture, but on the intermediate instructions. This kind of rule is specified by the keyword intermediate instead of replace. These intermediate instructions are then subject to the matching process as usual. The benefit of the intermediate instructions is to allow *compositional* specifications. For example, a vectorial tan instruction has to be implemented by sequences of native sin and cos instructions just like in Example 4, followed by a division; a much more complicated way than just specifying an *intermediate level* sin-cos-div sequence and letting the

---

[2] The components need not be consecutive.

matcher then generate the component-wise native instructions from these intermediate instructions, as shown in Example 5.

Example 5 (Rule that compiles instructions by creating an intermediate basic block). Using the immediate keyword the developer expresses that a matched sequence of instructions is to be treated as the sequence specified in the immediate pattern. In this fashion, the developer is able to implement complex code generation or optimization steps on a higher level of abstraction.

```
rule tan {
    search:       [ UnOp($$->opcode == OP_TAN) ]
    cost:         { return $1->tgt->components; }
    intermediate: [ UnOp(OP_SIN, TmpVar($1->op->type), $1->op),
                    UnOp(OP_COS, TmpVar($1->op->type), $1->op),
                    BinOp(OP_DIV, $1->tgt, $2->tgt, $3->tgt) ]
}
```

3.2. Rule Sets

An OORS pattern matcher specification comprises rules, which are organized into *profiles.* A profile represents a set of rules dedicated to a specific target architecture. During runtime, the host compiler (the compiler using the generated code-generator) selects the corresponding profile to be used for processing the input program(s). In this section, we first describe the matching process with respect to a single profile. Afterwards, we discuss the specifications mechanism concerning multiple profiles and their relationships.

Matching For a given profile, the pattern matcher tries to match the input instruction sequence against the search patterns until every input symbol has been covered. In a *greedy* matching mode, the input is processed left-to-right, where always the rule with the lowest costs is selected. In case of a tie, the first specified rule is chosen. The pattern matcher employs backtracking in case the current match cannot be enlarged further while some input instructions remain unmatched.

In contrast to the greedy matching, an *optimal* matching mode investigates all possible matches to select the one with the globally optimal costs. If costs are not negative, we do not need to exhaustively explore the search space, but can prune the search space as soon as it can be determined that the current, incomplete match cannot outpace a previously found match.

Independent of the used matching mode, the pattern matcher only generates the target instruction sequence after a complete match of the input

stream has been found. In such a case, the pattern matcher sequentially applies the matched rules by first generating the instruction objects specified in the replace patterns and then deleting the matched instruction objects.

Inheritance To keep an OORS pattern matcher specification readable and thus maintainable in the long run, the OORS language comprises rule and profile *inheritance* mechanisms. The developer can derive a new profile from existing ones. The new profile inherits all rules, may add new rules and may omit and modify inherited rules. Example 6 demonstrates the profile and rule inheritance mechanism.

Example 6 (Profile and rule inheritance). The source language features a dot-product operating on vector operands. The target architectures support the dot-product for vectors of three and four components. The newer G80 architecture supports also the dot-product for vectors of two components, whereas before it had to be realized using multiplication and addition. Thus, the NV40 profile comprises two distinct rules to cover all kinds of operands.

```
profile NV40 {
  rule dp2 {
    search:    [ BinOp($$->opcode == OP_DP) ]
    condition: { return $1->tgt->components == 2; }
    cost:      { return 2; }
    replace:   [ /* MUL, ADD */ ]
  }

  rule dp : extends dp2 {
    condition: { return $1->tgt->components > 2; }
    cost:      { return 1; }
    replace:   [ GPUBinOp(OP_DP, $1->tgt, $1->op1, $1->op2) ]
  }
}

profile G80 : extends NV40 {
  omit NV40::dp, NV40::dp2;
  rule dp : extends NV40::dp {
    condition: { return $1->tgt->components > 1; }
  }
}
```

The G80 profile is specified as an extension of the NV40 profile. So, it inherits per default all NV40 rules. The G80 profile omits the two NV40 rules dp and dp2 and specifies a new, general rule as a modification of the inherited dp-rule.

However, when deriving rules from each other, the developer has to keep certain constraints in mind. Example 7 demonstrates two common pitfalls that might occur.

Example 7 (Two common rule inheritance pitfalls). In this example it is assumed that instances of the classes `A` and `B` may occur in any input instruction stream, whereas instances of the classes `C` and `D` may appear in the generated instruction stream. Each class, except `A`, is assumed to implement the function *check*, which takes no arguments and returns a boolean value.

Finally, it is assumed that the developer has implement the rule `base` as follows:

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

The rule `base` matches the sequence `AB` and translates that sequence into the target sequence `CD`. Apart from that, the rule is *virtual*, because the cost function is not defined. Thus, the rule `base` is not used during runtime. Instead of redefining that rule, the developer specifies the following other rules:

– First, the user derives the rule `first`, which features a condition and cost function. In contrast to the rule `base`, the rule `first` only accepts those input sequences `AB`, where the function *check* of the matched object `B` returns *true*:

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

As every instance of the class `B` implements the function *check*, it is valid to derive the rule `first` from the rule `base` in this fashion.
– Furthermore, the user derives the rule `second` from the rule `first` and replaces the inherited search pattern as follows:

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

However, the rule specification is not valid, because the inherited condition function accesses a second matched object of the search pattern, which only matches one object of the input stream. The developer must override the rule's condition function to make the rule specification valid. This kind of error can be detected statically.
– Instead, the user modifies the rule `second`, such that the search pattern now matches two instances of the class `A`:

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

On the first look, this rule definition appears to be valid, as the rule's condition is now able to access the second matched object. However, this rule specification is also invalid, because the class `A` does not implement the function *check*. This kind of error can also be checked statically by the type checking during the compilation of the generated code. Again, the developer must reimplement the rule's condition to make the specification valid.

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

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

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

Because the rule third derives from a valid rule and its replace pattern is also valid, there is nothing wrong with that rule. The interesting question is now, which object instance is passed to the constructor of the class `D`, when the pattern matcher applies the rule `fourth`. According to the semantics of the $-operator, one would expect that the second instance of the class `B`, is passed to the constructor. If so, the new search pattern would implicitly modify the inherited replace pattern, which contradicts the common notion of inheritance[3]. However, as it is known that `$3` has been specified in a different context, it is statically possible to associate the pattern access with the correct object instance. So, if the developer overrides the search pattern, the inherited replace pattern needs not be reimplemented, if the replace pattern contains inter-pattern accesses.

The rule inheritance mechanism is very powerful, enabling the developer to specify the behavior of an OORS pattern matcher on a very high level of abstraction. The developer no longer has to cope with the actual matching. Instead, the user simply has to identify the instruction sequence patterns the pattern matcher should replace.

---

[3] This means that an inherited property remains unmodified unless it has been explicitly overridden.

3.3. Optimization

The pattern matching approach can also be used for code optimization. In contrast to the generational mode discussed in Section 3.2, both matching and replacement are performed on a single sequence. The pattern matcher searches for instances of the search patterns and literally replaces matched instructions with the generated instructions. This process is repeated until no more rules can be applied, that is, until a fixpoint has been reached. Thus, there is no need for backtracking.

In this way, it is possible to represent low-level code optimizations in OORS, such as instruction rescheduling (see Example 8, Figure 3) or instruction merging, such as `MUL` and `ADD` into `MAD` (see Example 9).

Example 8 (TriCore instruction rescheduling optimization). On many recent architectures, the instruction order has a major influence on the execution time. For instance, the TriCore architecture [21] is only able to dispatch two instructions at once, if the first instruction will be executed in the arithmetic-logical unit (ALU) and the second instruction will be issued to the load-store unit (LSU). So, the following rule tries to pull a distant memory instruction behind an arithmetic-logical instruction. Although the implicit condition will verify whether possible side effects occur, the developer has to check manually whether it is safe to push the memory instruction in front of the second matched instruction (see the `condition` line; the absence of conflicts with the wildcard is guaranteed by the implicit condition). The cost function favors the match with the greatest distance between the ALU- and the LSU-instruction (for a wildcard pattern, the $-operator denotes the number of instructions the pattern has matched). Figure 3 shows this graphically.

```
rule reschedule {
  search:    [ Op($$->isIssuedTo(ALU)),
               Op(!$$->isIssuedTo(LSU)),
               *,
               Op($$->isIssuedTo(LSU)) ]
  condition: { return !$2->conflictsWith($4); }
  cost:      { return -$3; }
  replace:   [ $1, $4, $2 ]
}
```

Example 9 (Instruction merging). Example 8 showed how to reorder instructions, but the optimization step of OORS can also create and delete instructions. If a target processor supports a *multiply-accumulate* operation computing $a \cdot b + c$ in a single step[4], a multiplication and an addition with appro-

---

[4] If the operation is fused (rounds only after the final addition, not after the intermediate multiplication), a transformation of a multiplication-addition sequence into this operation may change the precision of the result.
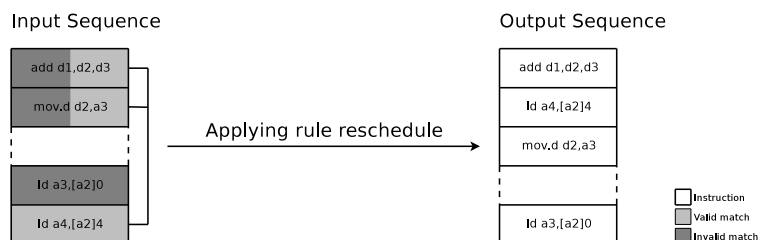
Fig. 3: Valid and invalid match of the `reschedule` rule. The first match (dark-gray) is invalid, because there is a definition-use dependency between `mov.d d2,a3` and `ld a3,[a2]0` (the first instruction reads from and the second instruction writes to register `a3`). The rule's condition prevents this match from being accepted.

priate targets be combined into a new, ternary operation. Local conditions on the matched `BinOp` instructions ensure that the target of the multiplication ($1) is present as exactly one operand of the addition ($3).

```
rule merge_mad {
  search:    [ BinOp($$->opcode == OP_MUL),
               *,
               BinOp($$->opcode == OP_ADD &&
                     ($1->tgt == $$->op1 ||
                      $1->tgt == $$->op2) &&
                     $1->op1 != $1->op2) ]
  condition: { /* $1->tgt used only in $3 */ }
  cost:      { return -1; }
  replace:   [ TerOp(OP_MAD, $3->tgt, $1->op1, $1->op2,
                     ($1->tgt == $3->op1)?$3->op2:$3->op1) ]
}
```

Because the replace pattern does not copy the matched instructions (by including them via the $-references), the matched instructions are deleted from the instruction stream. The cost function returns -1, because the static instruction count will decrease by 1 after applying the rule `merge_mad`.

4. Practice

In Section 4.1, we introduce the pattern matcher generator OORG and show how to integrate OORG-generated pattern matchers into existing applications. Furthermore, we discuss requirements on the integration process in Section 4.2. Afterwards, we present experimental results with OORS code generation and code optimization pattern matchers being employed in the

CGıS compiler in Section 4.3. Section 4.4 briefly introduces the debugging capabilities of OORG-generated pattern matchers.

4.1. Compiler Integration

The integration of OORS pattern matchers works in much the same way as for tools such as yacc or lex (see Figure 4). From the pattern matching specification, OORG generates a C^{++}-file implementing the matcher. This file has to be compiled and linked with the main compiler. The compiler selects at runtime the profile to be used in a particular compilation and calls the generated matcher for each basic block, passing the sequence of intermediate instructions and receiving the sequence of target instructions. To easily cater for minor variations of the desired rule set, the application program can switch on and off certain rules. For example, different sets of optimizations can be selected in this way.

The CGıS compiler [17, 14] can compile a common input program for GPUs and for SIMD CPUs, using a common intermediate representation. OORG-generated matchers are employed in three phases; the actual matchers used in a compilation are selected at runtime depending on the actual target. An early optimizer performs various transformations on intermediate code, which are needed for implementation on SIMD CPUs. A generator transforms the sequence of of intermediate instructions, which are common for GPUs and SIMD CPUs, into a target-specific instruction sequence. To this end, there are three hierarchies of profiles: For various generations of GPUs, SSE, and AltiVec. In a later phase, the GPU code is transformed by an OORG-generated peephole optimizer.
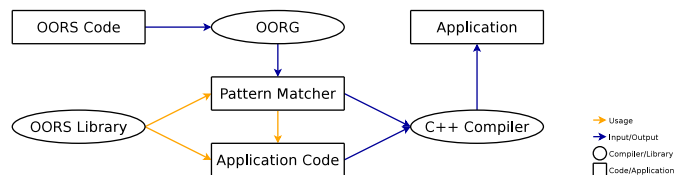
Fig. 4: Integration of OORS into an existing application.

4.2. Requirements on the Integration

To integrate an OORG-generated pattern matcher into a compiler, a certain infrastructure needs to be present. The representation of the intermediate

code as a sequence of virtual instructions per basic block, where each instruction is an instance of some class, is the basic requirement for the tool's applicability. Other work has to be done, however, to achieve a complete integration.

Currently, our pattern matcher generator OORG only supports C^{++}. This inescapably means that at least those parts of the compiler must be implemented in C^{++}, where OORS pattern matchers should come into play. Furthermore, the current implementation uses the standard template library STL [20] for internal representation. Thus, the compiler likewise has to use the STL to represent sequences of instructions. However, adopting OORG to a different object-oriented language, such as Java or C#, or to using different types of data structures for internal representation does not pose an insuperable problem.

When planning to integrate OORS into an existing compiler, the developer inevitably has to face these restrictions. When it comes to implementing rules for code generation or code optimization, the developer has to take other problems into account, as discussed in the following.

One requirement is that of accurate *liveness* information at the instructions. The liveness is explicitly needed by certain transformations, e. g., the elimination of the intermediate multiplicative result by the rule `merge_mad` in Example 9 is valid only if it is not live after the addition instruction. In the same rule, it is guaranteed that the liveness of the intermediate result spans until the addition instruction[5], because this is verified by the reordering constraint across a $*$-pattern checked by the implicit condition (see Section 3.1). That *implicit condition* itself has to be written by the designer of the rule set. However, this dependency analysis is quite simple given the instruction's representation.

In our examples, the *cost* function was rather straightforward, because there were no conflicting optimization cases: Although a particular subsequence of instructions could be matched in a multitude of ways, there always was one match which could be statically and locally determined to be preferable. Thus, the cost functions in the rules needed only to make sure that the preferable match is chosen to achieve the optimal result.

In general, however, the situation is more complex: Different optimizations might preclude one another. For example, consider the combined Examples 8 and 9. In an architecture which can issue the simple arithmetical operations of multiplication and addition simultaneously to the memory instructions, but cannot do so for the more complicated accumulation instruction, the two optimization goals conflict, and it is not immediately obvious how the conflict can be resolved locally.

---

[5] We assume that it is live at all.

In these kinds of situation, the compiler writer has to use heuristics to statically approach a predicted, dynamic result, just like he would have to do in other code optimization methods. However, OORS can still aid the programmer by its backtracking or global search, which can achieve a guaranteed static optimum.

4.3. Experimental Results

This section demonstrates the OORG-generated code generation and code optimization pattern matcher that are employed in the CGIS compiler. We have compiled eight examples for the NV40 and the G80 architecture. The NV40 code generator pattern matcher comprises 42 rules, whereas the G80 code generator pattern matcher contains 48 rules. The G80 code generation profile inherits most NV40 rules, but replaces some NV40 rules with more specialized ones. The NV40 and the G80 profile of the code optimization pattern matcher comprise 12 rules, which realize simple optimizations, such as dead code elimination, constant folding, and constant propagation. Both optimization profiles use the same rule set.

The example applications comprise image filters (*demosaic*, *laplace* and *skeleton*), simulations (*game of life* and *wave propagation*), a mathematical algorithm (*mandelbrot*), a *raycaster* and an encryption algorithm (*RC5*). We have compiled these examples on a Pentium 4 2.6GHz with 512MB RAM running under Linux (Ubuntu 6.06). To determine the values shown in Table 1 and Table 2, we have compiled the test examples seven times and omitted the worst and the best run.

Table 1 shows the time required to compile and optimize the examples using the NV40 profile. On average, the NV40 code generation pattern matcher compiles an abstract instruction within 0.23*ms*. The code optimization pattern matcher is slightly slower and optimizes an instruction within 0.35*ms*.

Table 2 shows the time to compile and optimize the examples for the G80 architecture. The G80 code generation pattern matcher is slightly slower than the NV40 code generation pattern matcher, which is expected, because the G80 profile comprises more rules than the NV40 profile. On average, it takes about 0.25*ms* to compile an abstract instruction. Unsurprisingly, the G80 code optimization pattern matcher is just as fast as the NV40 code optimization pattern matcher.

For both profiles, the CGIS compiler spends approximately 10% of the total compile time within the code generation and the code optimization pattern matcher. So, the influence of OORG-generated code generation and code optimization pattern matchers of the overall runtime is negligible.

Gernot Gebhard and Philipp Lucas

Table 1: Time to compile and optimize examples for the NV40 architecture. RC5 could not be compiled, because the NV40 architecture does not support integer arithmetic.

Test	Abstract Instr.	Gen. Instr.	Gen. Time (*ms*)	Opt. Instr.	Opt. Time (*ms*)
demosaic	113	88	17.8	83	4.0
laplace	93	70	10.4	66	5.4
life	85	76	7.2	74	13.2
mandelbrot	145	105	18.8	92	14.8
raycaster	673	471	100.6	447	319.6
RC5	–	–	–	–	–
skeleton	760	456	313.6	456	67.0
wave	346	255	56.4	243	77.2
Average	316.4	217.3	75.0	208.7	75.2

However, there is room available for performance improvements. Currently, OORG-generated pattern matchers match the rules one after another, which is somewhat inefficient. A great deal of time could be saved, if the generated pattern matchers would match the used rules in parallel. Additionally, other minor improvements to the OORS library could further decrease the runtime of OORG-generated pattern matchers.

Table 2: Time to compile and optimize examples for the G80 architecture.

Test	Abstract Instr.	Gen. Instr.	Gen. Time (*ms*)	Opt. Instr.	Opt. Time (*ms*)
demosaic	113	88	17.8	83	6.0
laplace	93	70	13.6	66	4.0
life	85	76	9.0	74	13.2
mandelbrot	145	105	18.5	92	15.8
raycaster	673	471	114.6	447	172.8
RC5	136	113	18.2	111	59.0
skeleton	760	456	339.2	456	232.0
wave	346	255	64.2	243	71.6
Average	293.9	204.3	74.4	196.5	71.8

Retargeting the CGiS compiler to the NV40 and G80 compiler was not much of an effort in terms of lines of OORS code (locs). The basis of the OORS code generation pattern matcher forms an NV30 profile, which comprises about 770 locs. The NV40 profile inherits from the NV30 profile adding about 200 locs to the pattern matcher specification. Adding support for the G80 GPU architecture required another 350 locs.

Retargeting the code optimization pattern matcher required even less effort. An initial NV30 optimization profile comprising about 300 locs provides the basic functionality. The NV40 optimization profile adds just a single rule in 20 locs. The G80 optimization profile is just one line of code (the G80 optimization profile is an alias of the NV40 optimization profile).

4.4. Debugging

To understand how a pattern matcher processes its input, the OORS library provides the use with a debugger interface. During runtime, this interface receives different kind of events that describe a state transition within the generated pattern matcher. Currently, the debugger interface emits five different classes of events, which are introduced in the following.

rule events: The *currentRule* event informs the debugger that a new rule starts to match the current input. If a rule has finished matching the current input instruction stream, the pattern matcher emits the *finishRule* event. When the pattern matcher is going to apply the rule, the debugger interface receives the *applyRule* event.

match events: Whenever a rule creates a new alternative[6], the generated pattern matcher emits the *newAlternative* event. To indicate which alternative is currently being processed, the pattern matcher produces the *currentAlternative* event. To report that an alternative could not be processed any further, the debugger interface receives the *deleteAlternative* event.

condition events: Before an item pattern may match an object of the input stream, the pattern matcher must first check the local side condition of that item pattern. The *checkItemPattern* event reports, whether the local side condition is satisfied (in case not, a *deleteAlternative* event follows).

pattern events: The events *matchItemPattern* and *matchWildcardPattern* indicate that an item pattern or a wildcard pattern respectively has been matched against a symbol of the input stream. Whenever a rule decides not to match the input stream against a wildcard pattern, the pattern matcher emits the *finishWildcardPattern* event.

basic block events: After processing a basic block has finished, the pattern matcher generates the *finishBasicBlock* event, which reports the sum of the costs of the applied rules.

---

[6] An *alternative* represents the current, unfinished match of a rule.

Gernot Gebhard and Philipp Lucas

5. Conclusion and Future Work

In this paper, we have presented the new object-oriented rewrite system OORS with applications in code generation and code optimization. Using the presented OORS language, a developer is able to implement the code generation and code optimization phase of a compiler's back end in terms of pattern matchers. Retargetability is achieved by organizing rules into profiles, one for each supported hardware architecture. In contrast to other approaches, the OORS language features constructs, such as a rule and profile inheritance mechanism, that make the reuse of existing specification possible. Thus, an OORS pattern matcher specification is maintainable as well as easily retargetable in the long run.

We have additionally introduced the pattern matcher generator OORG that compiles an OORS specification into a C^{++} dynamically retargetable pattern matcher. By means of the CGIS compiler, we have demonstrated the usage of OORG-generated pattern matchers in a real world application. OORG is open source and available for download on our homepage: http://rw4.cs.uni-sb.de/ gebhard/projects/oors/.

The OORS language offers room for further improvements. Currently, OORS pattern matchers process only basic blocks. This restriction decreases the efficiency of certain optimizations, such as dead-code elimination. A dead-code elimination rule is currently not in itself able to determine whether the target register is still live, if e.g., a register is written at the end of a basic block. Thus, we want to extend OORS such that matches over the whole control flow graph are possible. Apart from that, all instructions are assumed to be pushed upwards past wildcard patterns (remember Example 8, where a load-store instruction is pushed upwards). In some cases however, a developer might want to push instructions the other way around. To further improve the expressiveness of OORS, we thus want to introduce a mechanism that indicates the *direction* of a rule. Finally, we plan to improve the performance of the OORG-generated pattern matchers. The current implementation generates pattern matchers that match each rule one after another. This matching method becomes inefficient if the search patterns of two (or more) rules share the same prefix. In such a case, the OORG-generated pattern matcher would match that prefix multiple times. So, to overcome this drawback, we want to improve OORG, such that the generator produces pattern matchers that match all rules in parallel.

References

1. M. Alt, U. Aßmann, and H. van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In P. Fritzson, editor, *Proceedings of the 5th Inter-*

national Conference on Compiler Construction (CC'94), volume 786 of *LNCS*, pages 278–293. Springer-Verlag, 1994.

2. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: Components for Transformation Systems. In J. Hatcliff and F. Tip, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'06)*, pages 95–99. ACM, January 2006.

3. M. Breternitz Jr., H. Hum, and S. Kumar. Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 135, Washington, DC, USA, 2003. IEEE Computer Society.

4. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

5. Quick C-- compiler. http://cminusminus.org/qc--.html.

6. J. R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3), August 2006.

7. J. W. Davidson and C. W. Fraser. Register Allocation and Exhaustive Peephole Optimization. *Software – Practise and Experience*, 14(9):857–865, September 1984.

8. J. Dias and N. Ramsey. Converting Intermediate Code to Assembly Code Using Declarative Machine Descriptions. In A. Mycroft and A. Zeller, editors, *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*, pages 217–231, 2006.

9. H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG – A Generator for Efficient Back Ends. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'89)*, pages 227–237, 1989.

10. S. Farfeleder, A. Krall, E. Steiner, and F. Brandner. Effective Compiler Generation by Architecture Description. In M. J. Irwin and K. D. Bosschere, editors, *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, pages 145–152. ACM Press, 2006.

11. C. Ferdinand, H. Seidl, and R. Wilhelm. Tree Automata for Code Selection. *Acta Informatica*, 31(9):741–760, 1994.

12. C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

13. C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG – Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.

14. N. Fritz, P. Lucas, and R. Wilhelm. Exploiting SIMD Parallelism with the CGiS Compiler Framework. In V. Adve, M. J. Garzarán, and P. Petersen, editors, *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, LNCS. Springer-Verlag, October 2007.

Gernot Gebhard and Philipp Lucas

15. G. Gebhard and P. Lucas. OORS: An Object-Oriented Rewrite System with Applications in Retargetable Code Generation and Optimization. In M. Mernik, editor, *Proceedings of the 1st Workshop on Advances in Programming Languages (WAPL'07)*, pages 1057–1069, October 2007.

16. S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. *SIGPLAN Notices*, 40(1):364–377, 2005.

17. P. Lucas, N. Fritz, and R. Wilhelm. The CGiS Compiler—A Tool Demonstration. In A. Mycroft and A. Zeller, editors, *Proceedings of the 15th International Conference on Compiler Construction (CC)*, volume 3923 of *LNCS*, pages 105–108. Springer-Verlag, 2006.

18. NVIDIA. Nv_gpu_program4. OpenGL Extension 322, 2007.

19. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

20. The Standard Template Library. http://www.sgi.com/tech/stl/.

21. TriCore microcontroller. http://www.infineon.com/tricore/.

22. M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In R. Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction (CC'01)*, pages 365–370, 2001.

23. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24:334–368, July 2002.

24. A. Warth and I. Piumarta. OMeta: an Object-Oriented Language for Pattern Matching. In P. Costanza and R. Hirschfeld, editors, *DLS '07: Proceedings of the 2007 ACM SIGPLAN symposium on Dynamic Languages*. ACM, October 2007.

Appendix

Although the primary focus of OORS and the reason for its conception is the use in code generation and optimization, it is, in fact, a very general list pattern matcher. The following two examples show the use of OORS in this more general sense. First, a list of elements is sorted by OORS' optimization capabilities: A sequence is not-optimal (and hence subject to an OORG-generated optimization) if it contains an unsorted subsequence. Second, we present an even more general example, where the pattern matcher is used for static expression evaluation.

List Sorting

This example shows how to realize a very simple list sorting algorithm, also known as bubble sort. The pattern matcher sorts (optimizes) an arbitrary list in either ascending or descending order with respect to the value of each item. The pattern matcher comprises two profiles, one to sort a list in ascending order and the other to sort a list in descending order. Both profiles contain a rule named sort that flips two adjacent items in the list depending on their value. The rule in the profile `Ascending` checks if the value of the first item is smaller than the value of the second item and flips both items to push the cheap item to left and the expensive item to the right. Note that it is not necessary to respecify the search pattern, the cost function or the replace pattern, if another sorting behavior is desired.

```
profile Ascending {
  rule sort {
    search:    [ Item, Item ]
    condition: { return $1->value() > $2->value(); }
    cost:      { return 1; }
    replace:   [ $2, $1 ]
  }
}

profile Descending {
  rule sort : extends Ascending::sort {
    condition: { return $1->value() < $2->value(); }
  }
}
```

Independent from the used sort profile, the pattern matcher sorts the list in the desired order after a finite number of steps. As expected, the sorting method is quite inefficient and has a worst runtime of $O(n^2)$, where n is the length of the list. In any case, this example shows that OORS can be used to implement various kinds of scheduling algorithms. Note in particular that the search pattern need not be specified consecutively, that is, that it can (re-)schedule distant elements.

Polish-Notation Calculator

This example demonstrates how to implement a Polish and a Reverse Polish notation calculator in OORS. The Polish notation is a special kind of notation for logic, arithmetic and algebra. Under the assumption that the arity of each operator is given, this notation is able to function without any kind of parenthesis. The Polish notation is also known as *prefix notation*, because it places the operators in front of their arguments. In contrast to the Polish

notation, the Reverse Polish notation, also known as *postfix notation*, places the operators after their arguments.

Given the expression $e = (2 + ((2 * 4.5)/0.5))/(3 - 1.5)$. The expressions e_{PN} and e_{RPN} are equivalent expressions in Polish and Reverse Polish notation respectively:

$$e_{PN} = /\ +\ 2\ /\ x\ 2\ 4.5\ 0.5\ -\ 3\ 1.5$$
$$e_{RPN} = 2\ 2\ 4.5\ x\ 0.5\ /\ +\ 3\ 1.5\ -\ /$$

Due to the simple structure of Polish notation expressions, a pattern matcher that evaluates these expressions can be easily realized. The pattern matcher "optimizes" a list of instances of the Object class, from which the classes Operator and Number derive. Each number has a unique value that can be accessed with the value function. An operator implements the eval function that computes the result of the operation. To simplify this example, it is assumed that all operators are binary. So, the pattern matcher is implemented as follows:

```
profile Polish {
  rule Step {
    search:   [ Operator , Number , Number ]
    cost:     { return 1; }
    replace:  [ Number($1->eval($2->value(), $3->value())) ]
  }
}
profile ReversePolish {
  rule Step {
    search:   [ Number , Number , Operator ]
    cost:     { return 1; }
    replace:  [ Number($3->eval($1->value(), $2->value())) ]
  }
}
```

Depending on the given profile, the generated pattern matcher evaluates the given expression by iteratively applying the rule `step` as long as possible. To detect an invalid expression, the user simply has to check whether the final expression only contains one instance of the class `Number`. The number of necessary steps increases linearly with the number of operators. So, the overall runtime is $O(n)$, where n is the number of operators.

Gernot Gebhard has studied from 2001 to 2006 at the Saarland University. He is now half-time employed as research assistant at the Compiler Design Lab of the Saarland University and is additionally half-time employed as software engineer at AbsInt Angewandte Informatik GmbH.

After having studied at RWTH Aachen, **Philipp Lucas** works at the Compiler Design Lab of Saarland University on languages and compilers for data-parallel programming.