Optimizing verified optimizations

Brae J. Webb[1 [0000−0003−3579−0244]], Ian J. Hayes[1 [0000−0003−3649−392X]], and Mark Utting[1 [0000−0003−3134−6306]]

School of Electrical Engineering and Computer Science,
The University of Queensland, Brisbane, Australia

Abstract. Compiler optimizations that transform expressions can be expressed as conditional rewriting rules. These rules are more straightforward to verify than the compiler code that implements them. Directly using a set rewriting rules to perform optimizations can introduce a high overhead during compilation. Our goal is to take verified a set of optimizations expressed as rewriting rules and generate a suitably efficient implementation that performs the rewrite without overhead. We propose a strategy for combining the rewriting rules and present methods for meta-optimizing the rule combinations, before generating code. For our collection of rewriting rules, the optimized generated implementation provides a 40% reduction in lines of code compared with the naive generated implementation.

Keywords: Compilers · Optimization · Verification · Rewriting rules.

1 Introduction

As part of ongoing efforts into verifying the optimization passes of the GraalVM compiler [8], a high-level Domain-Specific Language (DSL) for expressing compiler optimizations as conditional rewriting rules has been developed [12]. This DSL is implemented in the Isabelle/HOL theorem prover [7] and enables simple specification and verification of optimizations. Below are four example conditional rewriting rules for optimizations applicable to an expression with a multiplication at the top level (where $\ll$ is left shift).

$$x * \mathsf{const}\ 1 \mapsto x \qquad \text{(Identity)}$$

$$\mathsf{const}\ c * \mathsf{const}\ d \mapsto \mathsf{const}\ (c * d) \qquad \text{(Evaluate)}$$

$$x * \mathsf{const}\ c \mapsto x \ll \mathsf{const}\ (\log 2\ c) \quad \textbf{when} \quad \mathrm{Power2}\ c \qquad \text{(Shift)}$$

$$\mathsf{const}\ c * y \mapsto y * \mathsf{const}\ c \quad \textbf{when} \quad \mathrm{Not}\ (\mathrm{IsConst}\ y) \qquad \text{(LeftConst)}$$

The verification of a rewriting rule shows that, if a rule is applicable to an expression, the rewriting rule preserves the semantics of the expression [13]. The rewriting rules are derived by inspecting (and untangling) the GraalVM compiler code and hence there is no guarantee that they correspond *exactly* to the compiler source code. There are two ways of tackling this issue: (a) verify that the Java code of the compiler implements the rewriting rules; and (b) generate the Java code from the rewriting rules and verify that process. We have chosen to explore the second approach because it is more amenable to automated verification.

Stages of generation. The process is split into the following stages: strategy language, lowering, meta-optimization and code generation.

Strategy language. A simple strategy language is introduced to combine rewriting rules using constructs like sequencing ($\fatsemi$) and non-deterministic choice. For example, the following applies rule LeftConst, then either Evaluate or Identity, or, if neither of those is applicable, Shift.

$$\mathsf{LeftConst} \fatsemi (\mathit{choice}\ [\mathsf{Evaluate}, \mathsf{Identity}]\ \mathit{else}\ \mathsf{Shift}) \tag{1}$$

When optimizing an expression, its sub-expressions are first optimized before optimization of the top-level of the expression takes place because this may enable optimizations at the top-level that were not applicable to the original expression. To optimize the expression $((2*2)*n)*1$, rule (1) can be applied to the sub-expression $2*2$ to reduce it to 4 using Evaluate (LeftConst fails on the subexpression $2*2$), then applying the rule to $4*n$ first applies LeftConst to give $n*4$ and then Shift to give $n \ll 2$ (both Evaluate and Identity are not applicable in this case); it then reduces the expression $(n \ll 2)*1$ to $n \ll 2$ using rule Identity (LeftConst and Evaluate are not applicable for this case). The strategy language is discussed in Sect. 4.

Lowering. Each rewriting rule is transformed to a sequence of simpler matching primitives. The lowered versions of our example rules are presented below.

$$\mathsf{Identity}_l = \mathsf{match}\ (x * y)\ ?\ \mathsf{matchv}\ y\ (\mathsf{const}\ 1)\ ?\ x \tag{2}$$

$$\mathsf{Evaluate}_l = \mathsf{match}\ (x * y)\ ?\ \mathsf{matchv}\ x\ (\mathsf{const}\ c)\ ?\ \mathsf{matchv}\ y\ (\mathsf{const}\ d)\ ?\ \mathsf{const}\ (c * d) \tag{3}$$

$$\mathsf{Shift}_l = \mathsf{match}\ (x * y)\ ?\ \mathsf{matchv}\ y\ (\mathsf{const}\ c)\ ?\ \mathsf{cond}\ (\mathrm{Power2}\ c)\ ?\ x \ll \mathsf{const}\ (\log 2\ c) \tag{4}$$

$$\mathsf{LeftConst}_l = \mathsf{match}\ (x * y)\ ?\ \mathsf{matchv}\ x\ (\mathsf{const}\ c)\ ?\ \mathsf{cond}\ (\mathrm{Not}\ (\mathrm{IsConst}\ y))\ ?\ y * \mathsf{const}\ c \tag{5}$$

Rule $\mathsf{Identity}_l$ first matches against a multiplication at the top level –with variables x and y introduced to stand for the left and right operands– then if that succeeds it proceeds to match y against the constant 1; if the matches succeed the result is x. The matching process accumulates a substitution mapping the variable names to the sub-expressions they matched. Sect. 3 discusses lowering.

Meta-optimisation. The optimisations expressed in the strategy language (in lowered form) are themselves subjected to meta-optimisation (with the aim of generating more efficient compiler code), for example, the meta-optimization combine-else factors out a common initial match primitive of both branches of an else:

$$((m\ ?\ r_1)\ \mathit{else}\ (m\ ?\ r_2)) \mapsto m\ ?\ (r_1\ \mathit{else}\ r_2) \qquad \text{(combine-else)}$$

Figure 1 gives the result of meta-optimisation of (1) after lowering. Sect. 5 discusses meta-optimisation.

$$(\mathsf{match}\,(x * y)\ ?\ \mathsf{matchv}\ x\,(\mathsf{const}\ c)\ ?\ \mathsf{cond}\,(\mathrm{Not}\ (\mathrm{IsConst}\ y))\ ?\ y * \mathsf{const}\ c)\ \S$$
$$(\mathsf{match}\,(x * y)\ ?\ ((\mathsf{matchv}\ x\,(\mathsf{const}\ c)\ ?\ \mathsf{matchv}\ y\,(\mathsf{const}\ d)\ ?\ \mathsf{const}\,(c * d))\ \mathit{else}$$
$$(\mathsf{matchv}\ y\,(\mathsf{const}\ 1)\ ?\ x)\ \mathit{else}$$
$$(\mathsf{matchv}\ y\,(\mathsf{const}\ c)\ ?\ \mathsf{cond}\,(\mathrm{Power2}\ c)\ ?\ x \ll \mathsf{const}\,(\log 2\ c))))$$

Fig. 1: Optimization strategy (1) after lowering and meta-optimization.

```
ValueNode canonical() {
    ValueNode e = this; // default if no optimisation performed
    if (e instanceof MulNode m) {
        ValueNode x = m.getX(); ValueNode y = m.getY();
        if (x instanceof ConstNode c && !(y instanceof ConstNode)) {
            e = MulNode.create(y, new ConstNode(c.value()));
        }
    }
    if (e instanceof MulNode m) {
        ValueNode x = m.getX(); ValueNode y = m.getY();
        if (x instanceof ConstNode c && y instanceof ConstNode d) {
            e = ConstNode.forInt(c.value() * d.value()); //Evaluate
        } else if (y instanceof ConstNode c && c.value() == 1) {
            e = x; // Identity
        } else if (y instanceof ConstNode c && isPower2(c.value())) {
            e = LeftShiftNode.create(x, new ConstNode(log2(c.value())));
            // Shift
        }
    }
    return e;
}
```

Fig. 2: Java code corresponding for optimization of a *MulNode*

Code generation. The previous stages are independent of the programming language in which the compiler is written and hence could be used for optimizers for compilers other than GraalVM. For GraalVM, Java code is generated from the meta-optimised optimisations expressed in the (lowered) strategy language. This step is a reasonably straight-forward translation. In the GraalVM compiler expressions are represented by term graphs (which can be thought of as expression trees that share common sub-expressions). The compiler creates the nodes (of class *ValueNode*) for an expression bottom up (i.e. it creates the nodes for sub-expressions before creating the node for an expression). Expression optimisations are performed as part of the node creation process by calling the method *canonical* for the corresponding class of node. For example, the Java code for

the *canonical* method for class *MulNode* corresponding to Figure 1 is given in Figure 2. The Java code has similar complexity to the handwritten GraalVM compiler code. Note that when the generated code in Figure 2 is compiled as part of compiling the compiler, the GraalVM optimisations will be applied to that code, removing some of the redundancy in the code.

Verification tasks There are a number of verification steps applicable throughout the process outlined above:

- verification that individual rewriting rules preserve the expression semantics — this has been completed in Isabelle/HOL, as described in [12];
- verification that lowering a rewriting rule preserves the semantics of the rewriting rule — discussed in Sect. 3;
- verification that an optimisation expressed in the strategy language is semantics preserving, provided each of the rewriting rules used in the optimisation is semantics preserving — discussed in Sect. 4;
- verification that the transformation of the optimisation performed by the meta-optimisation process preserves the semantics of the optimisation — discussed in Sect. 5; and
- verification that the semantics of the Java code generated from an optimisation expressed in the (lowered) strategy language implements the optimisation — verification of this step has not been attempted as the lowered strategy language and Java are at a similar level of abstraction, so that the translation is a simple syntactic transformation.

An important property of this approach is that as new optimizations (rewriting rules) are added to the optimiser, it suffices to verify that each rewriting rule is semantics preserving. The remaining verifications are generic and only need to be done once. As GraalVM is a Just-in-Time compiler, the overhead for performing optimizations must be minimal. To accomplish this, we generate efficient code implementing the rewriting rules. We evaluate the effectiveness of the approach in Sect. 7.

2 Rewriting Semantics

The abstract syntax of expressions is given by the datatype $\mathbb{E}$, where we have elided the alternatives for leaf expressions, such as method parameters.

$$\mathbb{E} ::= \mathsf{UnaryExpr}\ \texttt{UnaryOp}\ \mathbb{E} \mid \mathsf{BinaryExpr}\ \texttt{BinaryOp}\ \mathbb{E}\ \mathbb{E} \mid \mathsf{CondExpr}\ \mathbb{E}\ \mathbb{E}\ \mathbb{E} \mid \mathsf{ConstExpr}\ \texttt{Value} \mid \ldots$$

To validate code generation from conditional rewriting rules, we develop a semantic description of rewriting at the expression level. The rewriting of an expression, e, to another expression, e', is defined by

1. a pattern, p_m, upon which e has to match,

2. a condition *cond* in terms of the free variables of p_m, and
3. a corresponding result pattern, p_r, the free variables of which must be a subset of those of p_m.

If e matches the pattern p_m then all free variables in p_m are bound to sub-expressions of e in a substitution σ, then if *cond* is satisfied by σ, the expression, e', is the result substituting the free variables in p_r according to σ, otherwise the rewrite rule fails.

The datatype $\mathbb{P}$ is effectively an extension of the base expression type, $\mathbb{E}$, where the pattern for constant expression may either match a particular value, ConstPat, (e.g. corresponding to const 1 in rule $\mathsf{Identity}_l$) or a symbolic name can be used to stand for the value of the constant, ConstVarPat , (e.g. corresponding to const c in rule $\mathsf{Evaluate}_l$), and VarPat $\mathbb{V}$ is used to match an arbitrary sub-expression and give it a name, where $\mathbb{V}$ is the type of variable names.

$$\begin{aligned}\mathbb{P} ::= {} & \mathsf{UnaryPat}\ \texttt{UnaryOp}\ \mathbb{P} \mid \mathsf{BinaryPat}\ \texttt{BinaryOp}\ \mathbb{P}\ \mathbb{P} \mid \mathsf{CondPat}\ \mathbb{P}\ \mathbb{P}\ \mathbb{P} \\ & \mid \mathsf{ConstPat}\ \texttt{Value} \mid \mathsf{ConstVarPat}\ \mathbb{V} \mid \mathsf{VarPat}\ \mathbb{V}\end{aligned}$$

Definition 1 (substitution type)**.** A substitution is a partial mapping from variable names to expressions, $\mathbb{S} ::= \mathbb{V} \to \mathbb{E}\ \texttt{option}$, where for a type T, $T\ \texttt{option} ::= \mathsf{None} \mid \mathsf{Some}\ T$.

Definition 2 (expression match)**.** The semantics of a pattern, p, matching an expression, e, is given by match-tree $p\ e$. The result is an optional substitution mapping all free variables in p to sub-expressions of e. None indicates a failed match. If a pattern expression is just a variable, then the substitution is the variable bound to the expression (6). A literal constant pattern can be matched against a constant that holds the same value (7), producing an empty substitution. A constant variable pattern can be matched against a constant, and the resulting substitution binds the variable to the constant (8). For unary and binary operations, the operation of the pattern must be the same as the operation of the expression. For unary (9), binary (10), and conditional (11) patterns, the pattern is matched against the expression recursively. The substitution union operator, $\sigma_1 \uplus \sigma_2$, combines two substitutions, σ_1 and σ_2, where σ_1 and σ_2 are compatible, i.e. any common entries are equal. If the substitutions are not compatible or either substitution is None, then the result is None. If none of the above cases apply, then the pattern expression does not match the expression and the result is None (12).

$$\mathsf{match\text{-}tree} : \mathbb{P} \to \mathbb{E} \to \mathbb{S}\ \texttt{option}$$

$$\mathsf{match\text{-}tree}\ (\mathsf{VarPat}\ vn)\ e = \mathsf{Some}\ \{vn \mapsto e\} \tag{6}$$

$$\mathsf{match\text{-}tree}\ (\mathsf{ConstPat}\ v)\ (\mathsf{ConstExpr}\ v) = \mathsf{Some}\ \emptyset \tag{7}$$

$$\mathsf{match\text{-}tree}\ (\mathsf{ConstVarPat}\ vn)\ (\mathsf{ConstExpr}\ v) = \mathsf{Some}\ \{vn \mapsto (\mathsf{ConstExpr}\ v)\} \tag{8}$$

$$\mathsf{match\text{-}tree}\ (\mathsf{UnaryPat}\ op\ p_x)\ (\mathsf{UnaryExpr}\ op\ e_x) = \mathsf{match\text{-}tree}\ p_x\ e_x \tag{9}$$

$$\begin{aligned}&\mathsf{match\text{-}tree}\ (\mathsf{BinaryPat}\ op\ p_x\ p_y)\ (\mathsf{BinaryExpr}\ op\ e_x\ e_y) = \\ &\quad (\mathsf{match\text{-}tree}\ p_x\ e_x) \uplus (\mathsf{match\text{-}tree}\ p_y\ e_y)\end{aligned} \tag{10}$$

$$\begin{aligned}&\mathsf{match\text{-}tree}\ (\mathsf{CondPat}\ p_c\ p_t\ p_f)\ (\mathsf{CondExpr}\ e_c\ e_t\ e_f) = \\ &\quad (\mathsf{match\text{-}tree}\ p_c\ e_c) \uplus (\mathsf{match\text{-}tree}\ p_t\ e_t) \uplus (\mathsf{match\text{-}tree}\ p_f\ e_f)\end{aligned} \tag{11}$$

$$\mathsf{match\text{-}tree}\ \_\ \_ = \mathsf{None} \tag{12}$$

Definition 3 (free variables)**.** We define $\mathcal{L} : \mathbb{P} \to \mathbb{V}\ \texttt{set}$ to be the set of *free variables* contained within a pattern, e.g. $\mathcal{L}\ ((x + 1) - y) = \{x, y\}$.

Definition 4 (substitution)**.** For a pattern p and substitution σ, the *substitution* of free variables in p with their values in σ is given by the notation, $p\ \$\ \sigma$, with type $\mathbb{P} \to \mathbb{S} \to \mathbb{E}\ \texttt{option}$. If a free variable in p is not in the domain of σ, the substitution fails, i.e. returns None.

To save space within the current paper, we only consider unconditional rewriting rules. The extension to conditional rules is straightforward.

Definition 5 (rewrite)**.** For patterns, p_m and p_r, and expression, e, the semantics of rewriting e is defined as e successfully matching p_m followed by substituting free variables in p_r via the substitution produced by the match, σ.

$$\begin{aligned}&\textsf{rewrite} : \mathbb{P} \to \mathbb{P} \to \mathbb{E} \to \mathbb{E}\ \texttt{option}\\&\textsf{rewrite}\ p_m\ p_r\ e = \textbf{if}\ \textsf{match-tree}\ p_m\ e = \textsf{Some}\ \sigma\ \textbf{then}\ p_r\ \$\ \sigma\ \textbf{else}\ \textsf{None}\end{aligned}$$

Soundness. Our goal is to lift expression refinement to pattern refinement. Pattern refinement is defined in terms of expression refinement; if all possible ground expressions of patterns satisfy expression refinement then the patterns are refined.

Definition 6 (expression refinement)**.** For each optimization rule, if expression e optimizes to e', we must show that $e \sqsupseteq e'$. This relation is semantics preservation stating that if e evaluates to a value then e' must also evaluate to the same value (see [12] for definition).

Definition 7 (pattern expression refinement)**.** If all ground instances of a pattern, p_1, are refined by the corresponding ground instance of a pattern, p_2, then p_2 is a *pattern refinement* of p_1.

$$\begin{aligned}&\_ \sqsupseteq_p \_ : \mathbb{P} \to \mathbb{P} \to \mathbb{B}\\&p_1 \sqsupseteq_p p_2 \equiv \forall \sigma\ e_1\ e_2.\ (p_1\ \$\ \sigma = \textsf{Some}\ e_1) \wedge (p_2\ \$\ \sigma = \textsf{Some}\ e_2) \implies e_1 \sqsupseteq e_2\end{aligned}$$

Lemma 8 (pattern refinement refines expressions)**.** For patterns, p_m and p_r, where p_m is refined by p_r, if performing a rewrite on an expression, e, produces e', then e' is a refinement of e.

$$p_m \sqsupseteq_p p_r \wedge \textsf{rewrite}\ p_m\ p_r\ e = \textsf{Some}\ e' \implies e \sqsupseteq e'$$

3 Match-Result Patterns

One challenge in code generation is the representation of rewriting rules. Traditional high-level rewriting rules are not optimal representations as they require additional transformations to be suitable for code generation. To address this, we propose lowering rewriting rules into match-result patterns. Lowering breaks

the matching down into a sequence of simpler steps more amenable for code generation. Match-result patterns allow one to express rewriting rules in a form that can be directly translated into implementation code. Consider again the identity rule for multiplication.

$$x * \mathsf{const}\ 1 \mapsto x \qquad \text{(Identity)}$$

To implement this rule imperatively, we need to

1. ensure that the expression we are matching is a multiplication expression,
2. ensure that the right operand is a constant with the value one, and
3. return the left operand of the multiplication.

This sequence of steps can be closely modelled by match-result patterns.

$$\mathsf{match}\,(x * y)\ ?\ \mathsf{matchv}\ y\,(\mathsf{const}\ 1)\ ?\ x \qquad (13)$$

As the pattern becomes more complex and nested, the number of matches in the match-result pattern grows as shown in (3), (4), and (5). Each match operation only performs a simple non-recursive match which makes it easier to generate implementation code from the pattern.[1] When additional conditions are required for an optimization rule, they can be included with a `when` condition. The condition is included in the chain of conditions in the match-result pattern. If any of the matches or conditions fail, the whole optimization fails to apply.

MATCH type. The first step in the translation process is to match the expression against the pattern. We first restrict the pattern allowed in the match to a non-recursive subset of the DSL. This restriction ensures that each match performs only a simple non-recursive operation.

The code pattern type, $\mathbb{C}$, has the same structure as the pattern type, $\mathbb{P}$, except that instead of the recursive instances of patterns of in $\mathbb{P}$, $\mathbb{C}$ has variable names, where the variable names correspond to a subexpression within a substitution. Note that the alternative, VarPat $\mathbb{V}$, for $\mathbb{P}$ is no longer needed because variable names are already used for sub-expressions in the alternatives for $\mathbb{C}$. This lowered form of matching is easier to translate into an implementation programming language.

$$\mathbb{C} ::= \mathsf{UnaryMatch}\ \texttt{UnaryOp}\ \mathbb{V} \mid \mathsf{BinaryMatch}\ \texttt{BinaryOp}\ \mathbb{V}\ \mathbb{V} \mid \mathsf{CondMatch}\ \mathbb{V}\ \mathbb{V}\ \mathbb{V} \mid \mathsf{ConstMatch}\ \texttt{Value} \mid \mathsf{ConstVarMatch}\ \mathbb{V}$$

The datatype $\mathbb{M}$ represents the steps for matching an expression against a pattern,

$$\mathbb{M} ::= \mathsf{match}\ \mathbb{C} \mid \mathsf{matchv}\ \mathbb{V}\,\mathbb{C} \mid \mathsf{matchl}\,(\mathbb{M}\ \mathit{list}) \mid \mathbb{V} == \mathbb{V}$$

where match p, matches the *focus* expression, e, against the pattern, p, binding the free variables in p to the corresponding subexpressions of e; matchv $v\,p$,

[1] This is similar to A-normalisation as used by Flanagan et al [3].

promotes the expression at v within the current substitution to be the focus and matches against a pattern, p, binding the free variables in p; matchl ps, sequentially matches the patterns in the list ps, accumulating the bindings of their free variables; and $v_1 == v_2$, asserts that the instantiations of two variables, v_1 and v_2 are equal; if they are not equal, the match fails. This definition does not include side conditions of rewrite rules (i.e. "when" clauses) to simplify the presentation.

Definition 9 (unify)**.** The relation, unify : $(\mathbb{V} \times \mathbb{E})\ list \to \mathbb{S} \to \mathbb{S} \to \mathbb{B}$, unifies a substitution, σ, with a list of (v, e) pairs where v is a variable name and e is an expression, to give a result substitution σ'. If σ already maps v to e, σ' is σ. If v is not mapped in σ, σ is updated to map v to e within σ'. Note that unify fails if the list is incompatible with the substitution.

Definition 10 (match semantics)**.** We define $[\![m]\!]\ e\ \sigma \approx \sigma'$ of type, $\mathbb{M} \to \mathbb{E} \to \mathbb{S} \to \mathbb{S} \to \mathbb{B}$, as the execution of the match pattern, m, against a focus expression, e, and an existing substitution, σ. If successful, execution of the match pattern produces an updated substitution, σ'. The updated substitution binds the free variables in the pattern to sub-expressions of e. If the pattern is not compatible with the existing substitution, the rule fails.

$$\frac{\sigma\, v = \text{Some}\ e' \qquad [\![\text{match}\ p]\!]\ e'\ \sigma \approx \sigma'}{[\![\text{matchv}\ v\ p]\!]\ e\ \sigma \approx \sigma'} \tag{14}$$

$$\frac{\text{unify}\ [(x, e_x)]\ \sigma\ \sigma'}{[\![\text{match}\ (\text{UnaryMatch}\ op\ x)]\!]\ (\text{UnaryExpr}\ op\ e_x)\ \sigma \approx \sigma'} \tag{15}$$

$$\frac{\text{unify}\ [(x, e_x), (y, e_y)]\ \sigma\ \sigma'}{[\![\text{match}\ (\text{BinaryMatch}\ op\ x\ y)]\!]\ (\text{BinaryExpr}\ op\ e_x\ e_y)\ \sigma \approx \sigma'} \tag{16}$$

$$\frac{\text{unify}\ [(c, e_c), (t, e_t), (f, e_f)]\ \sigma\ \sigma'}{[\![\text{match}\ (\text{CondMatch}\ c\ t\ f)]\!]\ (\text{CondExpr}\ e_c\ e_t\ e_f\ )\ \sigma \approx \sigma'} \tag{17}$$

$$[\![\text{match}\ (\text{ConstMatch}\ c)]\!]\ (\text{ConstExpr}\ c)\ \sigma \approx \sigma \tag{18}$$

$$\frac{\text{unify}\ [(v, \text{ConstExpr}\ c)]\ \sigma\ \sigma'}{[\![\text{match}\ (\text{ConstVarMatch}\ v)]\!]\ (\text{ConstExpr}\ c)\ \sigma \approx \sigma'} \tag{19}$$

$$\frac{v_1 \in \text{dom}\ \sigma \qquad v_2 \in \text{dom}\ \sigma \qquad \sigma\, v_1 = \sigma\, v_2}{[\![v_1 == v_2]\!]\ e\ \sigma \approx \sigma} \tag{20}$$

$$\frac{[\![m]\!]\ e\ \sigma \approx \sigma' \qquad [\![\text{matchl}\ ms]\!]\ e\ \sigma' \approx \sigma''}{[\![\text{matchl}\ m{\cdot}ms]\!]\ e\ \sigma \approx \sigma''} \tag{21}$$

$$[\![\text{matchl}\ [\,]]\!]\ e\ \sigma \approx \sigma \tag{22}$$

Where a match occurs against a variable, the expression that variable maps to is promoted to the focus and execution continues against the pattern (14); note that e is not used here. For matches with patterns that contain variables (15, 16, 17), the relation holds if σ can be unified with the free variables mapped to the corresponding subexpressions. Match patterns without variables (18) hold if the constants in the pattern and expression are the same. Constant variable patterns (19) match a constant expression if v is already bound to the constant expression. Equality ($v_1 == v_2$) holds if both v_1 and v_2 are contained in the domain of σ and their expressions within σ are equal (20). A list of match patterns matches if its first pattern matches giving an updated substitution σ' and then the rest of the list of patterns matches using the updated substitution σ' (21). An empty list of match patterns always matches and does not change the substitution (22).

Lowering pattern expressions. The left-hand pattern of a rewrite rule may be arbitrarily nested. As a step towards generating code, the pattern is translated into a sequence of matches (of type $\mathbb{M}$). To avoid more complex variable unification that the target language (Java) is not natively capable of, we introduce new fresh variables when identifiers are reused.

Definition 11 (fresh variable)**.** For a set of used variable names, fresh-var : $\mathbb{V}$ `set` $\to \mathbb{V}$, provides a fresh identifier.

Definition 12 (lower pattern)**.** The relation $(p, \Sigma) \rightsquigarrow (m, v, \Sigma')$ lowers a pattern, $p : \mathbb{P}$, to a match pattern, $m : \mathbb{M}$. The set $\Sigma : \mathbb{V}$ `set` represents the used variable identifiers. The variable $vn : \mathbb{V}$ is the input variable when matching against m.

$$\frac{vn \notin \Sigma}{((\mathsf{VarPat}\ vn), \Sigma) \rightsquigarrow (\mathsf{matchl}\,[\,], vn, \Sigma \cup \{vn\})} \tag{23}$$

$$\frac{v' = \text{fresh-var}\ \Sigma \qquad vn \in \Sigma}{((\mathsf{VarPat}\ vn), \Sigma) \rightsquigarrow ((v' == vn), v', \Sigma \cup \{v'\})} \tag{24}$$

$$\frac{v' = \text{fresh-var}\ \Sigma}{(\mathsf{ConstPat}\ c, \Sigma) \rightsquigarrow (\mathsf{matchv}\ v'\,(\mathsf{ConstMatch}\ c), v', \Sigma \cup \{v'\})} \tag{25}$$

$$\frac{\begin{array}{c} v' = \text{fresh-var}\ \Sigma \\ (p_x, \Sigma \cup \{v'\}) \rightsquigarrow (m_x, v_x, \Sigma_x) \qquad m = \mathsf{matchv}\ v'\,(\mathsf{UnaryMatch}\ op\ v_x) \end{array}}{(\mathsf{UnaryPat}\ op\ p_x, \Sigma) \rightsquigarrow (\mathsf{matchl}\,[m, m_x], v', \Sigma_x)} \tag{26}$$

$$\frac{\begin{array}{c} v' = \text{fresh-var}\ \Sigma \qquad (p_x, \Sigma \cup \{v'\}) \rightsquigarrow (m_x, v_x, \Sigma_x) \\ (p_y, \Sigma_x) \rightsquigarrow (m_y, v_y, \Sigma_y) \qquad m = \mathsf{matchv}\ v'\,(\mathsf{BinaryMatch}\ op\ v_x\ v_y) \end{array}}{(\mathsf{BinaryPat}\ op\ p_x\ p_y, \Sigma) \rightsquigarrow (\mathsf{matchl}\,[m, m_x, m_y], v', \Sigma_y)} \tag{27}$$

$$\frac{\begin{array}{c} v' = \text{fresh-var}\ \Sigma \qquad (p_c, \Sigma \cup \{v'\}) \rightsquigarrow (m_c, v_c, \Sigma_c) \qquad (p_t, \Sigma_c) \rightsquigarrow (m_t, v_t, \Sigma_t) \\ (p_f, \Sigma_t) \rightsquigarrow (m_f, v_f, \Sigma_f) \qquad m = \mathsf{matchv}\ v'\,(\mathsf{CondMatch}\ v_c\ v_t\ v_f) \end{array}}{(\mathsf{CondPat}\ p_c\ p_t\ p_f, \Sigma) \rightsquigarrow (\mathsf{matchl}\,[m, m_c, m_t, m_f], v', \Sigma_f)} \tag{28}$$

When a variable is lowered, if it has not already been used, the match always succeeds, which is represented here by an empty match list (23). Otherwise, a new identifier is generated and an equality check is inserted (24) to ensure multiple occurrences of the same pattern variable match the same expression. Whenever a match operation is generated a fresh identifier is generated to match against (25, 26, 27, 28).

To avoid renaming all identifiers used in a pattern, we want to ensure that newly generated identifiers do not occur in the free variables of a pattern. To accomplish this, all fresh variables have a prefix that cannot occur in a pattern expression. valid-pattern p asserts that no free variables in p have the illegal prefix.[2]

Lemma 13 (valid patterns preserve freshness)**.**

$$\text{valid-pattern } p \implies \forall \Sigma.\ \text{fresh-var } \Sigma \notin \mathcal{L}\ p$$

We relate evaluating lowered pattern expressions to the semantics of matching ground expressions.

Lemma 14 (lower pattern sound)**.** For any valid pattern, p, lowered to a match pattern, m, evaluating m with a substitution that maps v to an expression, e, produces an updated substitution, σ'. The updated substitution, σ', (restricted to only mappings within the free variables of p) equals the substitution produced by match-tree $p\ e$.

$$\frac{\text{valid-pattern } p \qquad \text{finite } \Sigma \qquad (p, \Sigma) \rightsquigarrow (m, v, \Sigma') \qquad [\![m]\!]\, e\, \sigma(v := e) \approx \sigma'}{\text{match-tree } p\ e = \sigma'|_{(\mathcal{L}\ p)}}$$

Exclusion of variables of σ' that are not free variables in p, accounts for additional fresh identifiers introduced by the match pattern.

Lowering rewriting rules. After successfully matching an expression against a pattern, we need to generate a new expression to replace the matched expression based on the result pattern. The variables in the result pattern must be a subset of the variables in the match pattern.

The rule datatype $\mathbb{R}$ represents the steps to perform a rewriting rule. It extends the pattern matching of the $\mathbb{M}$ datatype with a base pattern for returning a result, and constructors that are used to combine rules; discussion of the combinator constructors is deferred to Sect. 4.

$$\mathbb{R} ::= \text{base}\, \mathbb{P} \mid \mathbb{M}\ ?\ \mathbb{R} \mid \mathbb{R}\ \mathit{else}\ \mathbb{R} \mid \mathbb{R}\ ;\ \mathbb{R} \mid \mathit{choice}\ (\mathbb{R}\ \mathit{list}) \tag{29}$$

As base is the only non-recursive constructor, all rules must eventually reach a base constructor. The constructor $m\ ?\ r$ represents that m matches an expression, which is then rewritten via r.

[2] The DSL checks for and warns about illegal prefixes when the rule is encoded.

Definition 15 (lower rule)**.** To generate the rewriting construct, we first generate the matching component from the pattern then combine the matching component with the result pattern.

$$\frac{(\_, \_) \rightsquigarrow (\_, \_) : \mathbb{P} \to \mathbb{P} \to \mathbb{V} \to \mathbb{R} \to \mathbb{B} \qquad (p_m, \emptyset) \rightsquigarrow (m, v, \Sigma)}{(p_m, p_r) \rightsquigarrow (v, m\ ?\ \text{base}\, p_r)}$$

The soundness of lowering rewriting rules relates the semantics of the datatype $\mathbb{R}$ (defined in Sect. 4 by the eval-rules relation) to the rewrite function.

Lemma 16 (lower rule sound)**.**

$$\frac{\mathcal{L}\ p_r \subseteq \mathcal{L}\ p_m \qquad (p_m, p_r) \rightsquigarrow (v, r) \qquad \begin{array}{c}\text{valid-pattern}\ p_m \\ \text{eval-rules}\ r\ e\ [v \mapsto e]\ (\text{Some}\ e')\end{array}}{\text{rewrite}\ p_m\ p_r\ e = e'}$$

Pattern refinement is lifted to the refinement of expressions via lowering. Lemma 16 is extended to give expression refinement.

Lemma 17 (lowered patterns refine expressions)**.**

$$\frac{p_m \sqsupseteq_p p_r \qquad (p_m, p_r) \rightsquigarrow (v, r) \qquad \begin{array}{c}\text{valid-pattern}\ p_m \qquad \mathcal{L}\ p_r \subseteq \mathcal{L}\ p_m \\ \text{eval-rules}\ r\ e\ [v \mapsto e]\ (\text{Some}\ e')\end{array}}{e \sqsupseteq e'}$$

The proof uses the fact that applying rewrite for which the match pattern is refined by the result pattern ($p_m \sqsupseteq_p p_r$) to an expression, e, produces an expression, e', that refines e (Lemma 8).

4 Combining Rules

The next challenge is combining the rewriting rules to form an optimization pass. We propose the following combination structures that allow the rules to be applied in a systematic manner. These combination structures include:

choice ℓ Given a list of rules, ℓ, non-deterministically pick any applicable rule. Non-applicable rules are ignored. If no rules are applicable, the choice fails.

r_1 ***else*** r_2 Apply r_1, if it fails, apply r_2.

r_1 ⨟ r_2 (READ: r_1 *sequential* r_2) Apply r_1, then apply r_2 to the result of the first. If r_1 is not applicable, r_2 is applied to the original expression.

Using these combinators, we can craft optimization passes by combining a set of verified rewriting rules. The combinators are defined in the $\mathbb{R}$ datatype (29).

Naive combination. Consider the example rules: `Identity`, `Evaluate`, `Shift`, and `LeftConst`. We can naively combine these rules by applying any rule that matches using the `choice` operator.

$$\mathit{choice}\ [\mathsf{Identity}, \mathsf{Evaluate}, \mathsf{Shift}, \mathsf{LeftConst}] \tag{30}$$

This combination non-deterministically applies one of the rules that is applicable. This strategy may not result in the most optimal implementation. Consider the expression, $8 * 8$. This expression matches both the `Evaluate` rule and the `Shift` rule. Ideally, we would like to apply the `Evaluate` rule, which results in 64. However, we might equally validly pick the `Shift` rule, which results in $8 \ll 3$.

Strategic combination. Our combinators offer more control over the order in which rules are applied. A more strategic combination of the rewriting rules is to first try to apply the `LeftConst` rule, then, as that produces a multiplication expression which might yet have more applicable rules, apply the remaining rules in a preferred order. For instance, `Evaluate` and `Identity` give the most optimal forms of the expression, otherwise `Shift` gives better runtime performance.

$$\mathsf{LeftConst} \fatsemi (\mathit{choice}\ [\mathsf{Evaluate}, \mathsf{Identity}]\ \mathit{else}\ \mathsf{Shift}) \tag{31}$$

Definition 18 (Rule semantics)**.** The semantics of the $\mathbb{R}$ datatype is defined by the eval-rules relation.

$$\textsf{eval-rules} : \mathbb{R} \to \mathbb{E} \to \mathbb{S} \to \mathbb{E}\ \texttt{option} \to \mathbb{B}$$

$$\textsf{eval-rules}\ (\text{base}\ p)\ e\ \sigma\ (p\ \$\ \sigma) \tag{32}$$

$$\frac{[\![m]\!]\ e\ \sigma \approx \sigma' \qquad \textsf{eval-rules}\ r\ e\ \sigma'\ e'}{\textsf{eval-rules}\ (m\ ?\ r)\ e\ \sigma\ e'} \tag{33}$$

$$\frac{\textsf{eval-rules}\ r_1\ e\ \sigma\ (\textsf{Some}\ e')}{\textsf{eval-rules}\ (r_1\ \mathit{else}\ r_2)\ e\ \sigma\ (\textsf{Some}\ e')} \tag{34}$$

$$\frac{\textsf{eval-rules}\ r_1\ e\ \sigma\ \textsf{None} \qquad \textsf{eval-rules}\ r_2\ e\ \sigma\ e'}{\textsf{eval-rules}\ (r_1\ \mathit{else}\ r_2)\ e\ \sigma\ e'} \tag{35}$$

$$\frac{r \in \mathit{rules} \qquad \textsf{eval-rules}\ r\ e\ \sigma\ (\textsf{Some}\ e')}{\textsf{eval-rules}\ (\mathit{choice}\ \mathit{rules})\ e\ \sigma\ (\textsf{Some}\ e')} \tag{36}$$

$$\frac{\forall r \in \mathit{rules}.\ \textsf{eval-rules}\ r\ e\ \sigma\ \textsf{None}}{\textsf{eval-rules}\ (\mathit{choice}\ \mathit{rules})\ e\ \sigma\ \textsf{None}} \tag{37}$$

$$\frac{\textsf{eval-rules}\ r_1\ e\ \sigma\ (\textsf{Some}\ e') \qquad \textsf{eval-rules}\ r_2\ e'\ \sigma\ e''}{\textsf{eval-rules}\ (r_1 \fatsemi r_2)\ e\ \sigma\ e''} \tag{38}$$

$$\frac{\textsf{eval-rules}\ r_1\ e\ \sigma\ \textsf{None} \qquad \textsf{eval-rules}\ r_2\ e\ \sigma\ e'}{\textsf{eval-rules}\ (r_1 \fatsemi r_2)\ e\ \sigma\ e'} \tag{39}$$

The `base` constructor substitutes the accumulated bindings into the result of the rule (32). If the left-hand side match (m) of a m ? r constructor succeeds, the right-hand side is evaluated passing along the bindings (33). A similar semantics is defined for the `else` operator except the right-hand side is only tried if the left-hand side fails (34, 35). The `choice` operator may evaluate to the successful application of any of its rules (36). If all the rules fail (37) then the `choice` operator fails. If r_1 when applied to e succeeds producing e', then $r_1 \fatsemi r_2$ then applies r_2 to e' to give the resulting e'' (38). If the first rule fails, then the second rule is applied to the original input expression (39).

Soundness. The soundness of combinations has been proven in Isabelle/HOL via an inductive proof over the structure of the rule combinators.

5 Meta-optimizing Rules

To improve the performance of the code generated for the optimizer, we apply a series of meta-optimizations to the generated rules. These optimizations primarily help to combine common conditionals, reducing the number of branches in the generated code and the number of conditions evaluated when performing the optimisations. Meta-optimizations must preserve the semantics of the original rules. That is, the meta-optimized rules must be equivalent to the original rules.

Definition 19 (rule equivalence)**.** A rule, r', is equivalent to a rule, r, if for all substitutions, σ, the evaluation of r and r' against σ produces the same result.

$$r' \doteq_r r = (\forall \sigma\ e\ e'.\ \mathsf{eval\text{-}rules}\ r\ e\ \sigma\ e' = \mathsf{eval\text{-}rules}\ r'\ e\ \sigma\ e')$$

To simplify the proof effort for individual meta-optimization rules, we define an Isabelle/HOL locale, rule-transform, to perform and verify the bulk of the transformation process. The locale is parameterized by a function, transform, that given some rule, r, optionally returns a new rule, r' that is equivalent to r but has been optimized in some way. If the meta-optimization does not apply to the given rule, then transform returns None.[3]

Definition 20 (meta-optimization locale)**.**

$$\begin{aligned}
&\mathsf{rule\text{-}transform} : (\mathbb{R} \to \mathbb{R}\ \texttt{option}) \to \mathbb{B}\\
&\mathsf{rule\text{-}transform}\ \mathsf{transform} \equiv \forall r\ r'.\\
&\quad (\mathsf{valid\text{-}rules}\ r \wedge \mathsf{transform}\ r = \mathsf{Some}\ r' \implies r' \doteq_r r) \wedge \qquad (40)\\
&\quad (\mathsf{valid\text{-}rules}\ r \wedge \mathsf{transform}\ r = \mathsf{Some}\ r' \implies \mathsf{valid\text{-}rules}\ r') \qquad (41)
\end{aligned}$$

[3] The locale is additionally parameterized by a measure function, which must be recursively decreasing and must decrease with respect to the transform function.

The important property to satisfy is that, given any valid rule, r, if the optimization function, transform, returns a new rule, r', then r' is equivalent to r (40). The valid-rules function ensures that the valid-pattern property holds for all patterns contained within the rules. Each rule must be valid after optimization (41), allowing meta-optimization to be composed.

Definition 21 (apply transform)**.** The application of a meta-optimization is defined by r-transform. The meta-optimization function, transform, is repeatedly applied to sub-rules until no further optimization is possible.

$$\text{r-transform} : \mathbb{R} \to \mathbb{R}$$

$$\text{r-transform}\ r = \begin{cases} \text{r-transform}\ r' & \text{if transform } r = \text{Some } r' \\ \text{base}\ p & \text{if } r = \text{base}\ p \\ m\ ?\ (\text{r-transform}\ r') & \text{if } r = m\ ?\ r' \\ (\text{r-transform}\ r_1)\ \mathit{else}\ (\text{r-transform}\ r_2) & \text{if } r = r_1\ \mathit{else}\ r_2 \\ (\text{r-transform}\ r_1)\ \fatsemi\ (\text{r-transform}\ r_2) & \text{if } r = r_1\ \fatsemi\ r_2 \\ \mathit{choice}\ (\text{map r-transform rules}) & \text{if } r = \mathit{choice}\ \text{rules} \end{cases}$$

If the meta-optimization can be applied, it will be applied repeatedly until no further meta-optimization is possible. Otherwise, the function applies the optimization recursively through the combination structure. Note that the above cases are applied in sequence.

Lemma 22 (meta-optimization soundness)**.** Given a function that optimizes a rule, transform, if the locale, rule-transform transform, is instantiated, then the rule application function, r-transform, produces optimized rules that are equivalent.

$$\frac{\text{rule-transform transform}}{(\text{r-transform}\ r) \eqsim_r r}$$

The rule-transform locale is instantiated for each optimization function presented.

Pre-processing rules. The generation of match-patterns from rewriting rules can produce cluttered results. We apply a series of pre-processing rules that are designed to clean match-patterns before optimization. For example, empty lists of matches, (i.e., matchl $[]$) are eliminated and lists of matches are promoted to sequences of conditionals, (i.e., matchl $[a, b]$? r becomes a ? b ? r), to expose further optimizations.

Meta-optimizations. We consider meta-optimisations for combining *else* conditions and combining *choice* conditions.

Combining `else` *conditions.* We can combine match-patterns that share the same condition. This reduces the number of instructions required to evaluate the condition and avoids repeated evaluations of the same condition.

$$\begin{aligned}&\textsf{combine-else} : \mathbb{R} \to \mathbb{R}\ \texttt{option}\\ &\textsf{combine-else}\ r = \begin{cases}\textsf{Some}\,(m\ ?\ (r_1\ \mathit{else}\ r_2)) & \text{if } r = ((m\ ?\ r_1)\ \mathit{else}\ (m\ ?\ r_2))\\ \textsf{Some}\,(m\ ?\ r') & \text{if } r = (m\ ?\ (m\ ?\ r'))\\ \textsf{None} & \text{otherwise}\end{cases}\end{aligned}$$

As an example, the rule, $\textsf{Evaluate}_l$ *else* $\textsf{LeftConst}_l$, using the lowered forms of $\textsf{Evaluate}_l$ (3) and $\textsf{LeftConst}_l$ (5), can be optimized by factoring out the conditions as they share many of the same initial conditions. In the below derivation the actual matches have been abbreviated as follows: $m_1 = \textsf{match}\,(x * y)$, $m_2 = \textsf{matchv}\ x\,(\textsf{const}\ c)$, $m_3 = \textsf{matchv}\ y\,(\textsf{const}\ d)$, $r_1 = \textsf{const}\,(c * d)$, $m_4 = \neg(\mathit{IsConst}\ y)$, and $r_2 = y * \textsf{const}\ c$.

$$\dfrac{\dfrac{\dfrac{\textsf{Evaluate}\ \mathit{else}\ \textsf{LeftConst}}{(m_1\ ?\ m_2\ ?\ m_3\ ?\ r_1)\ \mathit{else}\ (m_1\ ?\ m_2\ ?\ m_4\ ?\ r_2)}\ \textsc{unfold}}{m_1\ ?\ ((m_2\ ?\ m_3\ ?\ r_1)\ \mathit{else}\ (m_2\ ?\ m_4\ ?\ r_2))}\ \textsf{combine-else}}{m_1\ ?\ m_2\ ?\ ((m_3\ ?\ r_1)\ \mathit{else}\ (m_4\ ?\ r_2))}\ \textsf{combine-else}$$

Combining `choice` *conditions.* In a similar vein, we can combine match-patterns that share the same condition and are combined using the `choice` operator. This optimization can be more aggressive than `else` condition combination as the non-determinism of the `choice` operator means that the order of the match-patterns does not matter. We can therefore pull out all match-patterns that share the same condition and combine them.

The two helper functions, join-common and join-distinct, are complementary functions used to extract match-patterns that share the same condition, m, and those that do not. The join-common function also drops the matching condition from the match-patterns.

$$\begin{aligned}&\textsf{join-common} : \mathbb{M} \to \mathbb{R}\ \texttt{list} \to \mathbb{R}\ \texttt{list}\\ &\textsf{join-common}\ m\ rs = \textsf{map-filter}\ (\lambda r.\ \textbf{if}\ r = m\ ?\ r'\ \textbf{then}\ \textsf{Some}\ r'\ \textbf{else}\ \textsf{None})\ rs\\ &\textsf{join-distinct} : \mathbb{M} \to \mathbb{R}\ \texttt{list} \to \mathbb{R}\ \texttt{list}\\ &\textsf{join-distinct}\ m\ rs = \textsf{map-filter}\ (\lambda r.\ \textbf{if}\ r = m\ ?\ r'\ \textbf{then}\ \textsf{None}\ \textbf{else}\ \textsf{Some}\ r)\ rs\end{aligned}$$

The combine-choice function separates the common and distinct match-patterns, grouping the common match-patterns under a single common condition.

$$\begin{aligned}&\textsf{combine-choice}\,(\mathit{choice}\ ((m\ ?\ r){\cdot}rs)) =\\ &\quad \textsf{Some}\,(\mathit{choice}\ (m\ ?\ \mathit{choice}\ (r{\cdot}(\textsf{join-common}\ m\ rs)))){\cdot}\textsf{join-distinct}\ m\ rs)\end{aligned}$$

An example of this optimization is shown below. Note that standard compiler optimizations of the generated code would not perform this optimization as the operations are re-ordering in a way that, in the general case, changes the semantics of the code but is safe in this context.

$$\begin{aligned}&\mathsf{combine\text{-}choice}\,(\mathit{choice}\,[\mathit{Evaluate}, \mathit{Shift}, \mathit{LeftConst}]) =\\ &\mathsf{match}\,(x * y)\ ?\\ &\mathit{choice}\ \left[\begin{array}{l}\mathsf{matchv}\ x\,(\mathsf{const}\ c)\ ?\ \mathit{choice}\ \left[\begin{array}{l}\mathsf{matchv}\ y\,(\mathsf{const}\ d)\ ?\ \mathsf{const}\,(c * d),\\ \mathsf{cond}\ \neg(\mathit{IsConst}\ y)\ ?\ y * \mathsf{const}\ c\end{array}\right],\\ \mathsf{matchv}\ y\,(\mathsf{const}\ c)\ ?\ \mathsf{cond}\,(\mathit{Power2}\ c)\ ?\ x \ll (\mathsf{const}\,(\mathit{log2}\ c))\end{array}\right]\end{aligned}$$

The translation from this representation to programming language code (such as a subset of Java) with the same structure is straightforward.

Composition. Meta-optimizations can be composed to form a single meta-optimization function. The composition of meta-optimizations applies each meta-optimization in sequence, passing along the result to the next meta-optimization. Composition of meta-optimizations can be shown generally for any composition of meta-optimization functions that satisfy the locale.

Lemma 23 (meta-optimization composition sound)**.** Our meta-optimization functions satisfy the locale, therefore, the composition of these meta-optimizations is sound.

$$(\mathsf{combine\text{-}choice} \bullet \mathsf{combine\text{-}else})\ r = \mathsf{Some}\ r' \implies r' \mathrel{\hat{=}_r} r$$

6 Related Work

Representing optimizations as rewrites has a long history. Sharlit [9] and Gospel [14] aim to provide a general framework for generating optimization implementations without a focus on verification. Cobalt [4] and Rhodium [5] aim to automatically prove soundness of optimizations, however, they both utilize Whirlwind [2] as their execution engine which compiles the optimizations to C, which is not amenable to the GraalVM compiler.

Representations of rewriting rules have also been studied extensively. The most notable example is the Stratego language [11], although many other languages exist [10]. General purpose rewriting languages are flexible but lack the control we sought for optimizing the generated code. Initially an implementation of Stratego was used to represent the rewriting rules, however, we found that the flexibility produced generated code that was difficult to optimize. We therefore decided to use a more restricted representation of rewriting rules.

The closest related work is [6] which develops infrastructure for deriving implementations of verified optimizations. Their work produces implementations that "are sometimes faster, and at most 40% slower, than hand-written counterparts". By comparison, we can express fewer types of optimizations, but maintain implementations that are near identical, including in performance, to the hand-written source.

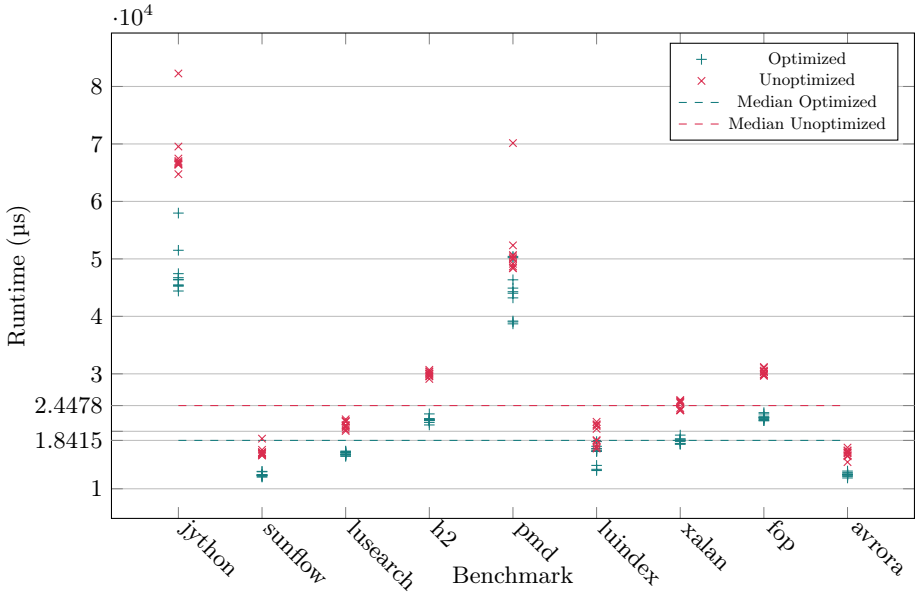

Fig. 3: Total runtime spent in optimization phase for DaCapo 9.12-MR1-git+2baec49 benchmarks.

7 Evaluation

We evaluate the effectiveness of our approach with a modest collection of 46 optimization rules, combined naively with the `choice` operator. These rules are derived from the GraalVM compiler.

First, we evaluate the effectiveness in terms of lines of generated code. Code generated for inclusion in the GraalVM compiler needs to be efficient. For our 46 rules prior to meta-optimization, 542 lines of generated code are produced. When the meta-level optimizations are applied, the generated code is reduced to 315 lines. This is a reduction of around 42%.

Second, we evaluate the effectiveness in performance. We implemented a new optimization phase in the GraalVM compiler that applies the generated 46 optimizations using the generated code. The optimization phase is naively implemented and simply applies the optimizations to all nodes in the graph during each pass. Fig. 3 shows the cumulative runtime spent in this new phase with our meta-optimizatons applied versus without. Each DaCapo benchmark [1] is run 10 times, optimized and unoptimized. While the performance is not a significant improvement, it is rarely worse than without the meta-optimizations.

Finally, we wish to know whether the meta-optimizations add value beyond allowing the GraalVM compiler itself to optimise code generated without meta-optimisation. We hypothesize that the more relaxed `choice` operator semantics allows the meta-level optimizations to perform more aggressive optimizations by

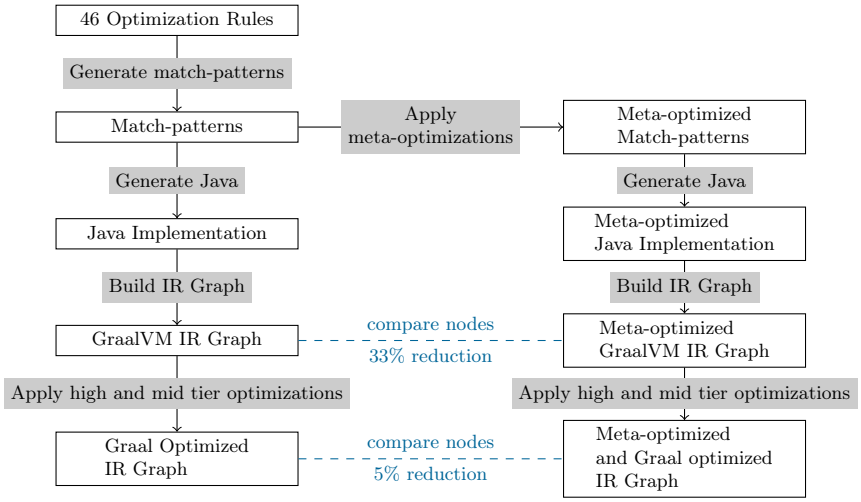

Fig. 4: Process for comparing impact of meta-optimizations.

re-ordering conditions. We evaluate this hypothesis by optimizing the generated code using the GraalVM compiler and comparing the resulting method graphs. This process is illustrated in Fig. 4. After the same standard GraalVM optimizations are applied to the meta-level optimized and unoptimized generated code, the method graphs do differ. The meta-level unoptimized graph consists of 3548 nodes (including 299 `If`, 48 `Return`), while the optimized graph consists of 3377 nodes (including 288 `If`, 43 `Return`). This is a reduction of around 5%, compared with a reduction of 33% when no general purpose optimizations are applied. This appears to confirm our hypothesis that the meta-level optimizations are able to perform more aggressive optimizations.

8 Conclusion

We have presented an approach for translating compiler optimizations expressed as rewriting rules into implementations in Java. The generated implementations are optimized by meta-level optimizations that produce more efficient code suitable for inclusion in a compiler. The meta-level optimizations are additionally able to perform more aggressive optimizations than the general purpose optimizations available in the compiler. Note that the meta-optimizations are optimizing the compiler, not optimizing the programs compiled by the compiler. We have implemented this approach in Isabelle/HOL and evaluated it on a collection of 46 optimization rules derived from the GraalVM compiler. The approach has been generalized via locales which allows instantiating the rewriting rules with different Isabelle types and reusing the meta-level optimizations. All steps in the process have been verified in Isabelle/HOL, except the final translation to the implementation programming language (Java in our case) and this translation is

a simple one-to-one syntactic translation. Because generation of optimiser code from a set of rewriting rules is automated, adding a new optimisation can be achieved by adding a rewriting rule and verifying that it preserves the semantics of the expression.

References

1. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. pp. 169–190. ACM Press, New York, NY, USA (Oct 2006). https://doi.org/10.1145/1167473.1167488
2. Chambers, C.: Whirlwind. https://wasp.cs.washington.edu/wasp_whirlwind.html (1998)
3. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. p. 237–247. PLDI '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/155090.155113
4. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. p. 220–231. PLDI '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/781131.781156
5. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 364–377. POPL '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1040305.1040335
6. Li, J.M., Appel, A.W.: Deriving efficient program transformations from rewrite rules. Proceedings of the ACM on Programming Languages **5**, 1–29 (2021). https://doi.org/10.1145/3473579
7. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
8. Oracle: GraalVM: Run programs faster anywhere (2020), https://github.com/oracle/graal
9. Tjiang, S.W.K., Hennessy, J.L.: Sharlit—a tool for building optimizers. SIGPLAN Not. **27**(7), 82–93 (Jul 1992). https://doi.org/10.1145/143103.143120
10. Visser, E.: A survey of strategies in rule-based program transformation systems. Journal of Symbolic Computation **40**(1), 831–873 (2005). https://doi.org/10.1016/j.jsc.2004.12.011
11. Visser, E., Benaissa, Z.e.A., Tolmach, A.: Building program optimizers with rewriting strategies. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming. pp. 13–26. ACM (1998). https://doi.org/10.1145/289423.289425

12. Webb, B.J., Hayes, I.J., Utting, M.: Verifying term graph optimizations using Isabelle/HOL. In: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 320–333. CPP 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3573105.3575673
13. Webb, B.J., Utting, M., Hayes, I.J.: A formal semantics of the GraalVM intermediate representation. In: Hou, Z., Ganesh, V. (eds.) Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 12971, pp. 111–126. Springer International Publishing, Cham (Oct 2021). https://doi.org/10.1007/978-3-030-88885-5_8
14. Whitfield, D.L., Soffa, M.L.: An approach for exploring code improving transformations. ACM Trans. Program. Lang. Syst. **19**(6), 1053–1084 (Nov 1997). https://doi.org/10.1145/267959.267960