Verifying and Improving Halide's Term Rewriting System with Program Synthesis

JULIE L. NEWCOMB, University of Washington, USA
ANDREW ADAMS, Adobe Research, USA
STEVEN JOHNSON, Google, USA
RASTISLAV BODIK, University of Washington, USA
SHOAIB KAMIL, Adobe Research, USA

Halide is a domain-specific language for high-performance image processing and tensor computations, widely adopted in industry. Internally, the Halide compiler relies on a term rewriting system to prove properties of code required for efficient and correct compilation. This rewrite system is a collection of handwritten transformation rules that incrementally rewrite expressions into simpler forms; the system requires high performance in both time and memory usage to keep compile times low, while operating over the undecidable theory of integers. In this work, we apply formal techniques to prove the correctness of existing rewrite rules and provide a guarantee of termination. Then, we build an automatic program synthesis system in order to craft new, provably correct rules from failure cases where the compiler was unable to prove properties. We identify and fix 4 incorrect rules as well as 8 rules which could give rise to infinite rewriting loops. We demonstrate that the synthesizer can produce better rules than hand-authored ones in five bug fixes, and describe four cases in which it has served as an assistant to a human compiler engineer. We further show that it can proactively improve weaknesses in the compiler by synthesizing a large number of rules without human supervision and showing that the enhanced ruleset lowers peak memory usage of compiled code without appreciably increasing compilation times.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: term rewriting system, verification, synthesis

ACM Reference Format:
Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and Improving Halide's Term Rewriting System with Program Synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 166 (November 2020), 28 pages. https://doi.org/10.1145/3428234

1 INTRODUCTION

To compile an image processing pipeline written in the Halide language, the compiler must perform a variety of analyses of the pipeline's properties. For example, if the user marks a loop to be fully unrolled, the compiler must infer a constant upper bound for the extent of the loop. If the user marks a loop as parallel, the compiler must prove the absence of data races. These analyses also affect performance more than in most compilers. In Halide, the compiler infers loop bounds and allocation sizes. If these are overestimated, the generated code may perform an amount of wasted

Authors' addresses: Julie L. Newcomb, University of Washington, USA, newcombj@cs.washington.edu; Andrew Adams, Adobe Research, USA, andrew.b.adams@gmail.com; Steven Johnson, Google, USA, srj@google.com; Rastislav Bodik, University of Washington, USA, bodik@cs.washington.edu; Shoaib Kamil, Adobe Research, USA, kamil@adobe.com.

2475-1421/2020/11-ART166
https://doi.org/10.1145/3428234

work sufficient to alter the computational complexity of the algorithm. These analyses all depend critically on the quality of Halide's expression simplifier. In fact, Halide relies so heavily on its simplifier that restricting it to mere constant-folding causes a geomean 5.1× increase in compilation times and a 26.4× increase in runtimes across Halide's benchmark suite.

This simplifier must balance three key criteria:

- **Completeness:** It must work in the theory of integers, which is undecidable, and make use of operations such as Euclidean division and maximum/minimum which can be especially difficult for automated reasoning. Although any solver in this theory is necessarily incomplete, but in general, the compiler can generate higher-performing code as the simplifier becomes more powerful.
- **High performance:** However, the simplifier is called many thousands of times over the course of a single compilation, and so requires high performance and low memory usage.
- **Determinism:** The compiler must always return the same result for the same program, regardless of what platform runs the compiler, so the simplifier must be deterministic. This is a hard requirement.

Halide addresses this problem with a *term rewriting system*. Using a custom algorithm that greedily applies rules in a fixed order and keeps only the expression being currently rewritten as state, the term rewriting system (TRS) provides the performance and determinism that the compiler requires. This algorithm scales well in terms of the number of rules in the TRS, so Halide developers continue to improve and increase its power by refining the ruleset and adding new rules by hand.

However, maintaining a TRS by hand presents significant challenges. Rules are carefully inspected and fuzz-tested, but are not formally proven sound. The rule application algorithm itself is not guaranteed to terminate, so changing, adding, or reordering rules may result in cycles on untested inputs. Debugging the system can be difficult, since it is not clear which rule or combination of rules is responsible for undesired behavior. The TRS is not complete enough to address all compiler queries (for example, it may fail to compute tight bounds on intermediate arrays, leading to over-allocations). Finally, although the overarching goal of the TRS is to simplify expressions, "simplify" is an imprecise notion. If developers are not familiar with the entire ruleset, or the full variety of applications of the simplifier, they may inadvertently add a rule that makes the simplifier worse for some of its usecases.

In this work, we show how techniques based on formal methods can help developers maintain a complex term rewriting system, first by providing formal guarantees of soundness and termination, then by growing the ruleset to increase the simplifier's power while ensuring the performance and determinism required by a compiler in industrial use. We further demonstrate a strategy for growing a simplifier's power when completeness is impossible, by synthesizing new rules that operate in the subset of the theory encountered during real-world compilation.

First, we formally verify the existing ruleset, demonstrating that proofs of soundness are possible despite the lack of a decision procedure for our theory. We model the Halide expression language and verify the ruleset via an SMT solver that can prove about 88% of the existing ruleset; we prove the remaining rules by hand via the proof assistant Coq [Coq Development Team 2019]. We find four unsound rules, as well as several rules which could be made more general by relaxing their predicate guards. The Halide developers changed the language's semantics for division during this work; we reran our verification process to find 44 rules which were incorrect under the new semantics, demonstrating the usefulness of our technique.

Next, we define the meaning of simplification in the context of the Halide TRS by formalizing what it means for a rule to usefully modify an expression. While many notions of "simpler" are possible, we encode the specific criteria for Halide expressions by defining an *ordering* over the

left-hand and right-hand terms of the rules in the TRS that captures the intentions behind the rewrites. This ordering means that every local change caused by the application of a rewrite rule moves the expression in some useful direction. By composing several orders lexicographically, we can express many different intuitions about what makes an expression simpler in a defined priority: we may want to remove as many vector operations as possible, then reduce the overall size of the expression, and so on. Usefully, this order also provides a termination guarantee: if every rewrite leaves the rewritten expression strictly less in terms of our order, then it is impossible for any sequence of rewrites to form cycles, so the rewrite algorithm must terminate. This is not a hypothetical issue: Halide developers have previously observed non-termination when adding rules. We devise an ordering that fits as many of the existing rules as possible, and then remove the few rules that do not obey this ordering, proving that the TRS will now always terminate. Maintaining this order as an invariant over any future rules means any additions to the ruleset will not undo progress made by existing rules; it also means rules can be freely removed or reordered without affecting termination.

Having guaranteed soundness and termination, we increase the solving power of the TRS. We do this through *synthesis* of new rules. Even given the constraints imposed by the termination order, the space of equivalences in the Halide expression language is infinitely large. How do we choose which rules to add? We observe that there is some bias on the distribution of expressions seen by the compiler on realistic inputs over the full expression space. We take advantage of this bias by gathering expressions from realistic compilations on which the current TRS is "stuck" and can make no further progress. We choose candidate left-hand sides for rules from this corpus and synthesize equivalent right-hand terms that obey the termination order. These expressions frequently contain constants, so we generalize rules by replacing constants with fresh variables and synthesizing predicate guards that indicate when it is safe to apply the rule. Our synthesis procedure finds large numbers of useful rules without human oversight; although the existing compiler is mature and well-tuned for our suite of benchmarks, we show some performance gains without increases in compilation time when our newly synthesized suite of rules is added to the TRS.

In the rest of the paper, we first provide some background on term rewriting systems and the Halide rewriting algorithm (Section 2). We then describe our approach to verifying rule correctness (Section 3.1) and discuss the relation between LHS and RHS terms that guarantees termination (Section 3.2). Once semantics and the termination property have been formalized, we present our synthesis algorithm (Section 4). We demonstrate the effectiveness of our approach through several experiments and case studies (Section 5), with observations on the ways Halide developers have been able to integrate our techniques into their workflows. Finally, we close in Sections 6–8 with a discussion of our current limitations and a review of related work.

2 TERM REWRITING IN HALIDE

The Halide compiler contains a term rewriting system composed of over a thousand rules, operating over the space of Halide expressions[1]. The language of Halide expressions operates over vectors and scalars of integers, booleans, and real values. However, in this work we concentrate on the TRS as it applies to integer and boolean values, for both vectors and scalars, because the most important uses of the TRS within the compiler apply to these types. In this section, we give some background on term rewriting systems. We then describe how the Halide compiler uses the TRS and the design decisions that motivate the custom rewriting engine, as well as the scope of our work.

[1]See Supplemental Material for the full Halide expression grammar

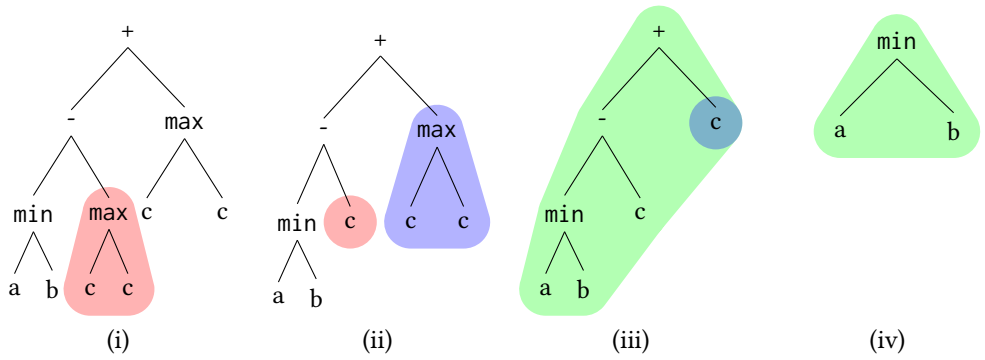


Fig. 1. We demonstrate the Halide rewriting algorithm using a TRS $R = \{\texttt{max}(x, x) \rightarrow_R x, (x - y) + y \rightarrow_R x\}$ and an expression $\texttt{min}(a, b) - \texttt{max}(c, c) + \texttt{max}(c, c)$. The algorithm attempts to simplify all subtrees bottom up; here, no rule applies to $\texttt{min}(a, b)$ so it is not changed. Next (i), rule 1 rewrites $\texttt{max}(c, c)$ to c. No rule applies to $\texttt{min}(a, b) - c$, so we move to the rightmost subtree and rewrite again (ii) to obtain c from $\texttt{max}(c, c)$. Finally, we consider the entire tree $\texttt{min}(a, b) - c) + c$ (iii) and apply rule 2 to produce $\texttt{min}(a, b)$. No rules match this expression, so we are left with $\texttt{min}(a, b)$ (iv).

2.1 Term Rewriting Systems

Term rewriting systems [Gorn 1967] are sets of *rewrite rules* used to transform expressions into a new form. Such systems are widely used in theorem proving [Baader and Nipkow 1999] and abstract interpretation [Cousot and Cousot 1977, 1979].

Terms are defined inductively over a set of variables V and a set of function symbols Σ. Every variable $v \in V$ is a term, and for any function symbol $f \in \Sigma$ with arity n and any terms $t_1, ..., t_n$, the application of the symbol to the terms $f(t_1, ..., t_n)$ is also a term. (Constants are considered zero-arity functions.) We refer to the set of terms constructed from the variables V and the function symbols Σ as $T(\Sigma, V)$.

A *rewrite rule* is a directed binary relation $l \rightarrow_R r$ such that l is not a variable, and all variables present in r are also present in l (i.e., $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$). A set of rewrite rules is called a *term rewriting system*.

Consider a set of terms $T(\Sigma, V)$ such that $\Sigma = \{\clubsuit, \diamond\}$ and V is an infinite set of variables. Let the term rewriting system R consist of a single rule:

$$R = \{x_1 \clubsuit x_2 \rightarrow_R x_1 \diamond x_2\}$$

We use R to rewrite the term

$$(y_1 \diamond y_1) \clubsuit (y_2 \clubsuit y_3)$$

The first step is matching; we find a substitution that will unify the left-hand side (LHS) of the rule with the term we are rewriting. Here, one possible substitution is:

$$\{x_1 \mapsto (y_1 \diamond y_1), x_2 \mapsto (y_2 \clubsuit y_3)\}$$

We then apply this substitution to the right-hand side (RHS) of the rule to obtain the rewritten version of the original term:

$$(y_1 \diamond y_1) \diamond (y_2 \clubsuit y_3)$$

2.2 Uses of the TRS in Halide

While the Halide compiler makes use of the TRS in numerous ways, the most important applications of the TRS are its uses as a fast simplifier and as a proof engine. In many parts of the compiler, the TRS is used to rewrite expressions into simpler forms, which are easier for the compiler to reason about, and result in less code being generated for LLVM to consume at the backend. Most importantly, the compiler uses the TRS to simplify expressions into constants or expressions that are monotonic with respect to loop bounds; these simplifications are core to Halide's ability to generate drastically different loop nests for different schedules.

For example, consider the simple two-stage imaging pipeline $g(x) = f(x-1) + f(x) + f(x+1)$. Halide enables programmers to fuse the computation of f into g at an arbitrary granularity using the `compute_at` scheduling directive. This requires Halide to automatically reason about which region of f is required for a specific sub-region (or tile) of g, using interval arithmetic over symbolic values for the size of a tile of g. For a tile size of 8, a tile of g is the region `[g.tile_min, g.tile_min+7]`; the region of f required is `[g.tile_min-1, g.tile_min+8]`; and the number of values of f to compute is then `g.tile_min + 8 + 1 - (g.tile_min - 1)`. If the TRS can determine this is a static value of 10, the Halide compiler can then safely perform transformations requested by the user. In this case, the compiler can use stack memory instead of inserting a dynamic allocation; or the loop can be completely unrolled; the loop can be vectorized; or f can be mapped to GPU threads (since a single threadblock must have a compile-time-known size). More generally, this kind of region analysis operates most effectively when the expressions are monotonic in the loop bounds; otherwise, interval arithmetic can result in vast overestimates of required regions. These simplifications are essential for the compiler to work, and are usually not as simple as this example.

The rules for simplifying to perform cancellations and ensure monotonicity are incredibly important for compiler performance. When we disabled all but the constant-folding rules to measure the importance of the simplifier, it was the absence of these specific rules that caused the (26.4×) slow-down mentioned in Section 1. Without these rules, Halide is useless for high-performance image processing.

The use as a proof engine occurs when the compiler must prove properties about the code in order to guarantee the correctness of specific transformations or the relationships between bounds of different loops or producer-consumer relationships. In such cases, the compiler constructs an expression that must be true or false in order to guarantee correctness, then applies the TRS to see if the expression simplifies to a single boolean value.

For example, Halide uses Euclidean division, which rounds according to the sign of the denominator. Lowering this to code requires emitting several instructions, which can be slower than native division. When the compiler can statically prove the signs of the numerator and denominator, in some cases the code can be replaced by native division or even a different instruction altogether. For example, for an expression `x / max(y, 1)` the compiler will try to prove $0 < \texttt{max}(y, 1)$. The TRS first invokes a rule to transform this to $0 < y \;||\; 0 < 1$, which then is transformed to true (since the second clause is always true). Thus, the compiler is able to replace Euclidean division with machine division.

TRS failures have adverse results on the compiler, making it unpredictable and difficult for programmers to use. When the TRS fails to properly simplify an expression or prove a property, the consequences include:

- Insufficiently tight bounds on loops and allocations, which may result in runtime failures (e.g. due to memory overallocation) or performance issues;
- Failure of the compiler to apply optimizations, also resulting in slow performance;

- Dynamic checks in the generated code for properties that could have been proven at compile time, leading to slower code;
- Compilation failures, when the compiler is unable to correctly produce code even though the properties required hold, or when the proof engine itself crashes or loops infinitely.

Thus, correctness and generality of the TRS are essential to make the compiler robust and able to generate fast code.

2.3 Why a Custom Algorithm?

In a term rewriting system, a single rule may be able to match an input expression in multiple ways, and there may be multiple rules in the ruleset which could be used to rewrite the expression. A term rewriting algorithm might choose one of many alternatives and later backtrack if it turns out not to be fruitful; it might make use of heuristics to choose a next step; it might exercise all the alternatives and keep the results in equivalence classes, as in an egraph. The Halide term rewriting algorithm keeps only one expression in state and applies rules greedily, in a fixed priority. This is very fast and requires very little memory; the tradeoff is that the algorithm may pick the "wrong" rule and have no way of undoing that decision. Since the rewriter is invoked thousands of times with each call of the compiler, it chooses to sacrifice some solving power in exchange for performance.

2.3.1 Halide's Custom TRS Algorithm. The Halide term rewriting algorithm simplifies an input expression in a depth-first, bottom-up traversal of the expression DAG. At each node, it uses the root node to pick a list of rules, then attempts to match the subtree expression with the rule LHSs in a fixed priority. Matching is performed purely syntactically, using C++ template metaprogramming. Halide rewrite rules contain special metavariables, called *symbolic constants*, that can match only with constant values; all other variables can match any subterm as usual. When a match is found, the algorithm rewrites the subtree expression using the RHS of that rule, and then attempts to simplify the subtree expression again. If no rule matches the subtree, the traversal continues; when the entire expression cannot be simplified further, the rewritten expression is returned. See Figure 1 for a worked example.

The rewrite rules optionally contain a compile-time predicate guard. These guards contain only symbolic constants[2]; when the LHS of a rule matches an expression, its guard is evaluated and only if it is true will the rewrite be applied.

Halide rewrite rules are applied in a fixed priority, organized so that the TRS first attempts very basic rules such as constant folding, then tries more specific rules before more general ones. (We do not evaluate the current rule priority in this work.)

Associativity and commutativity laws are particularly troublesome for term rewriting systems. For one expression e, the number of semantically equivalent expressions grows exponentially in terms of the number of AC operations e contains. Some term rewriting systems perform a full *AC matching* step during rewriting. Halide's TRS does not perform this matching, but instead includes multiple AC variations of rules. However, a small number of Halide's rewrite rules have the effect of canonicalizing some commutative expressions. (For example, if a commutative expression has a multiplication as its first operand and a subtraction as its second operand, a rule will switch their positions.) These rules are all early in the application priority, so later rules can rely on expressions having a quasi-canonical form.

2.3.2 Why Not Z3? Given that we make use of the Z3 solver [De Moura and Bjørner 2008] for both verification and synthesis, it is natural to ask why Halide could not simply call Z3 for simplification.

[2]Existing rules sometimes have predicates that check if non-constant variables can be shown to have certain properties at compile time, but these are expensive and used sparingly.

Table 1. We compare the performance of Z3 and the Halide TRS in proving a set of 4304 expressions gathered from realistic compiler output. Note that expressions in the "not proven" column include expressions that are true but not found to be so by the solvers as well as expressions that are not true.

Tool	Runtime	Proven expressions	Not proven
Z3	7m29s	1125	3179
Halide TRS	2s	885	3419

Z3 is the product of extensive development and is a very powerful, general-purpose solver. However, the Halide term rewriting system has a few key properties that Z3 does not: deterministic output, low memory and compute requirements, and domain-specific optimizations.

As discussed above, the Halide compiler must return the same schedule every time the same pipeline is run. Z3 can fix a random seed, but long-running queries may complete on a more powerful server while timing out on a different machine.

While the Halide algorithm is less powerful than Z3, its deterministic, greedy rule application strategy gives it a smooth performance curve, whether it succeeds or fails in simplifying an input expression. A solver like Z3 tends to give very good performance most of the time but gets bogged down in difficult cases, requiring the use of timeouts. The Halide algorithm "fails fast": on an input expression which does not match any rule, the Halide algorithm will complete in time linear to the size of the expression, taking on the order of one CPU cycle per term in the expression per rule in the TRS. To demonstrate this performance tradeoff, we gathered 4304 expressions from queries the Halide compiler made when compiling realistic pipelines, including provably true expressions and expressions that are not provably true. Z3 could prove more expressions true (within a 60 second timeout), but was starkly less performant. As shown in Table 1, Z3 took over 7 minutes to check the set of expressions while the Halide TRS took just 2 seconds. This set of expressions is much smaller than the number of calls the compiler makes to the rewriter in compiling a single pipeline.

Because the Halide algorithm at every step chooses one rule to apply to the single expression it is working on, it scales well in terms of the number of rules in the TRS. See Section 5.3 for an evaluation of the effects of adding newly synthesized rules on the performance of the compiler.

Finally, although Z3 can simplify expressions, simply reducing the size of an expression is not necessarily the goal for the compiler. For example, gathering like terms in some cases can actually prevent Halide or LLVM optimizations from applying. The Halide rewriter uses a domain-specific strategy to guide expressions into more optimizable forms and can be changed or tuned as needed if further optimizations are discovered.

2.4 Completeness of the Halide TRS

We know by observation that the current Halide TRS cannot prove certain equalities that are in fact true, or reduce certain expressions that can be further simplified. Our goal is to learn new rules that would strengthen the TRS and allow it to make further progress on these "stuck" expressions. This goal seems similar to that of *completion*, which constructs a decision procedure through syntatic rewrites for a set of identities. We do not use completion directly, although our synthesis algorithm could be considered analogous to completion in some ways.

In the standard Knuth-Bendix completion algorithm [Knuth and Bendix 1983], we take a finite set of identities E and a reduction order $>$ on terms as inputs; if successful, the algorithm will return a finite, convergent set of rules R that is equivalent to E. The algorithm may also fail, or fail to terminate. At each step the algorithm maintains a set of identities and a set of rules, both of which can be updated; the algorithm may transform an identity into a new rule, find a new identity

as a consequence of the ruleset, or use the present ruleset to refine either an identity or a rule. The algorithm runs until the ruleset has converged; specific implementations may use some conditions under which to terminate with a failure.

No finite set of identities exists for the theory of integers. We could fix a set of identities to use in a completion procedure, but choosing these axioms is a non-trivial task. One issue is that the theory contains axioms such as commutativity; an identity such as $x + y \equiv y + x$ cannot be oriented by any possible reduction order, so our completion algorithm cannot make use of this fact. Another is that any sufficiently powerful set of identities would almost certainly result in a non-terminating completion procedure. In addition, even if we use a subset of the Halide TRS for our identities (thus yielding a confluent Halide TRS), our experience shows that many failures in the compiler's use of the TRS are due to missing semantic facts that are not derivable from the current Halide ruleset.

In the absence of a finite set of identities, we treat an SMT solver as a decision procedure to determine if a suggested identity holds in our theory (of course, the solver itself is also incomplete; we only make use of the soundness of the solver and cannot derive any information in the case where the solver cannot show that an identity holds). If the identity holds and can be oriented using our reduction order, it is added as a rule. It is possible that the newly-synthesized rule may be a consequence of the existing ruleset and thus could have been found by running completion, but we know that many synthesized rules contain information that is previously unknown to the TRS.

If we consider our solver as standing in for an infinite set of identities that make up our theory, we clearly could synthesize an infinite number of rules. Here we make use of the fact that expressions encountered by the compiler have some bias and are not sampled randomly from the entire expression space. In a preliminary experiment, we tried generating LHSs at random within a certain expression size and synthesizing RHSs to serve as new rules. We were able to find an extremely large number of "missing" rules not represented in the current TRS, but the new ruleset had no measurable performance impact on benchmarks. Thus, we only synthesize rules if their LHS could be applied to at least one expression observed by the compiler under realistic usecases.

2.5 Scope: Robust, Fast, Non-Backtracking Ruleset

In this work, we operate within the scope of Halide's TRS algorithm and work to make the TRS as correct, general, and robust as possible. Because the space of expressions we consider constitutes an undecidable theory, a complete TRS is impossible. Instead, we strive to improve correctness by ensuring the TRS will always terminate on any expression and that each individual rewrite preserves semantics; and we improve generality by expanding the ruleset to contain rewrites that apply to real-world expressions, rather than arbitrary new rules that may not apply to any expressions the compiler will encounter.

These improvements require overcoming challenging obstacles. First, we must perform a post-hoc verification of a large body of existing rules; proving a subset of rules correct or that a subset of rules do not result in infinite rewriting loops is insufficient to guarantee robust behavior. Secondly, these rewrites operate in an undecidable theory, making automated verification difficult. Finally, because of this undecidability, we cannot necessarily rely on traditional automated techniques to discover new rules.

3 SOUNDNESS

We improve the Halide term rewriting system by ensuring its soundness in two ways: first, we verify that each individual rule is correct, meaning that the rewrite preserves semantics. Then we verify that the term rewriting system is guaranteed to terminate on all inputs by ensuring that no sequence of rule applications, on any input expression, can form a cycle.

3.1 Rule Verification

We verify each individual rule is correct by modeling Halide expressions in SMT2 and using the SMT solver Z3 [De Moura and Bjørner 2008] to prove that the rule's left- and right-hand sides are equivalent. Most Halide expression semantics map cleanly to SMT2 formulas. The functions `max` and `min` are defined in the usual way, and `select` in Halide is equivalent to the SMT2 operator `ite`. Division and modulo are given the Euclidean definitions in both Halide and SMT2 [Boute 1992], though division and modulo by zero is handled differently (in Halide both evaluate to zero). Halide's TRS uses two vector-constructing operators, `broadcast` and `ramp`; all other integer operators can be coerced to vector operators. `broadcast(x, l)` projects some value x to a vector of length l; because of the type coercion, we can simply represent `broadcast(x, l)` as the variable `x` in SMT2. `ramp(x, s, l)` creates a vector of length l whose initial entry has the value x and all subsequent entries increase with stride s. In SMT2, we represent this term as the symbolic expression $x + l * s$, where l must be zero or positive.

Given this modeling, for each rule, we assert any predicate guards are true, then ask Z3 to search for a variable assignment that makes the LHS and RHS not equivalent. If Z3 indicates no such assignment exists, the LHS must be equivalent to the RHS and the rule must be correct. We implemented an SMT2 printer for Halide rewrite rules that automatically constructs an SMT2 verification problem for each rule. Rule verification using Z3 is fully automated and can be run for the current set of rewrite rules used in the compiler via a script.

However, for 123 rules, Z3 either timed out or returned unknown. Nearly all of these rules used either division or modulo. We used the proof assistant Coq to manually prove the correctness of these remaining rules. In the course of these proofs, we discovered we were also able to relax the predicate guards of 17 rules; for example, in some cases a rule with a guard requiring some constant to be positive would be equally valid if the constant was non-zero.

Evolving Semantics. This mostly-automated approach to verification assists with changing the language semantics. Our initial work on verification was not on the semantics described above: division or modulo by zero was originally considered undefined behavior. Since we had already modeled Halide semantics in SMT2, it was easy to alter the definitions of division and modulo and re-run the verification scripts. We proved 141 rules manually in Coq after Z3 failed to verify them; since in the previous round all Coq proofs included the assumption that all divisors were non-zero, in most cases we had only to add a case to show that the rule was true when the divisor was zero as well. In the course of reviewing these proofs, we identified 37 rules whose predicates included the condition that a divisor be non-zero and where that condition could safely be lifted. We found that the remaining 44 rules were not correct under the new semantics and submitted a patch to amend them.

Overall results for verifying rule correctness are described in Section 5.2.1.

3.2 Termination

Under the umbrella goal of simplifying expressions, the Halide TRS uses many strategies: it may attempt to make expressions as short as possible; it may factor out vector operations or more expensive operations such as division; it may attempt to canonicalize subexpressions so they can cancel or be shown equivalent. These strategies are not necessarily aligned and may even undo each other. Crafting new rules can thus require a detailed understanding of the ruleset and its various applications. In this section we formalize the Halide expression simplification strategy that was previously only encoded in the ruleset itself. In doing so, we also prove that since each rule strictly makes progress in accordance to this strategy, the Halide TRS always terminates.

Consider a term rewriting system containing only one rule: $x + y \ \rightarrow_R \ y + x$. The term $3 + 5$ matches the LHS of the rule and is rewritten to $5 + 3$, which can again be matched to the rule and rewritten to $3 + 5$, and so on. Termination failures in the Halide TRS have occurred in the past[3], causing unbounded recursion and eventually a stack overflow in the compiler. This is tricky to debug, and may not always be reported by users, since the error is fairly opaque. To show that this type of error has been eliminated, we must prove that there is no expression in the Halide expression language that can be infinitely rewritten by some sequence of rules that form a cycle.

Intuitively, we can think of Halide expressions as existing in some multi-dimensional space; when an expression is rewritten by a rule, it moves from one point in that space to another. If each rule always rewrites expressions such that they move monotonically in some direction through the expression space, then no sequence of rules can form a cycle. These directions correspond to our intuition about why certain rules are useful (like the examples at the beginning of this section). We can consider each of these directions as a dimension in the expression space. If we formalize this desirable ordering and show that all rewrites from one expression to another strictly obey it, then we will have a proof of termination.

We provide this formalism and prove that the Halide term rewriting system must terminate by constructing a *reduction order*, a strict order with properties that ensure that, for an order $>$ and a rule $l \ \rightarrow_R \ r$, if $l > r$, then for any expression e_1 that matches l and is rewritten by $l \ \rightarrow_R \ r$ into e_2, it must be true that $e_1 > e_2$. Crucially, this order is evaluated over rule terms, and not over all expressions that those terms may match. We take the definition of a reduction order and the next two theorems from Baader and Nipkow [1999].

Theorem 3.1. *A term rewriting system R terminates iff there exists a reduction order $>$ that satisfies $l > r$ for all $l \ \rightarrow_R \ r \in R$.*

A reduction order is a strict order that must be well-founded, meaning that every non-empty set has a least element with regard to the order, to prevent infinitely descending chains. It must be *compatible with Σ-operations*: for all expressions s_1, s_2, all $n \geq 0$, and all $f \in \Sigma$:

$$s_1 > s_2 \implies f(t_1, ...t_{i-1}, s_1, t_{i+1}, ..., t_n) > f(t_1, ...t_{i-1}, s_2, t_{i+1}, ..., t_n)$$

for all $i, 1 \leq i \leq n$ and all expressions $t_1, ...t_{i-1}, t_{i+1}, ..., t_n$. This property means that if a rewrite rule transforms a subtree in some expression e, the $>$ relation is preserved between the original expression e and the rewritten expression e'. Finally, a reduction order is *closed under substitution*: for all expressions s_1, s_2 and all substitutions $\sigma \in \mathcal{S}ub(T(\Sigma, V))$, $s_1 > s_2 \implies \sigma(s_1) > \sigma(s_2)$. When we match some left-hand side term l to some expression e, we are defining a substitution for each of the variables in l with some subtree in e; we then use that substitution to rewrite e to e'. If our order is closed under substitutions, we know that for any expression we match to l, the resulting rewritten expression will obey the ordering.

Choosing a single monotonic direction in which to rewrite expressions would be overly restrictive. The Halide TRS is used both to prove expressions true and to simplify them; when using it as a prover, we want to put both sides of an equality into some normal form, but it doesn't particularly matter what that form is. When using the TRS to simplify expressions, on the other hand, reducing the size of an expression has important performance benefits. Since we need an ordering that covers the full Halide simplification strategy, we make use of the following theorem:

Theorem 3.2. *The lexicographic product of two terminating relations is again terminating.*

Thus, our strategy in finding a reduction order to cover the handwritten ruleset is to pick an order $>_a$ such that for all rules $l \ \rightarrow_R \ r$, either $l >_a r$ or $l =_a r$. Then, we pick another order $>_b$

[3]See for example https://github.com/halide/Halide/pull/1525

such that for all rules $l \rightarrow_R r$ where $l =_a r$, either $l >_b r$ or $l =_b r$. We continue in this way until a sequence of orders has been found such that for their product $>_\times$, $l >_\times r$ holds for the entire ruleset. Our final ordering consists of 13 component orders.

Many of our component orders are defined using measure functions that count the number of particular operations or other features in a term. We say that $s > t$ when s has more vector operations than t, then when s has more division, modulo and multiplications operations, and so on. As a sample proof sketch of this flavor of order, consider an order $s_1 >_* s_2$ that holds when the number of multiplication operations is greater in s_1 than in s_2. We represent this through a measure function $|s_1|_*$ that returns the count of multiplication operations in s_1; since this function maps a term to a natural number, the order is clearly well-founded. The order is also compatible with Σ-operations; we compute our measure function as follows:

$$|f(t_1, ..., t_n)|_* = \sum_i^n |t_i|_* + \begin{cases} 1 & \text{if } f = * \\ 0 & \text{otherwise} \end{cases}$$

It clearly follows that given $|s_1|_* > |s_2|_*$, it must be true that:

$$|f(t_1, ...t_{i-1}, s_1, t_{i+1}, ..., t_n)|_* > |f(t_1, ...t_{i-1}, s_2, t_{i+1}, ..., t_n)|_*$$

To ensure the order is closed under substitution, we need to add one more constraint. Imagine a rule $x * 2 \rightarrow_R x + x$. Although there are fewer $*$ symbols in the righthand term than on the left, that would not be true for a substitution $\sigma = \{x \mapsto (z * z)\}$. We add a condition that for every variable present in s_1, it must occur either fewer or an equal number of times in s_2. With this constraint there is no possible substitution that increases the value of the measure function in s_2 that would not result in an increase by an equivalent or greater amount in s_1. This gives us the order:

$$s_1 >_* s_2 \text{ iff } |s_1|_* > |s_2|_* \wedge \forall x \in \mathcal{V}ar(s_1).|s_1|_x \geq |s_2|_x$$

Most of the component orders in the full reduction order take the form above. These orders guarantee termination no matter what sequence rewrite rules are applied to an expression. However, for part of the existing ruleset, we were obliged to take into account the order in which rules are applied in the Halide TRS algorithm.

For example, one existing rule is the canonicalization $(c_0 - x) + y \rightarrow_R (y - x) + c_0$ where c_0 is a constant. If y is also a constant, this rule forms a cycle with itself, and could not possibly obey any reduction order. Fortunately, the rule immediately before it in the TRS handles that specific case ($((c_0 - x) + c_1 \rightarrow_R \texttt{fold}(c_0 + c_1) - x)$), so by this sort of non-local reasoning we know that y is not a constant, and therefore the rule strictly decreases a measure which counts the number of constants on the right-hand side of an addition.

Relying on non-local reasoning makes our order more brittle; if the simplifier algorithm were to be changed, the termination guarantee could be lost. However, we use only a small number of basic rules in this way, which are unlikely to be changed.

Besides giving a termination guarantee, the reduction order is necessary if we want to synthesize new rewrite rules. If we do not constrain newly-synthesized rules to obey a consistent reduction order with the existing human-written ones, they form cycles with the existing rules and cause infinite recursion in the TRS. Additionally, the reduction order is the formal encoding of the types of transformations we find desirable, so the reduction order limits synthesis to rules that rewrite expressions in a useful direction.

In constructing the reduction order, we found 8 rules that contradicted a desirable ordering, and submitted patches to either delete or modify them. With this amendment, the reduction order can be shown to hold over the entire Halide ruleset, and the guarantee of termination is complete. To

ensure this guarantee is preserved, we build a script that automatically checks the full set of rules in the compiler to ensure they respect the reduction order. A full description of the reduction order is given in the supplemental material.

4 INCREASING COMPLETENESS: SYNTHESIZING REWRITE RULES

Although the Halide term rewriting system is necessarily incomplete, we can strengthen it by finding expressions on which the TRS can no longer make progress and creating rules that will further simplify them. In this section, we describe a workflow for automatically augmenting the Halide TRS with new rules.

Given an *input expression* that the TRS failed to simplify, our goal is to find a rule that can rewrite it. A high-level view of the synthesis pipeline is shown in Figure 2. We begin with an expression we will attempt to further simplify; first, we synthesize rules that contain concrete constants from the input expression. Next we generalize those rules by replacing constant values with symbolic constants and synthesizing compile-time predicate guards that check the validity of the rule on the values matched by the symbolic constants. If we find such a rule, we know that adding it to the TRS will enable it to simplify the input expression as well as any similar expressions it may encounter.

The set of input expressions may come from a bug report, or may be gathered from compiler logs. With logging enabled, the compiler records two kinds of problematic expression for which new TRS rules may be helpful: non-monotonic expressions, which can result in over-conservative bounds for loops and memory allocations; and proof failures, which may prevent Halide from performing certain optimizations (see Section 2.2). Of course, absent an oracle, it is difficult to know if the TRS has fully simplified some expression or if it lacks the solving power to continue simplification. When the TRS is used as a proof engine, its goal is to reduce an expression to true. In this case, we can fuzz-test failed proofs by assigning all variables in an expression random values and evaluating; if we cannot find an assignment that evaluates to false, the expression may indeed be reducible to true, so we log it as an input expression.

4.1 Generating LHS Patterns

Our first step is to find LHS terms that could match the input expression, or any portion of it. We can enumerate all such terms through a kind of inverse matching. When we rewrite an expression with a rule, we match the expression to the rule's LHS by finding a substitution for all variables in the LHS that will unify it with the input expression. Here, we start with an input expression, then fix a substitution by mapping some of its subterms to fresh variables. We replace those subterms with the new variables, constructing a term that can be matched with the input term. If we perform this inverse matching for all sets of subterms, we find *all possible LHSs* that could match the input expression. When a subterm occurs more than once in the input expression, we construct a LHS that uses the same variable to replace it in multiple places and LHSs that replace its occurrences with different variables. We repeat the procedure on all *subterms* of the input expression. The result is the set of all LHSs that match any part of the input expression. See Figure 3 for a worked example.

This number of LHSs is exponential in the size of the input expression, so we use a few heuristics to narrow our search. We bound the size of candidate LHSs to have seven or fewer leaves, since longer terms are less likely to result in rules general enough to justify inclusion in the ruleset. Additionally, since we process input expressions in batches, we remove duplicate LHSs as well as LHSs that differ only in the values of their constants. Finally, we have found it helpful to keep a blacklist of LHSs for which we previously failed synthesize rules; for example, $v_1 + v_3$ cannot form a rule, so we filter it out as a candidate.

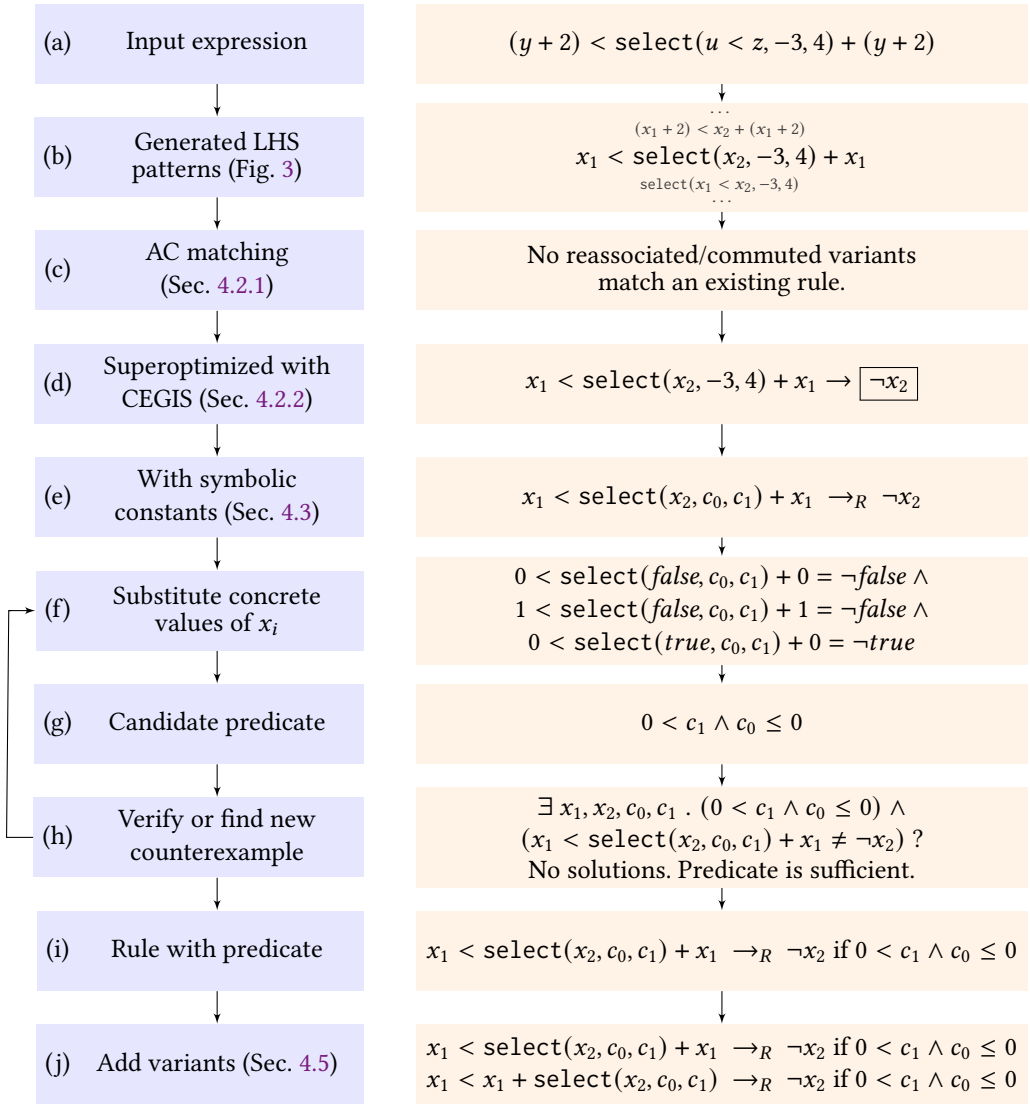
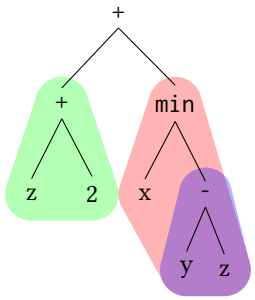


Fig. 2. Overall flow of the synthesis pipeline (in blue) with worked example (in orange). (a) We harvest expressions from real compilations on which the TRS could make no further progress. (b) We enumerate all subtrees of these to generate left-hand sides that would match each expression. Our example will focus on one such pattern. (c) We obtain a right-hand side by first checking if any reassociated or commuted variants of it match an existing TRS rule. (d) If not, we superoptimize the pattern using CEGIS. (e) This rule is specific to the particular values of any constants that appear. We then replace any constants with new variables $c_0, c_1, etc.$, to obtain a more general version of the rule. We must now synthesize a sufficient condition on these new variables under which the rule still holds. (f) To do this, we treat the rewrite as an equality and take the conjunction over a set S of different values for the non-constant variables $x_0, x_1, etc.$ (g) Simplifying the result gives a candidate predicate. This is a *necessary* condition. (h) We then check if it is also *sufficient* condition using Z3. (i) If a counterexample is found, we add these new values of x to S to obtain a new candidate predicate and repeat until we have a sufficient condition to serve as our predicate. (j) Finally, we construct variants of the rule in which the LHS has been commuted.



$(z+2)+\texttt{min}(x, y-z)$ $\quad \texttt{min}(x, y-z)$ $\quad z+2$ $\quad y-z$
$v_1+\texttt{min}(x, y-z)$ $\quad \texttt{min}(x, v_2)$
$v_1+\texttt{min}(x, v_3)$
$(z+2)+\texttt{min}(x, v_3)$
v_1+v_2

Fig. 3. Given the input expression $(z+2)+\texttt{min}(x, y-z)$, we find all possible LHS patterns by substituting fresh variables for subterms, for all valid combinations. Then, we repeat the process for each individual subterm. This process yields the list of candidate LHS terms on the right.

4.2 Synthesizing Right-Hand Sides

Given a candidate left-hand side, we attempt to synthesize a right-hand side that is semantically equivalent and respects the reduction order, namely $LHS > RHS$. We employ two strategies for synthesizing right-hand sides: delayed AC matching, and counter-example guided inductive synthesis (CEGIS) of the RHS followed by synthesis of the rule predicate guard.

4.2.1 Finding Right-Hand Sides through AC Matching. The first strategy reflects the Halide design decision not to perform any AC matching in the TRS, for efficiency reasons. Instead, AC matching is effectively performed during rule synthesis, by checking whether the LHS could be rewritten by the existing TRS after a suitable application of associativity and commutativity laws to the LHS. To this end, we generate all possible reassociations and commutations of the candidate LHS term and pass them to the existing TRS. If any of them can be simplified, we create a new rule that rewrites the original, untransformed LHS term to the result of the simplification. Note that this result may include applications of more than one rewriting step, so the new rule is not merely an AC-variant of an existing rule.

For example, assume our TRS includes the rule $(x+y)-x \ \rightarrow_R \ y$, and let $((u+2)+v)-u$ be a candidate LHS term. The rule does not match the candidate but it matches its variant $(u+(v+2))-u$, rewriting it to the result $v+2$. The candidate and the result give us the rule $((u+2)+v)-u \ \rightarrow_R \ v+2$.

We can consider this procedure a kind of lazy offline AC matching, because if the Halide TRS performed full AC matching while rewriting expressions, it would be able to apply the rule $(x+y)-x \ \rightarrow_R \ y$ to the candidate expression $((u+2)+v)-u$ after reassociating it to $(u+(v+2))-u$, obtaining the result $v+2$. Delaying AC matching to synthesis has the effect of restricting the system to a single, offline round of AC and memoizing the result in the form of a new TRS rule if we are successful. Note that the synthesis procedure below could have found this rule, but checking for AC variants of existing rules is far cheaper. About three-quarters of our synthesized rules are generated by this method.

4.2.2 Finding Right-Hand Sides through CEGIS. If the first method fails, we apply counterexample guided inductive synthesis (CEGIS) [Solar-Lezama 2009] to superoptimize the left-hand side pattern. In superoptimization [Massalin 1987], we take a program and search for an equivalent program within some grammar that is preferable according to some cost function. Here our grammar is that of the Halide expression language, the method for testing program equivalence is the Z3 solver, and we use the node count of the programs as a proxy for our full reduction order.

Table 2. Sample rules synthesized by our process.

LHS	RHS	Predicate
$(x * y) - (z + (w * x))$	$(x * (y - w)) - z$	
$x < (y + x) + z$	$0 < (y + z)$	
$\texttt{max}(x * x, y) + \texttt{max}(z, w * w) < c_0$	false	$c_0 <= 0$
$\texttt{select}(x, c_0, y) < \texttt{min}(\texttt{select}(x, c_1, y), c_2)$	false	$\texttt{min}(c_1, c_2) <= c_0$
$\texttt{min}((x + ((y - x)/c_0) * c_0) + c_1, y)$	y	$1 <= c_1 \wedge -1 <= (-1/c_0) * c_0 + c_1$

Similar to prior work in superoptimization [Phothilimthana et al. 2016a; Sasnauskas et al. 2017b], we search the expression space for an equivalent RHS using a CEGIS loop. This loop alternately calls Z3 as a verifier, which checks if a candidate RHS is equivalent to the LHS on all inputs, and a learner, which finds a candidate RHS that is equivalent to the LHS on a limited set of inputs. We begin by choosing a single-op RHS and ask the verifier if it is equivalent to the LHS. If it is not, we get back a counterexample of assignments to the variables for which the right- and left-hand side are not equivalent, which we keep as a set of test inputs. We then ask the learner for a new RHS that is equivalent to the LHS only on the counterexample assignments we found in the last step. If we cannot find an equivalent single-op sequence, we iteratively increase the number of operations, ensuring we find shorter sequences first. If CEGIS returns a sequence semantically equivalent to the LHS pattern with fewer operations, we use it together with our LHS to form a candidate rule.

The learner portion of the CEGIS loop creates a candidate RHS by creating a sketch [Solar-Lezama 2009; Torlak and Bodik 2014] that consists of a small bytecode interpreter that encodes the possible operations and operands the RHS can use, along with a bound on the number of instructions. The learner uses Z3 to query for a sequence of bytecodes within the bound, that, when run through the interpreter, is semantically equivalent to the LHS over the test inputs. If a solution is found, substituting the produced bytecode values into the sketch and applying the TRS reduces it to a concrete candidate RHS. One complication arising from this approach is that a bytecode sequence of a fixed number of ops may produce expression trees of a larger size if intermediate values are reused. We reject any such solutions in a post-pass by checking each synthesized RHS against the LHS using the full reduction order. An alternative solution would be introducing let bindings into our search space so that the size of the expression tree could be bounded by the number of ops in its SSA form. However, we could not identify any significant rewrite rules lost to this filtering, so we deemed this an unnecessary complication.

While Z3 is a powerful tool for synthesis, there are certain types of expressions containing division or modulo that Z3 nearly always fails to reason about during the CEGIS process. (We experimented with the SMT solvers Yices2 [Jovanović 2017] and MathSAT5 [Cimatti et al. 2013], but were not able to obtain appreciably better results.) Z3 is better able to reason about expressions containing concrete constants, rather than universally quantified variables, so we synthesize rules using candidate LHSs with concrete constants from the input expression and generalize them later. We limit the use of division and modulo in our op-codes to be division or modulo by 2 only, and rely on the generalization step described next to widen the set of denominators for which a rule applies. Because of this restriction, our synthesized rules cannot contain non-constants in denominators or the right-hand side of a modulo. As a result, our synthesis system cannot construct all rules a human can.

4.3 Generalizing Constants and Finding Predicate Guards

If either AC-matching search (Section 4.2.1) or CEGIS-based synthesis (Section 4.2.2) were successful, we now have a candidate rewrite rule that contains concrete values originating from the input

expression. To generalize the rule, we replace such constants with fresh *symbolic constants* and synthesize a guard that is true when the rule is valid. Recall that in the Halide TRS, a variable in the LHS matches any subterm, while a symbolic constant matches only a constant value (see Section 2.3.1); the guards, which are predicates over symbolic constants, can thus be evaluated at compile time.

Our goal is to generalize the equality by synthesizing a guard predicate ϕ over the symbolic constants in the LHS and RHS terms such that our rule is valid whenever ϕ evaluates to true:

$$\forall\vec{c}\forall\vec{x}\ .\ \phi(\vec{c}) \implies LHS(\vec{x},\vec{c}) = RHS(\vec{x},\vec{c})$$

First, we check to see if this condition is satisfied when ϕ is always true. If it is, then no predicate guard is needed. Otherwise, we need to synthesize an expression for ϕ. We find candidates for ϕ iteratively by first choosing a small set of values S for the variables in $\vec{x}$ and finding the candidate guard ϕ_S. We check to see if ϕ_S is a sufficient predicate guard for all $\vec{x}$; if it is not, we add counterexamples to the set S and repeat.

$$\forall\vec{c}\forall\vec{x}\in S\ .\ \phi_S(\vec{c}) \implies LHS(\vec{x},\vec{c}) = RHS(\vec{x},\vec{c})$$

We initialize S with all basis vectors, which are values $\vec{x} = (0,\ldots,0,1,0,\ldots,0)$ that include exactly one unit value, plus the zero vector. We then unwind the right-hand side of the implication and substitute in the concrete values from S to get:

$$\forall\vec{c}\forall\vec{x}\in S\ .\ \phi_S(\vec{c}) \implies (LHS(\vec{x_1},\vec{c}) = RHS(\vec{x_1},\vec{c}) \wedge \ldots \wedge LHS(\vec{x_k},\vec{c}) = RHS(\vec{x_k},\vec{c}))$$

We use the Halide TRS itself to simplify the conjunction on the right-hand side of the implication. Since all occurrences of $\vec{x}$ have been replaced with concrete values, we get back an expression that contains only symbolic constants, which we use as our candidate guard ϕ_S.

We test whether ϕ_S is sound on all $\vec{x}$.

$$\exists\vec{c}\ \exists\vec{x}\ .\ LHS(\vec{x},\vec{c}) \neq RHS(\vec{x},\vec{c})$$

If this query has a solution $\vec{x}$, then the guard is unsound. If so, we add the counterexample $\vec{x}$ to S, and construct a new guard ϕ_S. We repeat this process for several iterations (four, in our experiments) and if we fail to find a sound guard, we switch to an alternative strategy that converts the current (unsound) candidate ϕ_S to disjunctive normal form and tests each clause in turn to check if it is a sufficient guard. If it is, that clause becomes the guard. If no clause is sound, we discard the rule. If the loop terminates with Z3 timing out or returning "unknown", we return the current ϕ_S, flagging it as requiring a manual proof. We exclude all such cases from our experiments.

As an example, consider the candidate rule:

$$x_0 < \texttt{select}(x_1, c_0, c_1) + x_0 \ \rightarrow_R\ \neg x_1$$

We initialize S with three basis vectors $\{(0, \mathit{false}), (0, \mathit{true}), (1, \mathit{false})\}$ and construct ϕ_S:

$$\begin{aligned}\phi_S(\vec{c}) &\iff \forall_{\vec{x}\in S}\ .\ LHS(\vec{x},\vec{c}) = RHS(\vec{x},\vec{c})\\ &\iff 0 < \texttt{select}(\mathit{false}, c_0, c_1) + 0 = \neg\mathit{false} \wedge\\ &\qquad 0 < \texttt{select}(\mathit{true}, c_0, c_1) + 0 = \neg\mathit{true} \wedge\\ &\qquad 1 < \texttt{select}(\mathit{false}, c_0, c_1) + 1 = \neg\mathit{false}\end{aligned}$$

Simplifying the RHS with the TRS, we obtain ϕ_S:

$$\phi_S(\vec{c}) \iff 0 < c_1 \wedge c_0 \leq 0$$

Next we check whether ϕ_S is sound for all $\vec{x}$. It is, so we have a completed rule:

$$x_0 < \texttt{select}(x_1, c_0, c_1) + x_0) \;\rightarrow_R\; \neg x_1 \;\text{ if }\; 0 < c_1 \wedge c_0 \leq 0$$

4.4 Adding Rule Variants

Once we have a generalized rule with a valid predicate, we eagerly compensate for the lack of AC matching in the Halide TRS by adding AC variants of the rule as well. We find all commuted variants of the rule's LHS, with respect to the partial commutative canonicalization as described in Section 2.3.1. (This is exponential in the size of the number of commutative operators, which is tractable given our bounds on LHS term size). Then, we find all reassociations of the rule's right-hand side. For each variant LHS, we choose a RHS variant by serializing expressions to strings and finding the RHS that has the shortest edit distance from that LHS.

For example, the LHS of the first rule below has four additions and can be commuted to 16 variants. The RHS of the rule can be reassociated in two different ways. For the commuted variant of the LHS on the second line, we choose the other means of reassociating the RHS as it has a smaller edit distance.

$$(x + (y - ((z + (w + x)) + u))) \;\rightarrow_R\; y - (z + (w + u)))$$
$$(y - (((w + x) + z) + u)) + x \;\rightarrow_R\; y - ((z + w) + u)$$

The intuition is that there is no a priori reason to prefer one reassociated variant to another; they are almost certainly equal in terms of our reduction order. Thus, we choose the RHS that perturbs the structure of the LHS as little as possible, in order to avoid rewriting common subexpressions in the hopes of canceling them out later.

4.5 Filtering Rule Output

As a final step, we check each output rule for redundancy with the rule batch found by the synthesis pipeline. For each new rule, we check that no earlier rule has precisely the same LHS and predicate; if so, it can be discarded. Then, we check that no earlier rule is more general than the current rule: a rule is more general than another if they have similar LHSs, but a variable appears in the first rule in a place where the second rule has a more specific subterm, or if they have the same LHS but the predicate of the first rule implies the predicate of the second.

Finally, we check that the candidate rule obeys our reduction order in order to preserve our termination guarantee. If the candidate rule passes these filters, and the predicate has not been flagged for human review, the rule can be added to the TRS ruleset automatically without any human auditing.

5 EVALUATION

In evaluating the benefits of the verifier and synthesizer, we answer the following questions:

- **Does the synthesizer produce better rules than a human expert?** The TRS has been manually extended five times in response to bug reports pointing out limitations of the compiler. We synthesized these five rulesets automatically and found that the human-authored rules were less general and in one case were incorrect. (Section 5.1.1)
- **Can verification contribute to the Halide TRS, or is testing alone sufficient?** Although handwritten rules have been extensively fuzz-tested and new rules are peer-reviewed before inclusion, we were still able to discover 4 unsound rules through verification. In addition, verification was able to identify 44 rules that needed to be changed after a significant semantics redefinition, which would have been challenging to discover through testing. (Section 5.2.1)

- **What is the best way to use synthesis and verification in development?** We survey several cases from recent Halide development where human experts used the synthesis machinery as an assistant, finding that this hybrid model is more powerful than either the human developer or the synthesizer alone. (Section 5.2.2)
- **Can synthesis be used for large-scale improvements of the TRS?** We gather a corpus of over 100,000 expressions on which the TRS can make no progress and iteratively synthesize rules using the corpus as input. We synthesize 4127 rules and add them to the TRS ruleset without a human audit. We find that the enhanced ruleset reduces peak memory usage in compiled code, sometimes dramatically, in 197 of our benchmarks. We also find no significant compile-time slowdown even with this 4.5-fold increase in ruleset size. (Section 5.3.1)
- **Could the entire TRS have been synthesized?** Encouraged by the large-scale experiment, we ask how far we are from being able to bootstrap the entire TRS automatically—something that we considered too ambitious originally. First, we find that 69% of the existing ruleset is accessible to our current synthesizer in principle; the remaining rules contain operators not yet supported by the tool. We test the synthesizer's power by removing 321 accessible rules from the original ruleset one by one and attempting to synthesize a replacement, successfully finding a replacement rule 58% of the time. We find this encouraging for future applications of the synthesizer. (Section 5.1.2)

We discuss these findings in more detail below, grouping them into three sections. First we examine bug reports from Halide's past and evaluate whether the machinery presented in this paper could have fixed them automatically. Second, we examine cases where beta versions of our verifier and synthesizer assisted humans both in fixing bugs and in correctly making larger changes to the compiler. Third, we fuzz the compiler to mine for issues that could be fixed with new simplifier rules, and automatically fix them before they ever appear as a bug in a real program. In this way we demonstrate that this machinery would have been useful in the past, is useful in the present, and will help avoid entire classes of bugs in the future.

5.1 Comparing the Synthesizer to Human-Authored Rules

5.1.1 Does the Synthesizer Produce Better Rules than a Human Expert? We searched through Halide's change history and selected the five pull requests that addressed issues by adding new rewrite rules to Halide's TRS. These pull requests occurred before the Halide developers started routinely using the verifier and synthesizer when changing the TRS. These can be found as summarized diffs $\mathbb{A}$-$\mathbb{E}$ in supplemental material, or in their original form on the Halide project website [4]. Creating these rewrite rules as a human is an amount of work disproportionate to the size of the change. The author of the rules must prove them correct on paper, and a second reviewer must check their work. As we will see, bugs can slip through despite this review.

In each case we take the test expressions committed as part of the change and feed them to our synthesizer to see if it would have produced the same rewrite rules as the humans did. In cases where humans did not check in tests for their new rules, we wrote our own. In total, across these five cases humans added 24 new rules. The synthesizer generated 42, covering all but one of the human rules, while correcting and generalizing others. In cases $\mathbb{A}$, $\mathbb{C}$, and $\mathbb{E}$, the rules generated by the synthesizer are an exact match to the human-generated rules. In case $\mathbb{B}$ the synthesizer matched the human but also crafted 8 commuted variants of the human rules, making them more widely applicable.

[4] $\mathbb{A}$: https://github.com/halide/Halide/pull/3719 $\mathbb{B}$: https://github.com/halide/Halide/pull/3761 $\mathbb{C}$: https://github.com/halide/Halide/pull/3765 $\mathbb{D}$: https://github.com/halide/Halide/pull/3770 $\mathbb{E}$: https://github.com/halide/Halide/pull/3780

As an example, for the human-written rule:

$$\texttt{max}(\texttt{max}(x, y) + c_0, x) \;\rightarrow_R\; \texttt{max}(x, y + c_0) \text{ if } c_0 < 0$$

The synthesizer produced effectively the same rule, along with a variant:

$$\texttt{max}((\texttt{max}(x, y) + c_0), x) \;\rightarrow_R\; \texttt{max}((y + c_0), x) \text{ if } c_0 \leq 0$$
$$\texttt{max}(x, (\texttt{max}(x, y) + c_0)) \;\rightarrow_R\; \texttt{max}(x, (y + c_0)) \text{ if } c_0 \leq 0$$

Case $\mathbb{D}$ is the most interesting. It contains four rules involving comparisons of `min` and `max` operations. What happened for each was identical, so we will only discuss the `min` rules. The first rule is:

$$\texttt{min}(x, c_0) < \texttt{min}(x, c_1) + c_2 \;\rightarrow_R\; \texttt{false} \text{ if } c_0 \geq c_1 + c_2)$$

This rule is incorrect (consider $c_0 = c_2 = 1$, $x = c_1 = 0$). It can be fixed by adding the term $c_2 \leq 0$ to the predicate. The synthesizer produced the correct version of this rule, along with two generalizations of it:

$$\texttt{min}(x, c_0) < \texttt{min}(x, c_1) + c_2 \;\rightarrow_R\; \texttt{false} \text{ if } c_2 \leq 0 \wedge c_1 + c_2 \leq c_0$$
$$\texttt{min}(x, c_0) < \texttt{min}(x, y) + c_1 \;\rightarrow_R\; c_0 - c_1 < \texttt{min}(x, y) \text{ if } c_1 \leq 0$$
$$\texttt{min}(x, c_0) < \texttt{min}(y, x) + c_1 \;\rightarrow_R\; c_0 - c_1 < \texttt{min}(y, x) \text{ if } c_1 \leq 0$$

The second human rule was:

$$\texttt{min}(x, c_0) < \texttt{min}(x, c_1) \;\rightarrow_R\; \texttt{false} \text{ if } c_0 \geq c_1$$

The synthesizer found a more general rule, along with three other commuted variants (elided for space):

$$\texttt{min}(x, y) < \texttt{min}(x, z) \;\rightarrow_R\; y < \texttt{min}(x, z)$$

Any expression which matches the human-written rule would also match the synthesized one. The synthesized version does not simplify to the constant false in a single step. However, after applying this rule to the case considered by the human, we get $c_0 < \texttt{min}(x, c_1)$ where $c_0 \geq c_1$. The simplifier then reduces this to false in a second step, so the human-written rule becomes unnecessary. The synthesizer considered the human-written rule, but discarded it as less general than the one above.

Case $\mathbb{D}$ also included the rewrite rule:

$$x \;\%\; x \;\rightarrow_R\; 0$$

which was the sole rule the synthesizer could not generate, as we did not include modulo by non-constants in our CEGIS interpreter.

With this one exception, across these five code changes the synthesizer generated more general, more correct rules than the humans, and would clearly have been a useful assistant to the Halide developers if they had had it at the time.

5.1.2 What Fraction of the Halide Rules Could Have Been Synthesized? To bootstrap a TRS, we could start with a TRS equipped with a basic set of rules and synthesize the remaining rules as the TRS encounters expressions needing those rules. How far is the synthesizer from supporting this ambitious vision?

Given the space of possible rewrites explored by the synthesizer, we believe that it can currently produce at most 69% of rules that exist in the current TRS. The obstacles to synthesizing all human-written rules include (i) the inability to automatically verify some rules or preconditions (see Question 3 above); and (ii) lack of support for some operators in our synthesizer.

We tested the synthesizer's ability to recreate the original ruleset in the following experiment. We instrumented the ruleset to associate expressions from compilations of Halide's correctness test suite with individual rules invoked when those expressions are rewritten. We gathered a set of rules for which we had at least three expressions that matched the rule, and filtered out those rules that are out of scope for the current synthesizer, because their right-hand sides contain operators we do not support. This gave us a set of 321 rules. For each of these rules, we disabled the rule in the TRS, then used its matching expressions as input to the synthesizer. The synthesizer was able to find rules in 186 cases, or about 58% of the rules. Of the other 135 cases, in 43 of them other rules in the existing TRS happened to combine to rewrite the specific input expression even without the target rule; for example, this often occurred when the matching expression contained combinations of constants that could be exploited by other rules. 10 of the 92 failure cases were due to timeouts in the synthesis process, while 15 specifically failed to synthesize a predicate. Given that it was difficult to target the desired rule precisely, we find these results to be a promising state of the technology.

5.2 Practical Uses of the Synthesizer and Verifier

5.2.1 Can Verification Contribute to the Halide TRS, or is Testing Alone Sufficient? The Halide TRS has a stringent development process: new rules are peer reviewed after they are proven on paper, and fuzzing has been discovering bugs for months. It is thus reasonable to ask whether mechanized verification can add any value. Our verification discovered 4 new soundness bugs and 17 instances of rules whose predicates were overly restrictive. The former bugs eluded the fuzzer; the latter are deemed too hard so the fuzzer does not look for them. Furthermore, because the verification infrastructure was in place, it was possible to verify a change of semantics without much additional effort, identifying 44 rules that were incorrect under the new semantics.

The first use of verification took place when the TRS had not yet been merged into the Halide master branch. We ran the verification pipeline and discovered 4 incorrect rewrite rules, listed in Table 3. The rules that could not be checked with Z3 were proved true using the Coq proof assistant (none of the manually proved rules were found to be incorrect). While these bugs were found automatically the fixes were performed by hand, as the synthesis pipeline did not yet exist.

Case $\mathbb{H}$ [5] is a change to the semantics of Halide that may not have even been attempted without the verifier. In this change, Halide defined division or modulo by zero to evaluate to zero, instead of being undefined behavior, in response to an issue discovered by Alex Reinking [Reinking 2019]. Existing tests and real uses of Halide were useless as a test of this change, as *they were all carefully written to never divide by zero*. Within the TRS, this change required rechecking every rewrite rule that involves the division and modulo operators. Whereas previously each rule assumed that a denominator on the LHS could not be zero, now it was necessary to either show that the rule was still correct in the case where a denominator was zero, or constrain the rule to only trigger when the denominator was known to be non-zero. This was done by encoding the new semantics into the verifier, and reverifying all rules. Because division and modulo is involved, these rules cannot always be mechanically verified. 141 rules were reverified with a human in the loop by revisiting and modifying existing Coq proofs. The mechanical re-verification was all but push-button; the

[5] $\mathbb{F}$: https://github.com/halide/Halide/pull/4721 $\mathbb{G}$: https://github.com/halide/Halide/pull/4772 $\mathbb{H}$: https://github.com/halide/Halide/pull/4439 $\mathbb{I}$: https://github.com/halide/Halide/pull/4850

Table 3. Rules corrected through the first round of verification.

	Rule	Counterexample
Wrong	$\frac{x*c_0}{c_1} \rightarrow_R \frac{x}{(c_1/c_0)}$ if $c_1 \,\%\, c_0 = 0 \wedge c_1 > 0$	$c_0 = -1, c_1 = 2, x = 1$
Fixed	$\frac{x*c_0}{c_1} \rightarrow_R \frac{x}{(c_1/c_0)}$ if $c_1 \,\%\, c_0 = 0 \wedge c_0 > 0 \wedge \frac{c_1}{c_0} \neq 0$	
Wrong	$(\frac{x+c_0}{c_1}) * c_1 - x \rightarrow_R x \,\%\, c_1$ if $c_1 > 0 \wedge c_0 + 1 = c_1$	$c_0 = 2, c_1 = 3, x = -5$
Fixed	$(\frac{x+c_0}{c_1}) * c_1 - x \rightarrow_R -x \,\%\, c_1$ if $c_1 > 0 \wedge c_0 + 1 = c_1$	
Wrong	$x - (\frac{x+c_0}{c_1}) * c_1 \rightarrow_R -(x \,\%\, c_1)$ if $c_1 > 0 \wedge c_0 + 1 = c_1$	$c_0 = 2, c_1 = 3, x = -5$
Fixed	$x - (\frac{x+c_0}{c_1}) * c_1 \rightarrow_R ((x + c_0) \,\%\, c_1) + -c_0$ if $c_1 > 0 \wedge c_0 + 1 = c_1$	

manual effort for updating the Coq proofs was non-trivial, but about half of the effort of writing the original proofs from scratch. In this process, 44 existing rules were found to be incorrect in the new semantics and fixed. (Two of them were in fact not related to division, but were instead the first discovery of the bugs injected in case $\mathbb{D}$ above.) The remaining 42 rules were modified to only trigger when the denominator was known to be non-zero, either by adding a predicate to the rule, or by exploiting the TRS's ability to track constant bounds and alignment of subexpressions. Three examples of now-incorrect rules were:

$$\begin{aligned} (x/y) * y + (x \,\%\, y) &\rightarrow_R x \\ -1/x &\rightarrow_R \texttt{select}(x < 0, 1, -1) \\ (x + y)/x &\rightarrow_R y/x + 1 \end{aligned}$$

The first was modified to:

$$(x/c_0) * c_0 + (x \,\%\, c_0) \rightarrow_R x \text{ if } c_0 \neq 0$$

and the other two were constrained to only trigger when the denominator is known to be non-zero via other means.

The cases discussed in Section 5.1.1 all concern fixing existing problems while not introducing new ones. By giving a proof of soundness, showing that the ruleset is correct and that the rules are cycle-free, we also remove two entire classes of future bugs. For reference, over the life of the Halide project there have been 14 pull requests that fix incorrect rules, and 3 pull requests that modify rules in order to avoid cycles. Fixing a reduction order also guarantees that no new cycles can be introduced as long as new rules obey this order; without such a guide, it is possible to introduce a rule that would close a loop in some sequence of existing rule applications and cause a cycle, resulting in infinite recursion during compilation.

5.2.2 What is the Best Way to Use Synthesis and Verification in Development? We have found that human experts can best leverage the strengths of the synthesis tool by using it as an assistant: for example by synthesizing a rule and then generalizing or simplifying it by hand, or by writing a rule and asking the tool to synthesize a valid predicate. Most importantly, this avoids committing new bugs, but it also accelerates rule development. Halide developers report that adding new rules by hand takes about 30 minutes per rule starting from sample input expressions through final review. Starting from the same input expressions, the synthesis tool can produce a batch of 10 verified rules in about 5 minutes.

Here we survey some cases from recent Halide development where the synthesis machinery was used in this way. The diffs are available as $\mathbb{F}$ through $\mathbb{I}$ in supplemental material or on the Halide project website[5].

In case $\mathbb{F}$ a Halide developer encountered expressions that seemed like they could be simpler while working on real code, and rather than inventing a rule from scratch searched the logs of the synthesizer project for a known-correct synthesized rule that handled the case in question. This added four new rules that are variants of:

$$\mathtt{max}(y, z) < \mathtt{min}(x, y) \;\rightarrow_R\; \mathit{false}$$

In case $\mathbb{G}$ a developer wrote 24 new rules, proving them on paper, and then checked their work by resynthesizing the predicates using the synthesizer, ensuring that the synthesized predicates agreed with the human's and that those predicates were as broad as possible. Here the synthesis machinery served as a reviewer of rules rather than an author. Eight of these rules were in fact manual rederivations of the synthesizer's output on case $\mathbb{D}$ above. The remaining 16 are generalizations that add constant terms. One example:

$$\mathtt{min}(y + c_0, z) < \mathtt{min}(y, x) \;\rightarrow_R\; \mathtt{min}(z, y + c_0) < x \text{ if } c_0 < 0$$

Halide endeavors to be a safe language, meaning that certain things are checked at compile time or runtime rather than being undefined behavior. In case $\mathbb{I}$, Halide was changed such that instead of asserting that no output value has a dependency on any out-of-bounds input values, it now asserts the stricter condition that no out-of-bounds loads occur on the input, even if those loaded values cannot possibly affect an output.

This is a harder thing for the compiler to check, and analysis often conservatively found that it was possible for code to read out of bounds, when in fact it would not. The Halide developers ameliorate this in part by minimizing the number of non-monotonic expressions using aggressive simplification. In total 59 new rewrite rules were added as part of this change. Eight of these were synthesized automatically by copy-pasting a non-monotonic expression from a bug report into the synthesis machinery. Another 28 came from generalizing those rules by hand and then verifying the result. Twenty more were written by hand and then verified. Finally, there were three rules that could not be verified, because they involve the interaction of division and modulus. During this process a large number of bugs were found in human-written early versions of these rules. We found that with the verifier and synthesizer in hand, humans work quickly and rely on the machinery to catch their mistakes.

Generalizing from these four cases, we have found that having the verifier and synthesizer available as tools reduces the number of bugs committed, uncovers and fixes old bugs, and helps developers work more quickly by not only mechanizing correctness checking but also by synthesizing correct code. We also found that the guarantees these tools provide mean that the developers can make large changes to the compiler with confidence. Anecdotally, developers also report that eliminating these classes of bug makes triaging new issues simpler, because they could now not possibly be due to an incorrect rule or an infinite loop in the term rewriting system.

5.3 Using the Synthesizer to Prevent Future Issues

5.3.1 Can Synthesis be Used for Large-scale Improvements of the TRS? Although we now have a guarantee of soundness, we have no such guarantee of completeness. There are almost certainly Halide compilations for which the addition of some desirable rule would strengthen the TRS enough to unlock some optimization or achieve a tighter bound on some region. However, we don't know what they are because no human has encountered them yet (or more likely, no human has been sufficiently motivated to submit a bug report for them yet).

We attempted to probe for such opportunities for improvement using fuzzing. We selected the 12 most complex example applications in the open source repository, and generated 64 random schedules for each using the autoscheduler [Adams et al. 2019], which can be configured to generate random likely-good schedules. This produced 768 separate compilations. We also instrumented most of the Halide code at Google for an additional 5032 compilations, this time using the original human-written schedules. Note that this is qualitatively different to randomly-generated schedules: we do not expect Halide users at Google would check in code that behaves poorly due to an issue with the compiler.

We instrumented these compilations to log expressions that might represent a TRS failure of some kind. From each compilation we log all integer expressions found that are non-monotonic with respect to a containing loop, and all failed proof attempts made during compilation. That is, we log all boolean expressions passed to the TRS in the hope that they will reduce to the constant true so that some optimization can correctly be performed. This resulted in a corpus of roughly one hundred thousand unique expressions.

Using this corpus as input, we synthesize new rewrite rules, add all rules found back to the ruleset, and rerun all compilations to gather new expressions, repeating this process until convergence. In total we created 4127 new rewrite rules in this way, more than quadrupling the number of rules in the TRS. For some examples, refer to Table 2.

We then generate a fresh set of 256 random schedules per application (to avoid testing on our training set), and compile and run all the generated code, looking for any compilations which behave significantly differently between the baseline condition (compiled using unmodified Halide) and the test condition (compiled with the 4127 additional rewrite rules). The interesting findings are summarized below.

Adding new rules lowers peak memory usage by up to 50%. Halide sizes internal allocations using symbolic interval arithmetic, which (as described in Section 4.2) is prone to overestimating bounds when expressions do not either monotonically increase or decrease with respect to some containing loop. By extending the TRS, we automatically fix 197 cases where this kind of error increases the peak memory usage of an application at runtime by more than 10%, including one case where the increase was more than a gigabyte. This represents nearly 6% of all compilations tested. We believe this captures a widespread problem, as instances of overallocation are a common source of complaint from users. See Figure 4 for the full distribution.

Term rewriting systems written without verification have bugs. On our initial run of this experiment, 55 compilations (1.6%) crashed at runtime with memory corruption errors in the baseline condition (no new rules added). We traced this to an incorrect transformation in a separate, unverified TRS in the Halide compiler (the "solver"). This bug had existed for four years, but had only recently become an important code path due to change $\mathbb{I}$ mentioned above. The incorrect transformation was $\texttt{min}(x-y, x-z) \rightarrow_R x - \texttt{min}(y, z)$, which should be $\texttt{min}(x-y, x-z) \rightarrow_R x - \texttt{max}(y, z)$. If this secondary TRS had been written using verification, this bug would never have been introduced. We intend to formalize and verify this secondary TRS next to flush out any other bugs lurking therein.

The TRS scales well with the number of rules. Remarkably, more than quadrupling the size of the TRS increased total compile times by only 0.3%. On further examination we found that the additional rules increased the amount of time spent inside the TRS by 30%, and that only 1% of the the total compile time of the average Halide program is spent inside the TRS.

We did not find significant effects on runtime of the generated code or code size. We also found no significant differences on any metrics within the Google corpus. This may be because the random

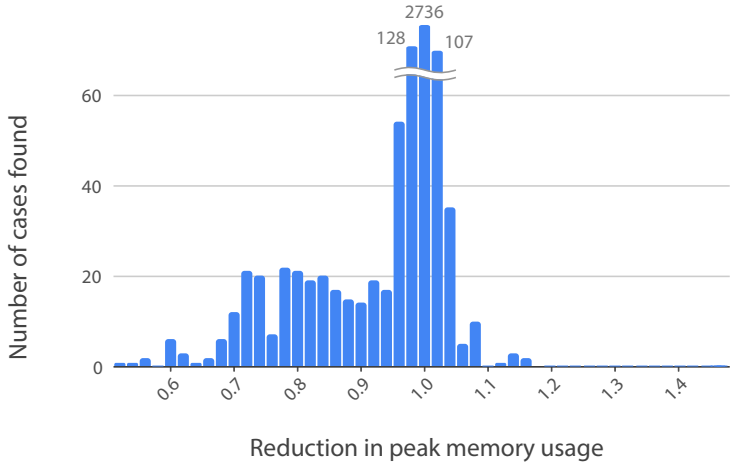


Fig. 4. Reduction in runtime peak memory usage of 3072 pieces of compiled code when 4127 synthesized rules are added to the TRS. The x-axis shows $\frac{after}{before}$, so below 1.0 means the synthesized rules reduced memory consumption. In 197 cases, peak memory usage drops by more than 10%.

schedules we generate are especially complex compared to human-written ones, or simply because humans don't commit code that causes the compiler to misbehave.

While adding this many new rules does not come at any significant cost we could measure to users of Halide, it increases the compile-time and code size of the compiler itself, so we do not propose adding all of these rules to the TRS permanently. Prior to proposing inclusion, we plan to triage the rules to select only those that are necessary to gain the peak memory reductions described above.

These results show that having verification and synthesis available as a tool for compiler authors fixes existing bugs, prevents entire classes of new bugs, and even helps compiler writers change the semantics of their language with confidence. We intend to continue to use verification and synthesis to maintain the TRS, and based on this experience plan to expand its use elsewhere in the compiler.

6 LIMITATIONS & FUTURE WORK

For this work, we considered only the subset of the term rewriting system that is used to prove properties over infinite integers; the full TRS includes rules for simplifying expressions with floating point values as well as rules for fixed-bitwidth integers. As a result, we do not consider cases where the TRS must also reason about whether overflow can occur. Extending our improvements and automation to such rules could be done in future work.

One major limitation of the synthesis process we use is that our solver, Z3, often cannot reason about expressions with divisions or modulo where the right operand is a variable. Though we work around this to synthesize rules with generalized predicates on right hand side constants, the overall synthesis machinery cannot generalize these to non-constants. Extending the synthesizer may be more tractable for rules that operate on integers with finite bitwidths.

The Halide TRS is used both to prove expressions true or false and to simplify expressions to more easily optimizable forms, but these two use cases are not always aligned. One could imagine two separate rewriters with different rulesets and reduction orders. One interesting direction for

future work might be to use the synthesizer to create these two rulesets, which would otherwise be a very human-effort intensive task.

Although we do not currently synthesize rules that contain vector operators, they are not incompatible with our approach. Since our reduction of Halide expressions to SMT2 formulas models vector expressions as integers, we would need to add a typechecking step to ensure correctness in the synthesis process, which we leave as future work.

The priority in which rules are considered for matching clearly can have performance implications, but evaluation and tuning is left as future work, and may require modifying our ordering or creating different orderings for different uses of the term rewriting system.

Finally, while enhancing the Halide TRS was not able to improve the performance of the mature Halide compiler, we suspect this may not be the case for other, less well-exercised compiler backends. In future work we plan to use this technique to enhance the ruleset used to compile Halide applications to GPU code or to the Hexagon ISA.

7 RELATED WORK

Perhaps the closest related work is the Alive project [Lopes et al. 2015; Menendez and Nagarakatte 2017]. The fundamental difference between Alive and this work is that Alive works within the decidable theory of bitvectors, while (because of Halide semantics) we must use the undecidable theory of integers; this constraint is the major reason for many of our design choices. In addition: Alive verifies optimizations (and Alive-Infer synthesizes preconditions), while we synthesize rewrites and predicates, as well as verify them; Alive must contend with more types of undefined behavior, which the Halide expression language need not consider; and Alive uses a simple reduction order in which all optimizations reduce program size, while our termination proof is more complex. We originally tried synthesizing rule predicates with the approach used by Alive-Infer but were not successful: using Z3 to generate positive and negative examples did not scale for us, requiring seconds to minutes per query due to the underlying theory of integers. Moreover, queries with division/modulo over the integers often did not work at all, simply returning "unknown."

Most recently, leveraging a TRS along with synthesized rules has been applied to optimizing fully-homomorphic encryption (FHE) circuits [Lee et al. 2020]. This system synthesizes equivalent circuits with lower cost from small example circuits, then applies the equivalences in a divide-and-conquer manner; the rewrites do not contain preconditions. In further contrast to our work, the domain of FHE yields a simple cost function (the depth of nested multiplications in the circuit), and the underlying theory of boolean circuits is decidable.

An *equivalence graph* or egraph, as introduced by Nelson [1980], is a data structure used to compute applications of the rules of a term rewriting system. The algorithm builds up equivalence classes by successively applying all rules to all expressions within those classes, then queries to see if two expressions are equivalent by checking if they are present in the same class. Like our algorithm, it does not backtrack, but the egraph can require significant amounts of memory, which our algorithm avoids.

Herbie [Panchekha et al. 2015], a tool for improving the accuracy of floating point arithmetic, uses an egraph term rewriting system made up of a small library of axioms to find repairs once a fault has been localized. Herbie assures termination by bounding the number of rewrites their system may apply, and achieves good performance by pruning the expression search space and applying rewrites only to particular expression nodes.

Besides the closely-related projects described above, program synthesis has been applied to term rewriting systems in several domains. Swapper [Singh and Solar-Lezama 2016] synthesizes a set of rewrite rules to transform SMT formulas into forms that can be more easily solved by theory solvers, similar to the use of the Halide TRS as a simplifier, using the SKETCH tool. Butler et al.

[2017] learns human-interpretable strategies (essentially rewrite rules) for puzzle games such as Sudoku or Nonograms and Butler et al. [2018] finds tactics for solving K-12 algebra problems, both using a CEGIS loop similar to our synthesis process. None of these address termination, although Swapper likely screens out non-terminating rulesets through its autotuning step. The Butler works both focus on synthesizing small, highly general rulesets that are similar to human rewriting strategies, unlike the Halide TRS which tolerates very large rulesets. The λ^2 tool [Feser et al. 2015] for example-guided synthesis performs inductive synthesis from examples, using a combination of inductive and deductive reasoning combined with enumerative search. While our rewrite rules do not have the benefit of examples, it may be possible to apply this technique to obtain more sophisticated predicate synthesis for our rewrites.

Superoptimization, a process of finding a shorter or more desirable program that is semantically equivalent to a larger one, is similar to our work synthesizing right-hand side terms for candidate LHSs. STOKE [Schkufza et al. 2013] uses Monte Carlo Markov Chain sampling to explore the space of x86 assembly programs, while Phothilimthana et al. [2016b] describes a cooperative superoptimizer that searches for better programs using multiple techniques in a way that allows them to learn from each other. Souper [Sasnauskas et al. 2017a] is a recent synthesis-based superoptimizer for LLVM, which was used in evaluating the effectiveness of Alive-Infer's precondition synthesis.

PSyCO [Lopes and Monteiro 2014] synthesizes preconditions that guarantee a compiler optimization is semantics-preserving, using a counterexample-driven algorithm similar to our rule CEGIS loop (although not like our predicate synthesis algorithm). PSyCO finds the weakest precondition from a finite language of constraints, while the space of our predicate search is theoretically inifinite but in practice bounded by our iteration limit. PSyCO must reason about side effects by tracking read and write behavior in optimization templates, while our expression language is side effect–free. More recently, Proviso [Astorga et al. 2019] finds preconditions for C# programs using an active learning framework composed of a machine learning algorithm for decision trees as a black-box learner and a test generator that acts as a teacher providing counterexamples. Like this work, the logic of preconditions they synthesize is in an undecidable domain.

SMT solvers seek to achieve practical results in theoretically challenging problems, such as the theory of non-linear integer arithmetic. For example, Jovanović [2017] describes a satisfiability procedure for NLIA that is effective in practice and implements it in the Yices2 solver. In this work we leverage Z3's NLIA solving abilities and extend synthesis to include certain types of non-linear expressions by using constants as operands and later generalizing.

A different use of term rewriting systems in pipeline scheduling language compilers is demonstrated in [Hagedorn et al. 2020], in which an algorithm and a schedule are rewritten using a TRS to low-level code, which can be then compiled for high performance.

8 CONCLUSION

In this work, we improved the Halide term rewriting system by applying formal techniques to verify rewrite rules and to ensure termination. In this process, we discovered 4 incorrect rules (which is remarkable, given that the TRS has been extensively fuzz tested) and 17 cases where we could make rules more general, as well as 8 rules that could potentially cause termination problems. We built an automated synthesis-based pipeline for constructing new rewrite rules and demonstrated that it can produce better rewrite rules than hand-authored ones in five historical bug fixes. We further described four case studies in which the synthesizer has served as an assistant to a human compiler engineer. Finally, we showed that applying the synthesizer can proactively improve weaknesses in the Halide compiler by bulk-synthesizing a large number of rules automatically and showing this ruleset lowers peak memory usage of compiled code with nearly no increase in compilation time.

Our improvements guarantee the soundness of the term rewriting system and increase its robustness and coverage, with no significant costs or downsides we could identify. Indeed, augmenting this part of the Halide compiler using verification and synthesis seems to constitute a free lunch, and so we intend to formalize other parts of the compiler as verified term rewriting systems as well.

ACKNOWLEDGMENTS

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (July 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. 2019. Learning Stateful Preconditions modulo a Test Generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA.

Franz Baader and Tobias Nipkow. 1999. *Term rewriting and all that.* Cambridge university press.

Raymond T Boute. 1992. The Euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 14, 2 (1992), 127–144.

Eric Butler, Emina Torlak, and Zoran Popović. 2017. Synthesizing interpretable strategies for solving puzzle games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*. ACM, 10.

Eric Butler, Emina Torlak, and Zoran Popović. 2018. A Framework for Computer-Aided Design of Educational Domain Models. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 138–160.

Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proceedings of TACAS (LNCS)*, Nir Piterman and Scott Smolka (Eds.), Vol. 7795. Springer.

The Coq Development Team. 2019. *The Coq Reference Manual, version 8.10.* Available electronically at http://coq.inria.fr/doc.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 269–282.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 11.

Saul Gorn. 1967. Handling the Growth by Definition of Mechanical Languages. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '67 (Spring))*. ACM, New York, NY, USA, 213–224. https://doi.org/10.1145/1465482.1465513

Bastian Hagedorn, Johannes Lenfers, Thomas Kœhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

Dejan Jovanović. 2017. Solving nonlinear integer arithmetic with MCSAT. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 330–346.

Donald E Knuth and Peter B Bendix. 1983. Simple word problems in universal algebras. In *Automation of Reasoning*. Springer, 342–376.

DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2020. Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*.

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 22–32.

Nuno P Lopes and José Monteiro. 2014. Weakest precondition synthesis for compiler optimizations. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 203–221.

Henry Massalin. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (1987), 122–126.

David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: Data-Driven Precondition Inference for Peephole Optimizations in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 49–63. https://doi.org/10.1145/3062341.3062372

C.G. Nelson. 1980. Techniques for program verification[Ph. D. Thesis]. (1980).

Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 1–11.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016a. Scaling Up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 297–310. https://doi.org/10.1145/2872362.2872387

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016b. Scaling up superoptimization. In *ACM SIGARCH Computer Architecture News*, Vol. 44. ACM, 297–310.

Alex Reinking. 2019. *Formal Semantics for the Halide Language*. Master's thesis. University of California at Berkeley.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017a. Souper: A Synthesizing Superoptimizer. arXiv:1711.04422 [cs.PL]

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017b. Souper: A Synthesizing Superoptimizer. *CoRR* abs/1711.04422 (2017). arXiv:1711.04422 http://arxiv.org/abs/1711.04422

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 305–316.

Rohit Singh and Armando Solar-Lezama. 2016. SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 185–192.

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science)*, Zhenjiang Hu (Ed.), Vol. 5904. Springer, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3

Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices* 49, 6 (2014), 530–541.