Augmenting and Synthesizing Term Rewriting Systems for Use in Compilers

Julie L. Newcomb

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:
Rastislav Bodik, Chair
René Just
Zachary Tatlock
Duane Storti

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Augmenting and Synthesizing Term Rewriting Systems

Julie L. Newcomb

Chair of the Supervisory Committee:
Professor Rastislav Bodik
Computer Science and Engineering

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.

We then turn from the goal of improving a handwritten term rewriting system to synthesizing an entirely new one. We argue that an artifact from a TRS termination proof, a special

type of order over terms known as a reduction order, is an effective and expressive means of writing a specification for a term rewriting system. We demonstrate its own on a case study of another component of the Halide compiler called the variable solver. We present a synthesis process that takes a specification written in the form of a reduction order and a set of sample input expressions and produces a new ruleset. Furthermore, we show that we can use this synthesis process to allow users to experiment with different specifications by automatically generating new TRSs and evaluating them empirically, without ever handing to write a rule by hand.

TABLE OF CONTENTS

Page

Chapter 1: Introduction 1
1.1 Maintaining an existing term rewriting system 3
1.2 Synthesizing a new term rewriting system 7
1.3 Outline of dissertation 8

Chapter 2: Background 10
2.1 Term Rewriting Systems 10
2.2 Halide and the Halide expression language 11
2.3 Term rewriting systems in the Halide compiler 12

Chapter 3: Verification 23
3.1 Correctness 23
3.2 Termination 24
3.3 Related Work 30

Chapter 4: Augmenting a Term Rewriting System through Synthesis 33
4.1 Synthesizing rewrite rules 33
4.2 Evaluation of Simplifier TRS Synthesis 42
4.3 Limitations & Future Work 53

Chapter 5: Specifying and Synthesizing a New Term Rewriting System 54
5.1 Rationale for synthesis pipeline 55
5.2 The variable solver reduction orders 63
5.3 The synthesis algorithm 71
5.4 Evaluation of synthesized TRSs 76
5.5 Related work 83

Chapter 6: Future Work and Conclusion 86
6.1 Future Work 86
6.2 Conclusion 91

Appendix A: Halide expression grammar 93

Appendix B: The full Halide simplifier reduction order 95

Bibliography 99

ACKNOWLEDGMENTS

I would like to express my sincerest gratitude

to Shoaib Kamil and Andrew Adams, my mentors at my internship at Adobe Research, who have been insightful and inspiring collaborators ever since;

to all of my fellow students in the PL synthesis group, past and present, at both the University of Washington and the University of California at Berkeley, who have been unfailingly welcoming and supportive from even before I was accepted into the program;

and to my advisor, Ras Bodik.

DEDICATION

to the memory of Catherine Anne Newcomb

Chapter 1
INTRODUCTION

Halide is a widely used, open source image processing language intended to deliver very high performance in its compiled programs. Like many compilers, in order to apply optimizations, the Halide compiler must perform a variety of analyses of the input program'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 work sufficient to alter the computational complexity of the algorithm. The reasoning engines that make these judgments within the Halide compiler are crucial to exploiting opportunities for these optimizations. In fact, Halide relies so heavily on one of these engines 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.

Such a reasoning engine 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. Any solver in this theory is necessarily incomplete, but in general, the compiler can generate higher-performing code as the reasoning engine becomes more powerful.

- **High performance:** However, the engine 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 engine must be deterministic.

Halide addresses these reasoning tasks with *term rewriting systems*. Using a custom algorithm that greedily applies rules in a fixed order and keeps only the expression being currently rewritten as state, a term rewriting system (TRS) can provide 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 the reasoning engines' power by refining the ruleset and adding new rules by hand.

However, maintaining term rewriting systems 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, the precise goal of a reasoning engine, or the strategy it uses to achieve that goal, may be recorded only in the form of the rewrite rules themselves. If developers are not familiar with an entire ruleset, or the full variety of applications in which the ruleset is invoked, they may inadvertently add a rule that makes the simplifier worse for some of its usecases.

In this work, we propose to show how authoring and maintaining term rewriting systems can be made easier through the use of formal methods and program synthesis. We will demonstrate how these techniques can support all degrees of human involvement: from verifying the correctness of an existing handwritten term rewriting system, to learning new rules to augment an existing system, to synthesizing a new ruleset entirely from scratch.

In this work we make the following contributions:

- *We show formal methods and program synthesis can usefully assist the authors of term rewriting systems.* Formal proofs of soundness and guarantees of termination can find bugs in large, widely-deployed term rewriting systems, and automatic program synthesis can be used to find new rewrite rules to augment these systems. Our work has identified and patched bugs in the Halide codebase, been used by Halide developers to support code maintenance, and automatically produced new rewrite rules that have been merged into the compiler.

- *We present an effective means of encoding a specification for the synthesis of term rewriting systems.* Besides providing a proof of termination, we claim that reduction orders can serve as a useful formalization of the intent behind a term rewriting system, and can be used as a specification when synthesizing rewrite rules.

We evaluate our work in two case studies. In the first, we investigate the first claim by using formal methods to enhance a pivotal term rewriting system within the Halide compiler called the *simplifier*. We found that we were able to identify bugs, prove the absence of future errors, and increase the rewriting power of the system. We review our evaluation of the first claim in section 1.1.

In the second case study, we take a look at a smaller and less mature term rewriting system in Halide called the *variable solver*. Previously, we synthesized new rules for the simplifier largely guided by its large existing ruleset. Here, we plan to evaluate our proposed means of writing specifications for term rewriting system by synthesizing a ruleset entirely from scratch. We lay out our means of writing specification and our evaluation of the synthesized rulesets in chapter 5.

1.1 Maintaining an existing term rewriting system

The Halide compiler contains a hand-authored term rewriting system commonly called the simplifier. It is used in many cases within the compiler for making expressions shorter and in a form better suited to downstream uses. Sometimes the simplifier is used as a prover:

the truth value of an expression is checked by rewriting it with the simplifier to see if it will be rewritten to the constant `true`. At the time we began this work, the simplifier was fairly mature: it had been deployed in production for over a year, was both unit tested and fuzz tested, and comprised almost a thousand rules.

Is it necessary to apply formal methods to the authoring of term rewriting systems? In chapter 3, we show that applying formal methods can identify and remedy real issues in the simplifier TRS. Conversely, we observe that implementing expression-transforming code as a term rewriting system is useful precisely because it allows easy integration with formal methods. We also show in chapter 4 that we can improve this term rewriting system by synthesizing new rewrite rules automatically and that this synthesis process can be used in the compiler's development process. This work has been published at OOPSLA 2020 [Newcomb et al., 2020].

1.1.1 Proof of soundness

First, we formally verify that each rule in the Halide simplifier ruleset is semantics-preserving, demonstrating that proofs of soundness are possible despite the lack of a decision procedure for our theory. We do this by modeling the semantics of the Halide expression language in SMT2. We then implemented a pretty-printer to translate each rule in the simplifier ruleset to an SMT query to be checked by the solver Z3 [De Moura and Bjørner, 2008]. About 12% of the ruleset could not be verified by Z3, and we proved those rules correct by hand using the proof assistant Coq [Coq Development Team, 2019]. Even though the code had been deployed for over a year and had been fuzz-tested, we found that four of the existing rules were incorrect and submitted patches. In constructing the Coq proofs, we also noticed that 17 rules had predicates that were overly conservative, and submitted patches for the relaxed predicates as well.

While this project was ongoing, the Halide semantics for division was changed: division by zero was no longer undefined behavior, but now returned zero. It was simple for us to amend our modeled semantics and rerun rule verification in Z3. Again about 12% of the ruleset

had to be hand-proven using Coq, but we were able to leverage many of the existing proofs from our previous verification. Our verification identified 44 rules which were not correct under the new semantics and 37 rules whose predicates could be relaxed under the semantics change. This shows the value of our formal methods infrastructure, which allowed Halide developers to push a fairly major change with a higher degree of confidence in the soundness of the simplifier than could have been achieved with either manual testing or fuzzing.

1.1.2 Proof of termination

The Halide simplifier algorithm successively applies rewrite rules until the resulting expression can no longer be rewritten. Thus, if some sequence of rewrites to some input expression can form a cycle, the simplifier algorithm will not terminate. These non-termination errors have been observed in the past (resulting in the compiler throwing a stack overflow error and crashing). Without a specific input expression on which to reproduce a cycle, it is very difficult to examine a set of around a thousand rules and find a subset on which a cycle could occur; even once a cycle has been identified, it is difficult to know the best way to repair it. Without insight into how these cycles might be formed, it was possible to introduce The ruleset was thus very brittle; deleting, altering, or reordering existing rules has caused new non-termination errors in the past as well.

A term rewriting system can be proven to terminate using a formalism called a *reduction order* [Baader and Nipkow, 1999]. A reduction order is an order over a language of terms; if for every rule in a term rewriting system, we can show that the rule's left-hand side is strictly greater than the right-hand side in this order, then we know that there can be no set of rules that can form a cycle, and thus the ruleset must always terminate. A reduction order is distinguished by a few special properties to ensure that these ordering holds no matter how input expressions are matched to the left-hand side term, as well as when a rule is used to rewrite a subterm inside of a larger input expression.

We devised a reduction order that fit as many of the existing simplifier rules as possible. Eight rules could not be fit to our ordering, and we submitted patches to either delete or

modify them. Once this was done, not only was a class of bugs eliminated, but the ruleset could now be safely modified without fear of introducing new non-termination behavior. It was also now safe to add any new rule so long as the rule conformed to the reduction order.

We observe that this reduction order serves to encode 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. Maintaining this order as an invariant over any future rules means any additions to the ruleset will not undo progress made by existing rules.

1.1.3 Strengthening the ruleset

We can empirically observe that the simplifier ruleset is not sufficiently powerful to deal with all expressions it may be called upon, by instrumenting the compiler and logging any expressions that the simplifier cannot further solve. In the past, ruleset authors might look at these failed expressions and write rules to address them by hand. In chapter 4, we automated this progress by synthesizing these rules instead.

However, 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 synthesize rules through a pipeline that takes these expression as input and uses CEGIS loops to synthesize an equivalent right-hand side and (if necessary)

a predicate guard that indicates when it is safe to apply the rule. Our pipeline produces more general rules by mining input expressions for larger patterns and by replacing constants with fresh variables. We also find variants on synthesized rules by applying associativity and commutitativity laws to their left-hand sides. We choose candidate left-hand sides for rules from this corpus and synthesize equivalent right-hand terms that obey the termination order. In our experiments, we found that our synthesis pipeline could produce patches to ruleset bugs as good or better than those that were authored by hand. We also observed Halide developers integrating the rule synthesizer into their workflow. Our synthesis procedure is also capable of finding 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.

1.2 Synthesizing a new term rewriting system

For the Halide simplifier, we synthesized individual rules to augment an already mature term rewriting system. Next, we demonstrate the synthesis of term rewriting systems completely from scratch. We do this using the Halide *variable solver* as a case study. The variable solver takes a Halide expression and a variable found within that expression (the *target variable*) and attempts to eliminate as many instances of that variable as it can and isolate the variable to one side of the expression as best it can. (If the expression is an equality, this would consitute 'solving' the expression for the chosen variable.)

An idealized version of our proposed synthesis procedure is laid out in algorithm 1. This procedure requires a goal function, which returns true if a term is in some desired form and false otherwise; a semantic equivalence relation; and a reduction order over terms, as well of a set of training expressions E, which should be representative of the types of expressions the term rewriting system will take as input and a set of test expressions S on which to evaluate the behavior of the term rewriting system.

The key input to the synthesis procedure is the reduction order over terms, which serves

as a specification for the synthesis of rules. The reduction order is a means of formalizing either the intent of the term rewriting system—the form into which the TRS should rewrite expressions—or the strategy by which the TRS will rewrite expressions to be closer to the desired form. While TRS authors are likely to have good intuition as to what rewriting strategy their system should employ, choosing the best reduction order for the task is still not a trivial exercise. Our synthesis process gives users the chance to experiment and determine which reduction order serves their purpose best; rather than hand-turning rules, they can synthesize a ruleset entirely from scratch for a given reduction order and evaluate it directly.

We have used the Halide variable solver as a case study to demonstrate how synthesis can be used to evaluate reduction order specifications. We lay out the refined synthesis algorithm, with some discussion of how best to search the space of candidate rules, and show how reduction orders can be encoded efficiently as metasketches. We evaluate our claims by synthesizing two rulesets from two different reduction orders in a fully automated way and evaluate their performance against the existing handwritten solver on a suite of benchmarks.

1.3 Outline of dissertation

In this dissertation, we first lay out some necessary background in chapter 2: we give a formal definition of term rewriting systems and related concepts, describe the two TRSs within the Halide compiler that will serve as our case studies, and provide some discussion of the rationale behind some of the Halide rewriting algorithm design choices. In chapter 3, we detail our work proving the correctness of the Halide simplifier, showing that the TRS is both semantics-preserving and guaranteed to terminate. Next, in chapter 4, we show how we were able to increase the robustness of the simplifier TRS by using a key artifact from our termination proof, a reduction order, as a specification to synthesize new rewrite rules. We demonstrate that our synthesis process can produce high-quality bug fixes, discuss some ways in which our techniques have been used to support Halide developers' work, and perform a large-scale experiment synthesizing an additional 4,000 new rules, with the effect of reducing peak memory usage of benchmark programs with no increase in compilation times.

In chapter 5, we synthesize a ruleset for the Halide variable solver entirely from scratch, using our method of specifying rulesets through reduction orders. We discuss the efficient encoding of reduction order specifications using sketches, and furthermore show that our synthesis pipeline gives users an automatic tool for empirically evaluating alternative specifications without ever having to manually write rules. Finally, in chapter 6, we conclude with some directions for future work.

Chapter 2
BACKGROUND

We define term rewriting systems and give some background. We briefly describe Halide and the Halide expression language we will be operating on throughout this work. We then describe the uses of term rewriting systems within the Halide compiler, including the term rewriting algorithm used by the compiler, the requirements that led to its design choices, and the two term rewriting systems we use as case studies in this work, the simplifier and the variable solver.

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 $\Sigma = \{\clubsuit, \diamondsuit\}$ and V is an infinite set of variables.

Let the term rewriting system R consist of a single rule:

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

We use R to rewrite the term

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

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:

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

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

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

2.2 Halide and the Halide expression language

The Halide compiler contains term rewriting systems that manipulate terms in the Halide expression language. This language 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. The Halide expression language contains the usual arithmetic operators and uses the Euclidean definition for division and modulo. It also contains the boolean operators and, or, and not, as well as the usual comparators. The `select` operator takes three arguments and behaves like an if-then-else statement, returning the second argument if its first evaluates to true and the third argument otherwise. There are two operators that can create vectors from scalars: `broadcast`(v, s), which creates a vector of size s with each value initialized to v, and `ramp`(v, s, k), which creates a vector of size s

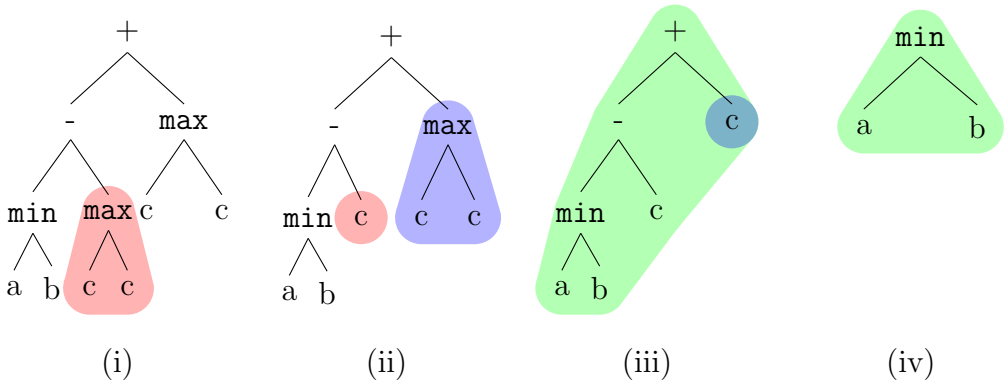


Figure 2.1: We demonstrate the Halide rewriting algorithm using a TRS $R = \{\texttt{max}(x, x) \rightarrow_R x, (x - y) + y \rightarrow_R x\}$ and an expression $\texttt{min}(a, b) - \texttt{max}(c, c) + \texttt{max}(c, c)$. The algorithm attempts to simplify all subtrees bottom up; here, no rule applies 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).

where the first value is initialized to v and each subsequent value is increased by the stride k. Most operators that take integer or boolean arguments can also take vector arguments whose values are integer or boolean type respectively. For the full Halide expression grammar, see appendix A.

2.3 Term rewriting systems in the Halide compiler

2.3.1 The Halide rewriting algorithm

A term rewriting system is simply a set of rewrite rules; when we use a TRS to transform an expression, we also need to define a procedure for applying those rewrite rules to the expression. In a TRS, 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 e-graph. 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.

The Halide term rewriting algorithm simplifies an input expression in a depth-first, bottom-up traversal of the expression AST. 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 recurses and starts applying rewrites on the newly rewritten expression. If no rule matches the subtree, the traversal continues; when the entire expression cannot be simplified further, the rewritten expression is returned. See Figure 2.1 for a worked example.

The rewrite rules optionally contain a predicate guard. These guards contain only special variables whose values are known at the time of rule application[1]; 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

[1]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.

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 The rewriting algorithm's design requirements

Because a reasoning engine implemented as a term rewriting system may be invoked thousands of times in a single compilation, the rewriting algorithm outlined above was designed to be fast and use very little memory. For that reason the algorithm is non-backtracking and keeps only one expression in state, sacrificing some potential solving power for performance. In this work, we operate within the scope of Halide's TRS algorithm and work to make the TRSs used by that algorithm as correct, general, and robust as possible. Because the space of expressions we consider constitutes an undecidable theory, any complete TRS is impossible. Instead, we strive to improve correctness by ensuring the TRSs will always terminate on any expression and that each individual rewrite preserves semantics; and we improve generality by expanding the rulesets 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.

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 as its reasoning

Table 2.1: 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

engine. 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 both provably true expressions and expressions that are not provably true. Z3 could prove approximately 30% more expressions true (within a 60 second timeout), but was starkly less performant. As shown in Table 2.1, 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 4.2.3 for an evaluation of the effects of adding newly synthesized rules on the performance of the compiler.

Finally, Z3 is a very powerful general-purpose solver, but Halide TRSs can be targeted for their specific use cases. For example, when rewriting an expression to be shorter, gathering like terms in some cases can actually prevent Halide or LLVM optimizations from applying. The Halide TRSs use domain-specific strategies to guide expressions into more optimizable forms and can be changed or tuned as needed if further optimizations are discovered.

2.3.3 The Simplifier

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 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.

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 constant 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 $0 < y \mid\mid 0 < 1$, 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.

Simplifier failures have adverse results on the compiler, making it unpredictable and thus difficult for Halide program authors to reason about the performance of their schedules. 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 simplifier are essential qualities for making the compiler robust and able to generate fast code.

2.3.4 The Variable Solver

The Halide compiler contains a `SolveExpression` class that implements functionality that we will refer to as a variable solver in this work. It takes an expression and a variable name as inputs and 'solves' the expression for that variable, isolating it on the left of an expression.

`SolveExpression` currently rewrites expressions using a recursive visitor that "matches" the expression using if statements. We have translated this into a term rewriting system that contains about a hundred rules. This system is much smaller and less mature than the simplifier, and using synthesis to author new rulesets will potentially result in much bigger performance gains than we saw with the more highly optimized simplifier.

We formally define the purpose of the variable solver as follows. Let $|t|_x$ represent the number of occurrences of the variable x in the term t. We use the special variable x^t to stand for the target variable, or the variable that the TRS is 'solving' for.

We say that a term t is in *solved form* if it is in the form:

1. $|t|_{x^t} = 0$ (the term t does not contain the target variable x^t). If we took the term $(x^t - x^t) + y$ and rewrote it to y, y would be in solved form, as it does not contain any instances of x^t.

2. $t = x^t$ (the term t is precisely the target variable x^t). If we took the term $x^t + (y * (z - z))$ and rewrote it to x^t, x^t would then be in solved form, as it consists of the target variable alone.

3. $t = x^t \odot t'$, where $\odot$ is any binary operator in the Halide expression language, and $|t'|_{x^t} = 0$. Terms in this form are only considered 'solved' if no term u in either of the above two forms exist such that $t =_e u$. If we took the term $(y + x^t) + z$ and rewrote it to $x^t + (y + z)$, it would then be in solved form. (Note that $x^t + (z + y)$ would also in solved form.) However, $x^t + (y - y)$ would not be in solved form, since a semantically equivalent term x^t is in the second form.

The variable solver in the Halide compiler is used to reason about the bounds over variables and about their dependencies. As a worked example, let's consider the TrimNoOps stage of the compiler, in which the compiler tries to identify regions of loops in which no work is performed and thus those loop iterations can be skipped completely.

Consider a simple Halide function over the variables x and y. Whenever $2 \cdot x$ is greater than y, the function returns 5, otherwise it returns nothing. The function is then tiled using a 4 by 4 tile size.

```
Func F;
Var x, y;
f(x, y) = select(2 * x < y, 5, undef<int>());

Var xi, yi;
f.tile(x, y, xi, yi, 4, 4);
```

The code is lowered to a set of nested for loops. The variables x and y now represent the number of tiles in their respective dimensions; at each iteration, they calculate the starting point for the next tile as the variables yi_base and xi_base. The variables xi and yi then iterate over each point in the current tile.

```
for(y=0; y < (y_extent+3)/4; y++) {
    yi_base = min(y * 4, y_extent - 4);
    for(x=0; x < (x_extent+3)/4; x++) {
        xi_base = min(x * 4, x_extent - 4);
        for(yi=0; yi<4; yi++) {
            for(xi=0; xi<4; xi++) {
                if (xi_base + xi)*2 < (yi_base + yi) {
                    f[idx] = 5;
                }
```

```
            }
        }
    }
}
```

Note that the body of the innermost for loop is an if statement. If the condition of the if statement is true, the loop will set the current location of the output (here idealized as idx) to the value 5. If the condition of the if statement is not true, then the loop will do nothing. Thus, if we can identify regions of the innermost loop where the condition of the if statement can never be true, we can optimize by skipping those regions entirely.

To see if we can identify such a region, we take the condition of the if statement:

```
(xi_base + xi) * 2 < (yi_base + yi)
```

and use the variable solver to 'solve' for the variable xi.

```
xi <= ((yi_base + yi) - (xi_base* 2) - 1)/2
```

Here the variable solver is successful in rewriting the condition in terms of xi. Furthermore, we are able to state the expression as an upper bound of xi. We can thus calculate the maximum value of xi for which the inner loop will perform any work.

```
xi_new_max = ((yi_base + yi) - (xi_base* 2) - 1)/2
```

We can then use this new maximum as a new extent for the loop, allowing us to skip loop iterations where no work will be performed:

```
    for(xi=0; xi<(xi_new_max + 1); xi++) {
        if (xi_base + xi)*2 < (yi_base + yi) {
            f[idx] = 5;
        }
    }
```

The variable solver is used in various other places in the compiler as well. For example, by 'solving' expressions within functions for relevant variables, the variable solver can remove spurious dependencies and allow the compiler to place computations over GPU code more efficiently. The variable solver is also used to attempt to prove associativity of functions. If the operations within a function can be shown to be associative (for example, a sequence of adds and multiplies), the compiler can rearrange the associative components in whatever order will be most efficient. 'Solving' such an expression for each variable in turn has the effect of normalizing it, making associativity easier to prove.

Chapter 3

VERIFICATION

We improve the Halide simplifier 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 Correctness

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). All integer and boolean operators can be coerced to vector operators on vector inputs of the appropriate type. Halide's TRS uses two vector-constructing operators, `broadcast` and `ramp`. `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.

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 44 rules were not correct under the new semantics and submitted a patch to amend them.

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[1], 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].

---

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

Theorem 3.2.1. *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 3.2.2. *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 $>_b$ 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, given in full in appendix B.

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, in a few exceptional cases, 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 may 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; this is discussed in greater detail in chapter 5.

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 built a script that automatically checks the full set of rules in the compiler to ensure they respect the reduction order.

3.2.1 Evaluation of verification work

Changes to the simplifier ruleset undergo a stringent development process: new rules are peer reviewed after they are proven on paper, and fuzz-testing is used to discover bugs. It is

thus reasonable to ask whether mechanized verification can add any value. Our verification discovered 4 previously-unknown correctness bugs and 17 instances of rules whose predicates were overly restrictive. The former bugs eluded the fuzzer; the latter are deemed too hard to detect so the fuzzer does not look for them. Some of the corrected rules are listed in more detail in table 3.1.

Furthermore, because the verification infrastructure was in place, it was possible to verify a change of division semantics without much additional effort, identifying 44 rules that were incorrect under the new semantics. This change to the semantics of Halide 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 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[2]. Two of them were in fact not related to division, but were instead the first discovery of the bugs injected in a patch after the initial ruleset verification. 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

[2] `https://github.com/halide/Halide/pull/4439`

exploiting the TRS's ability to track constant bounds and alignment of subexpressions. Three examples of now-incorrect rules were:

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

The first was modified to:

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

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 4.2.1 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.

3.3 Related Work

An *equivalence graph* or e-graph (introduced by [Nelson, 1980], see [Willsey et al., 2021] for a recent treatment) 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

Table 3.1: 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 $c_1 > 0 \wedge c_0 + 1 = c_1$	

backtrack, but the e-graph can require significant amounts of memory, which our algorithm avoids.

AProVE [Giesl et al., 2004, 2017] is a tool that automatically generates proofs of termination for term rewriting systems (as well as programs in Java, C, Haskell, and so on). It employs a variety of techniques for doing so. It may prove a TRS terminations through direct termination proofs, or finding a reduction order that fits all rules in the TRS, as we do in our work, searching classes of orders including path orders, Knuth-Bendix orders, and polynomial orders. It may also prove termination through dependency pairs (finding all instances in which terms of RHSs can unify with rule LHSs, then showing that a weakly monotonic ordering holds over all dependency pairs) or by abstracting rules by their effect on term height and proving that rule application must cause term height to decrease. It also employs techniques to remove portions of the ruleset that have no effect on termination and for reducing the size of the ruleset to make termination proof search more efficient. A proof of termination by term height abstraction would not be useful for synthesis, since it encodes no information about progress towards a goal state. A proof of termination through dependency pairs could do so, and since it uses weakly rather than strongly monotonic orders,

it could permit rewriting strategies that our technique cannot. However, this method requires reasoning over the full ruleset rather than individual rules, so using it as a specification for synthesis would result in a significantly more complicated synthesis task. However, it would be interesting to see if AProVE can find a direct termination proof for the Halide simplifier and handwritten variable solver TRSs and if those direct termination proofs are human-interpretable. If AProVE can find reduction orders to fit, it would also be interesting to see what rulesets could be synthesized using them as an input to our synthesis pipeline.

Chapter 4

AUGMENTING A TERM REWRITING SYSTEM THROUGH SYNTHESIS

Although a Halide-expression 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 rewrite them further. In this section, we describe a workflow for automatically augmenting the Halide simplifier TRS with new rules. We evaluate the usefulness of our process by comparing synthesized rules to those written by human programmers; describing some uses of the process by Halide developers; and carrying out a large-scale experiment in synthesizing rules for the simplifier in bulk, which resulted in new rules being merged into the Halide compiler.

4.1 Synthesizing rewrite 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 4.1. At a high level, we begin with an expression we would like to be able to further simplify, and use it to choose patterns to act as candidate LHSs for new rules. We then attempt to synthesize RHSs that match those candidates LHSs 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.

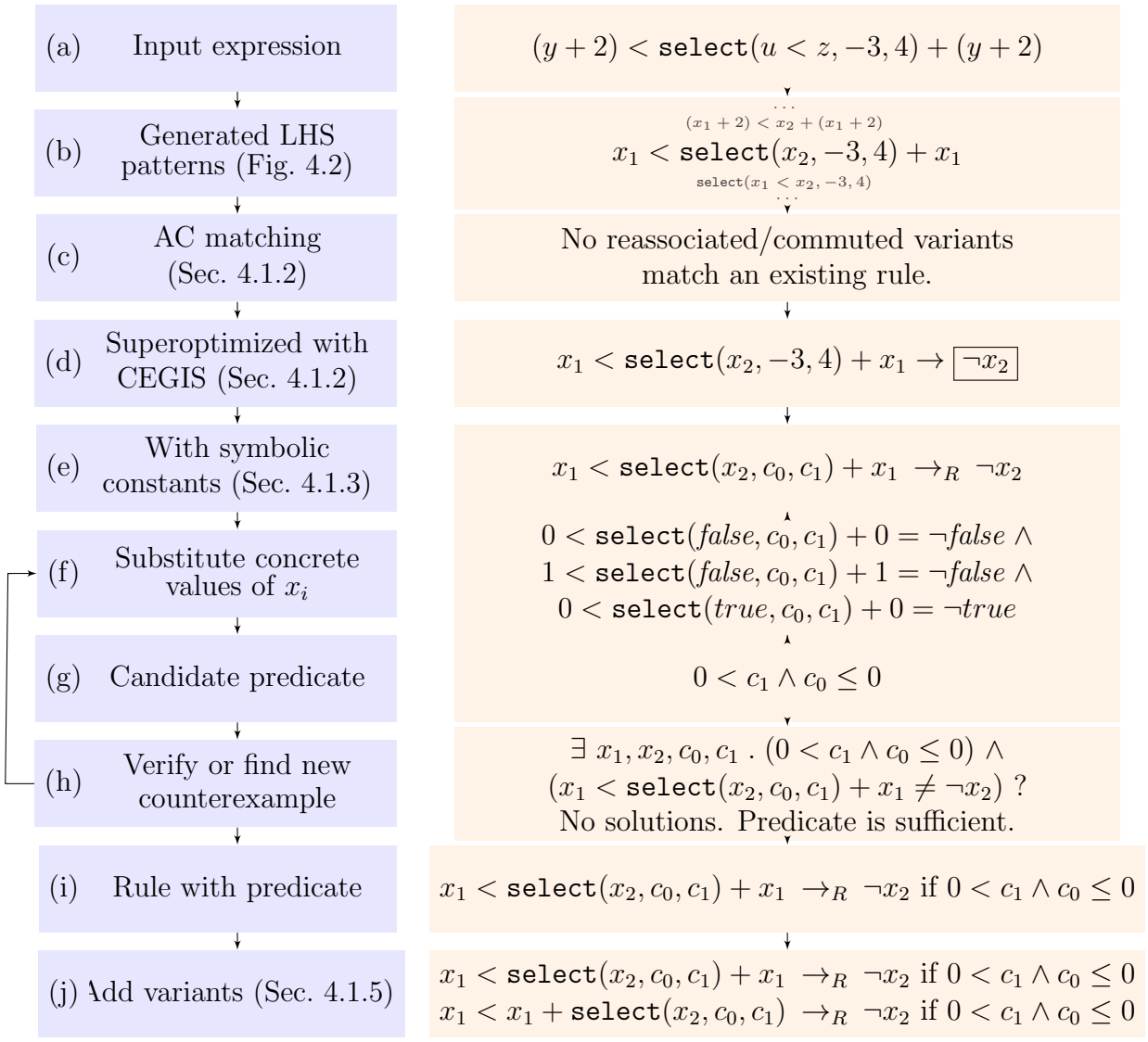


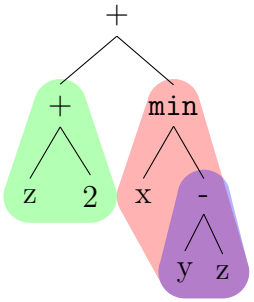
Figure 4.1: Overall flow of the synthesis pipeline (in blue) with worked example (in orange). (a) We harvest expressions from real compilations on which the TRS could make no further progress. (b) We enumerate all subtrees of these to generate left-hand sides that would match each expression. Our example 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.

4.1.1 Generating LHS Patterns

Useful 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 simplifier 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 2.3.3). 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 simplifier 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.

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 4.2 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



$(z + 2) + \texttt{min}(x, y - z)$ $\texttt{min}(x, y - z)$ $z + 2$ $y - z$

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

$v_1 + \texttt{min}(x, v_3)$

$(z + 2) + \texttt{min}(x, v_3)$

$v_1 + v_2$

Figure 4.2: 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.

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.

4.1.2 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 simplifier reduction order, such that $LHS > RHS$. 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.

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 can effectively performed through adding additional rules. It is possible that our candidate LHS, which currently cannot be rewritten

by the simplifier rules, could be matched by an existing rule 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.

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.

Similar to prior work in superoptimization [Sasnauskas et al., 2017b, Phothilimthana et al., 2016a], 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.

Table 4.1: 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	$1 <= c_1 \wedge -1 <= (-1/c_0) * c_0 + c_1$

(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.

4.1.3 Generalizing Constants and Finding Predicate Guards

If either AC-matching search (section 4.1.2) or CEGIS-based synthesis (section 4.1.2) 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:

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

4.1.4 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.

4.1.5 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.

4.2 Evaluation of Simplifier TRS Synthesis

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 4.2.1)

- **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 4.2.2)

- **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 4.2.3)

- **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 4.2.1)

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.

4.2.1 Comparing the Synthesizer to Human-Authored Rules

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, labeled here as $\mathbb{A}$-$\mathbb{E}$. These can be found in their original form as patches on the Halide project website [1]. 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.

---

[1] $\mathbb{A}$: `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`

In case $\mathbb{B}$ the synthesizer matched the human but also crafted 8 commuted variants of the human rules, making them more widely applicable.

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.

What Fraction of the Halide Rules Could Have Been Synthesized?

If the simplifier ruleset had not yet been written by hand, would it have been possible to synthesize it automatically? 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 found this to be promising; efforts to synthesize a term rewriting system entirely from scratch are described in chapter 5.

4.2.2 Practical Uses of the Synthesizer and Verifier

What is the Best Way to Use Synthesis and Verification in Development?

We have found that human experts can 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, labeled here as $\mathbb{F}$ through $\mathbb{I}$. The diffs are available as on the Halide project website[2].

---

[2] $\mathbb{F}$: `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`

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:

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

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:

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

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

---

$\mathbb{I}$: `https://github.com/halide/Halide/pull/4850`

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.

4.2.3 Using the Synthesizer to Prevent Future Issues

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 4.1.

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 4.1.2) 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 4.3 for the full distribution.

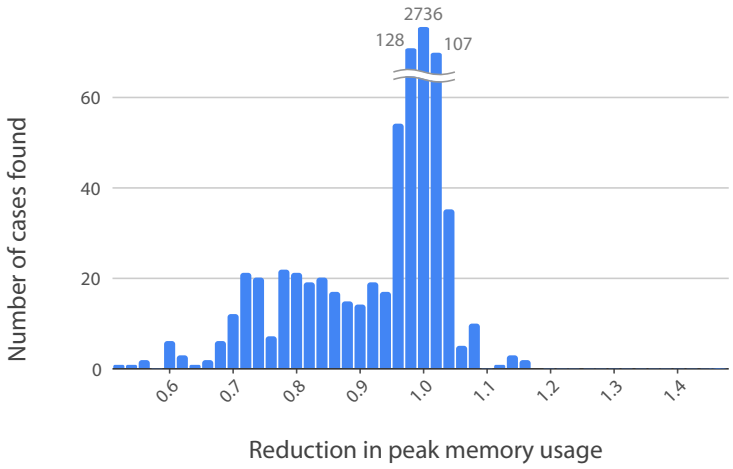


Figure 4.3: Reduction in runtime peak memory usage of 3072 pieces of compiled code when 4127 synthesized rules are added to the TRS. The x-axis shows $\frac{after}{before}$, so below 1.0 means the synthesized rules reduced memory consumption. In 197 cases, peak memory usage drops by more than 10%.

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 the variable solver, which had never been verified (and was not yet implemented as a formal TRS). 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 discuss efforts to replace the current implementation of the variable solver with a correct-by-construction TRS in chapter 5.

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 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 the synthesized rules does not come at any significant cost we could measure to users of Halide, Halide developers were reluctant to add the full 4,000 rule set for two reasons. One is that adding 4,000 lines of code to the simplifier caused a significant slowdown in the compilation time of the Halide compiler itself, which was burdensome for compiler developers. The other was that while the thousand handwritten rules were difficult for programmers to reason about and maintain, adding 4,000 more rules made it almost completely unreadable. In the summer of 2021, Evan Lee of the University of Waterloo undertook a Google Summer of Code project to analyze which rules were crucial to the performance gains observed above. He found that the AC variants of 11 rules, or 31 rules overall, were sufficient to realize the performance gains on the standard application suite benchmarks. Those rules were merged into the Halide codebase in August 2021[3].

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. Halide developers plan 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.

---

[3] `https://github.com/halide/Halide/pull/6174`

4.3 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.

Chapter 5

SPECIFYING AND SYNTHESIZING A NEW TERM REWRITING SYSTEM

In chapter 4, we devised a reduction order to prove that the simplifier ruleset was guaranteed to terminate, and then made use of that order in synthesizing new rules for that ruleset. In a way, we could consider this an extended instance of programming by demonstration—we received around a thousand example programs, manually derived a specification that described all of those examples, and then used it to synthesize new programs. The natural question to ask is whether a user might simply write the specification and synthesize the desired TRS without seeding it with any handwritten rules at all.

The Halide simplifier was a fairly mature system, in production for over a year and consisting of almost a thousand rules. The Halide variable solver, by contrast, is a much smaller system, currently implemented as general-purpose C++ code rather than a formal term rewriting system, and consisting of the equivalent of less than a hundred rules. The simplifier's use cases were quite varied and complex, and the existing ruleset required a complex composite reduction order to cover their full purpose. The variable solver is much smaller and more focused and can be captured by much simpler reduction orders. With the variable solver, therefore, we have the opportunity to experiment with various reduction orders to find the one that fulfills the system's purpose best, and to synthesize a TRS entirely from scratch, rather than augment an existing system.

In this chapter, we argue that the reduction order is a useful and powerful way of writing specifications for term rewriting systems. We demonstrate this using the Halide variable solver term rewriting system as a case study. First we will discuss the rationale behind using reduction orders as specifications and the design of our synthesis pipeline. We define

some possible reduction orders that represent strategies or gradients along which to rewrite expressions to bring them closer to the desired solved form. We then evaluate these orders by synthesizing entirely new term rewriting systems using the orders as specifications and evaluating their performance on a suite of benchmarks.

5.1 Rationale for synthesis pipeline

Here we lay out the rationale for our proposed synthesis pipeline. In particular, we will discuss three key assumptions: that a reduction order can be an effective heuristic for guiding term rewrites to a desired form; that a term rewriting system can be synthesized incrementally rule by rule; and that choosing candidate left-hand sides gathered from realistic inputs is an efficient means of finding rules that will have high impact on the performance of the term rewriting system.

When we specify a term rewriting system in this work, we say that it must have three properties:

- it must be semantics-preserving
- it must terminate on all inputs
- it must rewrite terms into a form that has some desired property

The first two properties are required, but the third is not achievable in absolute terms. As our language is undecidable, we can't create a TRS that is able to rewrite *all* terms into the desired form. Our incremental synthesis pipeline is thus designed to grow the ruleset rule by rule, such that with each successful iteration of the synthesis algorithm, the ruleset is able to move more terms further toward the desired form.

In chapter 3, we showed that the Halide simplifier was semantics-preserving by verifying that for each rule $(l \ \rightarrow_R \ r) \in R$, $l =_e r$, where the equality relation $=_e$ was checked via an SMT solver. We showed that the TRS was terminating by choosing an order $>$ over

terms and showing that every rule in R rewrote terms to be strictly less in that order. When we synthesized new rules to add to the simplifier ruleset, we did not explicitly define the third criterion, but allowed the termination order fitted to the existing ruleset to guide the synthesis pipeline.

In this section, we will define more precisely the goal or intent behind a term rewriting system. We argue that an effective means of specifying this intent for the purpose of synthesis is to use a reduction order to capture the notion of rewriting a term to be closer to some goal state.

First, we define the goal property that describes the underlying intention of the term rewriting system. We write the goal property as a predicate function ϕ such that $\phi(s)$ is true if s is in the goal state and false otherwise. When a term s satisfies ϕ, we say that s is in *goal form*. We denote a language of terms, $T(\Sigma, V)$, and a semantic equivalence relation over those terms $=_e$ (as opposed to a syntactic equivalence relation, which we will write $=$).

We can thus write the domain of the term rewriting system as a language $\mathcal{L} \subseteq T(\Sigma, V)$, defined as

$$\mathcal{L} = \{s | s \in T(\Sigma, V) \wedge (\exists t \in T(\Sigma, V) \; . \; s =_e t \wedge \phi(t))\}$$

We want our term rewriting system to put every term in $\mathcal{L}$ into a form that satisfies our goal function ϕ. When we specify how a term rewriting system rewrites terms into a goal state, we refer only to terms in $\mathcal{L}$; when the TRS rewrites a term that in is $T(\Sigma, V)$ but not in $\mathcal{L}$, it need only obey the semantics-preserving and termination properties.[1]

To illustrate these definitions, we will develop formal specifications for two example term rewriting systems that represent different aspects of the Halide simplifier TRS. As discussed in section 2.3.3, the simplifier is used in many places in the Halide compiler under a number of difference circumstances, perhaps explaining the extremely complex reduction order we devised to fit its ruleset. Here, we will isolate two of the simplifier's important function:

[1]We might prefer that the term rewriting system act on terms not in $\mathcal{L}$ as little as possible, but this is purely for performance reasons and can be set aside for now.

showing that a term must always evaluate to true, and showing that a term is strictly increasing or strictly decreasing.

As in the rest of this work, we assume that $T(\Sigma, V)$ is the Halide expression grammar. First, we consider a prover TRS, which tries to determine if an expression is equivalent to the booleans values true or false, or if its truth value is unknown. We state that the prover has a domain language $\mathcal{L}_P$, which contains all terms in the Halide expression language that are equivalent ($=_e$) to `true` or `false`, and a goal state function ϕ_P that returns true if its input syntactically equivalent ($=$) to `true` or `false` and false otherwise. This goal function encodes the purpose of the TRS, which is to transform all terms that are semantically equivalent to true or false to the boolean constants `true` or `false`. (Again, the goal function says nothing about terms in the Halide language, e.g. $x < y$, that are not provably true or provably false.)

$$\phi_P(s) := s = \texttt{true} \vee s = \texttt{false}$$

$$\mathcal{L}_P = \{s | s \in T(\Sigma, V) \wedge (s =_e \texttt{true} \vee s =_e \texttt{false})\}$$

The second example is a TRS that seeks to show that a given expression is monotonic. The goal function for this TRS is a bit more subtle. The Halide compiler's means of querying the monotonicity of an expression is sound but not complete: it walks the expression to see if it is in a syntactic form that can be recognized as monotonically increasing, monotonically decreasing, or constant. If it cannot recognize the expression as any of these three, it returns unknown. The job of the monotonic rewriter TRS is to put expressions into the syntactic form the query function recognizes wherever possible. We say then that the domain language of the monotonic rewriter $\mathcal{L}_M$ is all terms s in $T(\Sigma, V)$ where there exists some term t such that $s =_e t$ and t is in a syntactic form that can be recognized as monotonic or constant by the function `is_monotonic` in the Halide codebase. The goal state function ϕ_M returns false when `is_monotonic` returns unknown on an input term, and true otherwise. (Again, ϕ_M may return false for a term that is not in monotonic form because it is not in $\mathcal{L}_M$, i.e. no

semantically equivalent monotonic form exists, but we can disregard this case.)

With these definitions in hand, we are now ready to formally specify the three properties we require of a term rewriting system (semantics-preserving, terminating, and rewrites terms into a desired form). We write $R(s)$ for the output of a term rewriting system R on the input term s. Assume we use the means of assuring semantics preservation and termination as used above. Then, for an equivalence relation $=_e$ and a goal function ϕ, we write that a specification for an ideal TRS R thus:

$$\forall s \in \mathcal{L} \,.\, \phi(R(s)) \wedge \forall t \in T(\Sigma, V) \,.\, t =_e R(t) \wedge R(t) \text{ terminates}$$

Of course, since our theory is undecidable, the $\forall s \in \mathcal{L} \,.\, \phi(R(s))$ portion of our specification is almost certainly unrealizable in the general case. Instead, we would like to encode a means of making progress towards this goal, rather than ruling out all TRSs that do not achieve it. We can think of this as a strategy for rewriting terms to be closer to the desired goal state. For example, imagine we started writing rules for the prover, and started with this pair:

$$x = x \;\rightarrow_R \texttt{true}$$
$$x + (y - y) = x \;\rightarrow_R \texttt{true}$$

We might note that the second rule is canceling like terms, and add more rules that cancel like terms without necessarily putting the rewritten term into the goal state, such as the rule $x + (y - y) \;\rightarrow_R\; x$. If we wanted to formally encode this strategy, we could do so using an order over terms, such that terms that satisfy the goal condition ϕ would be least elements in that order. In the case of the prover, the order could be to compare terms by their length; since the boolean constant `true` is of the shortest possible length in the Halide expression language, it is a least element of this order. Let us call such an order a *goal order* ($>_\phi$) and add it as a condition to our specification:

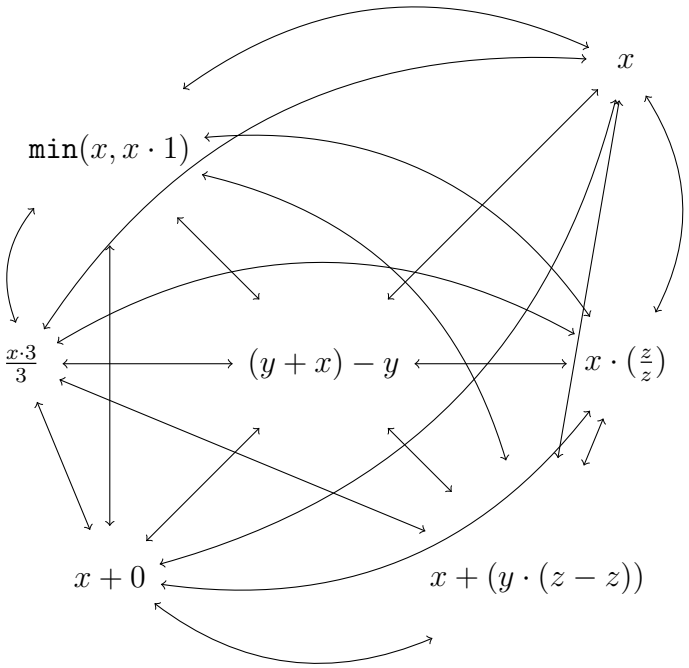


Figure 5.1: A set of terms that are semantically equivalent. In a TRS with no reduction order requirement, a rewrite from any term to any term would be legal. The terms and all legal rewrites between them thus form a fully-connected, bidirectional graph.

$$\forall s \in \mathcal{L} \,.\, \phi(R(s)) \wedge s \geq_\phi R(s) \wedge \forall t \in T(\Sigma, V) \,.\, t =_e R(t) \wedge R(t) \text{ terminates}$$

Imagine terms as nodes in a graph and rewrite rules as transitions between those nodes. In figure 5.1, we see a set of terms which are semantically equivalent; in the absence of an ordering over terms, any rewrite from any term to any other would be a valid rule. In figure 5.1, we fix an ordering in which shorter terms are less than longer terms; now only rewrites that obey that ordering are available as possible transitions. We wish to fix an order over the terms in $\mathcal{L}$; the goal order $>_\phi$ could be considered an order traversal from terms not in the goal state to terms that are.

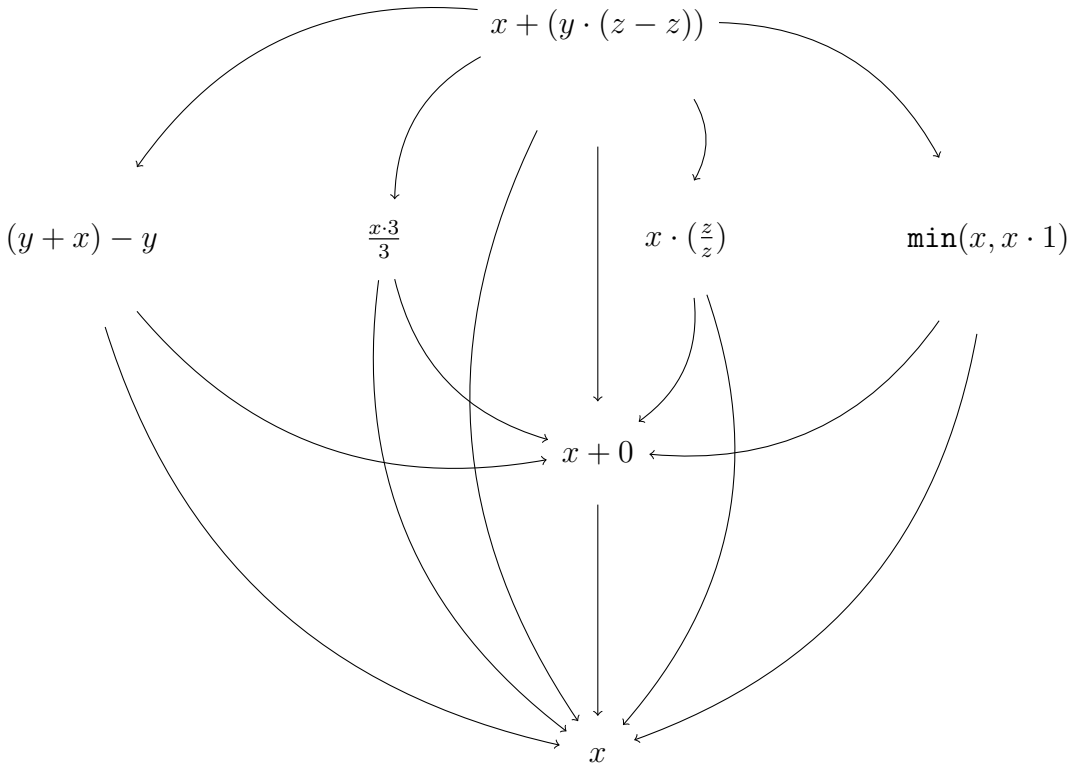


Figure 5.2: A set of terms that are semantically equivalent, ordered by size from top to bottom. In a TRS using a reduction order that sorts terms by size, a rewrite is only legal if its LHS is longer than its RHS. The unidirectional edges in this graph show all possible legal rewrites that obey this order.

Now, assume we have been given some solution to our specification in the form of a term rewriting system R. How can we prove that it fulfills our criteria? In chapter 3, we proved that a term rewriting system was semantics-preserving ($\forall t \in T(\Sigma, V)\ .\ t =_e R(t)$) and terminating ($\forall t \in T(\Sigma, V)\ .\ R(t)$ terminates) by showing that those properties held over every rule in the ruleset. We can use a similar strategy here: if the goal order $>_\phi$ holds for every rule in R, then it must hold for all of R as a whole.

Assumption 1. *We can relax our requirement that the ruleset R rewrite every term in $\mathcal{L}$ to the goal state to the requirement that every rule in R rewrites terms to be closer to the goal state, as described by a goal order $>_\phi$.*

In order for the goal order to hold over every term that may match and be rewritten by a rule, it must be a valid reduction order: well-founded, Σ-operation compatible, and closed under substitution (see [Baader and Nipkow, 1999]). Usefully, this means we do not need a separate termination order; since a goal order requires that each rewrite moves a term monotonically lower in the order, it also provides a termination guarantee.

We can now restate our term rewriting system specification like so:

$$\forall s \in \mathcal{L}\ .\ \phi(R(s)) \wedge \forall (l\ \rightarrow_R\ r) \in R\ .\ l =_e r \wedge l >_\phi r$$

Since any term rewriting system we wish to synthesize will be semantics-preserving, the major decision we need to make in specifying a term rewriting system is to pick a goal order:

Assumption 2. *We can effectively formalize the goal of a term rewriting system by choosing a* goal order *over terms such that, for two terms s and t such that $s >_\phi t$ in this order, t is assumed to be closer to the goal state.*

Finding a goal order that encodes progress towards a goal state is a difficult task, and requires human intuition and ingenuity. Our work does not directly aid in this task, which is still up to the human author of the term rewriting system; however, our synthesizer machinery can aid in experimenting with and selecting goal orders.

TRS	Goal function ϕ	Goal order $s > t$ holds if
Prover	$\phi(s)$ if s is `true` or `false`	s is longer than t
Monotonic rewriter	$\phi(s) :=$ `is_monotonic(s)`	s has more % ops than t s has more constants than t

Table 5.1: Two term rewriting systems, goal functions that encode their intent, and goal orders that represent moving towards those goals

As discussed above, the prover could adopt a strategy of making expressions shorter, with the aim of eventually being able to reduce them to the constants `true` or `false`. For the monotonic rewriter, one expression in the goal state is $x \cdot c_0$ where c_0 is a positive constant; the monotonicity checker recognizes this form as monotonically increasing. So, one possible strategy could be to move constants to the right side of an equation wherever possible. We may also want to employ sub-strategies: if the prover TRS contains the rule $x = x \ \rightarrow_R$ `true`, we might choose a strategy of putting expressions into a kind of canonical form wherever possible, so we can rewrite an expression $e_1 = e_2$ into $e' = e'$ and then apply the identity rewrite rule. The simplifier reduction order, which was composed of several orders lexicographically, could be said to employ a series of sub-strategies in this way. See table 5.1 for the prover and monotonic rewriter goal functions and some goal orders that approximate them.

If we accept the assumption that a reduction order can encode the strategy of rewriting a term to be closer to our goal, we now have a specification that should hold for every individual rule we synthesize:

$$\forall (l \ \rightarrow_R \ r) \in R \,.\, l =_e r \wedge l >_\phi r$$

Finally, we need a strategy for choosing which rule to add to growing ruleset at each step. As the space of potential rules is so large, even if term sizes are bounded, simply sampling at random may not be effective. Our synthesis pipeline defines a strategy for growing a ruleset

so that it is able to rewrite more and more of $\mathcal{L}$, in a way that is of practical benefit to the compiler. In prior work, we gathered a corpus of expressions seen by the compiler in realistic circumstances and mined them for general patterns to form candidate LHS terms. We then attempted to synthesize RHSs for each candidate LHS term to form rules. We assert that this is an effective method of creating a ruleset that is effective under realistic workloads.

Assumption 3. *An effective heuristic for choosing candidate left-hand side terms for new rules is to gather expressions seen by the compiler under realistic circumstances and mine them for general patterns.*

Note that when we gathered input expressions in prior work, we did not check to see if those expressions were in the TRS domain $\mathcal{L}$. Any means of checking membership in $\mathcal{L}$ is necessarily incomplete, so any ruleset learned from a checked input list would inherit the incompleteness of the checker. Also, note that we will want to rewrite subterms of expressions that are not themselves in $\mathcal{L}$: in the prover TRS, although all input expressions are boolean-valued, we will want to rewrite subterms of all possible types. For example, given the ruleset $R = \{x = x \ \rightarrow_R \ \texttt{true}, x + x \ \rightarrow_R \ x \cdot 2\}$, we need the second rule that rewrites a numeric-valued expression so that we can rewrite $x + x = x \cdot 2$ to `true`. In section 5.4 we will see that useful rules can be learned from expressions not in $\mathcal{L}$.

Given these three assumptions, we propose the synthesis pipeline in algorithm 1.

5.2 The variable solver reduction orders

The variable solver's goal is to pull out a specified target variable from an input expression as much as possible. Here we refer to *target-matching variables* as variables that can be unified with the target variable or subterms that contain the target variable. Variables that cannot be unified with the target variable or subterms that contain the target variable are called *non-target-matching variables*. All variables that occur in the variable solver ruleset are either target-matching or non-target-matching. Allowing variables that can match either target variable-containing subterms or non-target variable containing subterms complicates

reduction order definitions.

Here we define a reduction order that encodes a strategy for rewriting terms to be closer to the variable solver's desired solved form, which we will call the *gradual order*. This reduction order composes three sub-orders: an order that reduces occurrences of target-matching variables; an order that moves target-matching variables to the left in an expression; and an order that moves target-matching variables higher up in an expression. We give proof sketches to show that these three orders are in fact valid reduction orders. We also define a reduction order, which we will call the *goal form order*, which partitions the term language into two equivalence classes, terms that are in the variable solver's goal form and terms that are not, and states that terms in goal form are ordered less than terms that are not. We show that this, too, is a valid reduction order.

We represent target-matching variables as x^t, y^t, etc., and non-target-matching variables as x^n, y^n, and so on.

5.2.1 Reduce occurrences of target-matching variables

$$s_1 >_t s_2 \iff \forall x^t \in \mathcal{V}ar(s_1) |s_1|_{x^t} \geq |s_2|_{x^t}$$

The proof that this order is a valid reduction order is straightforward. The order is clearly well-formed, since $|s|_{t_i}$ has a minimum value of 0. The order is compatible with Σ-operations; for any

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

$$\sum_{x^t \in \mathcal{V}ar} \sum_{0}^{i-1} |t_k|_{x_k^t} + |s_1|_{x_k^t} + \sum_{i+1}^{n} |t_k|_{x_k^t} > \sum_{x^t \in \mathcal{V}ar} \sum_{0}^{i-1} |t_k|_{x_k^t} + |s_2|_{x_k^t} + \sum_{i+1}^{n} |t_k|_{x_k^t}$$

Since $|s|_{x^t}$ is always non-negative, the implication clearly holds.

Finally, the order is closed under substitutions. A substitution σ could increase the number of target-matching variables in s_2 by replacing some x^t with a subterm containing

multiple target-matching variables, but since all target-matching variables in s_2 must occur an equal or greater number of times in s_1, the order must be preserved. Any substitution for a non-target-matching variable will have no effect on the number of target-matching variables in s_1 or s_2 by definition.

Note that this order does not require that the number of non-target-matching variable occurrences in s_1 be equal or greater to the number of occurrences in s_2; it is perfectly fine to increase the size of the rewritten expression so long as the count of target-matching variables goes down. For example, the rule $t_0 <= (t_0 + n_1) \ \rightarrow_R \ n_1 >= 0$ introduces a new constant on its RHS, but obeys the order as it reduces the number of occurrences of t_0.

5.2.2 Move occurrences of target-matching variables to the left

The intuition behind the order is simple, but we will have to take care in our definition to make sure it is a valid reduction order. We define this order by transforming terms into strings and then comparing the strings lexicographically.

To transform a term into a string (we write this function as `varstr`(s)), we first take the sequence of all occurrences of variables and constants in the term, in the order in which they occur. We truncate the sequence by removing everything after the last occurrence of a target-matching variable. We then replace all target-matching variables with 'a' and all non-target-matching variables or constants with 'b', and compare the resulting string with that of another term in lexicographic order (such that a < b).

$$n_0 + t_0 > t_0 + n_0$$

$$(n_0 t_0) > (t_0 n_0)$$

$$(n_0 t_0) > (t_0)$$

$$\text{ab} > \text{a}$$

This order is clearly compatible with Σ-operations; if $\texttt{varstr}(s_1) > \texttt{varstr}(s_2)$, then the order will still hold if identical prefixes and suffixes are added to both terms.

$$\begin{aligned} s_1 > s_2 &\implies f(t_1, s_1, t_n) > f(t_1, s_2, t_n) \\ \texttt{varstr}(s_1) > \texttt{varstr}(s_2) &\implies \texttt{varstr}(t_1) + \texttt{varstr}(s_1) + \texttt{varstr}(t_n) > \\ &\qquad \texttt{varstr}(t_1) + \texttt{varstr}(s_2) + \texttt{varstr}(t_n) \end{aligned}$$

However, we need to add an additional condition to make the order closed under substitution. Consider a rule

$$(n_0 + n_1) + (t_0 + n_2) \ \to_R \ (n_2 + t_0) + (n_0 + n_1)$$

The rule appears to move the target-matching variable to the left and thus seems to be in conformance with the order. However, given the substitution $\sigma = \{n_0 \mapsto y^n, n_1 \mapsto z^n, t_0 \mapsto x^t, n_2 \mapsto ((u^n + v^n) + w^n)\}$, we obtain the transformation

$$(y^n + z^n) + (x^t + ((u^n + v^n) + w^n)) \ \to_R \ (((u^n + v^n) + w^n) + x^t) + (y^n + z^n)$$

which actually shifts the target-matching variable to the right. The problem occurs because the non-target-matching variables have been reordered from the LHS term to the RHS. A similar issue arises if we reorder the target-matching variables:

$$\begin{gathered} (t_0 + n_0) + t_1 \ \to_R (t_1 + t_0) + n_0 \\ \sigma = \{t_0 \mapsto x^t, n_0 \mapsto y^n, t_1 \mapsto ((z^n + w^n) + x^t)\} \\ (x^t + y^n) + ((z^n + w^n) + x^t) \ \to_R (((z^n + w^n) + x^t) + x^t) + y^n \end{gathered}$$

To disallow this ordering, we add the condition that a rewrite rule that follows this order cannot permute the sequence of target-matching variables or the sequence of non-target-matching variables; it can only change how the two sequences are interleaved. More precisely,

if we take the sequence of target-matching variable instances from s_1 and s_2, we should be able to fix a mapping from each variable instance in s_2 to an instance of that same variable in s_1 such that, after deleting all variable instances in s_1 that are not mapped, the two sequences are identical. We must be able to do the same with the non-target-matching variables in both terms, after omitting all non-target-matching variables that occur after the final instance of a target-matching variable in their respective terms. With these condition, there is no substitution that can introduce a non-target-matching variable in the s_2 term without introducing at the same or earlier position in s_1, and thus the order is closed under substitution.

5.2.3 Move occurrences of target-matching variables up

Much like the previous order, this order is an intuitive step in isolating the target variable to the left of a top-level binary operator. To define this order formally, we first define a function `bfs_hash` that takes a term and a variable, and returns a hash whose keys are the depths of the AST representation of that term and whose values are the number of instances of that variable that occur at those depths. We say that $\texttt{bfs\_hash}(s_1, x) < \texttt{bfs\_hash}(s_2, x)$ if $\texttt{bfs\_hash}(s_1, x)[0] < \texttt{bfs\_hash}(s_2, x)[0]$, or if $\texttt{bfs\_hash}(s_1, x)[0] = bfs_hash(s_2, x)[0] \wedge \texttt{bfs\_hash}(s_1, x)[1] < \texttt{bfs\_hash}(s_2, x)[1]$, and so on.

We then say that $s_1 >_\uparrow s_2$ if, for every target-matching variable x_t that occurs in s_1, there are an equal or greater number of occurrences of that variable in s_1 than in s_2, and $\texttt{bfs\_hash}(s_1, x) \leq bfs_hash(s_2, x)$, and if there exists at least one target-matching variable whose `bfs_hash` value is strictly greater in s_2 than in s_1.

$$\begin{aligned} s_1 >_\uparrow s_2 \iff & \forall x^t \in \mathcal{V}ar(s_1) \,.\, (|s_1|_{x^t} \geq |s_2|_{x^t} \wedge \\ & \texttt{bfs\_hash}(s_1, x_t) \leq \texttt{bfs\_hash}(s_2, x_t)) \\ & \wedge \exists x^t \in \mathcal{V}ar(s_2) \,.\, \texttt{bfs\_hash}(s_1, x_t) < \texttt{bfs\_hash}(s_2, x_t)) \end{aligned}$$

For example, consider the rule

$$(t_0 - n_0) < n_1 \ \rightarrow_R \ t_0 < (n_0 + n_1)$$

Both the LHS and RHS have the same number of target-matching variables. `bfs_hash`$((t_0 - n_0) < n_1, t_0) = \{0 : 0, 1 : 0, 2 : 1\}$, while `bfs_hash`$(t_0 < (n_0 + n_1), t_0) = \{0 : 0, 1 : 1, 2 : 0\}$. As `bfs_hash`$((t_0 - n_0) < n_1, t_0)[0] =$ `bfs_hash`$(t_0 < (n_0 + n_1), t_0)[0]$ and `bfs_hash`$((t_0 - n_0) < n_1, t_0)[1] <$ `bfs_hash`$(t_0 < (n_0 + n_1, t_0))[1]$, the order holds over this rule.

This order has a least element with regard to a fixed number of target-matching variables: one in which all target-matching variables occur in the highest level of the AST. As the number of target-matching variables in any RHS must be less than or equal to the number of target-matching variables in the LHS, it is not possible to construct an infinitely descending chain in this order, so the order is well-founded.

The order is Σ-compatible; for each target-matching variable, `bfs_hash`$(f(t_1, \ldots, t_{i-1}, s_1, t_{i-1}, \ldots, t_1)$ is the sum of the `bfs_hash` of its subterms (with the depths incremented by one). As `bfs_hash`$(f(t_1, \ldots, t_{i-1}, s_2, t_{i-1}, \ldots, t_1)$ is the sum of precisely the same subterms but with `bfs_hash`(s_1) replaced by `bfs_hash`(s_2), the requirement holds.

The order is also closed under substitutions. As all variables are necessarily terminals in a given AST, substituting a subterm for a given variable cannot "push down" any part of the term that is not directly part of the substitution. Any substitution for a non-target-matching variable will have no effect on the `bfs_hash` output. For every instance of a target-matching variable in a RHS term, we can fix a corresponding instance of that same variable in the LHS term at an equal or lower depth. Any change to the `bfs_hash` counts therefore cannot possibly result in a violation of the order.

5.2.4 Put rewritten expression into fully solved form

The orders described above describe a strategy for manipulating terms towards the goal state of putting them in fully solved form. Is it possible to encode the goal state directly as an order? Recall that we defined 'solved form' as:

1. $|t|_{x^t} = 0$ (the term t does not contain the target variable x^t).

2. $t = x^t$ (the term t is precisely the target variable x^t).

3. $t = x^t \odot t'$, where $\odot$ is any binary function in Σ, and $|t'|_{x^t} = 0$. Terms in this form are only considered 'solved' if no term u in either of the above two forms exist such that $t =_e u$.

Reduction orders must be checkable syntactically, so we relax the requirement that the second form is only considered solved if no equivalent term in the first form exists and so on. Then, we can formalize these forms into orders such that terms that are solved are ordered less than terms that are not solved.

1. $s_1 >_{f1} s_2 \iff \exists x^t \in \mathcal{V}ar(s_1) \wedge \neg\exists x^t \in \mathcal{V}ar(s_2)$

2. $s_1 >_{f2} s_2 \iff \exists x^t \in \mathcal{V}ar(s_1) \wedge s_1 \neq x^t \wedge s_2 = x^t$

3. $s_1 >_{f3} s_2 \iff s_1$ cannot match $f(t_0, n_0)$ for any $f \in \Sigma$, and s_1 can match $f(t_0, n_0)$ for some $f \in \Sigma$

Are these valid reduction orders? All three are well-founded, in that they partition the set of all expressions into two equivalence classes (expressions in solved form and expressions not in solved form) and orders all members of the first class as less than the second. However, none of the three are Σ-compatible; if an expression in solved form is added as a subterm in a larger expression, the new expression is necessarily not in solved form. The second

and third orders are also not closed under substitution; in the second and third type of solved form expression, if the single x_t variable is substituted for a longer term containing a target-matching variable, the resulting expression will not be in solved form. (The first order is closed under substitution, since we can substitute any non-target-matching variable with any term not containing a target-matching variable and remain in solved form.)

This demonstrates that any specification for a ruleset will represent a strategy for rewriting expressions towards a goal state, and not necessarily a means of putting expressions into a goal state in all circumstances. Consider the following rule, whose RHS is in solved form

$$(t_0 + n_1) == n_0 \ \rightarrow_R \ t_0 == (n_0 - n_1)$$

which can be used to rewrite the following three expressions

$$\begin{aligned} x_t + y_n = z_n \ &\rightarrow_R \ x_t = z_n - y_n \\ (z_n < w_n) \wedge x_t + y_n = z_n \ &\rightarrow_R \ (z_n < w_n) \wedge x_t = z_n - y_n \\ (x_t * 3) + y_n = z_n \ &\rightarrow_R \ (x_t * 3) = z_n - y_n \end{aligned}$$

In the first instance, the rewritten expression is in solved form, but the second and third instances are not.

Although the 'solved form' orders are not valid reduction orders, it seems clear that rulesets following these orders must terminate, and in fact this is true. For any rule that obeys one of the above orders, its RHS will be the least element of one of the reduction orders we outlined above. Thus the rule will conform to a reduction order, and any such ruleset is guaranteed to terminate.

$$\begin{aligned} s_1 >_{f1} s_2 &\iff s_1 >_t s_2 \\ s_1 >_{f2} s_2 &\iff s_1 >_{\leftarrow} s_2 \\ s_1 >_{f3} s_2 &\iff s_1 >_{\uparrow} s_2 \end{aligned}$$

We could thus think of rules that obeyed the 'solved form' orders as rules that take the largest possible step in the direction of the goal state. If we limited our ruleset to such rules, would the result be more powerful than a ruleset that allowed all rules that obey the reduction orders above? We will investigate this in our evaluation.

5.3 The synthesis algorithm

Our synthesis algorithm takes as inputs

- A goal function ϕ, which returns true when applied to a term in the goal state and false otherwise.
- A semantic equivalence relation $=_e$ over terms. A concrete implementation of this relation is required to be sound, but it will almost certainly be incomplete (unless working in a decidable theory).
- A reduction order $>$, which encodes a desired strategy of moving terms in the direction of the goal state.
- A set of training expressions E, which will be used to choose candidate left-hand sides for new rules. These expressions should be likely inputs to the TRS when it is used for its desired task.

An idealized version of the algorithm is presented in Algorithm 1. We initialize with an empty ruleset R. We choose an expression from our training set E, and from that expression choose a pattern that could match with it to serve as a candidate left-hand side. We query the synthesizer for a right-hand side that is semantically equivalent to the LHS and is ordered less than the LHS in our reduction order. If we cannot find such a RHS, we choose another candidate LHS and continue. When we do find a RHS that satisfies our specification, we have learned a new rule. We add this new rule to the end of the ruleset R. We then use the updated ruleset to rewrite every training expression in E, discarding any expressions that can

satisfy ϕ after being rewritten (since the newly updated TRS is capable of rewriting those expressions to the goal state, we do not need to learn any more rules from these expressions). The remaining expressions in the updated set E have been rewritten as far in the direction of the goal state as the new ruleset is capable of. If the test set condition is not yet met, we want to learn rules that will bring these expressions even further toward the goal state, so we choose a candidate LHS on our next iteration from this newly normalized set.

Algorithm 1: Idealized procedure for synthesizing a term rewriting system

Input: goal function ϕ, semantic equivalence relation $=_e$, reduction order $>$, set of training expressions E

Result: a term rewriting system R

while $E \neq \varnothing$ **do**

  choose $e \in E$, $l \in$ `lhs_patterns(e)`;

  query synthesizer for r such that $l =_e r \wedge l > r$;

  if r *is found* **then**

    $R \leftarrow R \bigcup \{(l \rightarrow_R r)\}$;

    $E \leftarrow \{R(e) | e \in E \wedge \neg\phi(R(e))\}$;

  end

end

We implement this algorithm using Rosette [Torlak and Bodik, 2014], a solved-aided programming language extending Racket for performing verification and synthesis tasks. We chose Rosette over the C++ implementation of the simplifier synthesis pipeline as it allowed for quick experimentation in development of our process. Rosette performs symbolic evaluation over functions taking symbolic values as inputs, easily lifting Racket code to SMT2-language formulas that can be passed to solvers. We wrote an interpreter for the Halide expression language in Rosette to allow for this symbolic evaluation of Halide expressions. In order to handle the disparity between the Halide/C++ semantics and Racket semantics (particularly the different truth values of non-boolean types), we had to add some lightweight typechecking to our interpreter to ensure that integer-typed values were not coerced to boolean values. As

our synthesis pipeline requires renormalization of inputs every time a rewrite rule is learned, we also implemented a term rewriting algorithm with target-variable-aware matching in Rosette (making use of the ML code in Baader and Nipkow [1999]), as well as checkers for our two reduction orders.

Our synthesis pipeline is implemented as a recursive function that takes an inputs a training set of expressions and an initially-empty TRS. We take our first training expression and iterate over its subterms in the order that the Halide rewriting algorithm would attempt to match them, in a bottom-up, depth-first search order. At each subterm, we find all patterns that could match it. On the rationale that more general rules are more useful, we sort the resulting patterns by size and try the smallest first.

Now we have a candidate left-hand side expression. We attempt to form a rule by synthesizing a matching right-hand side. We do this by issuing a query to synthesize a RHS that is semantically equivalent to the candidate LHS. When we ask Rosette to synthesize an expression, we typically give it a *sketch*: a skeleton program that the solver will complete in a way that satisfies our requirements. To encode the requirement that the synthesized RHS must obey the desired reduction order with regard to the LHS, we can write sketches in such a way that any valid solution to them must satisfy the reduction order. This constrains the search space for valid solutions: the solver will then only look for ways of completing the sketch in such a way that the completed sketch is equivalent to the candidate LHS for all possible values of the variables.

Both our goal form order and gradual order are compositional orders. We encode this by issuing not one synthesis query per candidate LHS, but several, first searching for a solution to the first order in the compositional order, and, if no solution can be found, moving on to search for the next. This further divides the search space, such that each individual query is likely to be either solved or found to be unsatisfiable quickly. As detailed below, each component order is further broken out into several queries. Sketches used for partitioning and ordering the search space are known as metasketches (see Bornholt et al. [2016]).

If any of those queries finds a solution, we have learned a new rule. We add it to our

TRS, and then use the updated TRS to rewrite our current training expression. We then begin transverse the rewritten expression from the beginning just as before. If none of our queries was able to find a satisfying RHS, we move on to the next in our list of candidate LHS patterns on the current subterm, and if none of those patterns yield a valid rule, we move on to the next subterm in our traversal. If our traversal continues all the way to the top of the expression and all synthesis queries have failed, we conclude that we cannot learn any new rules from this expression, discard it, and continue with the next expression in our training set. When the training set is empty, the function returns the learned TRS.

5.3.1 The fully solved order as sketches

The fully solved order is a composition of three orders, encoded as the sketches detailed below.

No target-matching variables sketch

The only constraint on a sketch of this kind is that the synthesized solution does not contain any of the target variables. The sketch is thus a valid Halide expression in which the solver is constrained to use only non-target-matching variables. For efficiency, we further break down the search space by first querying for an expression that contains only one Halide operator, then only two, incrementing until we reach the number of operators present in the candidate LHS. At that point, we conclude that no RHS in this form exists and continue to the next type of sketch.

Target-matching variable alone sketch

This type of sketch is very simple: for each unique target-matching variable present in the candidate LHS, we query to see if that variable is equivalent to the LHS. Note that while the no target variable order is ahead of this order in the fully solved order composition, if a solution in this form exists, then there is no solution in the form of a term that does not

contain any target variables, and vice versa. After checking each target-matching variable in the LHS, if none are found to be equivalent to the LHS, we move on to the third type of sketch.

Target variable-operator-expression sketch

Here, we take each unique target-matching variable in the candidate LHS in turn. We ask the solver to complete a sketch where it can choose any binary operator in the Halide grammar where its first operand is the currently-considered target-matching variable and its second operand is an expression that contains no target-matching variables. This second operand is the completion of a sketch just like the 'no target-matching variable' sketch above. Similarly, we check for solutions in which the number of operators in the second operand term is one, then two, up until the number of operators in the candidate LHSs.

5.3.2 The gradual order as sketches

Similarly, the gradual order is encoded as a series of three sketches.

Fewer target-matching variables order sketch

This is the only one of our sketches that is not guaranteed to return a solution that fulfills its reduction order, but is only a heuristic. We simply ask the solver for the smallest expression it can find that is equivalent to the LHS, using any variable present in the LHS pattern. We do this by asking for an expression using first only one operator and then incrementing with each query, as in the sketches above, stopping when the number of operators present in the LHS is reached. Any RHS expression that is smaller than the LHS pattern in operator count is guaranteed to use fewer instances of variables than the LHS does. If a solution that uses fewer instances of target-matching variables exists, the solver is likely to discover it. As it is possible for the solver to find a solution that uses the same number of target-matching variables, however, any solutions are checked to conform to the reduction order. If the solution

does not satisfy the reduction order, we assume no satisfying solution exists, and move on.

Move target-matching variables left or up order sketch

Here, we cannot allow the solver to find any expression of a predetermined size; we also need to constrain the *position* of the target-matching variables in the RHS. Our insight here is that if we knew the positions of the target-matching variables, we could check that those positions would be ordered less in the reduction order than the candidate LHS, even without knowing which operators were present in the solution or even exactly which target-matching variables were in those positions. We precalculate all possible AST shapes for expressions up to a certain size (3 operators). Then, for a given candidate LHS, we check all possible assignments of target-matching and non-target-matching status to the terminals of those ASTs. For an assignment that satisfies the reduction order, we then ask the solver to choose operators to fill in the AST nodes and to choose variables to fill in the terminals of the appropriate target-matching or non-target-matching status. Multiple variable-status assignments may satisfy the reduction order, so we find all possible satisfying sketches and then sort them using the reduction order itself, querying the solver for solutions for the sketch that is least in the order and continuing up. Since all RHSs that move a target variable left are ordered less than RHSs that move a target variable up, we can check for both types of sketches in the same step.

5.4 Evaluation of synthesized TRSs

We evaluate our synthesis process by synthesizing two term rewriting systems, one using the goal form order as a specification and one using the gradual order. We gathered a set of expressions by instrumenting calls to the existing Halide variable solver and running a suite of applications with randomly generated schedules, as we did in section 4.2.3. As our current synthesis pipeline does not support synthesis of rules guarded by predicates, there was no need to consider constant values, so we replaced all constants in the gathered expressions with fresh variables. After removing all duplicate expressions modulo alpha renaming, we had

a set of 4218 expressions to serve as benchmarks. Note that not all of these benchmarks can be put into solved form, and since we lack a decision procedure for this undecidable problem, we do not know how many of the benchmarks can be solved.

We chose 33 of those expressions as a training set. We chose these expressions randomly, excluding expressions that were so large as to cause performance issues during synthesis. Recall that the Halide rewriting algorithm attempts to rewrite each subterm of an expression, and that each subterm can be matched by many potential LHS patterns. Thus, the set of 33 expressions yielded about 500 candidate LHS expressions (the exact number depends on how successful the learned TRS is at rewriting expressions—if a rule is learned that can transform an input expression, the newly rewritten expression may yield more candidate patterns).

We ran two synthesis experiments, taking as a specification the goal form order and the gradual order respectively. The goal form order task synthesized 21 rules, while the gradual order task synthesized 32. All of the rules in the goal form TRS were also present in the gradual TRS. Both the goal form and gradual TRS are able to solve 11 of the 33 training set expressions.

Of the 4218 benchmarks (including the 33 training set expressions), the goal form TRS was able to put 949 of them into solved form, while the gradual TRS was able to solve 966 of them; in other words, the goal form TRS performed fairly well and the gradual TRS performed slightly better. There were 18 expressions that the gradual TRS was able to solve that the goal form TRS was not.

In all 18 cases, both TRSs applied the same sequence of rewrites up to a certain point, when the goal form TRS was unable to continue. The gradual TRS was able to further rewrite the expression with a rule that was not present in the goal form TRS, and from there was able to continuing rewriting the expression into a solved form.

For example, consider this input expression [2]

[2]The expression has been simplified by replacing subterms that have no bearing on the final result with fresh variables. For example, $\frac{y+c_0}{c_1} \cdot c_1$ has been replaced by y.

$$\texttt{max}(((y + ((x + z) - w)) + c_1), (u + ((z + x) - w))) + (w - (x + z))$$

Both the goal form and gradual TRS rewrite it to

$$(x + \texttt{max}((((z - w) + y) + c_1), ((z - w) + u))) + (w - (x + z))$$

at which point no rules in the goal form TRS can be applied. However, the gradual TRS can apply the rule

$$t_0 + (n_1 - t_2) \ \rightarrow_R \ t_0 - (t_2 - n_1)$$

to produce the expression

$$((x + max((((z - w) + y) + c_1), ((z - w) + u))) - ((x + z) - w))$$

The gradual TRS can then apply further rewrites to cancel the target variable and arrive at an expression in solved form:

$$\texttt{max}((((z - w) + y) + c_1), ((z - w) + u)) - (z - w)$$

Interestingly, in 15 of the 18 expressions that the gradual TRS was able to solve when the goal form TRS was not, the rule that unlocked the expression for the gradual TRS was actually one that obeyed the goal form order. Recall that when we synthesize rules from an input expression, we attempt to rewrite it just as we would with a concrete TRS, but rather than rewriting with existing rewrite rules we ask the solver if we can find a new rule that would allow us to make progress. On the input expression

$$(x \cdot c_0) + \Big((y + z) - \Big(\Big(\Big(\frac{(select(c_1 < x, c_2, c_1) + z) + y}{c_3} + (x \cdot c_3)\Big) \cdot c_3\Big)\Big)\Big)$$

Both tasks were able to normalize to the following form using rules they already knew

$$(x \cdot c_0) + \Big((y + z) - \Big(\Big(\Big(\frac{(select((x > c_1), c_2, c_1) + (z + y))}{c_3}\Big) + (x \cdot c_3)\Big) \cdot c_3\Big)\Big)$$

From there, the goal form order task was not able to learn any new rules. However, the gradual order task was able to apply one more rule to the expression, which moves a target variable to the left but does not put the term into fully solved form.

$$(n_0 + n_1) - t_2 \ \rightarrow_R \ n_0 - (t_2 - n_1)$$

The gradual order task was then able to learn two new rules from the resulting expression, including one in goal form order

$$(t_0 - n_1) - n_2 \ \rightarrow_R \ t_0 - (n_2 + n_1)$$

Although this rule obeys the goal form order, the goal form order task never encountered the input expression that would have matched its LHS, and so it never synthesized the rule.

It is perhaps not surprising that a TRS is able to successfully rewrite more expressions than a TRS that is its strict subset. More surprising is the fact that the goal form TRS was able to solve one expression that the gradual TRS could not. On the benchmark expression [3]

$$(((y + z) - x) \cdot c_1 - w) + (x \cdot c_1 + u)$$

the goal form TRS immediately applies a rule that cancels out the target variable.

$$((((n_0 - t_1) \cdot n_2) - n_3) + ((t_1 \cdot n_2) + n_4)) \ \rightarrow_R \ ((n_4 - n_3) + (n_0 \cdot n_2))$$
$$(((y + z) - x) \cdot c_1 - w) + (x \cdot c_1 + u) \ \rightarrow_R \ ((u - w) + ((y + z) \cdot c_1))$$

However the gradual TRS first applies a rule not in the goal form TRS

[3]Similarly simplified.

$$(n_0 + n_1) - t_2 \;\rightarrow_R\; n_0 - (t_2 - n_1)$$

$$(((y + z) - x) \cdot c_1 - w) + (x \cdot c_1 + u) \;\rightarrow_R\; ((y - (x - z)) \cdot c_1) - w) + (x \cdot c_1 + u)$$

Although the gradual TRS contains the rule the goal form TRS used to reach solved form, this additional rewrite means that the rule no longer applies, and the gradual TRS gets stuck. We will discuss this type of problem more fully in section 6.1.2, but for now we will observe that adding new rules is guaranteed to absolutely improve performance on benchmarks.

In developing our synthesis process, we constructed several other variable solver TRSs; we compare their performance against our two synthesized TRSs here (see table 5.2 for the full results). First, we translated the 'rules' in the C++ version of the variable solver currently in the Halide compiler into formal rewrite rules. We removed any rules that were guarded by predicates, as our synthesis process does not currently support rule predicates. We then filtered that ruleset by the two reduction orders for comparison. (In fact, when running the unfiltered ruleset on the benchmarks, it failed to terminate, from which we can conclude that there is no reduction order that can be made to fit the full original ruleset. The Halide compiler version terminates now because it does not recurse on every rewrite but in certain cases simply returns.)

When the 'original' ruleset was filtered by the gradual order, it consisted of 75 rules, more than twice the size of the synthesized gradual TRS. However it performed surprisingly poorly, solving only 106 benchmarks. However, it was able to solve 21 benchmarks that the synthesized gradual TRS was not.

When verifying the rewrite rules in the simplifier, we had to prove some of them were sound using the proof assistant Coq. The Coq standard library contains a large set of integer axioms, and we wondered how powerful those axioms might be as a TRS. We chose the relevant integer axiom libraries, scraped them for axioms, and transformed those axioms into rewrite rules by considering each assignment of target-matching and non-target-matching status to the variables in each equality and retaining any equality that could be oriented

using one of our two reduction orders. The resulting TRS using the gradual reduction order was quite large, 480 rules, but was only able to solve 792 benchmarks, 174 fewer than the synthesized ruleset. It was able to solve 107 benchmarks that the synthesized TRS could not, however. Filtering the Coq TRS by the goal form order resulted in a much smaller ruleset (148 rules) with very similar solving performance (787 solved benchmarks).

In an earlier synthesis experiment, we tried using smaller inputs for our training set. Our reasoning was that smaller inputs would quickly yield small, highly general rules, which would allow a TRS to be bootstrapped more quickly. We sorted the 4218 benchmarks by size and chose the smallest 400 expressions as a training set, synthesizing using the gradual reduction order as a specification. The resulting 38-rule TRS was able to solve only 492 of the full benchmark set; it was able to solve 66 expressions that the first synthesized gradual ruleset was not. Its training set contained patterns that the randomly selected training set did not; for example, both TRSs learned the rule $n_0 \leq t_1 \ \rightarrow_R \ t_1 \geq n_0$, but only the TRS trained on the shorter expressions learned the rule $n_0 \geq t_1 \ \rightarrow_R \ t_1 \leq n_0$. For comparison's sake, we filtered the small-inputs synthesized TRS by the goal form order, resulting in a TRS of 29 rules. This was the only case in which the goal form order version of a TRS performed markedly worse than its gradual counterpart; it solved only 251 benchmarks and only 44 that the synthesized gradual TRS could not solve.

Finally, observing that all of these TRSs solved at least one expression that the synthesized gradual TRS could not, we took the union of *all* of the TRSs by simply concatenating all the rulesets into one. The resulting union TRS performed the best of all, solving 1124 benchmarks including 179 that the synthesized gradual TRS could not. Filtering the union TRS by the goal form order, however, resulted in even better performance with 1180 expressions solved. Much like the one expression that the synthesized goal form TRS could solve but the synthesized gradual TRS could not, clearly some of the rules in the union gradual TRS are preventing helpful rewrites from occurring. Again, we will further discuss this issue in 6.1.2.

Table 5.2: Comparison of the performance of several variant TRSs on the variable solver benchmark set.

Name	$\|R\|$	Solved exprs	Solved exprs not solved by Gradual
Synthesized Gradual	32	966	0
Synthesized Goal form	21	949	1
Original (gradual)	75	106	21
Original (goal form)	53	105	20
Coq axiom	480	792	107
Coq axiom (goal form)	148	787	103
Small inputs	38	492	66
Small inputs (goal form)	29	251	44
Union	625	1124	179
Union (goal form)	253	1180	218

5.5 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.

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.

Other software that can solve equations in the manner of the Halide variable solver include SymPy [Meurer et al., 2017], MATLAB [Higham and Higham, 2016], and GNU Octave [Eaton et al., 1997]. These languages use a set of rewrite rules to perform this task as well. However, none of them precisely fulfill the variable solver's objective, generally solving only equalities or inequalities.

Chapter 6
FUTURE WORK AND CONCLUSION

Our work in synthesizing term rewriting systems suggests several further directions. We detail some intended future work here and conclude.

6.1 Future Work

6.1.1 Replacing the Halide variable solver

In our evaluation (section 5.4), we showed that we were able to synthesize effective term rewriting systems to address the variable solver's task. However, our ultimate, practical aim is to synthesize a term rewriting system that can replace the existing variable solver in the Halide compiler altogether. There are three main issues that need to be addressed before we can achieve that goal.

Support for rule predicates

In our previous evaluation, we did not attempt to synthesize predicates that must be fulfilled for a rule to be applied, as we did in the simplifier work. For that reason, we replaced all constant values in our benchmarks with fresh variables. However, calls to the variable solver frequently involve constants that are known at compiler time, as seen in the `TrimNoOps` example in section 2.3.4. Rules that can exploit information about those constants to apply further rewrites could be crucial in solving expressions that our synthesized TRSs cannot solve now.

Our process for synthesizing rule predicates for the simplifier was fairly involved and computationally expensive. However, we believe we could take a much simpler approach for the variable solver. For the simplifier, we first synthesized a RHS for a LHS expression that

contained constants, then replaced those constants with fresh variables. If the equivalence between LHS and RHS no longer held, we attempted to synthesize a predicate that implied that equivalence. For the variable solver, rather than attempt to synthesize arbitrary predicates, we believe we could get similar performance with a small set of fixed predicates, such as $c > 0$, $c < 0$, $c \neq 0$, and conjunctions of the above. For each candidate LHS, we might choose one of those fixed predicates and a (symbolic) constant in the LHS to apply the predicate to, and ask the solver to find a RHS that is equivalent to the LHS assuming that predicate is true. This would multiply the number of queries to the solvers up to the size of the predicate set, but for a relatively small set this is still highly tractable.

Support for division and modulo

Integer division and modulo operations are difficult for SMT solvers such as Z3 to reason about, as discussed in section 4.1.2. In earlier experiments, candidate LHSs that contained division and modulo operations almost always resulted in solver queries that timed out, and almost no RHS expressions that contained those operations were ever synthesized. For that reason, in the variable solver experiments in section 5.4, we skipped all candidate LHSs that contained division or modulo and removed those operations from the language grammar from which RHSs were synthesized. This saved large amounts of processing time and made virtually no difference to the synthesized results. However, the variable solver is frequently invoked on expressions that contain those operations, so it will be important for the variable solver TRS to be able to handle them. (Indeed, the TRS based on Coq axioms described in section 5.4 included many rules which used division and modulo, which may be why it was able to solve many benchmarks that our synthesized TRSs could not, even though it solved few benchmarks than the synthesized TRSs overall.) We plan to investigate more performant ways to reason about these operations, such as approximating integers with bitwidth encodings.

Conclusive benchmarks

Our evaluation showed that a relatively small, randomly sampled training set is sufficient to learn a ruleset that is capable of solving a sizable fraction of a much larger test set. However, in order to propose a TRS that can serve as a substitute for the existing variable solver, we need a benchmark set that acts as a good proxy for realistic workloads the Halide compiler will encounter in the wild. We propose to work with Halide developers to gather such a benchmark set. We also did not demonstrate that a TRS-based variable solver with greater solving power would produce beneficial downstream effects in compiled pipelines, as we did for the simplifier; such a demonstration would be a useful argument for replacing the existing code with a synthesized TRS.

No matter how confident we are in our choice of benchmarks, it is difficult to draw a bright line and declare a synthesized TRS to be good enough—it is always possible to continue learning new rules, as our problem remains undecidable in the general case.

6.1.2 Completion

We know by observation that the Halide simplifier 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.

As we observed in the variable solver evaluation, adding a new rule is not guaranteed to strictly increase the number of benchmarks a TRS can solve. Particularly, we saw a case in which the synthesized goal form TRS was able to solve a particular expression when the synthesized gradual TRS was not; the application of a rule that only existed in the gradual TRS blocked the use of a rule that could have fully solved the expression. This occurred because the gradual TRS was not *confluent*. In a confluent ruleset R, for any expression s, if some sequence of rule applications from R can transform s to t, and some other sequence of rule applications can transform s to t', then both t and t' must be eventually rewritten by R to the same form u. In other words, R will always normalize its inputs to the same form no matter what order rewrites are applied in.

If a ruleset R is finite and terminating, then a decision procedure exists to determine whether or not R is confluent. Thus, we are able to detect if adding a new rule to a ruleset we are synthesizing would change it from confluent to non-confluent, giving rise to potential issues like the failing benchmark above. If we can detect these issues, can we address them at the time the rule is synthesized?

One strategy is suggested by *completion* algorithms. A completion algorithm takes a set of (undirected) equalities and a reduction order and returns a terminating, confluent term rewriting system that has the same expressive power as the set of equalities. As a term rewriting system is also a set of (directed) equalities, these algorithms can also transform a non-confluent TRS into a confluent one. However, success is not guaranteed; the algorithm may succeed, fail, or fail to terminate. Thus naively running a completion algorithm on a synthesized ruleset will not necessarily prevent these issues and may also be computationally expensive. However, we plan to investigate how a modified completion algorithm could be used to prevent some instances of non-confluence without paying a large penalty in either performance or in ruleset size.

6.1.3 Other Future Work

As discussed earlier, choosing a gradual order that describes a strategy or gradient towards a desired goal form is not a trivial task and requires human intuition. Not only must the order describe a gradient over terms towards the goal, but it also must be a valid reduction order; if it is not, then the resulting term rewriting system (as well as the synthesis pipeline that produces it) is not guaranteed to terminate or indeed to actually make progress towards the goal. Proving an order is a valid reduction order is sometimes tricky, and appealing orders are often found not to be valid reduction orders after closer inspection. Reduction orders over terms often follow a set of patterns, such as orders that reduce the size of expressions or reduce the number of particular operations or variables. It would be a useful contribution to produce a handbook of these reduction order families, as well as strategies for efficiently encoding them into sketches for synthesis.

Finally, we intend to explore other applications of this work to other problems, other types of rewriting, and other domains. For example, synthesizing rewrite rules for a very different rewrite algorithm such as that used in an e-graph might require a very different strategy. Synthesizing a term rewriting system for a decidable theory, such as bitwidths of finite width or linear integer arithmetic, might also look very different.

6.2 Conclusion

Term rewriting systems can serve as a key reasoning component within compilers, allowing them to successfully apply optimizations to produce performant code. In this work, we have demonstrated how formal methods such as verification and synthesis can assist compiler authors in creating and maintaining correct and powerful TRS-based reasoning engines. We have shown how these techniques can be used on existing, handwritten TRSs, by first providing proofs of correctness and termination for an existing system, and then using synthesis to grow the system's reasoning power. Our work has resulted in bug fixes and improvements to the system that have since been merged into the main Halide codebase. We then showed

how synthesis can be used to create a new term rewriting system, saving the programming effort of writing rules by hand. We argued that an efficient way to write specifications for a term rewriting system is through a reduction order, and discussed ways of translating the purpose of a TRS into this formalism. We showed that our refined synthesis pipeline is able to produce general and robust TRSs that perform well when compared to TRSs created using other means for the same purpose. Finally, we presented some directions for future work, including the goal of synthesizing a TRS that can act as a drop-in replacement for an existing compiler component written in general-purpose code.

Appendix A

HALIDE EXPRESSION GRAMMAR

⟨*Expr*⟩ ::= ⟨*BoolExpr*⟩
| ⟨*IntExpr*⟩
| ⟨*VectorExpr*⟩

⟨*BoolExpr*⟩ ::= '`true`'
| '`false`'
| ⟨*IntExpr*⟩ '`<`' ⟨*IntExpr*⟩
| ⟨*IntExpr*⟩ '`>`' ⟨*IntExpr*⟩
| ⟨*IntExpr*⟩ '`<=`' ⟨*IntExpr*⟩
| ⟨*IntExpr*⟩ '`>=`' ⟨*IntExpr*⟩
| ⟨*IntExpr*⟩ '`=`' ⟨*IntExpr*⟩
| ⟨*IntExpr*⟩ '`!=`' ⟨*IntExpr*⟩
| ⟨*VectorExpr*⟩ '`<`' ⟨*VectorExpr*⟩
| ⟨*VectorExpr*⟩ '`>`' ⟨*VectorExpr*⟩
| ⟨*VectorExpr*⟩ '`<=`' ⟨*VectorExpr*⟩
| ⟨*VectorExpr*⟩ '`>=`' ⟨*VectorExpr*⟩
| ⟨*VectorExpr*⟩ '`=`' ⟨*VectorExpr*⟩
| ⟨*VectorExpr*⟩ '`!=`' ⟨*VectorExpr*⟩
| ⟨*BoolExpr*⟩ '`&&`' ⟨*BoolExpr*⟩
| ⟨*BoolExpr*⟩ '`||`' ⟨*BoolExpr*⟩
| '`!`' ⟨*BoolExpr*⟩

⟨*IntExpr*⟩ ::= ⟨*IntExpr*⟩ '`+`' ⟨*IntExpr*⟩
| ⟨*IntExpr*⟩ '`-`' ⟨*IntExpr*⟩

| ⟨*IntExpr*⟩ '`*`' ⟨*IntExpr*⟩

| ⟨*IntExpr*⟩ '`/`' ⟨*IntExpr*⟩

| ⟨*IntExpr*⟩ '`%`' ⟨*IntExpr*⟩

| '`max`' ⟨*IntExpr*⟩ ⟨*IntExpr*⟩

| '`min`' ⟨*IntExpr*⟩ ⟨*IntExpr*⟩

| '`select`' ⟨*BoolExpr*⟩ ⟨*IntExpr*⟩ ⟨*IntExpr*⟩

| integers

⟨*VectorExpr*⟩ ::= '`broadcast`' ⟨*IntExpr*⟩

| '`ramp`' ⟨*IntExpr*⟩ ⟨*IntExpr*⟩

| ⟨*VectorExpr*⟩ '`+`' ⟨*VectorExpr*⟩

| ⟨*VectorExpr*⟩ '`-`' ⟨*VectorExpr*⟩

| ⟨*VectorExpr*⟩ '`*`' ⟨*VectorExpr*⟩

| ⟨*VectorExpr*⟩ '`/`' ⟨*VectorExpr*⟩

| ⟨*VectorExpr*⟩ '`%`' ⟨*VectorExpr*⟩

| '`max`' ⟨*VectorExpr*⟩ ⟨*VectorExpr*⟩

| '`min`' ⟨*VectorExpr*⟩ ⟨*VectorExpr*⟩

| '`select`' ⟨*BoolExpr*⟩ ⟨*VectorExpr*⟩ ⟨*VectorExpr*⟩

Appendix B

THE FULL HALIDE SIMPLIFIER REDUCTION ORDER

$$s >_{\mathcal{V}ar} t \text{ iff } \exists x.|s|_x > |t|_x \wedge \forall c \in \Sigma^0.\neg\exists\sigma.\sigma(x) = c \tag{B.1}$$

The order $>_{\mathcal{V}ar}$ holds if there is one variable with strictly fewer occurrences in s than in t, and if that variable cannot be a ground term.

$$s >_{vec} t \text{ iff } |s|_{vec} > |t|_{vec} \wedge \forall x \in \mathcal{V}ar.|s|_x \geq |t|_x \tag{B.2}$$

The order $>_{vec}$ holds if there are strictly more vector operations in s than in t.

$$s >_{\text{dmm}} t \text{ iff } |s|_{\text{dmm}} > |t|_{\text{dmm}} \wedge \forall x \in V, |s|_x \geq |t|_x \tag{B.3}$$

The order $>_{\text{dmm}}$ holds if the total count of division, modulus, and multiplication operations is strictly greater in s than in t.

$$s >_{\text{leaf}} t \text{ iff } |s|_{\text{leaf}} > |t|_{\text{leaf}} \wedge \forall x \in V, |s|_x \geq |t|_x \tag{B.4}$$

We define the measure function $|s|_{\text{leaf}}$ as the number of leaves or terminals in the term s represented as an abstract syntax tree. Thus $>_{\text{leaf}}$ holds if s has more leaves than t.

$$s >_{\text{op}} t \text{ iff } \sum_{f\in\Sigma} |s|_f > \sum_{f\in\Sigma} |t|_f \wedge \forall x \in V, |s|_x \geq |t|_x \tag{B.5}$$

The order $>_{\text{op}}$ holds if the count of all operations (all symbols in Σ that are not zero-arity) in term s is strictly greater that of t.

$$
\begin{aligned}
s >_{0/1} t \text{ iff } & |s|_{leaf} - |s|_0 + |s|_1 > |t|_{leaf} - |t|_0 - |t|_1 \\
& \wedge |s|_{leaf} = |t|_{leaf} \\
& \wedge \forall x \in \mathcal{V}ar(s_1).|s_1|_x \geq |s_2|_x \\
& \wedge \forall x \in \mathcal{V}ar(s).\neg\exists\sigma.(\sigma(x) \neq 0 \wedge \sigma(x) \neq 1)
\end{aligned}
\tag{B.6}
$$

The order $>_{0/1}$ holds if there are strictly more occurrences of the constants 0 and 1 in t than in s and if for all variables in s, there is no substitution such that x can be substituted with the constants 0 or 1.

$$
s >_f t \text{ iff } |s|_f > |t|_f \wedge \forall x \in \mathcal{V}ar.|s|_x \geq |t|_x \tag{B.7}
$$

This order represents a composition of orders over each operation in the Halide expression grammar. The operations are checked in this order:

1. `ramp`
2. `broadcast`
3. `select`
4. division
5. multiply
6. modulus
7. subtraction
8. addition
9. `min`, `max`

10. or
11. and
12. $\geq$
13. $>$
14. $\leq$
15. $<$
16. $\neq$
17. $=$
18. not

$$s >_{lpo} t \tag{B.8}$$

The order $>_{lpo}$ is a lexicographic path order induced by an order defined over the Halide operations. We further refine the Halide operations by distinguishing addition, multiplication, select, min, and max where both arguments must not be ground terms from their counterparts whose arguments may be a ground term. We also separate subtraction where the left operand is the constant 0 from substractions where the left operand cannot be the constant 0.

- add by constant $<$
- multiply by constant $<$
- {subtract from 0, select from constant } $<$

- max with constant $<$
- $\{$min with constant, max, min, add, mul, subtract $\}$ $<$
- all other operators

$$s >_{str} t \text{ iff } \phi_{str}(s) < \phi_{str}(t) \wedge \forall x \in V, |s|_x \geq |t|_x \tag{B.9}$$

The order $>_{str}$ is checked by calculating the function ϕ_{str}, which traverses the input term depth-first and replaces each constant with "b" and each variable with "a". The resulting strings are then compared lexicographically.

$$s >_{negc} t \text{ iff } |s|_{negc} > |t|_{negc} \wedge \forall x \in V, |s|_x \geq |t|_x \tag{B.10}$$

The order $>_{negc}$ holds when s contains strictly more negative constants than t.

$$s >_{nmodc} t \text{ iff } |s|_{nmodc} > |t|_{nmodc} \wedge \forall x \in V, |s|_x \geq |t|_x \tag{B.11}$$

The order $>_{nmodc}$ holds when s contains strictly more constants that are not within the range $0 \leq c < |c_m|$, for some constant c_m.

$$s >_{uniq} t \text{ iff } |s|_{uniq} > |t|_{uniq} \wedge \forall x \in V, |s|_x \geq |t|_x \tag{B.12}$$

The order $>_{uniq}$ holds when s contains strictly more unique subtrees than t.

Finally, these two rules are well-ordered because they associate min and max to the left.

$$\texttt{max}(\texttt{max}(x, y), \texttt{max}(z, w)) \rightarrow_R \texttt{max}(\texttt{max}(\texttt{max}(x, y), z), w) \tag{max106}$$

$$\texttt{min}(\texttt{min}(x, y), \texttt{min}(z, w)) \rightarrow_R \texttt{min}(\texttt{min}(\texttt{min}(x, y), z), w) \tag{min106}$$

BIBLIOGRAPHY

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. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322967. URL `http://doi.acm.org/10.1145/3306346.3322967`.

Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. Learning stateful preconditions modulo a test generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127.

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

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 775–788, 2016.

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

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

Eric Butler, Emina Torlak, and Zoran Popović. A framework for computer-aided design of

educational domain models. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 138–160. Springer, 2018.

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

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

Patrick Cousot and Radhia Cousot. 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*, pages 238–252. ACM, 1977.

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

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

John Wesley Eaton, David Bateman, Søren Hauberg, et al. *Gnu octave*. Network thoery London, 1997.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686.

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with aprove. In *International Conference on Rewriting Techniques and Applications*, pages 210–220. Springer, 2004.

Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

Saul Gorn. Handling the growth by definition of mechanical languages. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 213–224, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465513. URL `http://doi.acm.org/10.1145/1465482.1465513`.

Desmond J Higham and Nicholas J Higham. *MATLAB guide*. SIAM, 2016.

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

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

DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 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, 2020.

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

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 22–32, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686.

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

David Menendez and Santosh Nagarakatte. 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*, PLDI 2017, page 49–63, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062372. URL `https://doi.org/10.1145/3062341.3062372`.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3: e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL `https://doi.org/10.7717/peerj-cs.103`.

C.G. Nelson. Techniques for program verification[ph. d. thesis]. 1980.

Julie L Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. Verifying and improving halide's term rewriting system with program synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically

improving accuracy for floating point expressions. In *ACM SIGPLAN Notices*, volume 50, pages 1–11. ACM, 2015.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 297–310, New York, NY, USA, 2016a. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872387. URL `http://doi.acm.org/10.1145/2872362.2872387`.

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

Alex Reinking. Formal semantics for the halide language. Master's thesis, University of California at Berkeley, 2019.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer, 2017a.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *CoRR*, abs/1711.04422, 2017b. URL `http://arxiv.org/abs/1711.04422`.

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

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

Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul,*

Korea, December 14-16, 2009. Proceedings, volume 5904 of *Lecture Notes in Computer Science*, pages 4–13. Springer, 2009. doi: 10.1007/978-3-642-10672-9_3. URL `https://doi.org/10.1007/978-3-642-10672-9_3`.

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

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.