TensorRight: Automated Verification of Tensor Graph Rewrites

JAI ARORA, University of Illinois Urbana-Champaign, USA
SIRUI LU, University of Washington, USA
DEVANSH JAIN, University of Illinois Urbana-Champaign, USA
TIANFAN XU, University of Illinois Urbana-Champaign, USA
FARZIN HOUSHMAND, Google, USA
PHITCHAYA MANGPO PHOTHILIMTHANA, Google DeepMind, USA
MOHSEN LESANI, University of California, Santa Cruz, USA
PRAVEEN NARAYANAN, Google, USA
KARTHIK SRINIVASA MURTHY, Google, USA
RASTISLAV BODIK, Google DeepMind, USA
AMIT SABNE, Google, USA
CHARITH MENDIS, University of Illinois Urbana-Champaign, USA

Tensor compilers, essential for generating efficient code for deep learning models across various applications, employ tensor graph rewrites as one of the key optimizations. These rewrites optimize tensor computational graphs with the expectation of preserving semantics for tensors of arbitrary rank and size. Despite this expectation, to the best of our knowledge, there does not exist a fully automated verification system to prove the soundness of these rewrites for tensors of arbitrary rank and size. Previous works, while successful in verifying rewrites with tensors of concrete rank, do not provide guarantees in the unbounded setting.

To fill this gap, we introduce TensorRight, the first automatic verification system that can verify tensor graph rewrites for input tensors of arbitrary rank and size. We introduce a core language, TensorRight DSL, to represent rewrite rules using a novel axis definition, called *aggregated-axis*, which allows us to reason about an unbounded number of axes. We achieve unbounded verification by proving that there exists a bound on tensor ranks, under which bounded verification of all instances implies the correctness of the rewrite rule in the unbounded setting. We derive an algorithm to compute this rank using the denotational semantics of TensorRight DSL. TensorRight employs this algorithm to generate a finite number of bounded-verification proof obligations, which are then dispatched to an SMT solver using symbolic execution to automatically verify the correctness of the rewrite rules. We evaluate TensorRight's verification capabilities by implementing rewrite rules present in XLA's algebraic simplifier. The results demonstrate that TensorRight can prove the correctness of 115 out of 175 rules in their full generality, while the closest automatic, *bounded*-verification system can express only 18 of these rules.

Authors' Contact Information: Jai Arora, University of Illinois Urbana-Champaign, USA, jaia3@illinois.edu; Sirui Lu, University of Washington, USA, siruilu@cs.washington.edu; Devansh Jain, University of Illinois Urbana-Champaign, USA, devansh9@illinois.edu; Tianfan Xu, University of Illinois Urbana-Champaign, USA, tianfan3@illinois.edu; Farzin Houshmand, Google, USA, farzinh@google.com; Phitchaya Mangpo Phothilimthana, Google DeepMind, USA, mangpo@google.com; Mohsen Lesani, University of California, Santa Cruz, USA, mlesani@ucsc.edu; Praveen Narayanan, Google, USA, pravnar@google.com; Karthik Srinivasa Murthy, Google, USA, ksmurthy@google.com; Rastislav Bodik, Google DeepMind, USA, rastislavb@google.com; Amit Sabne, Google, USA, asabne@google.com; Charith Mendis, University of Illinois Urbana-Champaign, USA, charithm@illinois.edu.

ACM 2475-1421/2025/1-ART29
https://doi.org/10.1145/3704865

CCS Concepts: • **Theory of computation → Semantics and reasoning**; **Logic and verification**; **Automated reasoning**; • **Software and its engineering → Compilers**; **Formal methods**.

Additional Key Words and Phrases: Unbounded Verification, Tensor Compilers, Denotational Semantics

ACM Reference Format:
Jai Arora, Sirui Lu, Devansh Jain, Tianfan Xu, Farzin Houshmand, Phitchaya Mangpo Phothilimthana, Mohsen Lesani, Praveen Narayanan, Karthik Srinivasa Murthy, Rastislav Bodik, Amit Sabne, and Charith Mendis. 2025. TensorRight: Automated Verification of Tensor Graph Rewrites. *Proc. ACM Program. Lang.* 9, POPL, Article 29 (January 2025), 32 pages. https://doi.org/10.1145/3704865

1 Introduction

Deep learning frameworks, such as TensorFlow [1], PyTorch [31], and JAX [8], along with their backend optimizing tensor compilers, such as XLA [11] and TorchInductor [2], have been instrumental in enabling machine learning (ML) practitioners to experiment, train, and deploy various neural network architectures. These tensor compilers manipulate computations with tensors as first-class objects, utilizing tensor computational graphs as their intermediate representation (IR). The nodes in these graphs represent tensor operators, while the edges denote input/output tensors. Examples include XLA's High Level Operators (XLA-HLO) [12], PyTorch's torch.fx operators [33], and ONNX's tensor operators [10]. Middle-end tensor compiler optimizations often transform these tensor graphs to produce more efficient variants. A key optimization which has attracted significant research [20, 38, 41] is tensor graph rewrites. This optimization is a common pass in industrial tensor compilers such as XLA[1].

Tensor graph rewrites transform a subgraph of the original tensor graph to an equivalent version that is more efficient. For example, consider the dot (einsum) operator that takes two tensors and a set of contraction and batch axes as input and performs sum-of-products over the specified contraction axes. If the batch and contraction axes are empty (precondition), an expensive dot operation may be decomposed into a simpler composition of element-wise multiplication and expand operations (represented as $\text{dot(A,B)} \Rightarrow_C \text{binary}(\text{expand(A)}, \text{expand(B)}, *)$ in our notation, with precondition C). In general, these rewrite rules are expected to be correct for tensors of arbitrary rank (number of axes) and size (individual axis sizes). We term this property as the rewrite rules being *rank-* and *size-polymorphic*. We found that most tensor graph rewrites in XLA's algebraic simplifier have this property. Hence, it is important that compiler developers ascertain that the rules are indeed correct for input tensors of arbitrary rank and size.

There have been multiple efforts at formally proving the correctness of these rewrites. However, *automatically* verifying tensor graph rewrites for tensors of arbitrary rank and size has remained challenging. Previous automatic verification techniques instantiate fixed-ranked, concrete-sized tensors with symbolic values as a part of their verification process. As a result, their proofs do not generalize to the *unbounded setting*, where input tensors can be of arbitrary rank and size. Further, existing verification systems do not support preconditions on rules, which we find abundant in compilers such as XLA. For example, TASO [20] proposes an axiomatic approach to verify tensor graph rewrites. The rewrite rules they synthesize from their axiom pool are rank- and size-polymorphic. However, the axioms themselves are only verified on small, concrete-sized input tensors. TENSAT [41] improves the search efficiency of TASO, but relies on TASO's rewrite rule synthesis and verification process. PET [38] uses statistical testing to give rigorous guarantees on more expressive rewrites for tensor computational graphs of concrete-rank and concrete-sized tensors. Successful works that have verified tensor graph rewrites in the unbounded setting have been manual verification efforts with proof assistants like Coq [25, 26].

[1] https://github.com/openxla/xla/blob/main/xla/hlo/transforms/simplifiers/algebraic_simplifier.cc

In this paper, we introduce the first automatic, push-button verification system, TensorRight, that allows users to succinctly express and verify tensor graph rewrite rules for input tensors of arbitrary rank and size. Further, in order to aid tensor compiler developers, we develop TensorRight to be able to handle the complexities of rewrite rules found in the XLA compiler. We have to overcome several key challenges in realizing these goals.

Representation. First, we need to succinctly represent tensor graph rewrite rules in a way that allows reasoning about their correctness in the unbounded setting. Second, we need to model the highly parameterized operators in XLA-HLO. For example, XLA-HLO's conv operator works with arbitrary batch, contraction, and spatial axes specifications and has rich padding and dilation attributes. Further, XLA rewrite rules can be guarded by complicated preconditions.

We overcome these challenges by designing a rewrite rule specification language, called TensorRight DSL, with tensor operators closely resembling those in XLA-HLO. TensorRight DSL introduces a novel axis definition called *aggregated-axis* that represents a possibly unbounded set of axes, rather than capturing one axis at a time. A tensor in TensorRight DSL consists of a *finite* number of aggregated-axes, where they can potentially be instantiated to any number of axes. This allows us to reason about how input tensors are mutated by tensor operators, treating similar axes collectively. All TensorRight DSL operators are defined to work with aggregated-axes, making the rewrite specifications rank- and size-polymorphic. Additionally, the representation with *aggregated-axes* is general enough to support a sizable subset of tensor operators and their parameterizations, as defined in XLA-HLO (e.g. conv). However, the representation cannot support *layout-sensitive* operators, such as reshape and bitcast. We implemented the most common operators appearing in XLA's rewrite rules to demonstrate TensorRight DSL's expressivity. Rewrite specifications in TensorRight DSL accept preconditions, which can also be rank- and size-polymorphic. Finally, we provide denotational semantics of these operators, which we use to verify these rules.

Verification. The next major challenge that we need to overcome is: given specifications in the TensorRight DSL, how can we automatically verify that the rewrites are correct? This requires proving them correct in the unbounded setting. Instantiating unbounded tensors is not feasible, while using symbolic tensors of concrete-rank and size during automatic verification can result in proofs that only hold for input tensors of that particular rank and size, as shown in §2.3.

To handle unbounded sizes, we leverage the capabilities of SMT solvers to perform unbounded reasoning using uninterpreted functions and unbounded integers. For unbounded ranks, we overcome the challenge by proving that there exists a bound on the ranks, such that if we prove the rule correct for all possible ranks within the bound, then the rule is also correct for arbitrary ranks and sizes. This allows us to reduce the unbounded verification problem to a set of bounded verification cases, which can be dispatched to an automatic verification engine. We derive a bound inference algorithm using the denotational semantics of TensorRight DSL. Given these theoretical foundations, TensorRight automatically verifies a given rewrite rule written in TensorRight DSL in two steps. First, it uses the bound inference algorithm to find a *sufficient* rank for each aggregated-axis. Next, for all ranks equal to or below this bound, TensorRight instantiates concrete-ranked input tensors for each aggregated-axis. It then uses big-step operational semantics, derived from the denotational semantics of TensorRight DSL, to create proof obligations as SMT queries using symbolic execution. If all of these bounded cases are proven by an SMT solver, TensorRight then concludes that the rewrite rule is correct.

We implement the TensorRight system in Haskell and use Grisette [29] as the symbolic evaluation engine. We further provide a TensorRight Frontend that abstracts away some core TensorRight DSL constructs to make developing rewrite rules easier. TensorRight dispatches all SMT queries to the Z3 [15] SMT solver to ascertain the soundness of a given rewrite rule. To evaluate

TensorRight's capabilities in representation and verification, we assessed it using a comprehensive set of rules incorporated within XLA's algebraic simplifier. We successfully represented 121 of these rules and verified 115 of them in TensorRight. Almost all rules were verified in the unbounded setting within a second. Comparatively, other bounded-verification systems, such as TASO [20] and PET [38], could only express 14 and 18 rules and verify 6 and 16 rules, respectively, exemplifying TensorRight's representation and verification capabilities. Further, we show case studies where TensorRight helps generalize rewrites with complicated preconditions, showcasing its usefulness during compiler development.

In summary, this paper makes the following contributions.

- We present a language, TensorRight DSL (§4) to specify tensor graph rewrites with (1) a novel axis construct called aggregated-axes, allowing representation of operators and rewrite rules that are rank- and size-polymorphic (2) operator specifications that closely resemble XLA-HLO (3) precondition specifications on rewrite rules.
- We provide denotational semantics for TensorRight DSL (§5). To the best of our knowledge, this is the first formalization of a sizable subset of operators in a production-quality tensor IR.
- We provide the first *automatic* verification strategy (§6) that can reason about the correctness of tensor graph rewrites that are rank- and size-polymorphic.
- We develop TensorRight that implements this verification strategy and evaluate it (§8) by representing and verifying tensor graph rewrites present in XLA's algebraic simplifier.

TensorRight is open-source, publicly available at https://github.com/ADAPT-uiuc/TensorRight

2 Background and Motivation

We first provide background on tensors and related concepts before motivating the need for *automatic* and *unbounded* verification of tensor graph rewrites. We then describe a key insight of our unbounded-verification methodology.

2.1 Preliminaries

Tensors are a generalization of scalars (0-dimensional tensors), vectors (1-dimensional tensors), and matrices (2-dimensional tensors) to n-dimensional objects. A popular implementation of a tensor is multi-dimensional arrays. An *axis* of a tensor (also commonly known as a dimension) represents a direction across which the tensor's data can be traversed. The *rank* of a tensor (also commonly known as its dimensionality) refers to the number of axes the tensor has. The *shape* of a tensor describes the *size* of each axis, i.e., the number of elements that exist along each axis. The *size* of a tensor refers to the total number of elements in the tensor, calculated as the product of individual axis-sizes. The axes of an n-dimensional tensor are numbered from 0 up to $n-1$. Each element of a tensor is uniquely identified by a list of *positional* indices, with one index for each axis.

For example, a 2-dimensional tensor m, containing 3 groups of elements along axis 0 and 4 groups of elements along axis 1, has a rank of 2, a shape of 3×4, and a size of 12. Such a tensor can be accessed by a pair of positional indices: $m[i, j]$ denotes the value at the i^{th} position along axis 0 and j^{th} position along axis 1. Another way to implement tensors, called *named tensors*, assigns explicit names to the axes of a tensor, referred to as *named-axes*. TensorRight adopts the latter approach, which we describe in detail in §4.1.

Tensor Graph Rewrites. A tensor operator refers to any operation that takes tensors as input and returns tensors as output. A tensor computational graph is a directed acyclic graph that represents a sequence of tensor operations. The nodes in the graph represent tensor operators, while the edges indicate the flow of data (tensors) between these operators. Tensor graph rewriting is a key optimization employed by tensor compilers, which replaces a subgraph of the input graph with

another, equivalent subgraph, subject to certain preconditions. This optimization is governed by a set of *rules*, called tensor graph rewrite rules.

The algebraic simplifier of the XLA compiler contains hundreds of tensor graph rewrites, executed during program compilation. Given that XLA consistently executes this simplifier, it is crucial to ensure the correctness of these rewrite rules. However, currently the rewrite rules are not verified. Therefore, the developers rely on unit tests and limit the generality of the rules to alleviate concerns of introducing compiler bugs.

We aim to automatically verify tensor graph rewrites deployed in XLA, which work with tensors of arbitrary ranks and sizes. Existing verified tensor graph rewrite systems are either not automatic, lack support for complex XLA-HLO operators and preconditions, or cannot verify rewrite rules in the unbounded setting . We now demonstrate the importance of automatic and unbounded verification of tensor graph rewrites and discuss a key insight that enables us to achieve these goals.

2.2 Need for Automatic Verification

The algebraic simplifier in XLA contains complex rewrite rules whose correctness is not intuitive. Developers often limit the generality of these rewrite rules by imposing preconditions. For instance, consider the FoldConvInputPad rule shown in Fig. 1.

The idea behind the FoldConvInputPad rule is simple: fold the padding operator into the convolution operator. It folds the edge padding, i.e., the lower and higher padding (S_{lp} and S_{hp}) into the convolution padding (S_l and S_h), but does not fold the interior padding (S_{ip}) into the base dilation (S_i). The precondition of this rule requires zero interior padding ($S_{ip} = 0$) and a base dilation of one ($S_i = 1$). The XLA repository contains the following comment[2] on the preconditions:

> *Edge padding composes with itself in the straightforward way, but composing interior padding is nontrivial, and we cowardly refuse to think about it. If we see interior padding in either the kPad or conv, bail if there's any sort of padding in the other.*

FoldConvInputPad :
let $S_{ol} = S_l + S_{lp}$ in
let $S_{oh} = S_h + S_{hp}$ in
$\text{conv}(\text{pad}(t, 0, S_{lp}, S_{hp}, S_{ip}), t',$
$B, F, O,$
$S_l, S_h, S_i, S_i')$
$\Longrightarrow_{S_{ip}=0 \,\wedge\, S_i=1}$
$\text{conv}(t, t', B, F, O,$
$S_{ol}, S_{oh}, S_i, S_i')$

Fig. 1. FoldConvInputPad rule taken from XLA's Algebraic Simplifier.

Developers restrict the rule because the general case is non-trivial. Existence of an automatic verification system would allow incremental refinement of the rule and provide counterexamples during development. As a consequence, it would enable developers to build more general rewrite rules and be confident that the rewrite rules are valid for arbitrary ranks and sizes. For example, TensorRight can prove a more general version of the FoldConvInputPad rule with interior padding, as discussed in §8.3. The generalization involves removing the precondition and computing the folded padding and dilation attributes as: $S_{ol} = S_l + S_i \times S_{lp}$, $S_{oh} = S_h + S_i \times S_{hp}$, and $S_{oi} = S_i + S_i \times S_{ip}$. This folding of attributes is non-trivial to come up with, but TensorRight can prove the rewrite rule to be valid. With the aid of TensorRight, we believe that compiler engineers will be able to quickly iterate through complex rewrite rules and get feedback on their correctness through counterexamples, thereby increasing productivity.

2.3 Need for Unbounded Verification

We demonstrate with an example that verifying a tensor graph rewrite for a certain rank may not sufficient to guarantee correctness in the unbounded setting, where input tensors can be of arbitrary

[2] https://github.com/openxla/xla/blob/ac380bb187abdb3efbbac776141e3a2300209232/xla/service/algebraic_simplifier.cc#L8764-L8767

ranks and sizes. Consider the SliceDyUpSlice rule shown in Equation 1, where S represents the shape of the input tensor Y, *zero* represents a tensor with all values as 0, and $\overline{v}$ represents a vector with all values as v. Other operator inputs like start, end, and stride are called *operator attributes*.

$$\text{dyup-slice}(\text{slice}(\text{Y}, \overline{0}, \left\lfloor \frac{S+1}{2} \right\rfloor, \overline{1}), \textit{zero}, \overline{1}) \implies \text{dyup-slice}(\text{slice}(\text{Y}, \overline{0}, \text{S}, \overline{2}), \textit{zero}, \overline{1}) \quad (1)$$

(start, end, stride, update, offset)

The left-hand side (LHS) expression first applies the slice operator, which extracts a sub-tensor from Y by picking elements from the 0^{th} index (start) up to the $\lfloor \frac{S+1}{2} \rfloor^{th}$ index (end) along each axis. It is then followed by the dyup-slice operator, which *zeroes* (update) out all the points whose axes indices are greater than or equal to 1 (offset). The right-hand side (RHS) expression first applies the slice operator, which extracts a sub-tensor from Y by picking every 2^{nd} element (stride) along each axis. It is then followed by the dyup-slice operator, which zeroes out all the points whose axes indices are greater than or equal to 1.

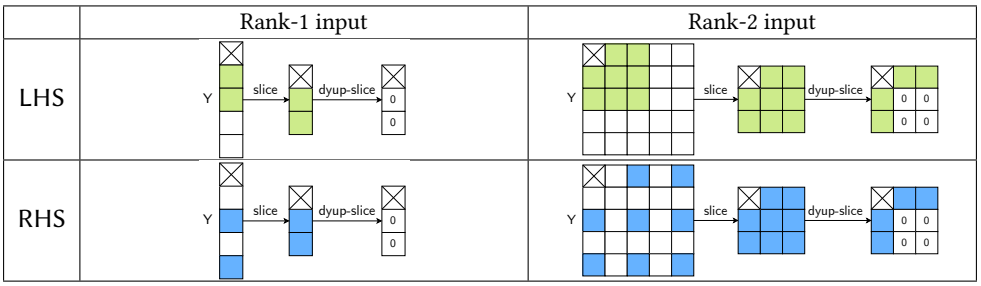


Fig. 2. Illustration for SliceDyUpSlice rule depicting various regions in the input tensor for ranks 1 and 2. The leftmost element is shown as crossed out. The green and blue regions indicate the elements extracted by slice in LHS and RHS, respectively. The zeroed out region after the dyup-slice is indicated by 0-elements.

Fig. 2 illustrates the rule applied to input tensors of rank 1 and 2. The crossed-out element corresponds to the point with all indices as 0. We refer to this as the *leftmost* element. The leftmost element is left untouched throughout the computation in both LHS and RHS. The green region, along with the leftmost element, corresponds to the sub-tensor obtained after the slice in LHS. The blue region, along with the leftmost element, corresponds to the sub-tensor obtained after the slice in RHS. The zeroed out region after the dyup-slice is represented using 0-elements.

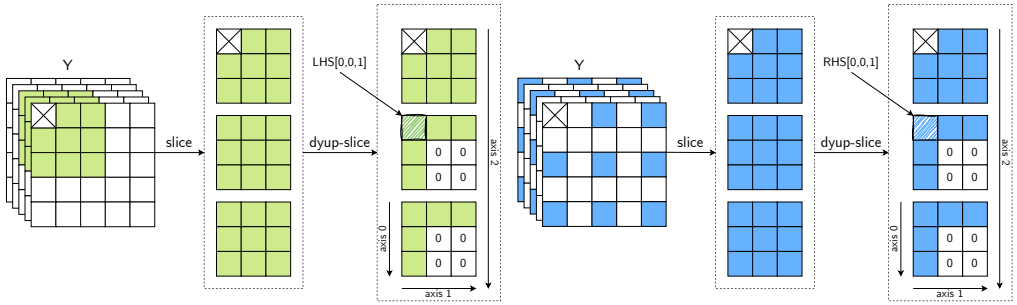
As we can observe for the 1-dimensional case, the final LHS and RHS expressions are equal since the green and blue regions get zeroed out completely. Meanwhile, for the 2-dimensional case, the green and blue regions are not completely zeroed out, so the LHS and RHS expressions have regions that do not match. Therefore, the rule is valid for rank 1 but is invalid for rank 2. In fact, the rule is invalid for any rank higher than 2. This example demonstrates that verifying the rule for a certain rank, in this case 1, does not guarantee correctness at other ranks, making it important to verify the rule for all possible ranks.

2.4 Key Observation

A rewrite rule is valid if the LHS and RHS expressions are equal for input tensors of any rank. Otherwise, the rule is invalid and would exhibit a *counterexample*. A counterexample contains a

valuation of all the variables in the rule (including tensors and operator attributes) and an access A (list of positional indices), such that LHS$[A]$ and RHS$[A]$ do not match.

Verifying a rule for each rank separately is infeasible since there are a denumerable number of such ranks. However, we make an observation that there exists a *sufficient* rank k, such that if the rule is valid for rank k, then it can be proven valid for any rank greater than k. This insight allows us to avoid verifying the rewrite rule for ranks greater than k. A more intuitive way to understand this is through its contraposition, i.e., if a counterexample exists at a rank greater than k, then a counterexample exists at rank k. We demonstrate with the same example rule from §2.3 that verifying the rule for rank 2 is sufficient to ensure correctness for all higher ranks.



(a) LHS for rank-3 input (b) RHS for rank-3 input

Fig. 3. The SliceDyUpSlice specialized for rank-3 inputs. The LHS and RHS expressions are presented using 2-dimensional cross-sections along axis 2. The access $A^3 = [0, 0, 1]$ is highlighted in LHS and RHS.

Consider the SliceDyUpSlice rule applied to input tensors of rank 3, as shown in Fig. 3. Clearly, the LHS (Fig. 3a) and RHS (Fig. 3b) have regions (green and blue) that do not match. Therefore the rule is invalid for rank 3 and exhibits a counterexample. The counterexample would contain an access A^3 at which LHS and RHS do not match. This access can correspond to any location in the green and blue regions. Without loss of generality, we consider the case when $A^3 = [0, 0, 1]$, highlighted as a sketched-out location in Fig. 3a and Fig. 3b.

Given this counterexample at rank 3, we try to construct a counterexample at rank 2, which would contain an access A^2. An obvious counterexample construction involves *projecting* out one of the axes. There are 3 choices for the axis to project out: axis 0, axis 1, and axis 2, as shown in Fig. 3. If we project out axis 2, then the resulting counterexample access A^2 would be $[0, 0]$, but LHS$[0, 0]$ and RHS$[0, 0]$ have to always match since it is the leftmost element. Therefore, this projection does not lead to a counterexample. We instead observe that projecting out any of axis 0 or axis 1 results in a counterexample at rank 2. In fact, any counterexample at rank 3 for the SliceDyUpSlice rule can be *lowered* to a counterexample at rank 2. Moreover, it can be shown that any counterexample at a higher rank can be lowered to a counterexample at rank 2. Therefore, if the SliceDyUpSlice rule is valid for rank 2, then it is valid for any higher-rank. Note that the same does not hold for rank 1: given a counterexample at rank 2, we cannot construct a counterexample at rank 1. Based on these observations, we conclude that 2 is a sufficient rank for this rule and verifying the rule for ranks 1 and 2 ensures correctness in the unbounded setting.

In TensorRight, we extend these observations to any arbitrary rule by first partitioning the axes of a tensor into "groups", where all axes in a group share the same "role" and are treated uniformly by the operators. We then present an algorithm to compute a sufficient rank for each group, allowing us to avoid verifying the rule for ranks beyond these sufficient ranks.

3 Overview

Our goal is to automatically verify rewrite rules for arbitrary tensors and operator attributes. Handling arbitrary tensors requires reasoning about tensor values, axis sizes, and ranks, all of which could be arbitrary. We illustrate the challenges in representing and verifying rewrite rules with the help of an example and present TensorRight, that helps us overcome these challenges.

3.1 TensorRight Rewrite Rules

Similar to many other tensor graph rewrite systems, TensorRight rewrites are modeled as rewriting an LHS tensor expression to an RHS tensor expression, subject to certain preconditions. The users use the constructs provided by TensorRight DSL to write tensor expressions and preconditions. We use the notation LHS $\Rightarrow_C$ RHS to represent a generic tensor graph rewrite, where LHS and RHS are tensor expressions and C is the precondition under which the rewrite rule is supposedly correct, which is verified by our system.

Example. Consider the DynsliceToSlice rule shown in Equation 2, extracted from XLA's algebraic simplifier, which desugars the dy-slice operator to the more efficient slice operator.

$$\text{dy-slice}(\mathsf{Y}, B, L) \Longrightarrow_{E-B'=L \,\wedge\, P=1 \,\wedge\, B'=B} \text{slice}(\mathsf{Y}, B', E, P) \tag{2}$$

Fig. 4 depicts the DynsliceToSlice rule visually. The dy-slice operator extracts a sub-tensor from the input tensor Y, where the start-index for each axis is specified in B and the length of the slice along each axis is passed in L. Meanwhile, the slice operator also extracts a sub-tensor from within a bounding box in the input tensor Y. The start-indices for the bounding box are specified in B', while the end-indices (exclusive) are specified in E. P specifies the stride for each axis, which determines the step size between elements in the bounding box.

The DynsliceToSlice rule is generally not correct, unless $E - B'$ (the size of the bounding box in slice) is equal to L (the length in dy-slice). The other requirements are that slice should skip no elements, i.e., $P = 1$, and the start indices in slice and dy-slice must be the same, i.e., $B' = B$. Since these are specified in the precondition, the RHS expression is equivalent to the LHS expression. Our goal is to *represent* and *verify* this rule for arbitrary tensors and operator attributes. We now discuss each challenge individually and explain how our system addresses them.

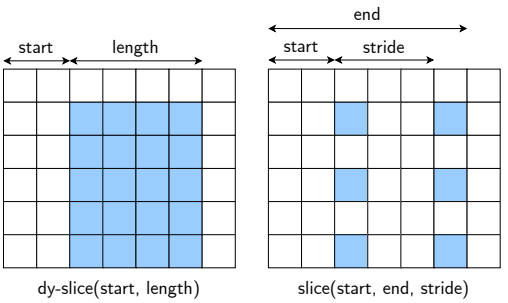


Fig. 4. Illustration of dy-slice and slice operators. The shaded regions denote the operator outputs.

3.2 Representation in TensorRight DSL

Challenges in representation. First, since the DynsliceToSlice rule should be correct for all instantiations of the tensor Y, our system should allow representing a tensor of arbitrary rank and size. Second, it should allow specifying arbitrary operator attributes like start, end, stride, and length, to ensure that the rule is correct for all possible operator attributes. It should also allow performing operations on attributes, like doing arithmetic on end and start while specifying the precondition. Third, the operators provided by the system should model those that are found in XLA-HLO. Last, it should allow defining preconditions on a rule (e.g., stride values being 1).

TensorRight Frontend. We present a frontend language in which users can express *abstract* rewrite rules along with preconditions. TensorRight will build internal representations (§4) of rewrite rules from the specification, which can be instantiated to arbitrary ranks.

```
1  rule = do
2      rcls <- newRClass "rcls"
3      [size, start, start', length, end, stride] <-
4          newMaps ["size", "start", "start'" "length", "end", "stride"] rcls
5      Y <- newTensor @TensorInt "Y" [rcls --> size]
6      lhs <- dynamicSlice Y [rcls --> start] [rcls --> length]
7      rhs <- slice Y [rcls --> start'] [rcls --> end] [rcls --> stride]
8      precondition [end, start', length] $ \[end, start', length] -> end - start' .== length
9      precondition [stride] $ \[stride] -> stride .== 1
10     precondition [start, start'] $ \[start, start'] -> start' .== start
11     rewrite "DynamicSlice(Y) => Slice(Y)" lhs rhs
12
13 verifyDSL rule
```

Listing 1. The DysliceToSlice rule represented in TensorRight DSL.

Listing 1 illustrates the DysliceToSlice rule implemented in the TensorRight Frontend. Instead of using fixed-rank tensors, the tensors in TensorRight DSL are represented with *aggregated-axes*, that can be instantiated to any number of axes. All axes in an aggregated-axis share the same "role" and are treated uniformly by all the operators, allowing us to reason about an unbounded number of axes compactly. There might be multiple aggregated-axes in a rule to capture different roles of axes. Some of them must be instantiated to the same rank in a correct rule and this constraint is represented by a *rank class* (RClass) as discussed in §6.

On line 2 in Listing 1, we declare a new RClass and we refer to it by `rcls`. In TensorRight DSL, we can refer to the aggregated-axis with the RClass itself, if an RClass has exactly one aggregated-axis. On lines 3-4, we declare multiple *abstract-maps* on `rcls`. These abstract-maps can be instantiated to *concrete-maps*, whose domain is the same as the axes in the aggregated-axis represented by `rcls`. When instantiated, they map axes in `rcls` to symbolic values, which can represent axes sizes, start indices, end indices etc. On line 5, we declare a new tensor containing integer elements, with the shape $\{\texttt{rcls} \mapsto \texttt{size}\}$. Similarly, the created tensor can also be instantiated with any number of axes (all of which behave in the same way) and symbolic sizes. We then construct the LHS and RHS expressions on lines 6 and 7, respectively. On line 8, we specify the precondition that the difference of the end and start indices should be equal to the length. On line 9, we specify that the stride values should be 1 for all axes. On line 10, we specify that the start indices should be same on both sides. Finally, we construct the rewrite rule on line 11.

This example demonstrates how we can represent tensors with arbitrary rank and sizes, specify arbitrary operator attributes, construct tensor expressions, and specify complex preconditions in TensorRight DSL. We now discuss our verification methodology built on top of this representation.

3.3 Verification

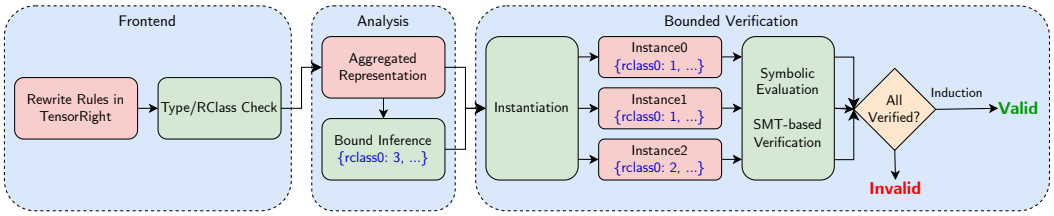


Fig. 5. TensorRight Overview and Workflow.

Fig. 5 describes our approach to verifying rules with tensors of arbitrary rank and sizes. With the aggregated representation created in §3.2, the system infers a bound for each RClass and instantiates them to all the ranks within the bound. Each instance can then be proven with symbolic evaluation by SMT solvers. This is based on a crucial theorem (§6.3) that for all RClasses in a rule, there exists a mechanically derivable bound on the ranks, such that proving all instances within the bound implies that the rule is correct for all ranks. With this bound, we convert the unbounded-verification proof obligation to a finite number of *bounded-verification* proof obligations. The correctness of the rule with ranks beyond the bound is then established with induction.

On line 13 in Listing 1, we call `verifyDSL` on the constructed rule, which is our main verification routine. First, it infers a bound for every RClass in the rule. The bound inference algorithm collects the number of unique *boolean conditions* and *tensor accesses* in the rewrite rule. In this case, there is only one RClass, no boolean conditions, and a single unique access to the tensor. We infer the bound to be 1. This means that if the rule is correct for all rank-1 tensors, then the rule is correct for tensors containing any number of axes. We discuss these conditions and accesses in detail in §6.

```
1  ruleLowered = do
2      # Instiate rcls to a concrete set
3      Ⓢ <- rcls
4      # Define concrete maps on rcls
5      size <- Map Ⓢ SymInteger
6      # Maps for other attributes like start, length
7      Y <- newTensor @TensorInt "tensor" [rcls --> size]
8      lhsSym <- [[...]] # Symbolic Representation of LHS
9      rhsSym <- [[...]] # Symbolic Representation of RHS
10     pre <- ... # Precondition of the rule
11     # Axes and Shape Checks
12     assert $ lhsShape .== rhsShape
13     A <- generalAccess(lhs, rhs)
14     verify $ pre && lhsValid -> lhsSym[A] .== rhsSym[A]
```

Listing 2. Frontend Rewrite Rule lowered to our core syntax.

Second, it specializes the rule for these ranks, ending up with fixed-rank but arbitrary-sized tensors. Listing 2 shows some details of the lowered code in our core TensorRight DSL. On line 3, we instantiate `rcls` with a known-rank (1 in this case) and get the set of *concrete*-axes in Ⓢ. On line 5, we declare concrete-maps corresponding to the abstract maps in Listing 1. These maps now have a concrete-domain, same as Ⓢ. In lines 7-10, we create concrete-ranked input tensors, generate symbolic representations of LHS and RHS with symbolic evaluation, and specify preconditions. We make an initial assertion on line 12 that both expressions have the same symbolic shape. One line 14, we express one of the main verification conditions, i.e., the equality between LHS and RHS expressions under a general access `A`, which is discharged to an SMT solver. The solver decides that the proof obligation is a tautology and the rule is deemed verified in the unbounded setting.

The example demonstrates how we take the abstract specification of a rewrite rule expressed in TensorRight Frontend, infer a bound for each RClass, instantiate every aggregated-axis, and discharge bounded-verification proof obligations. We describe the TensorRight DSL in §4, the denotational semantics in §5, and our verification methodology in detail in §6.

4 Rewrite Rule Representation

In this section, we show our rank- and size-polymorphic rewrite rule representation constructed by the TensorRight Frontend introduced in §3.2. We take a set of operators from `XLA-HLO` and model them in TensorRight. We show a subset of the modeled operators in Fig. 6 for discussion (more modeled operators can be found in Appendix A). We can extend the set to other *layout-insensitive* operators, whose semantics do not depend on the particular physical layout of the operands. Some operators that do *not* fall into this category include reshape and bitcast.

The key distinction of our DSL from XLA-HLO is that we group axes into *aggregated-axes*, where all the axes in the same group share the same "role" and are treated uniformly by the operators. This allows us to describe rewrite rules with tensors containing any number of axes in a compact manner. In §6, we extend this representation by tagging the aggregated-axes with *rank classes* to help us with the instantiation of aggregated-axes into concrete-axes for verification purposes.

4.1 Named Axes

Following named-tensors in PyTorch [13] and named-axes in JAX [4], we give explicit names to the axes of a tensor and call them *named-axes*. We can treat the named-axes of a tensor as an unordered set for layout-insensitive operators and express tensor shapes as mappings from names to sizes. For example, we may give the names h and v to the horizontal and vertical axes respectively of a 2×3 tensor t and the shape of this tensor would be the mapping $\{h \mapsto 3, v \mapsto 2\}$. Such a tensor can be accessed with an *access map*, which is a mapping from named-axes to indices. We then have $t[\{h \mapsto 1, v \mapsto 1\}] = w$, given the domain of the access map is exactly the set of the named-axes, there's no out-of-bounds access, and w is the value at that access.

τ	$:=$	Int | Bool | Real	Type
a	$\in$	$\mathcal{A}$	Named-axes
x	$\in$	$\mathcal{X} = \mathcal{P}(\mathcal{A})$	Aggregated-axes
f	$\in$	list[Int] $\to$ Int	Map function
m	$:=$	$\mathcal{M}$ | fmap$(f, m+)$	Maps
X	$\in$	$\mathcal{P}(\mathcal{X})$	Set of aggregated-axes
S, I	$\in$	$m^{\mathcal{X}}$	Shapes and indices
R	$\in$	$\mathcal{X}^{\mathcal{X}}$	Relabel maps
v	$:=$	i : Int | b : Bool | r : Real	Scalar literal
e	$:=$	$\mathcal{T}$ (Literal)	Tensor expression
	|	$\mathcal{V}$ (Variable)	
	|	const(v, S)	
	|	iota(S, x)	
	|	expand(e, S)	
	|	binary$(\oplus, e_l, e_r)$	
	|	pad(e, v, S_l, S_h, S_i)	
	|	slice(e, I_s, I_e, I_p)	
	|	dy-slice(e, I, S)	
	|	dyup-slice(e, e_u, I)	
	|	reduce$(\oplus, e, X)$	
	|	relabel(e, R)	
	|	concat(e_l, e_h, x)	
g	$\in$	list[Int] $\to$ Bool	Predicate function
P	$:=$	fold$(g, m+)$	Precondition
$Rule$	$:=$	$e_{lhs} \Rightarrow_{P*} e_{rhs}$	Rewrite rule

Fig. 6. Core rewrite rule representation with selected operators.

An operator that works on multiple tensors will need to match them by the named-axes, as shown in the following examples.

- binary$(+, t_1, t_2)$, where t_1, t_2 have the shapes $\{a_1 \mapsto 2, a_2 \mapsto 3\}$ and $\{a_1 \mapsto 2, a_2 \mapsto 3\}$, respectively. The two shapes match as they have the same set of named-axes and the corresponding named-axes have the same sizes.
- dot$(t_1, t_2, \{a_2, a_3\}, \{a_4\})$, where the shapes of t_1 and t_2 are $\{a_1 \mapsto 2, a_2 \mapsto 3, a_3 \mapsto 4, a_4 \mapsto 5\}$ and $\{a_5 \mapsto 6, a_2 \mapsto 3, a_3 \mapsto 4, a_4 \mapsto 5\}$, respectively. For the dot operator, we need to match the contraction and batch axes, in this case $\{a_2, a_3\}$ and $\{a_4\}$, respectively. The resulting tensor has the shape $\{a_1 \mapsto 2, a_4 \mapsto 5, a_5 \mapsto 6\}$.

Note that sometimes, we need to rename the axes to avoid name clashes. For example, with the dot operator, if the two tensors share some *spatial* named-axes (neither contraction nor batch axes), we need to rename them before applying the operator to make sure that the named-axes in the resulting tensor are unique.

4.2 Aggregated Axes

As we've shown in the dot example in §4.1, we have matched some *set* of axes between the two tensors. This partitions the named-axes in a tensor into sets of axes that have the same "role" in the expression. For example, a_2 and a_3 have the same roles as contraction axes. Based on this observation, we introduce *aggregated-axes*, which is a set of named-axes.

We can partition the set of named-axes of a tensor into *disjoint* aggregated-axes and write expressions directly using *uninterpreted* aggregated-axes. The aggregated-axes can be instantiated to some concrete set of named-axes, and the number of instantiated named-axes is called the *rank* of an aggregated-axis. This allows us to write expressions with an arbitrary number of named-axes in a uniform and simple way.

For example, in the expression $\text{dot}(t_1, t_2, \{x_2, x_3\}, \{x_4\})$, if we assume that the set of named-axes in t_1 and t_2 are $x_1 \cup x_2 \cup x_3 \cup x_4$ and $x_5 \cup x_2 \cup x_3 \cup x_4$, respectively, then the resulting tensor has $\{x_1, x_4, x_5\}$ as the set of aggregated-axes. It's easy to see that we can get back the dot example shown in §4.1 by instantiating all aggregated-axes to singleton sets. This instantiation is not arbitrary, and we elaborate on how to specify the constraints on the instantiations with *rank classes* in §6.

We can then lift the tensor semantics to aggregated semantics: shapes or indices can be expressed with, or instantiated from aggregated-axes. Instead of being a mapping from named-axes to integers, we now need a *nested mapping* that maps aggregated-axes to another map from names in the aggregated-axes to integers. As a convention, we will refer to the inner mappings as a *map* and the outer mapping as an *aggregated-map*. For example, the following is valid aggregated-map:

$$\{\{i_1, i_2\} \mapsto \{i_1 \mapsto 2, i_2 \mapsto 3\}, \{i_3\} \mapsto \{i_3 \mapsto 4\}\}$$

Definition 1. An *aggregated-map* M is valid if it is a nested mapping from aggregated-axes to maps from named-axes to integers such that:

- $\forall x_1, x_2 \in \text{dom}(M), x_1 \neq x_2 \rightarrow x_1 \cap x_2 = \varnothing$, and
- $\forall x \in \text{dom}(M), \text{dom}(M[x]) = x$.

Note that $M[x]$ represents the value mapped to x in M. Shape and indices are aliases for aggregated-maps in specific contexts and they have their additional validity conditions, depending on the context. Here, we give the validity conditions for tensor shapes and access indices:

Definition 2. A valid *tensor shape* S is a valid *aggregated-map*, such that $\forall x \in \text{dom}(S), \forall a \in x, S[x][a] \geq 0$. The shape of a tensor t is denoted as $\text{Shape}(t)$.

Definition 3. A valid *access* A (used for accessing tensors) with respect to a valid tensor shape S, is a valid *aggregated-map* such that

- $\text{dom}(A) = \text{dom}(S)$, and
- $\forall x \in \text{dom}(A), \forall a \in x, 0 \leq A[x][a] < S[x][a]$.

The set of all valid *accesses* given a tensor shape S, is denoted by $\text{Access}(S)$. A tensor t is then viewed as a mapping from the set $\text{Access}(\text{Shape}(t))$ to elements, and the element at the access A is denoted as $t[A]$. The set of all aggregated-axes of a tensor t is denoted as $\text{Axes}(t)$.

Operator attributes are also expressed using aggregated-maps, with each operator having its own validity conditions. For example, the pseudo operator pad-low, which only does low-padding, can pad a tensor with l_1, followed by l_2, for the axes in x_1 using the expression: $\text{pad-low}(\text{pad-low}(t, \{x_1 \mapsto l_1\}), \{x_1 \mapsto l_2\})$. The validity condition for pad-low allows padding with negative shapes but disallows creating a tensor with a negative shape (more details in §5). Assuming t has the shape s_0 in the aggregated-axis x_1, the resulting shape in the pad-low expression will be $s_0 + l_1 + l_2$. Note that we are doing an element-wise combination of maps, where the maps must have the same named-axes

and the resulting map contains the sum of the corresponding axis sizes. Element-wise combination (fmap) is the only allowed operation on the maps to combine them, as we define aggregated-axes as set of named axes that have the same "role" in the expression. Note that we may also combine a map with scalars by lifting the scalar to a constant map.

4.3 Rewrite Rule

Similar to many other tensor graph rewrite systems, TensorRight models rewrite rules as rewriting an LHS expression to an RHS expression, subject to certain preconditions. See the following example:

$$\text{pad-low}(\text{pad-low}(t, 0, \{x_1 \mapsto l_1\}), 0, \{x_1 \mapsto l_2\}) \Rightarrow_{l_1 \geq 0 \,\wedge\, l_2 \geq 0} \text{pad-low}(t, 0, \{x_1 \mapsto l_1 + l_2\})$$

In this rule, we aggregated all the named-axes in the tensor t into the aggregated-axis x_1. l_1 and l_2 are two maps from named-axes in x_1 to padding sizes. In the RHS, the two maps are combined in an element-wise way. Similarly, our preconditions are predicates lifted to operate on the maps in an element-wise way. The condition $l_1 \geq 0$ here means that the padding sizes in l_1 must be greater than or equal to 0 for all named-axes.

5 Denotational Semantics

We give the denotational semantics of the XLA-HLO operators in Fig. 7. We will use denotational semantics notations for deriving a bound on the ranks, but note that since our semantics map from our language to computable tensor objects, we can easily derive a big-step operational semantics and perform symbolic evaluation. Due to space limitations, we will only show the semantics of some selected operators. More operators are available in Appendix A. They are usually simple or can be expressed using existing operators or the techniques introduced here.

The domain of our denotational semantics are tensors, which map accesses to elements. The elements can be boolean, integers, or real numbers. There is also a special type of element called a *Reduction Element*, denoted as $\text{Red}^{\oplus}_{I_0,I_1,\ldots} f(\{\text{dom}(I_0) \mapsto I_0, \text{dom}(I_1) \mapsto I_1, \cdots\})$. Here, $\oplus$ is a binary operator and $I_0, I_1, \cdots$ are called *reduction indices*. We may sometimes omit the indices and write $\text{Red}^{\oplus}_{X} f(X)$, where $X = \{\text{dom}(I_0), \text{dom}(I_1), \cdots\}$ is the set of aggregated-axes being reduced.

The introduction of a *reduction element* is based on pragmatic reasons. As the sizes of reduced axes are unbounded, we cannot expand the reduction to sum all the values being reduced. Thus, we leave the sum uninterpreted and provide special treatment for such elements during verification.

Fig. 7 shows the denotational semantics of some selected operators. We overload some functions for convenience: $\text{Shape}(e)$ means $\text{Shape}(\llbracket e \rrbracket)$, $\text{Access}(e)$ means $\text{Access}(\text{Shape}(e))$, and $\text{Axes}(e)$ means $\text{Axes}(\llbracket e \rrbracket)$. We also introduce some helper functions on valid aggregated-maps. Given a comparison operator $\odot$ and a binary operator $\oplus$, we lift them to aggregated-maps as

$$M_1 \odot M_2 = \bigwedge_{x \in \text{dom}(M_1)} \bigwedge_{a \in x} M_1[x][a] \odot M_2[x][a]$$

$$M_1 \oplus M_2 = \{x \mapsto \{a \mapsto M_1[x][a] \oplus M_2[x][a] \mid a \in M_1[x]\} \mid x \in \text{dom}(M_1)\}$$

All these binary operations implicitly introduce the assumption that the two aggregated-maps are valid and the domain of the two aggregated-maps are the same. We will omit these from our rules. Sometimes, we may overload the notations to operate with constants. This is treated as operating with a nested map where the inner map are constant maps.

In our rules, we introduce bindings with let $name = \cdots$ notation. Some rules like Iota and Concat require that an aggregated-axis is singleton. In these rules, we use the syntax let $\{a\} = x$ to say that x is singleton and bind the singleton element in x to a. We use the notation $S|_K$ to denote the mapping restriction of S to K, where S is any map and K is a subset of keys from S.

$$\frac{}{[\![\text{const}(v,S)]\!] = \{A \mapsto v \mid A \in \text{Access}(S)\}}\ \textsc{Const}$$

$$\frac{\text{let } \{a\} = x \qquad x \in \text{dom}(S)}{[\![\text{iota}(S,x)]\!] = \{A \mapsto A[x][a] \mid A \in \text{Access}(S)\}}\ \textsc{Iota}$$

$$\frac{\text{dom}(S) \cap \text{Axes}(e) = \varnothing}{[\![\text{expand}(e,S)]\!] = \{A \mapsto [\![e]\!]\,[A|_{\text{Axes}(e)}] \mid A \in \text{Access}(\text{Shape}(e) \cup S)\}}\ \textsc{Expand}$$

$$\frac{\text{Shape}(e) = \text{Shape}(e')}{[\![\text{binary}(\oplus,e,e')]\!] = \{A \mapsto [\![e]\!]\,[A] \oplus [\![e']\!]\,[A] \mid A \in \text{Access}(e)\}}\ \textsc{BinOp}$$

$$\frac{\text{let } S = \text{Shape}(e) \qquad \text{let } S' = S + S_l \geq 0 \qquad \text{let } \textit{not-pad} = \lambda A.A \geq S_l}{[\![\text{pad-low}(e,v,S_l)]\!] = \{A \mapsto \text{if } \textit{not-pad}(A) \text{ then } [\![e]\!]\,[A - S_l] \text{ else } v \mid A \in \text{Access}(S')\}}\ \textsc{PadLow}$$

$$\frac{0 \leq I_s \leq I_e \leq \text{Shape}(e) \qquad I_p > 0}{[\![\text{slice}(e,I_s,I_e,I_p)]\!] = \{A \mapsto [\![e]\!]\,[I_s + A \times I_p] \mid A \in \text{Access}(\left\lceil \frac{I_e - I_s}{I_p} \right\rceil)\}}\ \textsc{Slice}$$

$$\frac{I + S \leq \text{Shape}(e) \qquad S > 0 \qquad I \geq 0}{\text{dy-slice}(e,I,S) = \{A \mapsto [\![e]\!]\,[A + I] \mid A \in \text{Access}(S)\}}\ \textsc{DySlice}$$

$$\frac{\begin{array}{c}\text{let } S_u = \text{Shape}(e_u) \qquad I + S_u \leq \text{Shape}(e) \qquad S_u > 0 \qquad I \geq 0 \\ \text{let } acc = \lambda A.\text{if } A \geq I \wedge A < I + S_u \text{ then } [\![e_u]\!]\,[A - I] \text{ else } [\![e]\!]\,[A]\end{array}}{\text{dyup-slice}(e,e_u,I) = \{A \mapsto acc(A) \mid A \in \text{Access}(e)\}}\ \textsc{DyUpdateSlice}$$

$$\frac{\begin{array}{c}\text{let } S = \text{Shape}(e) \qquad \text{let } \{x_0 \cdots x_k\} = X \subseteq \text{Axes}(e) \\ \text{let } acc = \lambda A.\text{Red}^{\oplus}_{I_0,\cdots,I_k}\,[\![e]\!]\,[\{x_0 \mapsto I_0, \cdots, x_k \mapsto I_k\} \cup A]\end{array}}{[\![\text{reduce}(\oplus,e,X)]\!] = \{A \mapsto acc(A) \mid A \in \text{Access}(S \setminus S|_X)\}}\ \textsc{Reduce}$$

$$\frac{\forall x_1, x_2 \in \text{dom}(R), x_1 \neq x_2 \rightarrow R[x_1] \neq R[x_2] \qquad \text{dom}(R) = \text{Axes}(e)}{[\![\text{relabel}(e,R)]\!] = \{A \mapsto [\![e]\!]\,[A \circ R] \mid A \in \text{Access}(S \circ R^{-1})\}}\ \textsc{Relabel}$$

$$\frac{\begin{array}{c}\text{let } \{a\} = x \qquad x \in \text{Axes}(e) \qquad x \in \text{Axes}(e') \qquad \text{let } S = \text{Shape}(e) \qquad \text{let } S' = \text{Shape}(e') \\ \forall x' \in \text{Axes}(e), x' \neq x \rightarrow S[x'] = S'[x'] \\ \text{let } S'' = \{x' \mapsto \text{if } x' = x \text{ then } S'[x] \text{ else } \{a' \mapsto 0 \mid a' \in x'\} \mid x' \in \text{Axes}(e)\} \\ \text{let } acc = \lambda A.\text{if } A \geq S'' \text{ then } [\![e']\!]\,[A - S''] \text{ else } [\![e]\!]\,[A]\end{array}}{\text{concat}(e,e',x) = \{A \mapsto acc(A) \mid A \in \text{Access}(S + S'')\}}\ \textsc{Concat}$$

Fig. 7. Denotational Semantics of some core operators.

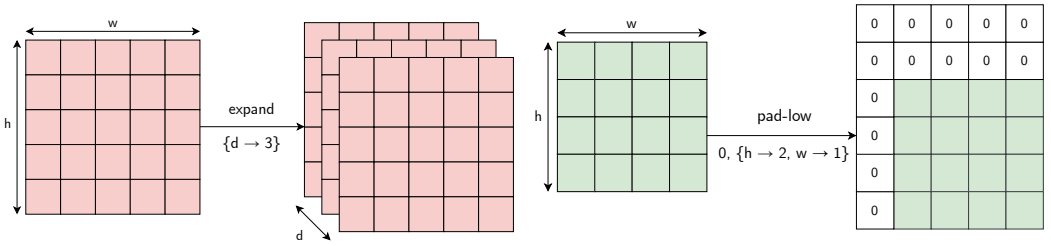
Tensor operators. Next, we explain the semantics of some select operators:

- Const: The const operator outputs a tensor with the desired shape S, with all accesses being mapped to the same constant element v.
- Iota: The iota operator projects the access-index for a specific singleton aggregated-axis x. The resulting tensor holds values starting at 0, incrementing by 1 along that axis.
- Expand: The expand operator introduces new aggregated-axes to a tensor by duplicating the data in the tensor. It takes a shape containing the sizes of the new axes and the resulting tensor is accessed as if we are accessing the input tensor after removing these new axes from the access. The set of new axes must be disjoint from the original set of axes.
  Fig. 8a demonstrates the expand operator with an example. The input tensor contains two aggregated-axes x_h and x_w, each instantiated with 1 named-axes. We refer to the corresponding named-axes by h and w, respectively. The input shape is $\{\{h\} \mapsto \{h \mapsto 5\}, \{w\} \mapsto \{w \mapsto 5\}\}$. We expand it by adding a new aggregated-axis x_d, containing 1 named-axis

d. The resulting tensor duplicates data across this new named-axis and has a shape of $\{\{h\} \mapsto \{h \mapsto 5\}, \{w\} \mapsto \{w \mapsto 5\}, \{d\} \mapsto \{d \mapsto 3\}\}$.

- BinOp: The binary operator performs an element-wise operation on two identically-shaped tensors.
- PadLow: We present a restricted version of the pad operator, pad-low, which pads only on the low-ends of each axis. The semantics test whether the access is in the padded region. If so, return the padded value, or access the original tensor, offset by the padding shape.
  Fig. 8b demonstrates the pad-low operator with an example. The input tensor contains two aggregated-axes x_h and x_w, each instantiated with 1 named-axes. We refer to the corresponding named-axes by h and w, respectively. The input shape is $\{\{h\} \mapsto \{h \mapsto 4\}, \{w\} \mapsto \{w \mapsto 4\}\}$. We perform zero-padding on h and w with padding attributes of 2 and 1 respectively. The zero-padding values are all present at the lower-ends of the axes. The resulting tensor a shape of $\{\{h\} \mapsto \{h \mapsto 6\}, \{w\} \mapsto \{w \mapsto 5\}\}$.
- Slice: The slice operator extracts a sub-tensor, which has the same named-axes as the input tensor and contains the values inside a bounding box within the input. The indices for the bounding box are given by the starting indices I_s, limit indices I_e (exclusive), and the positive strides I_p. The slice picks every $I_p[x][a]$ element along each named-axis $a \in x \in \mathsf{dom}(I_p)$.
- DySlice: The dy-slice operator extracts a sub-tensor from the input tensor. The indices for the bounding box are given by the starting indices I and size of the bounding box S. The sizes must be positive and should not cause out-of-bounds accesses. Note that this is different from the XLA semantics, where our starting indices are not represented as a tensor, but as a map. We are then only able to express rewriting rules where the indices are used in an opaque way, or as a constant, or computed with element-wise operations, e.g., binary. We found that this change does not affect the effectiveness of TensorRight for verification purposes, and our approximation is able to express all rewrite rules involving dy-slice.
- DyUpdateSlice: The dyup-slice operator generates a result with a slice overwritten by e_u, starting at indices I. Our dyup-slice operator follows the same approximation as dy-slice.
- Reduce: The reduce operator takes a tensor and a set of aggregated-axes X as inputs, then returns a tensor mapping to uninterpreted reduction elements as the result. The resulting tensor has the shape $S \setminus S|_X$, essentially removing all the aggregated-axes in X. We extend the semantics of reduce to make verification easier in Appendix A.
- Relabel: In TensorRight, as we take an unordered view of the axes, we no longer need transpose. However, we still need to re-match the axes, for example, when we want to describe some expressions such as $t + \mathsf{transpose}(t)$. The relabel operator is introduced for this axes-matching operation. It renames aggregated-axes and does not change tensor contents.
- Concat: The concat operator is another example where we introduce the singleton constraint on an aggregated-axis. The two tensors should have the same shape on other axes. For the concatenating axis, the resulting size will be the sum of the operand sizes. The resulting tensor will then compute which operand tensor the access belongs to and perform the access.

Handling reduction elements. In real-world XLA rewrite rules, the reduction might not always be the top-level operation and a reduction of a tensor may be performed in several steps. For instance, in the rewrite rule $\mathsf{reduce}(\mathsf{concat}(\mathsf{reduce}(\mathsf{A}), \mathsf{reduce}(\mathsf{B}))) = \mathsf{reduce}(\mathsf{concat}(\mathsf{A}, \mathsf{B}))$, the reduction in LHS is done in two steps. To handle rules like this, we provide a *limited* set of arithmetic rules for reduction elements in Fig. 9. If no known rule applies, TensorRight will report an error, indicating that the rule isn't supported. The equivalence of two reduction elements is currently verified by a stronger condition, where we view them as sets and verify that there is a one-one mapping between them. Note that not all correct rules meet the strengthened condition, but we



(a) expand: $\{d \longrightarrow 3\}$ denotes the expansion shape

(b) pad-low: $\{h \mapsto 2, w \mapsto 1\}$ denotes the padding attributes and 0 denotes the padding value

Fig. 8. Illustration of (a) expand and (b) pad-low operators. h, w, and d denote named-axes.

find that it covers most rewrite rules with reductions. Additionally, user-provided hints are needed for verifying the set equivalence, as discussed in §6.5. This is the only manual hint from the user during our verification. Providing better mechanisms for handling reductions is a future work.

$$v * \mathsf{Red}^{+}_{X} f(X) \longrightarrow \mathsf{Red}^{+}_{X} v * f(X) \qquad \mathsf{Red}^{+}_{X} f(X) * \mathsf{Red}^{+}_{Y} g(Y) \longrightarrow \mathsf{Red}^{+}_{X,Y} f(X) * g(Y)$$
$$\mathsf{Red}^{\oplus}_{X}(\mathsf{Red}^{\oplus}_{Y} f(X,Y)) \longrightarrow \mathsf{Red}^{\oplus}_{X,Y} f(X,Y)$$

Fig. 9. Rules on reduction elements.

6 Verification of Rewrite Rules

After defining the representation of rewrite rules in §4 and its denotational semantics in §5, this section describes how TensorRight verifies the rewrite rules given the semantics. We will first overview our verification approach, which is based on k-induction [36], then provide proof sketches for our induction steps.

6.1 Overview of the Verification

To prove that a rewrite rule is correct, intuitively, we need to verify that the two expressions have the same denotation, possibly under some assumptions. Given a rule $\mathsf{LHS} \Rightarrow_C \mathsf{RHS}$, we prove that

$$\forall v \in vars, C \wedge \text{valid-expr}(\mathsf{LHS}) \longrightarrow [\![\mathsf{LHS}]\!] = [\![\mathsf{RHS}]\!] \tag{3}$$

Here, *vars* is the set of all variables appearing in the rule. It contains all possible tensor variables and operator attributes, such as slice attributes and expand shapes. Note that we only consider the case where the term prior to rewriting is valid, i.e., when LHS is valid.

The challenge here is that a rewrite rule can usually be applied to tensors with arbitrary number of axes and arbitrary sizes in each axis. To handle arbitrary number of axes, as discussed in §4, we express rewrite rules with a finite number of aggregated-axes and each of them may be instantiated to arbitrary ranks. The equivalence of the two expressions then boils down to verifying that they are equivalent under all *valid* instantiations I, where the LHS is valid:

$$\bigwedge_{I} \text{valid}(I) \quad \text{where} \quad \text{valid}(I) = \forall v \in vars(I), C \wedge \text{valid-expr}(\mathsf{LHS}(I)) \longrightarrow [\![\mathsf{LHS}(I)]\!] = [\![\mathsf{RHS}(I)]\!]$$

For a given instantiation, we now have concrete ranks but *unbounded sizes*. To handle arbitrary sizes, we model each tensor as an uninterpreted function and use unbounded integers to model the indices and sizes in each named-axis. This is supported natively by SMT solvers with good performance. We then leverage a symbolic execution approach to convert the rewrite rule into a set of constraints. We call this a *bounded-verification* proof obligation, which is used to verify a *given instantiation* of our rewrite rule. This is described in detail in §6.4.

We still need to answer two questions: (1) what does it mean for an instantiation to be valid and (2) how to verify the rule for a possibly infinite number of instantiations? Our approach to these questions is to introduce a new concept called *rank class* (RClass). A *rank class* is a (required) property of an aggregated-axis, such that all the aggregated-axes with the same RClass are always instantiated to the same rank. An instantiation of a rewrite rule can then be expressed as instantiating these RClasses to some rank. We then derive a *sufficient* rank for each RClass, such that any instantiation with higher ranks could be proven inductively, given that we have verified all instances within the rank. This approach follows the k-induction technique [36].

To show the intuition for an RClass, consider the rewrite rule $(A + A^\top)^\top = A + A^\top$, where t has the shape $\{x_1 \mapsto m_1, x_2 \mapsto m_2\}$. Note that transpose is a no-op with unordered axes semantics, and relabel is provided for renaming and matching axes.

$$\mathsf{relabel}(\mathsf{binary}(+, t, \mathsf{relabel}(t, \{x_1 \mapsto x_2, x_2 \mapsto x_1\})), \{x_1 \mapsto x_2, x_2 \mapsto x_1\}) \Rightarrow \\ \mathsf{binary}(+, t, \mathsf{relabel}(t, \{x_1 \mapsto x_2, x_2 \mapsto x_1\}))$$

It is easy to see that we must instantiate x_1, x_2 with the same number of axes to make sure that both LHS and RHS are valid. In other words, x_1 and x_2 must have the same rank, in which case we say that these two aggregated axes are in the same *rank class* (RClass) and the RClass constraints the possible instantiations. When we instantiate a rule with relabeling, we also need to be able to establish a consistent mapping between the named-axes of x_1 and x_2. These facts are expressed in the following two definitions.

Definition 4. A *rank class (RClass)* c is a property of a family of aggregated-axes, such that

- Each aggregated-axis x is in exactly one RClass c, written as $x : c$.
- For all $x_0 : c$, $x_1 : c$, x_0 and x_1 are instantiated to the same rank r in a valid instantiation. In such an instantiation, the rank of the RClass c is defined to be r.

Definition 5. An RClass c provides a canonical bijection mapping between each pair of the aggregated axes associated with it. We denote such a mapping that maps from the named axes in $x_0 : c$ to $x_1 : c$, as $\mathsf{MapAxes}(c, x_0, x_1)$. The canonical mapping must satisfy:

- $\mathsf{MapAxes}(c, x_0, x_1) \circ \mathsf{MapAxes}(c, x_1, x_0) = \mathsf{id}$, and
- $\mathsf{MapAxes}(c, x_1, x_2) \circ \mathsf{MapAxes}(c, x_0, x_1) = \mathsf{MapAxes}(d, x_0, x_2)$.

where $\circ$ refers to function composition. Such a mapping can be trivially constructed during the instantiation as the aggregated axes of the same rank class are instantiated to the same rank. In practice, without loss of generality, to instantiate an aggregated axis $x_i : c$ to rank r, we can instantiate it to the set of axes $a_{i,0}, \cdots, a_{i,r-1}$. The canonical mapping between two aggregated axes $x_i : c$ and $x_j : c$ can then be established as $\{a_{i,k} \mapsto a_{j,k} \mid k \in \{0 \cdots r-1\}\}$, and the semantics of relabel follows this mapping to relabel the instantiated axes.

We can then reduce our problem to verifying the rule for all possible RClass instantiations:

$$\bigwedge_{r_1, r_2, \cdots} \mathsf{valid}(\overline{\{c_i \mapsto r_i\}}) \tag{4}$$

where $\overline{\{c_i \mapsto r_i\}}$ is a map containing the ranks for all RClasses $c_1, c_2, \cdots$ in the rule. $\mathsf{valid}(\overline{\{c_i \mapsto r_i\}})$ denotes the proof obligation for a concrete-ranked instance, where the RClass c_i is instantiated to rank r_i, for all $i \in \{1 \cdots p\}$. We still need to verify the rule for *all possible* ranks of all RClasses.

Our Approach: To simplify the discussion, let's assume that the rule only has one aggregated-axis x, with RClass c (i.e., $x : c$). We can then rewrite Equation 4 as

$$\bigwedge_{i} \mathsf{valid}(i) \tag{5}$$

where valid(i) (short for valid($\{c \mapsto i\}$)) is true if and only if the rule is valid when rank of c (and x) is i. We observe that for every RClass in a rule, there exists a *bound* corresponding to a sufficient rank required for unbounded verification, i.e., there exists a rank k, such that,

$$\forall i \geq k,\ \mathsf{valid}(i) \rightarrow \mathsf{valid}(i+1) \tag{6}$$

This means that for all $i \geq k$, if the rule is valid when c has rank i, then the rule is valid when c has rank $i+1$. Given such a k, we can do unbounded verification for the rule using k-induction:

- Basis: Use bounded verification to prove that the rule is valid for all ranks until k:

$$\bigwedge_{i=1\cdots k} \mathsf{valid}(i)$$

- Induction case: Use induction on the rank of c, with Equation 6 as induction hypothesis:

$$\mathsf{valid}(k) \wedge [\forall i \geq k,\ \mathsf{valid}(i) \rightarrow \mathsf{valid}(i+1)] \ \Rightarrow \bigwedge_{i \geq k} \mathsf{valid}(i)$$

This would imply that the rule is correct for an arbitrary number of named-axes in c. Now what remains is finding a sufficient rank k for any RClass in a rule. We show how to derive such a bound with the help of an example rewrite rule.

6.2 Bound Computation Example

Consider an input tensor Y which has one aggregated-axis, say x, having the RClass c (i.e. $x : c$). The rewrite rule PadLowCombine is shown below:

$$\mathsf{pad\text{-}low}(\mathsf{pad\text{-}low}(\mathsf{Y}, 0, L_1), 0, L_2) \Rightarrow_{L_1 \geq 0\ \wedge\ L_2 \geq 0} \mathsf{pad\text{-}low}(\mathsf{Y}, 0, L_1 + L_2) \tag{7}$$

where $L_1 = \{x \mapsto l_1\}$ and $L_2 = \{x \mapsto l_2\}$, for some maps l_1 and l_2.

The PadLowCombine rule merges two pad-low operators into a single pad-low operator. The precondition requires that both padding attributes should be non-negative. For this rule, the precondition also implies that the LHS expression is valid. The variables appearing in this rule are the padding attributes L_1, L_2 and the input tensor Y. Using Equation 3, we can express the validity condition for this rewrite rule as

$$\forall\, \mathsf{Y}, L_1, L_2,\ L_1 \geq 0 \wedge L_2 \geq 0 \rightarrow [\![\mathsf{LHS}]\!] = [\![\mathsf{RHS}]\!] \tag{8}$$

It is easy to see that both LHS and RHS have the same shape, i.e., they have the same domain of accesses. Thus, we can rewrite $[\![\mathsf{LHS}]\!] = [\![\mathsf{RHS}]\!]$ by interpreting the tensors under a general, valid access to these tensors. Let $A \in \mathsf{Access}(\mathsf{LHS})$ be an arbitrary access from the domain of the output tensors. It has the form $A = \{x \mapsto a\}$, where a maps named-axes in x to (symbolic) indices. We use the denotational semantics of pad-low described in §5, to symbolically execute these expressions.

$$\begin{aligned}
&[\![\mathsf{LHS}]\!] = [\![\mathsf{RHS}]\!] \\
\Leftrightarrow\ &\forall A,\ [\![\mathsf{LHS}]\!]\,[A] = [\![\mathsf{RHS}]\!]\,[A] \\
\Leftrightarrow\ &\forall A,\ [\![\mathsf{pad\text{-}low}(\mathsf{pad\text{-}low}(\mathsf{Y}, 0, L_1), 0, L_2)]\!]\,[A] = [\![\mathsf{pad\text{-}low}(\mathsf{Y}, 0, L_1 + L_2)]\!]\,[A] \\
\Leftrightarrow\ &\forall A,\ \text{if } A \geq L_2 \text{ then } (\text{if } A - L_2 \geq L_1 \text{ then } \mathsf{Y}[A - L_1 - L_2] \text{ else } 0) \text{ else } 0 = \\
&\qquad\qquad \text{if } A \geq L_1 + L_2 \text{ then } \mathsf{Y}[A - L_1 - L_2] \text{ else } 0 \\
\Leftrightarrow\ &\forall a,\ \text{if } a \geq l_2 \text{ then } (\text{if } a - l_2 \geq l_1 \text{ then } \mathsf{Y}[x \mapsto a - l_1 - l_2] \text{ else } 0) \text{ else } 0 = \\
&\qquad\qquad \text{if } a \geq l_1 + l_2 \text{ then } \mathsf{Y}[x \mapsto a - l_1 - l_2] \text{ else } 0
\end{aligned} \tag{9}$$

In the last step, we made use of the fact that the aggregated maps like A, L_1, L_2 contain only one aggregated-axis x. Equation 9 holds for any number of named-axes in x.

Observation. We can syntactically partition the above equation as follows:

$$\forall a,\ \text{if } a \geq l_2 \text{ then } (\text{if } a - l_2 \geq l_1 \text{ then } \mathsf{Y}[x \mapsto a - l_1 - l_2] \text{ else } 0) \text{ else } 0 = \\ \text{if } a \geq l_1 + l_2 \text{ then } \mathsf{Y}[x \mapsto a - l_1 - l_2] \text{ else } 0$$

We explain each part below:

$\mathsf{Y}[\_]$: represents accesses to the tensor Y.

$x \mapsto a - l_1 - l_2$: represents an *access expression* for a tensor access. In this case, $a - l_1 - l_2$ is the access map for the aggregated-axis x. The rank of this access map depends on the number of named-axes in x. We observe that we can rewrite this expression as follows:

$$a - l_1 - l_2 = \mathsf{fmap}(e, a, l_1, l_2) \quad \text{where} \quad e \stackrel{\text{df}}{=} \lambda v, p, p'.(v - p - p')$$

where fmap takes a function and applies it to a list of maps. For instance, if $m = \{i \mapsto v_i, j \mapsto v_j\}$ and $f = \lambda v.(v + 1)$, then $\mathsf{fmap}(f, m) = \{i \mapsto v_i + 1, j \mapsto v_j + 1\}$. Here, e is independent of the rank of x and only a, l_1, l_2 change as the rank of x changes. Thus, we are able to capture all the rank-independent information in the function e. We call such a function an *index transformer* because it transforms output index-values to input index-values.

$a \geq l_1 + l_2$: these are boolean values, referred to as *conditions*, occurring inside if-then-else blocks. They capture the dependency of the output tensor value on the input tensor values, based on the value of the access. They originate from the operator semantics. For instance, *not-pad* in the pad-low semantics takes an access A and tells if it lies in the padded area or not. We observe that we can rewrite this condition as follows:

$$a \geq l_1 + l_2 = \mathsf{fold}(g_1, a, l_1, l_2) \quad \text{where} \quad g_1 \stackrel{\text{df}}{=} \lambda v, p, p'.(v \geq p + p')$$

where fold takes a boolean valued function, applies it to a list of maps, and returns true if all values are true, false otherwise. fold can be defined as:

$$\mathsf{fold}(g, m_1, m_2, \cdots) = \bigwedge_{i \in \mathsf{dom}(m_1)} g(m_1(i), m_2(i), \cdots)$$

Similarly, we can write $a \geq l_2$ as $\mathsf{fold}(g_2, a, l_2)$, where $g_2 \stackrel{\text{df}}{=} \lambda v, p.(v \geq p)$. Here, g_1 and g_2 are independent of the rank of x and only a, l_1, l_2 change as the rank of x changes. We capture all the rank-independent information in the functions g_1 and g_2.

$\text{if } \_ \text{ then } (\text{if } \_ \text{ then } \_ \text{ else } 0) \text{ else } 0 = \text{if } \_ \text{ then } \_ \text{ else } 0$: this is a function which returns a boolean, denoting if the values of LHS and RHS at the access A are equal or not. We call this scalarf since it contains the core, *scalar* computation in the expressions and may consist of arithmetic and conditionals. For this rule, we can define scalarf as

$$\mathsf{scalarf}(y, b_1, b_2) \stackrel{\text{df}}{=} (\text{if } b_2 \text{ then } (\text{if } b_1 \text{ then } y \text{ else } 0) \text{ else } 0) = (\text{if } b_1 \text{ then } y \text{ else } 0)$$

$$y \stackrel{\text{df}}{=} \mathsf{Y}[x \mapsto \mathsf{fmap}(e, a, l_1, l_2)] \quad b_1 \stackrel{\text{df}}{=} \mathsf{fold}(g_1, a, l_1, l_2) \quad b_2 \stackrel{\text{df}}{=} \mathsf{fold}(g_2, a, l_2)$$

Based on the arguments to scalarf, we say scalarf has *1 access to Y* and *2 conditions*. Note that there are 2 occurrences each of y and b_1 in the scalarf but we only care about distinct accesses and conditions. Just like index transformers, scalarf is also independent of the rank of x. This property will be crucial for our bound computation algorithm. We use this observation to write Equation 9 in terms of scalarf and substitute it back in Equation 8 to get the validity condition of the rule:

$$\forall\, \mathsf{Y}, l_1, l_2,\ l_1 \geq 0 \wedge l_2 \geq 0 \rightarrow \\ \forall a,\ \mathsf{scalarf}(\mathsf{Y}[x \mapsto \mathsf{fmap}(e, a, l_1, l_2)],\ \mathsf{fold}(g_1, a, l_1, l_2),\ \mathsf{fold}(g_2, a, l_2)) \tag{10}$$

Bound Computation. We first look at the validity condition of the rule when x is instantiated with some rank i, i.e., $\mathsf{valid}(i)$. We instantiate the aggregated-axis x to x^i, which contains i named-axes, say $\{\underline{1}, \cdots, \underline{i}\}$. We also instantiate the input tensor, padding attributes, and the general access map. Thus, we can rewrite $\mathsf{valid}(i)$ as

$$\begin{aligned}&\forall\ \mathsf{Y}^i, l_1^i, l_2^i,\ l_1^i \geq 0 \wedge l_2^i \geq 0 \rightarrow \\ &\quad \forall a^i,\ \mathsf{scalarf}(\mathsf{Y}^i[x^i \mapsto \mathsf{fmap}(e, a^i, l_1^i, l_2^i)],\ \mathsf{fold}(g_1, a^i, l_1^i, l_2^i),\ \mathsf{fold}(g_2, a^i, l_2^i))\end{aligned} \tag{11}$$

As noted before, scalarf, e, g_1, and g_2 are independent of the rank of x, so they remain unchanged irrespective of the value of i. We want to find a k such that Equation 6 holds. The idea is to start with $[\mathsf{valid}(k) \rightarrow \mathsf{valid}(k+1)]$ and try to find a k which satisfies the induction hypothesis. We instead work with its contrapositive,

$$\mathsf{valid}(k) \rightarrow \mathsf{valid}(k+1) \Leftrightarrow \neg\mathsf{valid}(k+1) \rightarrow \neg\mathsf{valid}(k)$$

Intuitively, $\neg\mathsf{valid}(i)$ is true if there is a counterexample for the rule at rank i. We want to find a sufficient k such that we can *lower* a counterexample at rank $k+1$ (and all higher ranks) to a counterexample at rank k. On expanding the validity conditions using Equation 11, we get:

$$\begin{aligned}&\exists\ \mathsf{Y}^{k+1}, l_1^{k+1}, l_2^{k+1},\ l_1^{k+1} \geq 0 \wedge l_2^{k+1} \geq 0 \bigwedge \exists a^{k+1}, \\ &\quad \neg\mathsf{scalarf}(\mathsf{Y}^{k+1}[x^{k+1} \mapsto \mathsf{fmap}(e, a^{k+1}, l_1^{k+1}, l_2^{k+1})],\ \mathsf{fold}(g_1, a^{k+1}, l_1^{k+1}, l_2^{k+1}),\ \mathsf{fold}(g_2, a^{k+1}, l_2^{k+1})) \\ &\qquad\qquad \downarrow \\ &\exists\ \mathsf{Y}^{k}, l_1^{k}, l_2^{k},\ l_1^{k} \geq 0 \wedge l_2^{k} \geq 0 \bigwedge \exists a^{k}, \\ &\quad \neg\mathsf{scalarf}(\mathsf{Y}^{k}[x^{k} \mapsto \mathsf{fmap}(e, a^{k}, l_1^{k}, l_2^{k})],\ \mathsf{fold}(g_1, a^{k}, l_1^{k}, l_2^{k}),\ \mathsf{fold}(g_2, a^{k}, l_2^{k}))\end{aligned}$$

This means:

- We are given a tensor Y^{k+1} which has $k+1$ named-axes in x^{k+1}, and whose shape is of the form $\mathsf{Shape}(\mathsf{Y}) = \{x^{k+1} \mapsto m\}$, where $m = \{\underline{1} \mapsto n_1,\ \cdots,\ \underline{k+1} \mapsto n_{k+1}\}$
- We are given padding attributes l_1^{k+1} and l_2^{k+1} such that the precondition is satisfied.
- We are given a map a^{k+1} such that output tensors do not match at the access $\{x^{k+1} \mapsto a^{k+1}\}$.
- We then need to construct a tensor Y^k which has k named-axes in x^k. We also need to construct padding attributes l_1^k and l_2^k such that the precondition is still satisfied, and a map a^k such that the output tensors do not match at the access $\{x^k \mapsto a^k\}$.

Counterexample Construction. Our counterexample construction algorithm involves *projecting* the $(k+1)$-ranked RClass to a k-ranked RClass, i.e., we would choose k named-axes from $\{\underline{1}, \cdots, \underline{k+1}\}$. There are $k+1$ such projections but all projections may not lead to a counterexample. We express our construction through a set of equations and derive constraints on the projection.

Let $\Gamma \subset \{\underline{1}, \cdots, \underline{k+1}\}$ be a projection of size k. The named-axes in Γ are currently unknown. We can then express the k-ranked attributes as follows: $\mathsf{Shape}(\mathsf{Y}^k) = \{x^k \mapsto m|_\Gamma\}$, $a^k = a^{k+1}|_\Gamma$, $l_1^k = l_1^{k+1}|_\Gamma$, and $l_2^k = l_2^{k+1}|_\Gamma$. These are unknown as well. To construct a k-ranked counterexample, we first make sure that the arguments to scalarf have the same values in both ranks. We equate scalarf arguments in the k-ranked counterexample to the corresponding arguments in the $(k+1)$-ranked counterexample and collect *constraints* on Γ. A constraint is a named-axis that needs to be in the projection for the k-ranked counterexample to exist. Thus,

- $\text{fold}(g_1, a^k, l_1^k, l_2^k)$ and $\text{fold}(g_1, a^{k+1}, l_1^{k+1}, l_2^{k+1})$ need to be equisatisfiable. Let C_1 be the set of constraints we get from this equation. We first expand the definition of fold,

$$\text{fold}(g_1, a^{k+1}, l_1^{k+1}, l_2^{k+1}) = \bigwedge_{i=1}^{k+1} a^{k+1}(\underline{i}) \geq l_1^{k+1}(\underline{i}) + l_2^{k+1}(\underline{i})$$
$$\text{fold}(g_1, a^k, l_1^k, l_2^k) = \bigwedge_{\underline{j} \in \Gamma} a^k(\underline{j}) \geq l_1^k(\underline{j}) + l_2^k(\underline{j})$$

  Let $b = \text{fold}(g_1, a^{k+1}, l_1^{k+1}, l_2^{k+1})$, which contains $k+1$ clauses, and $b' = \text{fold}(g_1, a^k, l_1^k, l_2^k)$, which contains k clauses. The value of b is known since it depends entirely on the $(k+1)$-ranked counterexample. Let r be the number of clauses in b which evaluate to true. The remaining $(k+1) - r$ clauses evaluate to false. We do a case analysis on r:
  - $r < k$: b is false for this case. We want b' to be false as well. We can see that for any projection Γ, b' will be false. There are no constraints in this case, so $C_1 = \emptyset$.
  - $r = k$: b is false for this case. We want b' to be false as well. There is exactly one named-axis, say $\underline{l}$, for which $a^{k+1}(\underline{l}) \geq l_1^{k+1}(\underline{l}) + l_2^{k+1}(\underline{l})$ is false. $\underline{l}$ needs to be in the projection for b' to be false. Leaving out $\underline{l}$ will make b' true, which is not desirable. For this case, $C_1 = \{\underline{l}\}$
  - $r = k+1$: b is true for this case, so we want b' to be true as well. We can see that for any projection Γ, b' will be true. There are no constraints in this case, so $C_1 = \emptyset$.

  As seen above, we get at most one constraint from this equation, so $|C_1| \leq 1$.
- $\text{fold}(g_2, a^k, l_2^k)$ and $\text{fold}(g_2, a^{k+1}, l_2^{k+1})$ need to be equisatisfiable. Let C_2 be the set of constraints we get from this equation. We do a similar analysis and get at most one constraint from this equation, so $|C_2| \leq 1$. The named-axes in C_2 may or may not be same as named-axes in C_1.
- $\mathsf{Y}^k[x^k \mapsto \text{fmap}(e, a^k, l_1^k, l_2^k)]$ needs to be set to $\mathsf{Y}^{k+1}[x^{k+1} \mapsto \text{fmap}(e, a^{k+1}, l_1^{k+1}, l_2^{k+1})]$. This does not introduce any constraint, irrespective of the projection. There could have been constraints introduced if the scalarf had more than 1 access to Y. We discuss more about the general case in §6.3.

The final set of constraints is computed as $C = C_1 \cup C_2$. If $|C| > k$, then we cannot get a valid projection. Thus, we need $|C| \leq k$ for a valid counterexample lowering. We know that $|C| = |C_1 \cup C_2| \leq |C_1| + |C_2| \leq 2$. From this, we get $2 \leq k$ as a sufficient condition for a valid counterexample lowering. Finally, we can fully construct the k-ranked counterexample as follows:

- Projection: The named-axes in C need to be a part of the projection Γ, but the other axes are unspecified. To get the final projection Γ, extend C by any $k - |C|$ named-axes from $\{\underline{1}, \cdots, \underline{k+1}\} \setminus C$.
- Tensor shapes and attributes: Compute the k-ranked attributes as follows: $\text{Shape}(\mathsf{Y}^k) = \{x^k \mapsto m|_\Gamma\}$, $a^k = a^{k+1}|_\Gamma$, $l_1^k = l_1^{k+1}|_\Gamma$, and $l_2^k = l_2^{k+1}|_\Gamma$. This ensures that the shape and the access map are valid and the precondition is satisfied in the k-ranked counterexample.
- Tensor values: Let $v = \mathsf{Y}^{k+1}[x^{k+1} \mapsto \text{fmap}(e, a^{k+1}, l_1^{k+1}, l_2^{k+1})]$. We only require Y^k to have the value v at the access $\{x^k \mapsto \text{fmap}(e, a^k, l_1^k, l_2^k)\}$ and the values at other points are unspecified.

Therefore, for all $k \geq 2$, $\text{valid}(k)$ implies $\text{valid}(k+1)$. This allows us to reduce the unbounded-verification proof obligation to two bounded-verification proof obligations: $\text{valid}(1)$ and $\text{valid}(2)$.

6.3 Bound Computation for the General Case

In §6.2, we derived a bound for the PadLowCombine rule and reduced the unbounded-verification proof obligation to bounded-verification proof obligations. This section first presents a theorem for unbounded verification for a general rewrite rule and an algorithm to compute a bound in the

general case. We then briefly discuss how we handle the complexities of the general case. The detailed proof is in Appendix E. The InferBound routine is described in Algorithm 1.

Lemma 1. Let R be any rewrite rule written in our DSL. Let m be a map containing ranks of RClasses in R. For any RClass c in the rule, if $k = \textsc{InferBound}(R, c)$, then

$$\forall i \geq k,\ \text{valid}(m[c \mapsto i]) \rightarrow \text{valid}(m[c \mapsto i+1])$$

Here, $m[c \mapsto j]$ denotes the map where the rank of c is updated to j, while all other ranks are unchanged. In other words, for all $i \geq k$ and for *any* ranks of the other RClasses, if the rule is valid when c has rank i, then it implies that the rule is valid when c has rank $i+1$. Lemma 1 allows us to use k-induction on the RClass ranks to verify the rule for arbitrary ranks of all RClasses.

Theorem 2. Let R be any rewrite rule written in our DSL. If $c_1 \cdots c_p$ are the RClasses appearing in the rule and $k_i = \textsc{InferBound}(R, c_i)$ for all $i \in \{1 \cdots p\}$, then R is a valid rule in the unbounded setting if and only if

$$\bigwedge_{1 \leq r_1 \leq k_1} \cdots \bigwedge_{1 \leq r_p \leq k_p} \text{valid}(\{c_1 \mapsto r_1, \cdots, c_p \mapsto r_p\})$$

This follows from using Lemma 1 as induction hypothesis for all RClasses.

The InferBound routine in Algorithm 1 takes a rewrite rule R and RClass c as input. On line 3, we use the TensorsWithRClass subroutine to get all the input tensors which have an aggregated-axis having the RClass c. We iterate through all the tensors in lines 4-9. For each tensor, we use the NumTensorAccess subroutine to get the number of distinct accesses to that tensor and add its contribution to the bound. On line 10, we use the NumConds subroutine to get the number of conditions having an aggregated-axis with the RClass c and add it to the bound. We also make sure that the computed bound is at least 1 since we do not want empty aggregated-axes.

We now briefly discuss the complexities that we encounter while tackling a general rewrite rule and how the bound computed by this algorithm is sufficient.

Arbitrary Operator Compositions. For any rule written in our DSL, we can symbolically evaluate both LHS and RHS expressions under a general, valid access. We observe that the result can be expressed in terms of a scalarf function, which could have any number of accesses and conditions, similar to the PadLowCombine rule. Therefore, we need to appropriately handle arbitrary number of accesses and conditions.

Arbitrary Number of Conditions. In general, a rule could have an arbitrary number of conditions, say m. Similar to the PadLowCombine rule, we observe that any condition introduces at most 1 constraint. We assume that all m conditions are independent and we get m constraints in the worst case.

Algorithm 1: Computing the bound for an RClass

Inputs : Rewrite rule R & RClass c
Output: *bound*, i.e., a sufficient rank for c

```
1  Function InferBound (R, c):
2      bound ← 0;
3      tensors ← TensorsWithRClass(R, c);
4      for t ∈ tensors do
5          n ← NumTensorAccess(R, t);
6          if n > 1 then
7              bound ← bound + (n choose 2);
8          end
9      end
10     bound ← bound + NumConds(R, c);
11     return Max(bound, 1)
12 End Function
```

Arbitrary Number of Tensor Accesses. In general, a rule could have an arbitrary number of accesses, say n, to a tensor. We need to make sure that during the projection, accesses containing unequal values do not get projected down to the same point in the tensor, since a point cannot have two different values. We do a pairwise analysis of the accesses ($\binom{n}{2}$ such pairs) and each pair can give at most 1 constraint. Thus, we get $\binom{n}{2}$ constraints from the accesses in the worst case.

Arbitrary Number of RClasses. In general, a rule could have any number of aggregated-axes and RClasses. This would require computing the minimum rank for each RClass. We do so by analyzing each RClass in isolation, i.e., finding a sufficient k for which we can do a $(k+1)$ to k counterexample projection while keeping the ranks of other RClasses unchanged. The bound for this case would be $\binom{n}{2} + m$, where n is the number of accesses and m is the number of conditions in which the RClass appears. The computed bound is independent of the ranks of the other RClasses. Therefore, to ensure correctness in the unbounded setting in presence of multiple RClasses, bounded-verification instances corresponding to all possible combinations of RClass ranks within the bounds need to be verified. This means that if $c_1 \cdots c_p$ are the RClasses appearing in a rule R and $k_i = \textsc{InferBound}(R, c_i)$ for all $i \in \{1 \cdots p\}$, then we need to verify $\prod_{i=1}^{p} k_i$ number of bounded-verification instances.

Arbitrary Number of Input Tensors. In general, a rule could have any number of input tensors. We still analyze each RClass in isolation, but while counting the number of accesses, we only consider accesses to the tensors which contain that RClass. We also make the observation that accesses across tensors do not lead to any constraints, which allows us to do a pairwise analysis of accesses per tensor. We then add the contribution of each tensor to the bound. This is necessary since we take the union of all constraints from all tensors.

6.4 Bounded Verification of Rewrite Rules

We reduced the unbounded-verification proof obligation to a finite set of bounded-verification proof obligations in §6.3. TensorRight infers a sufficient rank for every RClass and instantiates them with all ranks up to that bound, so we end up with fixed-rank but arbitrary-sized tensors. We handle tensors of unbounded size using uninterpreted functions from accesses to values. For any rewrite rule, TensorRight symbolically executes the LHS and RHS tensor expressions using operator semantics and interprets them under a general access with symbolic indices, chosen from the domain of accesses of the two expressions. During the symbolic execution, TensorRight decomposes the verification into two kinds of checks below.

Checks performed during symbolic execution. Each tensor operator constructs the shape of the output tensor using the shapes of the input tensor(s). TensorRight checks that the final LHS and RHS tensors have the same rank and the same named-axes, i.e., `(= (axes lhs) (axes rhs))`. This check is performed entirely by the symbolic execution engine, rather than by the solver.

Checks delegated to the SMT solver. During symbolic execution, TensorRight collects assertions which are then sent to an SMT solver as verification conditions. These assertions check that the axes sizes for LHS and RHS are the same; that accesses fall within axes sizes; and that the values stored in tensor expressions are the same. These verification conditions are described below:

- Assertions related to axes sizes: under the precondition, assuming LHS is valid, assert that the LHS shape and RHS shape are the same. This can be represented as `(=> (&& precond lhsValid) (= (shape lhs) (shape rhs)))`. Asserting the equality of two shapes involves checking if the shapes have the same named-axes and the corresponding axes have the same sizes. As discussed, the former check is done entirely by the symbolic execution engine. However, the solver is needed for the latter check.

- Assertions related to access ranges: under the precondition, assuming LHS is valid, all valid accesses to LHS lead to valid accesses to RHS. This can be represented as `(=> (&& precond lhsValid lhsAccessValid) rhsAccessValid)`.
- Assertions related to final tensor expressions: under the precondition, assuming that LHS is valid, RHS should be valid and they contain the same values under a general access. This can be represented as `(=> (&& precond lhsValid) (&& rhsValid rewriteEquivalent))`.

If the symbolic execution and SMT checks succeed, the rule is deemed verified for that rank.

6.5 Verifying Rules with Reduction Operators

As discussed in §5, automatically verifying expressions with reductions is challenging because:

- The sizes of reduced axes are unbounded.
- Reductions can be performed in multiple steps, such as when tiling the tensors or distributing reduction over concatenation. One such example is the rule $\text{reduce}(\text{concat}(\text{A}, \text{B})) \Rightarrow \text{reduce}(\text{concat}(\text{reduce}(\text{A}), \text{reduce}(\text{B})))$.

In TensorRight, the key idea to verify rewrite rules with reductions is to represent the reduction results as *uninterpreted* reduction elements, $\text{Red}_X f(X)$ (see §5). We observe that the equivalence of two reduction elements $\text{Red}_X f(X)$ and $\text{Red}_Y g(Y)$, can *often* be proven by showing that the LHS and RHS are sums of the same values, i.e., $f(X)$ and $g(Y)$ represent the same (multi-)set.

One way to prove set equivalence is to establish a bijection between X and Y and show each pair of values in $f(X)$ and $g(Y)$ are equal, regardless of tensor instantiation and operator attributes. In TensorRight, the user provides a relation between X and Y as a hint. We then use an SMT solver to verify that it is a bijection: (1) for all valid $x \in X$, a unique $y \in Y$ exists under the relation, and vice versa and (2) the relation can take on all valid $x \in X$ and $y \in Y$. After establishing the bijection, we prove each pair of elements $f(x)$ and $g(y)$ are always the same, regardless of tensor instantiation. Successfully passing the checks reduces our proof to the case discussed in §6.4.

A majority of rules with reductions (13 out of 17) in our system can be proven by establishing the bijection *with* user-provided *hints*. However, there are rules where bijectivity cannot be proven due to limitations of SMT solvers on quantified formulas (1 out of 17), or no such bijection relation exists due to the fact that the cardinalities of the sets of valid reduction indices in LHS and RHS are different (3 out of 17). In these cases, it is up to the user to further complete the proof of set equivalence based on the verifier output. Note that this is generally much easier than proving full correctness from scratch.

7 Discussion

Choice of Tensor Compiler. We chose XLA since it is a production quality compiler and is integrated into leading ML frontend-frameworks like TensorFlow, PyTorch, and JAX. The XLA compiler takes model graphs from these frontends and converts them into XLA-HLO, which is much more expressive than these frameworks. Operators in these frameworks either have direct counterparts in XLA-HLO (e.g., concat, expand, slice), or can be expressed using existing XLA-HLO operators (e.g., squeeze, shrink, split). Some XLA-HLO operators are more general than their counterparts in other frameworks. For instance, tf.tensordot in TensorFlow allows specifying only the contracting axes, whereas DotGeneral in XLA-HLO allows specifying both contracting and batch axes. tf.pad in TensorFlow and Pad in ONNX allow specifying only low and high padding attributes, whereas pad in XLA-HLO allows specifying interior padding as well. Moreover, many operators in the jax.lax module [9] are thin wrappers around equivalent XLA-HLO operators. As for other IR frameworks like ONNX [10], their operators also are either similar to XLA-HLO operators, or can be expressed using XLA-HLO operators. Therefore, XLA-HLO supports a more general set of operators than other frameworks.

Minimum vs Sufficient Rank for Unbounded Verification. §6.3 shows how we can use the InferBound routine to compute a bound for every RClass in a rewrite rule and get a set of bounded-verification instances. These bounded-verification instances are sufficient to imply correctness in the unbounded setting. It is worth noting that this computed bound is only a *sufficient* rank and not the *minimum* rank required for such a property to hold. For instance, in §6.2, we compute a bound for the PadLowCombine rule by calculating the number of conditions and the number of accesses. We assume the two conditions to be independent, each contributing 1 constraint in the worst case, hence getting 2 as the final bound. However, we observe that given the precondition, the condition b_1 implies the condition b_2, which reduces the bound to 1. In general, the conditions may have dependencies and such insights can help us derive a smaller, or maybe even the minimum bound.

8 Evaluation

We evaluated the TensorRight verification framework on the following aspects:

- **Q1:** How expressive is TensorRight DSL compared to other automatic tensor graph rewrite verification systems? (§8.1)
- **Q2:** How good is TensorRight at performing unbounded verification? (§8.2)
- **Q3:** Can TensorRight be used to aid compiler developers in rapid development? (§8.3)

To answer these questions, we selected all rules from the XLA's Algebraic Simplifier (AS) for evaluation. The AS rewrite pass has 175 rules. These rules are implemented to speed up execution and allow further optimizations like fusion in other compiler passes.

8.1 Expressiveness of TensorRight DSL

We compared the expressiveness of TensorRight with two other automatic tensor graph rewrite engines, TASO [20] and PET [38], across all the 175 rules. These rules are categorized into 5 classes, as shown in Table 1. We assessed whether each system can represent rules from these classes. While TASO and PET do not support automatic, unbounded verification, we evaluated whether they can express these rules. We found that TASO can represent 14 rules, and PET can represent 18 rules. Comparatively, TensorRight can represent 121 rules.

We also checked whether TASO and PET can perform bounded verification on the rules they can represent. TASO and PET can prove 6 and 16 rules, respectively. In contrast, TensorRight verified 115 rules in an unbounded setting. Verification statistics are detailed in §8.2.

Categories. Table 1 categorizes all the rules into 5 classes and summarizes representable rules in the systems. Numbers in parentheses indicate verifiable rules, with TensorRight supporting unbounded verification and others using bounded verification strategies. There are two high-level classes: element-wise and non-element-wise rules. Element-wise rules are expressed with element-wise arithmetic operations. They are further divided into rules with basic operators (e.g., +, ∗, div, rem) and rules with advanced operators without good support by solvers (e.g., exp, power). Non-element-wise rules involve operators that change axes sizes or perform reductions. This category is further divided into reductions (e.g., conv, dot, and reduce), layout-sensitive operators (e.g., reshape, bitcast), and others. We separately categorized rules with preconditions. Note that rules with preconditions usually perform non-element-wise operations.

TensorRight. We implemented TensorRight in Haskell using the Grisette [29] symbolic evaluation engine. We represented 121 rules in TensorRight and verified 115 of them in the unbounded setting, using Z3 [15] for most verifications and cvc5 [6] for one advanced element-wise rule.

Table 1 shows that TensorRight was able to represent 46 element-wise simple rules and verified 45 of them. In the advanced element-wise class, TensorRight was able to verify only one rule

Table 1. Number of supported rules per disjoint category. The numbers in parentheses indicate rules that have been implemented and verified. † means verified using cvc5 solver.

Category (disjoint)	Number of Supported Rules			
	XLA	TensorRight	TASO	PET
Elementwise: simple	56	46 (45)	11 (5)	11 (11)
Elementwise: advanced	10	1 (1†)	0 (0)	0 (0)
Non-elementwise: reductions	22	20 (17)	0 (0)	2 (0)
Non-elementwise: layout-sensitive	29	0 (0)	0 (0)	2 (2)
Non-elementwise: others	58	54 (52)	3 (1)	3 (3)
Rules with Preconditions	61	40 (34)	0 (0)	0 (0)
Total	175	121 (115)	14 (6)	18 (16)

involving exp with cvc5. The limitations are due to unsupported operators in Z3 or cvc5 (e.g., log, power) and the inability to reason about precision of floating point expressions efficiently.

Among non-element-wise rules, TensorRight was able to represent 74 and verified 69 rules. For reduction rules, 20 rules were represented, with 3 unproven due to insufficient normalization lemmas and handling cases with extra zeroes. TensorRight DSL cannot represent layout-sensitive rules. We further discuss this in §10. The unsupported rules in the other category are due to unimplemented operators (e.g., scatter, gather), while two rules failed to verify due to timeouts.

Comparison to Prior Works. We compared TensorRight with TASO and PET, which use axiomatic and statistical proof mechanisms, respectively. Table 1 shows that TensorRight was able to support significantly more rules than TASO and PET. The main hurdles for TASO and PET were unsupported operators, too strict operator definitions (e.g., not supporting all the attributes for dot and conv), and inability to handle preconditions. Unlike TensorRight, TASO and PET cannot express rules requiring preconditions, precluding them from supporting 61 XLA rules.

The axiomatic approach requires axioms to prove equivalence of tensor expressions. Out of the 14 representable rules, 8 rules needed new axioms in TASO. It can be even more cumbersome for an axiomatic approach like TASO when we need new axioms with preconditions.

The statistical approach in PET has benefits and drawbacks. It does not need verification conditions proven by SMT solvers, making it potentially more flexible for operations not modeled in SMT. However, PET can only verify linear expressions, which limits its scope. It is potentially feasible to add support for preconditions by only generating test inputs meeting the precondition.

8.2 Verification Capabilities of TensorRight

Experimental Setup. All evaluations were conducted on a system equipped with an Intel Core i9-13900K processor and 128 GB of RAM. We supported boolean, integer, and real-valued tensors in our DSL, and we verified the rules for all valid tensor types for that rule. The timeout per SMT solver query was set to 10 seconds.

Out of the 121 rules that we can express in our DSL, we implemented 118 rules and verified 115 rules in the unbounded setting. Fig. 10a shows cumulative distribution of total verification times for the 115 verified rules. TensorRight was able to verify 108 rules under 1 second, with verification times ranging from a minimum of 0.023 s to a maximum of 23.33 s. Fig. 10b shows the number of bounded-verification proof obligations (tasks) discharged for the verified rules. The number of tasks is simply the product of the computed bounds of all RClasses in a rule. 110 of the rules only required 1 task to guarantee correctness in the unbounded setting.

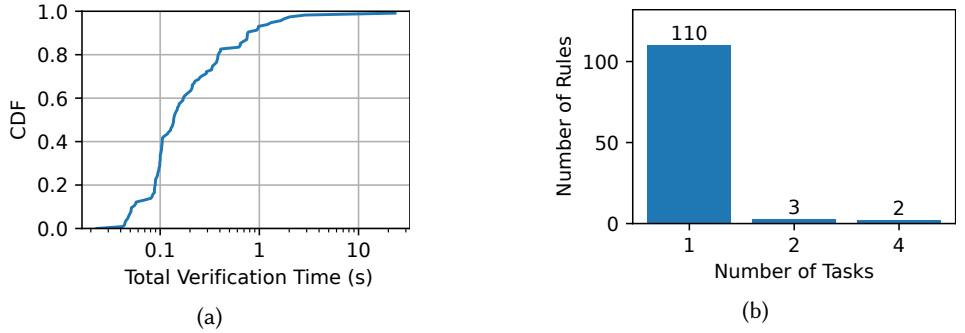


Fig. 10. (a) The cumulative distribution of total verification time and (b) the number of tasks (bounded-verification proof obligations) discharged.

TensorRight was unable to verify 6 rules. This included 3 timeouts, mainly due to those rules having operators like div and rem, which solvers are slow at handling. The remaining 3 rules (not implemented) cannot be proven correct due to missing rules on reduction elements.

8.3 Generalizing Rewrite Rules

Using TensorRight, we found that some XLA rewrite rules are overly constrained. Compiler engineers intentionally impose these constraints to avoid reasoning about cases where spurious bugs might be introduced. We used TensorRight to generalize the following rule by relaxing its precondition.

Fig. 11 (top) presents the FoldConvInputPad rule as it exists in XLA using the TensorRight DSL notation. The goal of the rule is to fold the pad operator into the operand arguments of conv itself (XLA–HLO convolutions support padding as operands). This rule does not support internal padding in the input tensor and gives up if this constraint is violated. This is largely because it is non-trivial to think about how the internal padding gets folded into the dilation attribute. TensorRight was able to prove a more general version of this rule as shown in Fig. 11 (bottom). The differences are put in boxes.

The key to generalizing the rule is to calculate the padding arguments that get fed into the conv operator. This is a function of the pad operator's interior, high, and low padding, as well as padding that may already exist in the conv operator. Fig. 11 (bottom) shows how to calculate the maps S_{ol}, S_{oh}, S_{oi} for the rule to be general. These maps are more complicated than the non-general version, but this allows the compiler writer to get rid of the precondition of the rule shown in Fig. 11 (bottom). It is not immediately clear why this formulation might be correct. Therefore, we encode it in our Grisette implementation and successfully prove that the generalized rule with these calculations is valid.

FoldConvInputPad(XLA):

let $S_{ol} = S_l + S_{lp}$ in
let $S_{oh} = S_h + S_{hp}$ in
$\text{conv}(\text{pad}(t, 0, S_{lp}, S_{hp}, S_{ip}), t',$
$B, F, O, S_l, S_h, S_i, S_i')$
$\Longrightarrow$ $\boxed{S_{ip} = 0 \ \wedge \ S_i = 1}$
$\text{conv}(t, t', B, F, O,$
$S_{ol}, S_{oh}, S_i, S_i')$

FoldConvInputPad(Generalized):

$\boxed{\text{let } S_{ol} = S_l + S_i \times S_{lp} \text{ in}}$
$\boxed{\text{let } S_{oh} = S_h + S_i \times S_{hp} \text{ in}}$
$\boxed{\text{let } S_{oi} = S_i + S_i \times S_{ip} \text{ in}}$
$\text{conv}(\text{pad}(t, 0, S_{lp}, S_{hp}, S_{ip}), t',$
$B, F, O, S_l, S_h, S_i, S_i')$
$\Longrightarrow$
$\text{conv}(t, t', B, F, O,$
$S_{ol}, S_{oh}, S_{oi}, S_i')$

Fig. 11. Fold input pad into conv.

9 Related Works

TensorRight is inspired by prior works on representing and verifying compiler transformations.

Tensor Language Formalisms. Glenside [37] formalizes the syntax and provides a composable abstraction to represent tensor graph rewrites in a purely functional form. TASO [20] uses s-expression based representations to functionally model tensor operators. It does not provide semantics for tensor operators and mostly relies on axioms built around the operators to perform verification related tasks. Such an axiomatic approach would not scale with addition of new tensor operators, as it requires axioms describing operator properties and how the operators interact with each other. TensorRight on the other hand only requires the users to specify operator semantics for every new operator once. PET [38] verifies rewrites rule via a statistical approach. PET symbolically infers the bounding boxes of the output tensor, where each box contains elements represented by the same linear expression of its input elements. Leveraging the linear property, PET statistically verifies the equivalence of the corresponding boxes by checking $m + 1$ specific positions in the box, where m is the number of axes of the output tensor.

There have been many works providing semantics for hardware instructions [14, 19] or general purpose compiler IRs such as LLVM IR [42]. ATL [25] is among the first works to provide denotational semantics to model a tensor language. It is closely modeled after the widely adopted Halide language [32]. In contrast, TensorRight models its core language around the production XLA compiler's High Level Operators. To the best of our knowledge, it is the first formalism supporting XLA-HLO's operators in their full generality, modeling all the parameterizations of operators. Similar to ATL, TensorRight provides denotational semantics of tensor operators with arbitrary rank and size, which is key to the proof that reduces unbounded verification into a bounded setting.

Verification of Rewrites with Proof Assistants. ATL [25] is among the first works to successfully prove correctness of tensor graph rewrites with input tensors of arbitrary shape using the Coq proof assistant. Comparatively, TensorRight does automatic verification given the rewrite specification and accepts preconditions which are prevalent in practical rewrite rules developed by compiler engineers. We note that ATL's Coq based approach supports layout-sensitive rewrites such as those that involve reshapes, which TensorRight does not cover currently. We provide a methodology to support those operators in §10. There are examples from other domains on mechanized proofs on rewrite systems, covering relational algebra [7, 16] and compiler construction tools [18].

Automated Verification of Rewrites. We take inspiration from many successful works focusing on automatically verifying rewrites for different program representations, mainly with the aid of SMT solvers. Alive [28] focuses on verifying rewrite rules in LLVM's Instruction Combiner pass, which is LLVM's peephole optimization. They mainly focus on scalar LLVM IR instructions. Many works on superoptimization use automated verification of rewrites as part of their synthesis process. For example, the STOKE project [35] and others [5] verify rewrites expressed in x86 instructions, Souper [34] verifies rewrites expressed in LLVM IR instructions, Minotaur [27] extends this to vector LLVM IR instructions and [30] proves rewrite rules in Halide IR. TASO [20], PET [38], and TENSAT [41] are examples of systems that automatically synthesize tensor graph rewrites. TensorRight is influenced by the success of these systems and for the first time proposes an automated process for verifying tensor graph rewrites on input tensors of arbitrary ranke and size. Further, TensorRight is the first system to incorporate preconditions in its verification process.

Compiler Verification. There is a lot of work in building verified general-purpose compilers. CompCert [24] is a formally verified C compiler with many verification efforts and extensions [17, 22, 39, 40]. CakeML [23] is a formally verified ML compiler. There are works on building associated verified transformations [21, 42, 43]. Comparatively, less works have explored verifying compiler

transformations in tensor compilers. ATL [25, 26] is one of the first successes on this front that builds upon a proof-assistant-aided verification process. To the best of our knowledge, TensorRight is one of the first efforts at automatically verifying tensor graph rewrites closely resembling the industrial strength XLA tensor compiler.

10 Limitations and Future Work

Currently, TensorRight does not support layout-sensitive rules with operators like reshape or bitcast, which change operand layouts or do not respect element boundaries. The reshape operator is particularly challenging to verify because it can collapse or flatten an arbitrary number of axes, complicating the representation and verification of rank-polymorphic rules. This complexity arises because reshape changes the interpretation of the input tensor by linearizing and de-linearizing its accesses, depending on the rank of input. However, some reshape rules in XLA do not use the operator's full generality. This suggests a pragmatic approach to extend TensorRight to support these simpler cases, addressing much of its practical usage in XLA. The exploration of reshape's full generality is left as potential future work.

As shown in §6.5, TensorRight can currently verify a subset of reduction rules. Users need to provide hints to establish relations between reduction indices, satisfying assumptions like no duplicate values being reduced or 1-1 relations. However, there are limitations, such as handling cases with extra zeroes being reduced, difficulty in proving bijectivity due solver limitations, and instances where no bijection relation exists. In these cases, users need to complete the proof of set equivalence based on verifier output, which is generally easier than proving full correctness from scratch. Future work may explore better proof strategies for reduction rules, possibly using a k-induction approach to establish bounds and finitize sizes for reduction axes.

11 Conclusion

In this paper, we presented TensorRight, the first automatic verification system that allows users to succinctly express and verify tensor graph rewrite rules in their full generality. To do so, we designed TensorRight DSL, which allows specification of rank- and size-polymorphic rewrite rules using a novel axis definition, called aggregated-axes. TensorRight DSL consists of highly parameterized tensor operators, closely resembling those in XLA-HLO. We also provided denotational semantics for TensorRight DSL and used them to convert the unbounded-verification proof obligation to a finite set of bounded-verification proof obligations. To the best of our knowledge, this is the first time a sizable subset of tensor operators from a production-quality tensor IR (XLA-HLO) was formalised. We demonstrated that TensorRight can verify the majority of complex rewrite rules from the production XLA compiler's algebraic simplifier in the unbounded setting, vastly surpassing the closest automatic, bounded-verification technique.

12 Data-Availability Statement

An artifact [3] associated with this paper was evaluated and is freely available.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We would also like to thank Wanyu Zhao for her feedback on early drafts of this paper. This work was supported in part by ACE, one of the seven centers in JUMP 2.0 and the CONIX Research Center, one of the six centers in JUMP, which are Semiconductor Research Corporation (SRC) programs sponsored by DARPA; by NSF under grants CCF-2338739, CCF-2122950, ITE-2132318, and CCF-2437238; by DARPA under grants FA8750-16-2-0032 and D22AP00146-00 as well as gifts from Adobe, Facebook, and Intel.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. https://doi.org/10.1145/3620665.3640366

[3] Jai Arora, Sirui Lu, Devansh Jain, Tianfan Xu, Farzin Houshmand, Phitchaya Mangpo Phothilimthana, Mohsen Lesani, Praveen Narayanan, Karthik Srinivasa Murthy, Rastislav Bodik, Amit Sabne, and Charith Mendis. 2024. *Artifact for "TensorRight: Automated Verification of Tensor Graph Rewrites"*. https://doi.org/10.5281/zenodo.14159871

[4] The JAX Authors. 2024. *Named axes and easy-to-revise parallelism with xmap*. https://web.archive.org/web/20240320140219/https://jax.readthedocs.io/en/latest/notebooks/xmap_tutorial.html archived 2024-03-20.

[5] Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. *SIGOPS Oper. Syst. Rev.* 40, 5 (oct 2006), 394–403. https://doi.org/10.1145/1168917.1168906

[6] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[7] Véronique Benzaken and Évelyne Contejean. 2019. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 249–261. https://doi.org/10.1145/3293880.3294107

[8] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

[9] JAX Contributors. 2024. *jax.lax module*. https://jax.readthedocs.io/en/latest/jax.lax.html

[10] ONNX Contributors. 2024. *ONNX*. https://onnx.ai/onnx/operators/

[11] OpenXLA Contributors. 2024. *OpenXLA Project*. https://web.archive.org/web/20241009145043/https://openxla.org/xla

[12] OpenXLA Contributors. 2024. *XLA-HLO Operation Semantics*. https://openxla.org/xla/operation_semantics

[13] PyTorch Contributors. 2024. *Named Tensors*. https://web.archive.org/web/20240703124627/https://pytorch.org/docs/stable/named_tensor.html archived 2024-07-03.

[14] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1133–1148. https://doi.org/10.1145/3314221.3314601

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[16] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 689–700. https://doi.org/10.1145/2676726.2677006

[17] Delphine Demange, David Pichardie, and Léo Stefanesco. 2015. Verifying Fast and Sparse SSA-Based Optimizations in Coq. In *Compiler Construction*, Björn Franke (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–252.

[18] Jason Gross, Andres Erbsen, Jade Philipoom, Miraya Poddar-Agrawal, and Adam Chlipala. 2022. Accelerating Verified-Compiler Development with a Verified Rewriting Engine. In *13th International Conference on Interactive Theorem*

Proving (ITP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 237), June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:18. https://doi.org/10.4230/LIPIcs.ITP.2022.17

[19] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified synthesis: automatically learning the x86-64 instruction set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 237–250. https://doi.org/10.1145/2908080.2908121

[20] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[21] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. 2021. Language-Parametric Compiler Validation with Application to LLVM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1004–1019. https://doi.org/10.1145/3445814.3446751

[22] Jérémie Koenig and Zhong Shao. 2021. CompCertO: Compiling Certified Open C Components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1095–1109. https://doi.org/10.1145/3453483.3454097

[23] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

[24] Daniel Kästner, Ulrich Wünsche, Jörg Barrho, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS 2018: Embedded Real Time Software and Systems*. SEE. http://xavierleroy.org/publi/erts2018_compcert.pdf

[25] Amanda Liu, Gilbert Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization Via High-level Scheduling Rewrites. In *POPL'22: Proceedings of the 49th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA). http://adam.chlipala.net/papers/AtlPOPL22/

[26] Amanda Liu, Gilbert Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2024. A Verified Compiler for a Functional Tensor Language. In *PLDI'24: Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Copenhagen, Denmark). http://adam.chlipala.net/papers/AtlPLDI24/

[27] Zhengyang Liu, Stefan Mada, and John Regehr. 2023. Minotaur: A SIMD-Oriented Synthesizing Superoptimizer. arXiv:2306.00229 [cs.PL] https://arxiv.org/abs/2306.00229

[28] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 22–32. https://doi.org/10.1145/2737924.2737965

[29] Sirui Lu and Rastislav Bodík. 2023. Grisette: Symbolic Compilation as a Functional Programming Library. *Proc. ACM Program. Lang.* 7, POPL, Article 16 (jan 2023), 33 pages. https://doi.org/10.1145/3571209

[30] Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and improving Halide's term rewriting system with program synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 166 (nov 2020), 28 pages. https://doi.org/10.1145/3428234

[31] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[32] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[33] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch.fx: Practical Program Capture and Transformation for Deep Learning in Python. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 638–651. https://proceedings.mlsys.org/paper_files/paper/2022/file/7c98f9c7ab2df90911da23f9ce72ed6e-Paper.pdf

[34] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2018. Souper: A Synthesizing Superoptimizer. arXiv:1711.04422 [cs.PL] https://arxiv.org/abs/1711.04422

[35] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *SIGPLAN Not.* 48, 4 (mar 2013), 305–316. https://doi.org/10.1145/2499368.2451150

[36] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design*, Warren A. Hunt and Steven D. Johnson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 127–144.

[37] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 21–31. https://doi.org/10.1145/3460945.3464953

[38] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 37–54. https://www.usenix.org/conference/osdi21/presentation/wang

[39] Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (jan 2019), 30 pages. https://doi.org/10.1145/3290375

[40] Yuting Wang, Ling Zhang, Zhong Shao, and Jérémie Koenig. 2022. Verified Compilation of C Programs with a Nominal Memory Model. *Proc. ACM Program. Lang.* 6, POPL, Article 25 (jan 2022), 31 pages. https://doi.org/10.1145/3498686

[41] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper_files/paper/2021/file/cc427d934a7f6c0663e5923f49eba531-Paper.pdf

[42] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 427–440. https://doi.org/10.1145/2103656.2103709

[43] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-Based Optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 175–186. https://doi.org/10.1145/2491956.2462164

Received 2024-07-11; accepted 2024-11-07