Generalizing Shape Analysis with Gradual Types

Zeina Migeed ORCID
University of California, Los Angeles (UCLA), USA

James Reed
Fireworks AI, USA

Jason Ansel ORCID
Meta, USA

Jens Palsberg ORCID
University of California, Los Angeles (UCLA), USA

Abstract

Tensors are multi-dimensional data structures that can represent the data processed by machine learning tasks. Tensor programs tend to be short and readable, and they can leverage libraries and frameworks such as TensorFlow and PyTorch, as well as modern hardware such as GPUs and TPUs. However, tensor programs also tend to obscure shape information, which can cause shape errors that are difficult to find. Such shape errors can be avoided by a combination of shape annotations and shape analysis, but such annotations are burdensome to come up with manually.

In this paper, we use gradual typing to reduce the barrier of entry. Gradual typing offers a way to incrementally introduce type annotations into programs. From there, we focus on tool support for *type migration*, which is a concept that closely models code-annotation tasks and allows us to do shape reasoning and utilize it for different purposes. We develop a comprehensive gradual typing theory to reason about tensor shapes. We then ask three fundamental questions about a gradually typed tensor program. (1) Does the program have a static migration? (2) Given a program and some arithmetic constraints on shapes, can we migrate the program according to the constraints? (3) Can we eliminate branches that depend on shapes? We develop novel tools to address the three problems. For the third problem, there are currently two PyTorch tools that aim to eliminate branches. They do so by eliminating them for just a single input. Our tool is the first to eliminate branches for an infinite class of inputs, using static shape information. Our tools help prevent bugs, alleviate the burden on the programmer of annotating the program, and improves the process of program transformation.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases Tensor Shapes, Gradual Types, Migration

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.11

Related Version *Full Version*: `https://web.cs.ucla.edu/~palsberg/paper/ecoop24.pdf`

Acknowledgements We thank Akshay Utture, Micky Abir and Shuyang Liu for helpful comments.

1 Introduction

Multidimensional data structures are a common abstraction in modern machine learning frameworks such as PyTorch [13], TensorFlow [1], and JAX [5]. A significant portion of programs written using these frameworks involve transformations on tensors. Tensors in this setting are n-dimensional arrays. A tensor is characterized by its *rank* and *shape*. The *rank* is the number of dimensions. For example, a matrix is two-dimensional; hence it is a rank-2 tensor. The *shape* captures the lengths of all axes of the tensor. For example, in a 2×3 matrix, the length of the first axis is 2 and the length of the second axis is 3; hence its shape is $(2, 3)$.

© Zeina Migeed, James Reed, Jason Ansel, and Jens Palsberg;
licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 11; pp. 11:1–11:46
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Programming with tensors provides the programmer with high level and easy to understand constructs. Furthermore, tensors can utilize modern hardware such as GPUs and TPUs for parallelization. For those reasons, programming with tensors is preferred over programming with scalars and nested loops.

Tensors in programming languages present the challenge that their shapes are hard to track. Modern machine learning frameworks support a plethora of operations on tensors, with complex shape rules. Addition for example, typically supports *broadcasting*, which is a mechanism that allows us to add tensors of different shapes, which is not intuitive. Complex shape rules make shapes hard to determine in programs, because shape information rarely explicitly appears in them. As a result, shape errors occur frequently [31].

When not caught statically, shape errors will appear at runtime, which is undesirable because we would only know about the error when the wrong operation is finally invoked on concrete runtime values. Tensor computations are costly and a program may take a long time to run before finally crashing with an error. Additionally, some shape errors occur only for specific input shapes.

The ability to reason about shapes is useful in various contexts in the machine learning area. It can prevent programmers from making mistakes and since programmers routinely transform machine learning programs [17], shape reasoning can also help program transformation tools to make valid program transformations because program transformations may depend on shape information.

Users often add asserts or comments to help them reason about shapes. These tasks have a high cognitive load on users, especially when they are dealing with complex tensor operations. Shape asserts present even further challenges; they can manifest in the form of branches on program shapes. We observed this pattern on various transformer benchmarks [30]. Thus, in that pattern, the result of a branch depends on the shape of the program input, so the branch result can vary over different inputs. In machine learning programs, branches can be undesirable because they limit the back-ends a program can be run on, such backends include TensorRT and XLA. The reason control-flow is undesirable is it complicates fix-point analysis, particularly in shape propagation [17]. In practice, various tools handle this challenge in different ways. Some tools reject such programs entirely while other tools run the program on a single input to eliminate branches. Running a program on a single input means that branch elimination is correct for just one input, which is an unsatisfactory solution.

Aiming to prevent the need for ad-hoc shape asserts, entire systems have been build to detect shape errors such as [15] and [24]. However, these systems are too specific. They lack a general theoretical foundation that enables their solution to be adapted to a variety of contexts, including incorporating their logic into compilers and program transformation tools.

A fundamental approach towards shape analysis is designing a type system that supports reasoning about shapes. In that approach, shapes are type annotations. Traditionally, types have been used to solve similar problems in the area of programming languages. A fully static type system with tensor shapes [20] has limitations. First, a static type system may need to be elaborate in order to capture the complexities of machine learning programs, which are typically written in permissive languages such as Python. As a result, refinement or polymorphic types may be needed. Second, a static type system has a high barrier of entry because it requires the user to come up with non-trivial type annotations in advance. Third, many machine learning programs are in Python, so they are usually only partially typed. Therefore, fully typed programs are not readily available, which prevents this approach from being backwards-compatible.

A common way to circumvent the requirement of having fully typed programs is to use gradual types. In a gradually typed system, type annotations are not needed for the program to compile, when a compiler does type erasure. However, for a gradually typed system to be widely usable, it should enable principled yet practical tool support. Previous work such as [9] designed a gradually typed system for shapes but it is so powerful that practical, elaborate tool support may be hard to obtain. *We believe that the key to shape analysis with gradual types is to balance between (1) the expressiveness of a gradually typed system and (2) the ease of tool support in that system.*

We show that gradual types can help us tackle shape-related problems in a principled and unified way. We introduce a gradual typing system that reasons about shapes and enables tool support.

We distill the challenge of shape analysis into three key problems that we can ask of every gradually typed tensor program, and we introduce a general theory to solve all of them:

- Q(1): *Static migration:* Does the program have a static migration?
- Q(2): *Migration under arithmetic constraints*: Given a program and some arithmetic constraints on shapes, can we migrate the program according to the constraints?
- Q(3): *Branch elimination*: Can we eliminate branches that depend on shapes?

We use PyTorch as the setting for our tool design and evaluation, though our approach is more generally applicable. For Q(1) and Q(2), PyTorch does not currently have any comparable tools, so our tools for those challenges do something new in the PyTorch setting.

For Q(3), we incorporate our shape reasoning into two existing PyTorch tools that aim to eliminate branches from PyTorch programs. After augmenting both tools with our logic, we are able to improve the performance and accuracy of both tools as we will describe below. Our contributions can be summarized as follows:

1. A gradually typed tensor calculus that satisfies static gradual criteria [23].
2. A formal characterization of Q(1), Q(2) and Q(3) and their solutions.
3. A demonstration of how our approach works for Q(1) and Q(2) on four benchmarks.
4. For Q(3), a comparison on six benchmarks, against HuggingFace Tracer (`HFTracer`) [30], a PyTorch tool. `HFTracer` eliminates all branches based on a single input, while we eliminate all branches based on infinite classes of inputs. We use constraints to represent infinite classes of inputs.
5. For Q(3), a comparison on five benchmarks against `TorchDynamo` [2], a PyTorch tool. `TorchDynamo` eliminates 0% of the branches in these benchmarks, while we eliminate branches by 40% to 100% on infinite classes of inputs.

The full version has Appendices A–F with definitions and proofs.

2 Three Migration Problems

In this section, we introduce our type system informally, and we postpone the formal details to Section 3. A tensor type in our system is of the form $\texttt{TensorType}(d_1, \ldots, d_n)$ where $d_1, \ldots, d_n$ are dimensions.

Every gradually typed system has a type `Dyn`, which represents the absence of static type information. In our system, `Dyn` can appear as a dimension, in which case the dimension is unknown. `Dyn` can also appear as a tensor annotation, in which case even the rank of the tensor is unknown.

In a gradual type system, a precision relation refers to the replacement of some of the occurrences of `Dyn` with static types. `Dyn` is the least precise type because it contains no type information. $\texttt{TensorType}(1, 2, 3)$ and $\texttt{TensorType}(1, 2)$ are unrelated by the precision relation because we cannot go from one type to another by replacing `Dyn` occurrences with more informative types, while $\texttt{TensorType}(\texttt{Dyn}, 2)$ is less precise than $\texttt{TensorType}(1, 2)$ because we can replace the `Dyn` in $\texttt{TensorType}(\texttt{Dyn}, 2)$ with 1 to get $\texttt{TensorType}(1, 2)$. This relation extends to programs. Program A is less precise than program B if we can replace some occurrences of `Dyn` in program A to get to program B. Intuitively, program B is more static than program A. Precision gives rise to the *migration space* [12]. Given a well-typed program P, its migration space is the set of well-typed programs that are at least as precise as P.

Intuitively, the migration space captures all ways of annotating a gradually typed program more precisely. Those possibilities form a partially ordered set, and our goal is to help the programmer find the migration paths they are looking for. With that in mind, let us look at examples of how reasoning about the migration space is beneficial for solving key problems about the shapes in a gradually typed program. Specifically, in Section 2, we will see two examples about Q(1) and Q(2) respectively, and in Section 2, we will see an example about Q(3).

For an example of static migration, consider Listing 1 which has a type error.

```
1 class ConvExample(torch.nn.Module):
2     def __init__(self):
3         super(BasicBlock, self).__init__()
4         self.conv1 = torch.nn.Conv2d(in_channels=2, ..)
5         self.conv2 = torch.nn.Conv2d(in_channels=4, ..)
6
7     def forward(self, x: TensorType([Dyn, Dyn])):
8         self.conv1(x)
9         return self.conv2(x)
```

Listing 1 Ill-typed convolution

In line 7, `x` is annotated with `TensorType([Dyn, Dyn])`. This is a typical gradual typing annotation which indicates that `x` is a rank-2 tensor. The annotation does not specify what the dimensions are. In line 8, we are applying a convolution to `x`. Intuitively, convolution is a variant of matrix multiplication; neural networks use it to extract features from images. According to PyTorch's documentation, for the convolution to succeed, `x` cannot be rank-2. Thus, the type error stems from a wrong type annotation. The migration space of this program can easily inform us that the program is ill-typed, because the space will be empty. The reason for that is that the migration space of a well-typed program should contain at least one element, which is the program itself. A tool that can reason about the migration space can easily catch this bug in a single step.

Let us fix this bug by replacing the wrong type annotation with a correct one. In Listing 2, we change `x`'s annotation from a rank-2 annotation to a rank-4 annotation: `TensorType([Dyn, Dyn, Dyn, Dyn]`, which is correct. This program compiles, but it contains a more subtle bug. Let us look closely at the code to understand why.

In line 4, we initialize a field, `self.conv1`, representing a convolution, `torch.nn.Conv2d`, which takes various parameters. The parameter that's relevant to our point is called `in_channels` and it is set to 2. In line 5, we are initializing another field, `self.conv2`, but this time, we set the `in_channels` to 4. In line 7, we have a function that takes a variable `x` and calls both convolutions on it in lines 8 and 9. To understand why this program contains a bug, we must ask: *how does the value of `in_channels` relate to `x`'s shape?* PyTorch's documentation [14] states that in the simplest case, the input to a convolution has the shape $(N, \texttt{in\_channels}, H, W)$. Indeed, in line 7, `x` is annotated with `TensorType([Dyn, Dyn, Dyn, Dyn]`, a typical gradual typing annotation indicating that `x` is a rank-4 tensor. The annotation does not state what the dimensions are, but it is still consistent with the shape stated in the documentation. Notice however that `x`'s second dimension should match the value of `in_channels`, while we have two values for `in_channels` that do not match. This mismatch will cause the program to crash if it ever receives any input, but not before. Our key questions can help us discover the bug statically across all inputs.

```
1 class ConvExample(torch.nn.Module):
2     def __init__(self):
3         super(BasicBlock, self).__init__()
4         self.conv1 = torch.nn.Conv2d(in_channels=2, ...)
5         self.conv2 = torch.nn.Conv2d(in_channels=4, ...)
6 
7     def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
8         self.conv1(x)
9         return self.conv2(x)
```

Listing 2 Gradually typed convolution

By determining whether we can replace all the `Dyn` dimensions with numbers (which is the answer to Q(1) from our key questions), we can discover that it is impossible to assign a number to the second dimension of `x` and thus detect the error before running the program. More generally, the absence of a static typing may reveal that a program cannot run successfully on any input.

How can we benefit from the migration space to answer Q(1) and thus detect that this program cannot be statically typed? The migration space for this program contains programs where `x` is annotated to be a rank-4 tensor. A tool that can reason about the migration space can then take an extra constraint on the second dimension of `x`. The constraint should say that the second dimension must be a number. This constraint will narrow down the migration space to an empty set. The reason is that there is no such well-typed program. Therefore, we can conclude that the program cannot be statically typed because the second dimension cannot be assigned a number.

Let us fix the bug. One way to fix the bug is by removing `self.conv1` from the program. We get the program in Listing 3.

```
class ConvExample(torch.nn.Module):
    def __init__(self):
        super(BasicBlock, self).__init__()
        self.conv2 = torch.nn.Conv2d(in_channels=4, ..)
    def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
        return self.conv2(x)
```

Listing 3 Gradually typed convolution

The program can run to completion and there can be various correct ways to annotate it. The current annotation for the variable x is that it is a tensor with four dimensions, but each dimension is denoted by `Dyn`, so the values of the dimensions are unknown. Suppose we want to specify constraints on those dimensions and determine if there are valid migrations that satisfy those constraints. This would be useful, not just for the user, but for compilers, since they can use those constraints to optimize for resources.

We can require some of the dimensions of `x` to be static and then provide arithmetic constraints on each of them. In this example, let us require all dimensions to be static. A tool can accept four constraints indicating this requirement. Then it can accept constraints that specify ranges on those dimensions. For example, the first dimension could be between

5 and 20. The second dimension can only have one possible value, which is 4. So it is enough to have a constraint requiring that dimension to be a number. The third dimension could also be between 5 and 20, while the fourth dimension could be between 2 and 10.

By giving these constraints as input to a tool, we are constraining the space to only the subspace that satisfies the constraints. A tool may find that this subspace indeed contains programs and outputs one of them. As a result, we may get the program in Listing 4. As shown, `x` has now been statically annotated with `TensorType([19, 4, 19, 9])`.

```
class ConvExample(torch.nn.Module):
    def __init__(self):
        super(BasicBlock, self).__init__()
        self.conv2 = torch.nn.Conv2d(in_channels=4, ..)
    def forward(self, x: TensorType([19, 4, 19, 9])):
        return self.conv2(x)
```

Listing 4 Statically typed convolution

```
1  class ConvControlFlow(torch.nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.conv = torch.nn.Conv2d(
5             in_channels=512, out_channels=512, kernel_size=3)
6
7     def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
8         if  self.conv(x).dim() == 4:
9             return torch.relu(x)
10        else:
11            return torch.nn.Dropout(x)
```

Listing 5 Branch elimination

The program in Listing 5 can run to completion, and interestingly it contains control-flow in the form of a branch. We want to eliminate this branch. We refer to eliminating branches from a program by the term *branch elimination*. Eliminating branches enables programs to run on back-ends where branches are undesirable. For example, `HFTracer` runs a program on a single input and computes the result of the branch and eliminates it accordingly. While the result of a branch could be fixed for all program inputs, the result may also vary. Thus, running a program on just a single input to eliminate a branch yields unsatisfactory branch elimination. We enable better branch elimination by finding all inputs for which a branch evaluates to a given result by reasoning about the program statically. We provide a mechanism to denote the set of inputs for which a branch evaluates to the given result. Notice that we reason about the static information given. Thus, if a variable has type `Dyn`, we optimistically assume that the program is well-typed and that the value for that variable will have the appropriate type at runtime.

The program in Listing 5 contains a condition that depends on shape information. This is a common situation, where ad-hoc shape-checks are inserted in a program to reason about its shapes. Line 8 has function that takes a variable `x` and applies a convolution to it, with `self.conv(x)`, and a condition that checks if the rank of `self.conv(x)` is 4. Since `x` is annotated as a rank-4 tensor on line 7, and convolution preserves the rank, `self.conv(x)` must also be rank-4. So the condition must always be true under the information given by `x`'s type annotation. We should be able to prove that the condition in line 8 always returns true without receiving any input for the program, by inspecting all the valid types that the program could possibly have. The migration space is useful for this analysis because it captures all possible, valid type annotations for a program.

Thus, under the convolution type rules, if `self.conv(x).dim() == 4` evaluates to true, then `x` is also rank-4, which is consistent with `x`'s current annotation.

In contrast, if `self.conv(x).dim() == 4` evaluates to false, i.e `self.conv(x).dim() != 4` is true, then this means that x is not rank-4. However, the migration space of a program can never include inconsistent ranks for a variable. Therefore, it is impossible to have `self.conv(x).dim() != 4`, while also having that `x` is rank-4. A tool that reasons about the migration space as well as arbitrary predicates can make this conclusion. In this example, we can make a definitive conclusion about the result of this condition and we can re-write our program accordingly, as shown in Listing 6. We will expand on and formalize this idea in Section 5. In particular, we will detail how we reason about the migration space in the presence of branches, and explain why our approach works.

```
class ConvControlFlow(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = torch.nn.Conv2d(
            in_channels=512, out_channels=512, kernel_size=3)

    def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
        return torch.relu(x)
```

Listing 6 Branch elimination

3 The Gradual Tensor Calculus

In this section, we describe our design choices, core calculus, and type system, and we prove that our type system satisfy gradual typing criteria.

Our design choices are guided by enabling four key requirements: (1) modularity and backwards compatibility, (2) tool support, (3) expressiveness, and (4) minimality of our language. We have made these four choices in the context of tool support for PyTorch, but they can be extended to other frameworks. Here, we outline those design choices.

First, we require our system to support *modularity and backwards compatibility* for programs. A gradually typed system suits our needs because it supports partial type annotations. One of the implications of this support is that gradually typed programs can compile with any amount of type annotations. In a gradually typed system, a missing type is represented by the `Dyn` type.

The `Dyn` type can sometimes be assigned to a variable that has been used in different parts of the program with different, possibly inconsistent types. This type is useful when the underlying static type system is not flexible enough to fully type that program. For example, we may have a program that takes a batch of images with a dynamic batch size, as well as dynamic sizes, but with a fixed number of channels. In this case, a possible type would be $\texttt{TensorType}(\texttt{Dyn}, 3, \texttt{Dyn}, \texttt{Dyn})$, which indicates a batch of images, where the batch size is dynamic and the sizes are dynamic but the number of channels, which is 3, is fixed. Another example is that a variable could be assigned a rank-2 tensor at one point in the program, then a rank-3 tensor at a different point. A suitable type for that variable could simply be `Dyn`. In both examples, if we did not have the `Dyn` type, we would need more complex annotations. The `Dyn` type allows the gradual type checker to admit programs statically, and determine how to handle variables with `Dyn` types at runtime. The flexibility of gradual types stems from the consistency relation, which is symmetric and reflexive but not transitive. This relation allows a gradual type checker to statically admit programs in the absence of type information.

$$
\begin{array}{rrcl}
(\textit{Program}) & p & ::= & \texttt{decl}^* \; \texttt{return} \; e \\
(\textit{Declaration}) & \texttt{decl} & ::= & x : \tau \\
(\textit{Expression}) & e & ::= & x \mid \texttt{reshape}(e, \tau) \mid \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) \mid \texttt{add}(e_1, e_2) \\
(\textit{Integer Tuple}) & \kappa & ::= & (c^*) \\
(\textit{Const}) & c & ::= & \langle \texttt{Nat} \rangle \\
(\textit{Tensor Type}) & t, \tau & ::= & \texttt{Dyn} \mid \texttt{TensorType}([d_1, \ldots, d_n]) \\
(\textit{Static Tensor Type}) & S, T & ::= & \texttt{TensorType}([D_1, \ldots, D_n]) \\
(\textit{Dimension Type}) & d, \sigma & ::= & \texttt{Dyn} \mid D \\
(\textit{Dimension}) & U, D & ::= & \langle \texttt{Nat} \rangle
\end{array}
$$

$$
\frac{x \notin dom(\Sigma)}{\Sigma, x \rightarrow^* \Sigma, 0, 1} \; (Var\ Fail) \qquad \frac{x : R \in \Sigma}{\Sigma, x \rightarrow^* \Sigma, R, 0} \; (Var)
$$

$$
\frac{\Sigma, e \rightarrow^* \Sigma, R, 1}{\Sigma, \texttt{reshape}(e, \texttt{TensorType}(d_1, \ldots, d_n)) \rightarrow^* \Sigma, R, 1} \; (Reshape\ Fail)
$$

$$
\frac{\Sigma, e \rightarrow^* \Sigma, R, 1}{\Sigma, \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) \rightarrow^* \Sigma, R, 1} \; (Conv2D\ Fail)
$$

$$
\frac{\Sigma, e_1 \rightarrow^* \Sigma, R_1, 1 \vee \Sigma, e_2 \rightarrow^* \Sigma, R_2, 1}{\Sigma, \texttt{add}(e_1, e_2) \rightarrow^* \Sigma, R_2, 1} \; (Add\ Fail)
$$

$$
\frac{\Sigma, e \rightarrow^* \sigma, R, 0}{\Sigma, \texttt{reshape}(e, \texttt{TensorType}(d_1, \ldots, d_n)) \rightarrow^* \Sigma, \textsc{Reshape}(R, (d_1, \ldots, d_n))} \; (Reshape)
$$

$$
\frac{\Sigma, e \rightarrow^* \Sigma, R, 0}{\Sigma, \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) \rightarrow^* \Sigma, \textsc{Conv2D}(c_{in}, c_{out}, \kappa, R)} \; (Conv)
$$

$$
\frac{\Sigma, e_1 \rightarrow^* \Sigma, R_1, 0 \quad \Sigma, e_2 \rightarrow^* \Sigma, R_2, 0}{\Sigma, \texttt{add}(e_1, e_2) \rightarrow^* \Sigma, \textsc{Add}(R_1, R_2)} \; (Add)
$$

Figure 1 Gradual tensor calculus, syntax and semantics

Second, we require *tool support*. We design a simple type system for a core language to enable us to define and solve problems for tool support in a tractable way. Tool support is tractable because we define type migration syntactically. We base our approach on capturing the migration space by extending the constraint-based approach of [12] to solve our three key questions.

Third, we require our system to be *expressive* enough to capture non-trivial programs. Our type system is more expressive than PyTorch's existing type-system, which does not reason about dimensions. Our language consists of a set of declarations followed by an expression. This structure is a convenient representation for the PyTorch neural network models we encountered, which mainly consisted of a function which takes a set of parameters. In the function body are tensor operations applied on those parameters. This calculus structure is inspired by the calculus from [18]. Rink highlighted that many DSLs can be mapped to their language. Besides adapting the structure of that calculus, we choose three core operations that present different challenges for tool support, and then extend our support to 50 PyTorch operations.

Fourth, we require our language to be *minimal* so we can focus on our core problems. First, we do not introduce branches to our core grammar since, in practice, all tools on which we ran our experiments either do not accept programs with branches or aim to eliminate branches. As [17] noted, many non-trivial tensor programs do not contain branches or statements. In Section 5 we extend the core language with branches and we show how to eliminate them.

Second, we do not consider runtime checks to support gradual types. Those checks are often a bottleneck for the performance of gradually typed programs [25, 8]. There has been extensive research to alleviate performance issues by weakening these checks. As shown by [7], the notion of soundness in gradual types is not an all-or-nothing concept. [7] discuss three notions of soundness at different levels of strength and how they relate to performance: higher-order embedding of [26], first-order embedding, as seen in Reticulated Python [28] and erasure embedding, as seen in TypeScript [4]. Similar to [18] and [17], we observe that a language free from higher-order constructs represents a large subset of programs that are written in the machine learning area. As such, runtime errors are not as interesting when compared to those that arise in languages with constructs such as branches and lambda-abstraction. Furthermore, runtime checks impose a computation cost on already costly tensor computations. A key goal of tensor programming is high performance so adding run-time checks seems undesirable. Thus, we leave out runtime aspects in this paper.

Figure 1 shows our core calculus. A program consists of a list of declarations followed by a return statement that evaluates an expression. We use ϵ to denote the empty list of declarations. The program takes its input via those declarations. The dynamic type is denoted by `Dyn`. A dimension can be `Dyn`, and a tensor can also be `Dyn`. A tensor is denoted by the constructor $\texttt{TensorType}(\sigma_1, \ldots, \sigma_n)$ where $\sigma_1, \ldots, \sigma_n$ are dimensions. However, if we denote a dimension by U or D, it means the dimension is a number and cannot be `Dyn`. Our language has four kinds of expressions. A variable x refers to one of the declared variables. The expression $\texttt{add}(e_1, e_2)$ adds two tensors e_1 and e_2. The expression $\texttt{reshape}(e, \tau)$ takes an expression e and a shape τ and reshapes e to a new tensor of shape τ if possible. Reshaping can be thought of as a re-arrangement of a tensor's elements. That requires the initial tensor to have the same number of elements as the reshaped tensor. We require that τ can have a maximum of one `Dyn` dimension. Finally the expression $\texttt{Conv2D}(c_{in}, c_{out}, \kappa, e)$ applies a convolution to e, given a number representing the input channel c_{in}, a number representing the output channel c_{out}, and a pair of numbers representing the kernel κ. For example, in Listing 2, we had `self.conv1(x)`, which in our calculus can be expressed as $\texttt{Conv2D}(2, 2, (2, 2), x)$. The full version of convolution in PyTorch has more parameters. We have accounted for those parameters in our implementation, but because they create no new problems for us, our quest for minimality led us to leaving them out.

The operational semantics in Figure 1 evaluates an expression in an environment Σ that maps each declared variable to a tensor constant. Specifically, if e is an expression, R is a tensor constant, and E an error state (0 for success, 1 for failure), then the judgment $\Sigma, e \to^* R, E$ means that e evaluates to R in error state E.

The semantics uses the helper functions Add, Reshape, and Conv2D that each produces both a tensor constant and an error state. In Appendix C, we give full details of those functions and we state their key properties. Here we summarize what they do. The function Add extracts shapes from T_1 and T_2 and pads them such that they match, and then checks if the tensors are broadcastable based on the updated shapes. If they are not

Consistency

$$\tau \sim \tau \ (\textit{c-refl-t}) \qquad d \sim d \ (\textit{c-refl-d}) \qquad d \sim \texttt{Dyn} \ (\textit{d-refl-dyn}) \qquad \tau \sim \texttt{Dyn} \ (\textit{t-refl-dyn})$$

$$\frac{t \sim \tau}{\tau \sim t} \ (\textit{c-sym-t}) \qquad \frac{d \sim \sigma}{\sigma \sim d} \ (\textit{c-sym-d})$$

$$\frac{\forall i \in \{1, \ldots, n\} : d_i \sim d'_i}{\texttt{TensorType}(d_1, \ldots, d_n) \sim \texttt{TensorType}(d'_1, \ldots, d'_n)} \ (\textit{c-tensor})$$

Type Precision

$$\tau \sqsubseteq \tau \ (\textit{refl-t}) \qquad d \sqsubseteq d \ (\textit{c-refl-d}) \qquad \texttt{Dyn} \sqsubseteq d \ (\textit{refl-dyn-1}) \qquad \texttt{Dyn} \sqsubseteq \tau \ (\textit{refl-dyn-2})$$

$$\frac{\forall i \in \{1, \ldots, n\} : d_i \sqsubseteq d'_i}{\texttt{TensorType}(d_1, \ldots, d_n) \sqsubseteq \texttt{TensorType}(d'_1, \ldots, d'_n)} \ (\textit{p-tensor})$$

Program and Expression Precision

$$\frac{\forall i \in \{1, \ldots, n\} : \texttt{decl}'_i \sqsubseteq \texttt{decl}_i \quad e' \sqsubseteq e}{\texttt{decl}'_1, \ldots, \texttt{decl}'_n \ \texttt{return} \ e' \sqsubseteq \texttt{decl}_1, \ldots, \texttt{decl}_n \ \texttt{return} \ e} \ (\textit{p-prog}) \qquad \frac{\tau' \sqsubseteq \tau}{x : \tau' \sqsubseteq x : \tau} \ (\textit{p-decl})$$

$$e \sqsubseteq e \ (\textit{p-refl})$$

Matching

$$\texttt{TensorType}(\tau_1, \ldots, \tau_n) \rhd^n \texttt{TensorType}(\tau_1, \ldots, \tau_n)$$
$$\texttt{Dyn} \rhd^n \texttt{TensorType}(l) \text{ where } l = [\texttt{Dyn}, \ldots, \texttt{Dyn}] \text{ and } |l| = n$$

Static context formation

$$\frac{}{\epsilon \vdash \emptyset} \ (\textit{s-empty}) \qquad \frac{\texttt{decl}^* \vdash \Gamma \quad x \notin dom(\Gamma)}{\texttt{decl}^* \ x : \tau \vdash \Gamma, \ x : \tau} \ (\textit{s-var})$$

Figure 2 Auxiliary functions

broadcastable, it returns the empty tensor with $E = 1$. Otherwise, it expands the tensors T_1 and T_2 according to the broadcasting rules of PyTorch that we omit here. It initializes a resulting tensor with the broadcasted dimensions and perform element-wise addition between the broadcasted tensors and return that tensor with $E = 0$. The function Reshape performs dimension checks to ensure that reshaping is possible, returning the empty tensor and $E = 1$ if the checks fails. Otherwise, it performs reshaping and returns the reshaped tensor with $E = 0$. The function Conv2D extracts the dimensions of the input tensor I, as well the dimensions for the kernel κ and uses them to determine the size of the output tensor. It then performs convolution and populates the output tensor one element at a time and return the updated tensor along with $E = 0$.

The semantics satisfies the following theorem, which says that in an environment, an expression evaluates to a tensor but may end with failure.

▶ **Theorem 1.** $\forall \Sigma, e : \exists$ *a tensor constant* $R : \exists E \in \{0, 1\} : \Sigma, e \to^* R, E$.

Figure 2 contains gradual typing relations that are used in our gradual typechecking, as well as the static context formation rules. Those relations allow the typechecker to reason about the `Dyn` type. Matching, denoted by $\rhd$, and consistency, denoted by $\sim$, are standard in gradual typing and are lifted from equality in the static counter part of the system. Matching and consistency are both weaker than equality because they account for absent

$$\frac{\texttt{decl}^* \vdash \Gamma \quad \Gamma \vdash e : \tau}{\vdash \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog)} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \ \textit{(t-var)}$$

$$\frac{\Gamma \vdash e : \texttt{TensorType}(D_1, \ldots, D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(U_1, \ldots, U_m)) : \texttt{TensorType}(U_1, \ldots, U_m)} \ \textit{(t-reshape-s)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{TensorType}(\sigma_1, \ldots, \sigma_m) \\ \prod_1^m \sigma_i \ \textit{mod} \ \prod_1^n d_i = 0 \vee \prod_1^n d_i \ \textit{mod} \ \prod_1^m \sigma_i = 0 \ \forall d_i, \sigma_i \neq \texttt{Dyn} \ \textit{and} \\ \texttt{Dyn} \ \textit{occurs exactly once in} \ d_1, \ldots, d_m, \sigma_1, \ldots, \sigma_n, \ \textit{or} \\ \texttt{Dyn} \ \textit{occurs more than once in} \ d_1, \ldots, d_m,\end{array}}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(d_1, \ldots, d_n)) : \texttt{TensorType}(d_1, \ldots, d_n)} \ \textit{(t-reshape-g)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \ \textit{where either} \ \tau = \texttt{Dyn}, \ \textit{or} \ \tau = \texttt{TensorType}(\sigma_1 \ \ldots \ \sigma_n) \ \textit{and} \\ \texttt{Dyn} \ \textit{occurs more than once with at least one occurrence in} \ \delta \ \textit{and} \ \sigma_1, \ldots, \sigma_m,\end{array}}{\Gamma \vdash \texttt{reshape}(e, \delta) : \delta} \ \textit{(t-reshape)}$$

$$\frac{\Gamma \vdash e : t \quad t \rhd^4 \texttt{TensorType}(\sigma_1, \sigma_2, \sigma_3, \sigma_4) \quad \tau = \texttt{calc-conv}(t, c_{out}, \kappa) \quad c_{in} \sim \sigma_2}{\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) : \tau} \ \textit{(t-conv)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad (\tau_1, \tau_2) = \texttt{apply-broadcasting}(t_1, t_2) \quad \tau_1 \sim \tau_2}{\Gamma \vdash \texttt{add}(e_1, e_2) : \tau_1 \sqcup^* \tau_2} \ \textit{(t-add)}$$

Figure 3 Type rules

type information. Thus, if some type information is missing, matching and consistency apply. Matching is a relation that pattern-matches two types. It is useful for arrow types in traditional type systems. Specifically, an arrow type $t_1 \to t_2$ matches itself. Type `Dyn` matches $\texttt{Dyn} \to \texttt{Dyn}$. The ability to expand `Dyn` to become a function type $\texttt{Dyn} \to \texttt{Dyn}$ is valid in gradual types because it allows the system to optimistically consider the type `Dyn` to be $\texttt{Dyn} \to \texttt{Dyn}$. We have adapted this definition to our system. First, we annotated matching with a number n to denote the number of dimensions involved. So we have that $\texttt{TensorType}(\tau_1, \ldots, \tau_n) \rhd^n \texttt{TensorType}(\tau_1, \ldots, \tau_n)$ because any type matches itself. Similar to how traditionally, $\texttt{Dyn} \rhd \texttt{Dyn} \to \texttt{Dyn}$, we have that $\texttt{Dyn} \rhd^n \texttt{TensorType}(\texttt{Dyn}, \ldots, \texttt{Dyn})$, where $\texttt{Dyn}, \ldots, \texttt{Dyn}$ are exactly n dimensions. Throughout this paper, we will only use matching with $i = 4$ so we may use matching as $\rhd$ instead of $\rhd^4$. Consistency is a symmetric, reflexive, and non-transitive relation that checks that two types are equal, up to the known parts of the types. For example, the type `Dyn` contains no information, so it is consistent with any type, while the dimensions 3 and 4 are inconsistent because they are unequal. Figure 2 contains the formal definitions for matching and consistency. The judgment $\texttt{decl}^* \vdash \Gamma$ says that from the declarations $\texttt{decl}^*$ we get the environment Γ. We do static context formation with the rules *(s-empty)* and *(s-var)*.

Figure 3 shows our type rules. We use shorthands that are defined in Appendix B. Let us go over each type rule in detail. *ok-prog* and *t-var* are standard.

t-reshape-s is the static type rule for reshape. It models that for reshape to succeed, the product of the dimensions of the input tensor shape must equal the product of dimensions of the desired shape. *t-reshape-g* assumes we have one missing dimension. Here we are modeling that PyTorch allows a programmer to leave one dimension as unknown (denoted by -1) because the system can deduce the dimension at runtime, see `https://pytorch.org/docs/stable/generated/torch.reshape.html`. We can still determine if reshaping is possible using the modulo operation instead of multiplication. In this approach, we admit a program if we cannot prove it is ill-typed statically. *t-reshape* admits the expression if too many dimensions are missing.

To maintain minimality, *t-conv* deals with only the rank-4 case of convolution. *t-conv* expects a rank-4 tensor, so it uses matching ($\triangleright^4$) to check the rank. Next, c_{in} should be equal to the second dimension of the input, so the rule uses a consistency ($\sim$) check. Since the output of a convolution should also be rank-4, then apply `calc-conv` which, given a rank-4 input and the convolution parameters, computes the dimensions of the output shape. If a dimension is `Dyn`, then the corresponding output dimension will also be `Dyn`.

Finally, *t-add* adds two dimensions. Unlike scalar addition, the types of the operands do not have to be consistent. The reason is that broadcasting may take place. Broadcasting is a mechanism that considers two tensors and matches their dimensions. Two tensors are broadcastable if the following rules hold:

1. Each tensor has at least one dimension
2. When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist

That tensors involved in broadcasting do not actually get modified to represent the modified shapes. This implies that the input shapes are not always consistent. Instead, the broadcasted result is only reflected in the output of the operation. Therefore, we have defined `apply-broadcasting` to simulate broadcasting on the inputs and consider what the types for these inputs would be, if broadcasting was to actually modify the inputs. In a static type system, the types of the modified inputs should be equal for addition to succeed. In gradual types, the types of the modified inputs should be consistent because equality lifts to consistency. We accomplish these requirements in our type rule. In particular, `apply-broadcasting` takes care of broadcasting the dimensions. Suppose that we are adding a tensor of shape $\texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn})$ to a tensor of size $\texttt{TensorType}(1, 2, 2)$. Then the output must be $\texttt{TensorType}(\texttt{Dyn}, 2, 2)$. The reason is that the first `Dyn` could be any number as per the broadcasting rules. So we cannot assume its value. The last dimension; however, must be 2 according to the rules. We have that:

$$\begin{aligned}&\texttt{apply-broadcasting}(\texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}), \texttt{TensorType}(1, 2, 2)) = \\ &\qquad\qquad (\texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}), \texttt{TensorType}(\texttt{Dyn}, 2, 2))\end{aligned}$$

After simulating broadcasting, we may proceed as if we are dealing with regular addition. In other words, we check that the modified dimensions are consistent and get the least upper bound: $\texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}) \sqcup \texttt{TensorType}(\texttt{Dyn}, 2, 2) = \texttt{TensorType}(\texttt{Dyn}, 2, 2)$.

We will cover one last special case for addition. Simply applying the least upper bound to the modified input types of addition is not general enough to cover the following case. Suppose we are adding a tensor of shape `Dyn` to a tensor of shape $\texttt{TensorType}(1, 2)$, then we must output `Dyn` because the output type could be a range of possibilities. In this case, `apply-broadcasting` does not modify the types because the tensor of shape `Dyn` could range over many possibilities. We then apply our modified version of the least upper bound denoted by $\sqcup^*$, which behaves exactly like $\sqcup$ except when one of the inputs is `Dyn`, where it returns `Dyn` to get that: $\texttt{TensorType}(1, 2) \sqcup^* \texttt{Dyn} = \texttt{Dyn}$.

We prove that our type system satisfies the static criteria from [23]. First, we prove the static gradual guarantee, which describes the structure of the migration space. Second, we prove the conservative extension theorem, which shows that our gradual calculus subsumes its static counter-part in Appendix A. This result is no coincidence: we first designed the statically typed calculus in Appendix A and then we gradualized it according to [6]. We denote a well-typed program in the statically typed tensor calculus by $\vdash_{st} p : \texttt{ok}$. The full definitions and proofs can be found in Appendix D.

▶ **Theorem 3.1** (Monotonicity w.r.t precision). $\forall p, p' : \textit{if} \vdash p : \texttt{ok} \wedge p' \sqsubseteq p \textit{ then} \vdash p' : \texttt{ok}.$

▶ **Theorem 3.2** (Conservative Extension). *For all static p, we have: $\vdash_{st} p : \texttt{ok}$ iff $\vdash p : \texttt{ok}$*

4 The Migration Problem as a constraint satisfiability problem

A migration is a more static, well-typed version of a program. We can define that P' is a migration of P (which we write $P \leq P'$) iff ($P \sqsubseteq P' \wedge \vdash P' : \texttt{ok}$). Given P, we define the set of migrations of P: $Mig(P) = \{P' \mid P \leq P'\}$. Our goal is to use constraints to capture the migration space. Every solution to our constraints for a program must map to a corresponding migration for the same program. In other words, one satisfying assignment to the constraints results in one migration.

Our approach involves defining constraints whose solutions are order-isomorphic with the migration space. However, due to the arithmetic nature of our constraints, our solution procedure uses an SMT solver to find a satisfying assignment, which would equate to finding a migration. Later in this paper, we will show how to use this framework to answer our three key questions.

We have two grammars of constraints, see Figure 4: one for source constraints and one for target constraints. We will generate source constraints and then map them to target constraints (as explained in Appendix E), and finally process the target constraints by an SMT solver. Having two grammars is not strictly necessary, but it makes the constraint generation process more tractable and simplifies the presentation. We can view the source grammar as syntactic sugar for the target grammar.

Our source constraint grammar has fourteen forms of constraints, the most interesting of which we will introduce here. A precision constraint is of the form $\tau \sqsubseteq x$. Here, x indicates a type variable for the variable x from the program. Thus, x in the constraint $\tau \sqsubseteq x$ captures all types that are more precise than τ. Because we prioritize tractability of the migration space, we set the upper bound of tensor ranks to 4, via a constraint of the form $|[\![e]\!]| \leq 4$. We make this decision because all benchmarks we considered had only tensors with ranks that are upper-bounded by this number. We also have consistency constraints of the form $D \sim \delta, \langle e \rangle \sim \langle e \rangle$, matching constraints of the form $[\![e]\!] \rhd \texttt{TensorType}(\delta_1, \delta_2, \delta_3, \delta_4)$, and least upper bound constraints of the form $\langle e \rangle \sqcup^* \langle e \rangle$. Those are gradual typing constraints that we use to faithfully model our gradual typing rules. Our constraint grammar also contains short-hands such as $\texttt{can-reshape}([\![e]\!], \delta)$ and $\texttt{apply-broadcasting}([\![e]\!], [\![e]\!])$. Those short-hands are good for representing the type rules as well. `can-reshape` expands to further constraints which evaluate to true if $[\![e]\!]$ can be reshaped to δ. Similarly, when expanded, $\texttt{apply-broadcasting}([\![e]\!], [\![e]\!])$ captures all possible ways to broadcast two types.

In our target constraint grammar, we use n to range over integer constants. We use v as a meta variable that ranges over variables that, in turn, range over $\texttt{TensorType}(list(\zeta)) \cup \{Dyn\}$ and we use ζ as a meta variable that ranges over variables that range over $\texttt{IntConst} \cup \{Dyn\}$. This grammar is useful for our constraint resolution process. In particular, the first step of solving our constraints is to translate them to low-level constraints, drawn from our target grammar, before feeding them to an SMT solver.

$$
\begin{array}{llll}
(\mathit{Source\ Constraints}) & \psi & ::= & \psi \wedge \psi \ \mid\ \psi \vee \psi \ \mid\ \texttt{True} \ \mid\ [\![x]\!] = x \ \mid\ [\![e]\!] = \tau \ \mid\ \tau \sqsubseteq x \ \mid \\
 & & & |[\![e]\!]| \leq 4 \ \mid\ D \sim \delta \ \mid\ \langle e \rangle \sim \langle e \rangle \ \mid \\
 & & & [\![e]\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4) \ \mid \\
 & & & [\![e]\!] = \langle e \rangle \sqcup^* \langle e \rangle \ \mid\ \texttt{can-reshape}([\![e]\!], \delta) \ \mid \\
 & & & [\![e]\!] = \texttt{calc-conv}([\![e]\!], c_{out}, \kappa) \ \mid \\
 & & & \langle e \rangle, \langle e \rangle = \texttt{apply-broadcasting}([\![e]\!], [\![e]\!])
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{Target\ Constraints}) & \psi & ::= & \psi \wedge \psi \ \mid\ \psi \vee \psi \ \mid\ \neg\psi \ \mid\ \texttt{True} \ \mid \\
 & & & v = \texttt{TensorType}(\zeta, \ldots, \zeta) \ \mid \\
 & & & v = \texttt{Dyn} \ \mid\ v = v \ \mid\ \zeta = n \ \mid\ \zeta = \texttt{Dyn} \ \mid\ \zeta = \zeta \ \mid \\
 & & & \zeta = \zeta \cdot n + n \ \mid\ (\zeta_1 \cdot \ldots \cdot \zeta_m) \ \mathit{mod}\ (\zeta'_1 \cdot \ldots \cdot \zeta'_n) = 0
\end{array}
$$

Figure 4 Source constraints and target constraints

Since our constraints involve gradual types, let us describe how we encoded types so that they can be understood by an SMT solver. Because we fixed the upper bound for tensor ranks to be 4, we chose to encode tensor types as uninterpreted functions, which means that we have a constructor for each of our ranks, of the form `TensorType1`, `TensorType2`, `TensorType3`, and `TensorType4`. Each of the functions take a list of dimensions. Moving on to the dimensions, we have that dimensions are either `Dyn` or natural numbers. We can easily represent natural numbers in an SMT solver but we must also represent `Dyn`. One way to encode a `Dyn` dimension d is as a pair (d_1, d_2). If $d_1 = 0$, then $d = \texttt{Dyn}$. Otherwise, d is a number, and its value is in d_2. Let us formalize the constraint generation process next.

From p, we generate constraints $\mathit{Gen}(p)$ as follows. Let p have the form $\texttt{decl}^*\ \texttt{return}\ e$. Let X be the set of declaration-variables x occurring in e, and let Y be a set of variables disjoint from X consisting of a variable $[\![e']\!]$ for every occurrence of the subterm e' in e. Let Z be a set of variables disjoint from X and Y consisting of a variable $\langle e_1 \rangle, \langle e_2 \rangle$ for every occurrence of the subterm $\texttt{add}(e_1, e_2)$ in e. Finally, let V be a set of variables disjoint from X, Y, and Z consisting of dimension variables ζ. The notations $[\![e]\!]$ and $\langle e \rangle$ are ambiguous because there may be more than one occurrence of some subterm e' in e or some subterm e'' in e. However, it will always be clear from context which occurrence is meant. For every occurrence of ζ, it is implicit that we have a constraint $0 \leq \zeta$ to ensure that the solver assigns a dimension in $\mathbb{N}$. We omit writing this explicitly for simplicity. With that in mind, we generate the constraints in Figure 5. Let us go over the rules in Figure 5. The rules use judgments of the form $\vdash x : \tau : \psi$ for declarations, and it uses judgments of the form $\vdash e : \psi$ for expressions. In both cases, ψ is the generated constraint.

t-decl uses the precision relation $\sqsubseteq$ to insure that a migration will have a more precise type, while *t-var* propagates the type information from declarations to the program.

t-reshape considers all possibilities of reshaping any tensor e with rank, at most 4, via the constraint $[\![e]\!] \leq 4$. This restriction constraint captures all rank possibilities for $[\![e]\!]$ in addition to $[\![e]\!]$ being `Dyn`. For each possibility, the number of occurrences of `Dyn` in δ and $[\![e]\!]$ varies. This impacts the arithmetic constraints that make reshaping possible, as we can see from the typing rules. As such, *can-reshape* simulates all such possibilities and generates the appropriate constraints.

$$\frac{}{\vdash x : \tau : \tau \sqsubseteq x \wedge |x| \leq 4}\ (\textit{t-decl}) \qquad \frac{}{\vdash x : x = [\![x]\!]}\ (\textit{t-var})$$

$$\frac{\vdash e : \psi}{\vdash \texttt{reshape}(e, \delta) : \psi \wedge\ [\![\texttt{reshape}(e, \delta)]\!] = \delta \wedge \texttt{can-reshape}([\![e]\!], \delta) \wedge |[\![e]\!]| \leq 4}\ (\textit{t-reshape})$$

$$\frac{\vdash e : \psi}{\begin{array}{c}\vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) : \psi\ \wedge [\![e]\!] \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4) \wedge c_{in} \sim \zeta_2 \wedge \\ [\![\texttt{Conv2D}(c_{in}, c_{out}, \kappa, e)]\!] = \texttt{calc-conv}([\![e]\!], c_{out}, \kappa)\end{array}}\ (\textit{t-conv})$$

$$\frac{\vdash e_1 : \psi_1 \quad \vdash e_2 : \psi_2}{\begin{array}{c}\vdash \texttt{add}(e_1, e_2) : \psi_1 \wedge \psi_2 \wedge [\![\texttt{add}(e_1, e_2)]\!] = \langle e_1 \rangle \sqcup^* \langle e_2 \rangle \wedge \\ (\langle e_1 \rangle, \langle e_2 \rangle) = \texttt{apply-broadcasting}([\![e_1]\!], [\![e_2]\!]) \wedge \langle e_1 \rangle \sim \langle e_2 \rangle \wedge \\ |[\![e_1]\!]| \leq 4 \wedge |[\![e_2]\!]| \leq 4 \wedge |[\![\texttt{add}(e_1, e_2)]\!]| \leq 4\end{array}}\ (\textit{t-add})$$

Figure 5 Constraint generation

t-conv contains matching and consistency constraints, to model matching and consistency in convolution's typing rule. We have a constraint `calc-conv`, which generates the appropriate arithmetic constraints for the output of the convolution, based on the convolution typing rule, again accounting for the possibility of the input e having a gradual type.

t-add contains least upper bound constraints and consistency constraints, similar to the add typing rule. We constrain the inputs e_1 and e_2, as well as the expression itself, $add(e_1, e_2)$ to all be either `Dyn` or tensor of at most rank-4, via a $\leq$ constraint. We use the function `apply-broadcasting`, which simulates broadcasting on the shapes, on dummy variables $\langle e_1 \rangle$ and $\langle e_2 \rangle$ (notice that the real shapes of e_1 and e_2 are represented by $[\![e_1]\!]$ and $[\![e_2]\!]$). We check $\langle e_1 \rangle$ and $\langle e_2 \rangle$ for consistency and obtain the least upper bound.

Let φ be a mapping from tensor-type variables to $\texttt{TensorType}(list(\zeta)) \cup \{Dyn\}$, and also from dimension-type variables to $\texttt{IntConst} \cup \{Dyn\}$. We define that a target constraint ψ has solution φ, written $\varphi \models \psi$, in the following way:

The following is true:	Provided:
$\varphi \models \psi \wedge \psi'$	$\varphi \models \psi$ and $\varphi \models \psi'$
$\varphi \models \psi \vee \psi'$	$\varphi \models \psi$ or $\varphi \models \psi'$
$\varphi \models \neg\psi$	not $(\varphi \models \psi)$
$\varphi \models \texttt{True}$	always
$\varphi \models v = \texttt{TensorType}(\zeta_1, \dots \zeta_n)$	$\varphi(v) = \texttt{TensorType}(\varphi(\zeta_1), \dots \varphi(\zeta_n))$
$\varphi \models v = \texttt{Dyn}$	$\varphi(v) = \texttt{Dyn}$
$\varphi \models v = v'$	$\varphi(v) = \varphi(v')$
$\varphi \models \zeta = n$	$\varphi(\zeta) = n$
$\varphi \models \zeta = \texttt{Dyn}$	$\varphi(\zeta) = \texttt{Dyn}$
$\varphi \models \zeta = \zeta'$	$\varphi(\zeta) = \varphi(\zeta')$
$\varphi \models \zeta = \zeta \cdot n + n'$	$\varphi(\zeta) = \varphi(\zeta') \cdot n + n'$
$\varphi \models (\zeta_1 \cdot \ldots \cdot \zeta_m)\, mod\, (\zeta'_1 \cdot \ldots \cdot \zeta'_n) = 0$	$(\varphi(\zeta_1) \cdot \ldots \cdot \varphi(\zeta_m))\, mod\, (\varphi(\zeta'_1) \cdot \ldots \cdot \varphi(\zeta'_n)) = 0$

▶ **Definition 2.** $\varphi \leq \varphi'$ *iff* $dom(\varphi) = dom(\varphi') \wedge \forall x \in dom(\varphi) : \varphi(x) \sqsubseteq \varphi'(x)$

Let $Gen(P)$ be the constraint generation function and $Sol(C)$ be the set of solutions to constraints C. Then we can state the order-isomorphism theorem as follows:

▶ **Theorem 4.1** (Order-Isomorphism).
$\forall P : \ (\mathit{Mig}(P), \sqsubseteq)$ *and* $(\mathit{Sol}(\mathit{Gen}(P)), \leq)$ *are order-isomorphic.*

The order-isomorphism theorem states that we have captured the migration-space with our constraints such that, for a given program, the solution space and the migration-space are order-isomorphic. For the proof, see Appendix F.

Our algorithm for code annotation is shown in Algorithm 1.

Algorithm 1 Code annotation

Input: Program P
Output: Annotated program P'
1: **Constraint Generation**. Generate constraints $C = \mathit{Gen}(P)$.
2: **Constraint Solving**. Solve C and get a solution φ that maps variables to types.
3: **Program Annotation**. In P, replace each declaration $x : \tau$ with $x : \varphi(x)$, to get P'.

Let us now revisit Listing 1 but this time with variable `x` annotated by `Dyn`. We will show how to migrate a calculus version of the program by generating constraints and passing them to an SMT solver. Let us recall that this listing had two expressions that map to the following expressions in our calculus: $\texttt{Conv2D}(2, 2, (2, 2), x)$ and $\texttt{Conv2D}(4, 2, (2, 2), x)$.

The first step is to generate high-level constraints:

$$\texttt{Dyn} \sqsubseteq v_1 \qquad (1)$$
$$v_1 \leq 4 \qquad (2)$$
$$v_1 \rhd \texttt{TensorType}(\zeta_3, \zeta_4, \zeta_5, \zeta_6) \qquad (3)$$
$$2 \sim \zeta_4 \qquad (4)$$
$$v_2 = \texttt{calc-conv}(v_1, 2, (2, 2), (2, 2), (2, 2), (2, 2)) \qquad (5)$$
$$v_1 \rhd \texttt{TensorType}(\zeta_9, \zeta_{10}, \zeta_{11}, \zeta_{12}) \qquad (6)$$
$$4 \sim \zeta_{10} \qquad (7)$$
$$v_8 = \texttt{calc-conv}(v_1, 2, (2, 2), (2, 2), (2, 2), (2, 2)) \qquad (8)$$

Let us go over what each equation is for. Constraint (1) denotes that the type annotation for the variable x must be as precise or more precise than `Dyn`. Constraint (2) denotes that the type annotation for x could either be `Dyn` or a tensor with at most four dimensions. We use the $\leq$ notation to denote this. Notice that the type variable for x is v_1. Constraints (3), (4), and (5) are for $\texttt{Conv2D}(2, 2, (2, 2), x)$, while constraints (6), (7), and (8) are for $\texttt{Conv2D}(4, 2, (2, 2), x)$. More specifically, constraints (3) and (6) determine the input shape of a convolution while constraints (5) and (8) determine the output shape of a convolution.

The main differences between the constraints for our core calculus and the ones in our implementation is that `calc-conv` takes some additional parameters in our implementation because we have implemented the full version of convolution.

The constraints above are high-level constraints which are yet to be expanded. For example, $\rhd$ and $\leq$ constraints get transformed to equality constraints. We will skip writing out the resulting constraints for simplicity. After expanding these constraints and running them through an SMT solver, we get a satisfying assignment. In case multiple satisfying assignments exist, we use the one that the SMT solver picks. The fact that we got a satisfying assignment lets us know that the migration space is non-empty, which means that the program is well-typed. Let us go through some of relevant assignments:

$$\varphi(v_1) = \texttt{Dyn}$$
$$\varphi(v_2) = \texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}, \texttt{Dyn})$$
$$\varphi(v_8) = \texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}, \texttt{Dyn}))$$

Here, v_1 is the type of x, v_2 is the type of the first convolution and v_8 is the type of the second convolution. We can see that these assignments are a valid typing to the program because the outputs of both convolutions should be 4-dimensional tensors with the second dimension being 2, which stands for the output channel. And since the input x has been assigned `Dyn` by our SMT solver, we cannot determine the last two dimensions of a convolution output. While this is a reasonable output, it may not be helpful to the programmer. Furthermore, this program would not accept any concrete output. We know this from our constraints. From constraints (3) and (7), we have that $\zeta_4 = \zeta_{10}$. Then from (4), (8), which are $2 \sim \zeta_4$ and $4 \sim \zeta_{10}$, we can see that the only satisfying solution is `Dyn`. This means that the program cannot be statically typed. Next, we will see how to prove this formally.

Let us discuss how to extend our approach to solve Q(1) and Q(2). In the example above, the migration space is non-empty and we may want to know if we can statically type the program. We have established that we cannot. As a first step, we may want to take our core constraints above, which we will call C, and restrict the input to a rank-4 tensor. So we can consider the constraint $C \wedge x = \texttt{TensorType}(\zeta'_1, \zeta'_2, \zeta'_3, \zeta'_4)$ where $\zeta'_1, \ldots, \zeta'_4$ are fresh variables. We can begin to impose restrictions on $\zeta'_1, \ldots, \zeta'_4$ to make them concrete variables. For example, if we restrict the last dimension to be a number, we can add the constraint $\zeta'_4 \neq \texttt{Dyn}$. After running our constraints through the solver, we get the following assignments:

$$\varphi(v_1) = \texttt{TensorType}(\texttt{Dyn}, \texttt{Dyn}, \texttt{Dyn}, 28470)$$
$$\varphi(v_2) = \texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}, 14236)$$
$$\varphi(v_8) = \texttt{TensorType}(\texttt{Dyn}, 2, \texttt{Dyn}, 14236)$$

To prove that no concrete assignment to the second dimension of x is possible, we simply add $\zeta'_2 \neq \texttt{Dyn}$ to our original constraints and the constraints will be unsatisfiable, so we conclude that the second dimension of x can only be `Dyn`.

We can also answer Q(2) by feeding the solver additional arithmetic constraints about dimensions. In our example, if we want the first dimension of x to be between 3 and 10, we can add the constraint $\zeta'_1 <= 3 \wedge \zeta'_1 >= 10$ to $C \wedge x = \texttt{TensorType}(\zeta'_1, \zeta'_2, \zeta'_3, \zeta'_4)$ and rerun our solver.

Our migration solution is based on a satisfiability problem: *is our migration problem decidable?* If so, what is the time complexity? The migration problem is decidable if the underlying constraints are drawn from a decidable theory. Those underlying constraints are the ones given by the grammar in Section 4. Let us for a moment ignore constraints of the form $(\zeta_1 \cdot \ldots \cdot \zeta_m)\ mod\ (\zeta'_1 \cdot \ldots \cdot \zeta'_n) = 0$. We observe that all the other constraints are drawn from a well-known decidable theory. Specifically, the other constraints are drawn from quantifier-free Presburger arithmetic extended with uninterpreted functions and equality. The satisfiability problem for this theory is NP-complete [21]. Once we add constraints of the form $(\zeta_1 \cdot \ldots \cdot \zeta_m)\ mod\ (\zeta'_1 \cdot \ldots \cdot \zeta'_n) = 0$, the decidability-status of the satisfiability problem is unknown, to the best of our knowledge. Fortunately, only three operations need this additional constraint: `Reshape`, `View`, or `Flatten`. All the other 47 operations that our implementation supports need only constraints in the NP-complete subset. Our implementation translates all of the constraints to Z3 format, and while our benchmarks do need constraints outside the NP-complete subset, our experiments terminated. In every case, Z3 terminated with either sat or unsat. Thus, the generated constraints are simple enough for Z3 to solve, even if the general case is undecidable.

The complexity of migration depends on the size of the constraint we generate. The bottleneck is the $\leq$ constraint; let us see how to expand it.

$$\text{From:} \quad |[\![e]\!]| \leq 4$$
$$\text{To:} \quad [\![e]\!] = \texttt{Dyn} \vee [\![e]\!] = \texttt{TensorType}(\zeta_1) \vee \ldots \vee [\![e]\!] = \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$$

where $\zeta_1, \ldots, \zeta_4$ are fresh variables. This yields a complexity of 4^n in the number of $\leq$ constraints. So assuming that any additional constraints are drawn from the NP-complete subset, the problem will still be decidable. Note that if we are working with a fixed rank, then these constraints will be generated in polynomial time in the size of the program. Below we will see how solving the problem for a fixed rank has practical benefits.

5 Extending our approach to do Branch Elimination

We introduce our approach to branch elimination via the following example.

```
class ReshapeControlFlow(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x: Dyn):
        if x.reshape(100).size()[0] < 100:
            return torch.dropout(x, p=0.5, train=False)
        else:
            return torch.relu(x)
```

Listing 7 An example of graph-break elimination

In contrast to listing 5, where the conditional depends of the rank of the input, listing 7 has a conditional that depends on the value of one of the dimensions in the input shape. Listing 7 uses the `reshape` function, which takes a tensor and re-arranges its elements according to the desired shape. In this case, we reshape `x` to have the shape $\texttt{TensorType}([100])$. For reshaping to succeed, the initial tensor must contain the same number of elements as the reshaped tensor. Notice that since `x` is typed as `Dyn`, the program will type check. In the expression `x.reshape(100).size()`, the expression `size()` will return the shape of `x.reshape(100)`, which is $[100]$. We are then getting the first dimension of the shape in the expression `x.reshape(100).size()[0]`, which is 100. Thus, by inspecting the conditional `if x.reshape(100).size()[0] < 100`, we can see that the conditional should always evaluate to false. Thus, we can remove the true branch from the program and produce listing 8. In contrast, `TorchDynamo` breaks Listing 7 into two different programs: one for when the condition evaluates to true, and another for when the condition evaluates to false.

```
class ReshapeControlFlow(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x: Dyn):
        return torch.relu(x)
```

Listing 8 An example of graph-break elimination

Let us see an example of how to extend our constraint-based solution to eliminate the extra branch. For listing 7, here are the constraints for `x.reshape(100).size()[0]` in line 6. The variable ζ_4 is for the result of the entire expression. Note that the PyTorch expression `x.reshape(100)` is the same as the calculus expression `reshape(x, TensorType(100))`.

$$\texttt{Dyn} \sqsubseteq v_1 \ \wedge \ v_1 \leq 4 \tag{1}$$

$$v_2 = \texttt{TensorType}(100) \ \wedge \ \texttt{can-reshape}(v_1, \texttt{TensorType}(100)) \tag{2}$$

$$v_2 = v_3 \tag{3}$$

$$\begin{aligned}&(v_3 = \texttt{Dyn} \wedge \zeta_4 = \texttt{Dyn}) \vee ((\zeta_4 = \texttt{GetItem}(v_3, 1, 0) \vee \zeta_4 = \texttt{GetItem}(v_3, 2, 0) \vee \\ &\quad \zeta_4 = \texttt{GetItem}(v_3, 3, 0) \vee \zeta_4 = \texttt{GetItem}(v_3, 4, 0))\end{aligned} \tag{4}$$

Above, the constraint (1) is for x. Notice that v_1 is the type variable for x. Constraint (2) is for `reshape(x, TensorType(100))`. Next, when encountering the `size` function in a program, we simply propagate the shape at hand with an equality constraint, which is seen in equation (3). If we are indexing into a shape, we consider all the possibilities for the sizes of that shape and generate constraints accordingly. In particular, we have that $(v_3 = \texttt{Dyn} \wedge \zeta_4 = \texttt{Dyn})$ because a shape could be dynamic, which means that if we index into it, we get a `Dyn` dimension. But since we restricted our rank to 4, we can consider the possibilities of the index being 1, 2, 3 or 4, which is what the remaining constraints do.

We extend our constraint grammar with constructs that enable us to represent `size()` and indexing into shapes. This includes constraints of the form $\zeta = \texttt{GetItem}(v, c, i)$, where v is the shape we are indexing into, c is the assumed tensor rank, and i is the index of the element we want to get. We can map the new constraints to Z3 constraints easily.

Next we generate a constraint $(\zeta_4 < 100)$ for the condition and a constraint $\neg(\zeta_4 < 100)$ for its negation. If C are the constraints for the program up to the point of encountering a branch, then we generate both $C \wedge \zeta_4 < 100$ and $C \wedge \neg(\zeta_4 < 100)$.

We evaluate both sets of constraints. One set must be satisfiable while the other must be unsatisfiable for us to remove the branch. If we are unable to remove the branch. this means that the input set is still too general such that for some inputs, the branch may evaluate to true and for other inputs, the branch may evaluate to false. In such case, we can ask the user to capture a stricter subset of the input by further constraining it. We can then re-evaluate our constraints again to see if we are able to remove the branch.

We extend our grammar with conditional expressions *if cond then* e_1 *else* e_2. Algorithm 2 describes how to eliminate a single branch.

Algorithm 2 Branch elimination

Input: Program p.
Output: A possibly modified p with a branch eliminated.

Let C = the constraints for p up to encountering a branch *if cond then* e_1 *else* e_2.
Let c_{cond} = the constraints for *cond*.
if $(C \wedge c_{cond})$ is satisfiable and $(C \wedge \neg c_{cond})$ is unsatisfiable **then**
    Rewrite the branch to e_1
else if $(C \wedge c_{cond})$ is unsatisfiable and $(C \wedge \neg c_{cond})$ is satisfiable **then**
    Rewrite the branch to e_2
else
    Require the user to change the shape information
end if

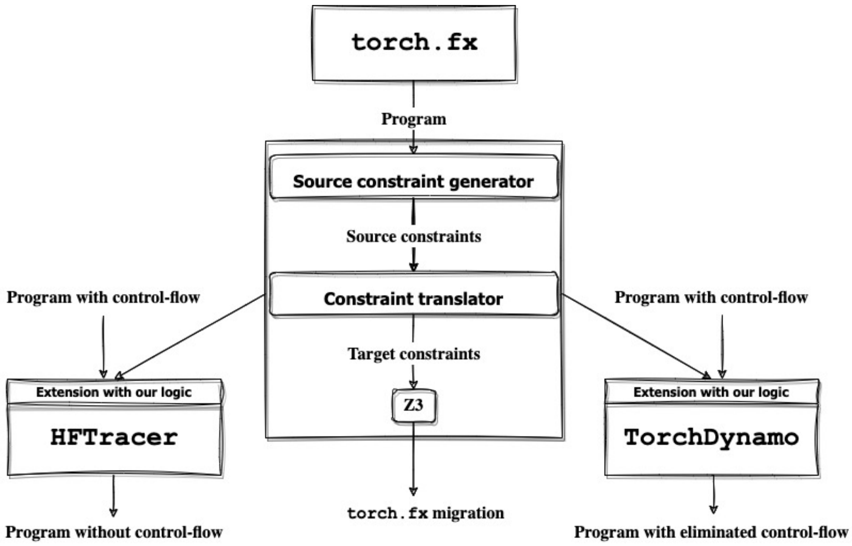

Figure 6 Our core tool and the three tracers

6 Implementation

PyTorch has three tool-kits that rely on symbolic tracers [3]. Let us go over each one. First, `torch.fx` [17] is a common PyTorch tool-kit and has a symbolic tracer. Symbolic tracing is a process of extracting a more specialized program representation from a program, for the purpose of analysis, optimization, serialization, etc. `torch.fx` does not accept programs containing branches and the `torch.fx` authors emphasize that "*most neural networks are expressible as flat sequences of tensor operations without control flow such as if-statements or loops* [17]". `HFtracer` [29] eliminates branches by symbolically executing on a single input. Finally, `TorchDynamo` [2] handles dynamic shapes by dividing the program into fragments. This process is called a *graph-break*. Specifically, when encountering a condition that depends on shape information and where shape information is unknown, the program is broken into two parts. One fragment is for when the result of the condition is true, and another is for when the result of the condition is false. Graph-breaks result in multiple programs with no branches.

As a technical detail, code annotation for the purpose of program understanding and better documentation is meant to be performed on a source language; branch elimination is done at trace-time, on an intermediate representation. For the purpose of better readability, we presented all the examples in Section 2 in source code syntax. In some of our larger benchmarks, the source code is different from the intermediate representation because more high-level constructs were used, such as statements. However, statements do not influence our theoretical results. We did not include sequences in our theory because they did not introduce additional challenges to our problem. Finally, there are some constructs in PyTorch that propagate variable shapes, such as dim() and size(). There are also getters which index into shapes. Those constructs were used to write ad-hoc shape-checks. We dealt with them in our implementation by propagating shape information accordingly.

We have implemented approximately 6000 LOC across three different tracers. Figure 6 summarizes how our implementation works. First, we implement a core constraint generator. This generator takes a program (in our benchmarks case, a program is generated via `torch.fx`), and generates core, source constraints for it. Next is the constraint translator which consists of two phases. In the first phase, it encodes the gradual types found in the program then translates the source constraints into target constraints. Note that a program is annotated, possibly with a `Dyn` type for every variable. In the second phase, it translates the target constraints into Z3 constraints, which is a 1:1 translation.

Next, we modify each of `TorchDynamo` and `HFtracer` to incorporate our reasoning and use it for branch elimination. We must incorporate our logic into the tracers because *branch elimination happens at trace-time*, unlike program migration which requires a whole program.

Our implementation faithfully follows our core logic, although we have made some practical simplifications. First, our implementation focuses on supporting 50 PyTorch operations that our benchmarks use. Each of those operations has its own constraints and supporting all 50 was multiple months of effort. Second, for the `view` operation (which is similar to reshape in terms of types, see `https://pytorch.org/docs/stable/generated/torch.Tensor.view.html`), we have skipped implementing dynamism and required the solver to provide concrete dimensions. This allowed us to carry out branch elimination without requiring an additional constraint that disables dynamism, although the same effect can be accomplished in this manner as well. Third, `Conv2D` may accept rank-3 or rank-4 inputs, but we have limited our implementation to the rank-4 case, since this is the case that is relevant to most of our benchmarks.

We ran our experiments on a MacBook Pro with an 8-Core CPU, 14-Core GPU and 512GB DRAM.

7 Experimental Results

We answer the following three questions.

- Q(1): Can our tool determine if the migration space is non-empty? If so, can it determine if the migration space contains a static migration and if so, can it find one? *Yes. Our tool is the first to affirmatively answer all three questions.*
- Q(2): Given an arithmetic constraint on a dimension, can our tool determine if there is a migration that satisfies it and if so, can it find one? *Yes. Our tool is the first to retrieve migrations that provably satisfy arbitrary arithmetic constraints.*
- Q(3): Can our tool prove that branch elimination is valid for an infinite set of inputs, not just for a single input? If so, does it allow us to represent the set of inputs for which a branch evaluates to true or false? *Yes. We incorporate our logic into two different tools and eliminate branches in all benchmarks we considered for infinite classes of input, characterized via constraints. Neither tool was able to achieve this without our logic.*

Figure 7 contains our benchmark names, the source of the benchmark, lines of code, and the number of `flatten` and `reshape` operations in each benchmark. The `flatten` and `reshape` operations are special because our analysis of them involves multiplication and modulo constraints. Our benchmarks are drawn from two well-known libraries, TorchVision and Transformers [30, 29], with the exception of two microbenchmarks that we use as examples in Section 2. We used different benchmarks for different experiments. The first four models do not contain branches, making them suitable for Q(1) and Q(2). They are interesting because BmmExample has a shape mismatch, ConvExample cannot be statically migrated, and AlexNet and ResNet50 are well-known neural-network models. Our experience is that tensor programs are tricky to type, and that our tool offers feedback that helps the user narrow down the migration space by adding constraints. The next six models are suitable

Benchmark	Source	LOC	Flatten	Reshape	Used for
BmmExample	this paper	4	0	0	Q(1)
ConvExample	this paper	6	0	0	Q(1)
AlexNet	TorchVision	24	1	0	Q(1)
ResNet50	TorchVision	177	1	0	Q(1)
Electra	Transformers	525	0	48	Q(2)
Roberta	Transformers	533	0	48	Q(2)
MobileBert	Transformers	2103	0	96	Q(2)
Bert	Transformers	528	0	48	Q(2)
MegatronBert	Transformers	1018	0	96	Q(2)
XGLM	Transformers	104	0	14	Q(2) and Q(3)
Marian	Transformers	1733	0	315	Q(3)
MarianMT	Transformers	1735	0	315	Q(3)
M2M100	Transformers	1762	0	319	Q(3)
BlenderBot	Transformers	2380	0	451	Q(3)

Figure 7 Benchmark information

	Q(1)		Q(2)	
Benchmark	**Static migration?**	**Time(s)**	**Arithmetic constraints?**	**Time(s)**
BmmExample	No	0.03	No	0.03
ConvExample	No	0.05	Yes	0.08
AlexNet	Yes	2	Yes	2
ResNet50	Yes	5	Yes	347

Figure 8 Q(1) and Q(2): static migration and migration under arithmetic constraints

for our `HFTracer` experiments. Those experiments required reasoning about whole programs and our tool was able to reason about them in under two minutes. The final four benchmarks are of a larger size. We do not support all the operations in those benchmarks. However, this did not pose a problem because in `TorchDynamo`, we were not required to reason about entire programs. Instead, we were required to reason about program fragments, which made our tool terminate in under three minutes.

We ran our tool in the following way to answer Q(1).

1. Generate the core constraints and check if they are satisfiable. If not, stop right away; The program is ill-typed.
2. Determine if the input variable can have a concrete rank by asking the solver for migrations of concrete ranks from one to four. If none exist, the input variable was used at different ranks throughout the program.
3. If the input variable can be assigned concrete ranks, pick one of them and ask the tool to statically annotate all dimensions.
4. If the solver cannot statically annotate all dimensions, relax this requirement for each dimension to determine which one cannot be statically annotated.

We first traced our benchmarks using `torch.fx`, then ran the above steps on the output. The first step simply involves running our tool, while the second and third steps require the user to pass constraints to the tool and rerun it. Determining if a variable has a certain rank requires a single run with our tool. Determining if a dimension can be static requires a single run with our tool. The final step involves removing constraints. Each time we remove a constraint from a dimension, we can run our tool once to determine a result.

The first part of Figure 8 summarizes our results. The first column in the figure is the benchmark name. The second column asks if the benchmark has a static migration and the third column measures the time it took to answer this question and retrieve a static migration. For ConvExample, the input can only be rank-4 and the second dimension can only be `Dyn`. BmmExample has a type error. Finally, ResNet50 and AlexNet can be fully typed and the inputs can only be rank-4 in both cases.

We ran our tool in the following way to answer Q(2). First we follow the steps for answering Q(1), and if any dimensions can be static, then we apply further arithmetic constraints on some of those dimensions and ask for a migration that satisfies them. We ran the steps above in our extension of `torch.fx`. The second part of Figure 8 summarizes our results. The fourth column asks if arithmetic constraints can be imposed on at least one of the dimensions and the fifth column measures the time it took to answer this question and retrieve a migration that satisfies an arithmetic constraint. For ResNet50 and AlexNet, we added arithmetic constraints. For ConvExample, we fixed the example like we did in Section 2 then added arithmetic constraints. We obtained valid migrations that satisfy our constraints for all benchmarks, except for BmmExample which is ill-typed and thus has an empty migration space.

We ran our tool in the following way to answer Q(3). We ran our extension of `HFtracer`, starting with annotating the input with `Dyn` and then gradually increasing the precision of our constraints to provide the solver with more information to eliminate more branches. The number of times we run our tool here depends on how much information the user gives the tool about the input. If the tool receives static input dimensions, then this will be enough to eliminate all branches that depend on shapes. But since we aim to relax this requirement, we could start with a `Dyn` shape then gradually impose constraints, first with rank information, then with dimension information.

We were able to eliminate all branches this way. We followed similar steps in our `TorchDynamo` extension but we faced some practical concerns because `TorchDynamo` currently does not carry parameter information between program fragments. We had to resolve this issue manually by passing additional constraints at every new program fragment.

Figure 9 details our `HFtracer` experiments on 6 workloads. Figure 9 contains the original number of branches in the program, the remaining branches after running our extension, without imposing any constraints on the input, and the number of remaining branches after running our extension, with the constraints in Figure 9 on the input. The second-to-last column of the figure is the time it takes to perform branch elimination with constraints.

`HFtracer` also eliminates all branches from the 6 workloads. However, it does this by running the program on an input. We can obtain a similar result by giving a constraint describing the *shape* of the input because we observed that for all benchmarks we considered, an actual input is not needed to eliminate all branches, and we can relax this requirement much further. Specifically, for some benchmarks, no constraints are needed at all to eliminate all branches, while for others, it is enough to specify rank information. For one of the benchmarks, we can specify a range of dimensions for which branches can be eliminated. Figure 9 details the constraints.

Finally figure 10 represents branch elimination for `TorchDynamo`. There are two modes of operation in `TorchDynamo` called static and dynamic. In the static mode, the tracer traces the program with one input which is provided by the user. Branch elimination is therefore valid for a single input. In Dynamic mode, the tracer also takes an input but it only records *rank* information and ignores the values of the dimensions. So if a branch

	# remaining branches				
		without	with	Time	
Benchmark	original	constr.	constr.	(s)	our constraints
Electra	3	3	0	1	$Tensor(x, y)$
Roberta	3	0	0	3	none
MobileBert	3	3	0	1	$Tensor(x, y)$
Bert	3	0	0	3	none
MegatronBert	3	0	0	5	none
XGLM	5	4	0	22	$Tensor(x, y) \ \wedge x > 0 \wedge 1 < y < 2000$

Figure 9 Q(3): `HFtracer` number of remaining branches

Benchmark	original	with constraints	Time(s)
XGLM	5	0	45
Marian	44	26	70
MarianMT	44	26	75
M2M100	47	22	130
BlenderBot	35	19	40

Figure 10 Q(3): `TorchDynamo` number of remaining branches

depends on dimension information, a graph-break will occur. We focused on benchmarks where branches depend on dimension information. In figure 10, we impose constraints on the dimensions and eliminate branches which decreases the number of times `TorchDynamo` breaks the program when tracing. The first column in the figure indicates the benchmark names. Next is the original number of branches with `TorchDynamo`. Then we have the remaining number of branches after incorporating our reasoning. Finally, we measure time in seconds. The input constraints are range and rank constraints, as exemplified by the constraints for XGLM shown in Figure 9.

From our experiments, we observed that slowdowns can be due to the kind of constraints involved and the number of constraints to solve. Our tool typically handles benchmarks that are under 1000 lines of code easily. However, range constraints impose overhead. For example, ResNet50 and XGLM contain such constraints and they were the slowest in Figure 9. For the experiments under Q(1) and Q(2), we let the tools run more than 5 minutes, but for Q(3) we limit to 5 minutes. The benchmarks in figure 10 are over 1000 lines, and for some branches, branch elimination with `TorchDynamo` times out after 5 minutes.

There are two limitations to our `TorchDynamo` experiments. First, since PyTorch has various operations with many layers of abstractions and edge cases, not every edge case was implemented. Given that this only affected a few branches, we chose to skip those branches. This did not affect our experiments because `TorchDynamo` does not require all branches to be removed. Each branch removed will result in one less graph-break. `TorchDynamo` induces graph-breaks for reasons other than control flow. When graph-breaks happen, we have to re-write an input constraint for the resulting fragments because there is currently no clear mechanism in passing parameter information from one fragment to another. We manually passed input constraints to program fragments until eliminating at least 40% of branches and have stopped after that due to the large size of the benchmarks and program fragments. We leave parameter preservation during graph-breaks to the `TorchDynamo` developers.

8 Related work

We first discuss related work about shapes in tensor programs.

[15] show how to do shape checking based on assertions written by programmers. Their assertions can reason about tensor ranks and dimensions, with arithmetic constraints. Our work also supports such constraints. Their tool executes a program symbolically and looks for assertion violations. The more assertions programmers write, the more shape errors their tool can report. Their tool uses Z3 to solve constraints of a size that can be up to exponential in the size of the program. Our approach is similar in that it enables programmers to annotate a program with types and to type check the program and thereby catch shape errors. Another similarity is that we use Z3 to solve constraints of exponential size. Our approach differs by going further: we have tool support for annotating any program with types and for removing unnecessary runtime shape checks. Additionally, we have proved that our type system has key correctness properties.

[9] define a gradually typed system for tensor computations and, like us, they prove that it has key correctness properties. They use refinement types to represent tensor shapes, they enable programmers to write type annotations, and they do best-effort shape inference. Their refinements share some characteristics with the assertions used by [15], as well as with our constraints. They found that, for each of their benchmarks, few annotations are sufficient to statically type check the entire program. They focus on shape checking and shape inference, while we focus on generalizing shape analysis for various tasks including program migration and branch elimination. Their approach adds the traditional gradual runtime checks [22] in cases where annotations and shape inference fall short. Our work differs by enabling program optimizations through removing runtime checks, while we leave out gradual runtime checks. Conceptually, our approach and the one from [9] differ in that we define type migration syntactically, while they follow a semantic interpretation of gradual types. It is unclear how migration would be defined in their context. Another difference is that we have demonstrated scalability: their benchmark programs are up to 258 lines of code, while our benchmark programs are up to 2,380 lines of code. We were unable to do an experimental comparison because our tool works with PyTorch, while their tool works with OCaml-Torch.

[31] analyzed the root causes of bugs in TensorFlow programs by scanning StackOverFlow and GitHub. They identified four symptoms and seven root causes for such bugs. The most common symptoms are functional errors, crashes, and build failure, while common root causes are data processing errors, type confusion, and dimension mismatches. Our type system can help spot those root causes because key parts of such code will have type `Dyn`, even after migration.

[11] use static analysis to detect shape errors in TensorFlow. Their approach statically detects 11 of the 14 TensorFlow bugs reported by [31], but has no proof of correctness. Our approach differs from [11] by being able to annotate a program with types and being able to remove unnecessary runtime checks. Our work can reason about programs without requiring any type annotations and only taking into account the shape information from the operations used in the program, while [11] requires a degree of type information. In contrast, we have proved that our type system has key migratory properties, such as that our constraints represent the entire migration space for a program, allowing us to extract and reason about all existing shape information from the program according to the programmer's needs.

[10] is a static analysis tool that detects shape errors in PyTorch programs. Their approach is different than ours in that it detects errors via symbolic execution. It considers

all possible execution paths for a program to reason about shapes. The number of execution paths can be large. In contrast, our approach reasons about shapes which can be given in the form of type annotations or can be detected from the program.

[27] consider a dynamic analysis tool for TensorFlow, called ShapeFlow, to detect shape errors. The advantage of this approach is that, like our approach, it does not require type annotations, but their analysis holds for only particular inputs, in contrast to our approach, which reasons about programs across all possible inputs. Unlike our work, their approach has not been formalized, but there is empirical evidence to support that it detects shape errors in *most* cases. Because we reason about programs statically, our work is more suitable for compiler optimizations and program understanding. Our shape analysis approach can be used to annotate programs. In contrast, ShapeFlow is more suitable if a programmer desires a light-weight form for error detection that works in most cases.

[20] designed an intermediate representation called Relay. It is functional, like our calculus, but is statically-typed, unlike our gradual type system. Its goals are similar to ours in that it aims to balance expressiveness, portability, and compilation. Unlike our system, as a static type system, Relay requires type annotations for every function parameter. Similar to our approach, their work focuses on the static aspect of the problem and has left the runtime aspect to future work.

[19] extends [20] by using a static polymorphic type system for shapes, which we leave to future work. This system has a type named `Any`, which enables partial annotations, but which appears to provide less flexibility than our `Dyn` type because of the absence of type consistency.

Next we discuss two closely related papers on migratory typing.

[12] defined the migration space for a gradually typed program as the set of all well-typed, more-precise programs. They represented the migration space for a given program by generating constraints where each solution represents a migration. The constraint-based approach enables them to solve migration problems for a λ-calculus. We adapted their definition of type migration and migration space to our context of a tensor calculus and rather different types. We use their idea of a migration space and constraints to give an algorithm that annotates a program with types and an algorithm that removes unnecessary runtime checks. In contrast to their approach, we use an SMT solver (Z3) because it can deal with the arithmetic nature of tensor constraints.

[16] build a tool which extends [12], by providing several criteria for choosing migrations from the migration space. Their work is about simple types, while our work is about tensor shapes. While their work is specifically focused on reasoning about the migration space for program annotation, we reason about the migration space more generally, by using it for general tensor reasoning tasks including program annotation and branch elimination. Their gradual language contains traditional gradual runtime checks, while we leave out runtime aspects.

9 Conclusion

We have presented a method that reasons about tensor shapes in a general way. Our method involves a gradual tensor calculus with key properties and support for decidable shape analysis for a large set of operations. Our algorithm is practical because it works on 14 non-trivial benchmarks across three different tracers. We expect that our approach to branch elimination can be extended to handle other forms of shape-based optimization.

References

1 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016. URL: `https://arxiv.org/abs/1603.04467`, `doi:10.48550/ARXIV.1603.04467`.

2 Jason Ansel. TorchDynamo. Software release, 1 2022. URL: `https://github.com/pytorch/torchdynamo`.

3 Jason Ansel, Animesh Jain, David Berard, Will Constable, Will Feng, Sherlock Huang, Mario Lezcano, CK Luk, Matthias Reso, Michael Suo, William Wen, Richard Zou, Edward Yang, Michael Voznesensky, Evgeni Burovski, Alban Desmaison, Jiong Gong, Kshiteej Kalambarkar, Yanbo Liang, Bert Maher, Mark Saroufim, Phil Tillet, Shunting Zhang, Ajit Mathews, Horace He, Bin Bao, Geeta Chauhan, Zachary DeVito, Michael Gschwind, Laurent Kirsch, Jason Liang, Yunjie Pan, Marcos Yukio Siraichi, Eikan Wang, Xu Zhao, Gregory Chanan, Natalia Gimelshein, Peter Bell, Anjali Chourdia, Elias Ellison, Brian Hirsh, Michael Lazos, Yinghai Lu, Christian Puhrsch, Helen Suk, Xiaodong Wang, Keren Zhou, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *Proceedings of ASPLOS'24, International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024.

4 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

5 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. JAX: composable transformations of Python+NumPy programs. Software release, 2018. URL: `http://github.com/google/jax`.

6 Matteo Cimini and Jeremy Siek. The gradualizer: A methodology and algorithm for generating gradual type systems. In *Proceedings of POPL'16, ACM Symposium on Principles of Programming Languages*, New York, 2016. ACM.

7 Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *Proc. ACM Program. Lang.*, 2(ICFP), jul 2018. URL: `https://doi.org/10.1145/3236766`, `doi:10.1145/3236766`.

8 Ben Greenman and Zeina Migeed. On the cost of type-tag soundness. In *PEPM*, 2018.

9 Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato. Gradual tensor shape checking, 2022. URL: `https://arxiv.org/abs/2203.08402`, `doi:10.48550/ARXIV.2203.08402`.

10 Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. A static analyzer for detecting tensor shape errors in deep neural network training code. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering (ICSE)*, 2022.

11 Sifis Lagouvardos, Julian T Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. Static analysis of shape in tensorflow programs. In *ECOOP*, Germany, 2020. LIPICS.

12 Zeina Migeed and Jens Palsberg. What is decidable about gradual types? *Proc. ACM Program. Lang.*, 4(POPL), December 2019. URL: `https://doi.org/10.1145/3371097`, `doi:10.1145/3371097`.

13 Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS 2017 Workshop on Autodiff*, 2017. URL: `https://openreview.net/forum?id=BJJsrmfCZ`.

14 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

15 Adam Paszke and Brennan Saeta. Tensors fitting perfectly, 2021. URL: `https://arxiv.org/abs/2102.13254`, `doi:10.48550/ARXIV.2102.13254`.

16 Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. Solver-based gradual type migration. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, New York, 2021. ACM.

17 James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. Torch.fx: Practical program capture and transformation for deep learning in python. Accessed Jul 12, 2024, 2021. URL: `https://arxiv.org/abs/2112.08429`, `doi:10.48550/ARXIV.2112.08429`.

18 Norman A. Rink. Modeling of languages for tensor manipulation. *ArXiv*, abs/1801.08771, 2018.

19 Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: A high-level IR for deep learning. *CoRR*, abs/1904.08368, 2019. URL: `http://arxiv.org/abs/1904.08368`, `arXiv:1904.08368`.

20 Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new IR for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, jun 2018. URL: `https://doi.org/10.1145%2F3211346.3211348`, `doi:10.1145/3211346.3211348`.

21 Sanjit A. Seshia and Randal E. Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds. *Logical Methods in Computer Science*, 1:1–26, 2005.

22 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

23 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL*, pages 274–293, Germany, 2015. LIPICS.

24 Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *ICSE'18, International Conference on Software Engineering*, 2018.

25 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 456–468, 2016.

26 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM. URL: `http://doi.acm.org/10.1145/1328438.1328486`, `doi:10.1145/1328438.1328486`.

27 Sahil Verma and Zhendong Su. Shapeflow: Dynamic shape interpreter for tensorflow, 2020. URL: `https://arxiv.org/abs/2011.13452`, `doi:10.48550/ARXIV.2011.13452`.

28 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. *SIGPLAN Not.*, 52(1):762–774, 2017.

29 Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical*

Methods in Natural Language Processing: System Demonstrations, pages 38–45. Association for Computational Linguistics, 10 2020. URL: `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

30 Wayne Wolf. *Computers as Components, Principles of Embedded Computing System Design*. Morgan Kaufman Publishers, 2000.

31 Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 129140, New York, NY, USA, 2018. Association for Computing Machinery. URL: `https://doi.org/10.1145/3213846.3213866`, `doi:10.1145/3213846.3213866`.

A Static Tensor types

$$
\begin{array}{rrcl}
(Program) & P & ::= & \texttt{DECL}^{*}\ \texttt{return}\ e \\
(Decl) & \texttt{DECL} & ::= & id : T \\
(Terms) & e & ::= & x \mid \texttt{add}(e_1, e_2) \mid \texttt{reshape}(e, T) \mid \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) \\
(IntegerTuple) & \kappa & ::= & (c^{*}) \\
(Const) & c & ::= & \langle \texttt{Nat} \rangle \\
(Tensor\ Types) & S, T & ::= & \texttt{TensorType}(list(D)) \\
 & U, D & ::= & \langle \texttt{Nat} \rangle \\
(Env) & \Gamma & ::= & \emptyset \mid \Gamma, x : T
\end{array}
$$

Figure 11 Tensor Calculus

$$
\frac{\texttt{decl}^{*} \vdash_{st} \Gamma \quad \Gamma \vdash_{st} e : T}{\Gamma \vdash_{st} \texttt{decl}^{*}\ \texttt{return}\ e\ \texttt{ok}}\ \textit{(ok-prog-s)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash_{st} x : T}\ \textit{(t-var)}
$$

$$
\frac{\Gamma \vdash_{st} e : \texttt{TensorType}(D_1, \ldots, D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash_{st} \texttt{reshape}(e, \texttt{TensorType}(U_1, \ldots, U_m)) : \texttt{TensorType}(U_1, \ldots, U_n)}\ \textit{(t-reshape-s)}
$$

$$
\frac{\begin{array}{l}\Gamma \vdash_{st} e : T \qquad T = \texttt{TensorType}(D_1, D_2, D_3, D_4) \\ S = \texttt{calc-conv}(T, c_{out}, \kappa) \quad c_{in} = D_2\end{array}}{\Gamma \vdash_{st} \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) : S}\ \textit{(t-conv)}
$$

$$
\frac{\Gamma \vdash_{st} e_1 : T_1 \quad \Gamma \vdash_{st} e_1 : T_2 \quad (S_1, S_2) = \texttt{apply-broadcasting}(T_1, T_2) \quad S_1 = S_2}{\Gamma \vdash_{st} \texttt{add}(e_1\ e_2) : S_1}\ \textit{(t-add)}
$$

Figure 12 Type Rules

B Gradual Tensor Types: Helper Notation

Least Upper Bound:

$$\tau \sqcup \tau' = \texttt{undefined}, \text{ if } \tau \nsim \tau'$$
$$\tau \sqcup \tau = \tau \quad \texttt{Dyn} \sqcup \tau = \tau \quad \tau \sqcup \texttt{Dyn} = \tau$$
$$\texttt{TensorType}(d_1, \dots, d_n) \sqcup \texttt{TensorType}(d'_1, \dots, d'_n) = \texttt{TensorType}(d_1 \sqcup d'_1, \dots, d_n \sqcup d'_n),$$
$$\text{if } d_1 \sim d'_1, \dots, d_n \sim d'_n$$
$$d_1 \sqcup d_2 = \texttt{undefined}, \text{ if } d_1 \nsim d_2$$
$$d_1 \sqcup d_1 = d_1 \quad d_1 \sqcup \texttt{Dyn} = d_1 \quad \texttt{Dyn} \sqcup d_2 = d_2$$

Least Upper Bound*:

$$\tau \sqcup^* \tau' = \texttt{undefined}, \text{ if } \tau \nsim \tau'$$
$$\tau \sqcup^* \tau = \tau \quad \texttt{Dyn} \sqcup^* \tau = \texttt{Dyn} \quad \tau \sqcup^* \texttt{Dyn} = \texttt{Dyn}$$
$$\texttt{TensorType}(d_1, \dots, d_n) \sqcup^* \texttt{TensorType}(d'_1, \dots, d'_n) =$$
$$\texttt{TensorType}(d_1, \dots, d_n) \sqcup \texttt{TensorType}(d'_1, \dots, d'_n), \quad \text{if } d_1 \sim d'_1, \dots, d_n \sim d'_n$$

Apply-Broadcasting:

$\texttt{apply-broadcasting}(\tau_1, \tau_2)$ is defined as follows:
if $\tau_1 = \texttt{Dyn} \vee \tau_2 = \texttt{Dyn}$ return τ_1, τ_2
else:
let τ_1 and τ_2 be equal in length by padding the shorter type with 1's from index 0
replace occurrences of 1 in τ_1 with the type at the same index in τ_2
replace occurrences of 1 in τ_2 with the type at the same index in τ_1

Calc-Conv:

$$\texttt{calc-conv}(t, c_{\text{out}}, \kappa) = \texttt{TensorType}(t'_0, t'_1, t'_2, t'_3)$$

$$t'_0 = \sigma_0, \quad t'_1 = c_{\text{out}},$$
$$t'_2 = \begin{cases} \sigma_2 - (\kappa[0] - 1) & \text{if } \sigma_2 \in \mathbb{N}, \\ \texttt{Dyn} & \text{otherwise,} \end{cases} \qquad t'_3 = \begin{cases} \sigma_3 - (\kappa[1] - 1) & \text{if } \sigma_3 \in \mathbb{N}, \\ \texttt{Dyn} & \text{otherwise.} \end{cases}$$

C Components of the Runtime Semantics

Algorithm 3 Reshape

procedure RESHAPE($R_{\text{in}}, S_{\text{out}}$)
  Input:
  R_{in}: Input tensor
  S_{out}: Target shape as tuple
  Output:
  R_{out}: Reshaped tensor with shape S_{out}, initialized as the scalar 0, which is a tensor of rank 0
  E: Error state (0 for success, 1 for failure), initialized as 0
  Validation:
  if R_{in} is not a tensor or S_{out} is not a tuple **then**
    $E \leftarrow 1$
    return (R_{out}, E)
  end if
  if "dyn" occurs in S_{out} more than once **then**
    $E \leftarrow 1$
    return (R_{out}, E)
  end if
  $S_{\text{in}} \leftarrow$ SHAPE(R_{in})
  if a single "dyn" dimension in S_{out} **then**
    Remove "dyn" from S_{out}
    $s_{\text{dyn}} \leftarrow (\prod_{d \in S_{\text{in}}} d)\ /\ (\prod_{d \in S_{\text{out}} \setminus \{\text{"}dyn\text{"}\}} d)$
    Replace "dyn" in S_{out} with s_{dyn}
  end if
  if $(\prod_{d \in S_{\text{in}}} d) \neq (\prod_{d \in S_{\text{out}}} d)$ **then**
    $E \leftarrow 1$
    return (R_{out}, E)
  end if
  Reshaping:
  Flatten R_{in} into *srcFlat*
  Create empty R_{out} with shape S_{out} and same data type as R_{in}
  indices $\leftarrow$ list of zeros for each dimension of S_{out}
  for each position in R_{out} **do**
    Assign a value from *srcFlat* to the position in R_{out} based on *indices*
    Update *indices* to navigate dimensions, ensuring wrapping when a dimension is exhausted
  end for
  return (R_{out}, E)
end procedure

Algorithm 4 Custom Broadcasted Addition

procedure ADD(R_1, R_2)
  Input:
  R_1, R_2: Input tensors
  Output:
  R_{out}: Resultant tensor after addition, initialized as the scalar 0, which is a tensor of rank 0
  E: Error state (0 for success, 1 for failure), initialized as 0
  Validation:
  if R_1 is not a tensor **or** R_2 is not a tensor **then**
    $E \leftarrow 1$
    return (R_{out}, E)
  end if
  $S_1 \leftarrow \text{SHAPE}(R_1)$
  $S_2 \leftarrow \text{SHAPE}(R_2)$
  $L_1 \leftarrow \text{LENGTH}(S_1)$
  $L_2 \leftarrow \text{LENGTH}(S_2)$
  if $L_1 < L_2$ **then**
    $S_1 \leftarrow \text{PADWITHONES}(S_1, L_2 - L_1)$
  else if $L_2 < L_1$ **then**
    $S_2 \leftarrow \text{PADWITHONES}(S_2, L_1 - L_2)$
  end if
  for $i = 0$ **to** $L_1 - 1$ **do**
    if $S_1[i] \neq 1$ **and** $S_2[i] \neq 1$ **and** $S_1[i] \neq S_2[i]$ **then**
      $E \leftarrow 1$
      return (R_{out}, E)
    end if
  end for
  Broadcasting and Element-wise Addition:
  $S_{out} \leftarrow$ the element-wise maximum dimensions of S_1 and S_2
  if a dimension in S_1 is 1 and the corresponding dimension in S_2 is greater than 1 **then**
    Expand the dimension in R_1 by copying elements to match S_2
  end if
  if a dimension in S_2 is 1 and the corresponding dimension in S_1 is greater than 1 **then**
    Expand the dimension in R_2 by copying elements to match S_1
  end if
  $R_{\text{out}} \leftarrow$ an initialized tensor with shape S_{out}
  Perform element-wise addition between the expanded R_1 and R_2 and store the result in R_{out}.
  return (R_{out}, E)
end procedure

Algorithm 5 2D Convolution

procedure CONV2D($C_{\text{in}}, C_{\text{out}}, K, R_{\text{in}}$)
  Input:
  C_{in}: Number of input channels
  C_{out}: Number of output channels
  K: Kernel tensor of shape $(C_{\text{out}}, C_{\text{in}}, H_k, W_k)$
  R_{in}: Input image tensor of shape $(B, C_{\text{in}}, H_{\text{in}}, W_{\text{in}})$
  Output:
  R_{out}: Output image tensor, initialized as the scalar 0, which is a tensor of rank 0
  E: Error state (0 for success, 1 for failure), initialized as 0
  Validation:
  if R_{in} is not a 4D tensor **or** K is not a 4D tensor **or**
      C_{in} is not an integer **or** C_{out} is not an integer **then**
    $E \leftarrow 1$
    return (R_{out}, E)
  end if
  if The dimensions of R_{in} or K are not valid for convolution **then**
    $E \leftarrow 1$
    return (R_{out}, E)
  end if
  Convolution:
  $H_{\text{out}} \leftarrow H_{\text{in}} - H_k + 1$
  $W_{\text{out}} \leftarrow W_{\text{in}} - W_k + 1$
  $R_{\text{out}} \leftarrow$ tensor of zeros with shape $(B, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$
  for $b \in \{0, \ldots, B-1\}$ **do**
    for $c_{\text{out}} \in \{0, \ldots, C_{\text{out}}-1\}$ **do**
      for $i \in \{0, \ldots, H_{\text{out}}-1\}$ **do**
        for $j \in \{0, \ldots, W_{\text{out}}-1\}$ **do**
          for $c_{\text{in}} \in \{0, \ldots, C_{\text{in}}-1\}$ **do**
            $R_{\text{out}}[b, c_{\text{out}}, i, j] \leftarrow R_{\text{out}}[b, c_{\text{out}}, i, j] +$
              $\sum_{p=0}^{H_k-1} \sum_{q=0}^{W_k-1} R_{\text{in}}[b, c_{\text{in}}, i+p, j+q] \cdot K[c_{\text{out}}, c_{\text{in}}, p, q]$
          end for
        end for
      end for
    end for
  end for
  return (R_{out}, E)
end procedure

Algorithm 6 Auxiliary Procedures

```
procedure SHAPE(T)
    Input:
    T: Input tensor
    Output:
    S: Shape of the tensor as a tuple
    Determine the dimensions of T and store in S
    return S
end procedure
procedure PADWITHONES(S, n)
    Input:
    S: Original shape as a tuple
    n: Number of ones to pad
    Output:
    P: Padded shape
    P ← tuple of ones of length n concatenated with S
    return P
end procedure
```

▶ **Theorem 3.** *$\forall R_{in}, S_{out}$: $\textsc{Reshape}(R_{in}, S_{out}) = (R_{out}, E)$ where R_{out} is a tensor and $E \in \{0, 1\}$.*

▶ **Theorem 4.** *$\forall R_1, R_2$: $\textsc{Add}(R_1, R_2) = (R_{out}, E)$ where R_{out} is a tensor and $E \in \{0, 1\}$.*

▶ **Theorem 5.** *$\forall C_{in}, C_{out}, K, R_{in} : \textsc{Conv2D}(C_{in}, C_{out}, K, R_{in}) = (R_{out}, E)$ where R_{out} is a tensor and $E \in \{0, 1\}$.*

D Static properties

▶ **Definition 6** (rank). $\mathit{rank}(\mathit{TensorType}(d_1, \ldots, d_n)) = n$.

▶ **Theorem D.1** (Monotonicity w.r.t precision). *$\forall p, p', \Gamma$: if $\Gamma \vdash p : \mathit{ok} \wedge p' \sqsubseteq p$ then $\Gamma \vdash p' : \mathit{ok}$.*

Proof. Proof by induction on the proof structure of $p' \sqsubseteq p$.

Case $\texttt{decl}^{*'}\ \texttt{return}\ e' \sqsubseteq \texttt{decl}^{*}\ \texttt{return}\ e$. Then by inspection, we have:

$$\frac{\forall i \in \{1, \ldots, n\}\ \texttt{decl}'_i \sqsubseteq \texttt{decl}_i \quad e' \sqsubseteq e}{\texttt{decl}'_1, \ldots, \texttt{decl}'_n\ \texttt{return}\ e' \sqsubseteq \texttt{decl}_1, \ldots, \texttt{decl}_n\ \texttt{return}\ e}\ \textit{(p-prog)}$$

We also have the following rule:

$$\frac{\texttt{decl}^* \vdash \Gamma \quad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{decl}^*\ \texttt{return}\ {}^*e\ \texttt{ok}}\ \textit{(ok-prog)}$$

We need to prove that $\Gamma' \vdash \texttt{decl}^{*'}\ \texttt{return}\ e'\ \texttt{ok}$.

We have that $\texttt{decl}^* \vdash \Gamma$. We consider $\texttt{decl}^{*'} \vdash \Gamma'$. Then we know that $\Gamma' \sqsubseteq \Gamma$.

Since $\Gamma \vdash e : \tau$, then by lemma 7, we have that $\Gamma' \vdash e' : \tau'$ where $\tau' \sqsubseteq \tau$. So we have that:

$$\frac{\texttt{decl}^{*'} \vdash \Gamma' \quad \Gamma' \vdash e' : \tau'}{\Gamma' \vdash \texttt{decl}^{*'}\ \texttt{return}\ e'\ \texttt{ok}}\ \textit{(ok-prog)}$$

◀

▶ **Lemma 7** (Monotonicity of expressions). *Suppose $\Gamma \vdash e : \tau$. Then for $\Gamma' \sqsubseteq \Gamma$ and $\Gamma' \vdash e : \tau'$ with $\tau' \sqsubseteq \tau$.*

We proceed by induction on e.

Case x.
We clearly have that $\Gamma \vdash x : \tau$ and $\Gamma' \vdash x : \tau'$ and $\tau' \sqsubseteq \tau$.

Case $\texttt{add}(e_1, e_2)$
We have that:

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad (\tau_1, \tau_2) = \texttt{apply-broadcasting}(t_1, t_2) \quad \tau_1 \sim \tau_2}{\Gamma \vdash \texttt{add}(e_1, e_2) : \tau_1 \sqcup^* \tau_2}\ \textit{(t-add)}$$

By applying the IH, we have that $\Gamma' \vdash e_1 : t'_1$ and $\Gamma' \vdash e_2 : t'_2$ where $t'_1 \sqsubseteq t_1$ and $t'_2 \sqsubseteq t_2$. Note that $\texttt{apply-broadcasting}$ preserves monotonicity, by lemma 8. Furthermore, $\sqcup^*$ and $\sim$ preserve monotonicity. Therefore we can apply *(t-add)* again to get that $\Gamma' \vdash \texttt{add}(e_1, e_2) : t'$ where $t' \sqsubseteq t$.

Case $\texttt{reshape}(e, \tau)$.
We will proceed with case analysis on the derivation rules.

Consider:

$$\frac{\Gamma \vdash e : \texttt{TensorType}(D_1, \ldots, D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(U_1, \ldots, U_m)) : \texttt{TensorType}(U_1, \ldots, U_n)} \ \textit{(t-reshape-s)}$$

By applying the IH, we have that $\Gamma' \vdash e : t$ where $t \sqsubseteq \texttt{TensorType}(D_1, \ldots, D_n)$. First, if $t = \texttt{Dyn}$ or has more than one occurrence of Dyn then we can either *t-reshape* or *t-reshape-g* depending on the occurrences to get that $\Gamma' \vdash \texttt{reshape}(e, \tau) : \tau$. If $t = \texttt{TensorType}(U_1, \ldots, U_n)$ then it must be the case that $D_1 = U_1, \ldots, D_n = U_n$. Otherwise, we know that $\prod_1^n D_i = \prod_1^m U_i$ and that τ' is the same as τ except that one dimension is replaced with Dyn. Therefore, $\prod_1^n D_i$ is divisible by the product of dimensions of τ' so we can apply *t-reshape-g* or *t-reshape* depending on the Dyn occurrences.

Next, consider:

$$\frac{\begin{array}{c} \Gamma \vdash e : \texttt{TensorType}(\sigma_1, \ldots, \sigma_m) \\ \prod_1^m \sigma_i \ mod \ \prod_1^n d_i = 0 \lor \prod_1^n d_i \ mod \ \prod_1^m \sigma_i = 0 \ \forall d_i, \sigma_i \neq \texttt{Dyn} \\ \textit{and } \texttt{Dyn} \textit{ occurs exactly once in } d_1, \ldots, d_m, \sigma_1, \ldots, \sigma_n \\ \textit{or} \\ \texttt{Dyn} \textit{ occurs more than once in } d_1, \ldots, d_m, \end{array}}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(d_1, \ldots, d_n)) : \texttt{TensorType}(d_1, \ldots, d_n)} \ \textit{(t-reshape-g)}$$

From the IH, we have that $\Gamma \vdash e : t$ with $t \sqsubseteq \texttt{TensorType}(\sigma_1, \ldots, \sigma_m)$. Consider t. If $t = \texttt{TensorType}(\sigma_1, \ldots, \sigma_m)$ then apply *t-reshape-g* or *t-resshape* depending on the Dyn occurrences

Finally, we consider:

$$\frac{\begin{array}{c} \Gamma \vdash e : \tau \ \textit{where} \\ \tau = \texttt{TensorType}(\sigma_1 \ \ldots \ \sigma_n) \\ \textit{and } \texttt{Dyn} \textit{ occurs more than once with at least one occurrence in} \\ \delta \textit{ and } \sigma_1, \ldots, \sigma_m \\ \textit{or } \tau = \texttt{Dyn} \end{array}}{\Gamma \vdash \texttt{reshape}(e, \delta) : \delta} \ \textit{(t-reshape)}$$

Then by the IH. we have that $\Gamma' \vdash e : t$ where $t \sqsubseteq \tau$. In this case, we will apply *t-reshape.*

Case $\texttt{Conv2D}(c_{in}, c_{out}, \kappa, e)$.

Then we have:

$$\frac{\Gamma \vdash e : t \quad t \rhd^4 \texttt{TensorType}(\sigma_1, \sigma_2, \sigma_3, \sigma_4) \quad \tau = \texttt{calc-conv}(t, c_{out}, \kappa) \quad c_{in} \sim \sigma_2}{\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) : \tau} \ \textit{(t-conv)}$$

From the IH, $\Gamma' \vdash e' : t'$ with $t' \sqsubseteq t$ and $e' \sqsubseteq e$. We know that $t' \rhd^4 (\sigma_1', \sigma_2', \sigma_3', \sigma_4')$ with $\sigma_i' \sqsubseteq \sigma_i$ for $i \in \{1, \ldots, 4\}$. Since calc-conv preserves monotonicity, by lemma 9, then $calc-conv(t', c_{out}, \kappa) = \tau'$ for $\tau' \sqsubseteq \tau$ so we can apply *t-conv* and we are done.

▶ **Lemma 8** (Monotonicity of broadcasting)**.** *For $t_1' \sqsubseteq t_1$ and $t_2' \sqsubseteq t_2$, we have that if $\texttt{apply-broadcasting}(t_1, t_2) = \tau_1, \tau_2$ then $\texttt{apply-broadcasting}(t_1', t_2') = \tau_1', \tau_2'$ where $\tau_1' \sqsubseteq \tau_1$ and $\tau_2' \sqsubseteq \tau_2$.*

Proof. If either $t_1 = \texttt{Dyn}$ or $t_2 = \texttt{Dyn}$ then we return t_1 and t_2. By the definition of precision, we must have that either either $t'_1 = \texttt{Dyn}$ or $t'_2 = \texttt{Dyn}$ then we return t'_1 and t'_2 and we already know that $t'_1 \sqsubseteq t_1$ and $t'_2 \sqsubseteq t'_2$ so we are done.

Otherwise, we know that t_1, t_2, t'_1 and t'_2 are tensor types. Consider $\texttt{apply-broadcasting}(t_1, t_2) = \tau_1, \tau_2$ and $\texttt{apply-broadcasting}(t'_1, t'_2) = \tau'_1, \tau'_2$. We know that $t_1 \sim t'_1$ and $t_2 \sim t'_2$. So $\texttt{rank}(t_1) = \texttt{rank}(t'_1)$ and $\texttt{rank}(t_2) = \texttt{rank}(t'_2)$. Broadcasting preserves length. Therefore, $\texttt{rank}(\tau_1) = \texttt{rank}(\tau'_1)$ and $\texttt{rank}(\tau_2) = \texttt{rank}(\tau'_2)$.

Now we must show that each of the elements are related by precision, so let $t_1 = \texttt{TensorType}(d_1, \ldots, d_n), t'_1 = \texttt{TensorType}(d'_1, \ldots, d'_n), t_2 = \texttt{TensorType}(k_1, \ldots, k_n)$, $t'_2 = \texttt{TensorType}(k'_1, \ldots, k'_n)$. Then we will have $\tau_1 = \texttt{TensorType}(\delta_1, \ldots, \delta_n)$, $\tau'_1 = \texttt{TensorType}(\delta'_1, \ldots, \delta'_n), \tau_2 = \texttt{TensorType}(\kappa_1, \ldots, \kappa_n)$, $\tau'_2 = \texttt{TensorType}(\kappa'_1, \ldots, \kappa'_n)$.

Assume $d_i = 1$ then $\delta_i = k_i$ and $d'_i = 1$ so $\delta'_i = k'_i$ and we know that $k'_i \sqsubseteq k_i$. Similarly, if $k_i = 1$ then $\kappa_i = d_i$ and $k'_i = 1$ so $\kappa'_i = d'_i$ and we have that $d'_i \sqsubseteq d_i$. ◀

▶ **Lemma 9** (Monotonicity of convolution). *For tensor types t', t : if $t' \sqsubseteq t$ and $\texttt{calc-conv}(t, c_{out}, \kappa) = \tau$ then $\texttt{calc-conv}(t', c_{out}, \kappa) = \tau'$ where $\tau' \sqsubseteq \tau$.*

Proof. Consider $t = \texttt{TensorType}(d_1, \ldots, d_n)$ and $t' = \texttt{TensorType}(d'_1, \ldots, d'_n)$. By applying calc-conv, we have that $d_1 = d'_1$ and $d_2 = d'_2$. By inspection, $d'_3 \sqsubseteq d_3$ and $d'_4 \sqsubseteq d_4$. ◀

▶ **Lemma 10** (Monotonicity of matching). *If $t'_1 \rhd^i t'_2$ and $t'_1 \sqsubseteq t_1$ then $t_1 \rhd^i t_2$ and $t'_2 \sqsubseteq t_2$.*

Proof. Straightforward. ◀

▶ **Theorem 11.** *Let $\tau_1 \sim \tau_2$. Then $\exists \tau_3$ such that $\tau_1 \sqcup^* \tau_2 = \tau_3$*

Proof. We proceed by induction on the derivation.

Consider $\tau \sim \tau$ (*c-refl-t*). Then $\tau \sqcup^* \tau = \tau$. Next, consider $\tau \sim \texttt{Dyn}$. Then we have that $\tau \sqcup^* \texttt{Dyn} = \texttt{Dyn}$.

Next, consider

$$\frac{\forall i \leq n : \tau_i \sim \tau'_i}{\texttt{TensorType}(\tau_1, \ldots, \tau_n) \sim \texttt{TensorType}(\tau'_1, \ldots, \tau'_n)}\ (\textit{c-tensor})$$

Then by induction, we have that $\forall i \in \{1, \ldots, n\} : \tau'_i \sim \tau_i$ so we have that $\tau'_i \sqcup^* \tau_i = \tau_i$". Then we get that

$$\texttt{TensorType}(\tau_1, \ldots, \tau_n) \sqcup^* \texttt{TensorType}(\tau'_1, \ldots, \tau'_n) \quad = \quad \texttt{TensorType}(\tau_1", \ldots, \tau_n")$$

◀

▶ **Theorem 12.** *Gradual Tensor Types are unique*

Proof. Straightforward. ◀

▶ **Theorem 13** (Conservative Extension). *For all static Γ, p, we have: $\Gamma \vdash_{st} p : \textit{ok}$ iff $\Gamma \vdash p : \textit{ok}$*

Forward direction.

We proceed by induction on derivation.

Proof. Case *ok-prog-s*

$$\frac{\texttt{decl}^* \vdash_{st} \Gamma \quad \Gamma \vdash_{st} e : T}{\Gamma \vdash_{st} \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog-s)}$$

so obviously:

$$\frac{\texttt{decl}^* \vdash \Gamma \quad \Gamma \vdash e : T}{\Gamma \vdash \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog)}$$

Case *t-var* is straightforward.

Case *t-reshape-s* maps directly to a rule in the gradual language so it is also straightforward.

Case *t-conv*

$$\frac{\Gamma \vdash_{st} e : T \quad T = \texttt{TensorType}(D_1, D_2, D_3, D_4) \quad S = \texttt{calc-conv}(T, c_{out}, \kappa) \quad c_{in} = D_2}{\Gamma \vdash_{st} \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) : S} \ \textit{(t-conv)}$$

So we have:

$$\frac{\Gamma \vdash e : t \quad T \rhd^4 \texttt{TensorType}(D_1, D_2, D_3, D_4) \quad T = \texttt{calc-conv}(T, c_{out}, \kappa) \quad c_{in} \sim \sigma_2}{\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) : T} \ \textit{(t-conv)}$$

Similarly for:

$$\frac{\Gamma \vdash_{st} e_1 : T_1 \quad \Gamma \vdash_{st} e_2 : T_2 \quad (S_1, S_2) = \texttt{apply-broadcasting}(T_1, T_2) \quad S_1 = S_2}{\Gamma \vdash_{st} \texttt{add}(e_1 \ e_2) : S_1} \ \textit{(t-add)}$$

we have:

$$\frac{\Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2 \quad (S_1, S_2) = \texttt{apply-broadcasting}(S_1, S_2) \quad S_1 \sim S_2}{\Gamma \vdash \texttt{add}(e_1, e_2) : S_1 \sqcup^* S_2} \ \textit{(t-add)}$$

Here, note that since S_1 and S_2 are static and $S_1 = S_2$ then $S_1 \sqcup^* S_2 = S_1$

Backwards direction.

We can proceed by induction on the derivation. We have:

$$\frac{\texttt{decl}^* \vdash \Gamma \quad \Gamma \vdash e : T}{\Gamma \vdash \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog)}$$

From $\texttt{decl}^* \vdash \Gamma$, we get that $\texttt{decl}^* \vdash_{st} \Gamma$.

From the induction on the sub derivation, we get that $\Gamma \vdash_{st} e : T$. Therefore, :

$$\frac{\texttt{decl}^* \vdash_{st} \Gamma \quad \Gamma \vdash_{st} e : T}{\Gamma \vdash_{st} \texttt{decl}^* \ \texttt{return} \ e \ \texttt{ok}} \ \textit{(ok-prog)}$$

t-var is straightforward.

t-reshape-g and *t-reshape* do not apply since they all involve the `Dyn` type.

For *t-reshape-s* we get:

$$\frac{\Gamma \vdash e : \texttt{TensorType}(D_1, \ldots, D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash \texttt{reshape}(e, \texttt{TensorType}(U_1, \ldots, U_m)) : \texttt{TensorType}(U_1, \ldots, U_n)} \ \textit{(t-reshape-s)}$$

we can apply the IH and get that $\Gamma \vdash_{st} e : \texttt{TensorType}(D_1, \ldots, D_n)$. Therefore:

$$\frac{\Gamma \vdash_{st} e : \texttt{TensorType}(D_1, \ldots, D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash_{st} \texttt{reshape}(e, \texttt{TensorType}(U_1, \ldots, U_m)) : \texttt{TensorType}(U_1, \ldots, U_n)} \; \textit{(t-reshape-s)}$$

For *t-conv* we get:

$$\frac{\Gamma \vdash e : S \quad S \rhd^4 \texttt{TensorType}(D_1, D_2, D_3, D_4) \quad T = \texttt{calc-conv}(t, c_{out}, \kappa) \quad c_{in} \sim D_2}{\Gamma \vdash \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) : T} \; \textit{(t-conv)}$$

From the IH, we get that $\Gamma \vdash_{st} e : S$. We know that $\rightarrow$ and $\sim$ are equality on static types, so we can directly apply *t-conv* to get

$$\frac{\begin{array}{l}\Gamma \vdash_{st} e : S \quad S = \texttt{TensorType}(D_1, D_2, D_3, D_4) \\ T = \texttt{calc-conv}(t, c_{out}, \kappa) \quad c_{in} = D_2\end{array}}{\Gamma \vdash_{st} \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e) : T} \; \textit{(t-conv)}$$

Next, we have:

$$\frac{\Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2 \quad (T_2, T_2) = \texttt{apply-broadcasting}(S_1, S_2) \quad T_1 \sim T_2}{\Gamma \vdash \texttt{add}(e_1, e_2) : T_1 \sqcup^* T_2} \; \textit{(t-add)}$$

We have that $\Gamma \vdash_{st} e_1 : T_1$ and $\Gamma \vdash_{st} e_2 : T_2$. We know that $T_1 \sim T_2$ so $T_1 = T_2$. Therefore, $T_1 \sqcup^* T_2 = T_1$ so we get:

$$\frac{\Gamma \vdash_{st} e_1 : S_1 \quad \Gamma \vdash_{st} e_2 : S_2 \quad (T_2, T_2) = \texttt{apply-broadcasting}(S_1, S_2) \quad T_1 = T_2}{\Gamma \vdash_{st} \texttt{add}(e_1, e_2) : T_1} \; \textit{(t-add)}$$

◀

E From Source Constraints to Target Constraints

We define a series of steps that together map source constraints to target constraints.

Precision constraints.

We transform every Precision constraint into zero, one, or more equality constraints. We leave the set of type variables unchanged and we proceed by repeating the following transformation until it no longer has an effect.

From	To
$\texttt{Dyn} \sqsubseteq x$	(no constraint)
$\texttt{TensorType}(D_1, \ldots, D_n) \sqsubseteq x$	$x = \texttt{TensorType}(D_1, \ldots, D_n)$
$\texttt{TensorType}(d_1, \ldots, d_n) \sqsubseteq x$	$x = \texttt{TensorType}(\zeta_1, \ldots, \zeta_n) \;\wedge\; \forall i \in \{1, \ldots, n\} : d_i \sqsubseteq \zeta_i$ where $\zeta_1, \ldots, \zeta_n$ are fresh type variables
$D \sqsubseteq \zeta$	$D = \zeta$
$\texttt{Dyn} \sqsubseteq \zeta$	(no constraint)

$\leq$ constraints.

We replace every $\leq$ constraint as follows.

From: $|[\![e]\!]| \leq 4$

To: $[\![e]\!] = \texttt{Dyn} \vee [\![e]\!] = \texttt{TensorType}(\zeta_1) \vee \ldots \vee [\![e]\!] = \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$

where $\zeta_1, \ldots, \zeta_4$ are fresh variables

Consistency constraints.

From $D \sim \zeta$ to $\zeta = \texttt{Dyn} \vee (D = \zeta)$.
From $\zeta_1 \sim \zeta_2$ to $(\zeta_1 = \texttt{Dyn}) \vee (\zeta_2 = \texttt{Dyn}) \vee (\zeta_1 = \zeta_2)$.

$$\begin{aligned}
\text{From:} \quad & \langle e_1 \rangle \sim \langle e_2 \rangle \\
\text{To:} \quad & \langle e_1 \rangle = Dyn \vee \langle e_2 \rangle = Dyn \vee \ldots \vee \\
& (\langle e_1 \rangle = \texttt{TensorType}(\zeta_1, \ldots, \zeta_4) \wedge \langle e_2 \rangle = \texttt{TensorType}(\zeta_1', \ldots, \zeta_4') \wedge \\
& \zeta_1 \sim \zeta_1' \wedge \ldots \wedge \zeta_4 \sim \zeta_4')
\end{aligned}$$

Matching constraints.

$$\begin{aligned}
\text{From:} \quad & [\![e]\!] \triangleright \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4) \\
\text{To:} \quad & ([\![e]\!] = \texttt{Dyn} \wedge \zeta_1 = \texttt{Dyn} \wedge \zeta_2 = \texttt{Dyn} \wedge \zeta_3 = \texttt{Dyn} \wedge \zeta_4 = \texttt{Dyn}) \vee \\
& ([\![e]\!] = \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4))
\end{aligned}$$

$\sqcup^*$ constraints.

$$\begin{aligned}
\text{From:} \quad & [\![e]\!] = \langle e_1 \rangle \sqcup^* \langle e_2 \rangle \\
\text{To:} \quad & ((\langle e_1 \rangle = \texttt{Dyn} \vee \langle e_2 \rangle = \texttt{Dyn}) \wedge [\![e]\!] = \texttt{Dyn}) \vee \\
& \forall i \in \{1, \ldots, 5\}(\langle e_1 \rangle = \texttt{TensorType}(\epsilon_1, \ldots, \epsilon_i) \wedge \\
& \langle e_2 \rangle = \texttt{TensorType}(\epsilon_1', \ldots, \epsilon_i) \wedge [\![e]\!] = \texttt{TensorType}(\zeta_1, \ldots \zeta_i) \wedge \\
& \zeta_1 = (\epsilon_1 \sqcup \epsilon_1') \wedge \ldots \wedge \zeta_i = (\epsilon_i \sqcup \epsilon_i'))
\end{aligned}$$

$\sqcup$ constraints

$$\begin{aligned}
\text{From:} \quad & \epsilon = \zeta_1 \sqcup \zeta_2 \\
\text{To:} \quad & \epsilon = \zeta_1 \wedge (\zeta_1 = \zeta_2) \vee (\epsilon = \zeta_2 \wedge (\zeta_1 = \texttt{Dyn})) \vee (\epsilon = \zeta_1 \wedge (\zeta_2 = \texttt{Dyn}))
\end{aligned}$$

Reshape constraints.

$$\begin{aligned}
\text{From:} \quad & \texttt{can-reshape}([\![e]\!], (D_1, \ldots, D_m)) \\
\text{To:} \quad & [\![e]\!] = \texttt{Dyn} \vee \\
& ([\![e]\!] = \texttt{TensorType}(\epsilon_1) \wedge (\epsilon_1 = \texttt{Dyn} \vee \epsilon_1 \neq \texttt{Dyn} \wedge \epsilon_1 = D_1 \cdot \ldots \cdot D_n)) \vee \ldots \vee \\
& ([\![e]\!] = \texttt{TensorType}(\epsilon_1, \ldots, \epsilon_4) \wedge \\
& (\exists i \in \{1, \ldots, 5\} : \epsilon_i = \texttt{Dyn} \wedge \forall \epsilon_j \neq \texttt{Dyn} : D_1 \cdot \ldots \cdot D_m \ mod \ \prod \epsilon_j = 0))
\end{aligned}$$

$$\begin{aligned}
\text{From:} \quad & \texttt{can-reshape}(\texttt{TensorType}([\![e]\!], (D_1, \ldots, \texttt{Dyn}, \ldots, D_m))) \\
\text{To:} \quad & [\![e]\!] = \texttt{Dyn} \vee \\
& ([\![e]\!] = \texttt{TensorType}(\epsilon_1) \wedge \epsilon_1 = \texttt{Dyn} \vee \epsilon_1 \neq \texttt{Dyn} \wedge \epsilon_1 \ mod \ D_1 \cdot \ldots \cdot D_m = 0) \vee \ldots \vee \\
& ([\![e]\!] = \texttt{TensorType}(\epsilon_1, \ldots, \epsilon_4) \wedge (\exists i \in \{1, \ldots, 5\} : \epsilon_i = \texttt{Dyn})) \vee \\
& ((\forall i \in \{1, \ldots, 5\} : \epsilon_i \neq \texttt{Dyn}) \wedge \\
& (\prod_1^5 \epsilon_i \ mod \ D_1 \cdot \ldots \cdot D_m = 0 \vee D_1 \cdot \ldots \cdot D_m \ mod \ \prod_1^5 \epsilon_i = 0))
\end{aligned}$$

Convolution constraints.

$[\![e]\!] = \texttt{calc-conv}([\![e']\!], c_{out}, \kappa)$

First, from a previous constraint, we know that $[\![e']\!] \triangleright \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$

$$
\begin{aligned}
\text{From:} \quad & [\![e]\!] = \texttt{calc-conv}([\![e']\!], c_{out}, \kappa) \\
\text{To:} \quad & [\![e]\!] = \texttt{TensorType}(\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4) \wedge \\
& \epsilon_1 = \zeta_1 \wedge \\
& \epsilon_2 = c_{out} \wedge \\
& ((\epsilon_3 = \texttt{Dyn} \wedge \zeta_3 = \texttt{Dyn}) \vee \\
& (\zeta_3 \neq \texttt{Dyn} \wedge \epsilon_3 = ((\zeta_3 - 1) \cdot (\kappa[0] - 1) - 1) + 1)) \wedge \\
& (\epsilon_4 = \texttt{Dyn} \wedge \zeta_4 = \texttt{Dyn}) \vee \\
& (\zeta_4 \neq \texttt{Dyn} \wedge \epsilon_4 = ((\zeta_4 - 1) \cdot (\kappa[0] - 1) - 1) + 1))
\end{aligned}
$$

Broadcasting constraints.

$$
\begin{aligned}
\text{From:}\quad & \langle e_1\rangle, \langle e_2\rangle = \texttt{apply-broadcasting}(\llbracket e_1\rrbracket, \llbracket e_2\rrbracket) \\
\text{To:}\quad & (\llbracket e_1\rrbracket = \texttt{Dyn} \wedge \langle e_1\rangle = \llbracket e_1\rrbracket \wedge \langle e_2\rangle = \llbracket e_2\rrbracket) \vee \\
& (\llbracket e_2\rrbracket = \texttt{Dyn} \wedge \langle e_2\rangle = \llbracket e_2\rrbracket \wedge \langle e_1\rangle = \llbracket e_1\rrbracket) \vee \\
& (\llbracket e_1\rrbracket = \texttt{TensorType}(\epsilon_1) \wedge \ldots) \vee \ldots \vee \\
& \llbracket e_1\rrbracket = \texttt{TensorType}(\epsilon_2) \wedge \llbracket e_2\rrbracket = \texttt{TensorType}(\sigma_1, \sigma_2) \wedge \\
& \langle e_1\rangle = \texttt{TensorType}(\epsilon_1', \epsilon_2') \wedge \langle e_2\rangle = \texttt{TensorType}(\sigma_1', \sigma_2') \wedge \\
& \epsilon_1' = \sigma_1 = \sigma_1' \wedge \\
& (\sigma_2 = \epsilon_2 = \sigma_2' = \epsilon_2' \vee \sigma_2 = 1 \wedge \epsilon_2 \neq 1 \wedge \sigma_2' = \epsilon_2 = \epsilon_2' \vee \\
& \epsilon_2 = 1 \wedge \sigma_2 \neq 1 \wedge \epsilon_2' = \sigma_2 = \sigma_2') \\
& \vee \ldots \vee \\
& (\llbracket e_1\rrbracket = \texttt{TensorType}(\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4) \wedge \llbracket e_2\rrbracket = \texttt{TensorType}(\sigma_1, \sigma_2, \sigma_3, \sigma_4) \wedge \\
& \langle e_1\rangle = \texttt{TensorType}(\epsilon_1', \epsilon_2', \epsilon_3', \epsilon_4') \wedge \langle e_2\rangle = \texttt{TensorType}(\sigma_1', \sigma_2', \sigma_3', \sigma_4') \wedge \\
& ((\epsilon_1 = \sigma_1 = \epsilon_1' = \sigma_1') \vee ((\epsilon_1 = 1 \wedge \zeta_1 \neq 1 \wedge \epsilon_1' = \zeta_1 \wedge \zeta_1' = \zeta_1) \vee \\
& (\zeta_1 = 1 \wedge \epsilon_1 \neq 1 \wedge \zeta_1' = \epsilon_1 \wedge \epsilon_1' = \epsilon_1)) \vee \ldots \vee \\
& (\epsilon_4 = \sigma_4 = \epsilon_4' = \sigma_4') \vee ((\epsilon_4 = 1 \wedge \zeta_4 \neq 1 \wedge \epsilon_4' = \zeta_4 \wedge \zeta_4' = \zeta_4) \vee \\
& (\zeta_4 = 1 \wedge \epsilon_4 \neq 1 \wedge \zeta_4' = \epsilon_4 \wedge \epsilon_4' = \epsilon_4))))
\end{aligned}
$$

F Proof of the Order-Isomorphism

We will prove Theorem 4.1:

$\forall P : (\mathit{Mig}(P), \sqsubseteq)$ and $(\mathit{Sol}(\mathit{Gen}(P)), \leq)$ are order-isomorphic.

Proof. Let P be given; it remains fixed in the remainder of the proof. If φ is a function from type variables to types, then we define the function G_φ from programs to programs:

$$G_\varphi(x_1 : \tau_1, \ldots, x_n : \tau_n \;\texttt{return}\; e) \quad = \quad x_1 : G_\varphi(x_1), \ldots, x_n : G_\varphi(x_n) \;\texttt{return}\; e$$

Now we define the function α_P with the help of G_φ:

$$\alpha_P \quad : \quad (\mathit{Sol}(\mathit{Gen}(P)), \leq) \to (\mathit{Mig}(P), \sqsubseteq)$$
$$\alpha_P(\varphi) \quad = \quad G_\varphi(P)$$

We will show that α_P is a well-defined order-isomorphism. We will do this in four steps: we will show that α_P is well defined, injective, surjective, and order-preserving.

Well defined.

We will show that if $\varphi \in \mathit{Sol}(\mathit{Gen}(P))$, then $\alpha_P(\varphi) \in \mathit{Mig}(P)$.

Suppose $\varphi \in \mathit{Sol}(\mathit{Gen}(P))$. We must show

$$P \sqsubseteq \alpha_P(\varphi) \text{ and } \vdash \alpha_P(\varphi) : \texttt{ok}.$$

In order to show $P \sqsubseteq \alpha_P(\varphi)$, notice that P and $\alpha_P(\varphi)$ differ only in the type annotations of bound variables. If we have no bound variables in P, then $P = \alpha_P(\varphi)$. Otherwise, notice that for every declaration of $x : \tau$ in P, we have that $\varphi \models \tau \sqsubseteq x$ and $G_\varphi(x : \tau) = x : \varphi(x)$. So we know that $P \sqsubseteq \alpha_P(\varphi)$.

Suppose $P = \texttt{decl}^* \;\texttt{return}\; e$. Let Γ be φ restricted to the set of variables declared in $\texttt{decl}^*$.

In order to show $\vdash \alpha_P(\varphi) : \texttt{ok}$, we first show the more powerful property:

$$\forall e' \text{ subterm of } e : \quad \Gamma \vdash e' : \varphi(\llbracket e' \rrbracket).$$

We proceed by induction on e'.

Case: $e' = x$. Notice that $\varphi \models x = \llbracket x \rrbracket$ so use *t-var*.

Case: $e' = \texttt{reshape}(e_0, \delta)$. We have

$$\varphi \models \llbracket \texttt{reshape}(e_0, \delta) \rrbracket \quad = \quad \delta$$

and $\varphi \models \texttt{can-reshape}(\llbracket e_0 \rrbracket, \delta)$. By induction, we have $\Gamma \vdash e_0 : \varphi(\llbracket e_0 \rrbracket)$. Consider the definition of $\varphi \models \texttt{can-reshape}(\llbracket e_0 \rrbracket, \delta)$. We have that if $\texttt{Dyn}\mathit{doesnotoccurin}\delta \; \mathit{and} \; \varphi(\llbracket e_0 \rrbracket) \prod \delta = \prod \varphi(\llbracket e_0 \rrbracket)$ then we can use *t-reshape-s*. Otherwise, based on the occurrences of $\texttt{Dyn}$ in both $\varphi(\llbracket e_0 \rrbracket)$ and δ, we can use *t-reshape-g* or *t-reshape*.

Case: $\texttt{Conv2D}(c_{in}, c_{out}, \kappa, e_0)$. We have $\varphi \models \llbracket e_0 \rrbracket \rhd \texttt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$ and $\varphi \models c_{in} \sim \zeta_2$ and $\varphi \models \llbracket \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e_0) \rrbracket = \texttt{calc-conv}(\llbracket e_0 \rrbracket, c_{out}, \kappa)$. By induction, we get that $\Gamma \vdash e_0 : \varphi(\llbracket e_0 \rrbracket)$. Then we use *t-conv*.

Case: $e' = \texttt{add}(e_1, e_2)$. Notice that $\varphi \models \llbracket e_1 \rrbracket = \langle e_1 \rangle \sqcup^* \langle e_2 \rangle$ and $\varphi \models (\langle e_1 \rangle, \langle e_2 \rangle) = \texttt{apply-broadcasting}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket)$ and $\varphi \models \langle e_1 \rangle \sim \langle e_2 \rangle$ From the induction hypothesis we have $\Gamma \vdash e_1 : \varphi(\llbracket e_1 \rrbracket)$ and $\Gamma \vdash e_2 : \varphi(\llbracket e_2 \rrbracket)$. Now we use *T-Add*.

Injective.

We will show that if $\alpha_P(\varphi) = \alpha_P(\varphi')$, then $\varphi = \varphi'$.

Suppose $\alpha_P(\varphi) = \alpha_P(\varphi')$. From the definition of α_P we see that for every declaration $x : \tau$ in P we have $\varphi(x) = \varphi'(x)$. We will show that for every declaration $x : \tau$, $\varphi(x) = \varphi'([\![x]\!])$. Note that for every variable declaration $x : \tau$, we have that $\varphi \models \tau \sqsubseteq x$ and $\varphi' \models \tau \sqsubseteq x$ and since $\alpha_P(\varphi) = \alpha_P(\varphi')$ then $\varphi(x) = \varphi'(x)$.

Next we show that for every occurrence of a subterm e' in the return expression e, we have $\varphi([\![e']\!]) = \varphi'([\![e']\!])$, and for every occurrence of a subterm $\mathtt{add}(e_1, e_2)$, we have that $\varphi(\langle e_1 \rangle) = \varphi(\langle e_1' \rangle)$ and $\varphi(\langle e_1 \rangle) = \varphi(\langle e_1' \rangle)$. We proceed by induction on E'.

Case: $e' = x$, where x is bound in E. From $\varphi \models [\![e']\!] = x$ and $\varphi' \models [\![e']\!] = x$, we have $\varphi([\![e']\!]) = \varphi(x) = \varphi'(x) = \varphi'([\![e']\!])$.

Case: $e' = \mathtt{reshape}(e_0, \delta)$. From the induction hypothesis, we have the property $\varphi([\![e_0]\!]) = \varphi'([\![e_0]\!])$. From $\varphi \models \mathtt{can\text{-}reshape}([\![e_0]\!], \delta)$ and $\varphi' \models \mathtt{can\text{-}reshape}([\![e_0]\!], \delta)$ we have $\varphi([\![e']\!]) = \varphi'([\![e']\!]) = \delta$.

Case $e' = \mathtt{Conv2D}(c_{in}, c_{out}, \kappa, e_0)$. From the induction hypothesis, we have the property $\varphi([\![e_0]\!]) = \varphi'([\![e_0]\!])$. From $\varphi \models [\![e_0]\!] \rhd \mathtt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$ and $\varphi' \models [\![e_0]\!] \rhd \mathtt{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$, $\varphi \models c_{in} \sim \zeta_2$ and $\varphi' \models c_{in} \sim \zeta_2$ and $\varphi \models [\![\mathtt{Conv2D}(c_{in}, c_{out}, \kappa, e_0)]\!] = \mathtt{calc\text{-}conv}([\![e_0]\!], c_{out}, \kappa)$ and $\varphi' \models [\![\mathtt{Conv2D}(c_{in}, c_{out}, \kappa, e_0)]\!] = \mathtt{calc\text{-}conv}([\![e_0]\!], c_{out}, \kappa)$ we have $\varphi([\![e']\!]) = \varphi'([\![e']\!])$.

Case $e' = \mathtt{add}(e_1, e_2)$. From the induction hypothesis, we have $\varphi([\![e_1]\!]) = \varphi'([\![e_1]\!])$ and $\varphi([\![e_2]\!]) = \varphi'([\![e_2]\!])$. Then we have $\varphi \models (\langle e_1 \rangle, \langle e_2 \rangle) = \mathtt{apply\text{-}broadcasting}([\![e_1]\!], [\![e_2]\!])$ and

$$\varphi' \models (\langle e_1 \rangle, \langle e_2 \rangle) \quad = \quad \mathtt{apply\text{-}broadcasting}([\![e_1]\!], [\![e_2]\!])$$

and $\varphi \models \langle e_1 \rangle \sim \langle e_2 \rangle$ and $\varphi' \models (\langle e_1 \rangle, \langle e_2 \rangle) = \mathtt{apply\text{-}broadcasting}([\![e_1]\!], [\![e_2]\!])$. So we have that $\varphi([\![e']\!]) = \varphi'([\![e']\!])$.

Surjective.

We will show that if $P_0 \in \mathit{Mig}(P)$, then $\exists \varphi \in \mathit{Sol}(\mathit{Gen}(P))$ such that $P_0 = \alpha_P(\varphi)$.

From $P_0 \in \mathit{Mig}(P)$ we have $P \sqsubseteq P_0$ and $\vdash P_0 : \mathtt{ok}$.

Suppose $P_0 = \mathtt{decl}^* \ \mathtt{return} \ e$ and consider a derivation D of $\vdash P_0 : \mathtt{ok}$. We define φ as follows. First, for a variable x declared in $\mathtt{decl}^*$ with the declaration $x : \tau$, define $\varphi(x) = \tau$. Second, for every occurrence of a subterm e' of the return expression e, find the judgment in D of the form $\Gamma \vdash e' : \tau'$, and define $\varphi([\![e']\!]) = \tau'$. Then for the subterm e' of the form $\mathtt{add}(e_1, e_2)$ in e_0, find the use of *T-Add* for e' and in that use, find the equation $((\tau_1, \tau_2) = \mathtt{apply\text{-}broadcasting}(t_1, t_2)$, and define $\varphi(\langle e_1 \rangle) = \tau_1$ and $\varphi(\langle e_2 \rangle) = \tau_2$.

We must show that $\varphi \in \mathit{Sol}(\mathit{Gen}(P))$. First note that for every variable declaration $x : \tau$ we have that $\varphi(x) = \tau$.

Next, we will do a case analysis of the occurrences of subterms e' in the return expression e.

Case: $e' = x$, where x is bound in E. From *(t-var)* we have that $\varphi([\![e']\!]) = \varphi(x)$ so $\varphi \models [\![e']\!] = x$.

Case: $e' = \mathtt{add}(e_1, e_2) : \tau_1$. The derivation D contains this use of *T-Add*:

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad (\tau_1, \tau_2) = \mathtt{apply\text{-}broadcasting}(t_1, t_2) \quad \tau_1 \sim \tau_2}{\Gamma \vdash \mathtt{add}(e_1, e_2) : \tau_1 \sqcup^* \tau_2} \ \textit{(t-add)}$$

So, $\varphi([\![e_1]\!]) = \tau_1$ and $\varphi([\![e_2]\!]) = \tau_2$. By examining our constraints and the fact that $\alpha_P(\varphi) = G_\varphi(P) = P_0$, we are done. We know that $\alpha_P(\varphi) = G_\varphi(P) = P_0$ is that P_0 differs from P only in the type annotations of variable declarations.

Case $e' = \texttt{Conv2D}(c_{in}, c_{out}, \kappa, e)$. We consider the use of *T-Conv2D* and inspect the constraints and apply the reasoning above.

Case $e' = \texttt{reshape}(e', \delta)$. We consider the use of either *T-reshape-s*, *T-reshape* or *T-reshape-g* and inspect the constraints and apply the reasoning above.

Order-preserving.

We will show that if $\varphi \leq \varphi'$, then $\alpha_P(\varphi) \sqsubseteq \alpha_P(\varphi')$.

Suppose that $\varphi \leq \varphi'$ and let $P = x_1 : \tau_1, \ldots, x_n : \tau_n$ `return` e. We have

$$\begin{array}{rcl} \alpha_P(\varphi) & = & G_\varphi(P) & = & x_1 : G_\varphi(x_1), \ldots, x_n : G_\varphi(x_n) \ \texttt{return}\ e \\ \alpha_P(\varphi') & = & G_{\varphi'}(P) & = & x_1 : G_{\varphi'}(x_1), \ldots, x_n : G_{\varphi'}(x_n) \ \texttt{return}\ e \end{array}$$

From $\varphi \leq \varphi'$ and from *p-prog* and *p-decl*, we have $\alpha_P(\varphi) \sqsubseteq \alpha_P(\varphi')$. ◀