CheckedShapes

Mark Pock
markpock@uw.edu
University of Washington
Seattle, USA

Matthew Taruno
mtaruno@uw.edu
University of Washington
Seattle, USA

Michael Xu
xu.michael@gmail.com
University of Washington
Seattle, USA

1 Introduction

Shape error is a common class of errors in deep learning programs which occurs when applying operations on tensors of incompatible shape. We propose a tool which statically checks for shape error in PyTorch programs, a task we will call "shape checking" PyTorch programs. We compile this fragment to the dependently typed language and interactive theorem prover Lean 4. We then use Lean 4 metaprogramming to automate verification of the shape correctness of the program. We conclude – supporting findings by Lagouvardos et al. in [5] – that shape checking of imperative deep learning programs in commonly used languages is best understood as a separate, pluggable, interprocedural whole-program analysis rather than as an element integrable with their existing type systems. In addition, we conclude that an interactive theorem prover does not add any inherent value to a pluggable whole-program analysis, either in interpretability or in ease of bootstrapping computation. However, the methodology we explore here may contain, in the rare applications that are already inclined towards dependently typed languages, the kernel of the methodology required to design usable verified DSLs in said languages.

1.1 Defining shape error

Before proceeding further, we will define what we mean by shape error. First, let us consider a small mathematical example of shape error: adding the two vectors $\langle 1, 2, 3\rangle$ and $\langle 4, 1\rangle$. Since vector addition is defined elementwise, the formula $\langle 1, 2, 3\rangle + \langle 4, 1\rangle$ has no meaning. We say that vector addition requires shapes (k), (k) in its operands – both operands need to have k entries. More generally, shape error in a mathematical context occurs when attempting to apply operations on tensors which do not satisfy their shape requirements.

Let's consider how this might manifest when writing a deep learning program. Consider the formula for transformer attention,

$$\text{Attention}(Q, K, V) \coloneqq \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

(where Q, K, V are matrices with n rows and d columns). The following PyTorch code appears to be a straightforward implementation of the attention formula.

```
def attention(query, key, value):
    d_model = query.shape[-1]
    scores = (query@key.T) / (d_model**0.5)
    return F.softmax(scores, dim=-1) @ value
```

where @ stands for matrix multiplication. This succeeds when query, key, and value are all matrices with n rows and d columns, but a common practice in deep learning is to store lists of matrices as higher-rank tensors – a list of length b of tensors of shape (n, d) gets converted into a tensor of shape (b, n, d). We thus consider running this code when query, key, and value are all tensors with shape (b, n, d). We attempt an undefined matrix multiplication on the second line between query, which has shape (b, n, d), and key.T, which has shape (d, n, b). This is because PyTorch's transpose operation will reverse the shape (b, n, d) to the shape (d, n, b) when what is required is a tensor of shape (b, d, n). In particular, matrix multiplication between two tensors of rank $r > 2$ requires that the first $r - 2$ dimensions match and that the last two dimensions of the two tensors involved are m, n and n, d for some m, n, d.

1.2 Motivating shape error

A typical deep learning program consists in the definition of a model, preprocessing of data, running the model on the data, and any postprocessing on the result. Many deep learning programs also have a training component, in which a model is either trained from scratch or finetuned on domain-relevant data. Shape error can occur in all parts of of deep learning programs. Islam et al.'s comprehensive study on deep learning bug characteristics [2] identifies shape errors occurring within the definition and training of the model separately from shape errors occuring within preprocessing and postprocessing of data. Together, they comprise 45% of the observed bugs in TensorFlow code.

Deep learning computations can be expensive. Any runtime errors that occur after a significant portion of a computation has been performed (but before results of the computation have been saved) waste the developer time and CPU time that went into the computation. Even if developers run computations on smaller amounts of data (which they often do) before upscaling to their full datasets, the time spent generally remains non-negligible. Thus, shape errors become more devastating the later in the program they occur. A shape error which occurs in the final stages of the pipeline wastes all of the time and computation that went into the previous stages of the pipeline. Identifying shape errors statically means that developers will not have to wait until the final stages of the pipeline and the subsequent crash of their program to diagnose error. We thus pose the research question: "With the aid of Python annotations, can we statically check the correctness of tensor operations with respect to shape?"

1.3 Contribution

We attempt to compile PyTorch programs to the dependently typed language and interactive theorem prover Lean 4 in order to verify their shape correctness. We chose PyTorch over the other popular machine learning framework, TensorFlow, in order to verify the shape correctness of machine learning programs "at compile time" without resorting to analyzing a secondary computation graph (which TensorFlow explicitly creates and TensorFlow analyses use). We originally hypothesized that

(1) Tensor operations could be easily specified via dependent types in Lean 4.

(2) Lean 4 would be a target language which would not impose significant restrictions on the code being verified.
(3) Composed tensor operations could be easily verified for shape correctness through various forms of proof automation.
(4) The generated Lean 4 output would be more interpretable than comparable output from non-modular whole-program analyses (insofar as it would provide a human-readable proof of shape correctness).

We now argue that all hypotheses except the first hypothesis fail. In particular, we affirm the argument made by Lagouvardos et al. that shape checking is best characterized as a *context-sensitive, whole-program* analysis and that "A statically-typed TensorFlow-like system would either require significant programmer assistance for sound reasoning, or curtail the flexibility of operators, to permit assigning them closed-form types" [5] (likewise for PyTorch). We thus conclude that we *fail* to statically check the shape correctness of tensor operations with Lean 4. In particular, we claim that treating shape checking of machine learning code with the expressivity of modern machine learning DSLs as a type checking problem leads to the necessary of *term synthesis* for existential witnesses, a problem that proof automation in interactive theorem provers is ill-equipped to deal with.

We do make some few unique technical contributions from the perspective of Lean 4. First, we show the feasibility of interacting with Python code in Lean 4 through C shims, opening the door for Lean 4 to interoperate with widely used Python libraries. Second, we demonstrate that Lean 4 metaprogramming can be usefully decoupled from the Lean 4 parser and used as a high-level code generator. Third, we demonstrate potential hurdles that a useful shape-correct machine learning library in Lean 4 would have to overcome to be usable.

2 Related work

In this section, we explore a variety of approaches to shape checking deep learning code. Before we do so, we should note that not all approaches to reduce shape error involve shape checking – some approaches instead involve making shapes more explicit in code. For example, einstein-inspired notation provides an alternative notation for tensor operations which makes shape explicit, which couples shape to operations, reducing shape errors [8].

2.1 Dynamic approaches

Dynamic approaches in software analysis involve analyzing a program during its execution to understand its behavior, performance, and interactions within its environment. Unlike static analysis, which examines code without running it, dynamic analysis is beneficial because it is precise as through running the code, we are able to ensure no false positives.

First, deep learning frameworks often support debugging routines that enhance runtime assertion capabilities (e.g. tensorflow.debugging.assert_shapes) at function entry and exit. Dynamic type checkers are in a similar vein. Libraries such as typeguard [10] perform runtime asserts that type requirements are satisfied at function entries and exits, aided by annotation libraries such as jaxtyping [3].

ShapeIt [12] performs dynamic analysis to infer symbolic relationships between tensor shapes during runtime given concrete values. For example, given a multiplication of two tensors with shapes $(3, 4)$ and $(4, 5)$, ShapeIt outputs symbolic shapes (a, b) and (b, c)) respectively, highlighting the relationship between the dimensions. By capturing these symbolic relationships, ShapeIt can detect inconsistencies. For instance, if a subsequent operation incorrectly assumes that (b, c) is of a different shape, ShapeIt would identify this mismatch. This approach enables the detection of shape mismatches without requiring users to explicitly define shape specifications in their code.

ShapeFlow [11] is a dynamic abstract interpreter tailored for TensorFlow. It constructs a secondary shape computation graph before the actual TensorFlow computation graph, which it then verifies for shape correctness. By executing this shape-centric graph, ShapeFlow can identify shape incompatibility errors before the actual computation occurs. This method leverages TensorFlow's static graph construction, allowing for pre-execution shape verification. However, implementing a similar approach in frameworks like PyTorch, which utilize eager execution, would be challenging and might necessitate replicating TensorFlow's graph-based execution model.

2.2 Static approaches

Static approaches to shape checking have largely fallen into two camps. First, we consider whole-program analysis tools which use solvers to validate all instances of primitive operations (e.g. matrix multiplication itself; more generally, the basic library routines provided by machine learning frameworks). Since many tensor operations depend on values produced far away in the code, these tools can certify a program correct in cases where the correctness is highly contingent on the particular shapes of tensors being passed into functions. Second, we consider tools which broadly rely on using powerful type systems to validate not only primitive operations but also more complex operations composed from these primitives. These tools provide extensibility and modularity.

We highlight two whole-program analysis tools with a similar flavor, one for PyTorch and one for TensorFlow. Pythia [5] (which runs over Tensorflow programs) performs a whole-program shape analysis building on a variety of interrelated other analyses (e.g. pointer analysis) expressed as a series of Datalog rules for each PyTorch operator. We originally hypothesized that we might challenge the claim made in [5] that a statically typed system capturing the nuances of modern machine learning code would be overly constraining and require programmer assistance. However, we now affirm the findings of [5]. In particular, in attempting to verify machine learning code, we find ourselves enmeshed in problems deeply reminiscent of logic programming which make the Datalog-based approach of Lagouvardos et al. extremely attractive.

PyTea [4] (which runs over PyTorch programs) generates constraints on paths that lead to successful execution of PyTorch programs without shape error, then sends these constraints to the SMT solver Z3. We highlight PyTea largely because it fills a void for PyTorch programs. Because PyTea examines all possible paths and generates verification conditions by abstractly executing the

machine learning code, PyTea cannot handle programs with unbounded loops or recursion, and in fact unrolls all loops to obtain pure straight-line code. We originally hypothesized that a type system approach could overcome this problem. However, the polymorphism required for an effective type system approach requires significant programmer annotations.

Previous research has often explored powerful type systems for tensor shape checking in languages outside of Python. For example, Hattori et al. [1] propose a gradually typed tensor shape inference and shape checking tool that operates over OCaml programs. Similarly, Stites et al. use gradual refinement types in Haskell to shape check tensor programs [9]. However, Python remains the lingua franca of the deep learning community, so integrating the benefits of these tools into everyday deep learning use remains challenging.

Variadic generics were introduced in 2022 in Python 3.11 through PEP 646, enabling the definition of classes and functions that can accept an arbitrary number of type parameters. This feature is particularly beneficial for annotating tensor shapes in numerical computing libraries, as it allows for more precise type definitions that can express the dimensions of arrays or tensors [6]. However, as of now, static type checkers like mypy do not fully support variadic generics, limiting its practical utility for static type checking in deep learning applications. Nevertheless, variadic generics have found some success in dynamic type checking. To use a dynamic typechecker to preemptively identify shape error at runtime, users annotate the relevant function with types like Tensor[..., 1, 2, 3]. Tools like typeguard are subsequently able to verify that tensors passed to the function actually do have these shapes.

Finally, we highlight one tool which does attempt a typechecking approach in Python. Roesch et al. [7] introduces Relay, a functional and statically typed intermediate representation (IR) designed to serve as an alternative computational graph to the Apache TVM backend. Relay uses dependent types to express the types of tensor computations, as we plan to do. However, we feel that Relay has a similar flavor to ShapeFlow [11] in the sense that both rely on shape-checking a secondary graph before the execution of the program. We aimed to verify shape correctness without explicitly constructing a secondary graph.

3 Approach

In this section, we present an approach for shape checking PyTorch code. We first compile PyTorch to Lean 4 via a fairly straightforward syntactic translation. The nature of our syntactic translation already makes many Python features difficult to express, and so they are not implemented in our artefact. Second, we apply Lean 4 metaprogramming to verify the resultant code as Lean 4 code through automated proof.

3.1 Research question

PyTorch and TensorFlow are two of the most common deep learning frameworks today. Traditionally, PyTorch programs are executed *eagerly* while TensorFlow programs are executed *lazily*. Since Python is an eagerly evaluated language, this means that a TensorFlow expression evaluates to a node in a global *computational graph*. The computational graph itself is then evaluated when necessary. Tools that operate over TensorFlow programs have been able to leverage the computational graph to great effect (e.g. ShapeFlow [11]). More broadly, tools that operate by wrapping a secondary layer around evaluation can then perform analysis right before evaluation, thus avoiding the worst costs of shape errors. Relay [7] takes this approach as well. This computational graph approach has proved to be viable for lazily evaluated deep learning programs. However, eagerly evaluated deep learning programs (such as PyTorch programs) demand different strategies. We attempted to present a methodology that works for eagerly executed programs, and so chose to focus on PyTorch programs. Furthermore, we were interested in the potential of dependent types for easy specification of tensor operations; and in the potential of interactive theorem provers for interpretable verification proofs. We thus refine our previously posed research question in (§1.2) – "can shape error be statically checked in PyTorch programs using dependently typed specifications?"

3.2 Dependent types

First, we will present the platonic ideal for shape checking PyTorch code with dependent types. We will first define some terms. For our purposes, a tensor is a multidimensional array. The shape of the tensor is a nonempty list of positive numbers where each entry gives the size of the ith dimension. For example, a tensor with shape $[2, 6]$ is a 2×6 array; a tensor with shape $[5, 1, 3]$ is a $5 \times 1 \times 3$ array. The rank of the tensor is the length of its shape (i.e. its number of dimensions). Now, consider the mathematical definition of matrix multiplication. The operation is well-defined exactly when the inner dimensions of its two arguments are the same. Then, the result of the matrix multiplication is now a matrix with the outer dimensions of both arguments.

We will now consider how to formalize this operation in a *dependently typed language*. In most programming languages, values are allowed to depend on values through the usual notion of function (which takes a value and output a value). The successor function is one such function, and its type may be written as

$$\mathsf{successor} : \mathbb{N} \to \mathbb{N}$$

Many programming languages also have constructions through which values are allowed to depend on types: polymorphism – one writes a function that takes a type as an argument and outputs a value, e.g. the identity function, whose type we write

$$\mathsf{identity} : \forall \alpha : \mathsf{Type}, \alpha \to \alpha.$$

Many programming languages also have constructions in which types are allowed to depend on types – generic types. For example, consider a function that takes a type α and returns the type of lists of this type $\mathsf{List}\ \alpha$ (this function is in fact just List itself). We write the type as

$$\mathsf{List} : \mathsf{Type} \to \mathsf{Type}.$$

The last kind of dependency, and the kind which concerns us most here, is the dependency of types on values. This is a much less familiar construction, but the pattern is the same. Consider a function that takes a number $n : \mathbb{N}$ and returns the types of lists of natural numbers *of length n*. Call this function NatVec. We write the type of this function as

$$\mathsf{NatVec} : \mathbb{N} \to \mathsf{Type}.$$

Now, consider an elementwise sum between a pair of vectors of naturals, which requires that both arguments have the same length. We can now express this constraint as

$$\text{ElementwiseSum} : \forall n, \text{NatVec}(n) \to \text{NatVec}(n) \to \text{NatVec}(n).$$

We now consider a type constructor $\text{Tensor} : \text{List}(\mathbb{N}) \to \text{Type}$. We can then express the matrix multiplication constraint that the inner dimensions must match as

$$\text{Matmul} : \forall m, n, d : \mathbb{N},$$
$$\text{Tensor}([m, n]) \to \text{Tensor}([n, d]) \to \text{Tensor}([m, d]).$$

With this representation, it now becomes clearer why dependent types are necessary: while we might imagine representing Tensor as an ordinary generic type, we would be unable to represent the dimension constraints on matrix multiplication at the type level. We require information about the dimensions of tensors at the type level in order to represent tensor operations.

In fact, the matrix multiplication operation commonly used in deep learning code is more expressive than the matrix multiplcation given before. This is because it admits tensors of any rank, and treats the product as a batched matrix multiply done by iterating over the first $\text{rank} - 2$ dimensions and computing the usual matrix product in the last 2 dimensions. We can now specify this operation as follows. This operation takes two tensors with all their dimensions except the last two identical. Then, if the last two dimensions of the first tensor are m and n and the last two dimensions of the second tensor are n and d, the two tensors admit a multiplication. This can be represented as

$$@ : \forall m, n, d : \mathbb{N}, \forall v : \text{List}(\mathbb{N}), \text{Tensor}(v +\!\!+ [m, n]) \to$$
$$\text{Tensor}(v +\!\!+ [n, d]) \to \text{Tensor}(v +\!\!+ [m, d])$$

where $+\!\!+$ is the list concatenation operation. We will call operations like these which do not constrain the rank of their arguments *rank-polymorphic* operations. We will see that rank-polymorphism (which is a critical part of modern deep learning APIs) poses a problem for our type system approach to verifying shape correctness of tensor operations.

In this Platonic ideal for shape checking PyTorch code, having specified matrix multplication, we can now compose it trivially, and the typechecker will without further effort either certify that our code is shape correct or will show us that it fails to be shape correct at a particular location. For example, when multiplying tensors of shapes $[3, 4, 2]$ and $[3, 2, 5]$, we should expect to get a result of $[3, 4, 5]$. When multiplying tensors of shapes $[3, 4, 2]$ and $[2, 2, 5]$; we should expect to see that no v could be found such that $[3, 4, 2] = v +\!\!+ [4, 2]$ and $[2, 2, 5] = v +\!\!+ [2, 5]$. We will see that this ideal fails in §4.2, and we will explore a number of responses to its failure.

4 Pipeline

We will now go through the structure of our pipeline in more detail and consider the challenges which we faced in attempting to verify the shape correctness of code in the manner supposed by § 3.2. We also elaborate a few technical contributions which we make from the perspective of software development in Lean 4 – showing the feasibility of Python-Lean interop, showing that the Lean 4 parser and elaborator can be usefully decoupled to use the Lean 4 elaborator as a code generator, and illustrating challenges that face any practical Lean 4 machine learning DSL.

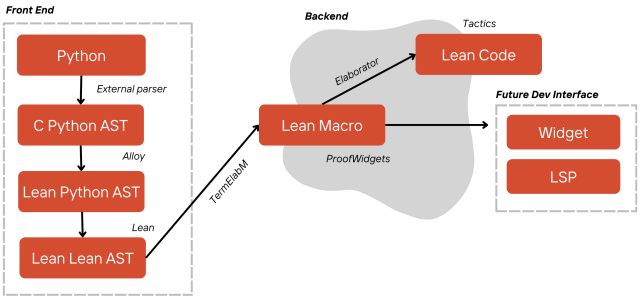

Figure 1: End-to-end pipeline for CheckedShapes. We start with the Python AST library to parse Python code. Subsequently, we translate the Python data structure into an appropriate Lean data structure via the C FFI. In particular, we leveraged Alloy, which embeds C into Lean. Finally, we leverage Lean metaprogramming to translate the data structure into Lean code, which will either typecheck or not typecheck.

4.1 Parsing and code generation

We only consider a limited subset of Python code for verification. We consider assignment statements, for loops, and function calls; and suppose that functions are in a normal form with one return statement closing the body of the function. We assume that annotations are given on the function parameters and possibly the return type. We expect annotations given on function parameters to be relatively simple – usually just constraining the minimum rank of input tensors and naming the upper or lower dimensions. However, annotations on the return type may be quite complicated, as the shapes of the results of tensor operations may not always be expressible in a simple way, especially for rank-polymorphic functions. For example, the transpose operation creates a new tensor whose shape is the reverse of the same of the original. If the rank of the original tensor with shape s is unknown, the best we can do is to say that its new shape is $\text{reverse}(s)$ (where reverse is a purely formal expression).

We use the `ast` Python library to parse Python code. We then interoperate with it through the Lean 4 foreign function interface. In particular, we use the Lean 4 DSL Alloy to write inline C in Lean (approximately speaking). It is worth reflecting on some other options for parsing Python code and the advantages or disadvantages they might confer.

(1) *Using an existing Python parser and interoperating with it through the FFI.* As noted above, this is the option we ultimately chose. Its primary benefit lies in being able to piggyback off code written in other languages – here, parsers, but an additional benefit would be in conducting an initial stage of analysis in an existing analysis framework, then sending these results to Lean 4. This horizon in particular would make this option the right choice if this line of work were continued, given the challenges we faced when conducting analysis entirely in Lean.

(2) *Using the Lean parser to parse Python as a DSL.* The Lean 4 parser is user-extensible – users can declare new syntactic categories for it to parse, which enables creating embedded DSLs through metaprogramming. We earlier mentioned *Alloy*, a Lean 4 DSL which enables writing inline C and expedites Lean-C interop. This strategy would take the same approach as Alloy – create a Lean 4 DSL which enables writing inline Python. For the scope of our project, this strategy would actually have been the best strategy. It would enable us to leverage the Lean 4 UI, including the infoview and error handling. Moreover, it would allow us to soundly enforce the constraints we place on Python code, including both the constraints we explicitly afford for above and the constraints which arose in the course of work. However, this would hamper any possible adoption by Python developers, since instead of sitting as a standalone analysis tool, this tool would be deeply enmeshed in the Lean 4 ecosystem.

(3) *Writing a Python parser from scratch in Lean.* We initially supposed that writing a Python parser from scratch would be a relatively uninteresting (with respect to the aims of the project) and ultimately extraneous task. However, given the amount of work that went into writing the C shim between the Python parser and the constraints we impose on Python code, this was probably the wrong determination. For our project, either this or the above option would have been better suited to our scope and goals.

As alluded to, we dwell on the method for our frontend largely because a) it consumed an unforeseen amount of work and b) because it points towards a development of technical relevance for Lean 4 developers – the possibility of effectively interoperating with Python, thus gaining access to widely used libraries.

We then use Lean 4 metaprogramming techniques to generate Lean 4 code which is roughly one-to-one syntactic translation of the input Python code. As mentioned above, the Lean 4 parser is user-extensible, and so Lean 4 provides extensive metaprogramming capabilities that allow users to write DSLs. Here, however, we use Lean 4 metaprogramming in a different way – instead of elaborating fragments of Lean 4 code parsed by the Lean 4 parser, from the perspective of Lean 4, we generate Lean code ex nihilo. Rather, we should say that we decouple the Lean 4 parser and the Lean 4 elaborator, and show that Lean 4 metaprogramming can be used with an alternative frontend (here, the Python `ast` library).

Here is an example of the "roughly one-to-one syntactic translation" between Python and Lean which we create using metaprogramming. The Python program

```
def f(
    a : 'Tensor (4, 5)',
    b : 'Tensor (4, 6)',
    c : 'Tensor(6, 5)'
) -> 'Tensor (4, 5)':
    d = b @ c
    e = a + d
    return e
```

compiles to the Lean program

```
def f :
    Tensor (Shape.append (Shape.lift 4)
        (Shape.append (Shape.lift 5)
        (Shape.nil))) ->
    Tensor (Shape.append (Shape.lift 4)
        (Shape.append (Shape.lift 6)
        (Shape.nil))) ->
    Tensor (Shape.append (Shape.lift 6)
        (Shape.append (Shape.lift 5)
        (Shape.nil))) ->
    Tensor (Shape.append (Shape.lift 4)
        (Shape.append (Shape.lift 5)
        (Shape.nil))) :=
fun a b c => Id.run do
    let mut d := Tensor.PrimOp.matmul b c
    (by resolvePrimOp)
    let mut e := Tensor.PrimOp.add a d
    (by resolvePrimOp)
    return e
```

There are several features of note which will become important when exploring how we attempted to verify shape correctness.

(1) *Annotation syntax.* We see that the parameters to f are annotated with *strings*, not Python type expressions. This is in order to support an annotation syntax which can express, e.g. the concatenation of shapes; reversing a shape; etc. This has the unwanted side effect of making our annotations not compatible with any existing annotations that use the type expression for Tensor with variadic generics already.

(2) *Deep rather than shallow embedding of shapes.* We then have to consider how to embed this syntax into Lean. In §3.2, we presented a Platonic ideal of shape checking which directly parameterizes tensors over lists as shapes – a shallow embedding. We found it considerably more expedient to work with a deep embedding – representing the allowable operations on lists as abstract syntax in a Shape type.

(3) *Monadic embedding.* We earlier hypothesized that Lean 4 could provide a "one-to-one" syntactic equivalent of Python in many respects. This is true, up to a point, but is only possible when using the syntactic features of do-notation, which requires working inside a monad. Here, we use the identity monad; we will explore complications of this choice in the next section.

(4) *Additional argument to operations.* The one major respect in which this code differs from Python is in the required third argument to matmul and add. This argument, given to both as `by resolvePrimOp`, attempts to explicitly construct the proof of shape correctness.

Regardless of the success or failure of the verifier, we show that we are able to specify tensor operations via Python shape annotations and translate an appropriate subset of Python in a faithful way to Lean 4.

4.2 Verification

In §3.2, we presented a Platonic ideal of shape checking in which the type checker would be able to automatically infer, at least in simple cases, the result type of a tensor operation. In fact, this Platonic ideal fails immediately, even for the most simple case involving the batched matrix multiply. Once again consider the type signature for the batched matrix multiplication

$$@ : \forall m, n, d : \mathbb{N}, \forall v : \mathsf{List}(\mathbb{N}), \mathsf{Tensor}(v +\!\!+ [m, n]) \to \mathsf{Tensor}(v +\!\!+ [n, d]) \to \mathsf{Tensor}(v +\!\!+ [m, d])$$

If we consider the very simple example of multiplying a tensor with shape $[3, 2, 4]$ by a tensor with shape $[3, 4, 5]$, we ourselves can immediately see that $v = [3]$, $m = 2$, $n = 4$, and $d = 5$. However, this is not an expression that the Lean 4 compiler can pattern match into. More appropriately, we might say that Lean 4 kernel does not attempt to solve the unification problem given by $[3, 2, 4] = v +\!\!+ [m, n]$, $[3, 4, 5] = v +\!\!+ [n, d]$. We might reasonably ask why not. We can tweak the example slightly. Suppose we are multiplying a tensor with shape $(\ell +\!\!+ [a, b]) +\!\!+ [2, 4]$ by a tensor with shape $\ell +\!\!+ ([a, b] +\!\!+ [4, 5])$. We might conclude that these can obviously be multiplied. However, it is not so obvious – if we want to assign anything to v, say $\ell + +[a, b]$, we can't simply normalize the terms and check if they are equal, because the variable ℓ stops evaluation from happening. In order to actually determine that they are equal, we need to know that list concatenation is associative, which the Lean typechecker does not know.

In order, then, that the Lean 4 typechecker not reject every single matrix multiplication we write, we need to take the explicit restrictions out of the type signature. How, then, will we enforce shape correctness? We require an additional proof term which guarantees shape correctness, which requires the user (rather, the prover – it may just as well be a tactic script) to prove the equalities which the Lean 4 type checker could not. We thus exhibit the modified type signature[1]

$$\begin{aligned} @ : {} & \mathsf{Tensor}(S_1) \to \mathsf{Tensor}(S_2) \to \\ & (h : \exists, m, n, d : \mathbb{N}, S_1 = l +\!\!+ [m, n] \wedge S_2 = l +\!\!+ [m, d]) \to \\ & \texttt{let } \langle l, m, \_, d, \_\rangle = h \texttt{ in} \\ & \mathsf{Tensor}(l +\!\!+ [m, d]) \end{aligned}$$

Of course, this proof term does not live in the Python code. We are thus required to show this to our satisfaction ourselves – in particular, by using tactics. As an interactive theorem prover, Lean 4 generates proofs largely by way of user-supplied tactics, which provide sound ways of manipulating hypotheses and goal states to eventually reach a trivial goal. Tactics can also be used to generate tactics in response to different goals in the environment. These higher-order tactics (sometimes called tacticals) are the method most commonly used in dependently typed languages for proof automation.

A few issues now arise. First, the problem of whether one list is equal to another without knowing the specific structure of the list is not necessarily decidable in general (rather, the problem of when two list-valued expressions with free variables denote the same list). So we cannot hope to decide the issue – and we cannot hope to use the resources Lean has for decidability, either. Second, proving existence claims is hard on two levels. First, it is genuinely hard in the sense that finding a witness to a proof – rather, synthesizing a witness to a proof – is very much a contingent problem which is hard to tackle in generality (and even hard to tackle in specificity, depending on the property to be proved). Second, it is made even harder by the fact that Lean 4 proof infrastructure is not designed to find witnesses for existential proofs. More generally, Lean 4 proof infrastructure is not designed to unify at all – and this problem is precisely a unification problem!

We should, however, note that the unification problem is tractable in a particular case – when the operations are not rank-polymorphic. An arbitrary list can be decidably matched into an expression with the form of a concrete list with a predetermined length – for example, the proposition $\exists a, b, c, x = [a, b, c]$ is decidable for any particular x. However, rank-polymorphism is important to the expressivity of modern deep learning code, so it is something we are hesitant to jettison; jettisoning rank-polymorphism does represent an alternative approach which hews away from the flavor of dependently typed specification.

We first attempted to use the primary Lean 4 tactic for proof search in a proof database, `aesop`, but before even reaching problems of scale, we found ourselves frustrated by the same issues as with the kernel with respect to unification. Even equipped with all the equalities required to produce a witness, `aesop` will not produce a witness. Consider the matrix multiplication of two tensors with shapes $[2, 3]$ and $[3, 4]$. To generate a witness satisfying the existential in the type specification above, all we need is the theorem that $[] +\!\!+ [2, 3] = [2, 3]$. Even explicitly equipped with this theorem, `aesop` would not solve the goal. More generally, most Lean 4 tactics do not aid in solving existential goals: they are resistant to instantiating metavariables.

Unable to modify these tactics to instantiate metavariables in obvious cases, we attempted a different approach – consider a *deep embedding* of shape expressions. Up until now, we have been considering a shallow embedding of shape expressions in our type signatures where shape expressions are just lists. However, we wanted to be able to use computation to heuristically solve simple cases.[2] The methodology this points to, however, is fundamentally a different methodology than that of verification by compilation to a dependently typed language – we would, in essence, be writing the kernel of our own dependently typed language, purpose built to solve shape checking problems.

This concludes our exploration of the struggles faced in *verifying* the shape correctness of generated code; it is apt to say that unification is the fundamental problem, and that rank-polymorphism exacerbates the problem of unification exceedingly.

5 Discussion

5.1 Evaluation

Because we remain unable to verify any code which is not annotated in a way that syntactically matches the structure of the existential proof, we cannot evaluate our verifier against any of the bugs

[1] Where $\exists$ below has computational content; i.e. is an intuitionistically provable $\exists$ (more properly is a dependent sum Σ), so as to legitimize eliminating out of the exists in the return type.

[2] At time of writing, this was our approach; it remains unfinished.

described in the literature (e.g. cases evaluated in [4]). We can only evaluate code generation on these tasks. Being largely focused on various attempts to a) create interoperability with Python and b) get a functional verifier, we cannot generate code for every Python language feature and every PyTorch function as of yet.

For PyTorch functions, it is largely a matter of writing down the specifications in whatever form should be possible for a verifier to actually verify. For Python language features, this is largely just an engineering concern which can to some extent be handled via monads (try/catch can be handled by monads, decorators can be handled by monads, while loops don't require any additional monadic structure). Notably, we properly compile local mutability, but not global mutability – we believe this can also be handled via additional monadic structure. We also have difficulty dealing with reference semantics (because Lean is naturally immutable), but because tensor code is generally immutable, this does not strike us as a problem (and again begs a solution with additional monadic structure). Both decorators and global mutability take us further away from the notion of a one-to-one syntactic transformation – a more significant apparatus is required to effectively translate either.

To summarize, at present, we can generate code with no tensor computational content for the subset of Python previously described – Python with for loops, assignments, function calls, list expressions, tensor expressions (zeros, ones, random, randn), some fundamental tensor operations (matrix multiply, add, transpose) and local mutability. This code cannot be verified as shape correct by the Lean 4 kernel unless annotations and type coercions are added to the subsequently generated Lean 4 code.

5.2 The character of shape checking

We have several times alluded to an argument made by Lagouvardos et al. [5] that shape checking should be regarded as a context-sensitive whole-program analysis. Though we initially attempted to challenge their findings, we conclude in support of them in facing fundamental problems with a type-based approach. Broadly speaking, the problems we faced when trying to apply a type-based approach to verifying deep learning code all came from *holes* in information. More concretely, the polymorphism required to appropriately type PyTorch constructs (rank-polymorphism in particular) creates metavariables which block evaluation of shape transformations and require non-automatic rewrites and insights to instantiate. When Lagouvardos et al. find multiple possible concrete shapes for a tensor at a call site, rather than abstracting away, they merely retain both shapes and proceed to verify shape correctness for both. Fundamentally, Lagouvardos et al. deal with multiplicity through accumulating helpful information, which a type-based approach precludes by abstracting away this information.

We further note that the logic programming approach taken by Lagouvardos et al. really appears to be the right way to solve the problem. In attempting to instantiate existential variables, we find ourselves asking relational queries explcitly expressed as such – what shape expressions C satisfy the relation of elementwise compatability between the two shape expression inputs A and B? This should not be surprising, insofar as the fundamental challenge we face is unification. We are still open to the possibility of a specialized unifier to deal with the theory of shape expressions in the general first-order form that we find it (not just in the concrete form given by Lagouvardos). This possibility is a fundamental departure from the notion of analysis by compilation to a dependently typed language, which takes as its starting position the idea that the kernel of the dependently typed language will be central to unification and type-checking. Accordingly, this possibility is not in any way inherently aided by implementation in a dependently typed language; in fact, the most natural expression of a specialized unifier for polymorphic shape types is likely to be found in a logic program such as Lagouvardos et al. develop.

Ultimately, we conclude just this: the unification problems we faced were the result of purposely discarding information about the concrete shape of tensors in an actual execution for the purpose of a type system paradigm. This should seem to indicate that a type system paradigm is not the right way to go about shape checking.

6 Conclusion

Deep learning programs are particularly susceptible to shape errors, which can cause costly runtime failures and hinder model deployment. In this work, we explore using dependent types to statically verify the correctness of tensor shapes in PyTorch programs. In the process, we make a few technical contributions from the perspective of software development in Lean 4, most notably exploring the horizon of interoperability between Lean 4 and Python.

We ultimately conclude that while specifying tensor operations with dependent types is possible and even desirable, compilation to a *dependently typed language* to *verify* tensor operations is a flawed idea. The notion of compilation to a dependently typed language suggests that the unifier of a dependently typed language could deal with the semantic concerns necessary to verify even the simplest tensor operation. Instead, the problem of verifying dependently typed tensor operations should be met by a semantic unifier expressly built to deal with the operations on lists which correspond to tensor shape operations. This unifier can be constructed in any language.

However, beyond an alternative approach for verifying dependently typed tensor expressions, we conclude, affirming arguments made by Lagouvardos et al. in [5], that shape checking of modern deep learning code should not be understood at all as a modular type checking problem but rather as a context-sensitive whole-program analysis. This is because the fundamental tensor operations are rank-polymorphic, and rank-polymorphism creates a difficult term synthesis problem which may strain at the bounds of what is possible with semantic unification. Having concrete data about tensor shapes at every call site vastly decreases the complexity of the problem of verifying tensor shape correctness.

We must, in the end, echo the conventional wisdom about dependent types: programming with dependent types is hard. Yet limiting the scope by focusing on dependently typed DSLs may still prove a useful technique.

References

[1] Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato. 2023. Gradual Tensor Shape Checking. arXiv:2203.08402 [cs.PL] https://arxiv.org/abs/2203.08402

[2] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia)

(ESEC/FSE 2019). Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[3] Patrick Kidger. 2024. Jaxtyping: Type annotations for JAX, PyTorch and TensorFlow. https://github.com/patrick-kidger/jaxtyping

[4] Jinhan Kim, Seulbae Kim, and Sukyoung Ryu. 2022. PyTea: Practical Tensor Shape Analysis for Deep Learning. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. ACM, 1013–1027. https://doi.org/10.1145/3510454.3528638

[5] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[6] Mark Mendoza. 2020. PEP 646 – variadic generics. https://peps.python.org/pep-0646/

[7] John Roesch, Tianqi Chen, Jared Roesch, Ziheng Jiang, Trevor L. M. Howard, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Relay: A New IR for Machine Learning Frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2018)*. ACM, 58–68. https://doi.org/10.1145/3211346.3211348

[8] Alex Rogozhnikov. 2022. Einops: Clear and Reliable Tensor Manipulations with Einstein-like Notation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=oapKSVM2bcj

[9] Sam Stites and Austin Huang. 2018. Hasktorch: A Comprehensive Haskell Library for Differentiable Functional Programming. In *International Conference on Functional Programming (Numerical Programming in Functional Languages)*.

[10] L. V. van der Palen. 2024. typeguard: Runtime Type Checking for Python. https://github.com/agronholm/typeguard

[11] Sahil Verma and Zhendong Su. 2020. ShapeFlow: Dynamic Shape Interpreter for TensorFlow. *arXiv preprint arXiv:2011.13452* (2020). https://arxiv.org/abs/2011.13452

[12] Dan Zheng and Koushik Sen. 2024. Dynamic Inference of Likely Symbolic Tensor Shapes in Python Machine Learning Programs. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice* (Lisbon, Portugal) *(ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 147–156. https://doi.org/10.1145/3639477.3639718