TeIL: A Type-Safe Imperative Tensor Intermediate Language

Norman A. Rink
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
norman.rink@tu-dresden.de

Jeronimo Castrillon
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Abstract

Each of the popular tensor frameworks from the machine learning domain comes with its own language for expressing tensor kernels. Since these tensor languages lack precise specifications, it is impossible to understand and reason about tensor kernels that exhibit unexpected behaviour. In this paper, we give examples of such kernels.

The tensor languages are superficially similar to the well-known functional array languages, for which formal definitions often exist. However, the tensor languages are inherently imperative. In this paper we present TeIL, an imperative tensor intermediate language with precise formal semantics. For the popular tensor languages, TeIL can serve as a common ground on the basis of which precise reasoning about kernels becomes possible. Based on TeIL's formal semantics we develop a type-safety result in the Coq proof assistant.

CCS Concepts • **Software and its engineering → Semantics**; *Domain specific languages.*

Keywords tensor frameworks, tensor kernels, tensor expressions, formal specification, formal proof, Coq

ACM Reference Format:
Norman A. Rink and Jeronimo Castrillon. 2019. TeIL: A Type-Safe Imperative Tensor Intermediate Language. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3315454.3329959

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ARRAY '19, June 22, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6717-2/19/06…$15.00
https://doi.org/10.1145/3315454.3329959

1 Introduction

In recent years, many frameworks have appeared that support the development of applications in which tensors are the central data structures [2, 3, 5, 16, 18, 19, 26, 27, 30]. Most of these frameworks target the machine learning domain, and one main motivation for using such a framework is the promise of an easy route to efficient application deployment on parallel and heterogeneous platforms. Each framework typically comes with its own programming language, a *tensor language* designed specifically for expressing tensor kernels.

Tensors are multi-dimensional arrays for which certain natural *collective* (viz. *global*) operations are defined. Collective operations can be expressed without explicitly indexing into an array or tensor. Prominent examples of collective operations are (1) *element-wise* arithmetic and (2) the *reduction* of a tensor along a given dimension. Writing programs with collective operations, as is possible for many relevant tensor kernels in the machine learning domain, has advantages for program analysis and code generation: the absence of explicit indexing eases analyses and can thus help with parallelization, and optimization in general.

Superficially, tensor languages have much in common with array languages. There is, however, one fundamental difference between the array languages from the literature [9, 12, 13, 15, 22, 23, 25] and the tensor languages from recent frameworks: while most array languages are functional, the tensor languages are inherently imperative. This is mostly unsurprising for two reasons. First, many of the tensor languages appear to be, at least syntactically, dialects of Python. Second, many commonly used platforms for heterogeneous computing are directly programmable through (relatively low-level) imperative languages.

While typed functional languages generally enjoy type-safety properties, which have been re-stated and proven for functional array languages [9, 24, 29], to the best of our knowledge, no results on type-safety have appeared for imperative tensor languages. Of a type-safe imperative tensor language one would expect that no out-of-bounds accesses occur during the execution of a well-typed tensor kernel. This can of course be ensured by inserting dynamic bounds checks [11, 12, 22]. Indeed, when arbitrary expressions are allowed as indices into tensors, the absence of out-of-bounds accesses is impossible to establish statically. However, when

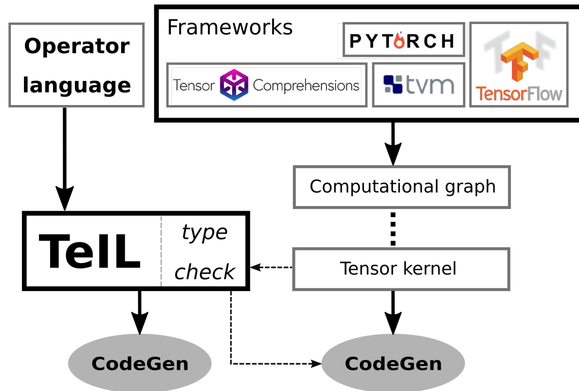

Figure 1. Guarding code generation with TeIL.

only collective tensor operations are allowed, static type-safety should hold.

In this paper we propose TeIL, a *Tensor Intermediate Language* that is imperative and type-safe, in the sense that there are no out-of-bounds accesses in well-typed TeIL programs. TeIL is intended to be a smallest common denominator of recent imperative tensor languages, and of newly emerging ones. As such, we do not expect that all programs that can be expressed in any of the commonly used tensor languages can also be lowered to TeIL.[1] Instead, successfully lowering a tensor kernel to TeIL should be considered a guarantee that executing the kernel does not cause out-of-bounds accesses.

Although we have implemented the generation of low-level C code from TeIL, we do not suggest that TeIL should replace existing code generators in tensor frameworks. Instead, as Figure 1 illustrates, TeIL can be used to guard existing code generation processes whenever a guarantee of type-safety is needed, for example in safety-critical applications such as autonomous vehicles or in security-sensitive processing of customer or patient data.

Establishing type-safety for TeIL, of course, requires that the language be formally specified, which makes TeIL attractive to users and implementers of tensor languages alike. Users can unambiguously reason about tensor kernels in terms of TeIL's semantics. Implementers can use TeIL as a tool for quality assurance: if a kernel written in a tensor language can be lowered to TeIL but is not accepted by the original language processor, this may hint at a bug in the language implementation. This may of course also be due to a mismatch between the semantics of the original tensor language and TeIL. We will see examples of practically relevant kernels that can be expressed in TeIL but that exhibit unexpected behaviour when processed within the tensor framework in which the kernels were originally implemented.

The specific contributions of this paper are as follows.

1. We show that commonly used tensor languages suffer from a lack of clarity as far as their semantics are concerned. This is evidenced by tensor kernels whose semantics should be in general agreement with their implementation language but that are rejected by the language processor. (Section 2.)
2. We give a formal specification for the TeIL language, including unambiguous definitions for common tensor operations such as, e.g., *reduction* or the *tensor product*. The previously introduced problematic kernels can be expressed correctly in TeIL. (Section 3.)
3. We formally establish, in the Coq proof assistant [28], type-safety for a core of TeIL. While slightly less flexible, the core language can express the same programs as full TeIL. (Section 4.)
4. TeIL can be used effectively as an intermediate language in code generation. To demonstrate this, we have implemented an *operator language* that is lowered to TeIL, which in turn is lowered to C. (Section 5.)

Our formal development in Coq and the implementation of our operator language are publicly available.[2]

2 Kernels with Unexpected Behaviour

In this section we study a number of tensor kernels, implemented in the following frameworks: *The Tensor Algebra Compiler* (TACO), TVM, and *Tensor Comprehensions* (TC).[3] For the given kernels either the behaviour of the language processor or that of the generated executable kernel code differs from what the user would expect. In the absence of precise specifications for tensor languages, the user's expectations can only be based on how a tensor framework reacts to similar kernels or on the typical behaviour of natural tensor operations.

2.1 The Tensor Algebra Compiler (TACO)

The TACO [16] documentation includes the tensor kernel reproduced in Figure 2a. The symbol B denotes a three-dimensional tensor that is multiplied with the vector C. The result is a matrix A that is indexed with i and j. This binds occurrences of i and j on the right-hand side, and thus leaves k the only *free* index on the right-hand side. TACO employs the convention that free indices are summed over. Hence, the assignment in Figure 2a expresses the mathematical formula $\mathtt{A}_{ij} = \sum_{k=1}^{n} \mathtt{B}_{ijk}\mathtt{C}_k$. This is meaningful only if the size of the vector C and the size of the third dimension of B (both of which are indexed by k) are both equal to n. The sizes of tensor dimensions are specified in the declarations of tensors,

[1] In particular, programs that are not built out of collective operations may not be expressible in TeIL without adding further primitives to the language.

[2] Coq development: https://github.com/normanrink/TensorIR.
Operator language: https://github.com/normanrink/cfdlang/tree/operators.

[3] The following versions are used:
TACO (https://github.com/tensor-compiler/taco) commit afbd65c8b9;
TVM (https://github.com/dmlc/tvm) commit 16abe31c92;
TC (https://github.com/facebookresearch/TensorComprehensions) commit 220b590264.

```
A(i,j) = B(i,j,k) * C(k)
```

(a) Tensor-vector multiplication.

```
A(i,j) = B(i,j,k) * C(i)
```

(b) Summation over index k.

```
A(i,j) = B(k,j,i) * C(i)
```

(c) Indices of B permuted.

Figure 2. Tensor assignments and expressions in TACO.

```
m, n = tvm.var('m'), tvm.var('n')
h = tvm.var('h')
A = tvm.placeholder((m,h), name='A')
B = tvm.placeholder((n,h), name='B')
k = tvm.reduce_axis((0, h), name='k')
C = tvm.compute((m, n), lambda y, x:
        tvm.sum(A[k, y] * B[k, x], axis=k))
```

(a) Incorrect declarations of the dimensions of A and B.

```
m, n = tvm.var('m'), tvm.var('n')
h = tvm.var('h')
A = tvm.placeholder((h,m), name='A')
B = tvm.placeholder((h,n), name='B')
k = tvm.reduce_axis((0, h), name='k')
C = tvm.compute((m, n), lambda y, x:
        tvm.sum(A[k, y] * B[k, x], axis=k))
```

(b) Correct declarations of A and B.

Figure 3. Transposed matrix multiplication in TVM.

```
def fun(float(m) B, float(m) C) -> (A) {
  A(i) = B(i)
  A(j) += C(j)  }
```

(a) Implementation rejected by TC.

```
def fun(float(m) B, float(m) C) -> (A) {
  A(i) = B(i)
  A(i) += C(i)  }
```

(b) Implementation accepted by TC.

Figure 4. Element-wise incrementing in TC.

which we have omitted in Figure 2. If there is a mismatch in the sizes of dimensions that are summed over, TACO rejects the kernel and issues an appropriate error message.

In the kernel in Figure 2b, the index i occurs twice on the right-hand side but is bound and hence not summed over. It is still checked that the sizes of the dimensions indexed with i match. The index k is free and hence summed over. However, since k occurs only once, no check is needed for the dimension indexed with k.

The kernel in Figure 2c is arrived at by permuting the indices of *B*. The expectation is again that the free index k should be summed over. However, TACO apparently loops endlessly when compiling the kernel in Figure 2c, producing neither output nor error messages. Since the kernel can be given a meaning that seems to be in agreement with TACO's semantics, it is likely that the looping is caused by a bug in the implementation of TACO. However, it is equally likely that the user's intuitive understanding of TACO's semantics is incomplete and that the kernel should indeed be rejected.

2.2 TVM

A recent publication on the TVM deep learning compiler stack [5] includes an example kernel for the transposed multiplication of matrices A and B, which is reproduced in Figure 3a. The matrix C that is computed by this kernel can be expressed mathematically as $C = A^T B$.

While the lambda expression in the definition of C performs the correct summation over the index k, the sizes of the dimensions of A and B appear to have been swapped. The first dimension of A has size m, and the first dimension of B has size n. However, in computing C, these dimensions are traversed with the index k that ranges from zero to h. If h is large compared with m, n, evaluating C produces an out-of-bounds memory access that results in a segmentation fault. Almost worse, when h is not large enough to cause a segmentation fault, the lambda expression in the definition of C silently computes incorrect results.

From this behaviour it is clear that TVM does not check the dimensions of tensors against the variables used to index them. This is somewhat surprising, given that TVM does check the dimensions of tensors that are given as input to compiled kernels when the kernels are executed.

The corrected version of the transposed matrix multiplication is given in Figure 3b. The only change from Figure 3a is in the order of the sizes of the dimensions with which A and B are declared. There is of course a possibility that the kernel in Figure 3b is not in fact what the user had intended. Nonetheless, it remains unsafe to execute the kernel in Figure 3a, and the user would benefit from this being detected.

2.3 Tensor Comprehensions (TC)

The *Tensor Comprehensions* (TC) framework [30] accepts all of the tensor kernels given so far, and the semantics are aligned with the user's expectations. In particular, TC circumvents out-of-bounds accesses in the kernel in Figure 3a. This is due to the *range inference* algorithm that TC employs to determine the ranges of indices used in tensor expressions. Whenever TC can successfully infer ranges, it does

$$
\begin{aligned}
\mathit{program} &::= \overline{\mathit{alloc}}\ \overline{\mathit{stmt}} \\
\mathit{alloc} &::= \textbf{alloc}\, x : \bar{\iota} \\
\mathit{stmt} &::= x = e \\
e &::= x \mid (e) \mid \texttt{add}\, e\, e \mid \texttt{mul}\, e\, e \mid \\
&\quad\ \texttt{prod}\, e\, e \mid \texttt{red}_{+}\, i\, e \mid \texttt{transp}\, i\, i\, e \mid \\
&\quad\ \texttt{diag}\, i\, i\, e \mid \texttt{expa}\, i\, i\, e \mid \texttt{proj}\, i\, i\, e \\
\bar{\iota}, \bar{\kappa} &::= [i, \ldots, i]
\end{aligned}
$$

Figure 5. Syntax of TeIL programs.

so conservatively in order to avoid out-of-bounds accesses. Specifically, after the kernel in Figure 3a has been translated into TC, the upper bound $\min(m, n)$ is inferred for the index k. Note also that TC generally issues a warning when it infers ranges that cannot statically be guaranteed not to cause out-of-bounds accesses at kernel runtime.

While TC's range inference algorithm may be capable of inferring ranges that mostly agree with the user's intent, it lacks a precise definition. This makes it impossible for the user to understand whether unexpected behaviour of either TC or of generated kernel code is intentional or not. Figure 4 underlines this. Both kernels in this figure initialize a vector A with the vector B, and then increment each element of A by the corresponding element in the vector C. TC rejects the kernel in Figure 4a with an error message saying that the variable j is undefined. The kernel in Figure 4b is accepted, despite the fact that it expresses the same computation, albeit using slightly different symbols. Without a precise specification of the employed range inference algorithm, the user is left wondering whether this behaviour is desired. As was the case for TACO, it is of course perfectly possible that a subtlety in TC's semantics enforces this behaviour.

3 The TeIL Language

We have seen that mismatches between the expected and actual behaviour of tensor kernels occur even for very simple kernels, and are not limited to only a single tensor framework. Without precise specifications for tensor frameworks and languages, it is impossible to know whether unexpected behaviour is the result of a genuine bug in the implementation or of a subtle design decision. This lack of clarity is particularly troubling given the wide-spread adoption of tensor frameworks, especially for machine learning applications that run in the data centre and, increasingly, also on embedded devices [1, 5]. TeIL, as a formally specified and type-safe language, attempts to introduce more clarity into the reasoning about the behaviour of tensor kernels.

TeIL is designed to express typical tensor kernels that consist of sequences of assignments of tensor expressions to tensor variables, where expressions are formed from collective tensor operations. As an intermediate language, TeIL resides at a fairly low level of abstraction. However, as we introduce the details of the TeIL language, including formal semantics, in this section, we will also see that TeIL can in fact be used flexibly at different abstraction levels.

3.1 Syntax

Figure 5 gives the syntax of TeIL. For any entity z, we use $\overline{z}$ to denote a sequence of similar entities. Thus, a program consists of a sequence of allocations, followed by a sequence of statements. Using the keyword **alloc**, an allocation declares a tensor variable x of type $\bar{\iota}$. The type of a tensor is a (finite) list of natural number literals i, in brackets.[4] The natural numbers in the list are the sizes of the dimensions of the tensor x. Collectively they determine the *shape* of x. Throughout this paper, natural numbers start from one (not zero), and we denote this set of natural numbers as Nat. As a consequence, dimensions of tensors cannot be empty.

Note that a scalar variable s is declared as an allocation **alloc** s : [], i.e. a scalar is a tensor with type the empty list.

A statement assigns an expression e to a tensor variable x. Expressions in TeIL are formed with built-in operations, the meanings of which are explained in Section 3.3.

As can be seen from the examples in Section 2, the typical tensor kernels that TeIL should be capable of expressing do not necessarily contain explicit control flow. Therefore, the TeIL language also does not include explicit control flow constructs. Note, however, that implicit control flow is present in typical tensor kernels, and also in TeIL, in the form of iterations over the index domains of tensor variables.

For similar reasons, TeIL does not allow user-defined functions. As a relatively low-level imperative language, TeIL ought not admit higher-order functions. The language can easily be extended with first-order functions by introducing, for example, global labels and jumps. Alternatively, every TeIL program can be considered a function in its own right, provided that some allocations are marked as function arguments and return values. In this situation, considerations of type-safety reduce to the treatment in the present paper. Hence we omit further discussions of functions in TeIL.

Note that marking some allocations as inputs or outputs of a tensor kernel, similar to function arguments and return values, is required to pass data into and out of TeIL programs. The passage of data is of course necessary in order to make a TeIL program perform computations that are useful to the outside world. However, for discussions of type-safety it is irrelevant where the data for allocations comes from. We therefore omit input and output markers from the presentation of TeIL in this paper.

[4] The notation $\bar{\iota}$ is a minor deviation from our convention that overlined entities denote sequences. By $\bar{\iota}$ we mean a list of entities i (natural number literals), enclosed in brackets. It is best to think of $\bar{\iota}$ as special notation for lists of natural numbers.

3.2 Memory Model

Through the use of the **alloc** keyword, every memory allocation in TeIL is associated with a unique name: the same name may not be used again for another allocation. In TeIL's memory model, a name identifies a map from a list of indices $\bar{\kappa}$, i.e. a list of natural numbers, to a fixed target domain of numbers, typically floating-point numbers in applications involving tensors. We denote the target domain as $\mathbb{D}$. Thus, every name x declared in an allocation is bound to a map such that $x(\bar{\kappa}) \in \mathbb{D}$, provided the map is defined for indices $\bar{\kappa}$. For most indices, this map will in fact not be defined: if x is declared with **alloc** $x : \bar{\iota}$, then $x(\bar{\kappa})$ is defined precisely if $\bar{\kappa} \leq \bar{\iota}$, where lists are compared element-wise. The purpose of TeIL's type system, introduced in Section 3.4, is to ensure that, even inside complex tensor expressions, allocations are never accessed at indices for which they are not defined.

The collection of maps that are declared in the allocations of a TeIL program are stored in the memory μ. An allocation is retrieved from μ by its name, i.e. the name x from the previous paragraph refers to the object $\mu(x)$. Thus, the memory μ is a function with the signature

$$\mu : \mathit{Name} \to (\mathit{list\ of}\ \texttt{Nat}) \to \mathbb{D}\,. \tag{1}$$

We usually omit parentheses when evaluating functions at arguments, i.e. $\mu(x)$ is written as $\mu\, x$, and $\mu(x)(\bar{\kappa})$ as $\mu\, x\, \bar{\kappa}$.

Updating memory means that μ is replaced with a new function μ'. The smallest possible update changes the value of μ at a fixed name y and indices $\bar{\eta}$, i.e.

$$\mu'\, x\, \bar{\kappa} = \begin{cases} r\,, & \text{if } x = y \text{ and } \bar{\kappa} = \bar{\eta} \\ \mu\, x\, \bar{\kappa}\,, & \text{otherwise} \end{cases} \quad ,$$

where $r \in \mathbb{D}$. From this it is clear that in TeIL's memory model there is no aliasing between distinct names.

The described memory model is purposefully abstract. By keeping the memory model abstract, the type-safety of TeIL can be carried over to any lower-level language that is obtained from TeIL by picking a concrete implementation for the memory μ from Equation (1). In particular, TeIL can be used to reason about tensor kernels in frameworks and languages that are built on different memory models. Note that while most frameworks assume a dense layout of tensors in memory, TACO allows mixing of densely and sparsely laid out dimensions even within the same tensor.

It has recently been pointed out that incompatibilities of memory models can be a major source of performance problems when translating between array languages [23]. To avoid incompatibilities when translating to and from TeIL, the abstract memory μ can be instantiated with a suitable concrete implementation. This mechanism enables the use of TeIL at different levels of abstraction.

3.3 Tensor Expressions

According to Figure 5, expressions in TeIL are formed by combining variables x, parentheses, and the collective operations `add`, `mul`, ..., `proj`. Some operations also require integer literals i as arguments. Figure 6 specifies the evaluation semantics of expressions in a denotational style. As the signature of the double bracket in the top-left corner of Figure 6 indicates, expressions are evaluated in a static context (Γ), a memory (μ), and at a list of indices of natural numbers. As before, lists of natural numbers are generally denoted by $\bar{\iota}$. To be precise, the symbol $\bar{\iota}$ in Figure 6 should be thought of as a *meta-variable* (and not as a list of syntactic entities, as in Figure 5).

The definitions for some expressions in Figure 6 require that indices be made more explicit, i.e. that lists $\bar{\iota}$ be expanded into their elements. In this case we enclose lists of natural numbers in pairs of (single) brackets. An example of this appears in the definition of the transposition operation `transp` that swaps the indices j_{i_0} and j_{i_1}, at positions i_0 and i_1 in the list $[j_1, \ldots, j_k]$. Here, again, the indices $j_1, \ldots, j_k$, and also i_0 and i_1, are meta-variables that stand in for concrete natural numbers from `Nat`.

The definition of the operation `diag` also employs an expanded list of indices $[j_1, \ldots, j_k]$ to express that the diagonal is taken of the argument expression e along the dimensions at positions i_0 and i_1. The `expa` operation expands an expression e along the dimension at position i, i.e. for any value of the index at position i, `expa` $i\, n\, e$ takes the same value as e. The projection `proj`, in a sense, is the inverse operation of `expa`: it projects an expression e onto the slice at fixed index m for the dimension at position i.

The operations `add`, `mul` are defined element-wise, except that `mul` also denotes multiplication of a tensor expression e_1 with a scalar e_0. In the definition of the tensor product `prod`, sometimes also referred to as the *outer product*, we use the symbol # to denote the concatenation of two lists of indices. The operation $\texttt{red}_+$ computes the reduction of an expression e by summing the slices of e along the dimension at position i. Other reduction operations, e.g. product, minimum, maximum, are defined analogously.

Finally, evaluating a variable x requires looking up the name x in the memory μ.

To illustrate the evaluation semantics of expressions in TeIL, let us look at the example of matrix multiplication. The top pane of Figure 7 gives the TeIL kernel for the multiplication of matrices A and B. First, the outer product of A and B is formed with `prod`. This results in a four-dimensional tensor, of which a suitable diagonal is selected with `diag`. Lastly, the index of the diagonal is summed over with a $\texttt{red}_+$ operation. The result is then assigned to the matrix C. The bottom pane of Figure 7 shows how the TeIL expression on

$$[\![\cdot]\!] : \mathit{Context} \to \mathit{Memory} \to (\mathit{list\ of}\ \mathtt{Nat}) \to \mathbb{D}$$

$$[\![x]\!]\,\Gamma\,\mu\,\bar{\imath} = \mu\,x\,\bar{\imath}$$

$$[\![(e)]\!]\,\Gamma\,\mu\,\bar{\imath} = [\![e]\!]\,\Gamma\,\mu\,\bar{\imath}$$

$$[\![\mathtt{add}\,e_0\,e_1]\!]\,\Gamma\,\mu\,\bar{\imath} = [\![e_0]\!]\,\Gamma\,\mu\,\bar{\imath} + [\![e_1]\!]\,\Gamma\,\mu\,\bar{\imath}$$

$$[\![\mathtt{mul}\,e_0\,e_1]\!]\,\Gamma\,\mu\,\bar{\imath} = \begin{cases} [\![e_0]\!]\,\Gamma\,\mu\,[\,] \cdot [\![e_1]\!]\,\Gamma\,\mu\,\bar{\imath}, & \text{if } \mathit{type}_\Gamma(e_0) = [\,] \\ [\![e_0]\!]\,\Gamma\,\mu\,\bar{\imath} \cdot [\![e_1]\!]\,\Gamma\,\mu\,\bar{\imath}, & \text{otherwise} \end{cases}$$

$$[\![\mathtt{prod}\,e_0\,e_1]\!]\,\Gamma\,\mu\,(\bar{\imath}_0 \# \bar{\imath}_1) = [\![e_0]\!]\,\Gamma\,\mu\,\bar{\imath}_0 \cdot [\![e_1]\!]\,\Gamma\,\mu\,\bar{\imath}_1\,,\quad \text{if } \mathit{rank}_\Gamma(e_0) = \mathit{length}(\bar{\imath}_0) \text{ and } \mathit{rank}_\Gamma(e_1) = \mathit{length}(\bar{\imath}_1)$$

$$[\![\mathtt{transp}\,i_0\,i_1\,e]\!]\,\Gamma\,\mu\,[j_1,\ldots,j_{i_0},\ldots,j_{i_1},\ldots,j_k] = [\![e]\!]\,\Gamma\,\mu\,[j_1,\ldots,j_{i_1},\ldots,j_{i_0},\ldots,j_k]$$

$$[\![\mathtt{diag}\,i_0\,i_1\,e]\!]\,\Gamma\,\mu\,[j_1,\ldots,j_{i_0-1},j_{i_0},j_{i_0+1},\ldots,j_{i_1-1},j_{i_1}\ldots,j_k] = [\![e]\!]\,\Gamma\,\mu\,[j_1,\ldots,j_{i_0-1},j_{i_0},j_{i_0+1},\ldots,j_{i_1-1},j_{i_0},j_{i_1}\ldots,j_k]$$

$$[\![\mathtt{expa}\,i\,n\,e]\!]\,\Gamma\,\mu\,[j_1,\ldots,j_{i-1},j_i,j_{i+1},\ldots,j_k] = [\![e]\!]\,\Gamma\,\mu\,[j_1,\ldots,j_{i-1},j_{i+1},\ldots,j_k]$$

$$[\![\mathtt{proj}\,i\,m\,e]\!]\,\Gamma\,\mu\,[j_1,\ldots,j_{i-1},j_i\ldots,j_k] = [\![e]\!]\,\Gamma\,\mu\,[j_1,\ldots,j_{i-1},m,j_i,\ldots,j_k]$$

$$[\![\mathtt{red}_+\,i\,e]\!]\,\Gamma\,\mu\,[j_1,\ldots,j_{i-1},j_i,\ldots,j_k] = \textstyle\sum_{m=1}^{n} [\![e]\!]\,\Gamma\,\mu\,[j_1,\ldots,j_{i-1},m,j_i,\ldots,j_k]\,,\ \text{if } \mathit{type}_\Gamma(e) = [n_1,\ldots,n_{i-1},n,n_{i+1},\ldots,n_{k+1}]$$

Figure 6. Evaluation of tensor expressions in TeIL.

$$\begin{aligned} &\mathbf{alloc}\,A : [l,m] \\ &\mathbf{alloc}\,B : [m,n] \\ &\mathbf{alloc}\,C : [l,n] \\ &C = \mathtt{red}_+\,2\,(\mathtt{diag}\,2\,3\,(\mathtt{prod}\,A\,B)) \end{aligned}$$

$$\begin{aligned} &[\![\mathtt{red}_+\,2\,(\mathtt{diag}\,2\,3\,(\mathtt{prod}\,A\,B))]\!]\,\Gamma\,\mu\,[j_1,j_2] \\ &\quad = \sum_k [\![\mathtt{diag}\,2\,3\,(\mathtt{prod}\,A\,B)]\!]\,\Gamma\,\mu\,[j_1,k,j_2] \\ &\quad = \sum_k [\![\mathtt{prod}\,A\,B]\!]\,\Gamma\,\mu\,[j_1,k,k,j_2] \\ &\quad = \sum_k \big([\![A]\!]\,\Gamma\,\mu\,[j_1,k]\big) \cdot \big([\![B]\!]\,\Gamma\,\mu\,[k,j_2]\big) \\ &\quad = \sum_k (\mu\,A\,[j_1,k]) \cdot (\mu\,B\,[k,j_2]) \end{aligned}$$

Figure 7. Matrix multiplication $C_{j_1 j_2} = \sum_k A_{j_1 k} B_{k j_2}$ in TeIL.

the right-hand side of the assignment to C is evaluated according to Figure 6. Each equality is obtained by applying precisely one of the definitions in Figure 6.

Several meta-functions ease the definitions in Figure 6. The function *length* yields the length of a list. The type of a tensor expression is a list of natural numbers, also referred to as the shape of the tensor expression. The precise typing rules are presented in Section 3.4. For now, it suffices to know that typing is syntax-directed, and hence computable. The function type_Γ computes the type of an expression in the context Γ, and rank_Γ is defined as the composition

$$\mathit{rank}_\Gamma = \mathit{length} \circ \mathit{type}_\Gamma\,. \tag{2}$$

TeIL's syntax, as given in Figure 5, allows the nesting of tensor operations in expressions. By restricting the syntax such that only names are allowed as arguments of tensor operations, one can lower the abstraction level of TeIL and effectively turn it into a 3-address code with tensor variables, similar to the language for tensor expressions and assignments in [27].

3.4 Static Semantics

Figure 8 gives the typing rules T-Var, T-Paren, …, T-Proj for tensor expressions in TeIL. These rules make TeIL a type-safe language, with respect to the evaluation semantics from Figure 6. This is achieved essentially by carefully crafting the typing rules such that no undefined values occur in the evaluation of well-typed expressions according to the definitions in Figure 6. We describe a formal proof of type-safety for a core of TeIL in Section 4.

The rules OK-Stmt, …, OK-Prog in Figure 8 define the judgement ok that must hold for well-formed TeIL programs. Well-formed programs are sequences of well-formed (assignment) statements, and are judged ok in the context Γ_{allocs}. Note that *allocs* is a meta-variable that stands in for a specific instance of an entity with syntax $\overline{alloc}$. Analogously, *stmts* stands in for any entity with syntax $\overline{stmt}$.

The context Γ_{allocs} is obtained from the sequence of allocations *allocs* by adding to the context a mapping $x \mapsto \bar{\imath}$ for each allocation $\mathbf{alloc}\,x : \bar{\imath}$ in *allocs*. As stated before, all names that appear in the allocations in a TeIL program are required to be unique.

3.5 Program Evaluation

Figure 9 specifies how sequences of statements and full programs are evaluated. Evaluation proceeds by transforming memory, as is to be expected of an imperative language. The rule St-Stmt enforces that assignments only take place at indices $\bar{\kappa}$ that are within the bounds of the allocation for the name x. This is expressed by the premise $\forall \bar{\kappa} \le \bar{\imath}.\ \bar{\kappa} \in \mathit{dom}(\mu\,x)$. The requirement $\forall \bar{\kappa} \le \bar{\imath}.$ let $r_{\bar{\kappa}} = [\![e]\!]\,\Gamma\,\mu\,\bar{\kappa}$ implicitly enforces that indexing the tensor expression e at $\bar{\kappa}$ yields a well-defined result. The update to the memory μ, in the conclusion of rule St-Stmt, takes place at the name x, for which each index list $\bar{\kappa}$ is then mapped to $r_{\bar{\kappa}} \in \mathbb{D}$.

The memory μ_{allocs} in Figure 9 is formed analogously to the context Γ_{allocs} in Figure 8. For each allocation $\textbf{alloc}\, x : \bar{\iota}$ that appears in *allocs*, a map (*list of* `Nat`) $\to \mathbb{D}$ is added to the memory. The map is defined only for arguments $\bar{\kappa}$ with $\bar{\kappa} \le \bar{\iota}$. Note that, for the purpose of discussing type-safety, the precise values taken by this map are not important.

3.6 Equational Reasoning

The specification of the tensor expressions in Figure 6 enables equational reasoning that can validate optimizations or, generally, transformations of tensor kernels expressed in TeIL. We list only a few example equalities that can be derived from the definitions in Figure 6. All of these examples can be used to reduce the complexity of tensor expressions and hence also the runtime of TeIL programs.

As already hinted at, projection is indeed the left-inverse of expansion,

$$\texttt{proj}\; i\, m\, (\texttt{expa}\; i\, n\, e) \cong e\,. \tag{3}$$

Transposing the dimensions that are involved in a diagonal has no effect,

$$\texttt{diag}\; i_0\, i_1\, (\texttt{transp}\; i_0\, i_1\, e) \cong \texttt{diag}\; i_0\, i_1\, e\,. \tag{4}$$

Finally, reducing along a constant dimension, as obtained from the `expa` operation, can be done in constant time

$$\texttt{red}_+\; i\, (\texttt{expa}\; i\, n\, e) \cong n \cdot e\,. \tag{5}$$

Note that reduction is generally a $O(n)$ operation. Analogous statements hold for reductions that, instead of the sum, compute the minimum, maximum, or product along a dimension.

3.7 Example Kernels

We demonstrate the usability of TeIL by expressing in it the example kernels from Section 2. All of these kernels, apart from the one corresponding to Figure 3a, are valid TeIL programs whose semantics match the expected meanings of the kernels as described in Section 2.

To simplify notation, we introduce *tensor contraction* as a composed operation in TeIL,

$$\texttt{contr}\; i_0\, i_1\, e := \texttt{red}_+\; i_0\, (\texttt{diag}\; i_0\, i_1\, e)\,. \tag{6}$$

It is clear from this definition that tensor contraction is a reduction operation along two dimensions. Contraction is a very natural operation: it generalizes the vector dot product and matrix multiplication to higher-dimensional tensors.

The TeIL versions of the TACO kernels in Figure 2 can then be written as in Figure 10a. Note that the allocations for A, B, and C have been omitted, just as the declarations were omitted in Figure 2.

The TVM kernel in Figure 3b can be expressed in TeIL as shown in Figure 10b. Note that the equivalent of the TVM kernel in Figure 3a would be rejected by TeIL as it violates the typing rule T-Diag.

Both TC kernels from Figure 4 can be expressed in TeIL as in Figure 10c. The problem with TC that it cannot make sense of the index `j` in Figure 4a is of course avoided since there is no explicit indexing in TeIL.

As a final example, independent of the pathological kernels from Section 2, we look at element-wise operations in TeIL that combine tensors of different dimensions. Specifically, Figure 11 demonstrates different ways of performing element-wise addition between a two-dimensional tensor A and a three-dimensional tensor B. The tensor C is obtained by adding to A the projection of B onto its second slice, with respect to its first dimension. This assumes $2 \le k$ (cf. rule T-Proj in Figure 8). For the tensor D, we add to B the expansion of A. Choosing the natural number k as the second argument of `expa` leads to a tensor the dimensions of which match those of B. Hence element-wise addition with B is possible.

4 Verified Core-TeIL

This section introduces Core-TeIL and presents the formal result of its type-safety. The formally verified core of TeIL differs from full TeIL only in the flexibility with which the individual dimensions of a tensor or tensor expression can be addressed. This restricted flexibility greatly eases our formal development in the Coq proof assistant [28]. At the same time, no generality is lost: Core-TeIL can express the same tensor operations as the full language.

4.1 Overview of Core-TeIL

The differences between Core-TeIL and full TeIL are best understood by looking at those typing rules of Core-TeIL that replace some of the rules for expression typing in full TeIL. Figure 12 lists these rules.

The `transp` operation in Core-TeIL transposes adjacent dimensions only. Hence, in Core-TeIL, the `transp` operation only takes a single index argument i. No generality is lost by this since any transposition can be expressed by composing adjacent transpositions. Analogously, $\texttt{red}_+$, `diag`, `expa`, and `proj` loose their index arguments and only affect the first dimension of a tensor expression (or the first pair of dimensions in the case of `diag`). This, again, presents no loss of generality: by pre- and post-composing any of these operations with suitable sequences of transpositions, the same operations can be expressed as in full TeIL.

It is straightforward to adjust the definitions in Figure 6 to the restrictions of Core-TeIL discussed in the previous paragraph. The rules for judging statements and programs `ok` (right-most column of Figure 8), as well as the rules for evaluating statements and programs (Figure 9) remain unchanged in Core-TeIL.

4.2 Formal Results and Type-Safety

The definitions in Figure 6 rely on the existence of a function $type_\Gamma$ that computes the type of a tensor expression in the context Γ. In Section 3.3, we used the observation that the typing rules of TeIL are syntax-directed to justify the

$$\frac{\Gamma(x) = \bar{\imath}}{\Gamma \vdash x : \bar{\imath}}\ \text{T-Var} \qquad \frac{\Gamma \vdash e : \bar{\imath}}{\Gamma \vdash (e) : \bar{\imath}}\ \text{T-Paren}$$

$$\frac{\Gamma \vdash e_0 : \bar{\imath}_0 \quad \Gamma \vdash e_1 : \bar{\imath}_1}{\Gamma \vdash \texttt{prod}\, e_0\, e_1 : (\bar{\imath}_0 \# \bar{\imath}_1)}\ \text{T-Prod}$$

$$\frac{\Gamma \vdash e_0 : \bar{\imath} \quad \Gamma \vdash e_1 : \bar{\imath}}{\Gamma \vdash \texttt{add}\, e_0\, e_1 : \bar{\imath}}\ \text{T-Add}$$

$$\frac{\Gamma \vdash e_0 : \bar{\imath} \quad \Gamma \vdash e_1 : \bar{\imath}}{\Gamma \vdash \texttt{mul}\, e_0\, e_1 : \bar{\imath}}\ \text{T-Mul}$$

$$\frac{\Gamma \vdash e_0 : [\,] \quad \Gamma \vdash e_1 : \bar{\imath}}{\Gamma \vdash \texttt{mul}\, e_0\, e_1 : \bar{\imath}}\ \text{T-SMul}$$

$$\frac{\Gamma \vdash e : [n_1, \ldots, n_i, \ldots, n_k]}{\Gamma \vdash \texttt{red}_+\, i\, e : [n_1, \ldots, n_k]}\ \text{T-Red}_+$$

$$\frac{\Gamma \vdash e : [n_1, \ldots, n_{i_0}, \ldots, n_{i_1}, \ldots, n_k]}{\Gamma \vdash \texttt{transp}\, i_0\, i_1\, e : [n_1, \ldots, n_{i_1}, \ldots, n_{i_0}, \ldots, n_k]}\ \text{T-Transp}$$

$$\frac{\Gamma \vdash e : [n_1, \ldots, n_{i_0}, \ldots, n_{i_1}, \ldots, n_k] \quad n_{i_0} = n_{i_1}}{\Gamma \vdash \texttt{diag}\, i_0\, i_1\, e : [n_1, \ldots, n_{i_0}, \ldots, n_k]}\ \text{T-Diag}$$

$$\frac{\Gamma \vdash e : [n_1, \ldots, n_{i-1}, n_i, \ldots, n_k]}{\Gamma \vdash \texttt{expa}\, i\, n\, e : [n_1, \ldots, n_{i-1}, n, n_i, \ldots, n_k]}\ \text{T-Expa}$$

$$\frac{\Gamma \vdash e : [n_1, \ldots, n_i, \ldots, n_k] \quad m \le n_i}{\Gamma \vdash \texttt{proj}\, i\, m\, e : [n_1, \ldots, n_k]}\ \text{T-Proj}$$

$$\frac{\Gamma(x) = \bar{\imath} \quad \Gamma \vdash e : \bar{\imath}}{\Gamma \vdash x = e : \texttt{ok}}\ \text{OK-Stmt} \qquad \frac{}{\Gamma \vdash \quad : \texttt{ok}}\ \text{OK-Empty}$$

$$\frac{\Gamma \vdash \mathit{stmts} : \texttt{ok} \quad \Gamma \vdash \mathit{stmt} : \texttt{ok}}{\Gamma \vdash \mathit{stmts}\ \mathit{stmt} : \texttt{ok}}\ \text{OK-Seq}$$

$$\frac{\Gamma_{allocs} \vdash \mathit{stmts} : \texttt{ok}}{\Gamma_{allocs} \vdash \mathit{allocs}\ \mathit{stmts} : \texttt{ok}}\ \text{OK-Prog}$$

Figure 8. Static semantics of TeIL.

$$\frac{\Gamma(x) = \bar{\imath} \quad \forall \bar{\kappa} \le \bar{\imath}.\ \bar{\kappa} \in \mathit{dom}(\mu\, x) \quad \forall \bar{\kappa} \le \bar{\imath}.\ \text{let } r_{\bar{\kappa}} = [\![e]\!]\, \Gamma\, \mu\, \bar{\kappa}}{\langle \mu,\ x = e \rangle \longrightarrow_\Gamma \mu\{x \mapsto \lambda \bar{\kappa}. r_{\bar{\kappa}}\}}\ \text{St-Stmt}$$

$$\frac{}{\langle \mu,\ \rangle \longrightarrow_\Gamma \mu}\ \text{St-Empty}$$

$$\frac{\langle \mu',\ \mathit{stmts} \rangle \longrightarrow_\Gamma \mu' \quad \langle \mu',\ \mathit{stmt} \rangle \longrightarrow_\Gamma \mu''}{\langle \mu,\ \mathit{stmts}\ \mathit{stmt} \rangle \longrightarrow_\Gamma \mu''}\ \text{St-Seq}$$

$$\frac{\langle \mu_{allocs},\ \mathit{stmts} \rangle \longrightarrow_{\Gamma_{allocs}} \mu'}{\langle \mu_{allocs},\ \mathit{allocs}\ \mathit{stmts} \rangle \Downarrow \mu'}\ \text{Eval-Prog}$$

Figure 9. Evaluation of statements and programs.

existence of the function $type_\Gamma$. In our formal development in Coq, we can make this rigorous.

Lemma 4.1 (Typing is computable)**.** *There exists a unique, total, computable function "type" such that for all contexts* Γ*, expressions* e*, and lists of naturals* $\bar{\imath}$*,*

$$\Gamma \vdash e : \bar{\imath} \Leftrightarrow type_\Gamma(e) = \bar{\imath}\,.$$

Proof. Structural induction on e and case analysis. For the $\Rightarrow$ direction, case analysis is applied to the typing judgement. For the $\Leftarrow$ direction, case analysis is applied to $type_\Gamma$. □

We omit the definition of $type_\Gamma$, which can be derived straightforwardly from the typing rules. Note that uniqueness follows directly from the uniqueness of types, another simple result that we do not state as an explicit lemma.

For type-safety, the exact contents of memories are not important. What matters, however, is that memories are suitably defined, i.e. have suitable domains. For the purpose of establishing type-safety, therefore, memories that have matching domains are considered equivalent, a notion that is made precise by the following definition.

Definition 4.2 (Equivalence of memories)**.** The memories μ_1, μ_2 are *equivalent*, in symbols $\mu_1 \sim \mu_2$, if

(a) for all names x, $x \in \mathit{dom}(\mu_1) \Leftrightarrow x \in \mathit{dom}(\mu_2)$, and
(b) for all $x \in \mathit{dom}(\mu_1)$ and all lists of naturals $\bar{\kappa}$, $\bar{\kappa} \in \mathit{dom}(\mu_1\, x) \Leftrightarrow \bar{\kappa} \in \mathit{dom}(\mu_2\, x)$.

The key ingredient in proving type-safety for Core-TeIL is the next lemma that states that the double bracket $[\![e]\!]$ is well-defined for well-typed expressions e.

Lemma 4.3 (Well-definedness of $[\![\cdot]\!]$)**.** *For all sequences of allocations "allocs", for all memories* μ*, for all expressions* e*, and for all lists of naturals* $\bar{\imath}$*, the following holds: If* $\Gamma_{allocs} \vdash e : \bar{\imath}$ *and* $\mu \sim \mu_{allocs}$*, then* $[\![e]\!]\, \Gamma_{allocs}\, \mu\, \bar{\kappa}$ *is well-defined for all lists of naturals* $\bar{\kappa}$ *with* $\bar{\kappa} \le \bar{\imath}$*.*

Proof. Structural induction on e. □

Through the definition of $[\![x]\!]$, where x is the name of an allocation (cf. Figure 6), the previous Lemma 4.3 amounts to type-safety for read accesses to the memory μ. That is, the memory μ is only ever read from at names x with $x \in \mathit{dom}(\mu)$, and at indices $\bar{\kappa}$ that are *in-bounds*.

Lemma 4.3 can be strengthened into an existence theorem for types of expressions that occur in well-formed programs. This is achieved by our next theorem, the essence of which is still the same part of type-safety as in Lemma 4.3, i.e. that there are no out-of-bounds reads from μ.

Theorem 4.4. *Let "allocs" be a sequence of allocations, and "stmts" a sequence of statements. Let "*$x = e$*" be a statement that occurs in "stmts", and let* μ *be a memory such that* $\mu \sim \mu_{allocs}$*. If* $\Gamma_{allocs} \vdash \mathit{allocs}\ \mathit{stmts} : \texttt{ok}$*, then there exists a unique list of naturals* $\bar{\imath}$ *such that*

1. $\Gamma_{allocs}(x) = \bar{\imath}$*,*
2. $\Gamma_{allocs} \vdash e : \bar{\imath}$*, and*
3. $[\![e]\!]\, \Gamma_{allocs}\, \mu\, \bar{\kappa}$ *is well-defined for all* $\bar{\kappa} \le \bar{\imath}$*.*

Proof. Case analysis of the judgement $\Gamma_{allocs} \vdash \mathit{allocs}\ \mathit{stmts} : \texttt{ok}$. Existence of $\bar{\imath}$ follows by inversion of the rule OK-Stmt, which must have been used in arriving at the judgement $\Gamma_{allocs} \vdash \mathit{allocs}\ \mathit{stmts} : \texttt{ok}$. The proof is completed by an application of Lemma 4.3. Uniqueness of $\bar{\imath}$ follows again from the uniqueness of types. □

(a) $A = \texttt{contr}\,3\,4\,(\texttt{prod}\,B\,C)$

(b) $A = \texttt{red}_+\,3\,(\texttt{diag}\,1\,4\,(\texttt{prod}\,B\,C))$

(c) $A = \texttt{red}_+\,1\,(\texttt{diag}\,3\,4\,(\texttt{prod}\,B\,C))$

(a) TeIL versions of the kernels in Figure 2.

$\textbf{alloc}\,A : [h, m]$

$\textbf{alloc}\,B : [h, n]$

$\textbf{alloc}\,C : [m, n]$

$C = \texttt{contr}\,1\,3\,(\texttt{prod}\,A\,B)$

(b) TeIL version of the kernel in Figure 3b.

$\textbf{alloc}\,A : [m]$

$\textbf{alloc}\,B : [m]$

$\textbf{alloc}\,C : [m]$

$A = B$

$A = \texttt{add}\,A\,C$

(c) TeIL version of the kernel in Figure 4.

Figure 10. Example kernels from Section 2 expressed in TeIL.

$\textbf{alloc}\,A : [m, n]$

$\textbf{alloc}\,B : [k, m, n]$

$\textbf{alloc}\,C : [m, n]$

$\textbf{alloc}\,D : [k, m, n]$

$C = \texttt{add}\,A\,(\texttt{proj}\,1\,2\,B)$

$D = \texttt{add}\,(\texttt{expa}\,1\,k\,A)\,B$

Figure 11. Combining a two-dimensional tensor A with a three-dimensional tensor B.

$$\frac{\Gamma \vdash e : [n_1, \ldots, n_i, n_{i+1}, \ldots, n_k]}{\Gamma \vdash \texttt{transp}\,i\,e : [n_1, \ldots, n_{i+1}, n_i, \ldots, n_k]}\ \text{T-Transp}^{core}$$

$$\frac{\Gamma \vdash e : [n_1, n_2, \ldots, n_k]}{\Gamma \vdash \texttt{red}_+\,e : [n_2, \ldots, n_k]}\ \text{T-Red}_+^{core}$$

$$\frac{\Gamma \vdash e : [n_1, n_2, \ldots, \ldots, n_k] \quad n_1 = n_2}{\Gamma \vdash \texttt{diag}\,e : [n_2, \ldots, n_k]}\ \text{T-Diag}^{core}$$

$$\frac{\Gamma \vdash e : [n_1, \ldots, n_k]}{\Gamma \vdash \texttt{expa}\,n\,e : [n, n_1, \ldots, n_k]}\ \text{T-Expa}^{core}$$

$$\frac{\Gamma \vdash e : [n_1, n_2, \ldots, n_k] \quad m \leq n_1}{\Gamma \vdash \texttt{proj}\,m\,e : [n_2, \ldots, n_k]}\ \text{T-Proj}^{core}$$

Figure 12. Static semantics of Core-TeIL. (Shown are only those typing rules that differ from the ones in Figure 8.)

The following lemma extends the absence of out-of-bounds accesses also to write access to the memory. This is somewhat implicit in the statement of the lemma, which says that a well-formed statement can take an evaluation step $\longrightarrow_{\Gamma_{allocs}}$. It is in fact the premise of the rule St-Stmt that ensures that no writes can occur out-of-bounds in this evaluation step.

Lemma 4.5 (Evaluation of well-formed statements)**.** *Let "allocs" be a sequence of allocations, and "$x = e$" a statement. Let μ be a memory with $\mu \sim \mu_{allocs}$. If $\Gamma_{allocs} \vdash x = e : \texttt{ok}$, then there exists μ' such that $\langle \mu, x = e\rangle \longrightarrow_{\Gamma_{allocs}} \mu'$ and $\mu' \sim \mu_{allocs}$.*

Proof. The premises of rule St-Stmt hold by inversion of $\Gamma_{allocs} \vdash x = e : \texttt{ok}$ and by Lemma 4.3. The domain of the memory is not changed by the update in the conclusion of rule St-Stmt. ☐

Finally, repeated application of the previous Lemma 4.5 yields our type-safety result for Core-TeIL.

Theorem 4.6 (Type-safety for well-formed programs)**.** *Let "allocs" be a sequence of allocations, and "stmts" a sequence of statements. If $\Gamma_{allocs} \vdash allocs\ stmts : \texttt{ok}$, then there exists a memory μ' such that $\langle \mu_{allocs}, allocs\ stmts\rangle \Downarrow \mu'$. Moreover, $\mu' \sim \mu_{allocs}$.*

Proof. Inversion of $\Gamma_{allocs} \vdash allocs\ stmts : \texttt{ok}$, followed by structural induction on the judgement $\Gamma_{allocs} \vdash stmts : \texttt{ok}$, which effectively amounts to induction on the length of *stmts*. The induction step uses Lemma 4.5. ☐

In prose, the previous type-safety theorem states that well-formed programs can be fully evaluated. As has already been explained, the absence of out-of-bounds accesses is implicit in this, and is in fact a more direct consequence of the lemmas that have led up to Theorem 4.6.

5 A Simple Operator Language

To demonstrate that TeIL can indeed be used effectively as an intermediate language, we now briefly motivate and describe a simple *operator language* that we have implemented on top of TeIL. Code generation for the operator language relies on our TeIL code generator that lowers TeIL programs to C.

One application of tensors and tensor expressions in scientific computing are the so-called *high-order methods* in computational fluid dynamics (CFD) [7]. A typical tensor kernel in high-order methods evaluates the action of a *tensor operator* on a three-dimensional tensor, given a fluid flow problem in three spatial dimensions. Tensor operators are constructed by forming the tensor product of one-dimensional operators, i.e. matrices A, B, C, where the matrices define the action of the tensor operator in each of the three spatial dimensions. The tensor operator constructed from A, B, C is the tensor product $A \otimes B \otimes C$, which can be expressed in TeIL

as prod A (prod $B\,C$). General tensor operators in high-order methods are linear combinations of operators of this form.

The tensor kernel corresponding to $A \otimes B \otimes C$ applies this operator to a three-dimensional tensor u, and stores the result in v. Making indices explicit, this kernel is

$$v_{i_1 i_2 i_3} = (A \otimes B \otimes C)_{i_1}{}^{j_1}{}_{i_2}{}^{j_2}{}_{i_3}{}^{j_3} u_{j_1 j_2 j_3} \quad (7)$$

$$= A_{i_1}{}^{j_1} B_{i_2}{}^{j_2} C_{i_3}{}^{j_3} u_{j_1 j_2 j_3} \,. \quad (8)$$

Doubly occurring indices are contracted (i.e. summed over) as usual, and we use the convention that raised/lowered indices denote the *input*/*output* dimensions of an operator.[5]

Our operator language is a domain-specific language (DSL) embedded in C++. It relies on operator overloading to express the construction and application of tensor operators. The running example of $A \otimes B \otimes C$ is written in the operator language as follows.

```
auto A = Matrix(m, n), B = Matrix(m, n),
     C = Matrix(m, n);
auto u = Tensor<3>(n, n, n);
auto v = (A*B*C)(u);
```

In our implementation of the operator language, code is lowered to TeIL. The code generation of TeIL then produces C code for tensor kernels. This has three key advantages.

First, the strong typing of tensor expressions in TeIL helps avoid mistakes in the lowering of operator language code. In particular, it ensures that contractions only take place over compatible index pairs. This is hugely beneficial given that the implicit index structure of operators and their applications can easily become very complex.

Second, because of TeIL's type-safety, the generated C code is guaranteed not to cause out-of-bounds accesses (i.e. no segmentation faults).

The third advantage is to do with performance. Contractions can easily be detected in TeIL if the composed contr operation from Equation (6) is made a primitive of the language. Nested contractions can then be lifted into a sequence of assignments of single contractions. Assuming $m \approx n$, this reduces the complexity of our operator $A \otimes B \otimes C$ from $O(n^6)$ to $O(n^4)$, and analogously for linear combinations of operators. Although n is typically small, i.e. $n \in \{2, \dots, 20\}$, this reduction in complexity has sizeable effects in practice, not least because the same tensor kernel is typically evaluated for each of several thousand volume elements, cf. [21]. TeIL enables the lifting of nested contractions in a type-safe manner, and can make this optimization accessible to all higher-level languages that generate code through TeIL.

[5] In general relativity, raised and lowered indices are referred to as *contravariant* and *covariant* indices, respectively. Usage of these indices in general relativity corresponds directly to the intuition of input and output dimensions of operators that we employ here.

6 Related Work

APL [13] is widely recognized as one of the first array languages. It is inspired by mathematical notation and relies heavily on collective operations. APL is interpreted, thus naturally allowing for dynamic checks, but compilers also exist, see, e.g., [4, 8] and references therein.

Over time, many array languages have been proposed, e.g. [9, 12, 15, 22, 23, 25]. While syntax and concepts, such as collective operations, are similar across these languages, there are notable differences both in static and dynamic semantics. For example, it has recently been shown [23] that translation between array languages, while retaining performance, is a non-trivial problem that is at least partly rooted in the differences between memory models.

The functional array languages SaC [22] and Futhark [12] come with carefully worked out type systems. Despite the strong static guarantees that are offered by the type systems, there remains a need for some dynamic bounds checks, see also [11]. Note that both SaC and Futhark allow restricted forms of explicit indexing into arrays.

The need for dynamic bounds checks can be fully eliminated by relying on the expressive power of dependent type systems. This expressive power stems from allowing types to depend on values, which, however, may impede phase separation and introduce undecidability into the type system. These issues can be avoided if the dependence of types on values is suitably restricted. Such restrictions are explored in [31] and also in [24, 29], where type-safety results are also given. Note that Lift [25] also relies on a restricted form of dependent types. The rewrite rules employed by Lift are justified by equational reasoning similar to what is enabled by the evaluation semantics for expressions in TeIL.

Lift is a functional language aimed at the generation of efficient code for data-parallel execution on GPU platforms. The addition of data-parallel arrays to Haskell has also been studied [14, 15]. NOVA is another functional language [6] that is specifically designed for parallel programming and hence for targeting execution on parallel platforms. A formal specification for NOVA is available.[6]

The mentioned array languages generally support the paradigm of *shape-polymorphic* programming. TeIL can be made to support this paradigm by replacing the natural number literals that determine the shapes of tensors with variables within the language. Note, however, that in typical use cases of TeIL, this flexibility may not even be desired since compile-time specialization of the shapes of tensors is generally preferred for performance reasons.

Recently, the need for intermediate representations with formal semantics and the curation thereof has been highlighted in [17]. A typed intermediate functional language was formally specified in [9]. The proposed language is an

[6] https://research.nvidia.com/sites/default/files/pubs/2013-07_NOVA-A-Functional/nvr-2013-002.pdf

array language with collective operations similar to the ones in TeIL, and type-safety results have also been established in [9]. To the best of our knowledge, the present paper is the first to report a type-safety result for an imperative tensor language that has also been formalized and checked in a theorem prover.[7]

7 Summary and Outlook

We have presented TeIL, an imperative intermediate tensor language geared at expressing typical tensor kernels as they appear, for example, in the context of machine learning applications. A key feature of TeIL is its type-safety, in the sense that a well-typed program does not cause out-of-bounds memory accesses when executed. This result has been formally developed within the Coq proof assistant. Moreover, the precise specification of TeIL enables users and implementers of tensor languages and frameworks to reason about the semantics of tensor kernels in unambiguous ways.

While we have demonstrated practical uses of TeIL, there are certainly many directions in which the language could grow. To extend TeIL in meaningful ways, it would first of all be helpful to have a precise classification of the tensor computations that can be expressed in TeIL. Such a classification would likely also produce hints at classes of kernels that are currently difficult or impossible to express, or for which code generation is impeded by the design of TeIL. For example, computations for which there is no direct support in the version of TeIL presented here are stencil or convolution kernels. However, to support stencils in TeIL, operations and primitives similar to the ones described in [10] can easily be added to the language. Even without modifying the language, it remains an open question what transformations, other than the lifting of nested contractions, are enabled or eased by TeIL.

One source of TeIL's flexibility is its purposefully abstract memory model. It would be interesting to see how this feature can be put to good use when the abstract memory model is instantiated with formalizations of concrete implementations. One would expect that our formal development is independent of the choice of concrete implementation of the memory model. However, this choice will affect the utility of transformations employed by TeIL's code generator.

There is also an interesting practical use for different memory models in TeIL: different stages of the same code generation process can use versions of TeIL that differ only in the specifics of the employed memory model. This can perhaps help overcome some of the performance problems recently encountered in translating between languages [23].

Due to our focus on type-safety, we have omitted discussions of mechanisms for composing larger programs in TeIL, e.g. functions, and for communicating with the outside world, e.g. passing arguments. To what extent our implementation of TeIL provides mechanisms that are well-suited to typical use cases remains to be determined.

[7] A formal specification of a prototype of TeIL, together with pen-and-paper proofs, appeared in [20].

References

[1] 2019. TensorFlow Lite. https://www.tensorflow.org/lite.

[2] 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 Mané, 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 Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. http://download.tensorflow.org/paper/whitepaper2015.pdf.

[3] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*.

[4] Robert Bernecky. 1997. *APEX: the APL parallel executor.* Master's thesis. University of Toronto.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Carlsbad, CA, 578–594.

[6] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. 2014. NOVA: A Functional Language for Data Parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY '14)*. ACM, New York, NY, USA, Article 8, 6 pages.

[7] M. O. Deville, P. F. Fischer, and E. H. Mund. 2002. *High-Order Methods for Incompressible Fluid Flow.* Cambridge University Press.

[8] Graham C. Driscoll and Donald L. Orth. 1986. Compiling APL: The Yorktown APL Translator. *IBM Journal of Research and Development* 30, 6 (1986), 583–593.

[9] Martin Elsman and Martin Dybdal. 2014. Compiling a Subset of APL Into a Typed Intermediate Language. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY '14)*. ACM, New York, NY, USA, Article 101, 6 pages.

[10] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO '18)*. ACM, New York, NY, USA, 100–112.

[11] Troels Henriksen and Cosmin E. Oancea. 2014. Bounds Checking: An Instance of Hybrid Analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY '14)*. ACM, New York, NY, USA, Article 88, 7 pages.

[12] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, New York, NY, USA, 556–571.

[13] Kenneth E. Iverson. 1962. *A Programming Language.* John Wiley & Sons, Inc., New York, NY, USA.

[14] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (Leibniz International Proceedings in Informatics (LIPIcs))*, Ramesh Hariharan, Madhavan Mukund, and V Vinay (Eds.), Vol. 2. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 383–414.

[15] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 2010. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 261–272.

[16] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (2017), 29 pages.

[17] Nuno P. Lopes and John Regehr. 2018. Future Directions for Optimizing Compilers. *CoRR* abs/1809.02161 (2018). arXiv:1809.02161

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

[19] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530.

[20] Norman A. Rink. 2018. Modeling of languages for tensor manipulation. *CoRR* abs/1801.08771 (2018). arXiv:1801.08771

[21] Norman A. Rink, Immo Huismann, Adilla Susungi, Jeronimo Castrillon, Jörg Stiller, Jochen Fröhlich, and Claude Tadonki. 2018. CFDlang: High-level Code Generation for High-order Methods in Fluid Dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL '18)*. ACM, New York, NY, USA, Article 5, 10 pages.

[22] Sven-Bodo Scholz. 2003. Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting. *J. Funct. Program.* 13, 6 (2003), 1005–1059.

[23] Artjoms Šinkarovs, Robert Bernecky, Hans-Nikolai Vießmann, and Sven-Bodo Scholz. 2018. A Rosetta Stone for Array Languages. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY '18)*. ACM, New York, NY, USA, 1–10.

[24] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer, Berlin, Heidelberg, Germany, 27–46.

[25] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM, New York, NY, USA, 205–217.

[26] Adilla Susungi, Norman A. Rink, Jerónimo Castrillón, Immo Huismann, Albert Cohen, Claude Tadonki, Jörg Stiller, and Jochen Fröhlich. 2017. Towards Compositional and Generative Tensor Optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017)*. ACM, New York, NY, USA, 169–175.

[27] Adilla Susungi, Norman A. Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. 2018. Meta-programming for Cross-domain Tensor Optimizations. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18)*. ACM, New York, NY, USA, 79–92.

[28] The Coq Development Team. 2018. The Coq Proof Assistant, version 8.8.0. https://doi.org/10.5281/zenodo.1219885.

[29] Kai Trojahner and Clemens Grelck. 2009. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming* 78, 7 (2009), 643 – 664.

[30] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730

[31] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, USA, 249–257.