A Verified Compiler for a Functional Tensor Language

AMANDA LIU, Massachusetts Institute of Technology, USA
GILBERT BERNSTEIN, University of Washington, USA
ADAM CHLIPALA, Massachusetts Institute of Technology, USA
JONATHAN RAGAN-KELLEY, Massachusetts Institute of Technology, USA

Producing efficient array code is crucial in high-performance domains like image processing and machine learning. It requires the ability to control factors like compute intensity and locality by reordering computations into different stages and granularities with respect to where they are stored. However, traditional pure, functional tensor languages struggle to do so. In a previous publication, we introduced ATL as a pure, functional tensor language capable of systematically decoupling compute and storage order via a set of high-level combinators known as reshape operators. Reshape operators are a unique functional-programming construct since they manipulate storage location in the generated code by modifying the indices that appear on the left-hand sides of storage expressions. We present a formal correctness proof for an implementation of the compilation algorithm, marking the first verification of a lowering algorithm targeting imperative loop nests from a source functional language that enables separate control of compute and storage ordering. One of the core difficulties of this proof required properly formulating the complex invariants to ensure that these storage-index remappings were well-formed. Notably, this exercise revealed a *soundness bug* in the original published compilation algorithm regarding the truncation reshape operators. Our fix is a new type system that captures safety conditions that were previously implicit and enables us to prove compiler correctness for well-typed source programs. We evaluate this type system and compiler implementation on a range of common programs and optimizations, including but not limited to those previously studied to demonstrate performance comparable to established compilers like Halide.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Domain specific languages**.

Additional Key Words and Phrases: functional programming, array programming, formal verification, type systems, tensors

ACM Reference Format:
Amanda Liu, Gilbert Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2024. A Verified Compiler for a Functional Tensor Language. *Proc. ACM Program. Lang.* 8, PLDI, Article 160 (June 2024), 23 pages. https://doi.org/10.1145/3656390

1 INTRODUCTION

Efficient programming with tensors in domains like image processing and machine learning often reduces to optimizing computation on arrays. This optimization process requires careful management of crucial performance factors such as computational intensity and data locality. It is invaluable to be able to manipulate computation order directly through loop-nest arrangements and storage

Authors' addresses: Amanda Liu, Massachusetts Institute of Technology, Cambridge, USA, lamanda@mit.edu; Gilbert Bernstein, University of Washington, Seattle, USA, gilbo@cs.washington.edu; Adam Chlipala, Massachusetts Institute of Technology, Cambridge, USA, adamc@csail.mit.edu; Jonathan Ragan-Kelley, Massachusetts Institute of Technology, Cambridge, USA, jrk@csail.mit.edu.

ACM 2475-1421/2024/6-ART160
https://doi.org/10.1145/3656390

patterns indicated by storage-access indices. However, the low-level detail required in specifying these scheduling decisions is generally at odds with the high-level, functional representations of algorithms. As a result, a common problem in functional array languages is that they conflate compute and storage order, eliminating a large family of optimizations from the set of programs these languages can describe, including but not limited to common, useful techniques like tiling.

To address this complaint, many user-scheduling frameworks for producing high-performance code provide users with pure, high-level ways to describe the mathematical algorithms to be computed and separately provide scheduling directives that describe compute and storage patterns as part of incremental lowering processes [2, 20]. While this approach has been effective, conflating the optimization and lowering processes makes it very difficult to implement new directives, in addition to complicating any verification process one might try to apply to these systems. ATL [14] is a pure, functional high-level tensor programming language designed with a set of combinators called *reshape operators* to decouple compute and storage order in controlled, structured ways emulating the high-level tensor operations they are named after such as transpose, flatten, concatenate, etc. Previously, ATL was implemented as a shallowly embedded language within the Coq proof assistant, allowing interactive program-optimization transformations to be expressed as verified source-to-source scheduling rewrites that could introduce reshape operators. By implementing scheduling decisions as source-to-source rewrites in ATL separate from its lowering process, the separation of concerns afforded greater ease in verifying these optimizations, ensuring correctness of the scheduling process itself, which can be justified using familiar equational reasoning on functional programs.

We used a trusted lowering algorithm implemented in a destination-passing style to generate flattened array programs in C to compile and benchmark ATL programs. While destination-passing style has been established as useful for compiling functional array code [21], the ATL lowering algorithm had to accommodate the compute and storage reordering effected in the source language. A similar approach was used by Lin and Dubach [12] who introduced views in their IR during their lowering process to express different storage-order choices. None of these compilers were formally verified.

In this paper, we provide the first proof of correctness for a lowering algorithm of a functional tensor language that enables separate compute and storage reordering. This proof is mechanized using the Coq proof assistant. While carrying out the proof, we found a soundness bug in our previously published lowering algorithm [14]. This bug involved the truncation reshape operators that were necessary for expressing computations partitioning arrays to align with vectors and cache lines. While this operator is generally unsafe, we expected it to be used and introduced in programs only in an idiomatic way that would remain safe. Our fix is a new type system capturing the essential safety property for correct compilation.

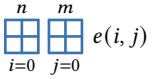
The implementation and proof are available open-source and as an artifact [13].

2 A MOTIVATING EXAMPLE

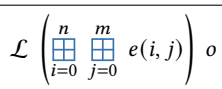
In this section, we will provide a review of fundamental ATL language constructs and their lowering as previously proposed [14]. We will do so by walking through the lowering of a program as well as its various derived, optimized forms and by introducing the specific implementation of the lowering algorithm we will be using in the rest of the paper. This section and the following one are entirely a review of previous work, sometimes introducing slightly different notations.

Tensors are defined recursively as either scalars (zero-rank tensors) or lists of tensors. One of the primary building blocks in the ATL language is a tensor-comprehension, or generation, operator

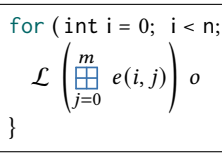
shown below.



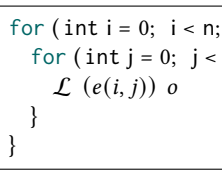
Two tensor-generation operators are used to produce a two-dimensional tensor of length n and width m, where each element is some function e of variables i and j. We can interpret the lowering algorithm as a function, $\mathcal{L}$, that takes in an ATL program and produces a C program that materializes the equivalent tensor into a previously allocated output buffer in memory as a one-dimensional array. Additionally, this lowering function takes an argument o representing an identifier to name the output buffer.



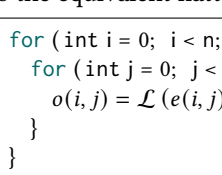
When the lowering function encounters a tensor generation, it generates a `for` loop with equivalent bounds for the iteration index i. Naturally, the body of the `for` loop contains the lowering of the body of the tensor generation.



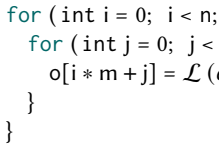
In this case, lowering encounters another tensor generation and produces the equivalent `for` loop, this time with an index j and the bound m.



Once the lowering encounters the body of the tensor generation, it must generate the flattened storage expression indexing into the output buffer o. We cannot assign $e(i, j)$ directly to o, since that buffer is the destination for the full array described by the original ATL program, not a single scalar expression. The storage expression we are assigning to, denoted as $o(i, j)$, is some access expression into o that is a function of i and j that would yield the logically two-dimensional access into the equivalent flattened array.



For this particular example, $o(i, j)$ should resolve to the following flattened access into o, given a convention of linear, row-major memory layout.



```
for (int io = 0; io < n / k1; io++) {
  for (int jo = 0; jo < m / k2; jo++) {
    for (int ii = 0; ii < k1; ii++) {
      for (int ji = 0; ji < k2; ji++) {
        o[(io * k1 + ii) * m + jo * k2 + ji] = ...
      }
    }
  }
}
```

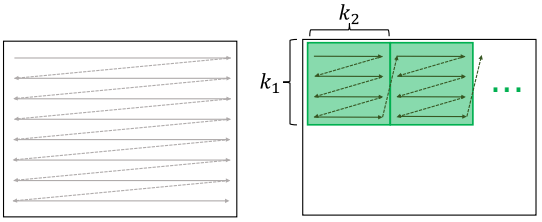


Fig. 1. A tiled program producing a matrix into untiled storage. Tiled computation order is shown in green and the untiled storage order in grey.

More generally, we need an ordered list of index expressions introduced per logical dimension in lowering, such as i and j, as well as the size of each associated dimension, such as n and m, in order to derive the flattened, physical storage-index expression. We represent a destination storage index as a list of 2-tuples of integer-valued expressions, with the first tuple component representing the index and the second representing the size of the associated dimension. The corresponding index data structure produced by the example above would be:

$[(\,i,n);(j,m)] : [Z_e * Z_e]$

The index data structure represents the logical multidimensional access into a tensor. This convention allows us to construct a flatten_index function to generate the integer expression representing the equivalent flattened access index into the physical flattened array.

flatten_index : $[Z_e * Z_e] \rightarrow Z_e$

flatten_index $[(i, n); (j, m)] = i * m + j$

Therefore we equip our lowering function $\mathcal{L}$ with an additional argument $\mathcal{I}$ to represent the destination index: the specific index in the output buffer into which the lowered value is to be written.

2.1 Reshape Operators

Consider the computation order shown in Figure 1 in grey, which produces its output row-by-row. We begin to investigate how the index structure is affected by reshape operators: a set of operators in the ATL language, where each rearranges the elements of a tensor in accordance to some idiomatic tensor operation such as transposition or flattening. What distinguishes reshape operators from other ATL constructs and the reason for their conception is how they affect code generation. Specifically, in the final storage statement generated at the center of a set of loop nests, the computed value must be stored into a buffer at some index. Reshape operators allow for manipulation of this index in accordance with the high-level tensor transformation it represents.

For our purposes in this example, we introduce the tiling reshape operator into this computation. Tiling introduces a new dimension into a tensor by chunking up an existing tensor into tiles of a fixed size. By tiling this computation into the computation order shown in green, we can compute rectangular tiles of size $k_1 \times k_2$, which can improve locality across iterations of e, keeping intermediate data values in fast local memory. To do so, the loop for each tiled dimension must be split into two new outer and inner loops with new outer and inner iteration indices. We want to produce code where the final stage of computation represented by the loop nest writing into the output buffer o has the structure shown in the code in Figure 1. To keep things simple in this example, we assume that the tiling factors k_1 and k_2 evenly divide dimensions n and m respectively. Note that the index-access expression used in storage here is the equivalent of calling flatten_index on the index $[(i_o * k_1 + i_i, n); (j_o * k_2 + j_i, m)]$. However, we know from lowering tensor generations before that to produce a program with that looping structure would mean the storage index structure passed to flatten_index would have to be $[(i_o, n/k_1); (k_o, m/k_2); (i_i, k_1); (j_i, k_2)]$, and the resulting

flattened index expressions would not be equivalent. Thus, in order to perform any tiling-style optimization, we need to decouple the compute order (loop structure) from the storage order (index-storage expression). In the ATL framework, we do so by using scheduling rewrites to introduce reshape operators, arriving at the following program for lowering.

$$\mathcal{L}\left(\text{flatten}\left(\left(\begin{array}{c}n/k_1\\ \square\\ i_o{=}0\end{array}\ \text{flatten}\left(\begin{array}{cccc}m/k_2 & k_1 & k_2 & \\ \square & \square & \square & \ldots\\ j_o{=}0 & i_i{=}0 & j_i{=}0 & \end{array}\right)^T\right)^T\right)\right) \circ \mathcal{I}$$

Immediately, the lowering encounters a call to the reshape operator flatten. As a reshape operator, it generates no code in lowering–instead its effect is entirely on the index data structure. The flatten reshape operator should modify a given index structure by collapsing the top two indices, effectively flattening the top two dimensions. However, we run into a problem. This effect of flatten on an index data structure should be applied on the index representing generation dimensions downstream in the lowering process, related to the subexpression *inside* the flatten operator. This index data structure is currently not accessible, since $\mathcal{I}$ only captures information on the indices that have already been introduced upstream in lowering.

2.2 Introducing Reindexers

To fix this issue, rather than passing down an index data structure of type $[Z_e * Z_e]$ as an argument to lowering, we pass down a function of type $[Z_e * Z_e] \to [Z_e * Z_e]$, which describes how to build upon or modify an index. Logically, this function describes the ongoing *transformation* on the index space, with one possible transformation being the expansion of the space by the introduction of another dimension and index. We call this argument the *reindexer*.

We replace $\mathcal{I}$ with θ to represent the reindexer argument to $\mathcal{L}$. This time when $\mathcal{L}$ encounters the flatten, it produces no code and instead composes the index modification induced by flattening that was described before with the current ongoing θ. The reindexer modification induced by a flatten operation is defined below as the function θ_{flatten}.

```
θflatten idx := match idx with
              | (i,dim1)::(j,dim2)::idx' ⇒ (i * dim2 + j,dim1*dim2)::idx'
              | _ ⇒ idx end.
```

We arrive at the following call to $\mathcal{L}$.

$$\mathcal{L}\left(\left(\begin{array}{c}n/k_1\\ \square\\ i_o{=}0\end{array}\ \text{flatten}\left(\begin{array}{cccc}m/k_2 & k_1 & k_2 & \\ \square & \square & \square & [\ \ldots\ ]\cdot\ldots\\ j_o{=}0 & i_i{=}0 & j_i{=}0 & \end{array}\right)^T\right)^T\right) \circ (\theta\ .\ \theta_{\text{flatten}})$$

The next construct encountered is the transpose operator. Again, since this is a reshape operator, it serves only to modify the reindexer passed to the subsequent call to $\mathcal{L}$. The transpose operator swaps the top two dimensions of a tensor, essentially flipping its rows and columns. Therefore, the transpose reindexer should swap the top two tuples of an index data structure. That reindexer is defined below.

```
θtranspose idx := match idx with
                | t1::t2::idx' ⇒ t2::t1::idx'
                | _ ⇒ idx end.
```

After flatten and transpose are lowered, we reach the following state in the lowering process.

$$\mathcal{L}\left(\left(\begin{array}{c}n/k_1\\ \square\\ i_o{=}0\end{array}\ \text{flatten}\left(\begin{array}{cccc}m/k_2 & k_1 & k_2 & \\ \square & \square & \square & [\ \ldots\ ]\cdot\ldots\\ j_o{=}0 & i_i{=}0 & j_i{=}0 & \end{array}\right)^T\right)\right) \circ (\theta\ .\ \theta_{\text{flatten}}\ .\ \theta_{\text{transpose}})$$

At this point, we encounter a tensor generation, the first construct in the lowering so far that is not a reshape operator. A `for` loop is generated like in the variant of lowering we demonstrated before, and the subsequent call to $\mathcal{L}$ must be made aware of the new loop iteration index and dimension. Rather than adding onto an existing index data structure, the reindexer is composed with a function that adds onto the index-dimension tuple.

`for ( int` $\mathsf{i_o} = 0; \mathsf{i_o} <$ `n //` $k_1; \mathsf{i_o}$`++) {`

$$\mathcal{L}\left(\text{flatten}\left(\boxplus_{j_o=0}^{m/k_2}\ \boxplus_{i_i=0}^{k_1}\ \boxplus_{j_i=0}^{k_2}\ [\ \ldots\ ] \cdot \ldots\right)^T\right)\ o\ (\theta\ .\ \theta_{\text{flatten}}\ .\ \theta_{\text{transpose}}\ .\ (\lambda \text{idx}.(\mathsf{i_o}, n/\mathsf{k_1}) :: \text{idx}))$$

}

We can proceed in lowering the remaining program structure. When we assume the default value for the top-level θ passed in to be the identity function, finally applying the composed reindexer produces the following index structure:

$$[(i_o * k_1 + i_i, n/k_1 * k_1); (j_o * k_s + j_i, m/k_2 * k_2)]$$

This structure is exactly what we wanted for tiled storage access.

To verify this lowering algorithm, much of our attention is given to characterizing an invariant on θ to describe the well-formedness of a reindexer and the properties it may take on during lowering. This invariant must capture the safety of the behavior of the reindexer under the transformations that can be induced by reshape operators introduced in well-formed ATL programs. Rather than exhaustively enumerating the possible reindexer modifications by reshape operators (a *syntactic* and monolithic approach), we chose to define this invariant behaviorally in terms of the index function that the reindexer fundamentally describes (a *semantic* and modular approach). This choice ensures that our proof approach remains general and can accommodate future additions to the language, as long as they adhere to the well-formedness constraint. By establishing this invariant, we are able to provide a formal proof of correctness of the tensor-lowering algorithm for the ATL language and its reshape operators.

3 THE COMPILER WE VERIFIED

3.1 Source Language: ATL

We provide in the appendix a formalization of the ATL language (after a precompilation normalization process) and its operational semantics. These semantics are consistent with the previously published denotational semantics provided for ATL [14]. All ATL programs are expressions that describe the computation of elements of tensor type T. This type is defined inductively to contain scalar values (i.e. real values) and lists of tensor elements.

$$T := \mathit{list}\ T \mid \mathbb{R}$$

For an ATL program e, we use the notation $\|e\|$ to denote the shape of the tensor computed from this program, represented as a list of symbolic integer expressions. For simplicity's sake, we will use $|e|$ to represent the integer expression giving the size of the top-level dimension, or the length of the tensor. Here we refer to symbolic expressions in the sense that a tensor's dimensions may depend on program variables. Similarly, we use $\|e\|_v$ to signify the shape of the ATL program e where each of its symbolic dimensions has been evaluated to concrete integers under the index context v that maps index variables to integers. Likewise, $|e|_v$ represents the evaluated integer of the first dimension. Finally, we define a function genpad that takes a list of integers sh as an argument and produces a tensor of that shape filled entirely with zeros.

While we previously defined ATL using a shallow embedding in Coq [14], in this work we instead formalize ATL with a deep embedding, i.e. explicit syntax trees. This choice also forces

us to write an explicit formal semantics, which was only written on paper in our prior work. We also implemented a Coq tactic that can reify shallowly embedded ATL programs into our deep embedding, generating proof of semantic equivalence.

3.2 Target Language: A Subset of C

The lowering algorithm generates low-level, imperative loop nests. Previous work used this algorithm to generate and benchmark C programs. In the appendix, we provide a formal big-step semantics for the subset of the C language this algorithm generates. We include state modeled as a stack and a heap. The stack stores scalar tensor values as floats[1], and the heap stores higher-dimensional tensors as flattened one-dimensional arrays of floats. The heap is modeled as a partial map of identifiers to flat arrays. Arrays themselves are modeled as partial maps of integers to real numbers. Notably, we do not model pointer arithmetic or aliasing, since code generated by ATL lowering does not use these C features. The C language subset modeled in our semantics also includes explicit memory allocation and deallocation, as well as standard C constructs like loop nests, `if` statements, and assignment.

Previously we had only generated C code via string manipulation (within Coq tactic scripts), with no formal syntax or semantics [14].

3.3 Lowering (Compilation)

We present here our implementation of the previously published lowering algorithm with explicit manipulation of the reindexer and index datatype as proposed above. The previous compiler implementation used Coq's dynamically typed tactic language Ltac, which is not suitable as a subject for Coq proofs [14]. Therefore, we reimplemented the algorithm in Coq's dependently typed logical language Gallina, operating on the two types of syntax trees introduced earlier in this section.

We can define the reindexer functions for the reshape operators much like we did for the lowering of the example in Section 2. These functions are shown in Figure 2. Each manipulates an index in a way consistent with the functional semantics of the reshape operator it is associated with.

We define the lowering algorithm $\mathcal{L}$ shown in Figure 3. We include a new argument a that indicates if the final storage into a buffer is an assignment (=) or reduction (+=) into memory to accommodate summation. We also include a new argument c that maintains a mapping of identifiers to shapes, represented as lists of integer expressions, used specifically to flatten multidimensional tensor accesses in some scalar expression s, denoted as $(\!|s|\!)_c$. This lowering will make use of the static symbolic integer expression representation of tensor dimension size. The size should not be data-dependent nor a function of iteration indices, making it constant at compile time.

4 COMPILER CORRECTNESS

We can express the compiler-correctness theorem as something like the statement below, although additional side conditions will be required. The first premise of the theorem states that a source ATL program e evaluates to a tensor t. The conclusion states that executing the lowered equivalent of this program will result in a state change described by the function tensor_to_array_delta. Variables st and h stand for the C stack and heap.

$$\langle e, v, \Gamma \rangle \Downarrow t \to \cdots \to \langle (\mathcal{L}\ e\ o\ a\ \theta\ c), v, st, h \rangle \Downarrow_C (st, h) \uplus \text{tensor\_to\_array\_delta}\ \theta\ t\ v$$

The arguments to tensor_to_array_delta are a tensor t, a reindexer θ, and an index context v. The function outputs an integer-domain partial map that contains a mapping for every element of

[1]The semantics actually represents floating-point numbers as mathematical real numbers, though it would be valuable future work to extend to sound reasoning about floating point.

```
Definition θflatten idx := match idx with
                         | (i1,dim1)::(i2::dim2)::idx' ⇒ (i1*dim2+i1,dim1*dim2)::idx'
                         | _ ⇒ idx end.
Definition θtranspose idx := match idx with
                         | (i1,dim1)::(i2::dim2)::idx' ⇒ (i2,dim2)::(i1,dim1)::idx'
                         | _ ⇒ idx end.
Definition θsplit k idx := match idx with
                         | (i,dim)::idx' ⇒ (i/k,dim//k)::(i % k,k)::idx'
                         | _ ⇒ idx end.
Definition θtruncr k idx := match idx with
                         | (i,dim)::idx' ⇒ (i,dim − k)::idx'
                         | _ ⇒ idx end.
Definition θtruncl k idx := match idx with
                         | (i,dim)::idx' ⇒ (i − k,dim − k)::idx'
                         | _ ⇒ idx end.
Definition θpadr k idx := match idx with
                         | (i,dim)::idx' ⇒ (i,dim + k)::idx'
                         | _ ⇒ idx end.
Definition θpadl k idx := match idx with
                         | (i,dim)::idx' ⇒ (i + k,dim + k)::idx'
                         | _ ⇒ idx end.
```

Fig. 2. Reshape-Operator Reindexers

tensor t. The key for each element in this map represents the physical, flattened index in the array that this element occupies.

This integer mapping is constructed for each element r at some multidimensional index I_Z by first building the corresponding index structure, by applying the zip function over I_Z and the dimensional size list representing the shape of the tensor, $\|t\|$. Here, zip is a function that takes two lists and constructs a new list where each element is a tuple of the elements of the original lists at that position. We then apply the reindexer to this index structure. Finally, we flatten the resulting index structure and evaluate it under the index valuation v to produce the concrete integer index for each mapping.

Although this intermediary mapping has the same type as an array in the heap, it is only meaningful once it is added onto an existing array in the heap. Hence, we refer to this as an *array delta*. In the theorem statement, we symbolize array addition as $\uplus$. Adding two arrays constructs a new array containing the union of all integer value mappings in both original arrays. Indices that are present in both original arrays are mapped to the sums of the original array mappings.

This definition decomposes very well to accommodate independent reasoning about the storage reordering effected by each reshape operator. Consider a reshape operator R with its associated reindexer θ_R like those shown in Figure 2. We can produce an array delta representing the application of R on some tensor t by directly applying the tensor_to_array_delta transformation with some reindexer θ. But we should be able to produce the same array delta by applying tensor_to_array_delta directly on the tensor t with a reindexer that is the composition of θ and θ_R, as when we apply tensor_to_array_delta composed with θ_R on the original tensor t.

This statement is very close to complete. The other conditions for correctness of lowering will include some standard properties regarding the well-formedness of e and equivalence of states and environments, among other invariants. Most notably, we will have to include invariants regarding

```
Fixpoint ℒ e o a θ c := match e with
| ⊞_{i=lo}^{hi} e' ⇒ for (int i = lo; i < hi; i ++) { ℒ e' o a (λ idx. θ ((i − lo, hi − lo) :: idx)) c }
| ∑_{i=lo}^{hi} e' ⇒ for (int i = lo; i < hi; i ++) { ℒ e' o (+=) θ c }
| [ p ] · e' ⇒ if (p) { ℒ e' o a θ c }
| let x := e1 in e2 ⇒ match ‖e1‖ with
                       | [] ⇒ (* Scalar *)
                       float x = 0; ℒ e1 x (=) (λx.x) c; ℒ e2 o a θ (c[x] = [])
                       | sh ⇒ (* Array *)
                       float *x = calloc (fold_left mul sh 1, sizeof float);
                       ℒ e1 x (=) (λx.x) c; ℒ e2 o a θ (c[x] = ‖e1‖); free(x); end
| flatten e' ⇒ ℒ e' o a (θ . θ_flatten) c
| split k e' ⇒ ℒ e' o a (θ . (θ_split k)) c
| e^T ⇒ ℒ e' o a (θ . θ_transpose) c
| e1 ∘ e2 ⇒ match (‖e1‖),(‖e2‖) with
             | n1::_,n2::_ ⇒ ℒ e1 o a (θ . (θ_padr n2)) c; ℒ e2 o a (θ . (θ_padl n1)) c
             | _,_ ⇒ ; end
| pad_l k e' ⇒ ℒ e' o a (θ . (θ_padl k)) c
| pad_r k e' ⇒ ℒ e' o a (θ . (θ_padr k)) c
| trunc_l k e' ⇒ ℒ e' o a (θ . (θ_truncl k)) c
| trunc_r k e' ⇒ ℒ e' o a (θ . (θ_truncr k)) c
| s ⇒ o[ flatten_index (θ []) ] a ⟪s⟫_c end.
```

Fig. 3. Lowering Algorithm

the behavioral properties and well-formedness of the reindexer θ, which is an entirely unconstrained quantified value in the statement as written.

4.1 Context and State

This correctness statement would not be sound if either program were executing in arbitrary environments. To begin, we must establish that the starting state in which the lowering is executing and the context in which the ATL program is being evaluated are equivalent. The semantics of both ATL interpretation and execution of C code include the iteration index valuation v as one of their arguments, so we can impose direct equivalence on the valuation v. Establishing equivalence between the ATL context Γ and the stack and heap is less straightforward. The ATL context maintains a map of names to tensors computed from previous `let` bindings. The equivalent flattened arrays of these tensors should also be present in the stack and heap. Formally, we can define an equivalence between Γ and the stack and heap (st, h) in terms of tensor_to_array_delta.

$$\Gamma \sim\sim (st, h) \;:=\; \forall x.\, \Gamma[x] = t \longrightarrow (st, h)[x] = \text{tensor\_to\_array\_delta}\ (\lambda i.i)\ t\ \emptyset$$

Note that we use a one-directional implication in formulating this invariant, because the presence of a mapping for an identifier in the stack or heap does not necessarily mean it has been bound in the context. Let us take a look at the code the lowering algorithm generates for `let` bindings.

$\mathcal{L}$(let x := e_1 in e_2) o a θ c

```
float *x = calloc (fold_left mul ‖e1‖ 1, sizeof float);
ℒ e1 x (=) (λi.i); ℒ e2 o a θ c; free(x);
```

The mapping in the heap might have been the result of a memory allocation, while the lowering of the `let`-bound expression has yet to be written to the allocated addresses, as is the case after the allocation. From here on, x is visible in the low-level state as being mapped to an array in the stack. However, there is no corresponding mapping in the ATL context yet. In a `let` binding, x is only in scope in the body of the binding. Therefore, it is only in the ATL context after line 3. If $\sim\sim$ were defined bidirectionally, this invariant would be broken by each `let` binding in between the allocation and the writing of the bound tensor. Therefore we simply state that if a tensor is bound in the ATL context, its equivalent flattened counterpart must be present in the low-level state's stack and heap.

4.2 Well-Formed Allocation

We also specify the presence and well-formedness of the allocated memory that a computation is writing into. The lowering algorithm is implemented in a destination-passing style, meaning that the argument o in $\mathcal{L}\ e\ o\ a\ \theta\ c$ must be a pointer or reference to stack/heap space that has already been allocated to accommodate the computation of e. The stack or heap must contain an existing mapping to store the values of e, even if a is an assignment rather than a plus-equals, since even an assignment to an index without an existing mapping is equivalent to attempting to write to unallocated memory.

Additionally, not only must o include the mappings to accommodate the *size* of e, it must contain the mappings for the indices to which θ may send the indices of e. In other words, if e computes a scalar and θ is the identity reindexer, then the stack must contain a mapping for o. If e computes a nonscalar, n-dimensional tensor t or θ constructs a nontrivial index space, the heap must contain a mapping for o to an array with a mapping for any index reachable by applying θ on any index within the index space of t. We define this property as follows, using the notation $[\![sh_t]\!]$ to denote the set of indices in a tensor with some shape sh_t.

$$
\begin{aligned}
&\text{well\_formed\_allocation } t\ \theta\ st\ h\ o\ :=\ \texttt{match}\ \theta\ (\text{zip}\ [I_0;\ldots;I_n]\ \|t\|)\ \texttt{with}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \mid\ []\ \Rightarrow \exists k.\ st[o] = k\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \mid\ \_\ \Rightarrow\\
&\exists a.\ h[o] = a\ \wedge \forall i.\ i \in [\![\|t\|]\!] \rightarrow \theta\ (\text{zip}\ i\ \|t\|) \in \text{dom}(a)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \texttt{end}
\end{aligned}
$$

4.3 Well-Formed Reindexer

We impose a variety of constraints to define the well-formedness of reindexers. The overall well-formedness of a reindexer θ depends on the iteration-index context v it is being evaluated in, as well as the tensor t that it is acting on. To begin, we remind ourselves of the type of reindexers.

$$\theta\ :\ [Z_e * Z_e] \rightarrow\ [Z_e * Z_e]$$

Reindexers are functions from index to index, where indices are represented as lists of tuples of syntactic integer expressions, denoted by the type Z_e. For the purposes of being able to characterize some functional properties of the flattening index function the reindexer actually represents, we will define a way to evaluate a reindexer using $\Downarrow_v$.

$$\Downarrow_v\ \theta\ : [\mathbb{Z}] \rightarrow \mathbb{Z}$$

By realizing the reindexer, we change it from a function that maps symbolic indices to symbolic indices, into a function that takes in an actual integer index and returns the integer representing its counterpart, physically flattened address. We must require properties including simple structural ones that only depend on the reindexer and the symbolic index form. However, we must also include behavioral properties that describe the functional properties of the interpreted reindexer.

4.3.1 Preservation of Variables. When a reindexer generates an expression for a flat index, intuitively that expression may mix variables present in the reindexer with variables present in the symbolic indexes that were given as input to the reindexer. In fact, the resulting expression should contain *exactly* those free variables, not just a strict subset of them, intuitively because a reindexer should only introduce and shuffle indices–it should not drop any.

$$\forall l.\ \mathsf{vars\_of}\ (\theta\ l) = (\mathsf{vars\_of}\ (\theta\ [])) \ \cup\ (\mathsf{vars\_of}\ l) \qquad \text{(Variable Preservation)}$$

4.3.2 Well-Scoped Variables. Another property we will use to characterize a well-formed reindexer with respect to some valuation v is the proper scoping of its own variables. Each variable present in the reindexer at a given point has been produced in the lowering process that binds that variable to an integer value, such as an iteration index from a tensor generation or summation, so it must be in scope.

$$\mathsf{vars\_of}(\theta\ []) \subseteq\ \mathsf{dom}\ v \qquad \text{(Well-Scoped Variables)}$$

4.3.3 Variable Substitution. In addition to preserving the variables of the reindexer's arguments, it should be possible for us to reason about the evaluation of those variables under the reindexer independently of what the reindexer does. In other words, if we are substituting a variable on the application of some opaque reindexer on an index, we should be able to distribute the variable substitution onto the index itself. In the case where the substituted variable is not present within the variables of the opaque reindexer itself, the substitution can be fully moved under the reindexer application.

$$\forall l, i, x.\ i \notin\ \mathsf{dom}\ v \rightarrow\ (\theta\ l)[x/i] = \theta\ (l[x/i]) \qquad \text{(Variable Substitution)}$$

4.3.4 Determinism. Another characteristic of a well-formed reindexer is determinism. In other words, if two indices are equivalent, then the results after applying the reindexer are equivalent. However, since indices are represented as lists of syntactic integer expressions, we relax our notion of equivalence. We need not require they be syntactically equivalent. Instead we define a notion of equivalence of integer expressions that states that, for any valuation, the expressions evaluate to the same integer.

$$x \sim_{Z_e}\ y := \forall v.\ [\![x]\!]_v = [\![y]\!]_v$$

From here, we can very naturally extend this equivalence from integer expressions to indices themselves.

$$\forall \mathsf{idx}_1, \mathsf{idx}_2.\ \mathsf{idx}_1\ \sim_{[Z_e * Z_e]}\ \mathsf{idx}_2 \rightarrow \theta\ \mathsf{idx}_1\ \sim_{[Z_e * Z_e]}\ \theta\ \mathsf{idx}_2 \qquad \text{(Determinism)}$$

4.3.5 Injectivity. Another well-formedness property we define for reindexers is injectivity. Specifically, the reindexer must be injective over the domain of possible indices over the shape of the tensor t to be computed. In other words, if we evaluate the reindexer and apply it on two literal integer indices in the index space of t, if the resulting integers are equal then the two integer indices must be equal.

$$\forall \mathsf{idx}_1, \mathsf{idx}_2.\ \mathsf{idx}_1 \in [\![\|t\|_v]\!] \rightarrow \mathsf{idx}_2 \in [\![\|t\|_v]\!] \rightarrow \Downarrow_v \theta\ \mathsf{idx}_1 = \Downarrow_v \theta\ \mathsf{idx}_2 \rightarrow \mathsf{idx}_1 = \mathsf{idx}_2$$
(Injectivity)

4.3.6 Non-destructive Assignment. The final well-formedness property we define for reindexers is non-destructivity in the case of assignment storage operators. The reindexer must not be able to produce an index and overwrite a value that was previously written. As a result, all indices to which the reindexer could send the indices of the tensor t must not have been written previously, so those indices retain their original value, which was 0 at the time of allocation.

$$h[o] = \mathsf{arr} \rightarrow a = (=) \rightarrow \forall \mathsf{idx}.\ \mathsf{idx} \in [\![\|t\|_v]\!] \rightarrow \mathsf{arr}[\Downarrow_v \theta\ \mathsf{idx}] = 0 \qquad \text{(Non-destructivity)}$$

Finally we can define the well_formed_reindexer property as the conjunction of the properties described above.

4.4 Compiler Correctness

To return to our correctness theorem, we include the preconditions defined above. This includes the well-formedness of the reindexer, the well-formedness of the allocation in the stack or heap, and the equivalence of execution state between the functional evaluation context and the low-level stack and heap. We have the following statement.

$\langle e, v, \Gamma \rangle \Downarrow t \rightarrow$
$\Gamma \sim\sim (st, h) \rightarrow$
well_formed_allocation $t\ \theta\ st\ h\ o \rightarrow$
well_formed_reindexer $\theta\ v\ t\ h\ o\ a \rightarrow$
$\langle\ (\mathcal{L}\ e\ o\ a\ \theta\ c)\ , v, st, h \rangle\ \Downarrow_C\ (st, h)\ \uplus$ tensor_to_array_delta $\theta\ t\ v$

5 A MOTIVATING COUNTEREXAMPLE

Let us revisit the example used to demonstrate lowering in Section 2. The optimized program was tiled under the assumption that the tensor dimensions were evenly divisible by the tiling factor. However, if we were to produce an optimized program without this assumption, it would look like the one below.

$$\text{trunc}_r\ (k_1 - n\ \%\ k_1) \left(\text{flatten} \left(\left(\boxplus_{i_o=0}^{n//k_1} \text{trunc}_r\ (k_2 - m\ \%\ k_2) \left(\text{flatten} \left(\boxplus_{j_o=0}^{m//k_2} \boxplus_{i_i=0}^{k_1} \boxplus_{j_i=0}^{k_2} [\ \ldots\ ] \cdot \ldots \right) T \right) \right)^T \right) \right)$$

This program structure is similar to the optimized program shown before, but the outer loop-nest bounds are calculated using ceiling division indicated by the `//` operator rather than floor division. Thus, the computation accounts for the elements at the end of the tensor that are not evenly divisible by the tile size. Without further adjustment, the output tensor would be larger than the original. To produce the expected output, truncation operators are introduced around the flattened tiled dimensions, to truncate the overcompute caused by the rounding. The body also contains a guard to limit the actual loop computation to the domain of the original tensor size.

While this program generates the proper tiled imperative code, the usage of truncation operators in general is unsafe, because the truncation reindexers reduce the dimension size, thereby reducing the possible index storage space while leaving the iteration space untouched. In fact, the two truncation reshape operators (trunc_l and trunc_r) do not satisfy the well-formedness conditions as stated and can be used to write programs that produce unsound code.

Consider the following ATL program and its lowered C program. Here, the leftmost k elements are removed from a tensor generation of length n. The lowering algorithm would produce the following C code.

$$\text{trunc}_l\ k \boxplus_{i=0}^{n} e(i)$$

```
for (int i = 0; i < n; i++) {
  o[ i − k ] = e(i);
}
```

While the index offset of k in the storage expression shifts the k-th evaluation of e into the first element of the buffer o, this program fails because this access is unguarded. The first k iterations of the loop make out-of-bounds accesses.

The lowering of the right-truncation operator can also introduce unsoundness. The right-truncation reindexer subtracts k from only the dimension in the index-dimension tuple, restricting

the index space. Consider the following usage of right-truncation and its corresponding lowered C program.

$$\square_{i=0}^{n} \text{trunc}_r\ k\ \square_{j=0}^{m} e'(i,j)$$

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    o[ i * (m − k) + j ] = e′(i, j);
  }
}
```

The expected output of this program is a two-dimensional tensor of size $n * (m-k)$ with elements $e'(i,j)$ evaluated for i up to n and j up to $m-k$. The lowering algorithm would have only allocated enough memory in the buffer o to store $n * (m-k)$ elements, so the last few iterations of this loop nest would also result in out-of-bounds memory accesses.

We can conclude that this lowering algorithm is generally unsound for arbitrary ATL programs. However, it was never intended for programmers to use reshape operators arbitrarily in their programs. The conceit of the ATL scheduling framework was to be able to start with a program written in core ATL constructs and to introduce reshape operators using verified scheduling rewrites [14]. The tiled program example above was indeed derived in that way. The unsound examples are unreachable using this derivation approach. Reshape operators should only be introduced in adjoint pairs. Therefore, truncation should only be in a program if it had been introduced with a complementary pad inside it, or some unfolded or downstream rewritten/optimized equivalent. (Pad is truncation's dual that adds extra zero values to array ends.)

The intuition behind why having a pad immediately inside a truncate would be a safe reindexing transformation is relatively straightforward: the composition of a truncation reindexer and its complementary pad reindexer $(\theta_{\text{truncr}}\ k) \circ (\theta_{\text{padr}}\ k)$ would yield the identity reindexer. The reasoning for why an unfolded and subsequently rescheduled pad inside a complementary truncation is safe is less obvious. It is safe because any program derived from the unfolding and rescheduling of a pad would have some form of a guard that evaluates to false at the indices of the padded cells. Likewise, pad reshape operators introduce padding since the increased dimension size increases the amount of memory allocated for this tensor, but they do not expand the computational space. These guarded and padded values take on the value of zero because our buffers are zeroed upon allocation. Hence, zero values introduced by a pad or a guard are present exclusively due to the absence of a storage operation being performed at that index. As a result, they are not to be included in the index space to be considered when evaluating reindexer properties or when transforming tensors to flattened heap representations.

6 PADDING

It is not the case that *any* zero value in a program is safe for truncating. A zero value may still have been computed explicitly, so that it still results in a storage access. Therefore, we must be able to make a formal distinction between zero values that were computed and zero values that were introduced by padding at the time of allocation. By making this distinction, we would be able to identify the tensor values that are safe to truncate, since a guarded or pad operator-induced zero value at an index symbolizes that there is no storage being performed at that address. Introducing explicit padding values allows our formalization to encode the safety property of a program only ever truncating padding. In doing so, we would be able to represent formally and prove the implicit safety properties we had in mind originally when designing the lowering algorithm. Therefore we introduce a new pad type system for statically tracking a conservative estimate of the padding

pattern within a tensor computation. Finally, we are able to prove correctness of the lowering algorithm on properly pad-typed programs.

6.1 Pad Values and Semantics

In order to represent padded zero values, we modify the tensor type so the scalar value can either be a real number or a pad value represented here as unit, ().

$$T := \mathit{list}\ T \mid \mathbb{R} \mid ()$$

A pad value should have the same algebraic behavior as a computed zero value. However, we do not need to concern ourselves with tracking pad values produced at the level of scalar expression computations—scalar ATL expressions cannot produce pad values. We focus on the padding produced by language constructs such as the pad operator and the guard. We modify ATL semantics so that pad operators and the false guard generate tensors of padding values rather than simply zeros.

6.2 Pad Type

We introduce pad type terms, notated as π, to track the padding of a tensor. Our pad type system can afford to be incomplete and not perfectly precise in tracking the presence of pad values throughout a tensor, since we are only ever truncating from the tensor ends. We only need a conservative estimation of padding in a tensor to ensure these operations are safe.

To capture this pad pattern information, we define a new inductive structure for pad types defined as Π shown below.

$$\Pi := (k, l, \pi_1, r, \pi_2, c) \mid (b)$$

In the recursive constructor, the arguments k and c are natural numbers denoting the numbers of pad elements on the left and righthand sides of the tensor respectively. The argument l represents the number of elements following the padding on the left-hand side of the tensor that have the pad type π_1. Similarly, r represents the number of elements of the right side of the tensor that have the pad type π_2. This pad type does not constrain the inner tensor elements to the same pad type and allows for reasoning about elements on the left and right sides independently with different pad types. Additionally, we need not reason about the pad structure of *all* elements in this tensor since l and r need not add to the remaining length of the tensor. The terminal pad type constructor is meant to type a scalar value. It contains only one argument b that is a Boolean indicating whether or not the scalar is a pad value.

We define the following function to relate a pad type to the padding pattern within a tensor.

```
Fixpoint has_pad pi t := match pi with
                         | (k, l, pi1, r, pi2, c) ⇒
                           match (‖t‖) with
                           | _::s ⇒ Forall (fun x ⇒ x = genpad s) (firstn k t) ∧
                                    Forall (fun x ⇒ x = genpad s) (firstn c (rev t)) ∧
                                    Forall (has_pad pi1) (firstn l (skipn k t)) ∧
                                    Forall (has_pad pi2) (firstn r (skipn c (rev t)))
                           | _ ⇒ False end
                         | (true) ⇒ t = () | (false) ⇒ True end
```

6.3 Pad Type Inference

We construct the following set of typing-judgment rules to infer the pad type of an ATL program in a largely syntax-directed manner. The inference rules are presented in Figure 4 and Figure 5. We formally enforce the colloquial constraint of only truncating padding in rules TruncrPad and

$$\textsc{GenPad}\ \frac{k+c+l+r \le [\![hi-lo]\!]_v \quad \begin{array}{c}\forall x.\ [\![lo]\!]_v + k \le x \to \\ x < [\![lo]\!]_v + k + l \to \\ \Gamma_\pi, v[i \mapsto x] \vdash e : \pi_1\end{array} \quad \begin{array}{c}\forall x.\ [\![hi]\!]_v - c - r \le x \to \\ x < [\![hi]\!]_v - c \to \\ \Gamma_\pi, v[i \mapsto x] \vdash e : \pi_2\end{array} \quad \begin{array}{c}\forall x.\ [\![lo]\!]_v \le x \to \\ x < [\![hi]\!]_v \to \\ x - [\![lo]\!]_v < k \vee [\![hi]\!]_v - c \le x \to \\ \Gamma_\pi, v[i \mapsto x] \vdash e : \pi_{\|e\|}\end{array}}{\Gamma_\pi, v \vdash \boxplus_{i=lo}^{hi} e : (k, l, \pi_1, r, \pi_2, c)}$$

$$\textsc{SumPad}\ \frac{\forall x.\ [\![lo]\!]_v \le x < [\![hi]\!]_v \to \quad \Gamma_\pi, v[i \mapsto x] \vdash e : \pi \qquad [\![lo]\!]_v < [\![hi]\!]_v}{\Gamma_\pi, v \vdash \sum_{i=lo}^{hi} e : \pi}$$

$$\textsc{EmptySumPad}\ \frac{[\![hi]\!]_v \le [\![lo]\!]_v}{\Gamma_\pi, v \vdash \sum_{i=lo}^{hi} e : \pi_{\|e\|}}$$

$$\textsc{FalseGuardPad}\ \frac{\|e\| = sh \qquad [\![p]\!]_v = \text{false}}{\Gamma_\pi, v \vdash [\ p\ ] \cdot e : \pi_{sh}}$$

$$\textsc{GuardPad}\ \frac{\Gamma_\pi, v \vdash e : \pi}{\Gamma_\pi, v \vdash [\ p\ ] \cdot e : \pi}$$

$$\textsc{LetPad}\ \frac{\Gamma_\pi, v \vdash e_1 : \pi_1 \qquad \Gamma_\pi[x \mapsto \pi_1], v \vdash e_2 : \pi_2}{\Gamma_\pi, v \vdash \text{let } x := e_1 \text{ in } e_2 : \pi_2}$$

$$\textsc{ScalarNotPad}\ \frac{}{\Gamma_\pi, v \vdash s : (\text{false})}$$

Fig. 4. Core ATL Pad Type Inference

TrunclPad. The truncation operator only type-checks if the number of elements to be truncated a is less than or equal to the amount of padding that exists on the end of the tensor to be truncated.

The pad type used to describe a multidimensional tensor of entirely padding is not unique. Any pad type with the same depth as dimensionality of an entirely padded tensor is semantically sound. However, this pad type will underspecify the amount of padding unless the numbers at each pad-type level sum to at least the size of that dimension. Although there are still numerous representations of an appropriate pad type, we will use the notation π_{sh} to represent a tensor pad type for a tensor of shape sh that is entirely padding, where each pad-type level includes left padding that is equivalent to that dimension size.

6.4 Pad Type Soundness

We prove the following soundness theorem of the pad type system using the Coq proof assistant. The theorem states that if a program types to a pad type π, then the tensor computed from that same program has the pad pattern indicated by π, defined by the relation has_pad.

Theorem 6.1 (Pad Type Soundness).

$$\forall e, t, v, \Gamma, \pi.\ \langle e, v, \Gamma\rangle \Downarrow t \ \to\ \Gamma, v \vdash e : \pi \ \to\ \text{has\_pad}\ \pi\ t$$

6.5 Strengthening Semantics and Main Compiler Theorem

By distinguishing pad values from standard computed tensor values, we strengthen various definitions of conditions and semantics given previously for the overall compiler-correctness theorem.

6.5.1 Tensor to Array Delta. We first revisit the function tensor_to_array_delta. The function of an array delta is to represent a map of tensor values to the flattened integer indices where the computation is meant to be stored. However, a pad value is meant to represent a zero value that was allocated but not actually computed and written. Therefore, we modify our original definition of tensor_to_array_delta to account for the distinction between scalar values and pad values by only mapping scalar values.

6.5.2 Injectivity. We similarly redefine our domain required for injectivity of well-formed reindexers. Previously, a well-formed reindexer over a tensor t had to be injective over the entire index

$$\textsc{ConcatPad}\ \frac{\Gamma_\pi, v \vdash e_1 : (k_1, l_1, \pi_1, r_1, \pi_3, c_1) \qquad \Gamma_\pi, v \vdash e_2 : (k_2, l_2, \pi_4, r_2, \pi_2, c_2) \qquad k_1 + l_1 + r_1 + c_1 \leq |e_1|_v \qquad k_2 + l_2 + r_2 + c_2 \leq |e_2|_v}{\Gamma_\pi, v \vdash e_1 \circ e_2 : (k_1, l_1, \pi_1, r_2, \pi_2, c_2)}$$

$$\textsc{TruncrPad}\ \frac{\Gamma_\pi, v \vdash e : (k, l, \pi_1, r, \pi_2, c) \qquad 0 \leq [\![a]\!]_v \leq c}{\Gamma_\pi, v \vdash \mathsf{trunc_r}\ a\ e : (k, l, \pi_1, r, \pi_2, c - a)}$$

$$\textsc{TruncrPad}\ \frac{\Gamma_\pi, v \vdash e : (k, l, \pi_1, r, \pi_2, c) \qquad 0 \leq [\![a]\!]_v \leq k}{\Gamma_\pi, v \vdash \mathsf{trunc_r}\ a\ e : (k - a, l, \pi_1, r, \pi_2, c)}$$

$$\begin{array}{lll}
\text{let } x' & := & x * [\![m]\!]_v + \quad (\min 1\ l) * a \text{ in} \\
\text{let } y' & := & y * [\![m]\!]_v + \quad (\min 1\ r) * b \text{ in} \\
\text{let } l' & := & (\min 1\ l) * \quad \text{match } a, c, (l_1 =? [\![m]\!]_v) \text{ with} \\
 & & \quad |0, 0, \text{true} => l * [\![m]\!]_v \\
 & & \quad |_,_,_ => l_1 \\
 & & \quad \text{end in} \\
\text{let } r' & := & (\min 1\ r) * \quad \text{match } d, b, (r_2 =? [\![m]\!]_v) \text{ with} \\
 & & \quad |0, 0, \text{true} => r * [\![m]\!]_v \\
 & & \quad |_,_,_ => r_2 \\
 & & \quad \text{end in} \\
\text{let } \pi & := & (x, l, (a, l_1{}', \pi_1, r_1', \pi_2, c), \quad r, (d, l_2', \pi_3, r_2', \pi_4, b), y) \text{ in}
\end{array}$$

$$\textsc{FlattenPad}\ \frac{\Gamma_\pi, v \vdash e : \pi \qquad \|e\|_v = n :: m :: sh \qquad a + c + l_1 + r_1 \leq m \qquad d + b + l_2 + r_2 \leq m \qquad x + y + l + r \leq n}{\Gamma_\pi, v \vdash \mathsf{flatten}\ e : (x', l', \pi_1, r', \pi_4, y')}$$

$$\begin{array}{lll}
\text{let } l_2 & := & \min a\ d \quad \text{in} \\
\text{let } r_2 & := & \min b\ c \quad \text{in} \\
\text{let } \pi & := & (x, l, (a, l_1{}', \pi_1, r_1', \pi_2, c), \quad r, (d, l_2', \pi_3, r_2', \pi_4, b), y) \text{ in}
\end{array}$$

$$\textsc{TransposePadStrong}\ \frac{\Gamma_\pi, v \vdash e : \pi \qquad \|e\|_v = n :: m :: sh \qquad n - x - y \leq l + r \qquad m - l_2 - r_2 \leq l_3 + r_3}{\Gamma_\pi, v \vdash e^T : (l_2, l_3, (x, 0, \pi_1, 0, \pi_2, y), r_3, (x, 0, \pi_3, 0, \pi_4, y), r_2)}$$

$$\begin{array}{lll}
\text{let } l_3 & := & \min a\ d \quad \text{in} \\
\text{let } r_3 & := & \min b\ c \quad \text{in} \\
\text{let } \pi & := & (x, l, (a, l_1, \pi_1, r_1, \pi_2, c), \quad r, (d, l_2, \pi_3, r_2, pi_4, b), y) \text{ in}
\end{array}$$

$$\textsc{TransposePadWeak}\ \frac{\Gamma_\pi, v \vdash e : \pi \qquad \|e\|_v = n :: m :: sh}{\Gamma_\pi, v \vdash e^T : (0, l_3, (x + l, 0, \pi_1, 0, \pi_2, y + r), r_3, (x + l, 0, \pi_3, 0, \pi_4, y + r), 0)}$$

$$\begin{array}{lll}
\texttt{let } a & := & [\![\mathrm{a}]\!]_v \quad \texttt{in} \\
\texttt{let } c' & := & c + ((a - n \,\%\, a) \,\%\, a) \quad \texttt{in} \\
\texttt{let } \pi_1' & := & (k \,\%\, a, \min l\ (a - k \,\%\, a), \pi_1, 0, \pi_1, 0) \quad \texttt{in} \\
\texttt{let } \pi_2' & := & (0, 0, \pi_2, \min r\ (a - c \,\%\, a), \pi_2, c' \%a) \quad \texttt{in}
\end{array}$$

$$\textsc{SplitPad}\ \frac{\Gamma_\pi, v \vdash e : (k, l, \pi_1, r, \pi_2, c) \qquad k + c + l + r \leq |e|_v \qquad 0 < a}{\Gamma_\pi, v \vdash \mathsf{split\ a}\ e : (k/a, k//a - k/a, \pi_1', c'//a - c'/a, \pi_2', c'/a)}$$

$$\textsc{PadrPad}\ \frac{\Gamma_\pi, v \vdash e : (k, l, \pi_1, r, \pi_2, c) \qquad 0 < |e|_v \qquad 0 < [\![a]\!]_v \qquad k + l + r + c \leq |e|_v}{\Gamma_\pi, v \vdash \mathsf{pad_r}\ a\ e : (k, l, \pi_1, r, \pi_2, c + a)}$$

$$\textsc{PadlPad}\ \frac{\Gamma_\pi, v \vdash e : (k, l, \pi_1, r, \pi_2, c) \qquad 0 < |e|_v \qquad 0 < [\![a]\!]_v \qquad k + l + r + c \leq |e|_v}{\Gamma_\pi, v \vdash \mathsf{pad_l}\ a\ e : (k + a, l, \pi_1, r, \pi_2, c)}$$

Fig. 5. Reshape Operator Pad Type Inference

space of t. With the static knowledge of some distribution of padding values, we can constrain the domain of indices we consider reindexers to be transforming. We redefine the domain for injectivity to be the indices of a tensor t that contain non-padding values.

$$\forall \mathsf{idx}_1, \mathsf{idx}_2.\ \mathsf{idx}_1 \in [\![\|t\|_v]\!] \to \mathsf{idx}_2 \in [\![\|t\|_v]\!] \to\ t[\mathsf{idx}_1] \neq () \to\ t[\mathsf{idx}_2] \neq () \to \Downarrow_v\ \theta\ \mathsf{idx}_1 = \Downarrow_v\ \theta\ \mathsf{idx}_2 \to \mathsf{idx}_1 = \mathsf{idx}_2 \quad \text{(Injectivity)}$$

6.5.3 Non-destructive Assignment. We similarly constrain the domain required for non-destructive assignments produced by the reindexer, since we reduced the domain of injectivity.

$$h[o] = \mathsf{arr} \to a = (=) \to \forall \mathsf{idx}.\ \mathsf{idx} \in [\![\|t\|_v]\!] \to t[\mathsf{idx}] \neq () \to \mathsf{arr}[\Downarrow_v\ \theta\ \mathsf{idx}] = 0 \quad \text{(Non-destructivity)}$$

6.5.4 Context and State Equivalence. We also modify the equivalence property between the functional evaluation context and the stack and heap. The previous definition of equivalence equated any tensor mapped in the functional context to the array delta produced by the application of tensor_to_array_delta. However, this condition no longer applies for our new definition of tensor_to_array_delta since it omits pad values. Although pad values are not present in the array delta, they should have still been present in the heap upon its original allocation. Therefore, the restatement of equivalence shown below adds the array delta produced from t and its original allocation.

$$\Gamma \sim\sim (st, h)\ :=\ \forall x.\ \Gamma[x] = t \to (st, h)[x] = \mathsf{tensor\_to\_array\_delta}\ (\lambda i.i)\ t\ \emptyset \uplus \mathsf{tensor\_to\_array\_delta}\ (\lambda i.i)\ (\mathsf{genpad}\ \|t\|)\ \emptyset \quad \text{(Environment Equivalence)}$$

6.5.5 Correctness Theorem. In addition to the redefinitions stated above, we must also restate our overall correctness theorem to include the precondition that an ATL program must be well-typed within the pad type system.

Theorem 6.2 (Strengthened Compiler Correctness).

$$\begin{aligned}
&\langle e, v, \Gamma\rangle \Downarrow\ t \to\\
&\Gamma \sim\sim (st, h) \to\\
&\mathsf{well_formed_allocation}\ t\ \theta\ st\ h\ o \to\\
&\mathsf{well_formed_reindexer}\ \theta\ v\ t\ h\ o\ a \to\\
&\Gamma_\pi \vdash e : \pi \to\\
&\langle\ (\mathcal{L}\ e\ o\ a\ \theta\ c), v, st, h\rangle\ \Downarrow_C\ (st, h)\ \uplus\ \mathsf{tensor_to_array_delta}\ \theta\ t\ v
\end{aligned}$$

From here we use this strengthened correctness theorem equipped with all the proper invariants to prove the following top-level correctness theorem, which corresponds to the top-level call to lowering in an empty environment except for the initial allocation in which the tensor computation is to be stored.

Theorem 6.3 (Compiler Correctness).

$$\begin{aligned}
&\langle e, \emptyset, \emptyset\rangle \Downarrow\ t \to\\
&\vdash e : \pi \to\\
&\langle\ (\mathcal{L}\ e\ o\ (=)\ \theta\ \emptyset), v, (\emptyset, \emptyset)[o \mapsto \mathsf{alloc}\ \|e\|_\emptyset]\rangle\ \Downarrow_C\ (\emptyset, \emptyset)[o \mapsto \mathsf{alloc}\ \|e\|_\emptyset]\ \uplus\ \mathsf{tensor_to_array_delta}\ (\lambda i.i)\ t\ \emptyset
\end{aligned}$$

Note that most of the intricate invariants we defined earlier do not appear in this final theorem statement, so bugs in their statements cannot lead us to accept unsound compilers. For a given program, its concrete typing derivation can be constructed easily enough, and the specific action of tensor_to_array_delta can be computed, so that we derive correct execution of the compiled

Program	# of Gen (⊞)	# of Concat	# of Truncate	# of Transpose	# of Flatten	# of Split
Gather	3	-	-	-	-	-
Scatter	3	-	-	-	-	-
Im2col Conv	3	-	-	-	-	-
Im2col Mat	7	-	-	-	-	-
Matmul	2	-	-	-	-	-
Tiled Matmul	4	-	2	1	2	-
Tiled+Tails Matmul	10	2	2	1	2	-
Two-Stage Blur	4	-	-	-	-	-
Fused Blur	2	-	-	-	-	-
Fused+Tails Blur	8	4	-	-	-	-
Tiled+Tails+Staged Blur	21	6	3	7	3	-
Tensor Add	4	-	-	-	-	-
Split Tensor Add	1	-	-	-	-	3

Fig. 6. Reshape Complexity of Evaluation Programs

program without needing to trust either the type system or the auxiliary functions used to describe action on the heap.

7 EVALUATION

We evaluate our implementation of the lowering algorithm and tactic machinery for type-checking on various programs. We include the programs that were used in our previous work to evaluate the scheduling expressivity of the ATL optimization framework, with scheduled programs for a number of algorithms including the two-dimensional box blur and various convolutional programs that demonstrate gather-to-scatter and im2col transformations [14]. We previously used these programs to evaluate the performance and scheduling expressivity of the rewrite optimization framework and lowering as a whole. In contrast, we use these programs to evaluate the implementation of the lowering algorithm and the effectiveness of our pad-type system and associated tactic machinery on various programs. We focus on optimizations that emphasize the use of reshape operators in nontrivial ways. We also introduce a new collection of additional programs that represent other common program structures and optimizations. Note that we do not report new performance results here, because we implement the lowering algorithm that was already evaluated for its optimization effectiveness [14].

In Figure 6 we list each program we used to evaluate our pad type system and compiler implementation along with occurrence counts for different language operators. We do so to try and capture the diverse reshape complexity as well as the pad-pattern complexity that our type system is able to account for to make a case for its applicability on desirable real-world optimizations.

To explore the extent to which certain reshape-operator patterns and idiomatic optimizations affect padding, compute order, and storage order, we will discuss in-depth the two new evaluation programs for computing matrix multiplication and tensor addition, as well as the most sophisticated prior example, the tiled box-blur program.

7.1 Matrix Multiplication

Matrix multiplication is a fundamental operation in linear algebra. It involves multiplying corresponding elements of each row of one matrix with the elements of the corresponding column from the second matrix and summing up these products.

7.1.1 Standard Matrix Multiplication. The following program is a standard matrix multiplication written in ATL.

$$\boxplus_{i=0}^{M} \boxplus_{j=0}^{N} \sum_{k=0}^{K} m_1[i;k] * m_2[k;j]$$

Provided that the tensor-generation bounds match input matrices' dimensions, this program trivially typechecks and passes our safety preconditions for compilation as it introduces no padding and performs no truncation operations.

7.1.2 Tiled Matrix Multiplication. For our optimized evaluation program, we focus on the technique of tiling, a common optimization that increases data locality within a program. It also involves a sophisticated usage of reshape operators. As we have discussed, it introduces padding when the tiling factor does not evenly divide the tiled dimension. The optimized program below is the tiled equivalent of the matrix-multiplication program from above that we achieved through applying rewrites from ATL's verified scheduling framework.

$$\begin{array}{l}
\mathsf{trunc_r}\ (C - M\%C) \quad \Big(\mathsf{flatten} \\
\quad \Big(\boxplus_{i_o=0}^{M//C} \Big(\mathsf{trunc_r}\ (C - N\%C) \quad \Big(\mathsf{flatten} \\
\qquad \Big(\boxplus_{j_o=0}^{N//C} \boxplus_{i_i=0}^{C} \boxplus_{j_i=0}^{C} \\
\qquad [\ j_o * C + i_i < N \wedge i_o * C + i_i < M\ ] \cdot \\
\qquad \sum_{k=0}^{K} m_1[i_o * C + i_i; k] * m_2[k; j_o * C + j_i] \Big) \Big) \Big)^T \Big) \Big)
\end{array}$$

The program structure includes two right-truncations–one per tiled dimension to accommodate the case where the dimensions M and N are not divisible by the tiling factor C. There is a flatten operator for each tiled dimension that collapses the storage of the new outer and inner loop dimensions. The transpose operator moves the outer loops next to each other and the inner loops next to each other.

This program properly type-checks in our pad type system. This result is notable since it is not obvious upon inspection that the truncations are safe, as they do not have matching explicit $\mathsf{pad_r}$ operators. It would not even suffice to limit the value of one explicit loop index. Instead, the safety of the truncation is determined by the guard conditions in the loop body that involve reasoning about the two inner and outer loop-index pairs.

7.1.3 Tiled Matrix Multiplication with Loop Separation. We can take our optimization a step further to extend the evaluation of our type-checking and safety mechanisms. One might notice that the innermost guard can only be false in one iteration of the index i_o and one iteration of the index j_o. Another common optimization technique in this case is to split these loops into two segments where the guard is always true in one of the segments and can be removed. When the segment in which the guard is always true dominates the latter in size, as in this case, the optimization is very useful: it reduces the number of times the guard has to be executed and produces a main loop structure more amenable to vectorization. We can schedule this optimization to produce the program below.

$$\text{trunc}_r\ (C - M\%K)\ \Bigg(\text{flatten}\ \Bigg(\boxplus_{i_o=0}^{M/C}\ \Bigg(\text{trunc}_r\ (C - N\%C)\ \Bigg(\text{flatten}\ \Bigg(\boxplus_{j_o=0}^{N/C}\ \boxplus_{i_i=0}^{C}\ \boxplus_{j_i=0}^{C}\ [\ j_o * C + i_i < N \wedge i_o * C + i_i < M\ ] \cdot \sum_{k=0}^{K} m_1[i_o * C + i_i; k] * m_2[k; j_o * C + j_i]\Bigg) \circ \boxplus_{j_o=N/C}^{N//C} \ldots\Bigg)\Bigg)^T\Bigg) \circ \boxplus_{i_o=M/C}^{M//C} \ldots\Bigg)$$

In addition to the reshape-operator complexity involved in the previous tiled matrix-multiplication example, this program uses the concatenation reshape operator to join the main loop with its tail regions. Being able to use concatenation in a nontrivially reshaped program like this one with padding is one of the primary cases that motivated us to adopt a tree-like pad type, decoupling the inner pad term on the left from the right. Our system is indeed able to type and compile this program.

7.2 Tensor Addition

In another example, we compute the sum of two tensors. We use an elementwise sum between two tensors with a different loop iterating for each tensor dimension. This kind of loop with a pointwise inner computation is a common target for optimization through parallelization.

$$\boxplus_{i=0}^{M}\ \boxplus_{j=0}^{N}\ \boxplus_{k=0}^{K}\ \boxplus_{l=0}^{C}\ t_1[i; j; k; l] * t_2[i; j; k; l]$$

The depth of this loop nest will increase linearly with the dimensionality of the input tensors. The control-flow complexity is excessive since physically we will be iterating straight through the flattened tensors in the heap. As a result, there will be unnecessary overhead introduced by each loop—especially high if we want to parallelize them all. Instead, we can use scheduling rewrites to produce the following program.

$$\text{split}\ K\ \Bigg(\text{split}\ N\ \Bigg(\text{split}\ C\ \Bigg(\boxplus_{i=0}^{MNKC}\ t_1[i/(NKC); i/(NC)\%K; i/C\%N; i\%C] * t_2[i/(NKC); i/(NC)\%K; i/C\%N; i\%C]\Bigg)\Bigg)\Bigg)$$

The loop nest has been flattened into one loop. The split reshape operator allows us to tile the storage, thereby producing the desired higher-dimensional tensor sum. Although we demonstrate this optimization on four-dimensional tensors, it can be applied to tensors of any dimension. Both the initial and optimized program here are properly safety- and type-checked by our system and are properly compiled.

7.3 Blur

A stencil is a common programming pattern that applies a fixed-sized, predefined computation pattern to a tensor. The two-dimensional box-blur is one such algorithm. An image is blurred by computing for each pixel the average of a rectangular region of surrounding pixels. Typical scheduled forms of this algorithm are exhibited in previous work [14]. All these programs were derived and proven to be equivalent within the ATL scheduling framework, including a fully fused schedule where the average of the surrounding window is computed at once for each output pixel.

Another version is a two-stage computation, where one direction of the blur is computed for the full image before being used to compute the other direction. There is also the tiled, staged program with loop separation to maximally eliminate guarded expressions. This optimized program was derived to incorporate the same set of optimizations that Halide performs on the blur algorithm.

Like the tiled matrix-multiplication example, tiling introduces guards. However, the tiling in this program occurs across computational stages, which yields additional guards since a stencil involves a neighborhood of computation. These guards are individually split into separate loop segments to produce one main, steady-state loop without any guards. Not only does this example cover a common programming pattern and optimization, it uses reshape operators extensively to achieve this structure. This reordering interacts nontrivially with the loop-splitting optimizations that separate out loop tails, thereby sharding padding within the tensor. We found that our pad type system is able to accept each of these scheduled forms of the blur algorithm and that compilation succeeds.

8 RELATED WORK

Compiler verification has become an extremely broad area, but the best-known example is CompCert [11], which compiles C to assembly. Tensor kernels could be written directly in C, but at dramatically increased cost in developer effort and risk of certain kinds of bugs. ATL [14] instead allows these kernels to be written in Coq's native functional language and thus amenable to comfortable equational reasoning to prove correctness of programs and program transformations. Verified compilation for functional programs has also received attention, perhaps most thoroughly in the CakeML [8] project, which compiles an ML language to use a verified garbage collector, imposing a performance cost. However, compute and storage order are conflated, as they are with most functional-language compilers, so that purely functional programs cannot be compiled to achieve competitive performance, even if we imagine that garbage-collection overhead dropped to zero. Fiat Cryptography [6] compiles a functional language to more idiomatic C code after extensive partial evaluation, but that partial evaluation must eliminate lists at compile time, making it unsuitable for tensor kernels.

There is other recent work that applies to programs with loop nests and mutable arrays. Courant and Leroy [4] produced a verified compiler for the polyhedral model with a common loop-nest IR. Clément and Cohen [3] verify the lowering of a subset of Halide programs using translation validation, meaning that there may be compiler soundness bugs that go uncaught until a novel test case (source program) triggers a failure of translation validation, in contrast to a classic compiler proof's coverage of all valid input programs. Kovach et al. [7] introduced indexed streams as an intermediate representation for tensor contractions (generalizations of matrix multiplication). They handle sparse tensors, whereas ATL and thus our work focus on dense tensors. However, their compiler is also unverified as in the past ATL work, and contractions, while an important and common operator family, are insufficient to express most of our case-study programs.

Another line of work within proof assistants uses tactics to drive generation of imperative programs from functional ones. Myreen and Owens [16] pioneered generation of functional-language ASTs from native functional programs of proof assistants, paving the way for integration with verified compilers, for instance more recently with CakeML [1]. CertiCoq [17] realizes a similar recipe in Coq. Lammich [10] showed how to refine functional programs into imperative programs with tactics in Isabelle/HOL, connecting to work on automatic refinement with efficient data structures [9]. The Fiat framework [5] used Coq to translate specifications into efficient functional programs via stepwise refinement, followed by more tactic-driven generation of imperative code [19]. The project Rupicola [18] improved some of these ergonomics. As far as we know, those frameworks have not been applied to tensor kernels. Some of the techniques we have introduced may be applicable to

them, though we have phrased our compiler as a standalone functional program with a traditional correctness theorem. As a result, we always have a formal guarantee of compiler correctness, which is hard to be sure of in frameworks like the ones we just summarized, which do compilation with tactic scripts.

Lowering algorithms for functional languages generally must tie the storage order with the compute order to generate code, because traditional high-level, functional languages only describe compute order in a program. Thus functional-language implementations are limited in their ability to produce high-performance array code that needs to make very specific decisions regarding compute and storage order in order to produce the performance gains required in these domains. As a result, there have been efforts to introduce low-level detail such as storage into functional array languages. Examples include allowing explicit memory allocation in the functional source language [15] as well as the introduction of views in lowering to control storage expressions [12]. However, all of these approaches are unverified.

9 CONCLUSION

We provide the first rigorous proof (mechanized or otherwise) of correctness for a lowering algorithm of a functional language that is able to generate high-performance array code through the use of combinators that decouple compute and storage order. To do so, we formulated a behavioral invariant on reindexer functions in compilation to ensure that the generated imperative storage-index remappings induced by any reshape operator are well-formed. We also implement a type system to capture formally the desired safety properties of the functional tensor language ATL.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grants CCF-1846502, CCF-2217064, and CCF-2313022 / CCF-2313023 and the National Science Foundation Graduate Research Fellowship Program under Grant No. 1745302. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. 2020. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *J. Autom. Reason.* 64, 7 (oct 2020), 1287–1306. https://doi.org/10.1007/s10817-020-09559-8

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

[3] Basile Clément and Albert Cohen. 2022. End-to-End Translation Validation for the Halide Language. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 84 (apr 2022), 30 pages. https://doi.org/10.1145/3527328

[4] Nathanaël Courant and Xavier Leroy. 2021. Verified Code Generation for the Polyhedral Model. *Proc. ACM Program. Lang.* 5, POPL, Article 40 (jan 2021), 24 pages. https://doi.org/10.1145/3434321

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

[6] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. *2019 IEEE Symposium on Security and Privacy (SP)* 54, 1 (May 2019), 1202–1219. https://doi.org/10.1109/sp.2019.00005

[7] Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 154 (jun 2023), 25 pages. https://doi.org/10.1145/3591268

[8] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://ts.data61.csiro.au/publications/nicta_full_text/7494.pdf 10.1145/2535838.2535841.

[9] Peter Lammich. 2013. Automatic Data Refinement. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7998)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer, Berlin, Heidelberg, 84–99. https://doi.org/10.1007/978-3-642-39634-2_9

[10] Peter Lammich. 2019. Refinement to Imperative HOL. *J. Autom. Reason.* 62, 4 (apr 2019), 481–503. https://doi.org/10.1007/s10817-017-9437-1

[11] Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning* 43, 4 (Dec. 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[12] Zhitao Lin and Christophe Dubach. 2022. From Functional to Imperative: Combining Destination-Passing Style and Views. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (San Diego, CA, USA) *(ARRAY 2022)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3520306.3534502

[13] Amanda Liu. 2024. *Verified Lowering Artifact*. https://doi.org/10.5281/zenodo.10932109

[14] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

[15] Philip Munksgaard, Cosmin Oancea, and Troels Henriksen. 2023. Compiling a Functional Array Language with Non-Semantic Memory Information. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (Copenhagen, Denmark) *(IFL '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 13 pages. https://doi.org/10.1145/3587216.3587218

[16] Magnus O. Myreen and Scott Owens. 2012. Proof-producing synthesis of ML from higher-order logic. In *International Conference on Functional Programming (ICFP)*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, New York, NY, USA, 115–126.

[17] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional Optimizations for CertiCoq. *Proc. ACM Program. Lang.* 5, ICFP, Article 86 (aug 2021), 30 pages. https://doi.org/10.1145/3473591

[18] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. Relational Compilation for Performance-Critical Applications: Extensible Proof-Producing Translation of Functional Models into Low-Level Code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 918–933. https://doi.org/10.1145/3519939.3523706

[19] Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *IJCAR'20: Proceedings of the 9th International Joint Conference on Automated Reasoning*. Springer, Paris, France, 119–137. https://doi.org/10.1007/978-3-030-51054-1_7

[20] 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. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[21] Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-Passing Style for Efficient Memory Management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing* (Oxford, UK) *(FHPC 2017)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/3122948.3122949

Received 2023-11-16; accepted 2024-03-31