Extending the ATL Framework with Wide Bindings and Sequential Computation

by

Felix Prasanna

S.B. Computer Science and Engineering,
Massachusetts Institute of Technology, 2026

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2026

Authored by:	Felix Prasanna Department of Electrical Engineering and Computer Science May 14, 2026
Certified by:	Adam Chlipala Arthur J. Conner (1888) Professor of Computer Science Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Extending the ATL Framework with Wide Bindings and Sequential Computation

by

Felix Prasanna

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 2026 in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The ATL framework is a formally verified framework for expressing and scheduling tensor computations. This works extends the ATL framework with two new constructs: wide generation and sequential computation. Wide generation is the process of generating multiple tensors from a single loop. Wide generation enables the expression of loop fusion and unrolling in ATL, two optimizations that often serve as precursors to autovectorization. We add a wide-generation construct to ATL's deep embedding and prove its lowering correct. We also present a small case study applying loop fusion to a variance computation. Sequential computation enables the expression of programs that read from their own output. The most important consequence of this new capability is making dynamic programming, a broadly useful algorithmic technique, expressible in ATL. We add a construct for sequential computation to ATL's deep embedding and prove it correct in the 1-dimensional case.

Thesis Supervisor: Adam Chlipala
Title: Arthur J. Conner (1888) Professor of Computer Science

Acknowledgments

I would sincerely like to thank my advisor Adam Chlipala. Thank you for taking a chance on me and introducing me to the world of PL. Your sharp questions often made me reconsider my assumptions, and I am grateful for your keen eye for detail. Just as earnestly, I would like to thank my mentor Amanda Liu. I am grateful for your limitless patience as I slowly learned about Rocq, ATL, and doing research. While you witnessed my week-to-week output fluctuate, your encouragement never did, and that has meant very much to me. I would also like to thank Chai Lewgasamsarn, who previously implemented sequential computations in ATL's shallow embedding, and whose ideas I relied heavily on.

Finally, thank you to my family and friends. I am aptly named "Felix," for I am so happy and lucky to have you in my life.

Contents

Chapter 1

Introduction and Background

1.1 What is ATL?

The setting of this work is a formally verified framework for writing and scheduling tensor programs in a language called ATL (A Tensor Language) [1, 2]. Scheduling is the process of transforming a program into an equivalent but more performant form. ATL has two variants: a shallow embedding used for scheduling programs, and a deep embedding used for compiling programs to C. In a shallow embedding, language constructs are directly represented using constructs from the host language and their values are determined by evaluation according to the host language's rules. Meanwhile, in a deep embedding, programs are represented as abstract syntax trees (ASTs). An AST is assigned a value by an inductive relation or interpreter rather than the evaluation rules of the host language. In ATL's shallow embedding, "programs" are simply Gallina terms, with specific Gallina functions representing language constructs such as tensor generation and summation. Because ATL programs are just Gallina terms, they can be manipulated using **scheduling rewrites**, or equality theorems between small ATL programs. This provides a convenient way for users to schedule programs by using Rocq's built-in rewriting facilities. Importantly, after each rewrite, the resulting program is equivalent to the original program by construction. Once a user has finished scheduling their program, ATL reifies it into a value in its deep embedding—an AST. The AST is then compiled to C code via a verified procedure, and the user can run the generated C code through normal means. The deep embedding is much more convenient than the shallow embedding for verifying such a compilation procedure because it allows for reasoning about *programs* rather than *values*.

1.2 Related Work

Before diving into the details of the ATL framework, let us examine what differentiates it from other scheduling frameworks for high-performance computing. While many other frameworks exists, ATL's key distinction is that it is end-to-end formally verified.

Halide [3] is a C++ DSL used mainly in the graphics industry. Its main contributions are a DSL that (like ATL) allows a user to specify how a computation will be executed

(e.g, tiling schemes, loop fusion) and a stochastic autotuner for finding good schedules automatically. Similarly, TVM [4] is a Python DSL aimed at expressing deep learning workloads. Like Halide, TVM has explicit scheduling directives. TVM also features a learning-based schedule generator. ATL is able to express most of the same optimizations as Halide and TVM, while being fully verified.

Steuwer et al. [5] implemented a DSL for GPU programming that represents computations in a high-level functional language. Users incrementally lower computations to a lower-level representation similar to OpenCL by applying hand-proven rewrites. Their compiler then emits OpenCL code corresponding to the final lowered expression. Aside from targeting CPUs instead of GPUs, ATL programs are end-to-end machine-verified, whereas the the lowering to OpenCL and rewrite rules are not verified in Steuwer et al.'s work.

1.3 ATL's Shallow Embedding

ATL's shallow embedding consists of a number of functions for defining tensor computations. The most-central two functions are `GEN` and `SUM` for tensor generation and summation respectively. `GEN [ x <= i < y ] e` is similar to a list comprehension. It generates a `(y-x)`-element tensor whose value at index `i` is equal to the expression `e`, which is really a function from index (integer) to tensor, evaluated at value `x + i`. Under the hood, the `GEN` notation expands to `genr m n (fun i => e)`:

```
Definition genr `{TensorElem X} (x y : Z) (f : Z -> X) : list X :=
  gen_helper (Z.to_nat (y - x)) x f.
```

which in turn makes use of the following definition:

```
Fixpoint gen_helper `{TensorElem X} n x (f : Z -> X) : list X :=
  match n with
  | O => []
  | S n' => f' x :: (gen_helper n' x (fun x' => f (x' + 1)%Z))
  end.
```

Notice that the variable `i` is not a normal Gallina value, unlike `x` and `y`. Rather it is bound in `e` by a lambda, making `e` the body of a unary function. Evaluating `e` at different indices thus amounts to varying `i`. `genr` and `gen_helper` also feature the `TensorElem` typeclass constraint. All tensor types in ATL implement `TensorElem`, which mandates members such as a null value, a binary operation, and a shape. The simplest tensor type is `R`, or real numbers, which are used to present floating-point numbers. Floating-point numbers can be thought of as 0-dimensional tensors. Next, `list T` implements `TensorElem` as long as `T` does, allowing the construction of arbitrary-dimensional tensors which bottom out with values of `R`.

`SUM [ x <= i < y ] (f i)` is defined similarly, except rather than aggregating the values of `f` at each index into a list, the values are summed:

```
Fixpoint sum_helper `{TensorElem X} n x (f' : Z -> X) : X :=
  match n with
  | O => null
  | S n' => bin (f' x) (sum_helper n' x (fun x' => f' (x' + 1)%Z))
  end.

Definition sumr `{TensorElem X} (x y : Z) (f : Z -> X) : X :=
  sum_helper (Z.to_nat (y - x)) x f.
```

Summation is performed element wise if the values produced by `f` are themselves tensors, as alluded to by the function `bin` in the `S` arm of `sum_helper`. `bin` is a generic binary operation for tensors, mandated by the `TensorElem`. For scalars it is defined as addition, while for tensors it is defined as element-wise summation.

`GEN` and `SUM`, along with arithmetic operations and a construct for tensor access (denoted using `_[...]`), are enough to express useful tensor computations, such as matrix multiplication:

```
Definition matmul A B C (m1 m2 : (list (list R))) :=
  GEN [ i < A ]
      GEN [ j < C ]
          SUM [ k < B ]
              (m1 _[i;k] * m2 _[k;j])%R.
```

Since ATL programs are just Gallina terms, they can be manipulated using Ltac, Rocq's tactic language. Most notably, rewriting using specialized equality theorems (scheduling rewrites) can be used to schedule programs. Ltac manipulation is convenient for scheduling tensor programs but is less convenient for verifying a complex compilation procedure.

1.4 ATL's Deep Embedding

ATL's deep embedding represents an ATL program as an AST using the type `ATLexpr`:

```
Inductive ATLexpr :=
  | Gen (i : string) (lo hi : Zexpr) (body : ATLexpr)
  | Sum (i : string) (lo hi : Zexpr) (body : ATLexpr)
  | Guard (p : Bexpr) (body : ATLexpr)
  | Lbind (x : string) (e1 e2 : ATLexpr)
  | Concat (x y : ATLexpr)
  | Flatten (e : ATLexpr)
  | Split (k : Zexpr) (e : ATLexpr)
  | Transpose (e : ATLexpr)
  | Truncr (n : Zexpr) (e : ATLexpr)
  | Truncl (n : Zexpr) (e : ATLexpr)
  | Padr (n : Zexpr) (e : ATLexpr)
```

```
  | Padl (n : Zexpr) (e : ATLexpr)
  | Scalar (s : Sexpr).
```

The `matmul` program from before might be represented in the deep embedding as the following value of `ATLexpr`:

```
(Gen "H" (| 0 |)%z (! "A" !)%z
   (Gen "H0" (| 0 |)%z (! "C" !)%z
      (Sum "H1" (| 0 |)%z (! "B" !)%z
         (Scalar
            (Mul (Get "m1" [(! "H" !)%z; (! "H1" !)%z])
               (Get "m2" [(! "H1" !)%z; (! "H0" !)%z])))))))
```

The values `(| 0 |)%z` and `(! "A" !)%z` are notation for `ZLit 0%Z` and `ZVar "A"`, values of `Zexpr`, ATL's AST type for integer expressions. Note that `Zexpr` is not used to produce tensor values; it is only used to express indexing calculations. There is also a corresponding type `Bexpr` for boolean expressions, which are used by the `Guard` construct. Finally, the type `Sexpr` represents *scalar* expressions, which are exactly those expressions that can appear on the right-hand side of an array assignment. A scalar expression consists of tensor reads and arithmetic operations:

```
Inductive Sexpr :=
| Var (v : string)
| Get (v : string) (i : list Zexpr)
| Mul (x y : Sexpr)
| Add (x y : Sexpr)
| Div (x y : Sexpr)
| Sub (x y : Sexpr)
| Lit (r : R).
```

`ATLexpr`'s denotational semantics (fully presented in Appendix A) are determined by the relation `eval_expr`, which relates `ATLexpr`'s and `results`:

```
Inductive scalar_result :=
| SS (r: R)
| SX.

Inductive result :=
| S (z : scalar_result)
| V (v : list result).
```

A `result` is just a nested list, bottoming out with elements of type `scalar_result`. A `scalar_result` is either padding (`SX`) or a normal float (`SS`). The full type of `eval_expr` is:

```
eval_expr : context -> valuation -> expr_context -> ATLexpr -> result -> Prop
```

A `context`, commonly denoted `sh`, is an `fmap string (list Zexpr)`, or finite map from `string`s (variable names) to `list Zexpr`s (tensor dimentions). In other words, a `context` provides a shape for each tensor in scope. Likewise, an `expr_context`, commonly denoted

`ec`, is an `fmap string result` and provide a value for each tensor in scope. A `valuation`, common denoted `v`, is an `fmap string Z`. It stores the values of index variables.

1.5 ATL's Formalization of C

ATL formalizes a subset of C called `stmt`, which `ATLexpr`'s are compiled to.

```
Inductive stmt :=
| Store (t : StoreType) (v : string) (i : list (Zexpr * Zexpr)) (rhs : Sstmt)
| If (cond : Bexpr) (body : stmt)
| AllocV (var : string) (size : Zexpr)
        (* var = calloc(size) *)
| AllocS (var : string)
        (* var = 0 *)
| DeallocS (var : string)
| Free (var : string)
| For (i : string) (lo hi : Zexpr) (body : stmt)
| Seq (s1 s2 : stmt).
```

ATL formalizes if-statements (`If`), stack allocation and deallocation (`AllocS` and `DeallocS`), heap allocation and deallocation (`AllocV` and `Free`), for loops (`For`), and the semicolon operator for sequencing statements (`Seq`). Computation of tensor values happens through the `Store` constructor, which represents assigning or add-assigning the result of `rhs` to index `i` of tensor `v`. `Sstmt` is the lowered analogue of `Sexpr`, representing C expressions involving arithmetic and array reads.

The operational semantics for `stmt` are defined in terms of stacks and heaps. The stack is represented as an `fmap var R`, or mapping from local variables names to reals. The heap is represented as an `fmap var array`, or mapping from variable names to flat arrays, which are represented as mappings from indices to reals. The relation `eval_stmt` (fully presented in Appendix B) relates a starting stack and heap, a `stmt`, and the stack and heap to which the execution of the statement leads. Its full type is:

```
eval_stmt : valuation -> stack -> heap -> stmt -> stack -> heap -> Prop
```

Note that `stack` and `heap` are type aliases for `fmap var R` and `fmap var array`. The valuation type is the same as in `eval_expr`.

1.6 Lowering

Lowering (also known as compilation) is the process of converting an `ATLexpr e` into a `stmt` that computes the value of `e`. The recursive function `lower e f p asn sh` performs this task:

```
lower : ATLexpr ->
        (list (Zexpr * Zexpr) -> list (Zexpr * Zexpr)) ->
        var -> StoreType -> context -> stmt
```

Firstly, `lower` first takes an `ATLexpr` to lower. The second argument is a **reindexer**, or function that maps indices to indices, usually denoted `f`. Reindexers are used to build up the eventual index expression used by the storage statement at the base of a loop nest. Such a storage statement is generated by lowering the terminal constructor of `ATLexpr`, `Scalar`, as follows:

```
| Scalar s => Store asn p (f []) (lowerS s sh)
```

`lowerS` lowers `s`, which is a `Sexpr` to an `Sstmt`. Thus, the lowering of `Scalar` writes the value of `s` to `f []`. `f []` is influenced by the series of recursive `lower` calls bottoming out at the current call to `Scalar`. For example, a tensor generation `Gen i lo hi body` adds an additional index to the final index expression using the following reindexer:

```
(fun l => f ((ZMinus (ZVar i) lo, ZMinus hi lo)::l))
```

Thus, if our `Scalar` were nested within three `Gen`s, the call to `f []` would produce a list of length 3, which each element corresponding to one of the surrounding `Gen`s. The third argument to `lower` is the name of the output buffer, which is presumed to already have been allocated on the stack or heap (depending on the size of the computation). The fourth argument, a `StoreType`, indicates whether writes into the output tensor should be assignments or add-assigns. Finally, the last argument is the same shapes context as discussed previously in Section 1.4.

The statement of correctness for `lower` is provided in Appendix C. There are also several important preconditions to the theorem, which are detailed in [2]. For the purposes of this work, they are largely uninteresting. Informally, the final statement is that if an `ATLexpr e` evaluates to `r` in valuation `v`, shapes context `sh`, and expression context `ec`, then executing `lower e (fun l => l) p asn sh` on starting stack and heap `st` and `h` that contain the values in `ec` results in a stack `st'` and heap `h'` satisfying the following:

- If `r` is a scalar, then `h` and `h'` are identical, and `st'` is simply `st` with an additional mapping of `p` to `r`. In other words, the only change is the addition of a local variable `p` to the stack, with value `r`.
- Likewise, if `r` is a tensor, `st` and `st'` are identical. `h'` is simply `h` with one additional variable, `p`, which takes value `r`.

Chapter 2

Wide Generation

A motivating optimization for wide generation is loop fusion. Loop fusion is the process of combining two loops that share the same bounds. The following two loops:

```
for (int i = m; i < n; i++) {
    A[i] = <e1>
}
for (int i = m; i < n; i++) {
    B[i] = <e2>
}
```

can be fused into a single loop:

```
for (int i = m; i < n; i++) {
    A[i] = <e1>
    B[i] = <e2>
}
```

as long as the `<e2>` does not read from `A` (we assume `<e1>` and `<e2>` are pure, as in ATL). Fusing the two loops can lead to a speedup if there are opportunities for instruction-level parallelism between `<e1>` and `<e2>`.

Previously, this transformation was impossible in ATL because each loop nest could only contain exactly one storage statement (an assignment or add-assign). To alleviate the restriction of one storage statement per loop, we introduce a construct denoted `WIDEGEN`, which generalizes `GEN` to produce multiple values from the same loop.

2.1 Wide Generation in the Shallow Embedding

Like `GEN`, `WIDEGEN` takes loop bounds. However, instead of taking one loop body, it takes multiple and evaluates them each in a manner similar to `GEN`. Intuitively, a `GEN` corresponds to the following loop structure:

```
for (int i = m; i < n; i++) {
    output[/* storage expression */] = /* computation */
}
```

`WIDEGEN` "widens" this structure to include many different outputs produced from many different computations:

```
for (int i = m; i < n; i++) {
    output_1[/* storage expression 1 */] (= or +=) /* computation 1 */
    output_2[/* storage expression 2 */] (= or +=) /* computation 2 */
    // ...
    output_K[/* storage expression k */] (= or +=) /* computation k */
}
```

We refer to each of the logical computations happening inside the `WIDEGEN` as a lane. Each lane of a `WIDEGEN` is a generalized `GEN` or `SUM`, in the sense that it can combine the values produced each iteration either by `cons` or by addition. We do not restrict `WIDEGEN` so that each lane must produce values of the same type, so our type for representing `WIDEGEN`'s must support some degree of heterogeneity. The lanes of a `WIDEGEN` are specified using the `widegen_args` type, which encodes a heterogenous list:

```
Inductive widegen_args : list (Set * Set) -> Type :=
    | Empty : widegen_args nil
    | Store : forall (x rx : Set) `{TensorElem x} `{TensorElem rx}
                     (ls : list (Set * Set)),
        (Z -> x) ->
        (x -> rx -> rx) ->
        widegen_args ls ->
        widegen_args ((x, rx) :: ls).
```

`widegen_args` is a dependent type indexed by a list of pairs of `Set`, the type of ordinary data types (e.g. real numbers and lists). Each of these pairs corresponds to one lane of the `WIDEGEN`. The first element of the pair is the type of value produced during each iteration. The second element of the pair is the type of value produced by the lane overall. The most common situation in which these two types are different is during tensor generation, in which case the type produced each iteration is `X` and the type produced by the lane is `list X`. A lane analogous to a `GEN` of scalars would have signature `(R, list R)`.

Dependent structure notwithstanding, `widegen_args` is simply a fancy list type. The `Empty` constructor creates an empty `WIDEGEN` with no bodies, representing an empty loop. Each instance of the `Store` constructor adds another lane. `x` and `rx` are the type of values produced by the lane each iteration and the type of the final value produced by the lane, respectively. They must both implement the `TensorElem` typeclass, which is implemented by all legal ATL value types (e.g. `R` and `list X` where `` `{TensorElem X} ``). The first proper argument to `Store` is a function from integer to `x`: a loop body whose parameter represents the current loop index value. The second argument is a reduction function. It takes the value produced on a loop iteration and combines aggregate values produced by the lane on previous iterations. The initial value of the accumulator is `null`, an "empty" value that any implementor of `TensorElem` provides. The `null` value for lists is `[]` and the `null` value

for scalars is `0%R`. The most common reductions are `cons` and `bin`, which exactly emulate the behaviour of `GEN` and `SUM`.

To evaluate a `WIDEGEN`, we use the `run_widegen` function:

```
Fixpoint run_widegen {ls} (gen : widegen_args ls) (x y : Z) : hlist (map snd
ls).
Proof.
  destruct gen as [| ? ? ? ? ? f red rest].
  - exact HNil.
  - exact (HCons (widegenr x y f red) (run_widegen ls rest x y)).
Defined.
```

`run_widegen` applies the function `widegenr` to each lane and aggregates the values into an `hlist`, or heterogenous list (its implementation is not important).

`widegenr` is defined as follows:

```
Fixpoint widegen_helper `{TensorElem X} `{TensorElem RX}
  (n : nat) (i : Z)
  (fx : Z -> X) (rx : X -> RX -> RX) : RX :=
  match n with
  | O => null
  | S n' => let ax := widegen_helper n' i (fun i' => fx (i' + 1)%Z) rx in
      rx (fx i) ax
  end.

Definition widegenr `{TensorElem X} `{TensorElem RX}
  (x y : Z) (f : Z -> X) (rx : X -> RX -> RX) : RX :=
  widegen_helper (Z.to_nat (y - x)) x f rx.
```

`widegenr` is really just a wrapper around `widegen_helper`, which evaluates the loop body for each iteration and reduces the values together using the provided reduction. `widegenr` is nothing special; with a `cons` reduction it perfectly emulates `GEN`, while using `bin` perfectly emulates `SUM`. Indeed, the following theorems are trivial to prove:

```
Theorem gen_widegenr `{TensorElem X}: forall m n (f : Z -> X),
  GEN [ m <= i < n ] f i = widegenr m n f cons.

Theorem sum_widegenr `{TensorElem X} : forall n m f,
  SUM [ m <= i < n ] f i = widegenr m n f bin.
```

Now that we can actually produce values using a wide generation, we still have a slight issue: the values are contained in a heterogeneous list, which does not implement `TensorElem`. To solve this mismatch, we introduce an "unwrapper" called `let_wide2`:

```
Definition let_wide2 {X Y RX RY O}
  (wgen : widegen_args [(X, RX); (Y, RY)]) (m n : Z) (inexp : RX -> RY -> O) :=
  let (lane1, lane2) := hlist2_pairify (run_widegen wgen m n) in
  inexp lane1 lane2.
```

`let_wide2` functions as a special let binding that evaluates a wide generation and then evaluates an expression that depends on the results. The function `hlist2_pairify` converts a heterogenous list of length 2 into a pair, which we may then destructure and use. Similar functions for leaving the dependent world can be defined for any length of heterogenous list.

2.2 Wide Generation in the Deep Embedding

In the deep embedding, we introduce a constructor of `ATLexpr` which corresponds to `WIDEGEN` from the shallow embedding:

```
| Wbind (i : string)
        (lo hi : Zexpr)
        (bodies : list (string * ATLexpr * StoreType))
        (inexpr : ATLexpr)
```

Just like `Gen`, we have loop bounds and a loop variable. We also carry a list of tuples, which represent the lanes of the wide generation. Each lane is represented as a variable to which the output of that lane is assigned, a body that is executed each loop iteration, and a reduction (recall that `StoreType` is either `Assign` or `Reduce`, corresponding to `cons` and `bin`). A lane with `StoreType Assign` is assumed to be a tensor generation, while `Reduce` indicates a summation. Note that at this point, all `bin` operations have been normalized out by the reification process, meaning that any additions of tensors appear explicitly as loops, and add-assigns only occur between scalars. Thus, if the `StoreType` for a lane is `Reduce`, the corresponding body produces a scalar. Since every ATL computation outputs a single tensor, a `Wbind` cannot evaluate to some heterogenous structure containing the final lane values, thus we make them available to a final computation, `inexpr`, that can read from them. Using the `Wbind` lane values as bound variables in another expression is exactly analogous to using an unwrapper like `let_wide2` in the shallow embedding.

2.3 Semantics of `WBind`

The semantics of `WBind` are cumbersome to define but intuitive. We begin by the following variables:

```
| EvalWbind : forall v ec i lo hi inexpr r wbodies wresults wsizes ec' sh',
```

`v`, `ec`, `i`, `lo`, `hi`, and `inexpr` carry their usual meanings. Note that `sh` is already in-scope, as the entire type `eval_expr` is parametrized over it. `r` is the result that our `WBind` will evaluate to. `wbodies` is our list of wide bodies that will be executed in the loop. `wresults` is an `fmap var result`; it maps wide-body variables to the results they will be assigned after execution of the loop. Similarly, `wsizes` is an `fmap var (list Zexpr)`, or a map from variable name to tensor size. It maps each wide body variable to the size of the result assigned to it. We will use these two maps to describe the shapes and expression contexts that `inexpr` will be evaluated in, `sh'` and `ec'`.

First, we ensure some well-formedness properties of our wide generation. Each variable that receives a wide-body result must be unique:

```
no_dup (map (fun '(var, _, _) => var) wbodies) ->
```

Furthermore, the keys of `wsizes` and `wresults` must be exactly the variables that will receive wide bodies:

```
(forall var,
  In var (map (fun '(var, _, _) => var) wbodies) <-> var \in dom wresults) ->
(forall var,
  In var (map (fun '(var, _, _) => var) wbodies) <-> var \in dom wsizes) ->
```

We assign meanings to the entries of `wresults` and `wsizes`:

```
(forall var body asn,
  In (var, body, asn) wbodies ->
  exists result,
  wresults $? var = Some result /\
  eval_expr sh v ec (wexec i lo hi body asn) result) ->
(forall var body asn,
  In (var, body, asn) wbodies ->
  exists size,
    wsizes $? var = Some size /\
    size_of (wexec i lo hi body asn) size /\
    match asn with
    | Assign => True
    | Reduce => size_of body []
    end) ->
```

Each tuple `(var, body, asn)` in `wbodies` has a corresponding mapping in `wresults` and `wsizes`. The entry in `wresults` for `var` is the value that `wexec i lo hi body asn` evaluates to. Analogously, the entry in `wsizes` for `var` is the size of the result it will contain. `wexec` serves a similar function to `widegenr` and is defined as:

```
Definition wexec i lo hi body asn :=
  match asn with
  | Assign => Gen i lo hi body
  | Reduce => Sum i lo hi body
  end.
```

It simply produces a `Gen` or `Sum` depending on the `StoreType` passed in.

Next, we use `wsizes` to define the relationship between `sh` and `sh'`, the starting shapes context and the shapes context that includes the lanes of the `Wbind`. If a variable is in `wsizes`, then it must not be present in `sh`, and it must be present in `sh'`. If a variable is not present in `wsizes`, then `sh'` and `sh` match on that variable. In other words, the domains of `sh` and `wsizes` are disjoint, and `sh'` is the union of the two maps.

```
(forall var size,
  wsizes $? var = Some size ->
  sh' $? var = Some size /\ sh $? var = None) ->
(forall var, wsizes $? var = None -> sh' $? var = sh $? var) ->
dom sh \cap dom wsizes = constant nil ->
dom sh \cup dom wsizes = dom sh' ->
```

A analogous relationship applies for `ec`, `wresults`, and `ec'`.

```
(forall var result,
  wresults $? var = Some result ->
  ec' $? var = Some result /\ ec $? var = None) ->
(forall var, wresults $? var = None -> ec' $? var = ec $? var) ->
dom ec \cap dom wresults = constant nil ->
dom ec \cup dom wresults = dom ec' ->
```

We have one final well-formedness constraint on `wbodies` that prevents variable shadowing.

```
Forall (fun '(var, body, _) =>
  ec $? var = None /\
  Forall (fun '(_, wbody, _) => ~ var \in vars_of wbody) wbodies /\
  ~ var \in vars_of inexpr /\
  vars_of body \cap vars_of inexpr = constant nil
) wbodies ->
```

The `vars_of` function computes the set of bound variables in an expression. An expression can be bound by either a let binding or wide binding. Thus, this constraint states the following:

1. Each lane variable is not already bound to a value
2. Each lane variable is not a bound variable in any lane body
3. Each lane variable is not bound in `inexpr`
4. Each lane body shares no bound variables with `inexpr`.

Finally, if `inexpr` evaluates to `r` in `sh'`, `v`, and `ec'`:

```
eval_expr sh' v ec' inexpr r ->
```

then the entire wide generataion evaluates to `r`

```
eval_expr sh v ec (Wbind i lo hi wbodies inexpr) r
```

2.4 Lowering of `WBind` and Correctness

`WBind` is lowered intuitively: execute all lanes together in a loop and then execute the `inexpr`.

In Rocq, we begin by generating a list of allocations and deallocations for each of the wide-body variables:

```
| Wbind i lo hi bodies inexpr =>
    let allocs := map (fun '(output, body, store_type) =>
      match store_type with
      | Assign => AllocV output (flat_sizeof (Gen i lo hi body))
      | Reduce => AllocS output
      end
    ) bodies
    in
    let deallocs := map (fun '(output, body, store_type) =>
      match store_type with
      | Assign => Free output
      | Reduce => DeallocS output
      end
    ) bodies
    in
```

We then recursively lower each wide body, adding a dimension to the reindexer if the wide body produces a tensor or using the identity reindexer otherwise.

```
    let lowered_bodies := map (fun '(output, body, store_type) =>
      match store_type with
      | Assign =>
        (lower body
          (fun l => ((ZMinus (ZVar i) lo, ZMinus hi lo)::l)) output Assign sh)
      | Reduce => (lower body (fun l => l) output Reduce sh)
      end
    ) bodies
    in
```

We generate the shapes context that `inexpr` will be lowered in

```
    let new_sh :=
      fold_left
        (fun sh '(output, body, store_type) =>
            match store_type with
            | Assign => sh $+ (output, (ZMinus hi lo)::(sizeof body))
            | Reduce => sh $+ (output, []) (* bin has been normalized out *)
            end)
        bodies
        sh
    in
```

Finally, the `seqify` helper converts a list of `stmt`s into a single `stmt` that executes the statements of the list in sequence. The final `stmt` is thus:

```
    Seq
      (seqify allocs)
      (Seq
        (Seq
          (For i lo hi (seqify lowered_bodies))
```

```
      (lower inexpr f p asn new_sh))
    (seqify deallocs))
```

2.4.1 Correctness

To prove correctness, we mandate additional well-formedness properties of a `WBind`. Firstly, the wide bodies must not "interfere" with each other in any way. Specifically, they may not write to any common variables that are read from or written to by any other wide body. Note that this is also the condition for loop fusion: to fuse any number of loops, no loop may read from or write to any variable modified by another loop. We formalize this non-interference property as follows:

```
Inductive bodies_dont_alias : list wbody_t -> Prop :=
  | BodiesDontAliasNil :
      bodies_dont_alias []
  | BodiesDontAliasCons : forall var body asn wbodies,
      Forall (fun '(wvar, wbody, _) =>
        read_set_expr body \cap write_set_expr wbody wvar = constant nil /\
        write_set_expr body var \cap read_set_expr wbody = constant nil /\
        write_set_expr body var \cap write_set_expr wbody wvar = constant nil
      ) wbodies ->
      bodies_dont_alias wbodies ->
      bodies_dont_alias ((var, body, asn) :: wbodies)
  .
```

The functions `read_set_expr` and `write_set_expr` calculate the read and write sets of an `ATLexpr`, as one would expect. Note that `write_set_expr` also takes an additional parameter, the name of the buffer or variable to which the `ATLexpr` will be output. This variable is also included in the write set. An empty list of bodies is automatically well-formed. If a list `wbodies` is well-formed, then we can add another body `body` and maintain well-formedness as long as:

1. `body` does not read from any variable written to by an element of `wbodies`
2. No element of `wbodies` reads from a variable written to by `body`
3. No element of `wbodies` writes to a variable written to by `body`.

In short, the wide bodies may read from shared variables, but they may not write to shared variables. Along with a few more well-formedness conditions, we have the following:

```
Definition well_formed_wbodies
  (p : var) (wbodies : list (var * ATLexpr * StoreType)) :=
  bodies_dont_alias wbodies /\
  no_dup (map (fun '(var, _, _) => var) wbodies) /\
  Forall (fun '(var, _, _) =>
    Forall (fun '(_, wbody, _) => ~ var \in vars_of_with_gets wbody) wbodies
  ) wbodies /\
  Forall (fun '(var, _, _) => var <> p) wbodies
```

We havve further stipulated that the wide body variables are unique and not bound anywhere. Furthermore, they may not be equal to the variable we are writing our output into.

Now, we are ready to move on to the proof of correctness. The first step is to show equivalence between our lowering to another lowering, which we will call the "sequential lowering". The sequential lowering is defined in the function `lower_sequential_widegen`. Defining this auxiliary lowering is useful because proving `lower_sequential_widegen` correct will be easier than proving `lower` correct directly. We will then show that if `eval_stmt v st h (lower e f p asn sh) st' h'`, then `eval_stmt v st h (lower_sequential_widegen e f p asn sh) st' h'`, completing the proof of correctness for `lower`.

`lower_sequential_widegen` is identical to `lower` on every constructor except `WBind`. On `WBind`, it executes each wide body in its own loop, sequentially:

```
| Wbind i lo hi bodies inexpr =>
    let fix aux bodies sh :=
      match bodies with
      | nil => lower_sequential_widegen inexpr f p asn sh
      | (x, wbody, asn) :: rest =>
        match asn with
        | Assign => seqify
            [ AllocV x (flat_sizeof (Gen i lo hi wbody))
            ; For i lo hi
                  (lower_sequential_widegen wbody
                    (fun l => ((ZMinus (ZVar i) lo, ZMinus hi lo)::l))
                    x Assign sh)
            ; aux rest (sh $+ (x, sizeof (Gen i lo hi wbody)))
            ; Free x
            ]
        | Reduce => seqify
            [ AllocS x
            ; For i lo hi
                  (lower_sequential_widegen wbody (fun l => l) x Reduce sh)
            ; aux rest (sh $+ (x, []))
            ; DeallocS x
            ]
        end
      end
    in
    aux bodies sh
```

This produces an unfused loop structure:

```
<allocate output_1>
for (int i = m; i < n; i++) {
    output_1[/* storage expression 1 */] (= or +=) /* computation 1 */
}
```

```
<allocate output_2>
for (int i = m; i < n; i++) {
    output_2[/* storage expression 2 */] (= or +=) /* computation 2 */
}
// ...
<allocate output_k>
for (int i = m; i < n; i++) {
    output_k[/* storage expression k */] (= or +=) /* computation k */
}
<inexpr>
<deallocations>
```

Using our non-interference properties, we are able to prove the following:

```
Theorem lower_lower_sequential : forall e v st h f p asn sh st' h',
  well_formed_wbinds p e ->
  eval_stmt v st h (lower                    e f p asn sh) st' h' ->
  eval_stmt v st h (lower_sequential_widegen e f p asn sh) st' h'.
```

In other words, we are able to unfuse the loop fusion generated by `WBind`. The workhorse for proving this theorem is the following lemma:

```
Theorem reorder : forall c1 c2 v1 st h st1 h1,
  read_set_stmt c1 \cap write_set_stmt c2 = constant [] ->
  read_set_stmt c2 \cap write_set_stmt c1 = constant [] ->
  write_set_stmt c1 \cap write_set_stmt c2 = constant [] ->
  eval_stmt v1 st h c1 st1 h1 ->
  forall v2 st' h',
  eval_stmt v2 st1 h1 c2 st' h' ->
    exists st2 h2,
    eval_stmt v2 st  h  c2 st2 h2 /\
    eval_stmt v1 st2 h2 c1 st' h'.
```

`reorder` states that if `stmts` `c1` and `c2` do not write to any common variables, then they can be executed in any order and produce the same result. We invoke the noninterference properties we mandate for `WBinds` during the proof of `reorder`.

Finally we prove `lower_lower_sequential` correctly lowers `Wbind i lo hi wbodies inexpr` by induction on `wbodies`. The base case follows immediately from our IH for `inexpr`. The inductive step is similar to the lowering proof for let bindings, since the sequential lowering for

```
WBind i lo hi ((var, body, asn) :: bodies) inexpr
```

is the same as for

```
let var := wexec i lo hi body asn in
  (WBind i lo hi bodies inexpr)
```

2.5 Variance using Wide Generation

In this section, we present a small case study that uses wide generation to schedule a variance program. The variance of a random variable X is defined as

$$\begin{aligned}\mathrm{Var}(X) &= \mathbb{E}[(X - \mathbb{E}[X])^2] \\ &= \mathbb{E}[X^2 - 2X\mathbb{E}[X] + \mathbb{E}[X]^2] \\ &= \mathbb{E}[X^2] - \mathbb{E}[2X\mathbb{E}[X]] + \mathbb{E}[\mathbb{E}[X]^2] \\ &= \mathbb{E}[X^2] - 2\mathbb{E}[X]^2 + \mathbb{E}[X]^2 \\ &= \mathbb{E}[X^2] - \mathbb{E}[X]^2\end{aligned}$$

Calculating a sample variance using the first formulation requires two passes over the data: one to calculate the expectation of X and a second to calculate the outer expectation. A program that executes the 2-pass strategy might look like:

```
int sum = 0;
for (int i = 0; i < n; i++) {
  sum += samples[i];
}
int mean = sum / n;
int squared_deviation = 0;
for (int i = 0; i < n; i++) {
  squared_deviation += (samples[i] - mean) * (samples[i] - mean);
}
int variance = squared_deviation / n;
```

The second formulation can naturally be calculated using a single loop that simultaneously calculates the expectations of X and X^2. We can use a wide generation to perform this simultaneous summation. We first rearrange terms slightly to reflect the actual loop structure we will use:

$$\begin{aligned}\mathrm{Var}(X) &= \mathbb{E}[X^2] - \mathbb{E}[X]^2 \\ &= \frac{\sum_{i=1}^{n} x_i^2}{n} - \left(\frac{\sum_{i=1}^{n} x_i}{n}\right)^2 \\ &= \frac{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}{n^2}\end{aligned}$$

Now, this exact expression can be encoded using a wide generation. Each summation corresponds to a lane:

```
let_wide2
  (Store (fun i : Z => l _[ i] * l _[ i]) Rplus
     (Store (fun i : Z => l _[ i]) Rplus
```

```
        Empty))
  0 (Z.of_nat n)
  (fun sum_squared sum =>
   (sum_squared * IZR (Z.of_nat n) - sum * sum)
    / (IZR (Z.of_nat n) * IZR (Z.of_nat n)))
```

`n` is the number of samples, which are stored in `l`. This transformation is verified.

2.6 Loop Unrolling using `unhify`

We now revisit wide generations in the shallow embedding to explore a different use case: loop unrolling.

Loop unrolling is another common transformation during optimization. It is the process of reducing the trip count of a loop by a constant factor of k by replicating the loop body k times. Assume for simplicity that in all examples the total trip count N is divisible by $k = 4$, our example unroll factor. Consider the following loop that copies one buffer to another:

```
for (int i = 0; i < N; i++) {
    A[i] = B[i];
}
```

Unrolling this loop by a factor of 4 would produce the following program:

```
for (int i = 0; i < N; i += 4) {
    A[i] = B[i];
    A[i + 1] = B[i + 1];
    A[i + 2] = B[i + 2];
    A[i + 3] = B[i + 3];
}
```

We now have a loop with exactly one quarter the trip count of the initial loop and a body four times larger. Loop unrolling can often reveal vectorization opportunities to the compiler. Furthermore, it can introduce instruction-level parallelism. In the previous example, all writes from `B` to `A` are independent, so eliminating the loop test in between each chunk of 4 moves can allow the CPU to issue them in parallel.

Notice that the unrolled program is simply the following 4 loops fused together:

```
for (int i = 0; i < N; i += 4) {
    A[i] = B[i];
}
for (int i = 0; i < N; i += 4) {
    A[i + 1] = B[i + 1];
}
for (int i = 0; i < N; i += 4) {
    A[i + 2] = B[i + 2];
}
```

```
for (int i = 0; i < N; i += 4) {
    A[i + 3] = B[i + 3];
}
```

The appearance of loop fusion is a hint to us that wide generation may be useful. However, we will need to "unwrap" the values it produces differently from before. In the case of loop unrolling, each lane is performing the same computation, so we know that the heterogeneous list of results produced by `run_widegen` will actually be convertible into a normal Rocq list. We introduce `unhify` (un-heterogeneous-ify) to perform this conversion:

```
Fixpoint unhify
  (_ty ty : Set) n (hls : hlist (map snd (repeat (_ty, ty) n))) : list ty.
Proof.
  destruct n eqn:Hn.
  - exact nil.
  - dependent destruction hls. exact (x0 :: unhify _ty ty n0 hls).
Defined.
```

The type signature of `hls` guarantees that it is actually homogeneous, since each element of `repeat (_ty, ty) n` is identical, and the same will be true after applying `map snd`. `unhify` recursively extracts each element and produces a normal `list ty`.

We are now ready to express loop unrolling. We begin with a `WIDEGEN` that performs k loop iterations per `WIDEGEN` iteration.

```
Fixpoint unroll_helper `{TensorElem X} `{TensorElem RX}
  (k : nat) (lane : Z) (iters_left : nat) (f : Z -> X) (red : X -> RX -> RX)
  : widegen_args (repeat (X, RX) iters_left) :=
  let kz := Z.of_nat k in
  match iters_left return widegen_args (repeat (X, RX) iters_left) with
  | O => Empty
  | S iters_left' => Store
      (fun i => f ((i * kz) + lane)%Z)
      red
      (unroll_helper k (lane + 1)%Z iters_left' f red)
  end.

Definition unroll `{TensorElem X} `{TensorElem RX}
  (k : nat) (f : Z -> X) (red : X -> RX -> RX)
  : widegen_args (repeat (X, RX) k) :=
  unroll_helper k 0 k f red.
```

The result of `unroll k f red` is a `WIDEGEN` with k lanes, which is only meant to be executed for one iteration. `run_widegen (unroll k f bin) i (i + 1)%Z` produces an `hlist` whose jth element is `f (i * k + j)`. Thus, applying `unhify` to it produces a list with the loop values on iterations $[i \cdot k, (i+1) \cdot k)$. Since each `WIDEGEN` represents a single loop, we were able to compute k consecutive loop values in one iteration, as desired. We prove the following

rewrite that allows the user to introduce loop unrolling by repeatedly running `unroll k f bin` for one iteration at a time:

```
Theorem flatten_soa2aos_unroll `{TensorElem X}
  : forall oi o k x y f (s : shape),
  (0 <= x)%Z ->
  (x < y)%Z ->
  0 < k ->
  (0 <= o)%Z ->
  (x = o * Z.of_nat k)%Z ->
  (Z.of_nat oi * Z.of_nat k = y - x)%Z ->
  (forall x, consistent (f x) s) ->
  GEN [ x <= i < y ] f i
  =
  flatten
    (GEN [ o <= oi0 < o + Z.of_nat oi ]
      unhify X X k (run_widegen (unroll k f bin) oi0 (oi0 + 1))).
```

The variable `oi` represents the number of "outer iterations," or iterations of our unrolled loop. The hypotheses simply ensure $x - y$ is divisible by k and that the starting index for the unrolled loop corresponds to the start index for the original loop. Since each `unhify` produces a list, the outer `GEN` on the right-hand side produces a list of lists. We know that the ith iteration of its inner `GEN` produces the values of `f` on $[i \cdot k, (i+1) \cdot k)$. Therefore, flattening the outer `GEN` produces a single list with the values of `f` on $[o \cdot k, (o + oi) \cdot k) = [x, y)$, as desired.

Chapter 3

Sequential Computation

The following sections build on research performed by Chai Lewgasamsarn. Chai implemented a frontend construct for sequential computation using the evaluation procedure presented in Section 3.1. The proof of termination of sequential computations presented in Section 3.3 is largely based on the proof of termination he implemented for the frontend construct. We reuse the ideas of `DependsOn`*,* `NotDependsOn`*,* `Ready`*, and* `DepChain` *from his proof.*

Oftentimes, a recursive computation can be effectively memoized using an array. This technique stores the results of subproblems in a data structure and retrieves those back instead of making recursive calls that have already been performed. An array is a common choice for computations that recurse on integers because it is natural to store the value of the computation on values $i_1, ..., i_2$ at indices $[i_1, ..., i_2]$. Consider the toy application of calculating Fibonacci numbers, which are defined by the following recurrence:

$$F(0) = 1$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

Calculating the nth Fibonacci number requires an exponential number of calls without any form of memoization. However, if we maintain a table of length n and write the ith Fibonacci number into the ith cell as we go, we can perform our calculation using $O(n)$ operations. We do so by retrieving $F(i)$ from the table rather than recursively computing $F(i)$ during the process of computing a larger Fibonacci number. The following is a simple program demonstrating this technique:

```
int *output = calloc(n, sizeof(int));
output[0] = 1;
output[1] = 2;
for (int i = 2; i < n; i++) {
    output[i] = output[i-1] + output[i-2];
}
```

In this program, the expression used to generate values of the output tensor includes a reference to that very output tensor. We call such a reference to the tensor being generated

a "lookback." We call programs with lookbacks **sequential computations**, since their indices must be computed in a sequence that respects the dependencies between then. In the Fibonacci example, calculating the indices in decreasing order would produce incorrect results, since upon calculating $F(i)$, the values at indices $i-1$ and $i-2$ of the table do not yet contain $F(i-1)$ and $F(i-2)$. Previously, programs with lookbacks were not expressible in ATL because the tensor being generated did not have access to itself. Specifically, the `EvalGenStep` constructor of `eval_expr` does not bind anything new into either the expression or shape contexts:

```
| EvalGenStep : forall ec v lo hi loz hiz i body l r,
    eval_Zexpr_Z v lo = Some loz ->
    eval_Zexpr_Z v hi = Some hiz ->
    (loz < hiz)%Z ->
    ~ i \in dom v ->
    ~ contains_substring "?" i ->
    eval_expr sh (v $+ (i,loz)) ec body r ->
    eval_expr sh v ec (Gen i (ZPlus lo (ZLit 1%Z)) hi body) (V l) ->
    eval_expr sh v ec (Gen i lo hi body) (V (r::l))
```

Therefore, there is no way for the execution of a `Gen` to read values computed on previous iterations. To express sequential computations, we introduce a new construct called `GenRec` for evaluating sequential computations, with completely separate semantics.

3.1 A Different Way of Evaluating Tables

The lookbacks in the Fibonacci program have a key property: each lookback only reads indices that are strictly less than the current index. We can define a relation $D(x, y)$ that is true if index x reads from index y during its execution—x depends on y. In the Fibonacci case, we have that $D(x, y) \Rightarrow y < x$. Therefore, if we calculate each index in increasing order, each index will read correct values during its lookbacks. More precisely, during the execution of index i, for all $j < i$, the table at j will contain $F(j)$.

In general, if the relation D is acyclic, then there is a topological order on the indices. Executing the indices in reverse topological order always produces the correct results, since during the execution of any index, its transitive dependencies have all been computed. An acyclic dependency structure also motivates the following process, which always executes the indices in the same order but eventually produces the correct result:

1. Initialize the table T arbitrarily
2. Run the computation, generating a new table T', with lookbacks reading from T rather than T'
3. If $T = T'$, stop. Otherwise, set $T := T'$
4. Return to Step 2

The value of the computation is the final T. A concrete example is helpful for gaining an intuition as to why this process is correct.

Suppose we would like to calculate the first 5 Fibonacci numbers. We initialize our table to all 0′s for simplicity (though any initialization will do):

Iteration 0

0	0	0	0	0

Now, we run our Fibonacci program, with lookbacks instead reading from the table we just initalized (Iteration 0). This produces the following table:

Iteration 1

1	1	0	0	0

Indices 0 and 1 have been set the the correct value because they do not look back. Meanwhile, indices 2, 3, and 4 are still 0 because their lookbacks read 0′s from Iteration 0. Now, we iterate this process, running the Fibonacci program again, but with lookbacks reading from Iteration 1, resulting in the following table:

Iteration 2

1	1	2	1	0

As before, indices 0 and 1 are correct. Even better, index 2 is also correct, since upon looking back, it read the correct values of indices 0 and 1 from Iteration 1. Iterating again, we calculate:

Iteration 3

1	1	2	3	3

Once more, we calculate:

Iteration 4

1	1	2	3	5

We now have the correct value in every index. Furthermore, if we iterate our table once again, we find we've reached a fixed point:

Iteration 5

1	1	2	3	5

Intuitively, each time we rerun the entire computation, at least one more index is set the correct value. On the first iteration, at least one index becomes ready, because there must be at least one index with no dependencies for the dependency structure to be acyclic. Suppose every index has a dependency for the sake of contradiction. Starting with any index i, we select one of its dependencies i'. We then select a dependency of i' and continue indefinitely, since every index has a dependency. However, there are a finite number of indices, so we must eventually reach i again, contradicting the fact that the dependencies are acyclic. Thus, after the first iteration of the entire computation, all indices with no dependencies attain their final correct values. On the next iteration, all indices that only depend on indices with 0 dependencies become correct, and so forth for each iteration. Once each index becomes correct, the process reaches a fixpoint.

Note that the existence of an index or indices with no dependencies implies that the initialization of the table is irrelevant—indices with no dependencies attain the same value under any initialization. Thus, on the second iteration, all lookbacks read the same values, regardless of initialization, and so on and so forth for all subsequent iterations.

This iterative process is useful because given a computation with lookbacks, we need not know anything about its dependency structure to assign it a value.

3.2 `GenRec`

`GenRec` is a new constructor of `ATLexpr`, defined as follows:

```
| GenRec (out : string) (i : string) (lo hi : Zexpr) (body : ATLexpr)
         (inexpr : ATLexpr)
```

`GenRec` carries the parameters for a sequential computation (`i`, `lo`, `hi`, `body`) along with two additional parameters: `out` and `inexpr`. The sequential computation defined in `body` is executed within the loop bounds `lo` and `hi`, using loop variable `i`. Then, the result is bound to the variable `out` and made visible to `inexpr` as it executes. The final value of the `GenRec` is the value of `inexpr`. We cannot simply return the value of the sequential computation because lowering is performed in a destination-passing style. Sequential computations contain references to their output buffers, so the name of the output buffer is fixed ahead of time. Indeed, the reason for the parameter `out` is to make sure an allocation with the correct name is created for the sequential computation. However, the function `lower` takes as an argument the name of the output buffer for the entire computation, so we cannot guarantee that the output buffer has the name the sequential computation expects. Therefore, we bind the result of the sequential computation to `out` and allow another expression read it. If the sequential computation itself is the ultimate desired value, the secondary expression can just be a copy.

With this in mind, `GenRec` is lowered as follows:

```
| GenRec out i lo hi body inexpr=>
    (* Inlined lowering of Lbind out (Gen i lo hi body) inexpr *)
    Seq (AllocV out (flat_sizeof (Gen i lo hi body)))
      (Seq
        (For i lo hi
          (lower body (fun l => ((ZMinus (ZVar i) lo, ZMinus hi lo)::l))
            out Assign (sh $+ (output,sizeof (Gen i lo hi body)))))
        (Seq (lower inexpr f p asn (sh $+ (output,sizeof (Gen i lo hi body))))
          (Free out)))
```

Firstly, a buffer with the correct name is allocated for the sequential computation to read from and write to. Secondly, the sequential computation is executed as a normal for loop. Finally, `inexpr` runs with access to the result, then the buffer holding the result is freed. The lowering for `GenRec i lo hi body inexpr` is exactly the same as the lowering for `Lbind out (Gen i lo hi body) inexpr`.

Meanwhile, `GenRec`'s semantics are defined using the iterative process detailed earlier:

```
Inductive repeat_eval (sh : context) :
  result -> var -> valuation -> expr_context -> ATLexpr -> result -> nat ->
Prop :=
  | RepeatEvalSingleton : forall init out e v ec,
    result_has_shape init
      (map Z.to_nat (map (eval_Zexpr_Z_total $0) (sizeof e))) ->
    repeat_eval sh init out v ec e init 0
  | RepeatEvalStep : forall init rmid n out e v ec r,
    result_has_shape init
      (map Z.to_nat (map (eval_Zexpr_Z_total $0) (sizeof e))) ->
    eval_expr (sh $+ (out, sizeof e)) v (ec $+ (out, init)) e rmid ->
    repeat_eval sh rmid out v ec e r n ->
    repeat_eval sh init out v ec e r (S n)
with eval_expr (sh : context) :
  valuation -> expr_context -> ATLexpr -> result -> Prop :=
| EvalGenRec : forall v ec i lo hi body r out inexpr rsz r' rs,
  size_of (Gen i lo hi body) rsz ->
  eval_Zexprlist $0 rsz rs ->
  ec $? out = None ->
  ~ out \in vars_of body /\ ~ out \in vars_of inexpr ->
  vars_of body \cap vars_of inexpr = constant nil ->
  repeat_eval sh (gen_pad (List.map Z.to_nat rs)) out v ec (Gen i lo hi body) r
    (length (mesh_grid (map (eval_Zexpr_Z_total $0) rsz))) ->
  eval_expr (sh $+ (out, rsz)) v (ec $+ (out, r)) inexpr r' ->
  eval_expr sh v ec (GenRec out i lo hi body inexpr) r'
    ... (* rest of eval_expr *)
```

`repeat_eval` formalizes the notion of iterating a sequential computation many times, as described in Section 3.1. Each iteration is really an application of `eval_expr`, so `repeat_eval` and `eval_expr` are defined mutually. `repeat_eval sh init out v ec e r n`

means that starting with `out` initialized to `init`, n iterations of `e` produce the result `r`. `ResultEvalSingleton` states that evaluating `e` zero times produces the initialization value as a result. `ResultEvalStep` states that if `e` evaluates to `rmid` with `out` initialized to `init`, and iterating `e` n more times with the initialization `rmid` produces result `r`, then iterating `e` $n+1$ times with initialization `init` produces `r`. To evaluate a `GenRec`, we iterate its sequential computation n times, where n is the size of the tensor produced by the sequential computation. Then, we evaluate `inexpr` with the result of the sequential computation stored in the expression context, and that is our final result.

3.3 Termination of Sequential Computations

Earlier, in Section 3.1, we defined the meaning of a sequential computation to be the fixpoint of iterating it, but in the semantics of `GenRec`, we iterate a set number of times determined by the size of the sequential computation. We prove in Rocq that for any **well-founded** sequential computation, a number of iterations equal to the size of the computation is sufficient to reach a fixed point, making our semantics valid. Well-foundedness is simply the property that no index depends transitively on itself, in other words, that the program's dependencies are acyclic.

In Rocq, we define dependence in terms of independence. An index `indep_index` is independent of an index `differing_index` if given two initializations that differ only on the index `differing_index`, running the sequential computation once on each initialization produces results that do not differ on `indep_index`. We formalize the notion of independence as follows:

```
Definition NotDependsOn
  (indep_index differing_index : list Z) :=
  let size := (map Z.to_nat (map (eval_Zexpr_Z_total $0) s)) in
  forall init1 init2 r1 r2,
    result_has_shape (V init1) size ->
    result_has_shape (V init2) size ->
    InBounds indep_index ->
    InBounds differing_index ->
    (forall index,
      InBounds index ->
      index <> differing_index ->
      result_lookup_Z index (V init1) = result_lookup_Z index (V init2)) ->
    eval_expr (sh $+ (out, sizeof e)) v (ec $+ (out, (V init1))) e (V r1) ->
    eval_expr (sh $+ (out, sizeof e)) v (ec $+ (out, (V init2))) e (V r2) ->
    result_lookup_Z indep_index (V r1) = result_lookup_Z indep_index (V r2).
```

Note that `e` (the sequential computation itself), `s` (its shape), `sh`, `v`, `ec`, `out`, and `init` are section variables. The `result_has_shape` hypotheses enforce that each initialization has the same shape as `e`. The `InBounds` hypotheses enforce that `indep_index` and `differing_index` are valid indices into `e`. The next hypothesis ensures that `V init1` and `V init2` only differ

on `differing_index`. Finally, if `e` evaluates to `V r1` and `V r2` on the two initializations, then reading each at `indep_index` produces the same result. Note that `NotDependsOn` to be useful, `e` must evaluate to a result under both `init1` and `init2`. However, `eval_expr` is not total because it forbids dividing by 0. Conservatively, we also mandate that `e` not contain divisions (using a section hypothesis). This makes `eval_expr` total. A finer-grained analysis is possible in which we only forbid divisions by values derived from the initialization. The property `DependsOn` is simply the negation of `NotDependsOn`:

```
Definition DependsOn dep_index differing_index :=
  ~ NotDependsOn dep_index differing_index.
```

Next, we define a relation for extending a relation to its transitive closure:

```
Inductive tc {A} (R : A -> A -> Prop) : A -> A -> Prop :=
| TcBase : forall x y, R x y -> tc R x y
| TcTrans : forall x y z, tc R x y -> tc R y z -> tc R x z.
```

Finally, we define a computation to be well-founded if no index transitively depends on itself:

```
Definition WellFounded := forall i, ~ (tc DependsOn i i).
```

Having defined well-foundedness, we can now state our theorem that a sequential computations reaches a fixpoint after a number of iterations equal to its size:

```
Theorem everything_ready : forall r1 r2,
  WellFounded ->
  repeat_eval sh init out v ec e (V r1)
    (length (mesh_grid (map (eval_Zexpr_Z_total $0) s))) ->
  repeat_eval sh init out v ec e (V r2)
    (S (length (mesh_grid (map (eval_Zexpr_Z_total $0) s)))) ->
  r1 = r2.
```

`mesh_grid` computes the set of valid indices into a tensor with shape `map (eval_Zexpr_Z_total $0) s`. Note that `eval_Zexpr_Z_total` evaluates a `Zexpr` into an integer. By hypothesis, our sequential computation has shape `s`, so the length of the `mesh_grid` is its size. To prove this statement, we introduce some more machinery. An index is deemed `Ready` after `n` iterations if executing the sequential computation another time leaves it fixed.

```
Definition Ready index n :=
  forall r1 r2,
    repeat_eval sh init out v ec e (V r1) n ->
    repeat_eval sh init out v ec e (V r2) (S n) ->
    result_lookup_Z index (V r1) = result_lookup_Z index (V r2).
```

Secondly, we formalize the idea of dependency chains, which represent transitive dependencies explicitly. A list of indices forms a dependency chain if each element of the list depends on the next (excluding the last index). Thus, any element in the dependency

chain that precedes another element transitively depends on that element. The following relation determines whether a list is a dependency chain:

```
Inductive DepChain : list (list Z) -> Prop :=
  | DepChainBase : forall i1 i2,
      DependsOn i1 i2->
      DepChain [i1; i2]
  | DepChainLink : forall i1 i2 chain,
      DependsOn i1 i2 ->
      DepChain (i2 :: chain) ->
      DepChain (i1 :: i2 :: chain)
  .
```

Our first proof step is to show that an index becomes `Ready` after a number of iterations equal to the length of the longest dependency chain it heads, plus one. After all but the last iteration, all the index's dependencies are ready, and on the last iteration, it attains its final value. Next, we bound the length of a dependency chain. Each dependency chain must consist of unique indices, otherwise an index would transitively depend on itself, contradicting well-formedness. Also, only in-bounds indices can be dependencies, since out-of-bounds accesses always return zero. Therefore, by the Pigeonhole Principle, a dependency chain has length at most the size of the sequential computation. Thus, each index must be `Ready` after this many iterations. If all indices are `Ready` after this many iterations, then from one execution of `e` to the next, no index changes, and hence the sequential computation has reached a fixed point.

3.4 Proof of Lowering of 1D Case

We prove the lowering of `GenRec` correct for the 1-dimensional case, which is when our sequential computation takes the form `Gen i lo hi (Scalar s)`. Recall that the sequential program is lowered into a normal for loop that computes `s` each iteration, while being able to read previously computed values. We will henceforth refer to `Gen i lo hi (Scalar s)` as `e` for short and its lowering as `c` for short. We first define a left-associative version of `repeat_eval` called `repeat_eval_left_assoc`. `repeat_eval_left_assoc` is useful because we want to reason about how the value of the sequential computation changes from one iteration to the next (much like how the value of an output tensor changes from one loop iteration to the next).

Then, we introduce a variation of `repeat_eval_left_assoc` that masks off all but the first n elements of the initialization on the nth iteration of `e`.

```
Inductive repeat_eval_left_assoc_restricted (sh : context) :
  result -> var -> valuation -> expr_context -> ATLexpr -> result -> nat -> Prop
  :=
  | RepeatEvalLeftRestrictSingleton : forall init out e v ec,
      result_has_shape init
```

```
      (map Z.to_nat (map (eval_Zexpr_Z_total $0) (sizeof e))) ->
    repeat_eval_left_assoc_restricted sh init out v ec e init 0
| RepeatEvalLeftRestrictStep : forall init rmid n out e v ec r,
    result_has_shape init
      (map Z.to_nat (map (eval_Zexpr_Z_total $0) (sizeof e))) ->
    repeat_eval_left_assoc_restricted sh init out v ec e (V rmid) n ->
    eval_expr (sh $+ (out, sizeof e)) v (ec $+ (out, (V (restrict n rmid))))
                                                    (*^^^^^^^^^^^^^^^^*)
      e r ->
    repeat_eval_left_assoc_restricted sh init out v ec e r (S n)
.
```

The `restrict` function is similar to `firstn`, which takes a prefix of a list. Unlike `firstn` however, `restrict` replaces any values after the desired prefix with `0%R`. Since the output tensor is initialized to zeroes, this modification ensures the nth initialization `e` receives matches the output tensor after n iterations on the last $hi - lo - n$ indices

We then show that for programs whose lookbacks always look back on strictly smaller indices[1], `repeat_eval_left_assoc` implies `repeat_eval_left_assoc_restricted`. This restriction is codified in the `WellFoundedAccess` proposition:

```
Definition WellFoundedAccess out i lo hi s :=
  forall v idx hiz loz,
    eval_Zexpr_Z v lo = Some loz ->
    eval_Zexpr_Z v hi = Some hiz ->
    idx < Z.to_nat (hiz - loz) ->
    OnlyLookBack (v $+ (i, (loz + Z.of_nat idx)%Z)) i out loz s.
```

`OnlyLookBack` is a relation defined on `Sexpr` that ensures that all lookbacks read strictly smaller indices. Its only interesting constructor is the following:

```
| OnlyLookBackGetOutput : forall index leader lookback buf,
    buf = out ->
    (* the leader is the current value of the iteration index *)
    v $? i = Some leader ->
    (* lookback is the index we will read from *)
    eval_Zexprlist v [index] [lookback] ->
    (lookback < leader - loz)%Z ->
    OnlyLookBack v i out loz (Get buf [index])
```

The `leader` is the current value of the iteration index in question. We enforce that `lookback`, which is the actual index used to access the output tensor, is less than the current index value minus its initial value, ensuring that we've already written to the index being accessed.

[1]Note that this is a stronger condition than the general well-formedness condition detailed in Section 3.3. We require lookbacks to only look back on strictly smaller indices because `Gen` writes values in increasing index order. However, it is possible to introduce a reindexer that reverses the order of writes, so that larger indices are written first. This is an area for future research.

Now, it only remains to show that `repeat_eval_left_assoc_restricted` calculates the same result as the loop-based lowering of `Gen i lo hi (Scalar s)`. We first show that if iterating `e` n times using `repeat_eval_left_assoc_restricted` produces result `r`, and if iterating one more time produces result `r'`, then `r` and `r'` will agree on their first n indices. Furthermore, index n (note 0-indexing) of `r'` matches the value produced `c` on its $n+1$ th loop iteration. Therefore, the fixpoint process and `c` produce the same values on each of their respective "iterations." Furthermore, the fixpoint process finishes in exactly the number of iterations as `c` takes, $hi - lo$, since each element of the output tensor is a scalar (size 1). Thus the fixpoint process for evaluating `e` and the lowering of `e` produce the same result, as desired.

Chapter 4

Future Work

4.1 Wide Generation

The current foremost constraint on wide generation is that lanes cannot "communicate" through common variables. Relaxing this restriction would allow for the expression of "summary statistics." For example, in FlashAttention's softmax calculation [6], the running sum of exponentials d is derived from the summary statistic m, the running maximum of numbers being softmax'ed. Currently, there is no way to express the computation of d because the lane calculating d cannot read from the output variable of any other lane, namely, the lane calculating m.

4.2 Sequential Computation

A concrete next step for sequential computation is to extend the correctness proof to the arbitrary-dimensional case. This 1-dimensional case was able to leverage the fact that the fixpoint process and lowered program execute for the same number of iterations, which we can no longer do in the arbitrary-dimensional case. Suppose the tensor being generation has n dimensions $l_1, ..., l_n$ with l_1 being the outermost dimension. `eval_stmt` specifies how a for loop is executed iteration-by-iteration; what happens within each iteration is opaque. In our n-dimensional case, this means that all we know is that an entire $n-1$ -dimensional tensor of size $\prod_{i=2}^{n} l_i$ is written to the heap every iteration of the lowered program. Meanwhile, the fixpoint process iterates normally $\prod_{i=i}^{n} l_i$ times, and all we know is that at least one index becomes correct each iteration. Thus, a potential proof strategy is to match the state of the fixpoint process every $\prod_{i=2}^{n} l_i$ iterations to the state of the lowered program every 1 iteration. Inducting on n would allow us to prove correctness for arbitrary-dimensional tensors. Note that the 1-dimensional proof undertaken in Section 3.4 can be considered the base case of this proof.

Appendix A

Full Semantics of `ATLexpr`

`eval_expr` provides a denotational semantics for the `ATLexpr` type:

```
Inductive eval_expr (sh : context) :
  valuation -> expr_context -> ATLexpr -> result -> Prop :=
  | EvalScatterBase : forall ec v lo hi n loz hiz nz i body rdx b l lz,
    eval_Zexpr_Z v lo = Some loz ->
    eval_Zexpr_Z v hi = Some hiz ->
    (hiz <= loz)%Z ->
    size_of body l ->
    eval_Zexprlist v l lz ->
    eval_Zexpr v n nz ->
    In i (vars_of_Zexpr rdx) ->
    eval_expr sh v ec (Scatter i n lo hi body rdx b)
      (gen_pad (map Z.to_nat (nz::lz)))
  | EvalScatterStepTrue :
    forall ec v lo hi n loz hiz nz i body rdx b l lz r r' r'' vec I,
    eval_Zexpr_Z v lo = Some loz ->
    eval_Zexpr_Z v hi = Some hiz ->
    (loz < hiz)%Z ->
    size_of body l ->
    eval_Zexprlist v l lz ->
    eval_Zexpr v n nz ->
    eval_expr sh v ec (Scatter i n (lo+|1|)%z hi body rdx b) (V vec) ->
    eval_expr sh (v $+(i,loz)) ec (Guard b body) r ->
    eval_Zexpr (v $+ (i,loz)) rdx I ->
    nth_error vec (Z.to_nat I) = Some r' ->
    add_result r r' r'' ->
    ~ i \in dom v ->
            ~ contains_substring "?" i ->
            constant (vars_of_Zexpr rdx) \subseteq constant [i] \cup dom v ->
            In i (vars_of_Zexpr rdx) ->
    eval_expr sh v ec (Scatter i n lo hi body rdx b)
      (V (update vec (Z.to_nat I) r''))
  | EvalGenBase : forall ec v lo hi loz hiz i body,
      eval_Zexpr_Z v lo = Some loz ->
      eval_Zexpr_Z v hi = Some hiz ->
      (hiz <= loz)%Z ->
```

```
    eval_expr sh v ec (Gen i lo hi body) (V [])
| EvalGenStep : forall ec v lo hi loz hiz i body l r,
    eval_Zexpr_Z v lo = Some loz ->
    eval_Zexpr_Z v hi = Some hiz ->
    (loz < hiz)%Z ->
    ~ i \in dom v ->
    ~ contains_substring "?" i ->
    eval_expr sh (v $+ (i,loz)) ec body r ->
    eval_expr sh v ec (Gen i (ZPlus lo (ZLit 1%Z)) hi body) (V l) ->
    eval_expr sh v ec (Gen i lo hi body) (V (r::l))
| EvalSumStep : forall v ec lo hi loz hiz i body r r' s,
    eval_Zexpr_Z v lo = Some loz ->
    eval_Zexpr_Z v hi = Some hiz ->
    (loz < hiz)%Z ->
    ~ i \in dom v ->
    ~ contains_substring "?" i ->
    eval_expr sh (v $+ (i,loz)) ec body r ->
    eval_expr sh v ec (Sum i (ZPlus lo (ZLit 1%Z)) hi body) r' ->
    add_result r r' s ->
    eval_expr sh v ec (Sum i lo hi body) s
| EvalSumBase : forall v ec lo hi loz hiz i body l lz,
    eval_Zexpr_Z v lo = Some loz ->
    eval_Zexpr_Z v hi = Some hiz ->
    (hiz <= loz)%Z ->
    size_of body l ->
    eval_Zexprlist v l lz ->
    eval_expr sh v ec (Sum i lo hi body) (gen_pad (List.map Z.to_nat lz))
| EvalGuardFalse : forall e v ec b l lz,
    eval_Bexpr v b false ->
    size_of e l ->
    eval_Zexprlist v l lz ->
    eval_expr sh v ec (Guard b e) (gen_pad (List.map Z.to_nat lz))
| EvalGuardTrue : forall e ec v b r,
    eval_Bexpr v b true ->
    eval_expr sh v ec e r ->
    eval_expr sh v ec (Guard b e) r
| EvalLbindS : forall v e1 e2 x r1 l2 ec,
    size_of e1 [] ->
    ec $? x = None ->
    ~ x \in vars_of e1 /\ ~ x \in vars_of e2 ->
    vars_of e1 \cap vars_of e2 = constant nil ->
    eval_expr sh v ec e1 (S r1) ->
    eval_expr (sh $+ (x,[])) v (ec $+ (x,S r1)) e2 l2 ->
    eval_expr sh v ec (Lbind x e1 e2) l2
| EvalLbindV : forall v e1 e2 x l1 l2 ec esh1 esh1z,
    esh1 <> [] ->
    size_of e1 esh1 ->
    eval_Zexprlist v esh1 esh1z ->
```

```
      ec $? x = None ->
      ~ x \in vars_of e1 /\ ~ x \in vars_of e2 ->
      vars_of e1 \cap vars_of e2 = constant nil ->
      eval_expr sh v ec e1 (V l1) ->
      eval_expr (sh $+ (x, esh1)) v
                (ec $+ (x,V l1)) e2 l2 ->
      eval_expr sh v ec (Lbind x e1 e2) l2
  | EvalConcat : forall ec v e1 e2 l1 l2,
      eval_expr sh v ec e1 (V l1) ->
      eval_expr sh v ec e2 (V l2) ->
      eval_expr sh v ec (Concat e1 e2) (V (l1++l2))
  | EvalTranspose : forall e v ec l n nz m mz esh eshz,
      eval_expr sh v ec e (V l) ->
      size_of e (n::m::esh) ->
      eval_Zexprlist v (n::m::esh) (nz::mz::eshz) ->
      eval_expr sh v ec (Transpose e)
                (transpose_result l (map Z.to_nat (mz::nz::eshz)))
  | EvalFlatten : forall e v ec l,
      eval_expr sh v ec e (V l) ->
      Forall (fun x => exists v, x = V v) l ->
      eval_expr sh v ec (Flatten e) (V (flatten_result l))
  | EvalSplit : forall e v ec l k,
      eval_expr sh v ec e (V l) ->
      Forall (fun x => exists v, x = V v) l ->
      eval_expr sh v ec (Split k e) (V (split_result (Z.to_nat
(eval_Zexpr_Z_total $0 k)) l))
  | EvalTruncr : forall e v ec k kz l,
      eval_Zexpr_Z v k = Some kz ->
      (0 <= kz)%Z ->
      eval_expr sh v ec e (V l) ->
      eval_expr sh v ec (Truncr k e)
                (V (List.rev (truncl_list (Z.to_nat kz) (List.rev l))))
  | EvalTruncl : forall e v ec k kz l,
      eval_Zexpr_Z v k = Some kz ->
      (0 <= kz)%Z ->
      eval_expr sh v ec e (V l) ->
      eval_expr sh v ec (Truncl k e) (V (truncl_list (Z.to_nat kz) l))
  | EvalPadr : forall e v ec l s n k kz sz,
      eval_Zexpr_Z v k = Some kz ->
      (0 <= kz)%Z ->
      size_of e (n::s) ->
      eval_expr sh v ec e (V l) ->
      eval_Zexprlist v s sz ->
      eval_expr sh v ec (Padr k e)
                (V (l++gen_pad_list ((Z.to_nat kz)::(List.map Z.to_nat sz))))
  | EvalPadl : forall e v ec l s n k kz sz,
      eval_Zexpr_Z v k = Some kz ->
      (0 <= kz)%Z ->
```

```
    size_of e (n::s) ->
    eval_expr sh v ec e (V l) ->
    eval_Zexprlist v s sz ->
    eval_expr sh v ec (Padl k e)
      (V (gen_pad_list ((Z.to_nat kz)::(List.map Z.to_nat sz))++l))
| EvalShear : forall e n m s v ec l sz nz mz,
    size_of e (n::m::s) ->
    eval_expr sh v ec e (V l) ->
    eval_Zexprlist v s sz ->
    eval_Zexpr_Z v n = Some nz ->
    eval_Zexpr_Z v m = Some mz ->
    eval_expr sh v ec (Shear e)
      (V (shear_result l (Z.to_nat mz) (map Z.to_nat sz)))
| EvalUnshear : forall e n m s v ec l sz az bz a b,
    size_of e (n::m::s) ->
    eval_expr sh v ec e (V l) ->
    eval_Zexprlist v s sz ->
    eval_Zexpr_Z v a = Some az ->
    eval_Zexpr_Z v b = Some bz ->
    eval_expr sh v ec (Unshear a b e)
      (V (unshear_result l az bz (map Z.to_nat sz)))
| EvalScalar : forall s v ec r,
    eval_Sexpr sh v ec s r ->
    eval_expr sh v ec (Scalar s) (S r).
```

Appendix B

Full Semantics of `stmt`

`eval_stmt` provides an operational semantics for the `stmt` type:

```
Inductive eval_stmt (v : valuation) :
  stack -> heap -> stmt -> stack -> heap -> Prop :=
| EvalAssignS :
    forall x st h rhs r cont a,
      eval_Sstmt v st h rhs r ->
      st $? x = Some a ->
      cont = [] ->
      eval_stmt v st h (Store Assign x cont rhs) (st $+ (x,r)) h
| EvalAssignV :
    forall x st h rhs r cont arr z a,
      eval_Sstmt v st h rhs r ->
      cont <> [] ->
      h $? x = Some arr ->
      eval_Zexpr_Z v (flatten_shape_index
                              (map snd cont)
                              (map fst cont)) = Some z ->
      arr $? z = Some a ->
      eval_stmt v st h (Store Assign x cont rhs) st (h $+ (x, arr $+ (z,r)))
| EvalReduceS :
    forall x st h rhs r old cont,
      eval_Sstmt v st h rhs r ->
      st $? x = Some old ->
      cont = [] ->
      eval_stmt v st h (Store Reduce x cont rhs) (st $+ (x,r+old)%R) h
| EvalReduceV :
    forall x st h rhs r cont arr z old,
      eval_Sstmt v st h rhs r ->
      cont <> [] ->
      h $? x = Some arr ->
      eval_Zexpr_Z v (flatten_shape_index
                              (map snd cont)
                              (map fst cont)) = Some z ->
      arr $? z = Some old ->
      eval_stmt v st h
                (Store Reduce x cont rhs) st (h $+ (x, arr $+ (z,r+old)%R))
```

```
| EvalIfTrue : forall b s st h st' h',
    eval_stmt v st h s st' h' ->
    eval_Bexpr v b true ->
    eval_stmt v st h (If b s) st' h'
| EvalIfFalse : forall b s st h,
    eval_Bexpr v b false ->
    eval_stmt v st h (If b s) st h
| EvalAllocS : forall x st h,
    eval_stmt v st h (AllocS x) (st $+ (x,0%R)) h
| EvalAllocV : forall x n st h nz,
    eval_Zexpr v n nz ->
    eval_stmt v st h (AllocV x n) st
              (alloc_array_in_heap [Z.to_nat nz] h x)
| EvalFree : forall x st h,
    eval_stmt v st h (Free x) st (h $- x)
| EvalDeallocS : forall x st h,
    eval_stmt v st h (DeallocS x) (st $- x) h
| EvalForStep : forall i st h lo hi body st' h' loz hiz st'' h'',
    eval_Zexpr_Z v lo = Some loz ->
    eval_Zexpr_Z v hi = Some hiz ->
    (loz < hiz)%Z ->
    eval_stmt (v $+ (i,loz)) st h body st' h' ->
    eval_stmt v st' h'
              (For i (ZPlus lo (ZLit 1%Z)) hi body)
              st'' h'' ->
    eval_stmt v st h (For i lo hi body) st'' h''
| EvalForBase : forall i st h lo hi body loz hiz,
    eval_Zexpr_Z v lo = Some loz ->
    eval_Zexpr_Z v hi = Some hiz ->
    (hiz <= loz)%Z ->
    eval_stmt v st h (For i lo hi body) st h
| EvalSeq : forall st h s1 s2 st' h' st'' h'',
    eval_stmt v st h s1 st' h' ->
    eval_stmt v st' h' s2 st'' h'' ->
    eval_stmt v st h (Seq s1 s2) st'' h''.
```

Appendix C

ATL Correctness Statement

Below is the full statement of correctness of `lower`:

```
Theorem lower_correct_weak :
  forall e,
    constant_nonneg_bounds e ->
    forall sh v ec r,
      (* functional evaluation of ATL *)
      eval_expr sh v ec e r ->
      forall l, size_of e l ->
      forall p st h st' h' reindexer asn,
        (* imperative evaluation of lowering *)
        eval_stmt v st h (lower e reindexer p asn sh) st' h' ->
        (* our environment is well-formed *)
        well_formed_environment st h p sh v (vars_of e) ec ->
        (* reindexer is well-formed *)
        partial_well_formed_reindexer reindexer v r ->
        (* allocation is well-formed *)
        partial_well_formed_allocation reindexer r st h p v ->
        (* expr context and imperative state agree *)
        contexts_agree ec st h sh ->
        forall pads g,
          pad_set v g e pads ->
        (forall pads (x : var) (r0 : result),
            g $? x = Some pads ->
            ec $? x = Some r0 ->
            relate_pads pads r0) ->
        assign_no_overwrite st h p reindexer r v asn ->
        match reindexer (shape_to_index
                           (result_shape_Z r)
                           (shape_to_vars (result_shape_Z r))) with
        | [] => h' = h
           /\ st' = st $+ (p, match st $? p with
                              | Some r => r
                              |_ => 0%R
                              end + match r with
                                    | S (SS s) => s
                                    | _ => 0%R
```

```
                                end)%R
  | _ =>
      (h' = h $+ (p,
                  array_add
                    (h $! p)
                    (partial_result_to_array_delta
                       (partial_interpret_reindexer reindexer
                                                    (result_shape_Z r) v) r))
       /\ st' = st)
  end.
```

Appendix D

Source Code

All work was performed in the repository https://github.mit.edu/lamanda/atl. The proof of correctness of `WBind`'s lowering can be found on the branch `fprasx/lower-widegen-correct`. The 1-dimensional proof of correctness of `GenRec`'s lowering can be found on the branch `fprasx/lower-genrec-1d`. The main correctness proof is in `LowerCorrect.v`.

Bibliography

[1] A. Liu, G. L. Bernstein, A. Chlipala, and J. Ragan-Kelley, "Verified tensor-program optimization via high-level scheduling rewrites," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, Jan. 2022, doi: 10.1145/3498717.

[2] A. Liu, G. Bernstein, A. Chlipala, and J. Ragan-Kelley, "A Verified Compiler for a Functional Tensor Language," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024, doi: 10.1145/3656390.

[3] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "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*, in PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. doi: 10.1145/2491956.2462176.

[4] T. Chen *et al.*, "TVM: an automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, in OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594.

[5] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "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*, in ICFP 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 205–217. doi: 10.1145/2784731.2784754.

[6] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.