End-to-End Translation Validation for the Halide Language

Basile Clément, Albert Cohen

End-to-End Translation Validation for the Halide Language

BASILE CLÉMENT, Inria, France and École Normale Supérieure, France
ALBERT COHEN, Google, France

This paper considers the correctness of domain-specific compilers for tensor programming languages through the study of Halide, a popular representative. It describes a translation validation algorithm for *affine* Halide specifications, independently of the scheduling language. The algorithm relies on "prophetic" annotations added by the compiler to the generated array assignments. The annotations provide a refinement mapping from assignments in the generated code to the tensor definitions from the specification. Our implementation leverages an affine solver and a general SMT solver, and scales to complete Halide benchmarks.

CCS Concepts: • **Theory of computation → Program verification**; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: Formal semantics, Domain-specific Languages, Tensor Compilers, Polyhedral Compilers, Validation

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

1 INTRODUCTION

Domain-Specific Languages (DSLs) for scientific computing, signal processing and machine learning allow the expression of high-level specifications of matrix and tensor computations and their compilation into efficient low-level code [Baumgartner et al. 2005; Ragan-Kelley et al. 2013; Vasilache et al. 2020]. The most popular representative is Halide[1], specialized on image processing and computer vision [Ragan-Kelley et al. 2012, 2013], with a wide range of uses from high-end graphics to on-device processing in cameras.

Optimizing compilation for these DSLs involves finding a good *schedule* describing the orchestration of elementwise computational, communication and storage instructions composing the tensor algebraic operations of interest. This schedule involves high-impact decisions about vectorization, thread-level parallelism, distribution, locality optimization, memory mappings and layouts, etc. Halide and TVM (an integrated tensor algebra compiler and machine learning framework, initially derived from Halide [Moreau et al. 2019]) typically involve a performance expert to program such a schedule manually, although machine-learning-based auto-tuners exist to partly automate the task [Moreau et al. 2019; Mullapudi et al. 2016]. Other approaches rely on optimization research algorithms such as integer linear programming, including Polymage [Mullapudi et al. 2015] and Tensor Comprehensions [Vasilache et al. 2018, 2020]. The schedule plays a central role in guiding efficient code generation from a high-level equational specification; it is a fundamental, unifying property of all these approaches.

[1] https://github.com/halide/Halide

Authors' addresses: Basile Clément, basile.clement@inria.fr, Inria, Paris, France and École Normale Supérieure, Paris, France; Albert Cohen, albert.cohen@google.com, Google, Paris, France.

2475-1421/2022/4-ART84
https://doi.org/10.1145/3527328

The above-mentioned frameworks however differ in their schedule representations: Tensor Comprehensions uses affine schedule trees [Grosser et al. 2015] while Halide and TVM uses custom scheduling directives with different trade-offs between expressiveness, expert control and automation. Whether they arise from DSLs like Halide or from black-box optimizers of imperative loop nests like Pluto [Bondhugula et al. 2008a,b], the transformations involving array indices and loop iterations have a strong impact on the structure of the generated code. Effectively, the efficient code generated by these compilers often bears little resemblance to the original specification. The control structure is altered at both local and global scales, by the application of many loop transformations such as loop tiling, fusion, unrolling, index set splitting, pipelining. In addition, functional tensors in the specification are lowered into imperative arrays in the generated code with possibly different dimensionalities, some of the original dimensions are coalesced to express parallelism opportunities, other dimensions are further decomposed (reshaped) to optimize locality, values can be promoted to registers, rematerialized (i.e. computed multiple times) or even completely disappear through algebraic simplifications or partial evaluation.

This paper aims at verifying that low-level generated code matches the high-level specification of tensor computations by exhibiting a refinement mapping [Abadi and Lamport 1988]. Focusing on the Halide language and compiler, we show that its architecture lends itself well to *translation validation* [Pnueli et al. 1998] despite the impressive semantic gap affecting both control and memory. We achieve this by formalizing both Halide specifications (called algorithms) and their low-level imperative representation. We then associate an equation in the algorithm to each assignment in the imperative code. This approach does not require formalizing the Halide scheduling language: while maybe counter-intuitive, this indicates the techniques in this paper should generalize beyond Halide itself to other compilers relying on affine transformations of control flow and storage mapping, combined with symbolic reasoning on value expressions — provided the compiler can produce the appropriate annotations.

The outline of the paper is as follows. We first present an intuitive explanation of *prophetic expressions*, annotations we use to allow symbolic evaluation of unbounded sequential loops, in Section 2. We then briefly recall the specifics of affine Halide specifications in Section 3, for which we formalize in Section 4 a translation-validation procedure using prophetic expressions. To our knowledge, our approach is the first to allow the verification of both general affine loop transformations and algebraic transformations. We then evaluate our approach on some reference Halide benchmarks in Section 5, and compare it to the state-of-the-art of affine program equivalence checking. We discuss related work in Section 6, and the limitations and possible extensions of our approach in Section 7. In particular, we mention two possible threats to the correctness of our validator, namely overflow checking and the "fast-math" optimizations performed by Halide on floating point numbers.

2 MOTIVATING EXAMPLES

Picture yourself a specification language for n-dimensional arrays. The exact details of the specification language may vary, but can be roughly thought of as follows. This language deals with tensors: functional values indexed by integer points in a multidimensional space. The tensors are defined by equations, read as assignments, and implicitly quantified over the multidimensional domain. For instance, the outer product of two vectors A and B is a two-dimensional tensor defined by the equation:

$$C(i, j) = A(i) \times B(j) \tag{1}$$

Here i and j are implicitly bound variables ranging over arbitrary integer values. These high-level specifications are compiled into imperative programs through a code generation process guided by

a (user-provided or compiler-generated) schedule, whose concrete details vary. In the generated program, the tensors are replaced by arrays representing finite subdomains of the corresponding tensor. Our first intuition is that the nature of the code generator makes it possible to keep track of a mapping between array writes and tensor definitions: this property will be at the basis of our verification algorithm.

Going back to Eq. (1), in an implementation of that specification, A, B and C become arrays, and the range of i and j are computed through auxiliary mechanisms such as inference from the shape of the input arrays (as in Tensor Comprehensions), or an explicit realization domain for the output array (as in Halide). In Halide, expert users can manually write schedules using a domain-specific language; in our example, the schedule `C.split(i, i0, i1, 4).reorder(i1, j, i0)` would generate the following implementation, expressed in pseudocode (in this paper, we will use lower-case names and brackets for array accesses, while upper-case names and parentheses are used for tensor accesses).

```
1  for i0 = 0 to (N + 3) / 4 - 1 do
2    for j = 0 to M - 1 do
3      for i1 = 0 to 3 do
4        let i = min(i0 * 4, N - 4) + i1 in
5        c[i, j] := b[j] * a[i]
```

The `split` directive indicates that the loop over `i` should be separated into an outer loop `i0` and an inner loop `i1`. The loop nesting order, from innermost to outermost, is specified by the `reorder` directive. More examples of the Halide scheduling language are shown by Ragan-Kelley et al. [2017]. The commutativity of $\times$ (i.e. `b[j] * a[i]` instead of `a[i] * b[j]`) has been applied manually for illustration purposes: it does not appear in the schedule. More generally, Halide features a simplification algorithm which can propagate constants and make use of algebraic properties such as commutativity, associativity and distributivity: the values stored in the array `c` can be computed using a different, but equivalent[2], expression than that defining tensor C.

The parameters N and M used in the code correspond to the size of the arrays `a` and `b`, and are not statically known; rather, the program ought to be valid for any such sizes.

In this case, it is relatively easy to convince oneself that the implementation follows the semantics prescribed by the specification. More precisely, for any tensors A, B and C such that $C(i, j) = A(i) \times B(j)$, and any initial memory mapping the `a[i]` to $A(i)$ and the `b[j]` to $B(j)$, running the above program will produce a final memory which, in addition, maps the `c[i, j]` to $C(i, j)$, where i ranges over $0, \ldots, N-1$ and j over $0, \ldots, M-1$.

There are multiple proofs of this fact, but we will focus on the following one, which exploits the remark that, when executing an assignment `c[i, j] := b[j] * a[i]`, the value written is always the corresponding $C(i, j)$ from the specification. More precisely, we can make the following observations:

- The set of locations written by the assignment `c[i, j] := b[j] * a[i]` is precisely

$$\{\mathtt{c}[\min(4i_0, N-4) + i_1, j] \mid 0 \le i_0 < \lceil N/4 \rceil \wedge 0 \le i_1 < 4 \wedge 0 \le j < M\}$$

  This set contains the "required" set of locations $\{\mathtt{c}[i, j] \mid 0 \le i < N \wedge 0 \le j < M\}$, which is expressed as an inclusion of sets defined by quasi-affine formulas. Quasi-affine formulas are affine formulas extended with division and modulo operations where the denominator is a constant. The decidability properties of Presburger arithmetic [Presburger 1929] extend to quasi-affine formulas, and efficient specialized solvers such as `isl` [Verdoolaege 2010] can be used to check its validity.

[2]Halide makes fast-math assumptions on floating-point numbers, which we discuss in Section 7.3.

- When the assignment is executed for some values of i and j, we have $0 \leq i < N$ and $0 \leq j < M$, which can also be checked using `isl`, and ensures that reads from arrays `a` and `b` are in bounds. Note that this is not the same check as above: here we bound the set of locations *upwards* to stay within the bounds for `a` and `b`, while earlier we bounded that set *downwards* to at least contain the required writes to `c`. Combining these two checks, the set of computed locations must be exactly $\{\texttt{c}[i, j] \mid 0 \leq i < N \wedge 0 \leq j < M\}$.
- When the assignment is executed for some values of i and j, the value in `b[j]` is $B(j)$ and the value in `a[i]` is $A(i)$, as per the previous point, and because arrays `a` and `b` are never written to. Hence, the value written to `c[i, j]` is equal to $B(j) \times A(i)$. This value is equal to $C(i, j)$, since $C(i, j) = A(i) \times B(j)$ by definition and assuming $\times$ is commutative. In general, proving this type of equality requires unfolding tensor definitions and checking the algebraic reasoning performed by the compiler's simplifier. Off-the-shelf general purpose SMT solvers such as Z3 [de Moura and Bjørner 2008] used in this work, are quite good at proving such formulas.

These three checks — coverage of writes, definedness of reads, equality of values — form the backbone of a verifier for array programs. Here, we have seen a simple example where a single value is ever written to each location; in general, the resulting program can contain recurrences: an iteration of a loop which depend on values read by a previous iteration of the same loop. This would make the proof fail: we have assumed that, when reading from arrays `a` and `b`, we know exactly the value they hold. In the presence of recurrences, this is in general impossible, as it requires unfolding many iterations of the loop at once. This property is also the main difficulty for program equivalence checking approaches, where best-effort techniques based on transitive closure [Shashidhar et al. 2005] or affine hulls [Verdoolaege et al. 2012] have been developed.

Instead, we can side-step the issue entirely by relying on the fact we are dealing with implementations generated by a scheduling compiler, which has a rich set of information about the assignments available. Let us assume that the assignment is annotated with a *prophetic expression*. The prophetic expression lives in the specification world, and predicts the value that will be written by the assignment in terms of tensors. Assuming that the compiler can produce those annotations along with the code, this allows breaking the cycle, because the prophetic expression uses tensors, and hence is independent of the program memory. In particular, this means that we can always know the value of the prophetic expression for any iteration of a loop without having to execute the previous iterations of the loop.

Let us examine an example of this by considering matrix multiplication, implemented with an explicit accumulator R:

$$R(i, j, k) = \texttt{if } k \leq 0 \texttt{ then } 0 \texttt{ else } R(i, j, k-1) + A(i, k) \times B(k, j)$$
$$C(i, j) = R(i, j, P-1)$$

Reusing the same schedule as earlier, we would get the following implementation, where we have annotated each assignment with a prophetic expression:

```
1  for i0 = 0 to (N + 3) / 4) - 1 do
2    for j = 0 to M - 1 do
3      for i1 = 0 to 3 do
4        let i = min(i0 * 4, N - 4) + i1 in
5        r {0} := 0
6        for k = 0 to P - 1 do
7          r {R(i,j,k)} := r + b[k, j] * a[i, k]
8        c[i, j] {C(i,j)} := r
```

In an assignment such as `c[i, j] {`$C(i,j)$`} := r`, the prophetic expression $C(i,j)$ between brackets is ignored during the execution of the program, and only used for the validation. It is an assertion than the value written when executing the statement will be $C(i,j)$.

Without the annotations, it would not be immediately clear what the value of `r` should be at iteration k of the loop. However, by using the annotations, we can build a *prophetic* version of the program, which reads from the specification and executes assignments by using their prophetic expression instead of the right-hand side.

This prophetic version of the program never reads from mutable memory, making its analysis much easier: knowing the last write to an array cell is enough to know its value. In particular, it is clear from the prophetic version of the program that `r` is equal to either 0 (when $k = 0$) or $R(i,j,k-1)$ (otherwise) when updating `r` within the loop on k, and equal to $R(i,j,P-1)$ (assuming $P > 0$) when writing to `c[i, j]`. In addition, the fact that $C(i,j)$ is written by the prophetic program to the cell `c[i, j]` for all $0 \leq i < N$ and $0 \leq j < M$ is also clear, by the same set inclusion as in the outer product case.

Finally, we have to prove that the regular version of the program has the same behavior as the prophetic version of the program. To do so, we have to prove the following side conditions for any values of i, j and k reachable during execution of the program:

$$R(i,j,k) = \texttt{if } k \leq 0 \texttt{ then } 0 \texttt{ else } R(i,j,k-1) + B(k,j) \times A(i,k)$$
$$C(i,j) = \texttt{if } P \leq 0 \texttt{ then } 0 \texttt{ else } R(i,j,P-1)$$

The right-hand side of the equalities are computed from the right-hand side of the assignments of `r` and `c[i, j]`, where the reads to `r` are computed using the prophetic expressions. The side conditions are an inductive invariant: they ensure that our reasoning propagates from one iteration of the loop to the next.

We now have reduced the correctness of our program to these two quantified equalities in the specification. In and of itself, that is valuable, thanks to the simplifications performed above: in the formulas, the structure of the implementation has been erased, yielding a simpler domain. The equalities in this case can be proven easily by an SMT solver.

To recapitulate, our approach requires two key ingredients to be automated:

- The assignments must be annotated with prophetic expressions in the specification, using tensors and independent of the program memory. This allows us to symbolically evaluate loops of parametric size.
- The expressions used in array indices (both for reads and writes), loop bounds, and conditionals must be quasi-affine. This ensures that we can keep track of the values written to and read from arrays.

The rest of this paper will show that this automation is indeed possible.

3 AFFINE SPECIFICATIONS OF ARRAY PROGRAMS

In order to represent and verify the high-level specification independently of the details of the Halide specification language, we internally convert the Halide algorithm into a System of Affine Recurrence Equations (SARE) [Karp et al. 1967; Le Verge et al. 1991]. In this section, we propose a simple but novel formalization of Halide algorithms, and its reduction to SAREs following Feautrier [1991].

3.1 Notations

We assume disjoint countably infinite sets of array names (ranged over by a), variables (ranged over by x, y), and function names (ranged over by f). We also assume a set of values $\mathcal{U}$, and a

set of types (ranged over by τ). Each type τ can be interpreted by a set of values $[\![\tau]\!] \subseteq \mathcal{U}$. In general, the $[\![\cdot]\!]$ notation is used to denote an interpretation or evaluation. We assume the existence of two distinguished types $\mathbb{A}$, which evaluates to the set of all integers, and $\mathbb{B}$, which evaluates to $\{\texttt{true}, \texttt{false}\}$. Each function name f has a function type $\tau_f = \tau_1 \times \cdots \times \tau_{a_f} \to \tau$, where a_f is the arity of f, and a semantic interpretation $[\![f]\!]$ as a well-typed a_f-ary function of the values. Constants are nullary functions.

A signature $\mathcal{S}$ is a pair $\langle \mathcal{P}, \mathcal{A} \rangle$ where $\mathcal{P}$ is a finite set of integer variables called the *program parameters*, and $\mathcal{A}$ is a finite set of *tensor names* ranged over by A. A tensor A is equipped with a type τ_A and an arity n_A. A model M over $\mathcal{S}$ is an assignment of a value $[\![x]\!]_M$ for each parameter $x \in \mathcal{P}$, and of a function $[\![A]\!]_M$ from $\mathbb{Z}^{n_A}$ to $[\![\tau_A]\!]$ for each $A \in \mathcal{A}$.

3.2 Affine Halide Specifications

In Halide, specifications are called *algorithms*. We propose a simple formalization of Halide algorithms as a system of equations augmented with imperative loops. We first recall the use of Halide algorithms through an example; the interested reader can refer to Ragan-Kelley et al. [2017] for a more thorough description by Halide's designers.

Halide algorithms are defined in a DSL embedded in C++ or Python, where the user defines tensors (called `Func`s) as symbolic multidimensional functions over an infinite domain. Definitions are written using an overloaded `=` operator. For instance, an algorithm computing the general matrix multiplication $D = \alpha AB + \beta C$ can be written as follows, where K is the size of the reduced dimension:

```
Var i, j;
Func D;
RDom k(0, K);
D(i, j) = beta * C(i, j);
D(i, j) = D(i, j) + alpha * A(i, k) * B(k, j);
```

In this algorithm, we define a tensor `D` and give it two definitions. The first definition of any `Func` is its (unique) *pure definition*, and it is treated specially to ensure it defines all the points in the domain. The pure definition make uses of *pure variables*, which have type `Var` in the DSL, and are implicitly quantified over the integers. Distinct pure variables must appear as arguments of the tensor in the left-hand side of a pure definition, and are bound on the right-hand side. Only bound pure variables can appear on the right-hand side of a definition. In the example, the pure definition of tensor `D` is `D(i, j) = beta * C(i, j)`.

Subsequent definitions are *update definitions*, and can refer to the previous values of the tensor in an imperative way. In addition to pure variables, update definitions can make use of *recurrence variables*[3]: variables of type `RVar` which encode imperative loops. Recurrence variables allow defining tensors inductively in multiple steps, by having the value at each step depend on the value at the previous step. Every update definition is implicitly nested in a sequential loop over the `RVar`s appearing in either the left- or right- hand sides of the definition. Recurrence variables are declared using the `RDom` constructor, representing a multidimensional loop as a tuple of `RVar`s. In the example, the definition `D(i, j) = D(i, j) + alpha * A(i, k) * B(k, j)` uses the recurrence variable `k`, and is semantically equivalent to the imperative loop:

```
for k = 0 to K - 1 do
  D(i, j) = D(i, j) + alpha * A(i, k) * B(k, j)
```

[3]In Halide, they are called "reduction variables", but their use is not limited to reductions, hence we prefer the term recurrence variables.

Update definitions can refer to the tensor currently being defined, in an imperative way. They are not equations: pure variables appearing in update definitions should be thought of informally as parallel loops over an infinite domain. It is not possible to give a semantics to some update definitions such as `D(i, j) = D(j, i) + 1`. In order to rule out such impossible definitions, Halide ensures that within a single update definition, the same tensor indices can only be accessed at the *same values of the pure variables*. This is enforced by ensuring that each pure variable appearing in the definition must appear as an argument at the same position in all accesses to that tensor in the definition. The pure variable can also appear at other positions; for instance, it is valid to write `D(i, i + 1) = D(i, i - 1) + D(i, 0)` because `i` appears as an argument in the first position for each of the accesses.

In addition, definitions of multiple functions cannot be interleaved, and recurrence domains can include an arbitrary boolean expression as a filter restricting the points where the update is performed. Halide specifications are laid out textually; once a tensor is used within the definition of another, further update definitions of the tensor are disallowed.

We can represent formally a Halide specification over a signature $\mathcal{S}$ as a tuple $\langle I, P, U, < \rangle$ where:

- I is the set of input tensors, which have no associated definition;
- P maps each tensor $A \in \mathcal{S} \backslash I$ to its *pure definition* P_A;
- U maps each tensor $A \in \mathcal{S} \backslash I$ to a (possibly empty) finite sequence $U_A^1, \ldots, U_A^{n_A}$ of *update definitions* for A;
- $<$ defines a total order over the non-input tensors in $\mathcal{S} \backslash I$, representing the textual order of the first pure definition to the tensor

The pure definition P_A for tensor $A \in \mathcal{S}$ is an equation written as follows:

$$\forall x_1, \ldots, x_{n_A},\ A(x_1, \ldots, x_{n_A}) = e$$

The right-hand side of a pure definition can only refer to input tensors and the defined tensors A' defined before A (i.e. with $A' < A$). In particular, the definition of A cannot refer to A itself.

An update definition U_A^i for tensor $A \in \mathcal{S}$ is also an equation, which involve some pure variables $x_1, \ldots, x_n$ and an arbitrary number of recurrence variables $y_1, \ldots, y_r$:

$$\forall x_1, \ldots, x_n.\ \texttt{for}\ y_1 : R_1, \ldots, y_r : R_r.\ \phi \implies A(e_1, \ldots, e_{n_A}) = e$$

Each of the y_i has a recurrence domain R_i, a parametrically bounded interval of $\mathbb{Z}$. The bounds of R_i can only depend on the parameters, not the pure variables nor other recurrence variables. The boolean expression ϕ is the filter: an additional condition on the points where the update is performed.

The well-formedness condition can be formalized as follows. We require the existence of a function π from the pure variables $\{1, \ldots, n\}$ to the argument positions $\{1, \ldots, n_A\}$ such that, for all $1 \leq j \leq n$:

(1) $e_{\pi(j)} = x_j$, and
(2) For each $A(e'_1, \ldots, e'_{n_A})$ appearing as a sub-term of either the right-hand side e or the filter ϕ, we have $e'_{\pi(j)} = x_j$.

π maps each pure variable to a shared position in all accesses of the tensor being defined in the update definition. In general, there might be less pure variables than argument positions ($n \leq n_A$); the other arguments can be any expression of the pure and reduction variables. In any case, this ensures that there is no circular dependencies between iterations of an update definition: the expression defining a location at some value of the pure variables can not read from locations defined for other values of the pure variables, since the pure variables are shared indices of all the accesses.

If all the filters and all the expressions occurring as index to any tensor in an algorithm are piecewise quasi-affine combinations of pure and recurrence variables, we say that the algorithm is an *affine algorithm*. In this paper, we only consider affine algorithms.

3.3 Systems of Affine Recurrence Equations

SAREs are a specification language for affine programs used in polyhedral compilers [Karp et al. 1967; Le Verge et al. 1991]. Our tool internally converts Halide algorithms to SAREs, and the formal proofs in Section 4 assume a SARE as a specification.

A SARE over a signature $\mathcal{S}$ is simply a set of equations:

$$\forall x_1, \ldots, x_{n_A},\ \phi \implies A(x_1, \ldots, x_{n_A}) = e$$

where ϕ is a Boolean-valued affine formula of $x_1, \ldots, x_{n_A}$ and the program parameters, and e is an expression built from functions and tensor accesses. The "recurrence" part in the name refers to the fact that an equation defining tensor A can itself read from tensor A. The indices of all tensor accesses in e must be affine expressions of $x_1, \ldots, x_{n_A}$ and the parameters. In addition, if two equations define the same tensor A in their left-hand side, then the domains of the two equations must be disjoint (that is, the corresponding ϕs are mutually exclusive). This allows viewing all the equations defining A as a single equation for A defined by case analysis.

The inputs of a SARE are the tensors which never appear as the left-hand side of an equation. A model M satisfies a SARE if all the equations of the SARE hold in M. Note that we do not require completeness of the SARE equations, nor do we preclude self-references. For instance, the SARE $A(0) = A(1)$ is satisfied by all models M with $[\![A]\!]_M(0) = [\![A]\!]_M(1)$, while the SARE $A(0) = A(0) + 1$ is not satisfied by any model.

3.4 Reduction from Affine Halide Algorithms to SAREs

Let us now explain the reduction from affine Halide algorithms to SAREs. For each tensor A in the Halide algorithm, we will introduce a tensor A^S to represent A in the SARE, as well as intermediate tensors A_0 to represent the pure definition and $A_1, \ldots, A_n$ to represent the update definitions of A. The tensor A_0 is directly defined using the pure definition of A, where each other tensor B is replaced with its SARE equivalent B^S. The tensor A_i representing an update definition U_A^i has r extra indices, where r is the number of recurrence variables in U_A^i: conceptually, the tensor is replicated for each value of the recurrence variables. Note that due to the update restriction, knowing the value of the array indices is enough to know the value of all pure variables appearing in the right-hand side.

Instead of formal minutiae, we explain the construction of the tensor A_i through examples. In the matrix multiplication algorithm above, the tensor D_1 has an extra index for dimension k. Within the reduction domain $0 \leq k < K$, we replace accesses to other tensors with their SARE equivalent, and replace accesses to D to accesses to D_i with the previous value of k. Outside the reduction domain, we replicate the last value of the previous stage, which is just $D_0(i, j)$ in this case:

$$\begin{aligned} 0 \leq k < K &\Rightarrow D_1(i, j, k) = D_1(i, j, k-1) + A^S(i, k) \times B^S(k, j) \\ k < 0 \vee k \geq K &\Rightarrow D_1(i, j, k) = D_0(i, j) \end{aligned}$$

The SARE tensor for D, D^S, is defined as the last value of D_1 lexicographically, i.e. $D^S(i, j) = D_1(i, j, K-1)$. Note that when $K \leq 0$, $D_1(i, j, K-1)$ is equal to $D_0(i, j)$.

There are a few subtleties here. First, if there are multiple reduction variables, the previous value must be computed lexicographically within the definition rectangle: for two reduction variables

$0 \leq x < X$ and $0 \leq y < Y$, the extra indices to a recursive access would be $\texttt{select}(y \leq 0, x-1, x)$ and $\texttt{select}(y \leq 0, Y-1, y-1)$ respectively. Second, if there is a filter, when the filter is false, the previous value of the current stage is directly reused. Finally, if the indices depend on the reduction variables, all non-updated locations are defined using the previous value of the current stage. For instance, the update `D(i, 2k) += D(i, k)` where i is a pure variable and $0 \leq k < K$ is a reduction variable becomes (within the recurrence domain):

$$0 \leq k < K \Rightarrow D_1(i, 2k, k) = D_1(i, 2k, k-1) + D_1(i, k, k-1)$$
$$0 \leq k < K \wedge j \neq 2k \Rightarrow D_1(i, j, k) = D_1(i, j, k-1)$$

In this construction, we never used the assumption that the Halide algorithm was affine; in fact, this construction can be used to give a formal semantics to any Halide algorithm as a system of equations. If the original Halide algorithm was affine, all the expressions introduced by the transformation are piecewise quasi-affine, and the resulting system of equations is a SARE. We note that the construction described above introduces unneeded copies, i.e. equations which are just defined in terms of another tensor. In general, it is not possible to eliminate such copies; since we are in the affine case, we use `isl` to symbolically solve for the last non-copy equation using a similar approach as that of Feautrier [1991].

4 TRANSLATION-VALIDATION OF AFFINE ARRAY PROGRAM

Let us now formalize the ideas presented in Section 2. We present Sched, an annotated imperative language, and a validator for Sched programs with respect to a SARE. Sched annotations indicate, for each imperative write, the equation it implements in the specification. We expect the compiler to be able to elaborate the annotations during the code generation process. In the case of Halide, we instrumented the Halide compiler to do so, as described in Section 5.

4.1 Verification Conditions

Generation of verification conditions by tools such as Why3 [Bobot et al. 2011] or Dafny [Leino 2010] is implemented using a "weakest precondition" predicate transformer in a Hoare-style logic. Assertions and loop invariants are used to link the code to a formal specification. The prophetic annotations we have introduced in Section 2 can be seen as assertions, as in this naive implementation of the matrix product from Section 2:

```
1  for i = 0 to N - 1 do
2    for j = 0 to M - 1 do
3      r[] := 0
4      assert (r[] = 0)
5      for k = 0 to P - 1 do
6        r[] := r[] + a[i, k] * b[k, j]
7        assert (r[] = R(i, j, k))
8      c[i, j] := r[]
9      assert (c[i, j] = C(i, j))
```

On its own, these assertions are not enough to prove the equivalence: loop invariants are required to propagate information across loop iterations, such as the value of `r[]` at line 6 which is needed to prove the assertion at line 7. In this case, we need to infer the invariant for the loop on k that we used in Section 2:

```
5      for k = 0 to P - 1 do
6        invariant { r[] = if k=0 then 0 else R(i,j,k-1) }
7        ...
```

Expressions

$$
\begin{array}{rr}
e, \iota ::= x \mid l & \text{variable and literals} \\
\mid a[e_1, \ldots, e_n] & \text{array indexing} \\
\mid A(e_1, \ldots, e_n) & \text{tensor indexing} \\
\mid \texttt{let } x = e_1 \texttt{ in } e_2 & \text{let expression} \\
\mid e_1 + e_2 \mid n \cdot e & \\
\mid \lfloor e/n \rfloor \mid e \bmod n & \text{linear arithmetic} \\
\mid e_1 = e_2 \mid e_1 \leq e_2 & \text{comparisons} \\
\mid e_1 \texttt{ \&\& } e_2 \mid \texttt{!}e & \text{Boolean connectives} \\
\mid \text{select}(e_1, e_2, e_3) & \text{conditional expression} \\
\mid f(e_1, \ldots, e_n) & \text{pure function call}
\end{array}
$$

Commands

$$
\begin{array}{l}
c ::= \texttt{skip} \\
\quad \mid c_1 \texttt{ ; } c_2 \\
\quad \mid a[e_1, \ldots, e_n]\ \{e'\} := e \\
\quad \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \\
\quad \mid \texttt{let } x = e \texttt{ in } c \\
\quad \mid \texttt{allocate } a : \tau[e_1 \times \cdots \times e_n] \texttt{ in } c \\
\quad \mid \texttt{for}_k\ x \leq e \texttt{ do } c
\end{array}
$$

Loop kind

$$k ::= \texttt{seq} \mid \texttt{par}$$

Fig. 1. Syntax of the implementation language Sched

In the general case, if `r[]` was an array, the invariant would also need to specify the values at the indices which are not written by the loop.

The process we use to generate invariants, presented in this section, proceeds as follows. We abstract the memory state of the program using a symbolic heap h, and we abstract the behavior of a statement s by a symbolic heap $\Delta h(s)$ which represents the prophetic writes performed by s. We note $\triangleright$ the (associative) update combinator on symbolic heaps, such that the prophetic evaluation of statement s in any abstract heap h is $h \triangleright \Delta h(s)$. For a loop `for i = 0 to N do s`, after $0 \leq i \leq N$ iterations starting in any heap h, we end up in $h \triangleright \Delta h(s[\texttt{i} := 0]) \triangleright \cdots \triangleright \Delta h(s[\texttt{i} := i])$. If the concrete evaluation agrees with the prophetic evaluation, this must be a loop invariant when taking h to be the result of the prophetic evaluation up to that point. Because we verify the equality of concrete and prophetic evaluation on each assignment, this inferred invariant is correct by construction and does not need to be checked.

In the remainder of this section, we assume that a signature $\mathcal{S}$ is given, with a specification as a SARE S over $\mathcal{S}$. We formulate the verification condition generator as a symbolic evaluator, using symbolic heaps whose locations are array names indexed by affine expressions of the outer variables. The values in the symbolic heaps are not referring to any mutable state, only to outer loop iterators and specification tensors, and can be considered a form of ghost state. The precise definition of the symbolic heaps using affine sets is postponed until Section 4.5.

The specification deals with possibly infinite domains, hence the implementation can only implement a subset of the specification. We want to express that evaluating the implementation in a memory where the input arrays match a subset of the input tensors results in a new memory where the output arrays match a corresponding subset of the output tensors.

We will occasionally write $\overline{e}$ for a sequence of expressions $e_1, \ldots, e_n$. When meaningful, different sequences in the same expression can have different lengths. The length of the sequence is written $|\overline{e}|$.

4.2 Expressions and Types

The grammars for the expressions of Sched are given in Fig. 1. The distinguished type $\mathbb{A}$ of affine expressions is used for array indices and loop bounds, while the distinguished type $\mathbb{B}$ of affine

propositions is used in conditionals. Expressions of those distinguished types are required to be piecewise quasi-affine expressions (resp. propositions) of the parameters and outer loop iterators. In particular, values of type $\mathbb{A}$ and $\mathbb{B}$ cannot be written to arrays, but we allow casting them to a value type such as `int32` using a conversion function. We will ignore the issue of possible overflows in the index computations of $\mathbb{A}$ throughout, and simply assume that $\mathbb{A}$ is represented using unbounded integers. We discuss possible ways of handling integer overflow in Section 7.2.

The typing contexts, ranged over by Γ, are heterogeneous. They can contain:

- Variable bindings $x : \tau$ from a name to its type. We only consider bindings of type $\mathbb{A}$ or $\mathbb{B}$ to simplify the presentation. Note that variables of value types can be represented using zero-dimensional arrays.
- Array bindings from names to array shapes $a : \tau[e_1 \times \cdots \times e_n]$. An array shape $\tau[e_1 \times \cdots \times e_n]$ represents a n-dimensional, rectangular array containing values of type τ, where dimension i has length e_i. Indices start at 0 in each dimension. Arrays are mutable, and are not initialized. A mutable variable is a zero-dimensional array.
- Boolean expressions e, to keep track of the bounds on loop indices and other conditionals. These are the path conditions of our symbolic evaluator.

Whenever we write a context $\Gamma, x : \tau$ (resp. $\Gamma, a : \tau[\overline{e}]$), we implicitly assume that x (resp. a) is not bound in Γ. In the case of an array binding, we also assume that the e_i are well-typed of type $\mathbb{A}$, using the typing rules below. More generally, we follow the Barendregt convention of α-renaming the bound variables to avoid name conflicts.

We assume that all contexts Γ start with a common prefix Γ_P which only contain variable bindings for the parameters. They are always usable in expressions.

The expressions of Sched can be separated into three categories. *Afine expressions* are used in array indices, loop bounds, and conditionals. They are restricted to syntactically affine combinations of program parameters and outer affine variables, and are typed using the judgement $\Gamma \vdash e : \tau$ (read "under assumptions Γ, the affine expression e has type τ"), where $\tau = \mathbb{A}$ for integer affine expressions or $\tau = \mathbb{B}$ for boolean affine expressions. *Executable expressions* have a value type and can contain array reads. They are found on the right-hand side of array assignments, and typed using the judgement $\Gamma \vdash_{\mathsf{a}} e : \tau$, which is read "under assumptions Γ, the expression e has executable type τ". *Prophetic expressions* have a value type but contain tensor calls instead of array reads, and are found as prophetic annotations on array assignments. Prophetic expressions are typed using the judgement $\Gamma \vdash_{\mathsf{A}} e : \tau$, read "under assumptions Γ, the expression e has prophetic type τ". The rules for all those judgements are fairly standard, and given in Fig. 2. `select` is an eager affine conditional and can appear in any expresion.

The grammar of commands c is given in Fig. 1. It is mostly unsurprising, except that assignments are annotated with a prophetic expression e' as presented in Section 2. In addition, loops can be either sequential or parallel, which is represented by an annotation. The difference between the kinds of loops will be explained in Section 4.3. Commands are imperative, and do not have a type; instead, we give a well-formedness judgement $\Gamma \vdash c : \Delta h$ in Fig. 3. The argument Δh in this judgement is a symbolic heap, that collects the set of *prophetic* updates performed by c. Symbolic heaps are symbolic representations of heaps, that is, mappings from array locations to symbolic expressions. The precise definition of symbolic heaps and their operations is postponed until Section 4.5.

Note that Sched lacks both recursion and unbounded (`while`) loops, making it non-Turing complete, like Halide. This restriction might seem significant; however, we follow the design of domain-specific compilers for image processing and machine learning such as Halide or Tensor Comprehensions on this. They are typically used on kernels which perform a fixed computation

$$\textsc{T-Var}\ \frac{\vdash \Gamma, x : \tau}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{T-Array}\ \frac{a : \tau[e'_1 \times \cdots \times e'_n] \in \Gamma \qquad \forall 1 \leq i \leq n,\ \Gamma \vdash e_i : \mathbb{A}}{\Gamma \vdash_{\mathsf{a}} a[e_1, \ldots, e_n] : \tau}$$

$$\textsc{T-Tensor}\ \frac{A \in \mathcal{S} \qquad \forall 1 \leq i \leq n_A,\ \Gamma \vdash e_i : \mathbb{A}}{\Gamma \vdash_{\mathsf{A}} A(e_1, \ldots, e_{n_A}) : \tau_A} \qquad \textsc{T-Bool}\ \frac{b \in \{\texttt{true}, \texttt{false}\}}{\vdash b : \mathbb{B}} \qquad \textsc{T-Int}\ \frac{}{\vdash n : \mathbb{A}}$$

$$\textsc{T-Call}\ \frac{\begin{array}{c}\forall 1 \leq i \leq n, k_i \in \{k, \emptyset\} \Rightarrow \Gamma \vdash_{k_i} e_i : \tau_i \\ k \in \{\mathsf{a}, \mathsf{A}\} \qquad f \in \mathcal{F} \qquad \tau_f = \tau_1 \times \cdots \times \tau_n \to \tau\end{array}}{\Gamma \vdash_k f(e_1, \ldots, e_n) : \tau}$$

$$\textsc{T-Select}\ \frac{k \in \{\mathsf{a}, \mathsf{A}, \emptyset\} \qquad \Gamma \vdash e_1 : \mathbb{B} \qquad \Gamma, e_1 \vdash_k e_2 : \tau \qquad \Gamma, \neg e_1 \vdash_k e_3 : \tau}{\Gamma \vdash_k \text{select}(e_1, e_2, e_3)}$$

$$\textsc{T-Let}\ \frac{k \in \{\mathsf{a}, \mathsf{A}, \emptyset\} \qquad \Gamma \vdash e_1 : \mathbb{A} \qquad \Gamma, x : \mathbb{A} \vdash_k \tau}{\Gamma \vdash_k \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} \qquad \textsc{T-Add}\ \frac{\Gamma \vdash e_1 : \mathbb{A} \qquad \Gamma \vdash e_2 : \mathbb{A}}{\Gamma \vdash e_1 + e_2 : \mathbb{A}}$$

$$\textsc{T-Mul}\ \frac{\Gamma \vdash e : \mathbb{A}}{\Gamma \vdash n \cdot e : \mathbb{A}} \qquad \textsc{T-Div}\ \frac{\Gamma \vdash e : \mathbb{A} \qquad n > 0}{\Gamma \vdash \lfloor e/n \rfloor : \mathbb{A}} \qquad \textsc{T-Mod}\ \frac{\Gamma \vdash e : \mathbb{A} \qquad n > 0}{\Gamma \vdash e \bmod n : \mathbb{A}}$$

$$\textsc{T-Cmp}\ \frac{\Gamma \vdash e_1 : \mathbb{A} \qquad \Gamma \vdash e_2 : \mathbb{A} \qquad \odot \in \{=, \leq\}}{\Gamma \vdash e_1 \odot e_2 : \mathbb{B}} \qquad \textsc{T-And}\ \frac{\Gamma \vdash e_1 : \mathbb{B} \qquad \Gamma \vdash e_2 : \mathbb{B}}{\Gamma \vdash e_1\ \texttt{\&\&}\ e_2 : \mathbb{B}} \qquad \textsc{T-Not}\ \frac{\Gamma \vdash e : \mathbb{B}}{\Gamma \vdash\ !e : \mathbb{B}}$$

Fig. 2. Typing rules for Sched expressions

$$\textsc{T-Allocate}\ \frac{\Gamma, a : \tau[e_1 \times \cdots \times e_n] \vdash c : \Delta h \qquad \forall 1 \leq i \leq n,\ \Gamma \vdash e_i : \mathbb{A}}{\Gamma \vdash \texttt{allocate}\ a : \tau[e_1 \times \cdots \times e_n]\ \texttt{in}\ c : \Delta h \backslash a} \qquad \textsc{T-Skip}\ \frac{}{\Gamma \vdash \texttt{skip} : \emptyset}$$

$$\textsc{T-Seq}\ \frac{\Gamma \vdash c_1 : \Delta h_1 \qquad \Gamma \vdash c_2 : \Delta h_2}{\Gamma \vdash c_1\ ;\ c_2 : \Delta h_1 \triangleright \Delta h_2} \qquad \textsc{T-If}\ \frac{\Gamma \vdash e : \mathbb{B} \qquad \Gamma, e \vdash c_1 : \Delta h_1 \qquad \Gamma, \neg e \vdash c_2 : \Delta h_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : (\Delta h_1 \restriction e) \uplus (\Delta h_2 \restriction \neg e)}$$

$$\textsc{T-SeqLoop}\ \frac{\Gamma \vdash e : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, 0 \leq x < e \vdash c : \Delta h}{\Gamma \vdash \texttt{for}_{\texttt{seq}}\ x \leq e\ \texttt{do}\ c : \texttt{lexmax}_{0 \leq x < e} \Delta h} \qquad \textsc{T-ParLoop}\ \frac{\Gamma \vdash e : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, 0 \leq x < e \vdash c : \Delta h}{\Gamma \vdash \texttt{for}_{\texttt{par}}\ x \leq e\ \texttt{do}\ c : \bigcup_{0 \leq x < e} \Delta h}$$

$$\textsc{T-Assign}\ \frac{\Gamma \vdash_a e : \tau \qquad a : \tau[e'_1 \times \cdots \times e'_n] \in \Gamma \qquad \Gamma \vdash_A t : \tau \qquad \forall 1 \leq i \leq n,\ \Gamma \vdash e_i : \mathbb{A}}{\Gamma \vdash a[e_1, \ldots, e_n]t := e : \{a[e_1, \ldots, e_n] \mapsto t\}}$$

Fig. 3. Well-formedness rules for statements

parameterized by the problem size, and which might be repeated in an unbounded outer loop. This precludes the compiler from optimizing across the boundary of that outer loop. Such optimizations can be beneficial in particular for the implementation of recurrent neural network, and their validation could be the topic for future work.

4.3 Dynamic Semantics

We will give a single semantics for all Sched expressions, targeting the set of values $\mathcal{V} = \mathcal{U} \cup \mathbb{Z} \cup \{\texttt{true}, \texttt{false}\} \uplus \{\bot\}$. $\bot$ is a distinguished value representing an undefined or unknown computation; affine expressions have values in $\mathbb{Z} \cup \{\texttt{true}, \texttt{false}, \bot\}$ while executable and prophetic expressions have values in $\mathcal{U} \uplus \{\bot\}$.

The semantics is given in a dynamic environment $\langle \mathcal{E}; \mu \rangle$ where the stack $\mathcal{E}$ is a map from variable names to $\mathbb{Z} \cup \{\texttt{true}, \texttt{false}\}$, and the memory μ is a map from locations to $\mathcal{U} \uplus \{\bot\}$. Locations, ranged over by ℓ, are multidimensional array cells, that is, array names indexed by integers:

$$\ell ::= a[n_1, \ldots, n_n]$$

The evaluation function $[\![e]\!]_{\mathcal{E};\mu}$ for expressions is fairly standard, and not reproduced here for brevity. It assumes the presence of a model M of the specification for the tensor reads. $\bot$ is treated as an error value, and is propagating: if any computation involves $\bot$, the result is $\bot$. Following Halide's design, `select` is eager, and evaluates all its arguments before choosing a value depending on the conditional, although a lazy version could be considered. Illegal expressions (such as accessing an undefined variable, reading an undefined location, or performing an ill-typed operation, including using a tensor or function with incorrect arity) evaluate to $\bot$.

If e does not contain any array reads (e.g. e is an affine or prophetic expression), then clearly $[\![e]\!]_{\mathcal{E};\mu}$ is independent of μ, and we write the corresponding value $[\![e]\!]_{\mathcal{E}}$.

The evaluation of a command c in environment $\langle \mathcal{E}; \mu \rangle$ into updates $\Delta\mu$ with reads ρ, written $\mathcal{E}; \mu \vdash c \Downarrow_u \Delta\mu; \rho$, is defined inductively in Fig. 4, and deserves some consideration. Let us ignore ρ for the moment and focus on the updates $\Delta\mu$. It is not a final memory; rather, it should be understood as a set of updates to be applied to the initial memory μ. Hence, we call our semantics an *update semantics*. The usual formulation of a dynamic semantics $\mathcal{E}, \mu \vdash c \Downarrow \mu'$, where μ' is the final memory, can be related to our update semantics as follows:

$$(\exists \rho. \mathcal{E}; \mu \vdash c \Downarrow_u \Delta\mu; \rho) \text{ if, and only if, } \mathcal{E}, \mu \vdash c \Downarrow \mu \triangleright \Delta\mu \tag{2}$$

The update operator $\triangleright$, read "then", is defined as $\mu_1 \triangleright \mu_2 = (\mu_1 \backslash \mathrm{dom}(\mu_2)) \uplus \mu_2$. It can best be understood through rule U-Seq: it contains the mappings of μ_2 and those of μ_1 not overwritten by μ_2. $\triangleright$ is associative, and we define the iterated update $\blacktriangleright_{0 \le i < n} \mu_i = \mu_0 \triangleright \ldots \triangleright \mu_{n-1}$.

The first motivation for using an update semantics instead of a more traditional presentation is that, combined with prophetic expressions, it allows for a presentation that can be easily evaluated symbolically. Consider rule U-For defining the semantics of a sequential loop. Since $\blacktriangleright$ is the repeated application of $\triangleright$, it amounts to repeating rule U-Seq $[\![e]\!]_{\mathcal{E};\mu_0}$ times: we evaluate an iteration i in a memory that has seen the updates from all iterations $j < i$. Indeed, if we can compute a symbolic representation of memory μ'_i as a function of i without knowing the input memory, then it only remains to build a symbolic version of the iterated update $\blacktriangleright$ to have a symbolic evaluator. Prophetic expressions give us the first component, while the affine restrictions mean that we can compute an exact symbolic representation of $\blacktriangleright$.

The second motivation for using an update semantics is that it allows us to express the semantics of parallel loops in a lightweight manner. Workflows expressed in n-dimensional array languages are often highly parallelizable; in fact, many workflows require parallelization to achieve the high performance promised by these languages, especially on highly parallel hardware such as GPUs.

The difficulty in designing a symbolic evaluator for parallel loops is linked to the handling of synchronization primitives. Fortunately, compilers for n-dimensional array languages mostly use synchronization within parallel loops using a rather predictable pattern: parallel loops can be separated in sequential stages, with an unconditional global barrier between stages. There are no other synchronization primitives than those global barriers.

We can represent a parallel loop with multiple stages separated by barriers as multiple parallel loops of the same size where each barrier is replaced by closing the current loop and starting a fresh one. In fact, we do not even have to do this transformation: compilers such as Halide or Tensor Comprehension use non-communicating parallel loops and only introduce barriers late in the compilation process, and only when targeting GPUs. The algorithm for the introduction of barriers could be verified independently.

Since our parallel loops are synchronization-free, it should be possible to execute the threads in any order and get the same result. However, this is not true if there are data races. Hence, we wish to only allow race-free executions. We do this by evaluating each iteration of the loop independently in rule U-Par, and then requesting that the memories μ_i and μ_j produced by distinct operations are disjoint, i.e. different threads never write to the same location. In addition, this is where we use the set of read locations ρ to ensure that no thread reads a location that is written by a different thread.

The rule U-Par ensures there are no race conditions using the disjointness operator # to ensure that different threads only access different regions of memory.

The rule U-Allocate allows for the local allocation of arrays. It creates a fresh local array and initializes it with undefined values. Local arrays are properly scoped: local arrays allocated in different threads do not alias. Writes and reads to the local arrays are erased when evaluating the `allocate` statement.

The rule U-Assign uses a domain check to ensure that there are no out-of-bounds writes, and computes the set of locations read during evaluation of the right-hand side e using the `reads` function. The definition of `reads` is not reproduced here; $\texttt{reads}(e)$ is the list of all sub-expressions of e which are array accesses. For instance, $\texttt{reads}(0 \times x[i]) = \{x[i]\}$. $[\![\texttt{reads}(e)]\!]_{\mathcal{E};\mu}$ is the set of locations obtained by evaluating these accesses.

4.4 Symbolic Evaluation

We will now build a symbolic evaluator for Sched. The symbolic evaluator will make use of the prophetic expressions in order to break the self-referential cycle of sequential loops.

The judgement $\Gamma; h \vdash C \implies c : \Delta h; R$ is presented in Fig. 5 and follows the dynamic evaluation rules of Sched. The typing environment Γ and the input heap h are symbolic representations of the dynamic environment $\langle \mathcal{E}; \mu \rangle$. The symbolic updates Δh and read locations R are symbolic versions of the updates $\Delta\mu$ and read locations ρ. C is an additional parameter: it is a symbolic set of equalities, with affine quantification. It will collect the verification conditions corresponding to the equality between prophetic expression and executable expression at each assignment.

Surprisingly, most of the rules for the symbolic evaluator are straightforward — assuming that the constraints, updates, and reads can be represented symbolically, which we will show later by exploiting the affine restrictions we place on programs. The rules match closely the dynamic semantics; in addition, except for the rule S-SeqLoop, they are algorithmic: the outputs Δh and R (in red) and the constraints C (in purple) never appear as inputs of the inductive applications of the predicate.

In the case of sequential loops, the output Δh appears as input to the recursive call inside the `lexmax`. This would block the symbolic evaluation, as it requires us to *invent* the summary Δh of a single iteration of the loop. Fortunately, we have prophetic expressions to solve the issue: by examining the rules, we can see that the output Δh only depends on the prophetic expressions,

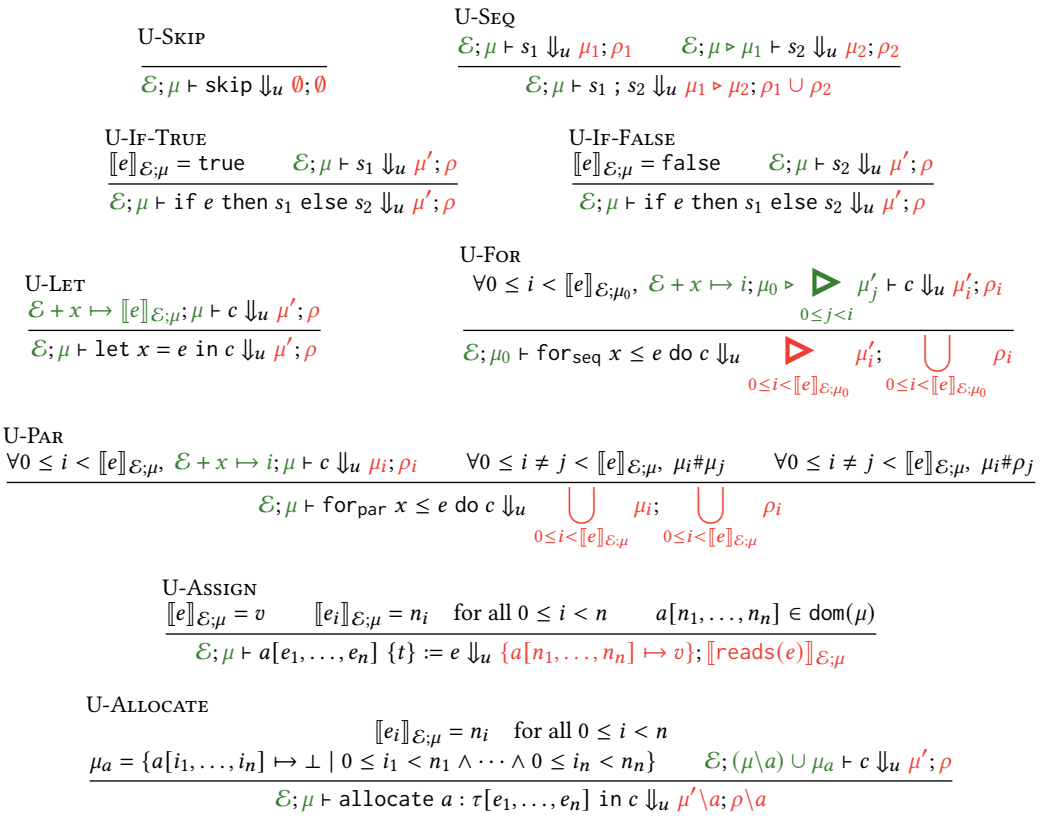

Fig. 4. Update semantics for Sched statements

ignoring the right-hand side of all assignments. In fact, the summary is always the one computed by the well-formedness judgement for statements:

Lemma 4.1. *If* $\Gamma;h \vdash C \implies c : \Delta h;R$ *holds, then* $\Gamma \vdash c : \Delta h$ *also holds.*

Proof. The proof follows by induction and remarking that the rules of $\Gamma \vdash c : \Delta h$ are exactly those of $\Gamma;h \vdash C \implies c : \Delta h;R$ with the premises involving h, C and R removed. □

Hence, the following strategy for the evaluation of sequential loops: first, we compute the iteration summary using the $\Gamma \vdash c : \Delta h$ judgement which does not require a precise description of the symbolic heap. Then, we plug the resulting Δh in the input heap $h \triangleright \texttt{lexmax}_{0\le x'<x} \Delta h[x/x']$ of the symbolic evaluation judgement to compute the constraint C and the reads R. By using $\Gamma \vdash c : \Delta h$, we rely on the fact that the concrete evaluation will follow the prophetic evaluation. The constraints C are the price we pay for that: they keep track of the equalities that must hold for this property to be true, tying the knot and ensuring the well-foundedness of our approach.

Another rule that requires some consideration is rule S-Assign. As mentioned above, $\texttt{reads}(e)$ is the list of the array accesses in e, interpreted here as a symbolic set of locations. This symbolic set of locations is then used in rule S-Par in order to check for race freedom. In addition, rule S-Assign has conditions $\Gamma \vdash [\![e]\!]_h \ne \emptyset$ (ensuring all array reads in e are defined) and $\Gamma \vdash 0 \le e_i < e'_i$

$$\textsc{S-Allocate}\quad \frac{\Gamma, a : \tau[e_1 \times \cdots \times e_n]; h \vdash C \implies c : \Delta h; R \qquad \Gamma \vdash e_i : \mathbb{A} \quad \text{for all } 1 \le i \le n}{\Gamma; h \vdash C \implies \texttt{allocate}\ a : \tau[e_1 \times \cdots \times e_n]\ \texttt{in}\ c : \Delta h \backslash a; R \backslash a}$$

$$\textsc{S-Skip}\quad \frac{}{\Gamma; h \vdash \emptyset \implies \texttt{skip} : \emptyset; \emptyset}$$

$$\textsc{S-Seq}\quad \frac{\Gamma; h \vdash C_1 \implies c_1 : \Delta h_1; R_1 \qquad \Gamma; h \triangleright \Delta h_1 \vdash C_2 \implies c_2 : \Delta h_2; R_2}{\Gamma; h \vdash C_1 \cup C_2 \implies c_1\ ;\ c_2 : \Delta h_1 \triangleright \Delta h_2; R_1 \cup R_2}$$

$$\textsc{S-Let}\quad \frac{\Gamma \vdash e : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, x = e; h \vdash C \implies c : \Delta h; R}{\Gamma; h \vdash C[x/e] \implies \texttt{let}\ x = e\ \texttt{in}\ c : \Delta h[x/e]; R[x/e]}$$

$$\textsc{S-SeqLoop}\quad \frac{\Gamma \vdash e : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, 0 \le x < e; h \triangleright \texttt{lexmax}_{0 \le x' < x} \Delta h[x'/x] \vdash C \implies c : \Delta h; R}{\Gamma; h \vdash \bigcup_{0 \le x < e} C \implies \texttt{for}_{\texttt{seq}}\ x \le e\ \texttt{do}\ c : \texttt{lexmax}_{0 \le x < e} \Delta h; \bigcup_{0 \le x < e} R}$$

$$\textsc{S-ParLoop}\quad \frac{\Gamma \vdash e : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, 0 \le x < e; h \vdash C \implies c : \Delta h; R \qquad \Gamma, x : \mathbb{A}, y : \mathbb{A}, 0 \le x \ne y < e \vdash \texttt{dom}(\Delta h) \# R[x/y] \quad \Gamma, x : \mathbb{A}, y : \mathbb{A}, 0 \le x \ne y < e \vdash \texttt{dom}(\Delta h) \# \texttt{dom}(\Delta h)[x/y]}{\Gamma; h \vdash \bigcup_{0 \le x < e} C \implies \texttt{for}_{\texttt{par}}\ x \le e\ \texttt{do}\ c : \bigcup_{0 \le x < e} \Delta h; \bigcup_{0 \le x < e} R}$$

$$\textsc{S-If}\quad \frac{\Gamma \vdash e : \mathbb{B} \qquad \Gamma, e; h \vdash C_1 \implies c_1 : \Delta h_1; R_1 \qquad \Gamma, \neg e; h \vdash C_2 \implies c_2 : \Delta h_2; R_2}{\Gamma; h \vdash (C_1 \restriction e) \uplus (C_2 \restriction \neg e) \implies \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : (\Delta h_1 \restriction e) \uplus (\Delta h_2 \restriction \neg e); (R_1 \restriction e) \uplus (R_2 \restriction \neg e)}$$

$$\textsc{S-Assign}\quad \frac{\begin{array}{c} a : \tau[e'_1 \times \cdots \times e'_n] \in \Gamma \qquad \Gamma \vdash_a e : \tau \\ \Gamma \vdash_A t : \tau \qquad \Gamma \vdash e_i : \mathbb{A} \quad \text{for all } 1 \le i \le n \qquad \Gamma \vdash 0 \le e_i < e'_i \quad \text{for all } 1 \le i \le n \qquad \Gamma \vdash [\![e]\!]_h \ne \emptyset \end{array}}{\Gamma; h \vdash [\![e]\!]_h \subset \{t\} \implies a[e_1, \ldots, e_n]\ \{t\} := e : \{a[e_1, \ldots, e_n] \mapsto t\}; \texttt{reads}(e)}$$

Fig. 5. Typing rules and prophecies

(ensuring the write is within the array bounds). These two conditions are quasi-affine formulas, whose entailment from the context Γ must be checked.

4.5 Affine Symbolic Heaps

Let us now explain in more details how the symbolic representations used by the evaluator are represented. A symbolic heap is a finite union of symbolic chunks, each represented by an array name, local variables $\overline{x}$, index expressions $\overline{e}$, value expression e, and condition ϕ:

$$\{\overline{x}.a[\overline{e}] \mapsto e \mid \phi\}$$

The index expressions are affine combinations of the local variables and the free variables of the chunk, while ϕ is a conjunction of affine equalities and inequalities of the same variables. However, the value expression e is an arbitrary prophetic expression (i.e. it does not contain array accesses, and can be evaluated independently of program memory).

Given a model M for the specification, we can evaluate a symbolic heap as a memory, by taking the union of its definition on chunks.

$$[\![\{\overline{x}.a[\overline{e}] \mapsto e \mid \phi\}]\!]_{\mathcal{E}} = \{a[[\![\overline{e}]\!]_{\mathcal{E}+\overline{x}\mapsto\overline{n}}] \mapsto [\![e]\!]_{\mathcal{E}+\overline{x}\mapsto\overline{n}} \mid [\![\phi]\!]_{\mathcal{E}+\overline{x}\mapsto\overline{n}}\}$$

The braces on the left are the syntactic objects defining the symbolic chunks, while the braces on the right describe a comprehension in the meta-theory. This evaluates to the mappings such that there exists integer values n_i for the x_i for which $\phi(\overline{n})$ holds.

Note that we do not enforce the functionality of our symbolic heaps. The same location could be associated with different expressions, or the same expressions for different values of $\overline{x}$, which ultimately evaluate to different values. If that is the case, the evaluation function replaces the value with $\bot$.

We will define symbolic sets of locations (ranged over by R), as symbolic heaps without a range, written $\{\overline{x}.a[\overline{e}] \mid \phi\}$. They evaluate to sets of locations. The domain of a symbolic heap, $\mathtt{dom}(h)$, is taken by erasing the range of the heap. Similarly, we define sets of symbolic expressions (ranged over by C) as symbolic heaps without locations, written $\{\overline{x}.e \mid \phi\}$. They evaluate to sets of values. For a set of boolean expressions C, we write $\mathcal{E} \vDash C$ to indicate that $\mathtt{false} \notin [\![C]\!]_{\mathcal{E}}$; that is, $\mathcal{E} \vDash C$ if all the equalities in C hold.

Since symbolic domains are equivalently defined by a single formula ϕ for each array, we can define the usual set operations (union, intersection, difference, etc.) by applying the corresponding connective to the ϕ formulas appropriately. The union is trivial by definition for each of those structures; we can also intersect a symbolic heap h (or symbolic domain, or symbolic range) with an affine condition e, written $h \upharpoonright e$.

In addition, we need to define the following operations, used in our dynamic semantics, for which we need a symbolic version:

- For the update combinator $h_1 \triangleright h_2$ we can simply use its defining equation $(h_1 \backslash \mathtt{dom}(h_2)) \cup h_2$ which is already expressible;
- For the iterated union $\bigcup_{0\le x<e} h$, where e is a symbolic expression, we can simply introduce an additional local quantifier:

$$\bigcup_{0\le y<e} \{\overline{x}.a[\overline{e}] \mapsto e' \mid \phi\} = \{\overline{x}y.a[\overline{e}] \mid e' \mid 0 \le y < e \wedge \phi\}$$

- The iterated update can also be expressed symbolically. If we recall its definition, the iterated update $\blacktriangleright_{0\le i<n} \mu_i$ of a sequence of memories is the memory μ which contains, for a location ℓ, the value $v = \mu_i(\ell)$ for the last i which writes to that location (if it exists). Hence, the contribution of an iteration i is the subset of μ_i which is not overwritten by any of the μ_j for $j > i$.
  Using this alternate formulation, we can represent $\blacktriangleright_{0\le x<e} h$ using the operations we have already defined and a substitution operation $h[x'/x]$, which replaces the variable x in h with the variable x':

$$\blacktriangleright_{0\le x<e} h = \bigcup_{0\le x<e} \left(h \backslash \mathtt{dom}\Big(\bigcup_{x<x'<e} h[x'/x] \Big) \right)$$

  If we extend this construction to handle multiple variables at once, only keeping the lexicographically latest write in a bounded multidimensional space, we obtain a lexicographic maximum combinator which is well-studied in the polyhedral literature and admits efficient computations using Parametric Integer Programming without nested quantification.
  Even though we are only computing the maximum one dimension at a time, we will write the symbolic version of our iterated update combinator $\mathtt{lexmax}_{0\le x<e} h$.

- The compatibility $h_1 \supset h_2$ of two heaps indicate that they agree on their common domain. It can be expressed using a set of equalities (where $a[\overline{e}] = a'[\overline{e'}]$ is `false` if $a \neq a'$ or $|\overline{e}| \neq |\overline{e'}|$, and $e_1 = e'_1 \wedge \cdots \wedge e_{|\overline{e}|} = e'_{|\overline{e}|}$ otherwise):

$$\{\overline{x}.a[\overline{e}] \mapsto e \mid \phi\} \supset \{\overline{y}.a'[\overline{e'}] \mapsto e' \mid \phi'\} = \{\overline{xy}.e = e' \mid \phi \wedge \phi' \wedge a[\overline{e}] = a'[\overline{e'}]\}$$

  We have $\mathcal{E} \vDash h_1 \supset h_2$ if, and only if, for all $\ell \mapsto v_1 \in [\![h_1]\!]_{\mathcal{E}}$ and $\ell \mapsto v_2 \in [\![h_2]\!]_{\mathcal{E}}$, $v_1 = v_2$. That is, $\mathcal{E} \vDash h_1 \supset h_2$ if and only if $[\![h_1]\!]_{\mathcal{E};M} \supset [\![h_2]\!]_{\mathcal{E};M}$.

Finally, we define the evaluation function of an expression e in a symbolic heap h, $[\![e]\!]_h$, which yields a symbolic set of expressions by replacing array indices in e with the corresponding expression(s) in h. If any of the array indices in e are not defined in h, the resulting evaluation $[\![e]\!]_h$ is empty.

We note that none of the operations presented here strictly depend on the affine restrictions we have placed on programs. However, if the formulas we consider are quasi-affine, we stay in a decidable subset of arithmetic, namely Presburger arithmetic with modulus, which also admits quantifier elimination. This allows us to use `isl` [Verdoolaege 2010], a specialized library to represent parametric integer sets, and efficiently implement the operations described above.

4.6 Correctness Proof

We now present the main arguments forming a proof that our symbolic evaluator correctly captures the dynamic semantics of Sched.

To that regard, we say that a local environment $\mathcal{E}$ is compatible with a typing environment Γ, denoted $\mathcal{E} \vDash \Gamma$, if:

- All variables in Γ have a corresponding value in $\mathcal{E}$ of the right type.
- For any array declaration in Γ, the array bounds evaluate to integers.

A memory μ is compatible with typing environment Γ in local environment $\mathcal{E}$ if $\mathcal{E} \vDash \Gamma$ and further for each array declaration $a : \tau[e_1 \times \cdots \times e_n]$ in Γ, the locations in μ for array a are a subset of the $a[i_1, \ldots, i_n]$ where $0 \leq i_i < [\![e_i]\!]_{\mathcal{E};\mu}$, with values in $[\![\tau]\!]$.

Then, we can quite easily show that well-typed prophetic and index expressions evaluate to values of the proper type.

Lemma 4.2. *If $\Gamma \vdash e : \tau$ or $\Gamma \vdash_A e : \tau$ then for all $\langle \mathcal{E}; \mu \rangle$ such that $\Gamma \vdash \langle \mathcal{E}; \mu \rangle$, we have $[\![e]\!]_{\mathcal{E};\mu} \in [\![\tau]\!]$. Furthermore, if a heap h is well-formed in Γ, then for all $\mathcal{E}$ compatible with Γ, $[\![h]\!]_{\mathcal{E}}$ is compatible with Γ in $\mathcal{E}$.*

Proof. The proof is immediate by induction on the typing judgement, since tensors accesses are total and prophetic and index expressions cannot contain array reads. □

The corresponding lemma for well-typed expressions is more involved, as it require additional conditions on the memory to ensure all arrays are defined.

Lemma 4.3. *If $\Gamma \vdash_a e : \tau$, h is well-formed in Γ, and $\Gamma \vdash [\![e]\!]_h \neq \emptyset$ then for all $\mathcal{E}$ compatible with Γ, we have $[\![e]\!]_{\mathcal{E};[\![h]\!]_{\mathcal{E}}} = [\![[\![e]\!]_h]\!]_{\mathcal{E}} \in [\![\tau]\!]$.*

Proof. The proof follows by structural induction on e. The inductive cases are mechanic, and the base case is that of an array access, where the condition $\Gamma \vdash [\![a[e_1, \ldots, e_n]]\!]_h \neq \emptyset$ ensures that in all environments $\mathcal{E}$ compatible with Γ, $a[[\![e_1]\!]_{\mathcal{E}}, \ldots, [\![e_n]\!]_{\mathcal{E}}]$ is defined in $[\![h]\!]_{\mathcal{E}}$ with a value of type τ_a, from which the equality follows. □

The same property holds for the symbolic evaluator, which is the main correctness condition for our verifier.

Theorem 4.4. *If* $\Gamma; h \vdash C \implies c : \Delta h; R$, *then for all* $\mathcal{E}$ *compatible with* Γ *such that* $\mathcal{E} \vDash C$, *we have* $\mathcal{E}; \mu \vdash c \Downarrow_u [\![\Delta h]\!]_{\mathcal{E}}; [\![R]\!]_{\mathcal{E}}$, *for any memory* $\mu \supseteq [\![h]\!]_{\mathcal{E}}$ *such that the locations defined in* μ *are exactly those declared in* Γ.

Proof. The proof is again by structural induction on the symbolic evaluation judgement. It relies on the fact that, due to the updating nature of our semantics, an evaluation in a memory μ stays valid in a larger memory $\mu' \supseteq \mu$ and evaluates to the same set of updates. The most interesting case is that of S-Assign, where we use Lemma 4.3, the fact that evaluation to a non-$\bot$ value is preserved in a larger memory, and the $\texttt{false} \notin [\![[\![e]\!]_h \subset \{t\}]\!]_{\mathcal{E}}$ condition to show that $[\![[\![e]\!]_h]\!]_{\mathcal{E}} = [\![r]\!]_{\mathcal{E}}$. Other cases worth mentioning are S-Allocate where the newly initialized memory is immediately weakened; and rule S-SeqLoop, which makes use of the equality $[\![\texttt{lexmax}_{0 \le x < e} h]\!]_{\mathcal{E}} = \blacktriangleright_{0 \le i < [\![e]\!]_{\mathcal{E}}} [\![h]\!]_{\mathcal{E}+x \mapsto i}$ to generate the premises for U-SeqLoop. □

5 EXPERIMENTS

This section discusses the implementation of the approach described in Section 4 in OCaml, using bindings to the `isl` library [Verdoolaege 2010] to represent affine expressions, and the Z3 SMT solver [de Moura and Bjørner 2008] to discharge the generated verification conditions. The implementation is available online [Clement and Cohen 2022].

5.1 Generation of Sched from Halide

The Halide compiler is a parameterized code generator: the schedule guides the generation of imperative code from the specification. We instrumented the Halide compiler to add prophetic annotations to the generated code, as described below. We also altered the compiler to produce a textual representation of the specification which can be parsed with our tool without having to interpret the C++ DSL.

Halide starts by generating an imperative loop nest where the arrays live in the specification index space, shifted to start at 0. We thus annotate each assignment with the stage, reduction variables, and a copy of the original tensor indices. Since transformations must preserve the semantics of the right-hand side of the assignment, subsequent transformations are applied to the annotations as if they were part of the right-hand side. We note that this approach can be applied to any compiler which generates a loop structure from the specification before possibly applying structure-preserving transformations. In particular, this is the case for polyhedral compilers.

Multidimensional arrays are eventually flattened into buffers in linear memory. We annotate accesses to the flat buffer with the original multidimensional array indices, so that we are able to recover the multidimensional affine program. Linearized and multidimensional indices are both kept, making the linearization step independently verifiable, as discussed in Section 7.1.

Finally, the cause of most non-trivial issues encountered when threading the annotations through the compiler pipeline are due to Halide's representation of vectorized loops. After the linearization to flat buffers, vectorized loops are replaced with a `ramp` intrinsic encoding multiple indices at once by implicitely representing the lane index. Halide relies on LLVM to transform this `ramp` intrinsic into hardware vector instructions when possible. Transformations involving this implicit lane index are not accurately captured by our annotations, requiring some engineering effort to support them. Although they do not pose theoretical issues, we have disabled two optimization passes that could cause complex multidimensional `ramp`s for the sake of simplicity.

5.2 OCaml Prototype

Our tool takes the annotated output generated from the Halide compiler and rebuilds a Halide algorithm and a Sched candidate implementation from that output.

We first convert all the indices and statement-level conditonals into piecewise quasi-affine functions represented using `isl`, and simplify them based on the context. `ramp`-based vectors are transformed into parallel loops with an explicit index. The code generated by Halide often features two "accidentally non-affine" constructs, which we convert into an equivalent affine representation. First, when multiple dimensions of sizes $e_1, \ldots, e_n$ are parallelized, Halide uses a single parallel loop with size $e_1 \times \cdots \times e_n$, which we recognize and split into nested parallel loops of affine sizes. Second, Halide can generate expressions of the form $\lfloor (e \times e' - 1)/e \rfloor$, which is always equal to $e' - \mathtt{sgn}(e)$ provided e is non-zero. Since Halide's simplifier fails to do so, we recognize and simplify this pattern.

After this initial conversion phase, our tool implements the algorithms described in Section 4. We rely on `isl` to represent symbolic heaps, domains, and ranges. The coalescing operation provided by `isl` has proven effective to keep simple representations of the symbolic heaps as they get updated. `isl` uses parametric integer programming [Feautrier 1988] to perform efficient quantifier elimination for the union and `lexmax` operators. Symbolic operations and simplifications on Presburger arithmetic is a unique asset of `isl` — tailored to the needs of polyhedral compilation; these operations would be much more cumbersome and inefficient to reproduce using Z3 (or any feasibility-focused solver).

We note two optimizations that we make in the representation to improve efficiency. In order to represent symbolic sets and heaps where the right-hand side expression does not have to be affine, we transform every such expression into a template where each index of a tensor is replaced with an affine hole. The templates are then de-duplicated. Similarly, to represent the set of constraints C, we make the observation that in a correct compilation, the indices in the right-hand side (the implementation expression) of the equalities generated by rule S-Assign must be deducible only from the indices in the left-hand side (the specification expression). Hence, we compute an expression of the former as a piecewise quasi-affine function of the latter, and only store the set of specification indices, thereby reducing the dimensionality of the set.

Once the toplevel set of constraints C has been computed, we generate Z3 queries to prove the corresponding equalities. The translation to Z3 is mostly straightforward. We first encode the Halide algorithm into a SARE, which are then encuded using the `define-funs-rec` facility [Barrett et al. 2010]. The well-formedness constraint on Halide algorithms ensures that these mutually recursive functions are well-defined, but it is not checked. Z3 has specialized support for such recursive functions which is typically more efficient than a direct encoding using quantifiers. Each constraint in C is a set of equalities which can then be expressed directly in Z3's logic. Note that thanks to the coalescing and simplifications performed by `isl`, there are typically fewer constraints to be verified than assignments in the program, because unrolled statements yield the same constraint. In addition, we observe that the domain of the formula is often, but not always, simple. It might be worth investigating whether the coalescing logic of `isl` can be improved to better coalesce the sets produced by our tool. In the outer product example of Section 2, the constraint is exactly $\forall 0 \leq i < N, 0 \leq j < M. A(i) \times B(j) = B(j) \times A(i)$.

Finally, we perform additional processing before sending the generated equalities to Z3, leveraging polyhedral checks on indices using `isl` in a best-effort strategy. Namely, before sending an equality $e_s = e_i$ where e_s comes from a prophetic annotation and e_i was inferred from the implementation:

- We unfold the tensors in e_s which do not appear in the transitive dependences of the tensors in e_i, excluding cycles.
- For each access a_s in e_s and a_i in e_i to the same tensor, if the indices are equal, we replace both accesses with the same `let`-bound variable to either a_s or a_i.

In most cases, after these simplifications, the equality contains the same accesses syntactically on both sides of the equality, helping Z3 focus on value-level reasoning rather than on resolving tensor indexings.

5.3 Benchmark Selection

Halide has a large benchmark suite in the `benchmarks/` subdirectory of its repository. Some are out of the scope of this paper due to containing non-affine specifications, including those with data-dependent accesses and histograms. Others use unsupported features, e.g. assigning an undefined value to simulate in-place input updates. We have run our tool on the remaining benchmarks, using the schedule provided in the original benchmark suite. The benchmarks have implicit assumptions on the required input sizes (e.g. that some dimensions are multiples of 16 or 32), implied by the scheduling directives used. They do not appear in the specification, but Halide generates runtime checks for them. These assumptions are given as contextual axioms to the verifier.

The benchmarks can be roughly separated into two application domains: linear algebra, including the convolution operators of deep learning, and image processing.

From the linear algebra domain, we consider the following benchmarks:

- `sdot` is a simple dot product manually implemented with a tiling factor of 8. This is expected to be easy to verify.
- `sgemm` is a general matrix-matrix multiply on floats, from the `linear_algebra` Halide application. It uses an optimized CPU schedule and specializations for small and large matrices. As a compute-intensive program, matrix-matrix multiply requires precise optimizations to get good performance: as such, this benchmark features heavy loop transformations and is a good stress test of our verifier as far as linear algebra benchmarks go. The `sgemmTA` and `sgemmTB` are variants where one of the input matrices is transposed. The Halide specification has a bug in these cases, and assumes that matrix A is square. As such, we expect the verification to fail.
- `cmm1024` is another matrix-matrix multiplication implementation, tuned for GPUs. This Halide benchmark is written for 1024×1024 square matrices, and we verify it for this concrete size only.
- `conv_layer` is a $2D$ convolution layer.
- `dsc` is a depthwise separable convolution. The specification contains non-affine indices, hence we use a constant grouping factor of 3, eliminating the non-affine component.

From the image processing domain, we consider the following benchmarks:

- `blur` is a two-dimensional blur filter, performing an average of three neighbors in each dimension. This example features storage folding: the schedule only stores four lines of the intermediate tensor at once in a rolling buffer.
- `sc` is a chain of large stencils of width 25 (i.e. each stage reads from 25 distinct neighbors from the previous stage). The default depth (i.e. number of stages) for the benchmark is 32, which is reported as `sc32`; we also include a variant `sc1` with a single stage.
- `harris`, `unsharp`, and `nl_means` are implementations of image processing algorithms, namely the Harris corner detector, unsharp masking, and non-local means.

In most cases, the Halide compiler applies algebraic transformations such as associativity and commutativity to the expressions in the specification. We represent signed and unsigned integer types using Z3's native representation based on bit-vectors. For floating-point numbers, we provide three possible encodings, discussed in Section 7.3. In most cases, we simply encode floats as real to capture the transformations which Halide makes under "fast-math" assumptions. For `harris`, `unsharp` and `nl_means`, Halide performs constant propagation in the floating point domain, which

we cannot validate when interpreting floats as reals. As such, we run Halide in "strict float" mode for these benchmarks, disabling floating point optimizations.

5.4 Evaluation

Table 1 shows that our translation validation system succeeds on 14 of the 17 examples, with running times from one second to 5 minutes for the successful examples. Two of the remaining examples are expected to fail due to a bug in the specification. The last example reaches a timeout of 15 minutes. To put these results in perspective, we ran the same examples through the ISA tool from Verdoolaege et al. [2012]. ISA is only able to verify 7 of the examples, and times out on 8 including one of the incorrect examples. Of the three remaining examples, one is correctly shown to be incorrect, while the others cannot be proven by ISA due to using simplifications beyond associativity and commutativity.

Unlike our approach, ISA is fully automatic, and does not rely on annotations. ISA is able to handle associative and commutative operators, but it is optional as it increases its runtime. We indicate with superscripts when ISA was allowed to use associativity (A) or commutativity (C) of an operator.

ISA takes C programs as inputs, which we generate from Sched by erasing the annotations. Halide can directly generate C code with linearized (non-affine) accesses and using vectorization primitives not supported by ISA. We also convert the specification to a C program, by creating a different loop nest for each assignment. A simple data-flow analysis is used to infer the bounds. Alternatively, we could compare the optimized Halide schedule with Halide's default schedule.

ISA takes the form of three command-line tools. `c2pdg` converts the C program into a polyhedral representation. We pass the option `--pet-signed-overflow=ignore` to `c2pdg`. `da` performs a data-flow analysis on the polyhedral representation. `eqv` performs the equivalence checking on the `da` output of two programs. We report the run-times of the different tools separately.

The results of our experimental evaluation are summarized in Table 1. The benchmarks have been run on a machine with an Intel® Core™ $i9-9900$ CPU, with a timeout of 15 minutes. For `isa`, the timeout applies separately to each step. For each benchmark, we indicate the run-time of each tool in seconds, as well as whether the equivalence was successfully proved. For isa, the `c2pdg` and `da` timings include the sum of times for both the implementation and specification.

When both our tool and ISA successfully validate the implementation, our tool has comparable or better performance, except on the `conv` benchmark. This is because for the `conv` benchmark, the affine conditions for the equalities sent to Z3 are quite complex. The conversion from the internal representation of `isl` to Z3 for sets with many disjuncts is a bottleneck of our approach, and in this specific case we suffer from a suboptimal implementation which does multiple conversion to and from the internal representation of `isl`. In addition, our tool is able to prove more cases than ISA. In the case of parametric matrix multiplications (`sqsgemm` and `sgemm`), ISA reached a timeout of 15 minutes. Parts of these kernels are specialized, with different implementations depending on the values of the parameters. The `bigsqsgemm` and `bigsgemm` entries present results when the size of the matrices are larger than 512: by allowing the pruning of some branches, this allows ISA to complete the verification. Note that this also drastically decreases runtime of our own algorithm.

Finally, let us mention the `nl_means` benchmark, which we fail to verify due to running out of time. Like in `conv`, Halide generates complex affine conditions involving many minimums and maximums: in consequence the `isl` representation of symbolic heaps contains many disjuncts, hurting the performance. In fact, our current implementation times out during the initial simplification of affine expressions. This could be mitigated by including more contextual information during the simplification: manual experiments on some expressions extracted from that benchmark indicate that it could result in up to an order of magnitude less disjuncts. Another avenue to explore would

Table 1. Results of the experimental evaluation (times in seconds)

	isa				Ours	
	c2pdg	da	eqv			
blur	<1	<1	1.1^{AC+}	✓	<1	✓
cmm1024	<1	1.2	10	✓	2.6	✓
sgemm1024	<1	2.5	$27.3^{C\times}$	✓	1	✓
sqsgemm	2.5	4min34	$> 15\text{min}^{C\times}$	?	15.1	✓
bigsqsgemm	1.6	17.5	$8\text{min}12^{C\times}$	✓	2.7	✓
sgemm	7.	>15min	N/A	?	3min24	✓
bigsgemm	2.8	1min19	>15min	?	12.1	✓
sc1	3.44	4m20	3.3	✗	12.5	✓
sc32	1min41	>15min	N/A	?	4min53	✓
dsc	10.2	13min49	>15min	?	1min43	✓
conv	2.	12.2	$27.9^{C\text{max}}$	✓	2min12	✓
sdot	<1	<1	<1	✓	<1	✓
harris*	7.9	31.5	1min8	✓	44.8	✓
unsharp*	1.3	6.1	13min44	✗	6.1	✓
nl_means*	>15min	N/A	N/A	?	>15min	?
sgemmTA†	21.5	>15min	N/A	?	10.4	✗
sgemmTB†	21.1	N/A	N/A	✗	34.7	✗

*No floating-point optimizations

†Expected to fail

be to find independent piecewise expressions that can be abstracted and factored out to reduce the number of disjuncts.

6 RELATED WORK

Translation validation. Pnueli et al. [1998] introduced translation validation for the verification of compilers for languages with no loops. Approaches using symbolic evaluation combined with specialized equivalence checking procedures are able to handle loop optimizations that mostly preserve the execution order of instructions. For instance, the work of Necula [2000] supports loop unrolling in the GCC compiler; other examples include software pipelining by Tristan and Leroy [2010], loop-invariant code motion by Tristan et al. [2011], and loop-peeling and induction variable strength reduction by Tate et al. [2011]. On the other hand, loop transformations such as permutation, fusion or tiling fundamentally change the structure of the program. They require more advanced validation techniques. The literature focuses on ad-hoc rules such as the permutation rule of Barrett et al. [2005] and the similar rules of Kundu et al. [2009]. These approaches require the compiler to trace the transformations that have been performed, an information that is not easy to extract from schedule-based compilers [Bagnères et al. 2016; Zinenko et al. 2018].

Specification of tensor and arrays optimizations. The previous approaches prove optimizations on intermediate representations and their transformations. Unfortunately, most tensor compilers do not provide a formal semantics or type system to reason about, or for that matter to prove their correctness w.r.t. some functional specification. TeIL is one significant effort in this direction [Rink and Castrillon 2019], but its semantics based on combinators is not at the appropriate abstraction

level to easily express the iterator-based specifications of most tensor programming languages. We rely on a simple equational language to capture the semantics of these specifications, while demonstrating the translation validation of a tensor compiler independently of the intermediate representations encountered along the flow.

Equivalence checking of affine programs. In the case of numerical programs with array manipulations, the problem of translation validation is known as an instance of program equivalence checking. Samsom et al. [1995] proposes an approach based on pattern-matching of the right-hand side of assignments on the original and optimized programs, then proves that the loop nests for each pair of matched assignments iterate over the same domains. This approach is restricted to code without recurrences, and the pattern-matching rules can only handle the most basic algebraic transformations. Shashidhar et al. [2005] introduces Array Data-flow Dependence Graphs (ADDG) to overcome the limitations of the original pattern-matching approach, and is able to handle finite applications of associative and commutative operators. The ADDGs allow matching to occur at the level of operator applications instead of statements, and it supports uniform recurrences by computing a transitive closure of the updates. Karfa et al. [2013b] extend the ADDG method to handle more algebraic transformations using a formalization procedure, but does not handle recurrences. Banerjee et al. [2016] proposed handling recurrences by matching the bodies of loop, effectively preventing structure-modifying transformations to be applied to recurrences.

Verdoolaege et al. [2012, 2010] extend the expressive power of the ADDG based methods considerably, by providing a generic method to handle possibly non-uniform recurrences using widening. Their approach is based on two passes. In the first pass, equalities between the two programs are inferred based on the requested output equalities. Widening is used to propagate equalities across loops. In a second pass, the verifier actually tries to prove that the inferred equalities are correct. Their approach is probably the closest to ours, although our verifier only performs the equivalent of their second pass, as we assume the equalities are provided by the compiler. Their approach is able to handle fixed-arity associative and commutative operators, but no other algebraic transformations.

Approaching the problem from a different angle, Barthou et al. [2002] tries to prove the equivalence of Systems of Affine Recurrence Equations. They show the problem to be undecidable in general, even without any algebraic transformations, and propose a best-effort algorithm for proving equivalence by constructing an equivalence automaton. Their approach does not handle algebraic transformations. Iooss et al. [2014] improves upon their algorithm to handle associative-commutative reductions, using a bipartite matching between the reduced indices in the original and transformed program, and is the only work we know of which is able to handle associative-commutative reductions in a generic way.

Karfa et al. [2013a] directly encode the program equivalence problem as a formula which is fed to an SMT solver. They show that the formulas this creates are too complex for SMT solvers to handle in practice. We avoid the issue by using prophetic expressions as natural stopgaps to generate multiple, simpler queries to the SMT solver.

Bao et al. [2016] propose a dynamic approach, dubbed PolyCheck, to the problem. It exploits the structure of affine program control and data-flow to build a checker with the same structure as the transformed program. If successful, it ensures the validity of all executions for a given problem size.

Journault and Miné [2018] propose abstract domains to represent and infer properties about matrix manipulating program. They successfully apply their approach in presence of loop tiling, as performed by the Pluto polyhedral compiler. Unlike ours, their approach does not rely on annotations but relies instead on a library of patterns to match assignments with a corresponding

semantic predicate. It is not clear how well this library of patterns would scale to arbitrary code transformations.

7 DISCUSSION AND FUTURE WORK

In this section, we discuss limitations of the current approach and planned extensions towards the translation-validation of a larger portion of the Halide compiler.

7.1 Array Linearization

The focus of our presentation has been on multidimensional tensors and arrays, as it is the way specifications are written. Ultimately, multidimensional arrays are represented as flat buffers in linear memory, which can create non-affine indices due to parametric array sizes. Reconstructing affine multidimensional indices from the linearized ones is possible, but difficult and typically involves runtime checks [Doerfert et al. 2017]. However, in our case, we know the multidimensional indices and simply have to check that the same injective linearization function is used for all accesses. This is a fairly simple non-affine check, which could be expressed as a piecewise polynomial equality. We experimented with simply sending the non-linear problem to Z3, which was able to prove it correct without issue in all our benchmarks.

7.2 Overflow Checking

We have formalized and implemented our validator using unbounded integers, and simply ignored the possibility of overflows in index computations. There are two general approaches to handle this threat to the formal soundness of our results when running the code using machine integers. The first possibility is to find a sufficient condition on the parameters to ensure the absence of overflow, which can be independently checked, following the approach of Cuervo Parrino et al. [2012]. The second possibility is to explicitely model overflowing computations (assuming some fixed well-defined behavior of signed integer overflow) using a piecewise representation or a modulo expression, as appropriate. This would increase the runtime of the algorithm by introducing disjunctions or auxiliary variables in the `isl` representation. ISA supports both approach (depending on whether signed or unsigned indices are used); in our experiments, we use signed indices but disable the overflow checking for a fair comparison.

To cover verification in full, overflow checking needs to take into account the linearization step from multidimensional indices to linear memory. If the linearization is computed inline, it is enough to ensure that the total array size fits in the appropriate integer type because we have proven that the multidimensional indices are within bounds.

7.3 Handling of Floating-Point Numbers

In Halide, and other languages targeting linear algebra, image processing, or deep learning, computations often involve floating-point numbers. These computations are transformed under "fast-math" assumptions, which are formally unsound. There are multiple possible answers to that problem, and we discuss three approaches supported by our implementation below.

- If bitwise equivalence is wanted, floating point numbers can be implemented using Z3's built-in support for IEEE floats. In this case, "fast-math" optimizations should be disabled; Halide supports both a global "strict float" mode and a local annotation to disable such optimizations within a given expression. Although we do not support these, some "fast-math" transformations are correct in the absence of NaNs, infinites, or negative zeros and could be supported with the appropriate assumptions on the input arrays [Menendez et al. 2016].

- On the other side of the spectrum, common practice in numerical fields is to accept program transformations which are valid on the reals, e.g. exploiting the associativity and distributivity of operators. Following this practice, our default representation of floats uses Z3's built-in type for reals. While this provides no formal guarantee on the output of the program when ran using floats, it ensures that only transformations valid on the reals have been applied. This can be enough to trust the absence of bugs in a non-adversarial compiler. Note that while this is mostly appropriate for the algebraic specifications encountered in linear algebra or deep learning, representing floats as reals is an issue for image processing pipelines. Image processing algorithms often provide opportunities for constant propagations: this propagation is performed by Halide using floating-point math, and Z3 will usually not be able to prove it correct in the reals (because it is not). Constant propagation could be verified in this context if Halide used exact rational math instead.
- Finally, we can implement floating-point numbers axiomatically using an abstract type in Z3. Using specific axioms, generic "fast-math" rules such as associativity can be supported, and also constant propagation on a case-by-case basis. The same caveats as when implementing floats using reals apply: if enough axioms are included, an adversarial compiler could build a program that is correct in this axiomatization but compute all zeroes when ran using floats.

One final remark is that if we were to allow casts from floats to indices when lifting the affine restrictions (see Section 7.4), special care would be required because "fast-math" transformations could lead the cast to result in a different index, possibly creating out-of-bounds accesses.

7.4 Beyond Affine Specifications and Schedules

In our work we have made the assumption that schedules for affine specifications result in affine implementations. While true in general, an important counter-example is that of specialization on an input value. For instance in a general matrix multiply $D = \alpha C + \beta AB$, implementations often have a "fast path" specialization when $\alpha = 0$, avoiding loads from C entirely. Handling such specializations would require keeping track of non-affine conditionals in path conditions, and giving them as additional context to Z3.

This would require ensuring that both branches of a non-affine `if` write (possibly different values) to the same locations, in order to be able to compute the last write to a location independently of the non-affine condition in a `lexmax` computation. This is typically the case for a specialization. Note that handling this would be more general than the approach of Verdoolaege et al. [2010], which only handles non-affine control that is present in both the specification and the implementation. That case is already handled in our approach through non-affine `select` operators — `select` being an expression-level conditional ensures that the locations written to are not dependent on the non-affine condition.

Another non-affine extension that can be integrated fairly easily is that of non-affine *read* accesses. When encountering a read access, we currently solve exactly for the expression present in the corresponding chunk using `isl`. If the access is non-affine, this is no longer possible. However, we can build an expression as a `select`-tree of affine conditions depending on the expressions present in the chunk, and apply that expression to the non-affine indices. We also have to check using Z3 (instead of `isl`) that the non-affine indices are within the chunk's domain.

7.5 Reductions

Associative and commutative reductions show up in many applications, the most common of them being tensor contractions and convolutions. Properly accounting for the algebraic properties of those reductions is crucial to be able to verify transformations such as those performed by Halide's

`rfactor` [Suriana et al. 2017] primitive, which permits reordering the reduction to increase the available parallelism or improve memory locality.

The difficulty here is twofold. The first difficulty is in identifying the associative-commutative operators, which is not immediate from Halide's specification of them as recurrences. When performing transformations on a reduction, Halide knows the operator it uses, and should be able to provide an annotation to that regard without difficulties. We would then have to check that the operator indeed is associative and commutative using Z3. The second one is in properly representing the now arbitrary order of the computations.

The proper tool for representing objects in an arbitrary order is that of a set, and we have started work on an extension of our system which represents a reduction on an array using an extra virtual dimension to represent the reduced values. More precisely, we need to identify sections during which the reduction is computed: this corresponds to a single stage of the computation for Halide, and can be annotated. Within that section, we represent that an array a of type τ_a implements a n-dimensional reduction using a special reduction type $\Sigma_n \tau_a$. The type $\Sigma_n \tau_a$ extends τ_a in the dynamic semantics with n dimensions, capturing the positions of the updates in the original recurrence. At the end of the reduction, we must check that the set of written values matches the specification.

This would essentially require Halide to annotate the reduction steps with the reordering it has performed, alleviating the need for a matching algorithm such as that of Iooss et al. [2014].

7.6 Mechanized Formal Proofs

As we are performing validation of a complex system, it could be interesting to mechanize our proof and provide an implementation in a proof assistant such as Coq [The Coq Development Team 2021]. Courant and Leroy [2021] recently mechanized the proof of a polyhedral code generator. Inspired by this achievement, we did mechanize an early version of the prophetic evaluation rules using Coq, from which the current implementation now deviates. We aim to have an automatic verifier, and rely on Z3 and `isl` for this. As such, mechanization could only guarantee that the verification conditions we generate ensure the correctness of the implementation.

8 CONCLUSIONS

This paper proposes a translation-validation algorithm for compilers of equational tensor specifications, and applies it to the verification of benchmarks from the Halide DSL.

Although specified and implemented in the context of the Halide compiler, our approach naturally generalizes to pointful tensor programming language. On the high-level side, by focusing on an affine subset through the reduction to SAREs, we decouple our verifier from the details of the specification language. On the low-level side, our verifier and formalization require an imperative language of counted loops and arrays, annotated with refinements from the array writes to the corresponding tensor definitions. This is a natural representation for the output of a pointful tensor compiler: in fact, frameworks such as TVM, Tensor Comprehensions, and Tiramisu use either Halide's own IR or variants thereof.

The approach has been tested on reference Halide benchmarks, and shown to be a viable candidate for the parametric verification of important primitives such as matrix multiplication. High-level libraries based on graphs of operators such as TensorFlow could benefit from our approach to validate the specialized implementations of those primitives.

9 DATA AVAILABILITY STATEMENT

The source code implementing the approach described in Section 4 and instructions for reproducing the experiments of Section 5 are archived online [Clement and Cohen 2022].

REFERENCES

Martin Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*. 165–175. https://www.microsoft.com/en-us/research/publication/the-existence-of-refinement-mappings/

Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening Polyhedral Compiler's Black Box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, Björn Franke, Youfeng Wu, and Fabrice Rastello (Eds.). ACM, 128–138. https://doi.org/10.1145/2854038.2854048

Kunal Banerjee, Chittaranjan Mandal, and Dipankar Sarkar. 2016. Translation Validation of Loop and Arithmetic Transformations in the Presence of Recurrences. *SIGPLAN Not.* 51, 5 (Aug. 2016), 31–40. https://doi.org/10.1145/2980930.2907954

Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. PolyCheck: Dynamic Verification of Iteration Space Transformations on Affine Programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 539–554. https://doi.org/10.1145/2837614.2837656

Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening (Eds.).

Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore D. Zuck. 2005. TVOC: A Translation Validator for Optimizing Compilers. In *Computer Aided Verification, 17th International Conference, CAV 2005 (Lncs, Vol. 3576)*. springer, 291–295.

Denis Barthou, Paul Feautrier, and Xavier Redon. 2002. On the Equivalence of Two Systems of Affine Recurrence Equations. In *Euro-Par 2002 Parallel Processing*, Burkhard Monien, Rainer Feldmann, Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen (Eds.). Vol. 2400. Springer Berlin Heidelberg, Berlin, Heidelberg, 309–313. https://doi.org/10.1007/3-540-45706-2_40

G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. 2005. Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models. *Proc. IEEE* 93, 2 (Feb. 2005), 276–292. https://doi.org/10.1109/jproc.2004.840311

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, 53–64.

Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008a. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC)*.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008b. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Basile Clement and Albert Cohen. 2022. End-to-End Translation Validation for the Halide Language - OOPSLA'22 Artifact. Zenodo. https://doi.org/10.5281/zenodo.6390003

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

Bruno Cuervo Parrino, Julien Narboux, Eric Violard, and Nicolas Magaud. 2012. Dealing with Arithmetic Overflows in the Polyhedral Model. In *IMPACT 2012 - 2nd International Workshop on Polyhedral Compilation Techniques*, Uday Bondhugula and Vincent Loechner (Eds.). Louis-Noel Pouchet, Paris, France. https://hal.inria.fr/hal-00655485

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C.R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Johannes Doerfert, Tobias Grosser, and Sebastian Hack. 2017. Optimistic Loop Optimization. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 292–304. http://dl.acm.org/citation.cfm?id=3049864

P. Feautrier. 1988. Parametric Integer Programming. *RAIRO Recherche Opérationnelle* 22, 3 (1988), 243–268.

Paul Feautrier. 1991. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming* 20, 1 (Feb. 1991), 23–53. https://doi.org/10.1007/bf01407931

Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More than Scanning Polyhedra. *ACM Transactions on Programming Languages and Systems* 37, 4, Article 12 (July 2015). https://doi.org/10.1145/2743016

Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. 2014. On Program Equivalence with Reductions. In *Static Analysis*, Markus Müller-Olm and Helmut Seidl (Eds.). Vol. 8723. Springer International Publishing, Cham, 168–183. https://doi.org/10.1007/978-3-319-10936-7_11

Matthieu Journault and Antoine Miné. 2018. Inferring Functional Properties of Matrix Manipulating Programs by Abstract Interpretation. *Formal methods in system design* 53, 2 (Feb. 2018), 221–258. https://doi.org/10.1007/s10703-017-0311-x

Chandan Karfa, K. Banerjee, D. Sarkar, and C. Mandal. 2013a. Experimentation with SMT Solvers and Theorem Provers for Verification of Loop and Arithmetic Transformations. In *Proceedings of the 5th IBM Collaborative Academia Research Exchange Workshop on - i-Care '13*. ACM Press. https://doi.org/10.1145/2528228.2528231

Chandan Karfa, Kunal Banerjee, Dipankar Sarkar, and Chittaranjan Mandal. 2013b. Verification of Loop and Arithmetic Transformations of Array-Intensive Behaviors. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 32, 11 (Nov. 2013), 1787–1800. https://doi.org/10.1109/tcad.2013.2272536

Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. 1967. The Organization of Computations for Uniform Recurrence Equations. *Journal of The Acm* 14, 3 (July 1967), 563–590. https://doi.org/10.1145/321406.321418

Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. *ACM SIGPLAN Notices* 44, 6 (May 2009), 327–337. https://doi.org/10.1145/1543135.1542513

Hervé Le Verge, Christophe Mauras, and Patrice Quinton. 1991. The ALPHA Language and Its Use for the Design of Systolic Arrays. *Journal of VLSI signal processing systems for signal, image and video technology* 3 (1991), 173–182. https://doi.org/10.1007/BF00925828

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Static Analysis (Lecture Notes in Computer Science)*, Xavier Rival (Ed.). Springer, Berlin, Heidelberg, 317–337. https://doi.org/10.1007/978-3-662-53413-7_16

Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A Hardware–Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro* 39, 5 (Sept. 2019), 8–16. https://doi.org/10.1109/mm.2019.2928962

Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July 2016). https://doi.org/10.1145/2897824.2925952

Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGPLAN Not.* 50, 4 (March 2015), 429–443. https://doi.org/10.1145/2775054.2694364

George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation - PLDI '00*. ACM Press, 83–95. https://doi.org/10.1145/349299.349314

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98 (Lncs, Vol. 1384)*. springer, 151–166.

M. Presburger. 1929. Über Die Vollständigkeit Eines Gewissen Systems Der Arithmetik Ganzer Zahlen, in Welchem Die Addition Als Einzige Operation Hervortritt. In *Comptes Rendus Du Premier Congrès de Mathématiciens Des Pays Slaves*. Warsaw, Poland, 92–101.

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4 (Aug. 2012), 1–12. https://doi.org/10.1145/2185520.2185528

Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Communications of The Acm* 61, 1 (Dec. 2017), 106–115. https://doi.org/10.1145/3150211

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

Norman A. Rink and Jeronimo Castrillon. 2019. TeIL: A Type-Safe Imperative Tensor Intermediate Language. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY 2019)*. Association for Computing Machinery, New York, NY, USA, 57–68. https://doi.org/10.1145/3315454.3329959

H. Samsom, F. Franssen, F. Catthoor, and H. De Man. 1995. System Level Verification of Video and Image Processing Specifications. In *Proceedings of the 8th International Symposium on System Synthesis - ISSS 95*. ACM Press. https://doi.org/10.1145/224486.224533

K.C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. 2005. Verification of Source Code Transformations by Program Equivalence Checking. In *Compiler Construction (Lecture Notes in Computer Science)*, Rastislav Bodik (Ed.). Springer, Berlin, Heidelberg, 221–236. https://doi.org/10.1007/978-3-540-31985-6_15

Patricia Suriana, Andrew Adams, and Shoaib Kamil. 2017. Parallel Associative Reductions in Halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Austin, USA, 281–291.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2011. Equality Saturation: A New Approach to Optimization. *Logical Methods in Computer Science* 7, 1 (March 2011). https://doi.org/10.2168/lmcs-7(1:10)2011

The Coq Development Team. 2021. The Coq Proof Assistant. Zenodo. https://doi.org/10.5281/zenodo.4501022

Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-Graph Translation Validation for LLVM. *ACM SIGPLAN Notices* 46, 6 (June 2011), 295–305. https://doi.org/10.1145/1993316.1993533

Jean-Baptiste Tristan and Xavier Leroy. 2010. A Simple, Verified Validator for Software Pipelining. *ACM SIGPLAN Notices* 45, 1 (Jan. 2010), 83–92. https://doi.org/10.1145/1707801.1706311

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-agnostic High-Performance Machine Learning Abstractions. arXiv:1802.04730 [cs.PL]

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2020. The next 700 Accelerated Layers. From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Transactions on Architecture and Code Optimization* 16, 4 (Jan. 2020), 1–26. https://doi.org/10.1145/3355606 arXiv:1802.04730

Sven Verdoolaege. 2010. Isl: An Integer Set Library for the Polyhedral Model.. In *ICMS (Lecture Notes in Computer Science, Vol. 6327)*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer, 299–302. http://dblp.uni-trier.de/db/conf/icms/icms2010.html#Verdoolaege10

Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences. *ACM Transactions on Programming Languages and Systems* 34, 3, Article 11 (Nov. 2012). https://doi.org/10.1145/2362389.2362390

Sven Verdoolaege, Martin Palkovič, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. 2010. Experience with Widening Based Equivalence Checking in Realistic Multimedia Systems. *Journal of Electronic Testing-theory and Applications* 26, 2 (Jan. 2010), 279–292. https://doi.org/10.1007/s10836-009-5140-4

Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2018. Visual Program Manipulation in the Polyhedral Model. *ACM Transactions on Architecture and Code Optimization* 15, 1 (2018), 16:1–16:25. https://doi.org/10.1145/3177961