Getting to the Point

Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming

ADAM PASZKE, Google Research, Poland
DANIEL D. JOHNSON, Google Research, Canada
DAVID DUVENAUD, University of Toronto, Canada
DIMITRIOS VYTINIOTIS, DeepMind, United Kingdom
ALEXEY RADUL, Google Research, USA
MATTHEW J. JOHNSON, Google Research, USA
JONATHAN RAGAN-KELLEY, Massachusetts Institute of Technology, USA
DOUGAL MACLAURIN, Google Research, USA

We present a novel programming language design that attempts to combine the clarity and safety of high-level functional languages with the efficiency and parallelism of low-level numerical languages. We treat arrays as eagerly-memoized functions on typed index sets, allowing abstract function manipulations, such as currying, to work on arrays. In contrast to composing primitive bulk-array operations, we argue for an explicit nested indexing style that mirrors application of functions to arguments. We also introduce a fine-grained typed effects system which affords concise and automatically-parallelized in-place updates. Specifically, an associative accumulation effect allows reverse-mode automatic differentiation of in-place updates in a way that preserves parallelism. Empirically, we benchmark against the Futhark array programming language, and demonstrate that aggressive inlining and type-driven compilation allows array programs to be written in an expressive, "pointful" style with little performance penalty.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Parallel programming languages**; • **Mathematics of computing** → **Automatic differentiation**.

Additional Key Words and Phrases: Array programming, automatic differentiation, parallel computing

ACM Reference Format:
Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *Proc. ACM Program. Lang.* 5, ICFP, Article 88 (August 2021), 29 pages. https://doi.org/10.1145/3473593

1 INTRODUCTION

Recent years have seen a dramatic rise in the popularity of the array programming model. The model was introduced in APL [Iverson 1962], widely popularized by MATLAB, and eventually made its way into Python. There are now tens of libraries centered around n-dimensional arrays (nd-arrays), each with its own unique strengths. In fact, this model is now so important that many

Authors' addresses: Adam Paszke, apaszke@google.com, Google Research, Poland; Daniel D. Johnson, ddjohnson@google.com, Google Research, Canada; David Duvenaud, duvenaud@cs.toronto.edu, University of Toronto, Canada; Dimitrios Vytiniotis, dvytin@google.com, DeepMind, United Kingdom; Alexey Radul, axch@google.com, Google Research, USA; Matthew J. Johnson, mattjj@google.com, Google Research, USA; Jonathan Ragan-Kelley, jrk@mit.edu, Massachusetts Institute of Technology, USA; Dougal Maclaurin, dougalm@google.com, Google Research, USA.

2475-1421/2021/8-ART88
https://doi.org/10.1145/3473593

hardware vendors often market their products by evaluating various nd-array workloads, and publish their own libraries of array kernels (e.g., NVIDIA's cuBLAS) or compilers for array programs (e.g., Google's XLA). We theorize that the nd-array model's recent success is due to a convergence of three factors: its reasonable *expressiveness* for concise, high-level specification of (relatively) large workloads; the *abundant parallelism* available within the individual nd-array operations; and the *preservation of parallelism under automatic differentiation.*

Preserving parallelism under automatic differentiation is a particularly important advantage. The trouble is that the most popular automatic differentiation algorithm (the so-called reverse-mode) inverts all data dependencies in the original program. This unfortunately turns perfectly parallel loops that read a datum repeatedly into sequential loops that have to write to the cotangent for that datum many times. Most modern nd-array libraries circumvent this problem by operating at a higher level of abstraction: instead of differentiating through the individual array accesses used to implement each operation (e.g., scalar multiply-adds in matrix multiplication), the high-level operations themselves are replaced with new high-level operations that compute the appropriate cotangents in bulk.[1] The result is that the reverse-AD transform for a typical nd-array program produces another typical nd-array program, namely a sequential composition of parallel blocks.

Forcing the original program to be written at a higher level of abstraction, however, has significant downsides. The first major flaw, in our view, is *insufficient expressiveness*: while composing a program out of array operations leads to wonderfully fast and concise code when the available array operations cover one's needs, one faces a steep complexity and performance cliff—dropping down to manually writing, optimizing, and differentiating new operations—as soon as they do not. Examples of algorithms that are difficult to express efficiently in this style include differential equation solvers, natural language processing algorithms, weather and physics simulations, and workloads that need non-linear stencil operations. In these domains, inner loops are still regularly written in loosely-typed, low-level Fortran or CUDA.

The model's second major flaw is *insufficient clarity*: for their code size, nd-array programs are notoriously difficult to read, reason about, and get right. Practitioners constantly complain about "shape errors," which are really mis-specification of the intended flow of data. Indeed, when an entire multidimensional array is referred to by a single variable name, its internal semantics (for instance, which dimensions are indexing what) are missing from the program—and have to be continuously reconstructed by the programmer, which is both tedious and error-prone. Logic errors are especially costly in two of the classic applications of array programming—stochastic simulation and learning algorithms—because tests are less effective in these domains.

This paper introduces Dex, a new array programming language oriented around safe, efficient, ergonomic, and differentiable typed indexing. The contributions are as follows:

- Typed indexing constitutes a novel, concise yet flexible notation for expressing array programs (Section 2.1).
- Typing array index sets creates an analogy between arrays and functions, which in turn leads to a fruitful convergence of array and functional programming (Section 3.3).
- We introduce an effect for associative accumulation in Dex's effect system. The `Accum` effect hits a sweet spot of expressiveness and parallelizability (Section 2.3).
- We improve upon the defunctionalization methods presented in Hovgaard et al. [2018], in a way that allows functions to be treated as first-class objects and can deal with arrays of functions or returning functions from conditionals (Section 4).

[1] Such transformation rules can either be written by hand for each operation [Bradbury et al. 2018], or derived using special-purpose techniques [Hückelheim et al. 2019].

- We implement efficient, parallelizable automatic differentiation with full coverage of Dex, including indexing (Section 5).
- We articulate several lessons we learned from treating AD as a first-class concern while designing the Dex language (Section 5.3).
- We describe our strategy for achieving competitive performance (Section 6). The main ingredients are:
  - Work-efficient automatic parallelization (preserved through AD) via the `Accum` effect (Section 6.1).
  - Straightforward loop fusion via inlining `for` expressions (Section 6.2).
  - Index-set-type directed compilation to elide bounds checks and make layout decisions (Section 6.3).

2 DEX BY EXAMPLE

Dex is a strict functional language for index-oriented[2] array programming defined by a confluence of several features:

- Typed array dimensions and indices (Section 2.1),
- A parallelism-friendly effect system (Section 2.3), and
- Work- and parallelism-preserving automatic differentiation (Section 5).

We start with building intuition and giving examples, especially in Section 2.2. A formal description of the language can be found in Section 3.

2.1 Arrays with Typed Indexing

Dex defines two syntactic forms for manipulating arrays. The first form is used to construct arrays:

```
array = for i:n. expr
```

The `for` expression is similar to the `build` function of $\tilde{F}$ [Shaikhha et al. 2019]. The result of such a `for` expression is an array constructed by repeatedly evaluating the body, `expr`, with `i` bound to each consecutive member of the *index set* `n`. The array will thus have as many elements as `n` has members. As we will see in the next few examples, `n` can usually be omitted, because Dex infers it from the body.

The second form is array indexing, written using the `(.)` operator:

```
element = array.j
```

The index `j` must be a member of the array's index set `n`, and `array.j` extracts the `j`th element. These two operations are the fundamental building blocks of all Dex programs.

The magic sauce is the type of arrays. We type an array as

```
array : [IndexSet n] n=>a,
```

which exposes both the index set `n` and the element type `a` to the type system.[3] Array elements can be any type, including functions and other arrays, but array indices have to obey an `IndexSet` class constraint. More on this in Section 3.4.

As our first example, consider transposing a matrix `matrix`:

```
transposed = for i j. matrix.j.i
```

This expression loops over two indices (one for each dimension of `matrix`) in the order `i` then `j`, but then applies them in reverse. The same pattern can be easily adapted to reorder any number of

[2] "Pointful", as opposed to point-free.

[3] The type system becomes value-dependent to accommodate index sets; more in Section 3.2.

```
def update (c:Complex) (z:Complex) : Complex = c + (z * z)
def inBounds (z:Complex) : Bool = complex_abs z < 2.0
def escapeTime (c:Complex) : Int =
  fst (yieldState (0, zero) \(n, z).
    for i:(Fin 1000).
      z := update c (get z)
      n := (get n) + (BToF (inBounds (get z))))

xs = linspace (Fin 300) (-2.0) 1.0
ys = linspace (Fin 200) (-1.0) 1.0
mandelbrot = for j i. escapeTime (MkComplex xs.i ys.j)
```

Fig. 1. Computing the Mandelbrot set in Dex.

dimensions of an n-dimensional array. Notice that we didn't have to give iteration bounds in the `for` expression—Dex's type system infers them from the type of `matrix`.

We can just as easily multiply two matrices `x` and `y`.

```
x_times_y = for i j. sum (for k. x.i.k * y.k.j)
```

In this example, all the `for`-bound variables are again used in array indexing, meaning that their ranges can be inferred from the types of `x` and `y`. Explicit indexing makes it straightforward to work out the semantics of this, even if one is not already familiar with it. Reading the expression inside out:

(1) We first construct element-wise products of vectors selected from the first dimension of the first input (rows) and second dimension of the second input (columns): `for k. x.i.k * y.k.j`.
(2) Then, we take the sum of these products by calling the function `sum` (which we return to in Section 2.3).
(3) This computation is repeated for all valid indices `i` and `j` (`for i j.`). Since the result of `sum` is a scalar, the result `x_times_y` of the repeated computation is a two-dimensional array (i.e., a matrix).

Note that we used the same index, `k`, for the second dimension of `x` as for the first dimension of `y`. Dex's type system will therefore statically check those array dimensions for equality. Typed index sets make a whole class of shape errors easier to detect and repair in Dex.

2.2 Complete Example

Other than typed indexing and the effect system we discuss in Section 2.3, the core of Dex looks like a strict functional programming language in the ML [Milner et al. 1997] syntactic family. Figure 1 shows a complete example Dex program for computing the Mandelbrot set. A few points to pay attention to when parsing this:

- The colon : is Dex for "has type", and introduces the mostly-optional type annotations for variable bindings and function return values.
- The back-slash \ is a lambda function expression: `\n z. body` is a function that accepts arguments `n` and `z` (in this case, references in the `State` effect).
- Note the syntactic overloading of the dot (`.`). It serves as a decimal point, as the array indexing operator, and as the delimiter between the binders and body of `for` and \ expressions.
- The `(Fin 1000)` is an index set representing 1000 elements. We will cover `Fin` when we discuss index sets in Section 3.4; in this program, it indicates that our loop should iterate 1000 times, and specifies the number of elements in the grid returned by `linspace`.

The `yieldState` function is a convenience wrapper which returns the final state after running a stateful action (using the primitive `runState`). We now proceed to a more in-depth discussion of Dex's effect system.

2.3 Effects

Pointwise transforms (maps)—the fundamental building block of array programs—are easy to write with Dex's `for` loops. But they are far from being sufficient for an expressive modern array language. Even the simple matrix multiplication example required a `sum` function, which cannot be written as a pure `for` expression because it must combine many different elements of the input array while iterating across the index set.

We thus need a means for controlled communication across different iterations of a `for`. To that end, we extend Dex with an *effect system*. The type of every expression is extended with the set of effects its evaluation will induce. We give the flavor of Dex's effects by discussing two different ways to spell `sum`.

2.3.1 Summing as State. One of the supported effects is `State`, allowing arbitrary reads and updates to a shared state, which is sufficient to make `sum` expressible in Dex:

```
sum = \x:n=>Float.
  (_, total) = runState 0.0 \ref.
    for i.
      ref := (get ref) + x.i
  total
```

Let us dissect this snippet line-by-line. We begin with a lambda definition that binds the vector to be reduced to the variable `x`, whose type we annotate as `n=>Float`. Here `n` denotes an arbitrary index set, since we want `sum` to sum over arrays of any size. Next, we apply the `runState` combinator, which allows us to locally execute a stateful action within the pure function `sum`. `runState` takes the initial value of type `s` for the state along with a stateful function, and applies the function once to an appropriately-initialized *mutable reference* of type `Ref h s`.[4]

```
runState : [Data s] s -> (h:Type ?-> Ref h s -> {State h} a) -> (a, s)
```

Once we have the reference `ref` in scope, we begin looping over indices that are valid for the vector `x`. At each step we retrieve the current value of the state using the `get` function, add to it the `i`-th value in `x`, and update the state with the result.

```
get  : Ref h a -> {State h} a
(:=) : Ref h a -> a -> {State h} Unit
```

Finally, `runState` returns a pair containing the result of its body and the value of the reference once the body has been evaluated. Since the body does not have any meaningful results (it is of type `n=>Unit`—an array full of unit values), we skip the first component and return the sum.

The mysterious `h` parameter appearing in the `Ref` type implements the classic rank-2 polymorphism trick used for Haskell's ST monad [Launchbury and Peyton Jones 1994]. The user function given to `runState` must accept an argument of type `Ref h s` for an arbitrary local type parameter `h`, which guarantees that the reference cannot escape the block in which it is valid. The `?->` function arrow in Dex indicates that `h` is an *implicit argument* which can be inferred from context and thus does not have to be written out by the user.

2.3.2 Summing as Accumulation. Unfortunately, expressing `sum` with a `State` effect as above is a mixed bag. While it does compute the correct sum, it sacrifices parallelism in the process. Whenever the body of a `for` expression induces a `State` effect, it is possible that each loop iteration can have

[4] The state has to be something we can allocate a mutable buffer for, which we model with the `Data` class constraint.

arbitrary dependencies on previous iterations. A sufficiently powerful analysis might let us discover some subset of stateful loops that are parallelizable, but such an analysis is difficult, and in the absence of such an an analysis we are forced to pessimistically evaluate the loop sequentially.

Instead of that, Dex includes a more specific *accumulation* effect, which we call Accum. The accumulation effect is similar to State in that it also exposes mutable references, but Accum imposes two critical restrictions on them. First, updates in Accum must be *additive contributions* to references with (finite-dimensional) *vector space* types:

```
(+=) : [VectorSpace w] Ref h w -> w -> {Accum h} Unit
```

Here VectorSpace w denotes a constraint on the (otherwise polymorphic) type w: it must be an array n=>b for some vector space b, a pair (b, c) of vector spaces b and c, or a Float. Note that in every vector space addition forms a monoid (is associative and has a neutral "zero" element).

Second, there is no "read" operation in Accum—the value of the accumulator cannot be retrieved until the reference goes out of scope at the end of the runAccum handler (and thus until all writes have completed; the reference is initialized to the zero element of the vector space):

```
runAccum : [VectorSpace w] (h:Type ?-> Ref h w -> {Accum h} a) -> (a, w)
```

Together, these constraints mean that all updates to a reference in Accum are associative (up to floating-point roundoff), and therefore functions using the Accum effect can be efficiently parallelized. Specifically, we can partition any sequence of updates into multiple subsequences, compute partial sums for each subsequence, and then finally combine the partial sums.

Using the Accum effect, we can implement sum in a parallel-friendly way:

```
sum = \x:n=>Float.
  snd (runAccum \total.
    for i. total += x.i)
```

The Accum effect turns out to be flexible enough to cover many numerical algorithms, including automatically generated derivatives, allowing them to be parallelized efficiently (see Section 6.1). However, since not all algorithms are compatible with the restrictions of Accum, Dex allows the user to choose between parallel-but-write-only Accum and serial-but-flexible State.

In the full surface language runAccum is extended to support arbitrary user-defined Monoid instances in the style of Haskell, which makes the Accum effect essentially isomorphic to the Writer monad. However, automatic differentiation only needs accumulation on finite-dimensional vector spaces, so for this paper we assume the more restrictive vector space constraint.

2.3.3 Reference Indexing. While the design of our effect system largely follows previous research, with Koka [Leijen 2014] being a huge inspiration, we have extended it a bit further to include *reference indexing*. Specifically, the references used by the State and Accum effects allow the following *pure* operation:

```
(!) : Ref h (n=>a) -> n -> Ref h a
```

Given a reference to an array and an index, the ! operator extracts a reference to an array element.

Many linear-algebra routines often perform operations over, e.g., the rows of a matrix, and being able to "slice" references in this way makes it especially convenient to implement such routines, especially since chained reference indexing efficiently takes apart nested arrays. But it turns out that it is also a crucial component for being able to express parallel patterns such as segmented reductions or histograms (at least in a work-efficient way, see 6.1.3):

```
histogram = \x:n=>bins.
  snd (runAccum \hist.
    for i. hist!(x.i) += 1)
```

Finally, reference indexing is important for the efficiency of reverse-mode automatic differentiation in the presence of indexing. More details can be found in Section 5.3.4.

2.4 Comparisons

We conclude our informal introduction to Dex by comparing its expressiveness[5] with three other array programming notations. Our running point of comparison is the matrix multiplication we have already seen, this time written as a Dex function:

```
def matrix_multiply (x:n=>m=>Float) (y:m=>o=>Float): n=>o=>Float =
  for i j. sum (for k. x.i.k * y.k.j)
```

2.4.1 Combinator Languages. Most of the other functional languages for array computing (e.g. Futhark [Henriksen et al. 2017], Accelerate [Chakravarty et al. 2011], LIFT [Steuwer et al. 2017], XLA) take a somewhat different approach than Dex. Instead of exposing the array building expressions and encouraging explicit indexing, they usually provide a set of array combinators built into the language that are the only way to manipulate arrays. The combinators are often modeled after higher-order functions common between the functional programming and parallel computing communities, such as `map`, `scan`, `zip`, and `reduce`. Since this is the currently prevailing approach to array computing, it is the target of many of our comparisons in this paper, and we will refer to the languages that follow this approach as *array-combinator languages*.

In an array-combinator language, matrix multiplication might be implemented as follows:

```
combinator_matrix_multiply = \x y.
  yt = transpose y
  dot = \x y. sum (map (uncurry *) (zip x y))
  map (\xr. map (\yc. dot xr yc) yt) x
```

While this implementation is quite terse as well, it arguably does not explicate the meaning of matrix multiplication quite as explicitly as the Dex version. Many simple operations, such as a dot product, have to be expressed using multiple combinators, with explicit zipping when multiple arguments are to be consumed. Finally, it is worth noting that this style is also available in Dex, and the above is a valid Dex function.

2.4.2 Einstein Notation. Another interesting reference point is the Einstein notation, for instance as realized in the domain-specific language for the first arument to the function `numpy.einsum`. This function has been hugely successful in recent years and is one of the more popular functions, especially in the field of machine learning. It generalizes a wide variety of linear algebra operations, including matrix multiplication:

```
def matrix_multiply(x, y):
  return np.einsum('ik,kj->ij', x, y)
```

But, as it turns out, it is also very close to the Dex notation! Notice how the `ik` and `kj` index sequences exactly follow the Dex indexing expressions, while the `ij` output annotation is exactly the order of `for` binders. Hence, Dex can be seen as a sort of generalized Einstein notation, thanks to type inference automatically binding dimension sizes to loop indices. But Dex is also more flexible, because it is not limited to loops that first multiply and then sum dimensions in the standard semiring of real numbers. Both of those extensions are similar in spirit to the extensions explored by projects such as Tensor Comprehensions [Vasilache et al. 2018].

2.4.3 C-style Imperative Languages. The de-facto language for low-level numerical computations is C, which benefits from a large body of research on optimization techniques, especially for numerical

[5] While expressiveness and efficiency can be at odds, we defer the discussion of performance implications until Section 6.

programs [Bondhugula et al. 2008; Grosser et al. 2012]. A straightfoward matrix multiplication routine in C might look something like this:

```
void matrix_multiply(
  float* x, int x_rows, int x_cols, float* y, int y_cols, float* out) {
  for(int i = 0; i < x_rows; i++) {
    for(int j = 0; j < y_cols; j++) {
      for(int k = 0; k < x_cols; k++) {
        out[i][j] += x[i][k] * y[k][j]; }}}}
```

Indeed, the inner loop body is almost identical with what we wrote in Dex, and has the same benefit of explicitness about how the dimensions of the input arrays are handled. However, by statically inferring the loop bounds, Dex is able to eliminate almost all of the syntactic noise—even more than the foreach syntax available in Java and C++. On top of that, Dex's effect system exposes the parallelism available in this code, making automatic parallelization relatively straightforward (see Section 6.1.1), and Dex's AD system breaks neither the parallelism nor the total work when differentiating through such computations (see Section 5.3).

3 THE DEX LANGUAGE

We now transition to describing Dex more formally and in more detail. The structure of the rest of the paper mirrors the architecture of the Dex compiler:

(1) Dex parses the surface language, elaborates syntactic sugar, and infers omitted type annotations, producing core IR. We describe the core IR in Section 3.1; we do not discuss parsing and desugaring since they are standard.
(2) We give the type system in Section 3.2. Type inference uses a bidirectional algorithm insipred by Peyton Jones et al. [2007], but we omit discussing it in the interest of space.
(3) Dex simplifies (Section 4) the core IR into a restricted subset that omits higher-order functions.
(4) As part of simplification, Dex differentiates (Section 5) all functions that appear as arguments to differentiation operators—the compiler treats AD as a subroutine of simplification. This is both logical, since differentiation is a higher-order function itself, and convenient, since it permits Dex's AD to consume post-simplification IR but emit core IR, and rely on continuing simplification to clean up after it (e.g., see Section 5.3.2).
(5) Dex optimizes the simplified and differentiated IR with standard techniques and generates LLVM bytecode for final compilation for CPU or GPU (Section 6). Parallelism is extracted automatically (Section 6.1).

3.1 Core Dex IR

While the full Dex language supports many features (e.g. algebraic data types, dependent pairs, type classes, implicit arguments, automatically synthesized arguments, etc), we focus our exposition here on the essential core of the language. Figure 2 presents a simplified version of the core intermediate representation (IR). This is the central data structure in the Dex compiler.

The core IR is very close to the surface language, although it is somewhat compressed and requires explicit type annotations on all binders. Incompletely annotated Dex surface terms are elaborated into this representation with a bidirectional type inference algorithm inspired by Peyton Jones et al. [2007].

Most of the core language has fairly standard semantics. We have already discussed the `for` expression (Section 2.1) and the effect system (Section 2.3), and we will introduce the `view` expression in Section 3.3 (which provides an alternate way to construct values of array types). One notable limitation of the Dex language is that `let` bindings are non-recursive, and as such it is impossible

Values (including types)	
$v, \tau ::= x$	variable
$\mid l$	literal
$\mid$ Type $\mid$ Unit $\mid$ Int $\mid$ Float	base types
$\mid$ Fin v	finite index set
$\mid \tau \rightarrow \epsilon\ \tau$	function type
$\mid \tau \Rightarrow \tau$	array type
$\mid \tau \times \tau$	pair type
$\mid$ Either $\tau\ \tau$	sum type
$\mid$ Ref $\tau\ \tau$	reference type
$\mid \backslash x{:}\tau.\ e$	function
$\mid$ view $x{:}\tau.\ e$	table view
$\mid (v, v)$	pair constructor
$\mid$ Left $\tau\ v$ $\mid$ Right $\tau\ v$	sum type constructors
$\mid$ case v of Left $x \rightarrow v\ x$ Right $x \rightarrow v\ x$	value case expression
Contexts	
$E ::= \bullet$	hole
$\mid$ let $x : \tau = e$ in E	let context

Expressions	
$e ::= v$	value
$\mid$ let $x : \tau = e_1$ in e_2	let expression
$\mid v\ v$	function application
$\mid v.v$	table indexing
$\mid$ for $x{:}\tau.\ e$	table builder
$\mid$ fst v $\mid$ snd v	pair projections
$\mid$ case v of Left $x \rightarrow e$ Right $x \rightarrow e$	case expression
$\mid v\ !\ v$	reference slicing
$\mid$ runState $v\ v$ $\mid$ get v $\mid$ put $v\ v$	State operations
$\mid$ runAccum v $\mid$ v += v	Accum operations
$\mid v + v \mid v * v$	arithmetic operations
$\mid$ linearize v v	linearization
$\mid$ transpose v v	transposition
Effects	
$\epsilon ::=$ Pure	no effects
$\mid$ State τ, ϵ	state effect
$\mid$ Accum τ, ϵ	accumulator effect

Fig. 2. Dex core IR. Dex's type system is value-dependent; here we distinguish τ and v only to hint whether a given value is expected to have type Type or any type, respectively, as they are not actually different in the Dex grammar.

to express recursive functions in both the surface language and core IR. This restriction is quite important for the ideas presented here. Among other reasons, it lets us ensure that the normalization procedure outlined in Section 4 terminates. While this limitation might seem significant, in our experience with Dex, the `for` iterator abstraction provides enough expressive power for quite a wide range of applications. Compared to general recursion, `for` is both quite familiar to the scientific computing community and much easier to compile to the flat parallelism required by modern hardware accelerators. As a simple substitute for recursion and fixed-point iteration, the full language additionally includes a `while` loop, which we omit here as an entirely conventional language construct, albeit with interesting implications for AD.

Contexts, denoted E, represent sequences of let bindings with a hole in the place of the final result. These are used during the simplification and linearization passes, and are not a part of the surface language. We write $E[e]$ for completing a context E with an open term e to obtain an expression. Contexts may also be composed, which we write $E_1 \circ E_2$, by inserting E_2 into the hole of E_1; this concatenates the let bindings.

3.2 Type System

In Dex, we use a form of value dependent types [Swamy et al. 2011], and hence our core language is separated into two syntactic categories: values and expressions. Values approximately correspond to the fully reduced terms that can be implicitly lifted to appear in types.[6] So, while we do not allow arbitrary expressions in types, we do allow arbitrary *expression results*, provided they are bound to a variable. This makes type checking quite straightforward, as type equality remains syntactic (up to alpha-equivalence). The downside is a small loss in precision: when two terms that reduce to the same value are bound to different variables, this can cause type mismatches between subsequent values that lift those variables into their types. In typical Dex programs, however, this implicit lifting is used only for array shapes, and those are fortunately usually each lifted just once in the program, diminishing the importance of this drawback.

[6] One notable exception is the "value case" construct, which is generally not considered work-free. We include it as a value on a technicality: we need it to represent the result of simplifying a `case` of function type (see rule SCase in the extended version). We do not recognize "value case" values when parsing user code, defaulting to `case` expressions. In particular, value case does not occur in types.

$$\frac{\epsilon,\ x{:}\tau_1,\ \Gamma \vdash e : \tau_2 \qquad \vdash_{\texttt{IdxSet}} \tau_1}{\epsilon,\ \Gamma \vdash (\texttt{for}\, x{:}\tau_1.\, e) : (\tau_1 \Rightarrow \tau_2)}\ \textsc{TypeFor} \qquad \frac{\texttt{Pure},\ x{:}\tau_1,\ \Gamma \vdash e : \tau_2 \qquad \vdash_{\texttt{IdxSet}} \tau_1}{\epsilon,\ \Gamma \vdash (\texttt{view}\, x{:}\tau_1.\, e) : (\tau_1 \Rightarrow \tau_2)}\ \textsc{TypeView}$$

$$\frac{\Gamma \vdash v_1 : (\tau_1 \Rightarrow \tau_2) \qquad \Gamma \vdash v_2 : \tau_1}{\epsilon,\ \Gamma \vdash (v_1.v_2) : \tau_2}\ \textsc{TypeIndex} \qquad \frac{\Gamma \vdash v_1 : \texttt{Ref}\ h\ (\tau_1 \Rightarrow \tau_2) \qquad \Gamma \vdash v_2 : \tau_1}{\epsilon,\ \Gamma \vdash (v_1!v_2) : \texttt{Ref}\ h\ \tau_2}\ \textsc{TypeSlice}$$

$$\frac{\Gamma \vdash v : \texttt{Ref}\ h\ \tau}{\texttt{State}\ h,\ \epsilon,\ \Gamma \vdash \texttt{get}\ v : \tau}\ \textsc{TypeGet} \qquad \frac{\Gamma \vdash v_1 : \texttt{Ref}\ h\ \tau \qquad \Gamma \vdash v_2 : \tau}{\texttt{State}\ h,\ \epsilon,\ \Gamma \vdash \texttt{put}\ v_1\ v_2 : \texttt{Unit}}\ \textsc{TypePut}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : (\texttt{Ref}\ h\ \tau_1 \longrightarrow (\texttt{State}\ h,\ \epsilon)\ \tau_2) \qquad \vdash_{\texttt{Data}} \tau_1}{\epsilon,\ \Gamma \vdash \texttt{runState}\ v_1\ v_2 : (\tau_2 \times \tau_1)}\ \textsc{TypeRunState}$$

Fig. 3. Subset of Dex's typing rules for arrays and effects. The judgement $\vdash_{\texttt{IdxSet}} \tau$ denotes a membership of type τ in the type-class IdxSet. The remaining rules and type-class definitions can be found in the supplementary material. For type checking, we model effects as capabilities in the spirit of Brachthäuser et al. [2020].

The most interesting typing rules for the core language are presented in Figure 3. The typing judgement $\epsilon, \Gamma \vdash e : \tau$ can be read as "given the capability to perform effects ϵ, expression e evaluated in an environment with variables of types Γ produces a result of type τ." Note that effects can also appear in types, but only on the right-hand side of a function type $\tau_1 \longrightarrow \epsilon\ \tau_2$.

While we only present the rules for type-checking here, the surface language of Dex supports extensive type-inference. We expect that inference plays a crucial role in enhancing the usability of the language, especially in presence of array shapes in types, but we omit the inference rules from this work due to space constraints.

3.3 Duality of Functions and Arrays

We chose => as the array type constructor for its similarity to the function type constructor ->. Indeed, functions and arrays are almost perfectly analogous. Both are language constructs with abstraction and application, and both have the same reduction rule: `(\x. expr) y` reduces to `[x->y]expr` just as well as `(for x. expr).y` does (at least when `expr` is pure). The only real difference is in *when* the abstraction body is evaluated. In the case of functions, this happens at each application site, whereas in the case of `for` expressions it happens eagerly when the array is first defined.

Arrays are in fact just a representation for a fully memoized function: the application of an array computes the element by looking it up in memory. Conversely, functions can be seen as "lazy" or "compressed" arrays that compute the elements just-in-time as they are requested. Functions also do not require their argument type to be enumerable, whereas precomputing an array forces that requirement on us. Likewise, if the body of a function has an effect, that effect occurs when the function is called, whereas a `for` expression executes all the effects for all the iterations immediately (when relevant, the order defined to be is the element order of the index set).

In some cases, it is useful to produce a value of type `n=>a` without fully memoizing it; this can be used to model *views* or *slices* of an existing (memoized) array without unnecessary memory copies. We thus include in the core IR a table view expression

```
view i:n.expr,
```

which has the same type as `for i:n.expr` but is evaluated lazily, each time the result is indexed, rather than eagerly. Note, however, that because array indexing is expected to not carry any effects, the body of a `view` has to be pure.

The utility of `view` expressions is arguable in the surface language, but it turns out to be a very useful tool in the compiler implementation. We often use `view` to represent arrays that would be wasteful to materialize in full. This is especially important in implementing automatic differentiation with the right efficiency guarantees, because it allows us to avoid materializing arrays full of zero values for certain tangent expressions (see $\texttt{Zero}[\tau]$ in Figure 6). As we will see later (Section 4, rule SFor), having such a lazy array type is also useful for creating "array coercions" that will be used throughout the simplification pass. Finally, one more interesting use case is interfacing with (immutable) foreign-memory. `expr` can turn an index type into an offset used to dereference a pointer underlying an imported array, while having sufficient flexibility to represent a variety of data layouts (column-major, row-major, strided, triangular, etc.).

One might imagine taking this analogy to the logical extreme and dispensing with the distinction between functions and arrays entirely, at least for pure expressions when the evaluation order matters less. We have not seen a system take that approach successfully, and in Dex, we instead leave the choice of representation in the hands of the programmer.

3.4 Types of Array Indices

As we noted when we introduced the `for` expression, not all types can be used as indices. We call every type that can be so used an *index set*. In this section we summarize the requirements for an index set, and briefly discuss how we discharge them in Dex.

Firstly, each index set is required to have a finite number of members, as otherwise we couldn't represent the array in finite memory. Secondly, because index sets are used to define iteration spaces in `for` expressions which can have side effects, we need to be able to enumerate the members of each such type in a fixed total order to be able to assign consistent semantics to the `for` loops. Hence, we additionally require each index set to specify a bijection to integers between 0 and the size of the index set (exclusive). The effects induced by a `for` loop are then guaranteed to be observed in the iteration order of the index set (even if the loop is chosen by the compiler to be parallelized). Both of those requirements can be conveniently summarized in a Haskell-style type class, here presented using Dex's syntax for type classes (which we do not discuss further, due to space concerns):

```
interface IndexSet a where
  size        : Int
  ordinal     : a -> Int   -- returns integers between 0 and size - 1
  fromOrdinal : Int -> a   -- partial and only defined on valid ordinals
```

Even though technically the type of, e.g., 64-bit integers or IEEE floating-point values satisfies those requirements, we do not consider them to be index sets, because it is unlikely that a user would want to have an array with as many as 2^{64} entries. Instead, the basic index set type provided by Dex is `Fin : Int -> Type`. It is a builtin type constructor which guarantees that `Fin n` has exactly `n` members. It is often convenient to think about it as a prefix of the natural numbers up to `n`, although there are no literals of this type available by default. Valid instances can only be obtained from `for` binders or `fromOrdinal`.

For example, a 5 × 4 matrix of floats in Dex can be typed as `(Fin 5)=>(Fin 4)=>Float`. But `Fin` is a regular function that can accept arbitrary integer values, not just literals. In particular, because the Dex type system implements a form of value-dependent types [Swamy et al. 2011], it is possible to represent arrays of statically unknown size:

```
n = sum (for i:(Fin 100). ordinal i * ordinal i)
x : (Fin n)=>Float = for i. 1.0
```

Values	
$v^d ::= x$	variable
$\mid l$	literal
$\mid \texttt{view}\ x{:}\tau^d.\ b^d$	table view
$\mid (v^d, v^d)$	pair constructor
$\mid \texttt{Left}\ \tau^d\ v^d \mid \texttt{Right}\ \tau^d\ v^d$	`Either` constructors
Types	
$\tau^d ::= \texttt{Type} \mid \texttt{Unit} \mid \texttt{Int} \mid \texttt{Float}$	base types
$\mid \texttt{Fin}\ v^d$	finite index set
$\mid \tau^d \Rightarrow \tau^d$	table type
$\mid \tau^d \times \tau^d$	pair type
$\mid \texttt{Either}\ \tau^d\ \tau^d$	sum type
$\mid \texttt{Ref}\ x\ \tau^d$	reference type
Blocks	
$b^d ::= v^d$	value
$\mid \texttt{let}\ x{:}\tau^d = e^d\ \texttt{in}\ b^d$	let expression

Expressions	
$e^d ::= x.v^d$	table indexing
$\mid \texttt{for}\ x{:}\tau^d.\ b^d$	table builder
$\mid \texttt{fst}\ x \mid \texttt{snd}\ x$	pair projections
$\mid \texttt{case}\ x\ \texttt{of Left}\ \ x \to b^d$ $\texttt{Right}\ \ x \to b^d$	case expression
$\mid x\ !\ v^d$	reference slicing
$\mid \texttt{runState}\ v^d\ (\backslash h{:}\texttt{Type}\ x{:}\tau^d.\ b^d)$	`State` handler
$\mid \texttt{get}\ v^d \mid \texttt{put}\ v^d\ v^d$	`State` operations
$\mid \texttt{runAccum}\ (\backslash h{:}\texttt{Type}\ x{:}\tau^d.\ b^d)$	`Accum` handler
$\mid v^d \mathrel{+}= v^d$	`Accum` update
$\mid v^d + v^d \mid v^d * v^d$	arithmetic operations
Contexts	
$E^d ::= \bullet$	hole
$\mid \texttt{let}\ x : \tau^d = e^d\ \texttt{in}\ E^d$	let context

Fig. 4. Post-simplification Dex IR, a first-order subset of the core IR given in Figure 2.

Dex also allows the definition of richer index sets than `Fin`, which we found useful for expressing a wide range of numerical algorithms. Space prevents us from covering the possibilities thoroughly here, but we highlight tuples as one particularly good example. A tuple of index sets is a valid index set if and only if all its components are valid index sets. Tuples let us capture a type-safe "flattening" and "unflattening" transformation as a form of currying for arrays:

```
x  : n=>m=>Float = ...
y  : (n & m)=>Float = for (i,j). x.i.j
x' : n=>m=>Float = for i j. y.(i,j)
```

Now, any index-polymorphic Dex library function of type, say, `g: (a=>Float) -> (a=>Float)` is usable with data of type `x` as

```
g  : (a=>Float) -> (a=>Float)
gx : n=>m=>Float =
  y = g (for (i,j). x.i.j)
  for i j. y.(i,j)
```

This captures one of the most common uses for the `reshape` operation common to bulk array programming, while both preserving static information about array sizes, and not requiring the type system to solve systems of Diophantine equations to check which `reshape`s are valid.

4 SIMPLIFICATION TO FIRST-ORDER PROGRAMS

Higher-order functions are a big part of what makes functional programming so much fun, but it's tricky to compile them to efficient machine code. Accelerators do not always support function calls, or oftentimes have a high overhead penalty associated with them. In addition, AD becomes especially tricky with higher-order functions [Manzyuk et al. 2019; Ritchie and Sussman 2021]. To avoid these problems we apply a full *simplification pass* in the Dex compiler prior to AD and code generation. After this pass, the only functions left are monomorphic first-order top-level functions that are immediately and fully applied at their use sites in a program, provided that this program returns a non-function type. Similar normalization procedures are not uncommon in array languages [Hovgaard et al. 2018; Najd et al. 2016], and are inspired by cut elimination in formal logic.

Figure 5 presents the details of this pass, with novel aspects tailored to Dex's constructs (arrays and effect handlers) and performance considerations. A point of departure from previous work is that the pass allows returned values to be of function type, which occurs when a first-order

$$\boxed{e \rightsquigarrow E^d, v}$$

$$\frac{}{v \rightsquigarrow \bullet, v}\ \textsc{SVal} \qquad \frac{e^d : \tau^d \qquad x \text{ fresh}}{e^d \rightsquigarrow \mathtt{let}\ x : \tau^d = e^d\ \mathtt{in}\ \bullet, x}\ \textsc{SExpr} \qquad \frac{[x \mapsto v]e \rightsquigarrow E^d, v'}{(\backslash x{:}\tau.\ e)\ v \rightsquigarrow E^d, v'}\ \textsc{SApp}$$

$$\frac{e_1 \rightsquigarrow E^d_1, v_1 \qquad [x \mapsto v_1]e_2 \rightsquigarrow E^d_2, v_2}{\mathtt{let}\ x : \tau = e_1\ \mathtt{in}\ e_2 \rightsquigarrow E^d_1 \circ E^d_2, v_2}\ \textsc{SLet} \qquad \frac{[x \mapsto v]e \rightsquigarrow E^d, v'}{(\mathtt{view}\ x{:}\tau.\ e).v \rightsquigarrow E^d, v'}\ \textsc{SView}$$

$$\frac{e \rightsquigarrow E^d, v \qquad \mathtt{binders}(E^d) \vdash v \triangleright \overline{x}^{1..n} \qquad x \notin \mathtt{free}(x_1, \ldots, x_n) \qquad y \text{ fresh}}{\mathtt{for}\ x{:}\tau.\ e \rightsquigarrow (\mathtt{let}\ y = \mathtt{for}\ x{:}\tau.\ E^d[(x_1, \ldots, x_n)]\ \mathtt{in}\ \bullet), (\mathtt{view}\ x{:}\tau.\ \mathtt{let}\ (x_1, \ldots, x_n) = y.x\ \mathtt{in}\ v)}\ \textsc{SFor}$$

$$\frac{e \rightsquigarrow E^d_1,\ v_1 \quad \Gamma \vdash \mathcal{L}_x[E^d_1[v_1]] \rightsquigarrow e' \quad e' \rightsquigarrow E^d_2,\ v_2}{\mathtt{linearize}\ (\backslash x{:}\tau.\ e)\ v \rightsquigarrow [x \mapsto v](E^d_2,\ v_2)}\ \textsc{SLinearize}$$

$$\frac{e \rightsquigarrow E^d_1,\ v_1 \quad \mathcal{T}_{x \to r}[E^d_1[v_1],\ v_t] \rightsquigarrow e' \quad \mathtt{yieldAccum}\ (\backslash h{:}\mathtt{Type}\ r{:}\mathtt{Ref}\ h\ \tau.\ e') \rightsquigarrow E^d_2,\ v_2 \quad r, h \text{ fresh}}{\mathtt{transpose}\ (\backslash x{:}\tau.\ e)\ v_t \rightsquigarrow E^d_2,\ v_2}\ \textsc{STranspose}$$

$$\boxed{\overline{x{:}\tau} \vdash v \triangleright \overline{y}}$$

$$\frac{}{\cdot \vdash v \triangleright \emptyset}\ \textsc{Empty} \qquad \frac{\begin{array}{c}\overline{x{:}\tau_x} \vdash v \triangleright \overline{y} \qquad x_1 \in \mathtt{free}(v) \\ \mathtt{free}(\tau_1) \cap \overline{y} = \emptyset\end{array}}{(x_1{:}\tau_1), \overline{x{:}\tau} \vdash v \triangleright x_1, \overline{y}}\ \textsc{Used} \qquad \frac{\overline{x{:}\tau_x} \vdash v \triangleright \overline{y} \qquad x_1 \notin \mathtt{free}(v)}{(x_1{:}\tau_1), \overline{x{:}\tau} \vdash v \triangleright \overline{y}}\ \textsc{NotUsed}$$

Fig. 5. Subset of simplification rules. See the extended version for the remaining rules. $\mathcal{L}$ and $\mathcal{T}$ are defined in Figure 6 and Figure 7 respectively.

function uses higher-order constructs internally. We are also able to get much more mileage out of this kind of simplification because Dex has no constructs for defining recursive functions.

The judgement form

$$e \rightsquigarrow E^d, v$$

represents the conversion of an expression e in Dex core IR to a *simplification context* E^d that contains all the computation that needs to be performed before e reaches a value form. Here v is an (ordinary) value in Dex core IR. Figure 4 presents the subset of Dex's core IR that can be reached through these simplification contexts. The expressions e^d that may appear in a simplification context are very much like the expressions of Core but only contain values that come from a syntactic subset of Dex values, which we denote with v^d. The post-simplification IR in particular does not include functions or function types.

Simplification is semantics-preserving and non-work-increasing, in the sense that if $e \rightsquigarrow E^d, v$ then completing $E^d[v]$ produces a term operationally equivalent to e which does not introduce any more work. We inline let bindings (rule SLet) and beta reduce wherever possible (rules SApp and SView). To avoid duplicating runtime work, we only substitute let- and lambda- bound variables with values v. We emit let bindings for expressions we want to evaluate at run-time (rule SExpr), adding them to the context, E^d.

The difficult part is simplification through control constructs like `for`. What happens if our source program builds a table of functions, like the following?

```
for i.
  y1 = f1 xs.i  -- f1 is an expensive function
  y2 = f2 y1    -- f2 is an expensive function
  \z. y1 + y2 + z
```

Each function in the table is different, but they all share the same code: `\z. y1 + y2 + z`. The only meaningful difference is the run-time value of the variables `y1` and `y2`, accessed from the function's lexical scope. These might be expensive to compute, so we don't want to just inline the table definition at its indexing sites. Instead, the simplified context captures the variables `y1` and `y2` (or rather their simplified counterparts) as a tuple, turning the table of functions into a table of data. The value we produce is a `view` that indexes into the table of data to reconstitute each function:

```
-- Simplified context
let xs = for i.               -- Residual value reconstructing original type
  y1 = f1 xs.i                view i.
  y2 = f2 y1                    (y1, y2) = xs.i
  (y1, y2)                      \z. y1 + y2 + z
in
```

This transformation is handled by the rule SFor. First we recursively simplify the body of the expression e into a context E^d and a residual value v. Next, the judgement $\texttt{binders}(E^d) \vdash v \triangleright \overline{x}^{1..n}$ to a first approximation calculates the free variables of v that are bound by E^d. The context we return, $(\texttt{let}\ y = \texttt{for}\ x{:}\tau.\ E^d[(x_1, \ldots, x_n)]\ \texttt{in}\ \bullet)$, effectively binds a table of tuples for the context-bound variables. Finally the value we return can deconstruct, for every element in that table, a tuple of these values and use them in v.

Returning to the $\overline{x{:}\tau} \vdash v \triangleright \overline{y}$ judgement, we observe that the judgement is conservative to ensure that the variables collected ($\overline{y}$) have types that do not depend on other variables and hence can be tupled together in a *non-dependent* pair. With more dependency the situation is more complex. Consider the simplification of:

```
  for x.
     n = ...
     xs = for i:(Fin n). ...
     \w. sum xs
```

Although `n` does not appear in the closure as a free variable, its type mentions the variable that does appear. As it stands Dex will not perform simplification of this program, and report an error. The solution to this is to introduce *dependent pairs* when floating definitions out of bodies (rule SFor, and rules SCase, SRunAccum, SRunState in the extended version) and convert our list of free variables into *telescopes* tracking their dependencies. We leave a full formal treatment of this idea and the implementation as future work.

Finally, this pass also serves as a monomorphization pass. Polymorphic functions are just ordinary functions that take a type as an argument. We inline and then beta-reduce these functions at their use-sites, just as we do for higher-order functions.

5 AUTOMATIC DIFFERENTIATION

Dex is designed around efficient automatic differentiation (AD), which is a mainstay of machine learning and increasingly important for broader numerical computing. We now explain how Dex implements AD, and then consider several ways in which the goal of efficient and complete AD interacted with the design of Dex in Section 5.3. For a more detailed introduction to automatic differentiation we refer to the survey by Baydin et al. [2017].

5.1 Linearization

The semantics of automatic differentiation of programming language functions are defined in terms of differentiation of mathematical functions. Mathematically, for a sufficiently nice function $f : \mathbb{R}^n \to \mathbb{R}^m$, its derivative ∂f evaluated at a point $x \in \mathbb{R}^n$ is the linear map $\partial f(x) : \mathbb{R}^n \to \mathbb{R}^m$, uniquely defined by

$$f(x + v) = f(x) + \partial f(x)(v) + o(\|v\|), \qquad \forall v \in \mathbb{R}^n.$$

Equivalently, $\partial f(x)$ can be seen as a linear function that best approximates the changes in the value of f for small perturbations of the input point x. That is, $\partial f(x)(v)$ approximates the quantity $f(x+v) - f(x)$, with the error of this approximation growing together with the norm of v (i.e. with larger perturbation). In particular, the definition of $\partial f(x)$ only depends on the behavior of f on an infinitesimally small neighborhood around x.

We call x and f the *primal* input and computation, respectively, and v the *tangent*; the computations on v that occur inside $\partial f(x)$ are similarly called tangent computations. It can be convenient to identify $\partial f(x)$ with an $\mathbb{R}^{m \times n}$ matrix of partial derivatives called the Jacobian, but representing $\partial f(x)$ as a function lets us capture the sparsity that arises from the data flow graph of the implementation of f.

To model this mathematical definition computationally, Dex provides a built-in function

```
linearize : [VectorSpace a, VectorSpace b] (a -> b) -> a -> (b & (a -o b))
```

Given a function of type `a -> b` representing a mathematical function f, and an input of type `a` representing a point x in f's domain, `linearize` produces an output of type `b` representing $f(x)$, and a *structurally linear* function of type `a -o b` representing $\partial f(x)$.[7] We do not formalize the concept of structural linearity here,[8] but the intuition is that the implementation of $\partial f(x)$ should never compute any intermediates that are non-linear in the input to $\partial f(x)$. As an example, a program such as `g = \x:Float. (x * x) / x` is linear mathematically (as it can be simplified to the identity function assuming infinite precision) but not structurally, because it computes the non-linear term `x * x` as an intermediate value.

Beyond the mathematical specification, `linearize` in Dex is a well-behaved computational object:

- The computational cost of linearizing a function f at a point x, and of applying the linearized function $\partial f(x)$, is bounded by a small constant multiple of the cost of applying f to x.
- The linearized function $\partial f(x)$ is structurally similar to f: it relies on the same effects, and therefore exposes the same degree of parallelism.
- Every Dex function of suitable type can be linearized,[9] including functions produced by or using linearization, allowing the computation of higher-order derivatives.

Linearization is an instance of *forward-mode AD* [Griewank and Walther 2008], as in e.g. Elliott [2018]. In Dex, we realize it in the form of a compile-time source transformation, with a subset of interesting rules outlined in Figure 6.

There are two types of rules we consider. First,

$$\mathcal{D}_\Delta[b^d] \rightsquigarrow E,\ e_p,\ e_t$$

is the main elaboration used in the process of linearization. Given a mapping Δ from primal program variables to their respective types and tangents, it translates a (simplified) expression or block b^d into three (core IR) elements: (1) a context E, (2) a primal expression e_p, and (3) a tangent expression e_t. The invariant for $\mathcal{D}_\Delta$ is that $E[e_p]$ is equivalent to b^d, and $E[e_t]$ computes the tangent corresponding to b^d assuming the free variables of b^d are given the tangent values in Δ.

[7] This type signature puns the type of the tangent space for `a` with `a` itself. That's reasonable when `a` is a fixed-shape structure of real numbers like `n=>Float`. We do permit f to use things like integers internally, for which we have to define an implementation-internal type function `Tan[·]`, below.

[8] And Dex currently does not enforce it—the linear arrow `-o` is provided purely as documentation and is treated equivalently to `->`. Verifying structural linearity as a typing judgement would be an interesting future extension.

[9] Notably, the `VectorSpace` constraint restricts f's type to first-order. AD of higher-order functions is subtle, [Manzyuk et al. 2019; Ritchie and Sussman 2021], so Dex eschews it. But note that f is free to use higher-order functions internally—the restriction is only that f cannot be higher-order itself.

$$\boxed{\texttt{Tan}[\tau]}$$

$$\texttt{Tan}[\texttt{Float}] = \texttt{Float} \qquad \texttt{Tan}[\texttt{Int}] = \texttt{Unit} \qquad \texttt{Tan}[\tau_1 \Rightarrow \tau_2] = (\tau_1 \Rightarrow \texttt{Tan}[\tau_2])$$

$$\texttt{Tan}[(\tau_1 \times \tau_2)] = (\texttt{Tan}[\tau_1] \times \texttt{Tan}[\tau_2]) \qquad \texttt{Tan}[\texttt{Either}\ \tau_1\ \tau_2] = \text{Unsupported!}$$

$$\boxed{\texttt{Zero}[\tau]}$$

$$\texttt{Zero}[\texttt{Float}] = 0.0 \quad \texttt{Zero}[\texttt{Unit}] = () \quad \texttt{Zero}[\tau_1 \Rightarrow \tau_2] = \texttt{view}\ \_{:}\tau_1.\ \texttt{Zero}[\tau_2] \quad \texttt{Zero}[(\tau_1 \times \tau_2)] = (\texttt{Zero}[\tau_1], \texttt{Zero}[\tau_2])$$

$$\boxed{\Delta[v]} \quad \Delta ::= x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n;\ \Gamma$$

$$(\ldots, x \rightarrow v, \ldots;\ \Gamma)[x] = v \qquad (\ldots;\ x : \tau)[x] = \texttt{Zero}[\texttt{Tan}[\tau]] \qquad \Delta[l] = \texttt{Zero}[\texttt{Tan}[\tau]]\ (\text{when } l{:}\tau)$$

$$\boxed{\Gamma_d \vdash \mathcal{L}_{x_1,\ldots,x_n}[b^d] \rightsquigarrow e}$$

$$\frac{\mathcal{D}_{x_1 \rightarrow t_1, \ldots, x_n \rightarrow t_n; \Gamma^d}[b^d] \rightsquigarrow E,\ e_p,\ e_t \qquad t_1 \ldots, t_n \text{ fresh}}{\Gamma_d \vdash \mathcal{L}_{x_1,\ldots,x_n}[b^d] \rightsquigarrow E[(e_p,\ \backslash t_1 \ldots t_n.\ e_t)]}\ \textsc{LinReify}$$

$$\boxed{\mathcal{D}_\Delta[b^d] \rightsquigarrow E,\ e_p,\ e_t}$$

$$\mathcal{D}_\Delta[v] \rightsquigarrow \bullet,\ v,\ \Delta[v]\ \textsc{LinBlockResult}$$

$$\frac{\mathcal{D}_\Delta[e^d] \rightsquigarrow E_1,\ e_{p_1},\ e_{t_1} \qquad \mathcal{D}_{x \rightarrow t,\,\Delta}[b^d] \rightsquigarrow E_2,\ e_{p_2},\ e_{t_2} \qquad t \text{ fresh}}{\mathcal{D}_\Delta[\texttt{let}\ x{:}\tau = e^d\ \texttt{in}\ b^d] \rightsquigarrow (E_1 \circ (\texttt{let}\ x{:}\tau = e_{p_1}\ \texttt{in}\ \bullet) \circ E_2),\ e_{p_2},\ (\texttt{let}\ t{:}\texttt{Tan}[\tau] = e_{t_1}\ \texttt{in}\ e_{t_2})}\ \textsc{LinLet}$$

$$\boxed{\mathcal{D}_\Delta[e_d] \rightsquigarrow E,\ e_p,\ e_t}$$

$$\mathcal{D}_\Delta[v_1 + v_2] \rightsquigarrow \bullet,\ v_1 + v_2,\ \Delta[v_1] + \Delta[v_2]\ \textsc{LinAdd}$$

$$\mathcal{D}_\Delta[v_1 * v_2] \rightsquigarrow \bullet,\ v_1 * v_2,\ ((v_1 * \Delta[v_2]) + (\Delta[v_1] * v_2))\ \textsc{LinMul}$$

$$\frac{i{:}\tau,\ \Gamma \vdash \mathcal{L}_{x_1,\ldots,x_n}[b^d] \rightsquigarrow e \qquad j \text{ fresh}}{\begin{array}{c}\mathcal{D}_{x_1 \rightarrow t_1, \ldots, x_n \rightarrow t_n, \Gamma}[\texttt{for}\ i{:}\tau.\ b^d] \rightsquigarrow \\ (\texttt{let}\ x = \texttt{for}\ i{:}\tau.\ e\ \texttt{in}\ \bullet),\ (\texttt{view}\ j{:}\tau.\ \texttt{fst}\ x.j),\ (\texttt{for}\ j{:}\tau.\ (\texttt{snd}\ x.j\ t_1 \ldots t_n))\end{array}}\ \textsc{LinFor}$$

$$\mathcal{D}_\Delta[v_1\ !\ v_2] \rightsquigarrow \bullet,\ v_1\ !\ v_2,\ \Delta[v_1]\ !\ v_2\ \textsc{LinSlice}$$

$$\mathcal{D}_\Delta[\texttt{get}\ v_1\ v_2] \rightsquigarrow \bullet,\ \texttt{get}\ v_1\ v_2,\ \texttt{get}\ \Delta[v_1]\ \Delta[v_2]\ \textsc{LinGet}$$

$$\mathcal{D}_\Delta[\texttt{put}\ v_1\ v_2] \rightsquigarrow \bullet,\ \texttt{put}\ v_1\ v_2,\ \texttt{put}\ \Delta[v_1]\ \Delta[v_2]\ \textsc{LinPut}$$

$$\frac{\Delta = x_1 \rightarrow t_1, \ldots x_n \rightarrow t_n, \Gamma \qquad h{:}\texttt{Type},\ x{:}\texttt{Ref}\ h\ \tau^d,\ \Gamma \vdash \mathcal{L}_{h,\,x,\,x_1,\ldots,x_n}[b^d] \rightsquigarrow e'}{\begin{array}{c}\mathcal{D}_\Delta[\texttt{runState}\ v^d\ (\backslash h{:}\texttt{Type}\ x{:}\texttt{Ref}\ h\ \tau^d.\ b^d)] \rightsquigarrow \\ (\texttt{let}\ ((x_{\texttt{ans}}, x_{\texttt{lin}}), x_s) = \texttt{runState}\ v^d\ (\backslash h{:}\texttt{Type}\ x{:}\tau^d.\ e')\ \texttt{in}\ \bullet), \\ (x_{\texttt{ans}}, x_s), \\ \texttt{runState}\ \Delta[v^d]\ (\backslash h'{:}\texttt{Type}\ x'{:}\texttt{Ref}\ h'\ \texttt{Tan}[\tau].\ (x_{\texttt{lin}}\ h'\ x'\ t_1 \ldots t_n))\end{array}}\ \textsc{LinRunState}$$

Fig. 6. Representative rules for linearization. The linearization environment Δ carries tangent values, and also carries the primal type environment to be able to construct zero tangents when needed.

Second,

$$\Gamma \vdash \mathcal{L}_{x_1,\ldots,x_n}[b^d] \rightsquigarrow e$$

encodes an elaboration rule that linearizes a simplified block or expression b^d with respect to some of its free variables $x_1, \ldots, x_n$, and reifies the result as a primal value and tangent function. The result e is of pair type. The first component of e is equivalent to b^d; and the second component of e evaluates to a function that accepts tangent values for the x_i and returns the corresponding tangent for b^d.

Figure 6 also includes a few supporting operations. $\texttt{Zero}[\tau]$ constructs a zero value of the vector space instance associated with τ. Note that having the array shape as part of the type is crucial for this to be well-defined. `Tan[`τ`]` maps the type τ to its *tangent type* (intuitively the type of infinitesimal perturbations to τ). Tangent types necessarily have to be vector spaces, which is why we leave `Tan[Either` τ_1 τ_2`]` undefined (it is unclear which case should be used for zero). This problem can be worked around by making the mapping to the tangent type depend on the *value* instead of its type, in which case the type of the tangent would match the tangent type of the constructor used in the primal value. We leave this extension for future work.

5.2 Transposition

In practice, AD is often used to compute the gradients of scalar-valued functions $f : \mathbb{R}^n \to \mathbb{R}$. This is useful for gradient-based optimization in machine learning, or for sensitivity analysis of computational models to their parameters.

The `linearize` transform is semantically sufficient for this purpose, as given `f: (Fin n)=>Float -> Float`, one could compute the gradient by applying `snd (linearize f x)` to n different inputs, each representing a standard basis vector of $\mathbb{R}^n$. But the cost to compute the gradient would then be proportional to n applications of `f`. For neural networks with $n \simeq 10^8$, that's untenable!

The difficulty is that the result of linearization gives computational access to $\partial f(x)$ only through application, which corresponds to multiplying by the Jacobian only on the left. Fortunately, structural linearity allows us to define another compile-time transformation that reverses the inputs and outputs of structurally linear functions:

```
transpose : [VectorSpace a, VectorSpace b] (a -o b) -> (b -o a)
```

This operation is named `transpose` because it models transposition of a linear map.

Computationally, `transpose` obeys similar desiderata to `linearize`:

- The computational cost of applying `transpose f` is within a small constant multiple of the cost of applying `f`.
- The transposed function uses the same effects as the original `f`, except that repeatedly reading a value becomes associative accumulation with the (0, +) monoid, and vice versa.[10]
- All structurally linear functions can be transposed, and the result is also structurally linear.

With `transpose`, we can compute gradients using

```
grad : [VectorSpace a] (a -> Float) -> a -> a
grad f x = (transpose (snd (linearize f x))) 1.0
```

This recovers the desired effect of computing the gradient of `f` in time proportional to the runtime of `f`, and this is how Dex implements *reverse-mode AD*. See Frostig et al. [2021] for further discussion of this approach. When `transpose` is used to compute gradients this way, the intermediate values are conventionally called *cotangents*.

Transposition is another source transform we implement in the Dex compiler, with a selection of rules displayed in Figure 7. The

$$\mathcal{T}_\Omega[e^d,\ v] \rightsquigarrow E$$

elaboration transposes the simplified structurally linear expression e^d by accumulating into the references corresponding to its free linear variables, starting with the cotangent value v, corresponding to the result of e^d. The environment Ω maps each variable to a reference storing its (incrementally constructed) value in the transposed program.

Note that the rule LetOtherTranspose inverts the order of let-bindings in the block by sequencing e_2 to happen before e_1. Similarly, the rule ForTranspose reverses the iteration order by

[10] The Dex effect system actually has a `Reader` effect to serve as the transpose of `Accum`, but a further compiler transform could in principle eliminate `Reader` and replace it with variable access.

$$\boxed{\mathcal{T}_{\Omega}[v^d,\ v] \rightsquigarrow e}$$

$$\mathcal{T}_{x\to r,\ \Omega}[x,\ t] \rightsquigarrow r \mathrel{+}= t\ \textsc{VarTranspose} \qquad \mathcal{T}_{\Omega}[0.0,\ t] \rightsquigarrow \bullet\ \textsc{ZeroTranspose}$$

$$\frac{\mathcal{T}_{\Omega}[v_1,\ t_1] \rightsquigarrow e_1 \quad \mathcal{T}_{\Omega}[v_2,\ t_2] \rightsquigarrow e_2}{\mathcal{T}_{\Omega}[(v_1,\ v_2),\ t] \rightsquigarrow \texttt{let}\ t_1 = \texttt{fst}\ t\ \texttt{in}\ \texttt{let}\ t_2 = \texttt{snd}\ t\ \texttt{in}\ (e_1; e_2)}\ \textsc{PairTranspose}$$

$$\boxed{\mathcal{T}_{\Omega}[e^d,\ v] \rightsquigarrow e}$$

$$\frac{\mathcal{T}_{\Omega}[v_1,\ t] \rightsquigarrow e_1 \quad \mathcal{T}_{\Omega}[v_2,\ t] \rightsquigarrow e_2}{\mathcal{T}_{\Omega}[v_1 + v_2,\ t] \rightsquigarrow e_1; e_2}\ \textsc{AddTranspose}$$

$$\frac{\mathcal{T}_{\Omega}[v_1,\ t'] \rightsquigarrow e \quad \Omega \cap \texttt{free}(v_2) = \emptyset}{\mathcal{T}_{\Omega}[v_1 * v_2,\ t] \rightsquigarrow \texttt{let}\ t' = t * v_2\ \texttt{in}\ e}\ \textsc{MulLeftTranspose}$$

$$\frac{\Omega \cap \texttt{free}(v_1) = \emptyset \quad \mathcal{T}_{\Omega}[v_2,\ t'] \rightsquigarrow e}{\mathcal{T}_{\Omega}[v_1 * v_2,\ t] \rightsquigarrow \texttt{let}\ t' = v_1 * t\ \texttt{in}\ e}\ \textsc{MulRightTranspose}$$

$$\frac{\mathcal{T}_{\Omega}[b^d,\ t'] \rightsquigarrow e}{\mathcal{T}_{\Omega}[\texttt{for}\ i{:}\tau^d.\ b^d,\ t] \rightsquigarrow \texttt{for}\ i{:}\tau^d.\ [i \mapsto \texttt{reverse}\ i](\texttt{let}\ t' = t.i\ \texttt{in}\ e)}\ \textsc{ForTranspose}$$

$$\mathcal{T}_{x\to v,\ \Omega}[\texttt{get}\ x,\ t] \rightsquigarrow \texttt{let}\ t' = \texttt{get}\ v\ \texttt{in}\ \texttt{put}\ v\ (t + t')\ \textsc{GetTranspose}$$

$$\frac{\mathcal{T}_{\Omega}[v^d,\ t'] \rightsquigarrow e}{\mathcal{T}_{x\to v',\ \Omega}[\texttt{put}\ x\ v^d,\ t] \rightsquigarrow \texttt{let}\ t' = \texttt{get}\ v'\ \texttt{in}\ (\texttt{put}\ v^d\ \texttt{Zero}[\tau];\ e)}\ \textsc{PutTranspose}$$

$$\frac{\mathcal{T}_{h\to h',\ x\to x',\ \Omega}[b^d,\ t_{\texttt{ans}}] \rightsquigarrow e_1 \qquad \mathcal{T}_{\Omega}[v,\ t'_{\texttt{s}}] \rightsquigarrow e_2 \qquad x',\ h'\ \text{fresh}}{\begin{array}{c}\mathcal{T}_{\Omega}[\texttt{runState}\ v\ (\backslash h{:}\texttt{Type}\ x{:}\texttt{Ref}\ h\ \tau^d.\ b^d),\ t] \rightsquigarrow \\ \texttt{let}\ (t_{\texttt{ans}}, t_{\texttt{s}}) = t\ \texttt{in}\ \texttt{let}\ ((), t'_{\texttt{s}}) = \texttt{runState}\ t_{\texttt{s}}\ (\backslash h'{:}\texttt{Type}\ x'{:}\texttt{Ref}\ h'\ \tau^d.\ e_1)\ \texttt{in}\ e_2\end{array}}\ \textsc{RunStateTranspose}$$

$$\boxed{\mathcal{T}_{\Omega}[b^d,\ v] \rightsquigarrow e}$$

$$\frac{\Omega \cap \texttt{free}(e^d) = \emptyset \qquad \mathcal{T}_{\Omega}[b^d,\ t] \rightsquigarrow e'}{\mathcal{T}_{\Omega}[\texttt{let}\ x{:}\tau^d = e^d\ \texttt{in}\ b^d,\ t] \rightsquigarrow \texttt{let}\ x{:}\tau^d = e^d\ \texttt{in}\ e'}\ \textsc{LetNonlinearTranspose}$$

$$\frac{\mathcal{T}_{x\to r', y\to r,\ \Omega}[b^d,\ t] \rightsquigarrow e}{\mathcal{T}_{y\to r,\ \Omega}[\texttt{let}\ x{:}\tau^d = y.v\ \texttt{in}\ b^d,\ t] \rightsquigarrow \texttt{let}\ r' = r!v\ \texttt{in}\ e}\ \textsc{LetIndexingTranspose}$$

$$\frac{\mathcal{T}_{x\to r', y\to r,\ \Omega}[b^d,\ t] \rightsquigarrow e}{\mathcal{T}_{y\to r,\ \Omega}[\texttt{let}\ x{:}\tau^d = y!v\ \texttt{in}\ b^d,\ t] \rightsquigarrow \texttt{let}\ r' = r!v\ \texttt{in}\ e}\ \textsc{LetSliceTranspose}$$

$$\frac{\Omega \cap \texttt{free}(e^d) \neq \emptyset \qquad \mathcal{T}_{x\to r,\ \Omega}[b^d,\ t] \rightsquigarrow e_2 \qquad \mathcal{T}_{\Omega}[e^d,\ t'] \rightsquigarrow e_1}{\mathcal{T}_{\Omega}[\texttt{let}\ x{:}\tau^d = e^d\ \texttt{in}\ b^d,\ t] \rightsquigarrow (\texttt{let}\ t' = (\texttt{runAccum}\ \backslash h{:}\texttt{Type}\ r{:}\texttt{Ref}\ h\ \tau^d.\ e_2)\ \texttt{in}\ e_1)}\ \textsc{LetOtherTranspose}$$

Fig. 7. A selection of transposition rules for the post-simplification language.

substituting the index variable with its inverted counterpart. This only matters if the body uses the state effect.

We close this section by acknowledging that this decomposition of reverse-mode AD into forward-mode AD followed by transposition is unusual, and most AD systems just implement reverse-mode monolithically, sometimes not exposing forward-mode to users at all. Unfortunately, formalizing this approach or defending its virtues would take us too far afield, but we hope the research community does that soon, at least more fully than Frostig et al. [2021] did.

5.3 Challenges Posed by Automatic Differentiation

We feel that Dex gained a great deal as a language from being co-designed with its automatic differentiation system. AD is something like a very demanding user of the language—it is always trying to write programs the compiler developers did not anticipate, and always producing compelling bug reports or feature requests when those programs do not work or are slower than they should be. In this section, we discuss a few specific subtleties in the design of Dex's AD, and the effects AD has had on the rest of the language and compiler.

5.3.1 Cheap Gradient Principle. One of Dex's core design goals is to uphold the "cheap gradient principle" [Griewank and Walther 2008]: evaluating a function linearization or its transpose should cost at most a small constant factor more than evaluating the function itself. A second goal is to allow programmers to reason about performance, at least asymptotically, using a simple operational semantics that doesn't overly rely on compiler optimizations. Taken together, these goals imply that we should aim to achieve desired efficiency by ensuring that linearization and transposition both straightforwardly preserve work, up to a constant factor, according to the simple operational semantics. We don't offer a formal proof of the cheap gradients principle for our AD system, unlike Bernstein et al. [2020], who use a formal operational semantics and a cost model, but we qualitatively discuss some of the hard cases, especially in Section 5.3.4.

5.3.2 Capturing Scoped Intermediates. Non-linear primitive operations such as `mul` need to capture the intermediate values computed inside the differentiated function. This is fine if the intermediate is in scope for the remainder of the function; but whenever we linearize an expression that may construct intermediates that go out of scope (such as `for`), we have to arrange for their values to be captured.

This is why we define differentiation to return core IR rather than the post-simplification subset of it. By emitting core IR, `linearize` and `transpose` can just capture whatever they need in a fresh lambda expression, whose type then doesn't need to reflect the type of the data it is carrying in the closure. We then recover post-simplification IR by running simplification again on the output. This architecture does mean that simplification and differentiation have to form a loop in the compiler, rather than being sequential passes, but we feel that the simplification of linearization and transposition thus won is worth it.

5.3.3 Differentiation of State-Mutating Code. Reverse-mode automatic differentiation is notoriously difficult in the presence of mutation in the program being differentiated. The issue is again storage of intermediate primal values: what if one of these primals is mutated by a later operation in the program?

In Dex, this problem disappears, because the type system distinguishes between the (mutable) *reference* being read and/or updated, and the (immutable) *value* one obtains from it with `get`. The value is saved by `linearize` as needed, by the same mechanism as all other values. The data in the reference itself does not need to be stored, because `get` is linear: knowing the cotangent of the value emitted is enough to compute the update that must be made to the mutable buffer, and no additional information about the primal data therein is needed. Conversely, the transpose of `put` just reads the currently accumulated cotangent of the reference.

Even in pure code, it remains important that only *non-linear* operations store their inputs. Many Dex loops operate on just one or a few index values of the state array per iteration, so making a complete copy of the state at each step could raise even the asymptotic complexity of the linearized function. Array indexing, however, is linear, so all those copies can be avoided. Indeed, given an expression such as `for i. sin (get ref).i`, only the argument of `sin` needs to be saved for each iteration of the loop.

5.3.4 Transposing Array Indexing. The cotangent of array indexing is a recurring problem in the design of automatic differentiation systems. The reverse-mode update due to reading `x = array.i` is of course the sparse update `array_cot!i += x_cot`, but how should we represent this?

In an imperative language, one is always free to just emit the mutating update, which has been the de-facto standard implementation technique of AD systems over decades. This correctly conserves work, but turns what used to be a parallelizable loop reading `array` into a sequential loop writing to `array_cot`. Of course, the parallelism can be recovered by downstream compiler analyses in simple cases, but complex cases are not difficult to come by.

On the other hand, in pure array languages (such as JAX [Bradbury et al. 2018] and other Python-based array-libraries) the conventional pattern used to implement reverse-mode AD is to just one-hot encode the indexing update, i.e., create an update array of almost all zeros. This is simple to implement, and costs no parallelism, but of course creates an asymptotic increase in memory usage and count of arithmetic operations. Given the importance of indexing in Dex, this is not a viable option for us.

Another alternative in a pure system is to perform cotangent accumulation functionally by accumulating the updates in sparse data structures that gather the index or indices where an update is to occur along with the value(s) to be added there. This is also asymptotically work-preserving, but imposes large constant or logarithmic factor overhead in performance, as well as a significant constant factor in developer effort. It is also quite unfriendly to hardware acceleration due to the inherent dynamism in the shape of the data structure and requirements on dynamic memory allocation. We are thus not aware of any AD system that actually differentiates indexing this way.

Instead, we note that while general mutation is certainly sufficient to preserve work, it's not actually necessary—the only mutation that reverse-mode AD needs is to accumulate sums. More importantly, because summation is associative (up to floating-point rounding), it can still be executed in parallel. The desire to capture this was the original reason we introduced an effect system into Dex, and in particular why we defined `Accum`. The `Accum` effect is expressive enough to capture cotangent updates, but restrictive enough to be easy to parallelize (Section 6.1). Having introduced `Accum` into the language, we promptly found other uses for it besides cotangent accumulation, for instance the `histogram` example from Section 6.1.2.

6 PERFORMANCE AND PARALLELISM

In this section we describe the remainder of the Dex compiler. After simplification (Section 4) and differentiation (Section 5):

(1) Dex optimizes the post-simplification and differentiated IR with standard techniques: inlining, dead code elimination, common subexpression elimination, loop invariant code motion, etc. We comment in Section 6.2 on why Dex does not need a suite of operation fusion optimizations.
(2) Dex generates LLVM bytecode for final compilation for CPU or GPU. Code generation is target-aware: Dex chooses different LLVM instructions depending on the hardware being compiled for. The interesting part is mapping Dex structures to parallel execution strategies, which we cover in Section 6.1.

Finally, in Section 6.3 we comment on a few places where Dex's rich type system makes the compiler simpler and more effective, and present some preliminary benchmark results in Section 6.4.

6.1 Automatic Parallelization

The surface language of Dex does not expose any way for the user to directly express an intent to evaluate a number of expressions in parallel. Instead, all parallelism-related decisions are made by the compiler. In this section, we describe Dex's automatic parallelization algorithm (Section 6.1.1), and then comment on how Dex's `Accum` effect navigates a common work-parallelism tradeoff

(Section 6.1.2) and allows the programmer to smoothly shift among different-seeming parallel computation patterns (Section 6.1.3).

6.1.1 Parallelism Allocation. Many modern hardware accelerators are highly constrained, and can only efficiently parallelize programs that transform batches of data in a uniform manner (much like the SIMD units in CPUs). Dex makes parallelization decisions at the granularity of `for` expressions, because they have exactly this characteristic: a sequence of instructions (the body expression) is evaluated repeatedly over slightly different data (the loop indices and values derived from them).

Furthermore, accelerators generally do not support nested parallelism, meaning that the total number of parallel invocations has to be decided at the top level and cannot be increased later. Due to this, Dex flattens nested `for` expressions that can be run in parallel. We do not describe this procedure formally, because it is largely analogous to the one proposed for Futhark by Henriksen et al. [2017] (if only one replaces Futhark's `map` with `for`, while treating the `Accum` effect similarly to `reduce` and `State` to `loop`).

6.1.2 Work-Efficiency Guarantees. While we approach parallelism extraction highly analogously to Futhark, it is a much less critical optimization for Dex than it is for array-combinator languages. The reason is that there are programs that can be expressed in most array-combinator languages in only two ways: one that is work-efficient but fully sequential, and another that is parallelizable but is *not* work-efficient. One good example of such a program is histogram calculation:

```
histogram_seq = \points:(n=>k).
  yieldState (for i:k. 0) \hist.
    for i. hist!(points.i) += 1

histogram_par = \points:(n=>k).
  sum (map one_hot points)
one_hot = \idx:k. for i. if i == idx then 1 else 0
```

The `histogram_seq` function has the right run-time complexity of $O(n + k)$ (where n and k are the sizes of the index sets `n` and `k`), but uses a stateful loop and so it cannot be made parallel without sophisticated analysis. On the other hand, `histogram_par` is a composition of the two parallel operators `map` and `sum`, but a naive sequential execution would suffer the significantly worse asymptotic complexity of $O(nk)$. Since the goal of array-combinator languages is to enable parallel execution, the first approach is considered undesirable. Instead, array-combinator languages *have to* perform an optimization step that turns the abundant parallelism present in `histogram_par` into a partially sequential loop, that lets them eventually achieve the desired work efficiency (see Henriksen et al. [2017] for details) or, as many do, accept the sequential performance penalty.

Meanwhile, in Dex, the same program can be expressed with an associative accumulator effect:

```
histogram_dex = \points:(n=>k).
  yieldAccum \hist.
    for i. hist!(points.i) += 1
```

If the `for` expression's body was to be evaluated sequentially, it would have the right complexity of $O(n + k)$. But, because the `Accum` effect exposes enough structure, the Dex compiler is able to take this work-efficient implementation and evaluate the body in parallel too.

In short, while we expect the eventual evaluation strategy employed by both approaches to be largely equivalent, Dex has the benefit of being able to naturally express a parallel *and* work-efficient program instead of relying on opaque compiler optimizations. While it might seem like a minor point, we strongly believe that this is a big step forward in terms of usability. Compiler optimizations are certainly useful, but they are largely outside of the control of the user and as such they should not be the only way to achieve the right asymptotics.

6.1.3 Reduction Patterns by Indexing. Every use of the accumulation effect corresponds to some form of reduction, but it is useful to classify them by strategies that can carry out different reductions in parallel. The simplest pattern is a *complete reduction*, where a large set of values is reduced to just a single one (and we do not try to take advantage of any structure the result might have). Then, a *regular segmented reduction* occurs when the reduced value is an array of equal-size segments, each of which contributes one component of the result. In that case, each parallel thread can be assigned a subset of segments to reduce and will not need to replicate the full accumulator, but only the entries it computes. Multiple compilation strategies for this pattern have been implemented in Futhark [Larsen and Henriksen 2017]. Finally, one gets an *irregular segmented reduction* or a *histogram* when the different segments are of varying sizes that may be difficult to predict even at run-time (for example, because the data of different segments is interleaved). Yet again, a special routine for this case has been implemented recently in Futhark [Henriksen et al. 2020].

In an array-combinator language, these three types of reductions are naturally expressed as distinct combinators. In Dex, however, they are just slightly different indexing patterns for the accumulator reference:

- A complete reduction is when the accumulator is never indexed.
- A regular segmented reduction is when the accumulator is indexed by a subset of the loop indices.
- An irregular segmented reduction is when the indices of the accumulator are derived form arbitrary expressions (such as an array lookup).

Those differences are readily visible in the following example:

```
complete_reduction = \values:n=>Float.
  yieldAccum \acc. for i. acc += values.i

segmented_reduction = \values:n=>m=>Float.
  yieldAccum \acc. for i j. acc!i += values.i.j

histogram = \classes:n=>k.
  yieldAccum \hist. for i. hist!(classes.i) += 1
```

While Dex does not take advantage of this observation and treats all reductions as though they were complete, we can imagine recovering more sophisticated execution strategies with a more carefully effect-aware loop parallelization transform.

6.2 Fusion Optimizations

One advantage of Dex's program representation is that Dex does not need a large and complex suite of fusion rules to achieve good performance.

Indeed, one of the drawbacks of the bulk array programming model is that each individual bulk operation needs to encapsulate enough work to amortize away the overhead of dispatching a kernel for it and allocating storage for its inputs and outputs. Expressiveness, however, calls for composing programs out of many small operations, because they can be put together in many different ways. Traditional array-combinator languages bridge this gap with *fusion* optimizations, each of which combines a pattern of smaller operations into a larger one. The quintessential example is combining a sequence of two `map` operations into one `map` of the composition of the two functions, thus eliding the intermediate array.

Unfortunately, such fusion rules have to be specified for almost every pair of combinators, leading to a drastic increase in the complexity of the system. To make matters even worse, fusion might necessitate adding more combinators to capture computation patterns which the fusion rules can emit; and these new combinators then need more fusion rules of their own. For example, Futhark uses an otherwise-redundant combinator named `redomap` [Henriksen et al. 2016] to represent a computation that constructs an array while reducing a set of values.

In Dex, in contrast, all fusion rules can be seen as instances of inlining followed by reduction. For example, consider this block, showing a reduction happening along a map, with its result being consumed in another map:

```
y = for j.
      acc += f j
      g j
x = for i. h y.i
```

Assuming that this is the only use of the value `y`, each element of `y` is consumed exactly once. This means that we can safely inline the `for` expression that constructs it, as long as we reduce the `(for j. ...).i` form to avoid duplicating effects. This yields just a single loop that avoids materializing the `y` array, but produces each of its elements on demand instead:

```
x = for i.
      acc += f i
      h (g i)
```

As it has been noted before, fusion optimizations in array languages are just instances of loop-fusion [Matsuzaki and Emoto 2010]. Hence, replicating a high-quality set of fusion optimizations in Dex is a matter of doing a good job of inlining `for` expressions. Fortunately, the literature on inlining strategies is vast [Mitchell 2010; Peyton Jones and Marlow 2002]. This is also where the effect system comes in handy, as it lets us easily decide which inlining decisions are legal (i.e., do not duplicate nor reorder effects).

6.3 Type-Directed Compilation

Dex's type information is also useful for making optimization decisions and emitting efficient code. In addition to the usual benefits of type-directed compilation, explicit index set types ensure that no out-of-bounds access could ever happen upon array indexing—a type-safe index value cannot, by definition, be out of bounds. In general, a run-time bounds check is still necessary when the index is constructed, e.g., by `fromOrdinal`. However, the vast majority of indices are constructed by `for` expressions, and in that case the bounds checks can be trivially elided, since we know that the ordinals a `for` iterates over are always in bounds.

Moreover, having the full (nested) array type available at lowering time, we are able to perform an array-of-structures to structure-of-arrays layout conversion which usually achieves much better performance on parallel architectures. After this conversion, we end up with a number of nested array types that only contain primitive scalar types, which makes it possible for us to emit code that computes the total number of array elements (or in many cases even derive it statically), and allocate a single flat memory buffer that holds unboxed values. Ultimately, this means that the rich array abstraction always gets translated into just a few pointers to unboxed data, along with a few integers used to compute offsets into the flattened arrays. As a result, Dex programs generally should not incur any significant overheads compared to the languages traditionally used for low-level array computing such as Fortran, C, and C++.

As mentioned previously, having explicit effect types allows us to statically distinguish the loops that are embarassingly parallel (no effects), parallelizable with some care (only a number of `Accum` effects), or necessarily serial (when the loop body induces the `State` effect). The fine-grained effect system presents us with both a simple compiler pipeline, and a straightforward mental framework that allows our users to understand the parallelization opportunities their code exposes.

6.4 Evaluation

As an early test of the effectiveness of our language and compilation strategy, we compare the runtime of Dex programs against Futhark [Henriksen et al. 2017] on four programs from two benchmark suites. We chose Futhark as a baseline because it is one of the closest languages to Dex in purpose and structure, and because it has already been shown to have broadly competitive performance with the low-level (usually C++/CUDA based) implementations by the original benchmark authors.

One important caveat: the benchmark programs we have selected are not necessarily representative of common array workloads. Specifically, given the relative unsophistication of our optimization pipeline, we do not expect Dex to be competitive yet on workloads that depend critically on specific, heavily hand-optimized compute-intensive kernels like matrix mutiplication. Instead, we selected the benchmark problems to highlight tasks that stress compilation of more-general array programs, where eliminating composition overheads is important.

6.4.1 Benchmark Results. We compare Dex to Futhark on four problems:

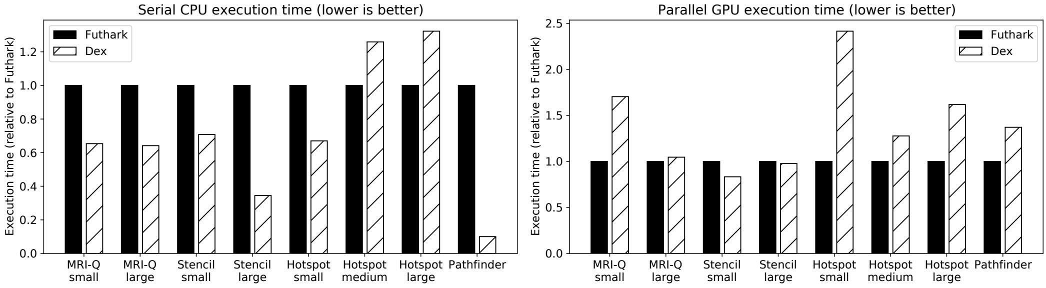
- Hotspot, a stencil computation that solves heat equations from Rodinia [Che et al. 2009];
- Pathfinder, a dynamic program, also from Rodinia;
- MRI-Q, a standard map-reduce operation from the Parboil suite [Stratton et al. 2012]; and
- Stencil, a 3-D stencil computation, also from Parboil.

Three of these benchmarks define multiple data sizes on which they can be evaluated, and we include this breakdown as well.

The results can be seen in Figure 8. To aid reproducibility, we have used an `n1-standard-8` Google Cloud instance (30GB RAM, 8 vCPUs) with a single NVIDIA V100 GPU and an Intel Skylake CPU. The relevant software versions are: CUDA 11.2, Dex from commit `069781e` and Futhark 0.18.6.

The general trend is that Dex is somewhat faster than Futhark on serial CPU execution, and somewhat slower on parallel GPU execution. The worst slowdowns in the parallel setting can be observed on the smallest benchmarks, which are small enough so that the overheads associated with calling the CUDA functions become noticeable. The difference usually becomes less pronounced as the size of the data grows, which is the most important use case for parallel accelerators.

We emphasize that our goal is not to demonstrate that Dex delivers performance improvements over existing systems. Rather, the goal at this stage is to check that Dex can more or less match



(a) Serial CPU results

(b) Parallel GPU results

Fig. 8. Comparison of Dex and Futhark execution times in serial and parallel settings. Note that all times are normalized and relative to the run time of the respective Futhark programs. The Y axis of both figures grows with execution time, so less is better. The Futhark programs were authored by the Futhark developers and published on GitHub for evaluation against their respective benchmark suites; we recompiled and re-ran them on the same hardware as our Dex programs for those benchmarks.

the state of the art, while exploring a novel approach to expressing numerical computing workloads. As Dex matures and we devote more effort to its optimization pipeline, we expect its absolute performance to improve, though the same is of course true of comparable systems as well.

7 RELATED WORK

While many ideas present in Dex have been explored separately, the language can be seen as a synthesis of a wide range of long-standing research topics.

7.1 Array Languages

Dex follows a long line of array-oriented and data-parallel languages, dating back to APL [Iverson 1962] and NESL [Blelloch 1993], which has recently seen a surge in activity [Hu et al. 2019; McDonell et al. 2013; Peyton Jones 2008; Ragan-Kelley et al. 2013; Shaikhha et al. 2019; Steele et al. 2011; Steuwer et al. 2017]. The most widely-used today are first-order nd-array languages in the NumPy family [Abadi et al. 2016; Bradbury et al. 2018; Harris et al. 2020; Paszke et al. 2019], whose advantages and disadvantages are discussed in Section 1. The Remora [Slepak et al. 2014] language, formalizes the automatic *lifting* of primitive functions to operate on arguments of higher-rank, and improves certain design aspects of this family of languages. Contrary to Remora, Dex makes lifting *explicit* via the `for` construct. Another relevant body of related work is in programming languages based on array combinators, treated in Section 2.4.1. Futhark is arguably the most similar language to Dex [Henriksen et al. 2017]. Similarly to the typed index sets in Dex, it tracks and propagates array shapes in its type system, making it possible to catch a whole class of (potential) shape errors at compile time. In-scope variables of integer type can be implicitly lifted to bounds for array dimensions, leading to a mechanism similar to the `Fin` index set constructor in Dex. Futhark also supports stateful computations, although it does so via a uniqueness typing system that guarantees safe in-place modification, instead of an effect system. Julia [Bezanson et al. 2017] is another interesting point of comparison. It shares many of the same goals as Dex: it's built for numerical array-oriented computing, it emphasises loopy scalar code instead of bulk array operations, and it's developing first-class AD [Innes 2018]. But it occupies a very different point in the design space. Julia is firmly imperative and dynamically typed, where Dex is functional and static. Julia's imperative semantics appear to pose the usual challenges for automatic parallelization and parallelism-preserving AD, as discussed in Section 7.3.

7.2 Type Systems

The Dex type system is based on a number of previously published ideas. While it breaks the highly conventional value-type boundary, it does so only by introducing a limited form of dependent typing, based on value-dependent types [Swamy et al. 2011]. In particular, types cannot depend on arbitrary expressions, but only on fully-evaluated values. Dex's effect system is modeled on the row-polymorphic effect system of Koka [Leijen 2014] and mostly implements a subset of it, with the slight exception of extending it with the reference slicing operation. However, to the best of our knowledge, the use of an associative state effect (`Accum`) in order to emit parallel code has not been considered previously in any other effect system. Our type inference algorithm largely follows the ideas of bidirectional type inference outlined by Peyton Jones et al. [2007], although in Dex higher-rank types are generally replaced by functions with implicit arguments.

7.3 Automatic Differentiation

Automatic differentiation has a deep history [Baydin et al. 2017; Griewank and Walther 2008; Pearlmutter and Siskind 2008]. Dex benefits from many influences, especially Elliott [2018] and Bradbury et al. [2018]. For brevity, we focus the discussion on array- and parallelism-oriented AD systems. For a more complete treatment, see the survey Baydin et al. [2017] and the related works of Shaikhha et al. [2019] and Bernstein et al. [2020].

Machine learning has made great use of automatic differentiation. Popular systems like Theano [Bergstra et al. 2010], Autograd [Maclaurin et al. 2014], Chainer [Tokui et al. 2019], TensorFlow [Abadi et al. 2016], DiffSharp [Baydin et al. 2017], PyTorch [Paszke et al. 2019], and JAX [Bradbury et al. 2018] employ the bulk array programming model, relying on accelerator-friendly parallelism within each bulk operation. This model is great for AD, since linearization and transposition mostly preserve program structure and hence performance. But array indexing is a weak spot, especially in the context of loops: transposing a loop of indexed reads falls outside the set of performant programs, resulting in programs that use too much memory, execute too many operations, or exhibit too little parallelism. There are mitigation techniques; for example, Autograd and TensorFlow add a runtime sparse data representation, while JAX relies on downstream whole-program optimizations. But these can be brittle, add overheads, or yield the wrong asymptotic complexity. For the most part, these AD systems work well because users constrain themselves to using bulk operations rather than more flexible loops and indexing.

Systems that support performant loops along with indexed reads and writes, like ADIFOR [Bischof et al. 1992], Tapenade [Hascoet and Pascual 2013], Zygote.jl [Innes 2018], RelayIR [Roesch et al. 2018], and DiffTaichi [Hu et al. 2020], are better equipped to provide AD with the right asymptotic complexity. But indexed writes, introduced by transposition of indexed reads, typically must be sequenced, giving up parallelism and accelerator-friendliness. The aim of Dex's AD is to support these performant loops with indexing, yet preserve parallelism through typed effects.

ATL [Bernstein et al. 2020] also tackles this issue head-on, achieving AD of loopy indexing code while preserving both complexity and parallelism. The core approach is to encode sparsity with APL's Iverson bracket, then rely on compiler optimizations. The language is carefully designed along with these optimizations so that the system is provably efficient, rather than handling only important special cases [Hückelheim et al. 2019; Li et al. 2018]. This is powerful, but limits expressiveness. Overall, guaranteed sparsity optimizations like ATL's and explicit `Accum` effects like Dex's may prove complementary.

8 CONCLUSION

We presented a synthesis of ideas from functional programming intended to support stateful and "pointful" numerical code in a richly-typed functional language. From the user's point of view, we expect that this approach can support a programming style roughly as expressive and flexible as low-level imperative numerical code, while catching a wider range of bugs at compile time and being more concise through high-level abstractions. From the compiler's point of view, a fine-grained typed effects system makes it possible to implement optimizations and program transformations (such as automatic differentiation and structurally-linear transposition) in a way that robustly preserves performance.

ACKNOWLEDGMENTS

We would like to thank Dan Zheng, Sasha Rush and Lyndon White for their contributions to the open source implementation of Dex. We are also very grateful for many helpful conversations with Roy Frostig, Gilbert Bernstein, George Necula, Martin Abadi and Gordon Plotkin.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A

System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 265–283.

Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic Differentiation in Machine Learning: A Survey. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 5595–5637.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, Vol. 4. Austin, TX, 1–7.

Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. 2020. Differentiating a Tensor Language. arXiv:2008.11256

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. 1992. ADIFOR — generating derivative codes from Fortran programs. *Scientific Programming* 1, 1 (1992), 11–29. https://doi.org/10.1155/1992/717832

Guy E. Blelloch. 1993. *NESL: A Nested Data-Parallel Language (Version 2.6).* Technical Report. USA.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428194

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

Manuel M T Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming*. ACM. https://doi.org/10.1145/1926354.1926358

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. IEEE, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

Conal Elliott. 2018. The Simple Essence of Automatic Differentiation. *Proc. ACM Program. Lang.* 2, ICFP, Article 70 (July 2018), 29 pages. https://doi.org/10.1145/3236765

Roy Frostig, Matthew Johnson, Dougal Maclaurin, Adam Paszke, and Alexey Radul. 2021. Decomposing reverse-mode automatic differentiation. In *LAFI '21: POPL 2021 workshop on Languages for Inference.*

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (second ed.). Society for Industrial and Applied Mathematics, USA. https://doi.org/10.1137/1.9780898717761

Tobias Grosser, Armin Größlinger, and C. Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Process. Lett.* 22 (2012). https://doi.org/10.1142/S0129626412500107

Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Laurent Hascoet and Valérie Pascual. 2013. The Tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)* 39, 3 (2013), 1–43. https://doi.org/10.1145/2450153.2450158

Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. 2020. Compiling Generalized Histograms for GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press, Article 97, 14 pages. https://doi.org/10.1109/SC41405.2020.00101

Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. 2016. Design and GPGPU Performance of Futhark's Redomap Construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2016)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/2935323.2935326

Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 556–571. https://doi.org/10.1145/3062341.3062354

Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2018. High-Performance Defunctionalisation in Futhark. In *International Symposium on Trends in Functional Programming*. Springer, 136–156. https://doi.org/10.1007/978-3-030-

18506-0_7

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Fredo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=B1eB5xSFvr

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* 38, 6, Article 201 (Nov. 2019), 16 pages. https://doi.org/10.1145/3355089.3356506

Jan Hückelheim, Navjot Kukreja, Sri Hari Krishna Narayanan, Fabio Luporini, Gerard Gorman, and Paul Hovland. 2019. Automatic differentiation for adjoint stencil loops. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10. https://doi.org/10.1145/3337821.3337906

Michael Innes. 2018. Don't Unroll Adjoint: Differentiating SSA-Form Programs. *CoRR* abs/1810.07951 (2018). arXiv:1810.07951

Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley & Sons, Inc., USA.

Rasmus Wriedt Larsen and Troels Henriksen. 2017. Strategies for Regular Segmented Reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC 2017)*. Association for Computing Machinery, New York, NY, USA, 42–52. https://doi.org/10.1145/3122948.3122952

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/178243.178246

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (Jun 2014), 100–126. https://doi.org/10.4204/eptcs.153.8

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)* 37, 4 (2018), 139:1–139:13. https://doi.org/10.1145/3197517.3201383

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. 2014. Autograd: Effortless gradients in numpy *(ICML '15 AutoML workshop)*.

Oleksandr Manzyuk, Barak A. Pearlmutter, Alexey Andreyevich Radul, David R. Rush, and Jeffrey Mark Siskind. 2019. Perturbation confusion in forward automatic differentiation of higher-order functions. *Journal of Functional Programming* 29 (2019), e12. https://doi.org/10.1017/S095679681900008X

Kiminori Matsuzaki and Kento Emoto. 2010. Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In *Implementation and Application of Functional Languages*, Marco T. Morazán and Sven-Bodo Scholz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–89. https://doi.org/10.1007/978-3-642-16478-1_5

Trevor L. McDonell, Manuel M T Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *ICFP '13: The 18th ACM SIGPLAN International Conference on Functional Programming*. ACM. https://doi.org/10.1145/2500365.2500595

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

Neil Mitchell. 2010. Rethinking Supercompilation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 309–320. https://doi.org/10.1145/1863543.1863588

Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything Old is New Again: Quoted Domain-Specific Languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2847538.2847541

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 8024–8035.

Barak A Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 1–36. https://doi.org/10.1145/1330017.1330018

Simon Peyton Jones. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS '08)*. Springer-Verlag, Berlin, Heidelberg, 138. https://doi.org/10.1007/978-3-540-89330-1_10

Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.* 12, 5 (July 2002), 393–434. https://doi.org/10.1017/S0956796802004331

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *J. Funct. Program.* 17, 1 (Jan. 2007), 1–82. https://doi.org/10.1017/S0956796806006034

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176

Sam Ritchie and Gerald Jay Sussman. 2021. AD on Higher Order Functions. Unpublished note.

Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A new IR for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 58–68. https://doi.org/10.1145/3211346.3211348

Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient Differentiable Programming in a Functional Array-Processing Language. *Proc. ACM Program. Lang.* 3, ICFP, Article 97 (July 2019), 30 pages. https://doi.org/10.1145/3341701

Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 27–46. https://doi.org/10.1007/978-3-642-54833-8_3

Guy L. Steele, Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu. 2011. *Fortress (Sun HPCS Language)*. Springer US, Boston, MA, 718–735. https://doi.org/10.1007/978-0-387-09766-4_190

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, 74–85. https://doi.org/10.1109/CGO.2017.7863730

J. A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, N. Anssari, G. Liu, and W. Hwu. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-Dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 266–278. https://doi.org/10.1145/2034773.2034811

Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. 2019. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2002–2011. https://doi.org/10.1145/3292500.3330756

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