Designing Dex: differentiation, parallelism, and index types

Dougal Maclaurin
Inria, March 14, 2022








The NumPy* model of array programming

First-order array ops called from an interpreted host language

The good

- Easily embeddable (no need for a new language and compiler)
- Access to data parallelism (GPUs! TPUs!)
- Primitive set closed under automatic differentiation

The bad

- Don't get to design a new language and compiler!
- **Expressiveness**
    - Fixed set of reductions
    - No sequential loops
    - Limited data types
    - Rectangular arrays only
- **Clarity**
    - Constrains program organization (e.g. loops forced inward)
    - Shape and indexing errors

* a.k.a. APL model, MATLAB model, TensorFlow model, PyTorch model, JAX model

Dex: a functional array language in the Haskell/ML family

```
map : (a -> b) -> n=>a -> n=>b =
  \f x. for i. f x.i
```

github.com/google-research/dex-lang

Goals

- Performance
- Parallelism
- Automatic differentiation
- Precise data modeling

Design choices

- Functional, static, strict
- Flat data, flat control
- Fine-grained effects
- Rich index types

⚠️ Research use only! ⚠️

Language

Function types, dually

	Function	Array
Type	`a -> b`	`a=>b`
Elimination	`f expr`	`f.expr`
Introduction	`\x:ty. expr`	`for x:ty. expr`
Construction	Cheap	Expensive, effectful
Application	Expensive, effectful	Cheap
Domain	Arbitrary	Finite (ordered)

Potential deja'vu if you've heard of representable functors

Quick examples

```
3d     : (Fin 3)=>Float
vector : (Fin n)=>Float                    (assuming n:Int in scope)
matrix : (Fin n)=>(Fin m)=>Float           (assuming n:Int and m:Int in scope)
sum    : n:Type ?-> n=>Float -> Float
intIndexed : Int=>Float
> Type error! Couldn't synthesize (Ix Int)!
```

Syntax benchmark: matrix multiply

SOAC

```
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
```

NumPy

```
matmul = lambda x, y: np.einsum('ik,kj->ij', x, y)
```

SaC

```
{ [i,j] -> sum ({ [k] -> A[i,k] * B[k,j] }) }
```

Dex

```
for i:(Fin n). for j:(Fin m). sum (for k:(Fin q). x.i.k * y.k.j)
for i:(Fin n) j:(Fin m). sum (for k:(Fin q). x.i.k * y.k.j)
for i j. sum (for k. x.i.k * y.k.j)
for i j. sum for k. x.i.k * y.k.j
```

By the way: you can be as pointfree as you'd like!

```
def uncurry {a b c} (f:a -> b -> c) : (a & b) -> c = \(x, y). f x y
def zip {n a b} (x:n=>a) (y:n=>b) : n=>(a & b) = for i. (x.i, y.i)
def map {n a b} (f:a -> b) (x:n=>a) : n=>b = for i. f x.i
def transpose {n m a} (x:n=>m=>a) : m=>n=>a = for i j. x.j.i

def combinator_matrix_multiply {n k m}
    (x:n=>k=>Float) (y:k=>m=>Float) : n=>m=>Float =
  yt = transpose y
  dot = \x y. sum (map (uncurry (*)) (zip x y))
  map (\xr. map (\yc. dot xr yc) yt) x
```

A pointful foundation doesn't make pointfree programming harder!

Type system

```
def broadcast {a} (v:a) (n: Type) [Ix n]: n=>a = for i. v

broadcast 2.0 (Fin 5)
> [2.0, 2.0, 2.0, 2.0, 2.0]

i5  = 2 + 3
i5' = 2 + 3
broadcast 2.0 (Fin i5) + broadcast 2.0 (Fin i5')
> Type error! Expected (Fin i5)=>Float, but got (Fin i5')=>Float!

-- in lib/prelude.dx
def Fin   (n:Int) : Type = Range 0 n
def Range (low:Int) (high:Int) = ...

x : (Fin 5) = ...
```

Loop bound inferred from *return type annotation*

Very limited normalization applied to types

But not entirely trivial!

Sum and (dependent) product types

```
data Maybe a =
  Just a
  Nothing

data List a =
  MkList (length:Int) (elements:(Fin length)=>a)

def filter {n a} (f:a -> Bool) (x:n=>a) : List a = …

MkList _ validData = filter isValid data
sum validData
```

Can tensor programming be liberated from integer indices?

- Traditional array *sizes* are integers
- Traditional array *indices* are integers

- Dex array *sizes* are *types*
- Dex array *indices* are elements of that type

Pale Ties Out
@PTOOP

Every time you see *numbers*, remember that Nat = List 1, and ask yourself what it is that the 1 has forgotten. Differences between numbers are often hacker-level proxies for differences between entities whose pertinence has become invisible. Numbers are a code smell.

19/01/2022, 23:18

Rich index sets

In Dex, any type *conforming to Ix* can be an array index:

```
interface Ix n where
  size n              : Int
  toOrdinal           : n -> Int
  unsafeFromOrdinal : Int -> n
```

size

& isomorphism with a prefix of natural numbers

```
def fromOrdinal {n} [Ix n] (o:Int) : Maybe n =
  case 0 <= o && o < size n of
    True  -> Just (unsafeFromOrdinal o)
    False -> Nothing
```

Basic shape arithmetic can be done using standard type constructors:

Products	`(n & m)`
Sums	`(n | m)`
Exponentials	`(n=>m)`

Basic examples

Reshapes

```
reshape (2, -1, 4) x
```

(n & m)-typed binder

```
for i (j, k) l. x.i.j.k.l
```

Concatenation

```
concatenate x y
```

(n | m)-typed binder

```
for ci. case ci of
  Left  xi -> x.xi
  Right yi -> y.yi
```

Named axes

`image[h, w]` or `image[w, h]?`

```
image.{height=h, width=w}
image.{width=w, height=h}
```

Boundary conditions

```
x: (Fin (1 + n))=>a
x[0] vs x[1 + i]
```

```
x: (Unit|n)=>a
x.(Left ()) vs x.(Right i)
```

Index sets for compilers

Integer-based indexing

```
nmp = n + m + p
for i in range(nmp).
  if i < n
    then x[i]
    else if i - n < m
      then y[i - n]
      else z[i - n - m]
```

Sum-type-based indexing

```
for i in (n|(m|p)).
  case i of
    Left ni -> x.ni
    Right i' -> case i' of
      Left  mi -> y.mi
      Right pi -> z.pi
```

A loop with a sum-typed index set either never inspects the index, or is a very good candidate for loop splitting!

Indexing lemmas

Array reversal

```
def reflect {n} (i:n) : n =
  unsafeFromOrdinal n (size n - 1 - ordinal i)
```

```
sequence : (Fin s)=>Int = …
for i in range(len(sequence)).
  sequence[len(sequence) - 1 - i]
```

Correctness reasoning requires non-local context (e.g. range of i)

```
sequence : n=>Int = …
for i.
  sequence.(reflect i)
```

Dynamic programming

```
def prev (i:n) : (Unit|n) =
  unsafeFromOrdinal _ (ordinal i)
```

```
x : (Fin s)=>Int = …
sumWithPrev = for i in range(len(x)).
  if i == 0
    then x[i]
    else x[i - 1] + x[i]
```

Easy to forget about the base case and read out of bounds!

```
x : (Unit|n)=>Int = …
sumWithPrev = for i.
  case i of
    Left  () -> x.i
    Right i' -> x.(prev i') + x.i
```

Index sets are user-definable

```
data RGB = Red | Green | Blue
instance Ix RGB
  size = 3
  toOrdinal = \x. case x of
    Red   -> 0
    Green -> 1
    Blue  -> 2
  unsafeFromOrdinal = …

data HSV = Hue | Saturation | Value
instance Ix HSV …

Image = \h w colorSpace. { height: (Fin h) & width: (Fin w) }=>colorSpace=>UInt8

imgRGB : Image 200 200 RGB = loadKnownSizeJPG "doggo.jpg"
imgHSV : Image _   _   HSV = RGBtoHSV imgHSV
hues = for h w. imgHSV.{height=h, width=w}.Hue
```

← Arrays can function as *named tuples*

Relational/dataframe programming

```
CREATE TABLE airports (
  airport TEXT PRIMARY KEY,
  city    TEXT REFERENCES (cities))

CREATE TABLE flights (
  flight TEXT PRIMARY KEY,
  from   TEXT REFERENCES airports(airport),
  to     TEXT REFERENCES airports(airport))

SELECT city, count(*)
FROM flights JOIN airports
ON flights.from = airports.airport
GROUP BY city;

 city      |  count
Boston     | 50
Paris      | 71
...        | ...
```

```
airports : Airport=>{city:City}

flights : Flight=>{ from : Airport
                  , to   : Airport}

count : [Ix a, Ix b] (a=>b) -> (b=>Int)

numFlightsByCity : City=>Int =
  count $ for f:Flight.
     airports.(((flights.f)~from))~city
```

Fencepost problems


```
data (n:Type) Gaps =
  UnsafeMakeGaps Int

instance Ix (Gaps n)

def leftEdge [Ix] (i:Gaps n) : n =
  UnsafeMakeGaps i' = i
  unsafeFromOrdinal i'

def RightEdge [Ix] (i:Gaps n) : n = ...

def leftGap [Ix] (i:n) -> Maybe (Gaps n) = ...

def RightGap [Ix] (i:n) -> Maybe (Gaps n) = ...
```

```
def diffs (x: n=>Float) : (Gaps n)=>Float =
  for i. x.(RightEdge i) - x.(leftEdge i)

def applyDiffs (x0:Float) (dxs: (Gaps n)=>Float) : n=>Float =
  ...
```

Array type zoo

🤔 If we have dependent functions... why don't we try dependent arrays?

Array kind	Example type
Static	`(Fin 10)=>(Fin 20)=>Float`
Dynamic	`(Fin n)=>(Fin m)=>Float`
Structured ragged	`(i:Fin 10)=>(...i)=>Float`
Ragged	`(i:Fin 10)=>(Fin lengths.i)=>Float`
Jagged	`(Fin 10)=>List Float`

Homogeneous ↕ **Heterogeneous**

Pushing the limits of our type system here

Also:
Position-dependent arrays and their application for high performance code generation, F. Pizzuti et al.
Generating High Performance Code for Irregular Data Structures using Dependent Types, F. Pizzuti et al.

ADTs in scientific computing

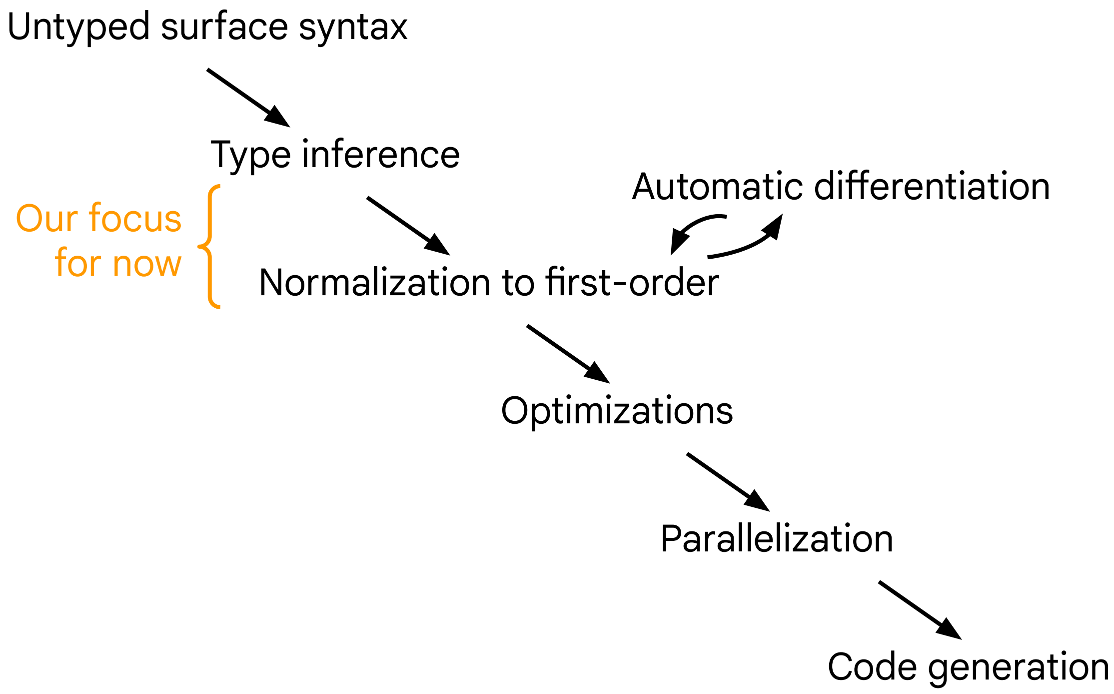
```haskell
type CT = Float

data PCRResult =
   Positive (Maybe CT)
 | Negative
 | Missing
```

```haskell
data RawPCRResult =
   Biofire
 | Cobas { eGene :: CT
         , nGene :: CT}
 | Thermofisher { orfGene :: CT
                , eGene   :: CT
                , nGene   :: CT }
 | NoAmplification
 | ControlFailure
 | ParseError String
```

Implementation

Going deeper



Also: High-Performance Defunctionalisation in Futhark, A. K. Hovgaard et al.

Zooming into AD

forward-mode AD ≈ linearize

```
linearize : (a -> b) -> a -> (b, a -o b)
```

Every linear transform has a *transpose*.

```
transpose : (a -o b)      -> (b -o a)
```

reverse-mode AD = linearize + transpose[1]

[1]Decomposing reverse-mode automatic differentiation, R. Frostig et al.

Implementing linearization

Multiplication

```
linearize \x. x * y   ↦   \x. (x * y,
                                \xt. x * xt + xt * y)
```

Composition

```
linearize \x. f (g x)   ↦   \x. (t, glin) = linearize g x
                                (y, flin) = linearize f t
                                (y, \xt. flin (glin xt))
```

For loops

```
linearize \x. for i. f x i   ↦   ???
```

(rematerialize)

```
\x. (for i. f (x, i),
     \xt. for i.
            snd (linearize f (x, i)) xt.i)
```

(arrays of functions)

```
\x. (ys, flins) = unzip (for i. linearize f (x, i))
    (ys, \xt. for i. flins.i xt.i)
```

Normalizing arrays of functions

First-order context

Arbitrary atoms (incl. lambdas!)

```
toFirstOrder : Nest Decl -> (Nest Decl, Substitution Name Atom)
```

Lambda for table type

```
toFirstOrder ( x = for i.        )  ↦  ( tmp = for i.          ,  x ->                                )
                 v1 = …                      fo1 = …                 view i.
                 …                           …                         atom[reconSubst][a1,…,an/tmp.i]
                 vn = …                      fom = …
                 atom                        (a1, …, ak)
```

```
((fo1 = …; …; fom = …), reconSubst) = toFirstOrder (v1 = …; …; vn = …)
```

Normalize block

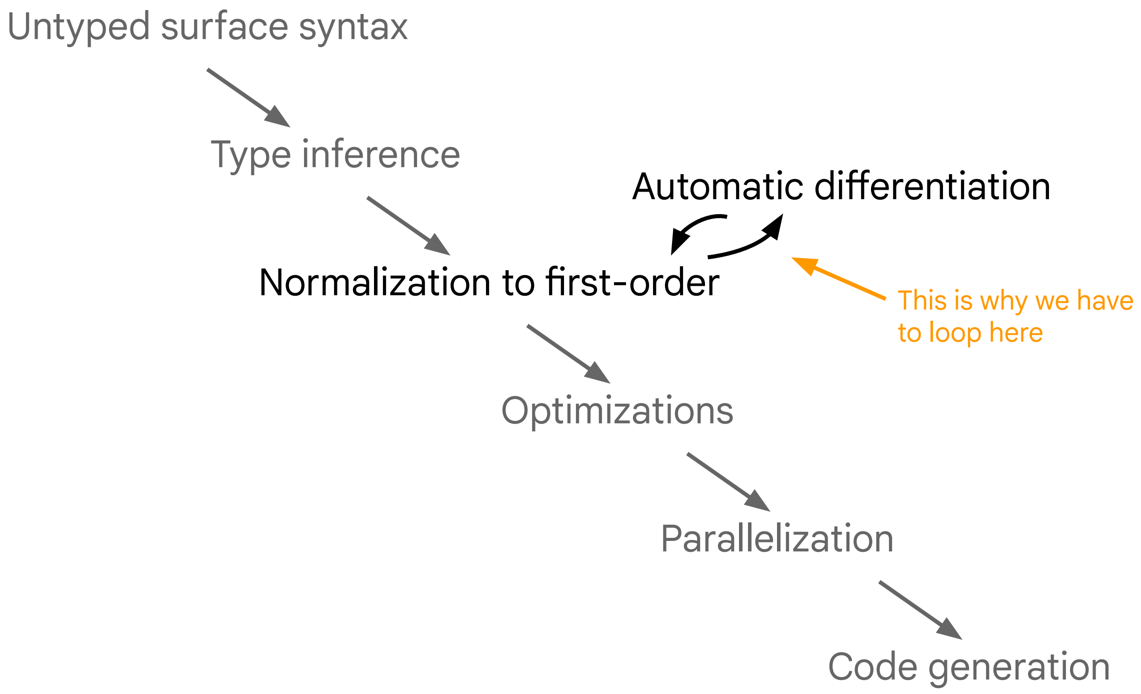
```
(a1, …, ak) = intersect (freeVars atom[reconSubst]) (fo1, …, fom)
```

Find first-order variables sufficient for reconstruction

Can only:
(1) reference functions defined outside of for, or
(2) lambda expressions with body FVs.

Similar trick also works (and is needed!) for `case` expressions

Going deeper



Efficiency issues loom

Scaling

```
\xt       . zt = xt * c        ↦   \zt. xt = zt * c
            zt                           xt
```

Addition

```
\(xt, yt). zt = xt + yt        ↦   \zt. xt = zt
            zt                           yt = zt
                                         (xt, yt)
```

Duplication

```
\xt       . zt = (xt, xt)      ↦   \zt. xt = fst zt
            zt                           xt = xt + snd zt
                                         xt
```

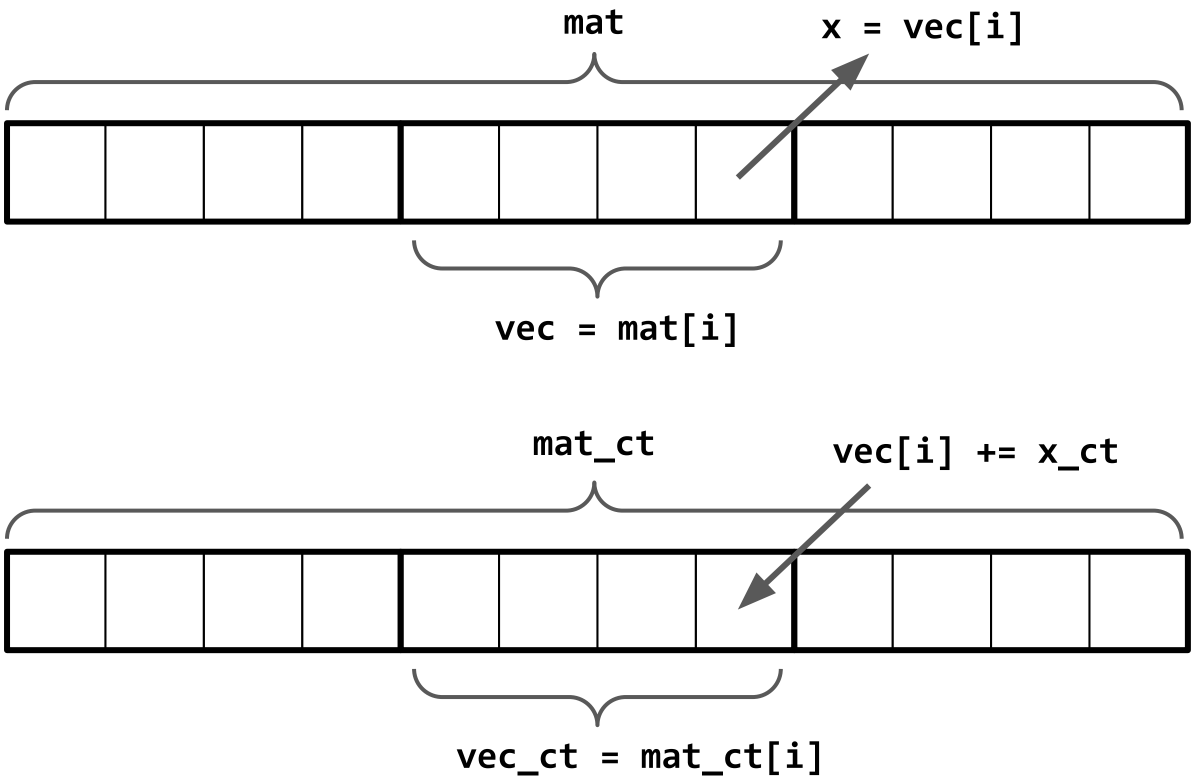
Broadcast

```
\xt       . zt = for i. xt     ↦   \zt. xt = sum zt
            zt                           xt
```

Indexing

```
\xt       . xt.i               ↦   ???
                                    xt[i] += zt"
```

FP's unstated cost model: indexing is aliasing



We need to alias writes like we alias reads!

Transposition of indexing

1️⃣ Imperative AD

```
store x_ct[i] ((load x_ct[i]) + y_ct)
```

❌ Unconstrained heap mutation

2️⃣ Dense updates

```
x_ct2 = x_ct + one_hot(y_ct, i)
```

❌ Lots of wasted work, wrong asymptotics

3️⃣ Sparse updates

```
x_ct2 = x_ct + sparse_one_hot(y_ct, i)
```

❌ Unacceptable constant factors, difficult on GPUs

3️⃣ Functional in-place (linear) updates

```
x_ct2 = consume_and_update(x_ct, i, y_ct)
```

❌ Sequentializes code

5️⃣ Associative accumulation effect

```
accumulate y_ct into x_ct[i]
```

Solution: effects

(Basic) Accumulation

```
def sum {n} (x:n=>Float) : Float =
  (_, total) = withAccum \acc.
    for i.
      acc += x.i
  total
```

Accumulator cannot be read

Final value obtained once the accumulator cannot be modified

State

```
def scan {n i o s eff}
    (f:i -> s -> {|eff} (o, s)) (init:s)
    (x:n=>i) : {|eff} n=>o =
  (result, final) = withState init \ref.
    for i.
      ref := f x.i (get ref)
  result
```

Arbitrary monoidal reductions

```
def reduce {n a} (m:Monoid a) (x:n=>a) : a =
  (_, total) = withAccum m \acc.
    for i.
      acc o= x.i
  total
```

Differentiation through reductions over arbitrary monoids is non-trivial![1]

[1] Parallelism-preserving automatic differentiation for second-order array languages, A. Paszke et al.

Efficient AD as a language design benchmark

There exists a constant c such that for every program P the cost of evaluating P' (P' being derived using forward- or reverse-mode AD from P) is at most c times larger than the cost of evaluating P.

Good reverse-mode autodiff support requires:

1️⃣ Closure under partial evaluation

2️⃣ Closure under data-flow duality

For example, reverse-mode AD of (parallel associative) `scan` is inefficient![1]

[1] Parallelism-preserving automatic differentiation for second-order array languages, A. Paszke et al.

Current / future work

- User-extensible (parallel-friendly) algebraic effects (see PEPM paper[1])
- Monomorphization without complete inlining
- Typeclass system rework (embracing overlap!)
- Recursion and recursive ADTs
- Develop relational/dataframe programming further
- Make Dex fast!
- …

[1]Parallel Algebraic Effect Handlers, N. Xie, D. J. Johnson et al.

Thank you!