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! ⚠

---

gp_regression :: (a -> a -> Real) -> n->a -> n->a -> Real
gp_regression kernel xs ys x_test =
gram. i. j = kernel xs. i xs. j
cov. i = kernel xs. i x_test
dot (solve gram cov) ys

dot : n->Real -> n->Real -> Real
= \x y. sum (for i. x. i * y. i)
# Function types, dually

<table>
<thead>
<tr>
<th></th>
<th>Function</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>[a \rightarrow b]</td>
<td>[a \Rightarrow b]</td>
</tr>
<tr>
<td>Elimination</td>
<td>[f \ expr]</td>
<td>[f.\ expr]</td>
</tr>
<tr>
<td>Introduction</td>
<td>[\lambda x:ty. \ expr]</td>
<td>[\text{for } x:ty. \ expr]</td>
</tr>
<tr>
<td>Construction</td>
<td>\text{Cheap}</td>
<td>\text{Expensive, effectful}</td>
</tr>
<tr>
<td>Application</td>
<td>\text{Expensive, effectful}</td>
<td>\text{Cheap}</td>
</tr>
<tr>
<td>Domain</td>
<td>\text{Arbitrary}</td>
<td>\text{Finite (ordered)}</td>
</tr>
</tbody>
</table>

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

\[
\text{combinator\textunderscore matrix\textunderscore multiply} = \lambda x, y. \\
\quad y_t = \text{transpose } y \\
\quad \text{dot} = \lambda x, y. \text{sum} (\text{map} \ (\text{uncurry} \ \ast) \ (\text{zip} \ x \ y)) \\
\quad \text{map} \ (\lambda x_r. \text{map} \ (\lambda y_c. \text{dot} \ x_r \ y_c) \ y_t) \ x
\]

NumPy

\[
\text{matmul} = \lambda x, y: \text{np.einsum}('ik,kj->ij', x, y)
\]

SaC

\{
\quad \{ [i,j] \rightarrow \text{sum} \ (\{ [k] \rightarrow A[i,k] \ast B[k,j] \}) \}
\}

Dex

\[
\text{for } i: (\text{Fin } n). \text{ for } j: (\text{Fin } m). \text{ sum} \ (\text{for } k: (\text{Fin } q). \ x.i.k \ast y.k.j) \\
\text{for } i: (\text{Fin } n) \text{ j: (Fin } m). \text{ sum} \ (\text{for } k: (\text{Fin } q). \ x.i.k \ast y.k.j) \\
\text{for } i, j. \text{ sum} \ (\text{for } k. \ x.i.k \ast y.k.j) \\
\text{for } i, j. \text{ sum} \ (\text{for } k. \ x.i.k \ast y.k.j)
\]
By the way: you can be as pointfree as you'd like!

```haskell
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 =
ty = 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

```gherkin
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

---

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 & isomorphism with a prefix
    unsafeFromOrdinal : Int -> n 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

<table>
<thead>
<tr>
<th>Reshapes</th>
<th><code>reshape (2, -1, 4) x</code></th>
<th>for <code>i (j, k) l. x.i.j.k.l</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td><code>concatenate x y</code></td>
<td><code>for ci. case ci of</code>&lt;br&gt;<code>Left xi -&gt; x.xi</code>&lt;br&gt;<code>Right yi -&gt; y.yi</code></td>
</tr>
<tr>
<td>Named axes</td>
<td><code>image[h, w] or image[w, h]?</code></td>
<td><code>image.{height=h, width=w}</code>&lt;br&gt;<code>image.{width=w, height=h}</code></td>
</tr>
<tr>
<td>Boundary conditions</td>
<td><code>x: (Fin (1 + n))=&gt;a</code>&lt;br&gt;<code>x[0] vs x[1 + i]</code></td>
<td>`x: (Unit</td>
</tr>
</tbody>
</table>
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

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

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

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

Dynamic programming

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

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

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

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

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

<table>
<thead>
<tr>
<th>city</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>50</td>
</tr>
<tr>
<td>Paris</td>
<td>71</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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?

<table>
<thead>
<tr>
<th>Array kind</th>
<th>Example type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homogeneous</td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td>((\text{Fin } 10)\Rightarrow(\text{Fin } 20)\Rightarrow\text{Float})</td>
</tr>
<tr>
<td>Dynamic</td>
<td>((\text{Fin } n)\Rightarrow(\text{Fin } m)\Rightarrow\text{Float})</td>
</tr>
<tr>
<td>Structured ragged</td>
<td>((i:\text{Fin } 10)\Rightarrow(...i)\Rightarrow\text{Float})</td>
</tr>
<tr>
<td>Ragged</td>
<td>((i:\text{Fin } 10)\Rightarrow(\text{Fin } \text{lengths}.i)\Rightarrow\text{Float})</td>
</tr>
<tr>
<td>Jagged</td>
<td>((\text{Fin } 10)\Rightarrow\text{List }\text{Float})</td>
</tr>
</tbody>
</table>

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

type CT = Float

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

data RawPCRResult =
  Biofire
  | Cobas { eGene :: CT
            , nGene :: CT}
  | Thermofisher { orfGene :: CT
                     , eGene   :: CT
                     , nGene   :: CT }
  | NoAmplification
  | ControlFailure
  | ParseError String
Implementation
Going deeper

Untyped surface syntax ➔ Type inference ➔ Normalization to first-order ➔ Optimizations ➔ Parallelization ➔ Code generation

Our focus for now: Automatic differentiation

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

\[
\text{forward-mode AD} \approx \text{linearize}
\]

\[
\text{linearize : } (a \rightarrow b) \rightarrow a \rightarrow (b, a - o b)
\]

Every linear transform has a \text{transpose}.

\[
\text{transpose : } (a - o b) \rightarrow (b - o a)
\]

\[
\text{reverse-mode AD} = \text{linearize} + \text{transpose}^1
\]

\[^1\text{Decomposing reverse-mode automatic differentiation, R. Frostig et al.}\]
Implementing linearization

**Multiplication**

\[ \text{linearize } \lambda x. x \times y \quad \mapsto \quad \lambda x. (x \times y, \lambda xt. x \times xt + xt \times y) \]

**Composition**

\[ \text{linearize } \lambda x. f (g x) \quad \mapsto \quad \lambda x. (t, glin) = \text{linearize } g x \\
(y, flin) = \text{linearize } f t \\
(y, \lambda xt. flin (glin xt)) \]

**For loops**

\[ \text{linearize } \lambda x. \text{for } i. f x i \quad \mapsto \quad ???? \]

\[ \text{(rematerialize)} \]

\[ \lambda x. (\text{for } i. f (x, i), \lambda xt. \text{for } i. \text{snd (linearize } f (x, i) \text{ ) } xt.i) \]

\[ \text{(arrays of functions)} \]

\[ \lambda x. (\text{ys, flins) = unzip (for } i. \text{linearize } f (x, i) \text{) } \text{(ys, } \lambda xt. \text{for } i. \text{flins.i } xt.i) \]
Normalizing arrays of functions

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

\[
\begin{align*}
\text{toFirstOrder} (x = \text{for } i. \quad v_1 = \ldots \quad \ldots \quad v_n = \ldots \quad \text{atom}) &\implies (\text{tmp} = \text{for } i. \quad \text{fo}_1 = \ldots \quad \ldots \quad \text{fom} = \ldots \quad (a_1, \ldots, a_k) \\
\text{view } i. \quad \text{atom[reconSubst]}[a_1,\ldots,a_n/tmp.i] &:\text{x ->}
\end{align*}
\]

\[
((\text{fo}_1 = \ldots; \ldots; \text{fom} = \ldots), \text{reconSubst}) = \text{toFirstOrder} (v_1 = \ldots; \ldots; v_n = \ldots)
\]

\[
(a_1, \ldots, a_k) = \text{intersect (freeVars atom[reconSubst]) (fo}_1, \ldots, \text{fom})
\]

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

Untyped surface syntax

Type inference

Normalization to first-order

Automatic differentiation

This is why we have to loop here

Optimizations

Parallelization

Code generation
Efficiency issues loom

<table>
<thead>
<tr>
<th></th>
<th>Equation</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scaling</strong></td>
<td>$z_t = x_t * c$</td>
<td>$z_t. x_t = z_t * c$</td>
</tr>
<tr>
<td><strong>Addition</strong></td>
<td>$(x_t, y_t). z_t = x_t + y_t$</td>
<td>$z_t. x_t = z_t$</td>
</tr>
<tr>
<td><strong>Duplication</strong></td>
<td>$x_t. z_t = (x_t, x_t)$</td>
<td>$z_t. x_t = \text{fst } z_t$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$x_t = x_t + \text{snd } z_t$</td>
</tr>
<tr>
<td><strong>Broadcast</strong></td>
<td>$x_t. z_t = \text{for } i. x_t$</td>
<td>$z_t. x_t = \text{sum } z_t$</td>
</tr>
<tr>
<td><strong>Indexing</strong></td>
<td>$x_t. i$</td>
<td>$&amp;&amp;[i] += z_t$</td>
</tr>
</tbody>
</table>
FP's unstated cost model: indexing is aliasing

We need to alias writes like we alias reads!

mat[i] = vec

vec = mat[i]

vec[i] += x

vec_ct = mat_ct[i]

We need to alias writes like we alias reads!
### Transposition of indexing

1. **Imperative AD**
   
   ```
   store x_{ct}[i] ((load x_{ct}[i]) + y_{ct})
   ```

2. **Dense updates**
   
   ```
   x_{ct2} = x_{ct} + one_hot(y_{ct}, i)
   ```

3. **Sparse updates**
   
   ```
   x_{ct2} = x_{ct} + sparse_one_hot(y_{ct}, i)
   ```

4. **Functional in-place (linear) updates**
   
   ```
   x_{ct2} = consume_and_update(x_{ct}, i, y_{ct})
   ```

5. **Associative accumulation effect**
   
   ```
   accumulate y_{ct} into x_{ct}[i]
   ```

- **Unconstrained heap mutation**
- **Lots of wasted work, wrong asymptotics**
- **Unacceptable constant factors, difficult on GPUs**
- **Sequentializes code**
Solution: effects

(Basic) Accumulation

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

State

```scala
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**

```scala
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

¹Parallelism-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:**

1. Closure under partial evaluation
2. 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!