Size-Dependent Types for Practical Data-Parallel Programming

Martin Elsman[1]

Department of Computer Science, University of Copenhagen (DIKU)

Oxford Seminar on Tensor Computation

November 12, 2021

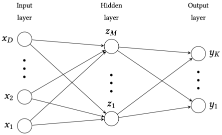
[1]Joint work with Troels Henriksen, DIKU

Increased Focus on Tensor Programming

The concept of **multi-dimensional arrays** (i.e., *tensors*) is currently undergoing a renaissance in terms of available programming libraries and code bases.

The increased attention is primarily driven by:

- The last decade's **machine learning revolution**, which is founded on tensor-programming.
- The move towards **massively data-parallel hardware** for high-performance computing, for which tensor-programming is a natural match.



Array-Size Mismatches are Rarely Checked Statically

- For most practical tensor-programming languages and libraries, array-size mismatches are rarely checked statically.
- With more complex code bases (e.g., user-defined layers of machine learning networks), static checking can lead to more reliable code.

Static Tracking of Array-Sizes

- We present a **practical** type system for array programming that uses a combination of size-dependent types and existential typing for giving static guarantees about array-size matching.
- The techniques are implemented in the data-parallel language **Futhark**, a purely functional array language targeting massively parallel hardware such as GPUs.

Futhark,[2] the Language (I) – Purely Functional & Data-Parallel

- Features a selection of Second-Order Array Combinators (SOACs) with **parallel semantics**:

```
val map    [n] 'a 'b : (a → b) → [n]a → [n]b
val scan      [n] 'a : (a → a → a) → a → [n]a → [n]a
val reduce        'a : (a → a → a) → a → []a → a
val filter        'a : (a → bool) → []a → []a
```

- Notice that type schemes may be **parameterised** by *array sizes*.

- "Empty" array sizes are implicitly quantified (universally, when in contravariant positions, existentially, when in covariant positions).

```
val filter [n] 'a : (a → bool) → [n]a → ?[m].[m]a
```

- Proper monoids are assumed to be passed to `reduce` and `scan`.

[2] Futhark is joint work with a number of researchers @ DIKU, including Troels Henriksen, Cosmin Oancea, Fritz Henglein, Ken Friis Larsen, and Philip Munksgaard.

Futhark, the Language (II)

- Some first-order functions (also with parallel semantics):

```
val rotate       [n] 'a : i64 → [n]a → [n]a
val iota                : (n:i64) → [n]i64    -- may fail
val indices      [n] 'a : [n]a → [n]i64
val zip      [n] 'a 'b : [n]a → [n]b → [n](a,b)
val unzip    [n] 'a 'b : [n](a,b) → ([n]a,[n]b)
```

- Notice that `iota` is given a dependent type! When not passed a variable or a constant, a "fresh" existential size variable is substituted for `n` in the result type.

- All **tuples are eliminated** through an array-of-structs to struct-of-arrays transformation.

- **Ex:** Inhabitants of the type `?[x].([x]i64,[x](bool,i64))` are pairs of **equally sized arrays**, where the first array contains integers and the second array contains pairs of a boolean and an integer.

Example: Matrix-Multiplication in Futhark

```
let matmul [n][m][p] (a:[n][m]f64) (b:[m][p]f64) : [n][p]f64 =
  map (\arow →
        map (\bcol → reduce (+) 0 (map2 (*) arow bcol))
            (transpose b)
      ) a
```

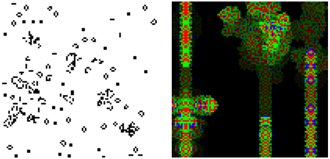
Notice:

- Futhark can assume **compatibility of array dimensions** and generate boundary check-free code.
- When calling `matmul`, Futhark must be able to establish that the array sizes match.
- The programmer may insert *type constraints* (`:>`) for which sizes are checked dynamically.

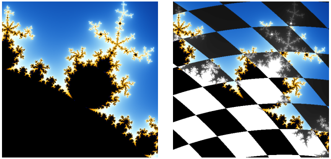
  Such type constraints are rarely needed and when they are, they are explicit (44k lines of benchmark code contains 66 size constraints, mostly in pre- and post-processing code).

Futhark, the Compiler (I)

- Supports **higher-order modules**, which are eliminated at compile time.
- Supports a restricted notion of **higher-order functions**, which are eliminated at compile time. (*functions may not appear in arrays or returned by branches of conditionals*)
- **Other features:** Open source, easy to download and use, used for educational and research purposes, package management... See `http://futhark-lang.org`...



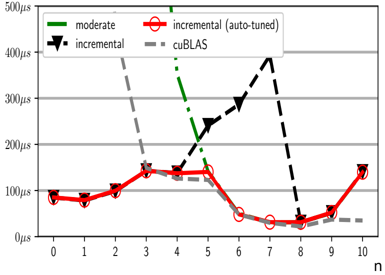
Modularised variants of Conway's Game of Life.



Functional images—Mandelbrot merged with skewed chess board.

Futhark, the Compiler (II)

- Targets a series of architectures, including **GPUs** and **CPUs** through CUDA, OpenCL, C, and PyOpenCL.
- Supports nested regular parallelism and generates **multi-versioned** code using a series of aggressive techniques for **fusion**, **flattening**, and **tiling**.



Autotuned Futhark code for multiplying a $2^n \times 2^{(20-2n)}$ matrix with its transposition.

- Irregular nested parallelism must be flattened manually by the user, for instance using segment arrays. Higher-order library functions encapsulate **certain patterns of irregular flattening**.

Type Soundness

A Type System for Type-Dependent Types

The implementation is based on a type system for a mini Futhark language, called *F*:

- *F* extends the simply-typed lambda calculus with pairs, conditionals, and **array contructs**.
- It also features a `let`-construct, which supports "**explicit opening**" of existential types *(making it possible for code to refer to sizes)*.
- *F* also features "**implicit opening**" of existential types.
- We assume expressions such as `map f (filter p s)` have been expanded into `let z = filter p s in map f z`.

Types and Expressions

d	$::=$		**Size sorts**
	$\mid$	n	constant
	$\mid$	x	variable
τ	$::=$		**Basic types**
	$\mid$	$\texttt{i64}$	integer
	$\mid$	$\texttt{bool}$	boolean
	$\mid$	(τ, τ')	pair
	$\mid$	$[d]\tau$	array
	$\mid$	$(x : \tau) \rightarrow \mu$	function
μ	$::=$		**Return types**
	$\mid$	$\exists x.\mu$	existential size
	$\mid$	τ	basic type
σ	$::=$		**Type schemes**
	$\mid$	τ	basic type
	$\mid$	$\star$	abstract size type

e	$::=$		**Expressions**
	$\mid$	n	integer
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	x	variable
	$\mid$	$\lambda(x : \tau).e$	function
	$\mid$	$e\ e$	application
	$\mid$	$[e, \cdots, e]$	array
	$\mid$	$\texttt{iota}\ e$	index array
	$\mid$	$e[e]$	array index
	$\mid$	(e, e)	pair
	$\mid$	$\texttt{fst}\ e \mid \texttt{snd}\ e$	projection
	$\mid$	$\textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$	conditional
	$\mid$	$e \triangleright \tau$	dynamic check
	$\mid$	$\textbf{let}\ x = e\ \textbf{in}\ e$	let
	$\mid$	$\textbf{let}\ [\bar{x}]\ x = e\ \textbf{in}\ e$	let-sz

Type Soundness (I)

We shall only give a brief account of type soundness [ARRAY'21].

- Contexts (Γ) map variables to type schemes (σ).
- A type scheme $\star$ indicates an *implicit size variable*, which may not be referenced by program expressions.
- Typing rules for [T-LET] (implicit-opening) and [T-LET-SZ] (explicit-opening):

$$\boxed{\Gamma \vdash e : \mu}$$

$$\frac{\Gamma \vdash e : \exists \bar{x}.\tau \quad \Gamma, \bar{x} : \bar{\star}, x : \tau \vdash e' : \mu}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \exists \bar{x} x.\mu}\ \text{[T-LET]}$$

$$\frac{\Gamma \vdash e : \exists \bar{x}.\tau \quad \Gamma \vdash \exists \bar{x}.\tau\ \underline{\mathbf{ok}} \quad \Gamma, \bar{x} : \overline{\mathtt{i64}}, x : \tau \vdash e' : \mu}{\Gamma \vdash \mathbf{let}\ [\bar{x}]\ x : \tau = e\ \mathbf{in}\ e' : \exists \bar{x} x.\mu}\ \text{[T-LET-SZ]}$$

...

Type Soundness (II)

Values

$$v \quad ::= \quad n \mid \texttt{true} \mid \texttt{false} \mid \langle x, e, \rho \rangle \mid [v, \cdots, v] \mid (v, v)$$

Dynamic semantics

$$\boxed{\rho \vdash e \rightsquigarrow v}$$

$$\frac{\rho \vdash e \rightsquigarrow v \quad \rho, x : v \vdash e' \rightsquigarrow v'}{\rho \vdash \textbf{let } x = e \textbf{ in } e' \rightsquigarrow v'} \text{[D-LET]}$$

$$\frac{\rho \vdash e \rightsquigarrow v \quad \tau \vdash_{\bar{x}} v \rightsquigarrow \bar{n} \quad \rho, \bar{x} : \bar{n}, x : v \vdash e' \rightsquigarrow v'}{\rho \vdash \textbf{let } [\bar{x}]\ x : \tau = e \textbf{ in } e' \rightsquigarrow v'} \text{[D-LET-M]}$$

$$\ldots$$

Type Soundness (III)

Proposition ((Partial) Type Soundness)[†]
If $\Gamma \vdash e : \mu$ *and* $\delta \models \rho : \Gamma$ *and* $\rho \vdash e \leadsto v$ *then* $\delta \models v : \mu$.

ρ	Dynamic environment (maps variables to values)
δ	Size environment (maps variables to sizes)
$\delta \models \_ : \_$	Logical proposition relating dynamics aspects with static aspects

[†] *By considering only value-terminating expressions, we avoid specifying dynamic evaluation rules for propagating dynamic errors!*

Different Ways Evaluation Can Go Wrong

1. Explicit array **index errors** ($e_1[e_2]$).
2. Type constraint ($e \triangleright \tau$) involving **array size mismatches**.
3. Type constraint ($e \triangleright \tau$) involving size variable that occurs only under a function type constructor (*can be disallowed statically*).
4. Explicit let-construct failing (i) due to the presence of an empty array, (ii) if an attempted extracted size name does not occur above a function type.

   (*Issue (ii) can be handled statically, issue (i) can be solved using dynamic shape vectors.*)

Size-Parametric Types

```
type sq [n] = [n][n]i64
```

Lifted types

Guarantees **array regularity** and **defunctionalisation**, while supporting **abstract types and type parameterisation**:

	Declaration	Constraint on k	Use of t disallowed in
Fully-lifted	`type^ t = k`		Array type, conditional type
Size-lifted	`type~ t = k`	No function types	Array type
Non-lifted	`type t = k`	No function types, no existentials	

Lifted type-parameters

```
type~ t '~a = (a, ?[n].[n]i64)
let f (b:bool) : t i64 = if b then (4,[3,5]) else (1,[8])
-- f : (b: bool) → ?[n].(i64, [n]i64)
```

Exotic Uses of Size-Dependent Types

1. A data-parallel tokeniser
2. Type-level size-computations for safe array deconstruction
3. Bounded naturals
4. Composition of neural network layers

A Data-Parallel Tokeniser

This tokeniser avoids the construction of irregular arrays!

```
module type tokenise = {
  type word [p]   -- word type carrying abstract origin witness
  val words [n] : [n]char → ?[p].(word [p] → ?[m].[m]char,
                                   ?[k].[k](word [p]))
}
```

```
module tokenise : tokenise = {
  let is_space (x: char) = x == ' '
  let f &&& g = \x → (f x, g x)
  type word [p] = ([p](), i64, i64)
  let words [n] (s: [n]char) =
    ( \(_, i, k) → #[unsafe] s[i:i+k]
    , segmented_scan (+) 0 (map is_space s)
       (map (\c → i64.bool(!(is_space c))) s)
      |> (id &&& rotate 1) |> uncurry zip |> zip (indices s)
      |> filter (\(_,(x,y)) → x>y) |> map (\(i,(x,_)) → ([],i-x+1,x))
    )
}
```

Type-Level Size-Computations for Safe Array Deconstruction

```
module catarr : {
  type S[n][m]    -- n+1=m
  val cons             'a [n] : a → [n]a → ?[m].([m]a, S[n][m])
  val decon         'a [n][m] : [m]a → S[n][m] → (a,[n]a)

  type P[n][m][k] -- n+m=k
  val concat        'a [n][m] : [n]a → [m]a → ?[k].([k]a, P[n][m][k])
  val split      'a [n][m][k] : [k]a → P[n][m][k] → ([n]a,[m]a)

  -- type-level operations
  val refl_P        [n][m][k] : P[n][m][k] → P[m][n][k]
  ...
} = ...
```

```
entry main (n:i64) (m:i64) : i64 =
  let xs = iota n
  let (zs,w) = catarr.concat xs (map (*10) (iota m))
  let w1 = catarr.refl_P w
  let (as,bs) = catarr.split (rotate 3 zs) w1
  in map2 (-) bs xs |> reduce (+) 0
```

Safe Array-Indexing with Bounded Naturals

```
module type natarr = {
  type nat[n]          -- bounded nat < n
  val toi64      [n] : nat[n]→i64
  val indices [n] 'a : [n]a→[n](nat[n])
  val sub     [n] 'a : [n]a→nat[n]→a
}
```

```
module type size = {
  type t[n]
  val mk : (n:i64)→t[n]
}
```

```
module example (na:natarr) = {
  let f [n] (xs:[n]f64) : f64 =
    let ns = na.indices xs
    in map2 (\i j → na.sub xs i + na.sub xs j)
            ns (rotate 1 ns) |> reduce (+) 0
}
```

```
module na : natarr = {
  type nat[n] = (i64, size.t [n])               -- carries bound witness
  let toi64 [m] (n:nat[m]) : i64 = n.0
  let indices 'a [n] (_ :[n]a) : [n](nat[n]) =
    map (\x → (x,size.mk n)) (iota n)
  let sub [n] 'a (arr:[n]a) (i:nat[n]) : a =
    #[unsafe] arr[i.0]                          -- unsafe array indexing
}
```

Composition of Neural Network Layers

Futhark size-types have been used for controling certain size-aspects of composing neural network layers:

```
type^ forwards  'i 'w 'o 'c = bool → w → i → (c, o)
type^ backwards 'c 'w 'ein 'eout '^u = bool → u → w → c → ein → (eout,w)
type^ NN 'input 'w 'output 'c 'e_in 'e_out '^u =
   { forward : (k: i64) → forwards ([k]input) w ([k]output) ([k]c),
     backward: (k: i64) → backwards ([k]c) w ([k]e_in) ([k]e_out) u,
     weights : w }

val connect_layers 'w1 'w2 'i1 'o1 'o2 'c1 'c2 'e1 'e2 'e22 '^u:
   NN i1 w1 o1 c1 e22 e1 u → NN o1 w2 o2 c2 e2 e22 u → NN i1 (w1,w2) o2 (c1,c2) e2 e1 u
```

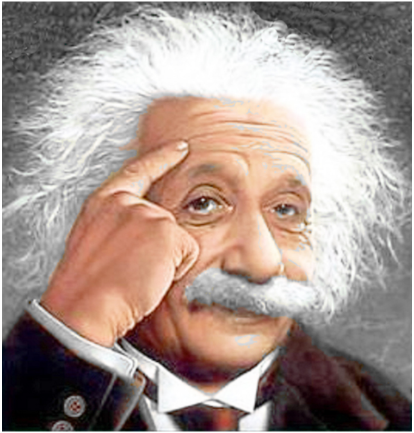
```
type std_weights [a][b][c] 't = ([a][b]t, [c]t)
type^ appgr2 'x 'y = (x, y) → (x, y) → (x, y)
type^ appgr3 't = (a:i64)→(b:i64)→appgr2 ([a][b]t) ([a]t)
type^ actfun 'o = {f:o → o, fd:o → o}
val dense: (m:i64) → (n:i64) → actfunc ([n]t) → i32 →
  NN ([m]t) (std_weights[n][m][n] t) ([n]t)
     ([m]t, [n]t) ([n]t) ([m]t) (appgr3 t)
```

Ex: MNist Convolutional Network†

```
module dl = deep_learning f32
let (>>) = dl.nn.connect_layers
let seed = 1
let nn =
  dl.layers.conv2d 1 28 28 5 1 32 24 24 dl.nn.relu seed
  >> dl.layers.max_pooling2d 32 24 24 12 12
  >> dl.layers.conv2d 32 12 12 3 1 64 10 10 dl.nn.relu seed
  >> dl.layers.max_pooling2d 64 10 10 5 5
  >> dl.layers.flatten 64 5 5 1600
  >> dl.layers.dense 1600 1024 (dl.nn.identity 1024) seed
  >> dl.layers.dense 1024 10 (dl.nn.identity 10) seed
```

† *For details, see https://github.com/HnimNart/deeplearning (extension of [FHPNC '19])*

Questions?



(Original)

(Stupid-Art in Futhark)