Lightweight Polytypic Staging: a new approach to Nested Data Parallelism in Scala

Alexander Slesarenko
Keldysh Institute of Applied Mathematics, 2012
The Domain - Nested Data Parallelism

- The original idea
  - Guy Blelloch, Gary Sabot: in the early 90’s ([1] is a good starting point)
  - NESL – proof of concept (first order, interpreted language)

- Generalizations (90’s – 00’s)
  - Chakravarty, Keller, S. P. Jones et al. (5 or 6 papers)
  - Data Parallel Haskell – higher-order, compiled language [2]
  - Language extension with special syntax

- A big promise but still in research

Motivation: NDP as an embedded DSL

- NDP is not a “silver bullet”
- Some applications fit to the model but others don’t
- For those that fit we want high-level declarative language
- IDEALLY: If it is expressible then it is automatically vectorizable (with asymptotic work-efficiency)
- Should interact with other DSLs and the host language
- Yet another tool in the Scala toolbox
## Polytypic DSL

<table>
<thead>
<tr>
<th>DSL</th>
<th>Monotypic (traditional)</th>
<th>Polytypic (data type generic)</th>
</tr>
</thead>
</table>
| Shallow embedding | ✓ Ordinary types and functions  
                  ✓ Execution on the JVM |                                |
| Deep embedding  |                                                             |                                |
## Polytypic DSL

<table>
<thead>
<tr>
<th>DSL</th>
<th>Monotypic (traditional)</th>
<th>Polytypic (data type generic)</th>
</tr>
</thead>
</table>
| Shallow embedding    | ✓ Ordinary types and functions  
 ✓ Execution on the JVM                                                                  |                                |
| Deep embedding       | ✓ Ordinary types and functions  
 ✓ Staging + transform  
 ✓ Execution on XXX by code generation                                                 |                                |
## Polytypic DSL

<table>
<thead>
<tr>
<th>DSL</th>
<th>Monotypic (traditional)</th>
<th>Polytypic (data type generic)</th>
</tr>
</thead>
</table>
| Shallow embedding  | ✓ Ordinary types and functions  
 ✓ Execution on the JVM                                                                 | ✓ Type-indexed types and functions  
 ✓ Execution on the JVM                                                                 |
| Deep embedding     | ✓ Ordinary types and functions  
 ✓ Staging + transform  
 ✓ Execution on XXX by code generation                                                |                                                     |
## Polytypic DSL

<table>
<thead>
<tr>
<th>DSL</th>
<th>Monotypic (traditional)</th>
<th>Polytypic (data type generic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shallow embedding</td>
<td>✓ Ordinary types and functions</td>
<td>✓ Type-indexed types and functions</td>
</tr>
<tr>
<td></td>
<td>✓ Execution on the JVM</td>
<td>✓ Execution on the JVM</td>
</tr>
<tr>
<td>Deep embedding</td>
<td>✓ Ordinary types and functions</td>
<td>✓ Type-indexed types and functions</td>
</tr>
<tr>
<td></td>
<td>✓ Staging + transform</td>
<td>✓ Staging + transform</td>
</tr>
<tr>
<td></td>
<td>✓ Execution on XXX by code generation</td>
<td>✓ Execution on XXX by code generation</td>
</tr>
</tbody>
</table>

- In the implementation we need “the best” of the two worlds
  - Type-indexed types from generic programming
  - Staged execution from deep embedding
## Polytypic DSL

<table>
<thead>
<tr>
<th>DSL</th>
<th>Monotypic (traditional)</th>
<th>Polytypic (data type generic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shallow embedding</td>
<td>✓ Ordinary types and functions</td>
<td>✓ Type-indexed types and functions</td>
</tr>
<tr>
<td></td>
<td>✓ Execution on the JVM</td>
<td>✓ Execution on the JVM</td>
</tr>
<tr>
<td>Deep embedding</td>
<td>✓ Ordinary types and functions</td>
<td>✓ Type-indexed types and functions</td>
</tr>
<tr>
<td></td>
<td>✓ Staging + transform</td>
<td>✓ Staging + transform</td>
</tr>
<tr>
<td></td>
<td>✓ Execution on XXX by code generation</td>
<td>✓ Execution by code generation</td>
</tr>
</tbody>
</table>

- In the implementation we need “the best” of the two worlds
  - Type-indexed types from generic programming
  - Staged execution from deep embedding

---

Lightweight Polytypic Staging: a new approach to Nested Data Parallelism in Scala
Framework: Polymorphic Embedding of DSLs

type Rep[A]   // abstract type constructor of representations

type PA[A] = Rep[PArray[A]]

trait PArray[A] {   // parallel array (to express parallelism)
  def length: Rep[Int]
  def map[R](f: Rep[A] => Rep[R]): PA[R]
  def zip[B](b: PA[B]): PA[(A,B)]
  ...
}

type Vector = PArray[Float]   // parallel array

def dotProduct(vec1: Rep[Vector], vec2: Rep[Vector]): Rep[Float] =
  sum((vec1 zip vec2) map { case Pair(v1,v2) => v1 * v2 })
Framework: Polymorphic Embedding of DSLs

```
type Rep[A]   // abstract type constructor of representations
  =

type PA[A] = Rep[PArray[A]]

trait PArray[A] { // parallel array (to express parallelism)
  def length: Rep[Int]
  def map[R](f: Rep[A] => Rep[R]): PA[R]
  def zip[B](b: PA[B]): PA[(A,B)]
  ...
}
```

```
type Vector = PArray[Float] // parallel array

def dotProduct(vec1: Rep[Vector], vec2: Rep[Vector]): Rep[Float] =
  sum((vec1 zip vec2) map { case Pair(v1,v2) => v1 * v2 })
```

<table>
<thead>
<tr>
<th>Shallow Embedding</th>
<th>Deep Embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Should be simple, good for testing and debugging</td>
<td>✓ Should generate an efficient code</td>
</tr>
</tbody>
</table>
The Key Idea – flattening transformation

- The data might be **irregular**
- ill-balanced and not very parallel at top level
- The one we want to write

- **regular** after flattening
- **Balanced** chunking
- The one we want to run
The Key Idea – flattening transformation

- The data might be **irregular**
- **ill-balanced and not very parallel** at top level
- The one we want to write

Sparse matrix

\[
\begin{pmatrix}
1.0 & 0 & 2.0 & 0 \\
3.0 & 4.0 & 5.0 & 0 \\
0 & 0 & 0 & 6.0
\end{pmatrix}
\]

Compressed row format

\[
(0, 1.0) (2, 2.0) \\
(0, 3.0) (1, 4.0) (2, 5.0) \\
(3, 6.0)
\]

- **regular** after flattening
- **Balanced** chunking
- The one we want to run
The Key Idea – flattening transformation

- The data might be *irregular*
- ill-balanced and not very parallel at top level

The one we want to write

<table>
<thead>
<tr>
<th>Sparse matrix</th>
<th>Compressed row format</th>
</tr>
</thead>
</table>
| \[
| \begin{pmatrix}
| 1.0 & 0 & 2.0 & 0 \\
| 3.0 & 4.0 & 5.0 & 0 \\
| 0 & 0 & 0 & 6.0 \\
| \end{pmatrix}
| \] |
| \[
| \begin{pmatrix}
| (0, 1.0) & (2, 2.0) \\
| (0, 3.0) & (1, 4.0) & (2, 5.0) \\
| (3, 6.0) \\
| \end{pmatrix}
| \] |

- regular after flattening
- **Balanced** chunking

The one we want to run

**Automatic Flattening**

Segment descriptors

- PU1
- PU2
- PU3

Sparse matrix

Compressed row format

Flat representation
### How flattening happens

<table>
<thead>
<tr>
<th>Nested Code</th>
<th>Flattened Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>p: A =&gt; B // primitive</td>
<td><strong>type</strong> PA[A] = PArray[A]</td>
</tr>
<tr>
<td><strong>def</strong> g(as: PA[A]) = as <strong>map</strong> p</td>
<td><strong>def</strong> g(as: PA[A]) = p^ (as)</td>
</tr>
<tr>
<td><strong>def</strong> h(m: PA[PA[A]]) = m <strong>map</strong> g</td>
<td><strong>def</strong> h(m: PA[PA[A]]) = m <strong>map</strong> p^ // inline g</td>
</tr>
<tr>
<td></td>
<td>= p^^ (m) // ???</td>
</tr>
</tbody>
</table>
# How flattening happens

<table>
<thead>
<tr>
<th>Nested Code</th>
<th>Flattened Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>p: A =&gt; B   // primitive</td>
<td>type PA[A] = PArray[A]</td>
</tr>
<tr>
<td>def g(as: PA[A]) = as map p</td>
<td>def g(as: PA[A]) = p^(as)</td>
</tr>
<tr>
<td>def h(m: PA[PA[A]]) = m map g</td>
<td>def h(m: PA[PA[A]]) = m map p // inline g</td>
</tr>
<tr>
<td>= p^(m) // ???</td>
<td></td>
</tr>
</tbody>
</table>

## The key insight (we don’t need p^^)

```scala
def p^^(m: PA[PA[A]]): PA[PA[A]] = unconcat(m, p^(concat(m)))
def concat[A](nested: PA[PA[A]]): PA[A] 
def unconcat[A,B](shape: PA[PA[A]], values: PA[B]): PA[PA[B]]
```
Data structures that support flattening

PA[Unit]
UnitArray

PA[T] where T – base type
BaseArray[Int]

arr → 7 1 5 4

Structure nodes as case classes
Data structures that support flattening

**Constant time operations**

```scala
def zip[A,B](a: PA[A], b: PA[B]): PA[(A,B)] = PairArray(a, b)
```

**Structure nodes as case classes**

PA[Unit]

UnitArray

len

10

PA[(A,B)]

PairArray[A,B]

a

b

PA[A]

PA[B]

PA[T] where T – base type

BaseArray[Int]

arr

7 1 5 4

Lightweight Polytypic Staging: a new approach to Nested Data Parallelism in Scala
Data structures that support flattening

- `PA[Unit]`: UnitArray
- `PA[T]` where `T` – base type: BaseArray[Int]
- `PA[(A,B)]`: PairArray[A,B]
- `PA[PA[A]]`: NArray[A]

**Constant time operations**

```scala
def zip[A,B](a:PA[A], b:PA[B]):PA[(A,B)]
= PairArray(a, b)
```

Structure nodes as case classes
Data structures that support flattening

PA[Unit]
UnitArray
len→10

PA[T] where T – base type
BaseArray[Int]
arr→7 1 5 4

PA[(A,B)]
PairArray[A,B]
a→PA[A]
b→PA[B]

PA[PA[A]]
NArray[A]
segs→BaseArray[Int]
arr→4 2 2

Constant time operations

def zip[A,B](a:PA[A], b:PA[B]):PA[(A,B)] = PairArray(a, b)
def concat[A](na: PA[PA[A]]): PA[A] = na match { case NArray(vs,_) => vs }
def unconcat[A,B](shape:PA[PA[A]], vs:PA[B]): PA[PA[B]] = shape match {
  case NArray(_,segs) => NArray(vs,segs)
}
def p^(m: PA[PA[A]]): PA[PA[A]] = unconcat(m, p^(concat(m)))

Structure nodes as case classes
Example (application specific types)

In the DSL we can construct types

\[ T = \text{Unit} \mid \text{Int} \mid \text{Float} \mid \text{Boolean} \quad \text{// base types} \]
\[ \mid (T_1, T_2) \quad \text{// product of types} \]
\[ \mid (T_1 + T_2) \quad \text{// sum of types} \]
\[ \mid \text{PArray}[T] \quad \text{// nested array} \]

\[
\begin{pmatrix}
(0, 1.0) & (2, 2.0) \\
(0, 3.0) & (1, 4.0) & (2, 5.0) \\
(3, 6.0)
\end{pmatrix}
\]

\[
\begin{pmatrix}
1.0 & 0 & 2.0 & 0 \\
3.0 & 4.0 & 5.0 & 0 \\
0 & 0 & 0 & 6.0
\end{pmatrix}
\times
\begin{pmatrix}
1.0 \\
2.0 \\
3.0 \\
4.0
\end{pmatrix}
= 
\begin{pmatrix}
7.0 \\
26 \\
24
\end{pmatrix}
\]

// Matrix in compressed row format

type SVector = \text{PArray}\[(\text{Int}, \text{Float})]\quad // \text{parallel array of products}
type SMatrix = \text{PArray}[\text{SVector}] \quad // \text{nested array of rows}
type Vector = \text{PArray}[\text{Float}] \quad // \text{dense vector}
### Example (Sparse Matrix Vector Multiplication)

$$
\begin{pmatrix}
1.0 & 0 & 2.0 & 0 \\
3.0 & 4.0 & 5.0 & 0 \\
0 & 0 & 0 & 6.0
\end{pmatrix}
\times
\begin{pmatrix}
1.0 \\
2.0 \\
3.0 \\
4.0
\end{pmatrix}
= 
\begin{pmatrix}
7.0 \\
26 \\
24
\end{pmatrix}
$$

// Matrix in compressed row format

```scala
type SVector = PArray[(Int,Float)] // parallel array of products
type SMMatrix = PArray[SVector] // nested array of rows
type Vector = PArray[Float] // dense vector
```

```scala
def sparseVectorMul(sv: SVector, vec: Vector): Float = 
  sum(sv map { case (i,v) => vec(i) * v })
```

```scala
def smvm(matr: SMMatrix, vec: Vector): Vector = 
  for (row <- matr) 
    yield sparseVectorMul(row, vec)
```

Inner parallelism

Outer parallelism
def sparseVectorMul(sv: SVector, vec: Vector): Float = 
  sum(sv map { case (i,v) => vec(i) * v })
def matrixVectorMul(matr: SMatrix, vec: Vector) = 
  for (row <- matr) yield sparseVectorMul(row, vec)

SVector = PA[(Int, Float)]
SMatrix = PA[SVector]
Vector = PA[Float]
Polytypic Staging

- Uses generic programming to capture domain semantics
- Allows the flattening of the DSL code by staged execution
- Is based on practical approaches: Scala-virtualized compiler, Polymorphic Embedding and LMS
- Not limited to NDP domain
Conclusions

- Nested data parallelism can be implemented in Scala as an embedded polytypic DSL
- To support flattening we need both staging and type-indexed data types
- Lightweight Polytypic Staging (LPS) is a framework for embedding of polytypic DSLs

- Nested Data Parallelism is a “killer app” for LPS

- WANTED: other polytypic domains?
References


Q&A

???