Lightweight Polytypic Staging of DSLs in Scala

Alexander Slesarenko
Keldysh Institute of Applied Mathematics, 2012

What will be discussed?

A **Domain Specific Language (DSL)** is a computer programming language of limited expressiveness focused on a particular domain. (Martin Fowler)

Nested data parallelism is our domain. The idea of nesting is to take a parallel function and apply it over multiple instances in parallel. (Guy Blelloch)

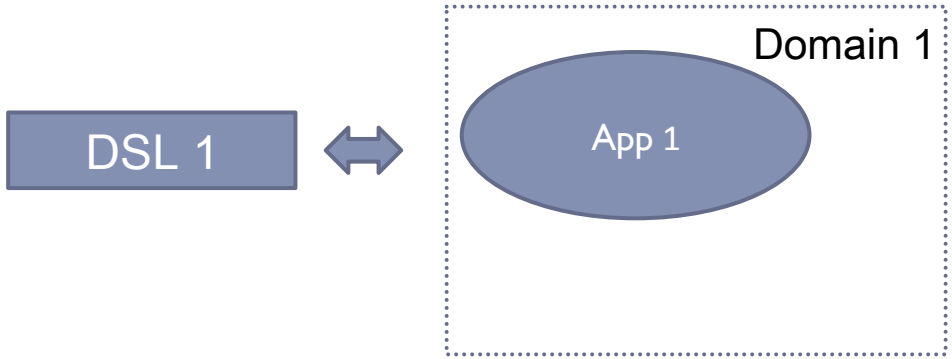
A **data-generic (or polytypic) function** is a function that can be instantiated on many data types to obtain data type specific functionality, it works for a whole family of types. (Ralf Hinze)

Staging is a program transformation that involves reorganizing the program's execution into stages. (Jørring, U.,Scherlis, W. L. Compilers and staging transformations.1986, Walid Taha.)

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. (scala-lang.org)

Polytypic Staging is a lightweight approach to introduce staging in the context of embedded DSL with polytypic meta-level.

Domain Specific Languages



Domain Specific Languages

External DSL

Specialized program

GraphViz | TeX | Yacc | …

Internal DSL

Embedded in **Haskell** as the host language

Parsec

Shallow (take everything from the host)

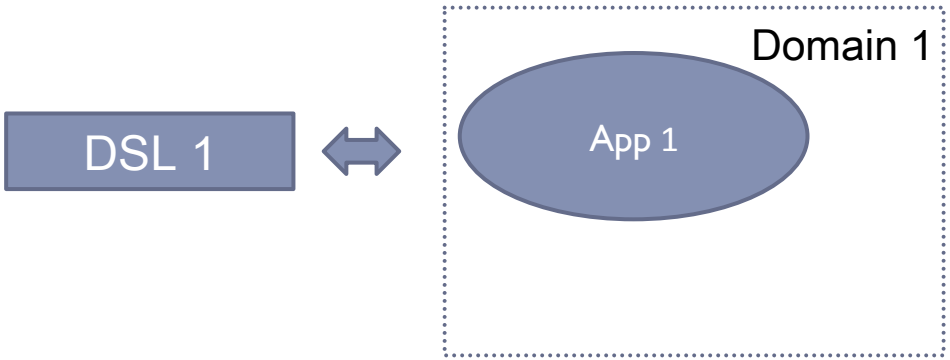
Embedded in **C#**

LINQ

Deep (take only front-end)

Language Extension

Data Parallel Haskell



Domain Specific Languages

External DSL

Specialized program

GraphViz | TeX | Yacc | …

Internal DSL

Embedded in **Haskell** as the host language

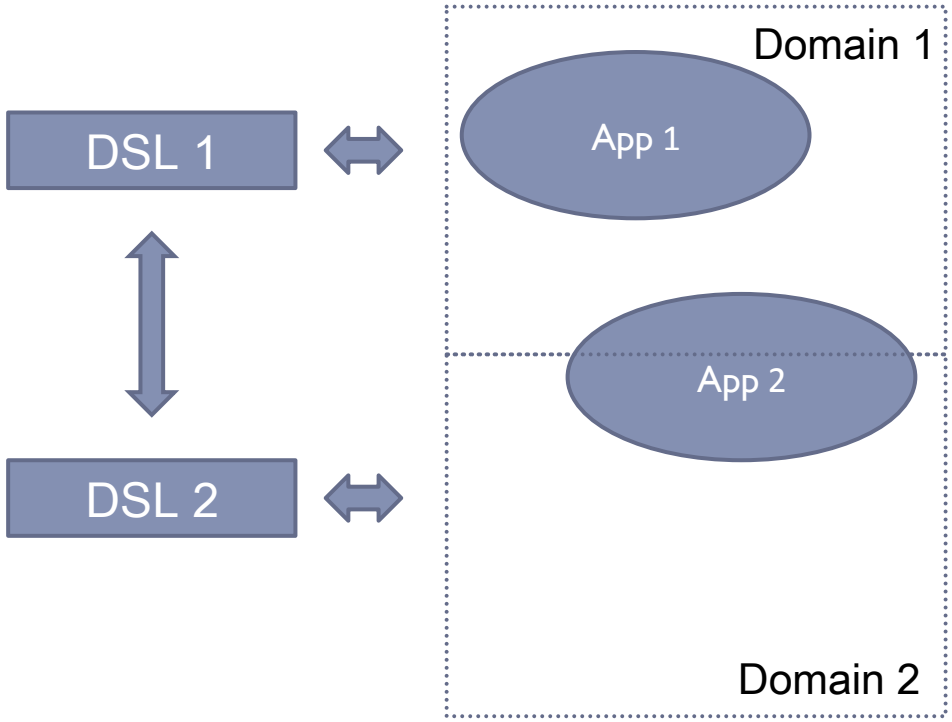
Parsec

Shallow (take everything from the host)

Embedded in **C#**

LINQ

Deep (take only front-end)



Domain Specific Languages

External DSL

Specialized program

GraphViz ⇔ TeX

Yacc …

Internal DSL

Embedded in **Haskell** as the host language

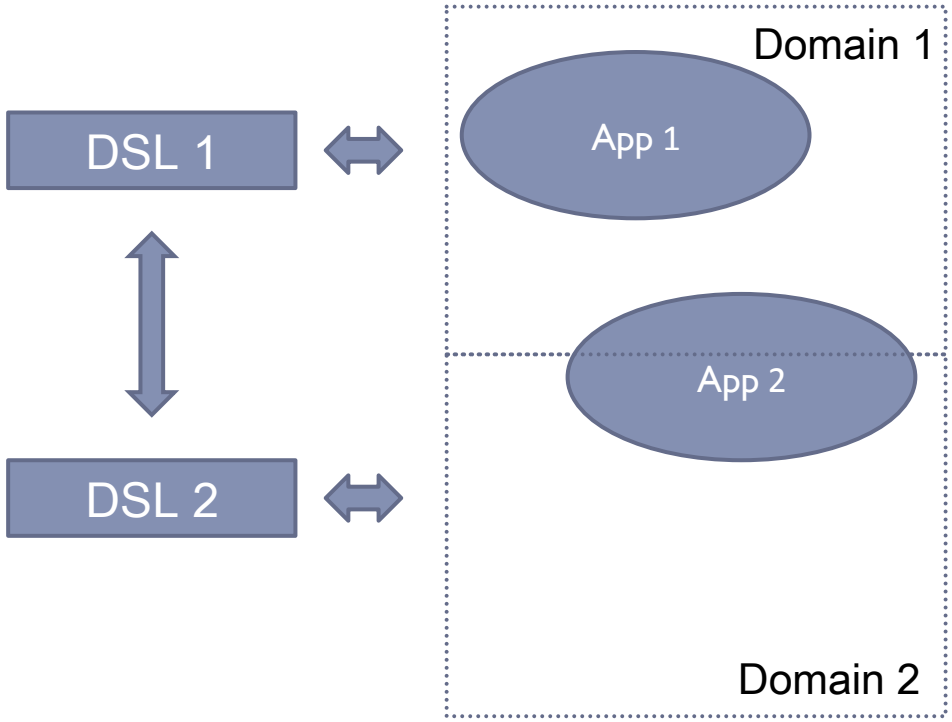
Parsec

Shallow (take everything from the host)

Embedded in **C#**

LINQ

Deep (take only front-end)



Domain Specific Languages

External DSL

Specialized program

GraphViz ⇔ TeX

Yacc …

Internal DSL

Embedded in **Haskell** as the host language

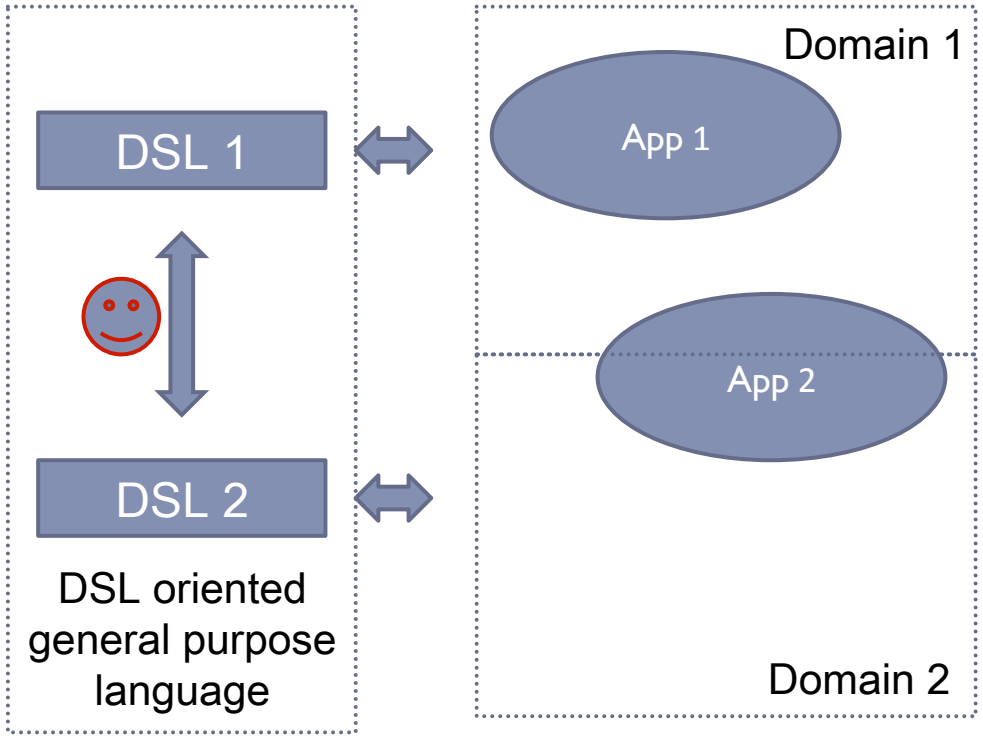
Parsec

Shallow (take everything from the host)

LINQ

Embedded in **C#**

Deep (take only front-end)



Domain Specific Languages

External DSL

Specialized program

GraphViz ⇔ TeX

Yacc …

Internal DSL

Embedded in **Scala** as the host language

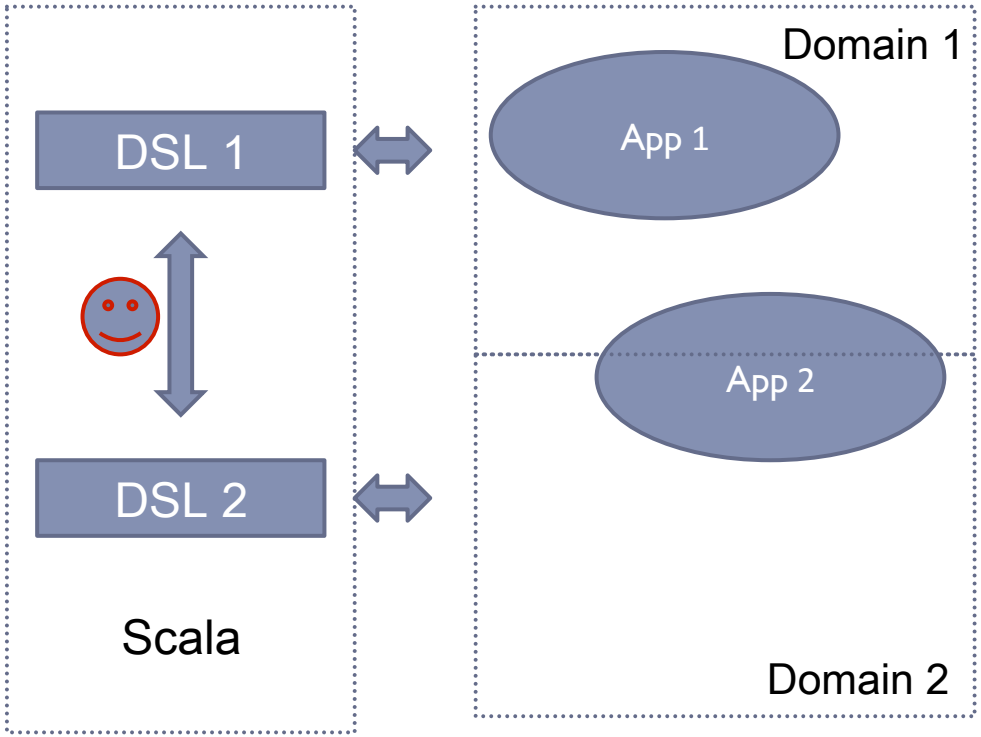
Parsers

Shallow (take everything from the host)

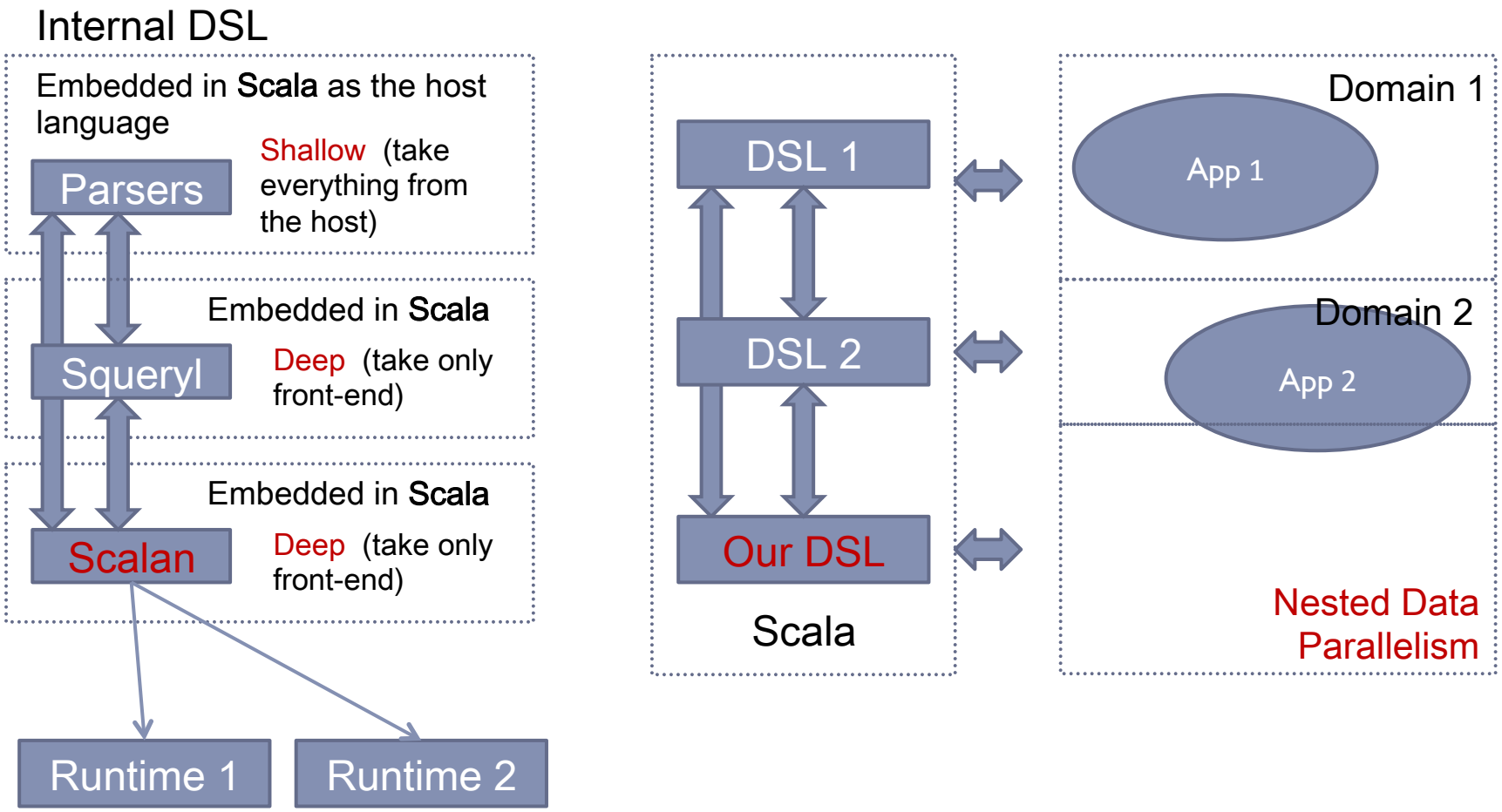
Embedded in **Scala**

Squeryl

Deep (take only front-end)

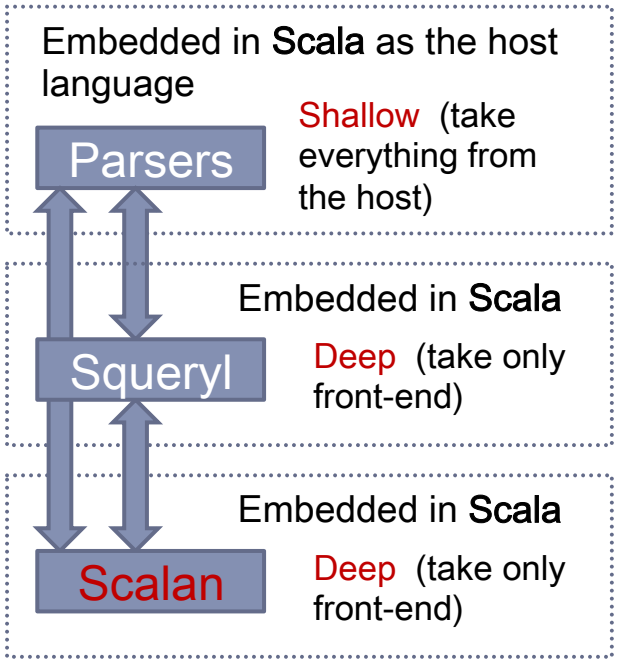


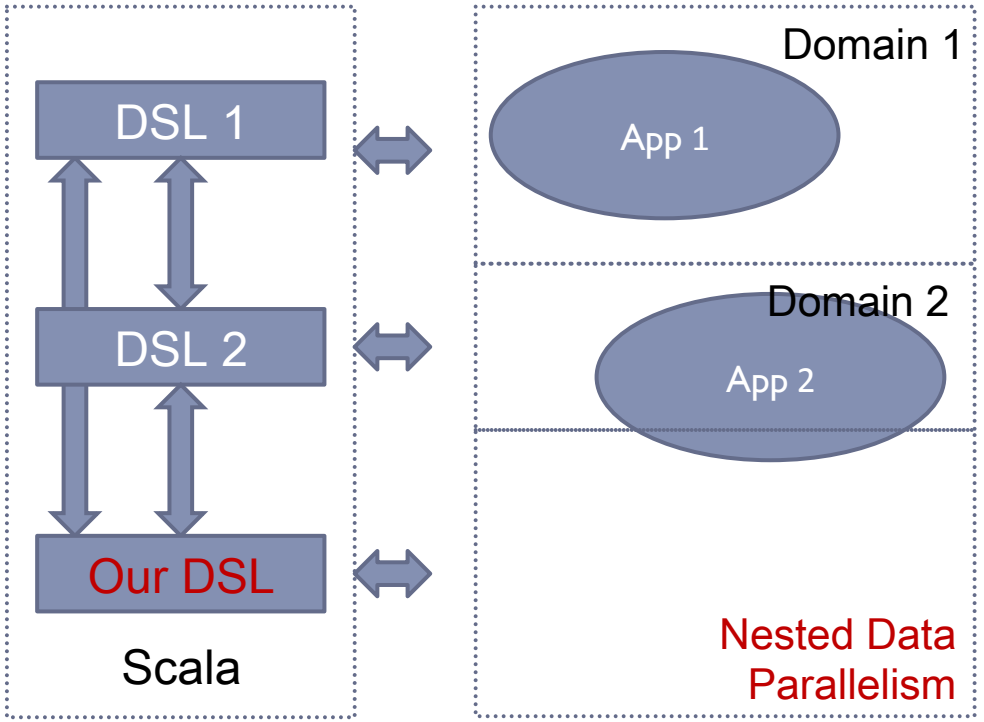
Domain Specific Languages



Domain Specific Languages

Internal DSL





High level code directly expresses domain semantics

```
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 })
```

Is it practical?

LINQ is most popular embedded DSL on .NET

```
IQueryable<Tuple<string, int>> selected =
  from c in db.Customers where c.City == “Moscow”
  from o in c.Orders where o.Amount > 1000000
  select new Tuple(c.Name, o.OrderNo)
```

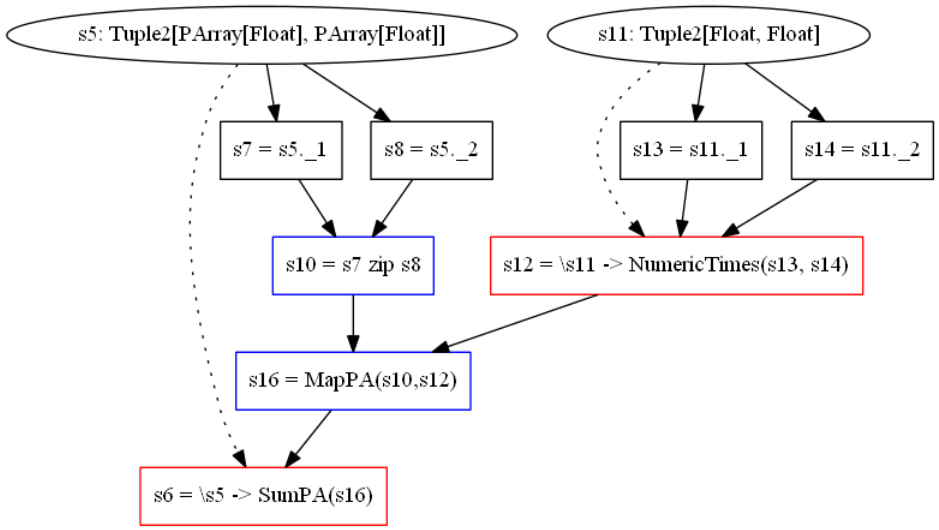
This is desugared into

```
IQueryable<Tuple<string, int>> selected =
  db.Customers.Where(c => c.City == “Moscow”)
    .SelectMany(c =>
       c.Orders.Where(o => o.Amount > 1000000)
        .Select(o => new Tuple(c.Name, o.OrderNo)))
```

```
IQueryable<T> Where<T>(this IQueryable<T> xs, Expr<Func<T,bool>> p)
IQueryable<R> SelectMany<T,R>(
  this IQueryable<T> source, Expr<Func<T, IEnumerable<R>>> selector)
```

Staged evaluation by deep DSL embedding

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



Optimizations for free

- Global Value Numbering (GVN)
- Copy Propagation
- Constant Propagation
- Common Subexpression Elimination (CSE)
- Dead Code Elimination (DCE)
- Domain-specific Rewrites based on properties of domain operations

Our DSL

```
type Rep[A]    // abstract type constructor of representations
trait PArray[A] {    // parallel array (to express parallelism)
  def length: Rep[Int]
  def map[R](f: Rep[A => R]): Rep[PArray[R]]
  def zip[B](b: Rep[PArray[B]]): Rep[PArray[(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 })
```

Our DSL

```
type Rep[A]    // abstract type constructor of representations
trait PArray[A] {    // parallel array (to express parallelism)
  def length: Rep[Int]
  def map[R](f: Rep[A => R]): Rep[PArray[R]]
  def zip[B](b: Rep[PArray[B]]): Rep[PArray[(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 })
```

Shallow Embedding	Deep Embedding
`type Rep[A] = A` ✓Should be simple, good for testing and debugging	`type Rep[A] = Exp[A]` ✓Should generate an efficient code

Our DSL

```
type Rep[A]    // abstract type constructor of representations
trait PArray[A] {    // parallel array (to express parallelism)
  def length: Rep[Int]
  def map[R](f: Rep[A => R]): Rep[PArray[R]]
  def zip[B](b: Rep[PArray[B]]): Rep[PArray[(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 })
```

Shallow Embedding	Deep Embedding
`type Rep[A] = A`	`type Rep[A] = Exp[A]`
✓Should be simple, for testing and debug	generate an code

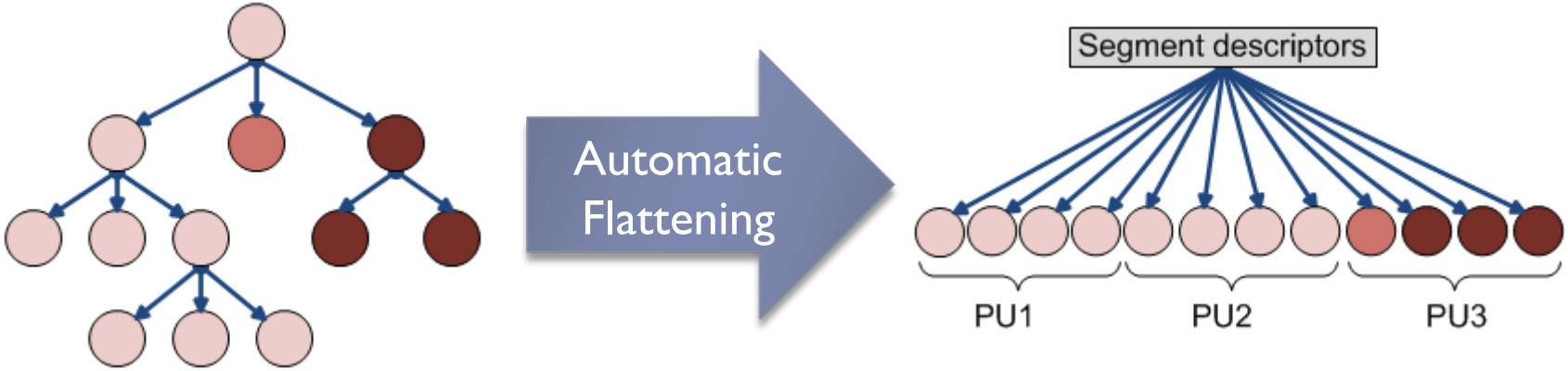
Should have equivalent semantics

The Domain: Nested Data Parallelism

- Original idea
  - Guy Blelloch, Gary Sabot: in the early 90's ([1] is a good starting point)
  - NESL – proof of concept and performance (first order, interpreted, functional language with list comprehensions)
- Generalizations (90's – 00's)
  - Chakravarty, Keller, S. P. Jones et al. (5 or 6 papers)
  - Data Parallel Haskell – higher-order, compiled language [2]
  - Haskell extension with special syntax
- A big promise but still in research

[1] Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, Cambridge, MA, USA, 1990
[2] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. *Harnessing the Multicores: Nested Data Parallelism in Haskell*, 2008

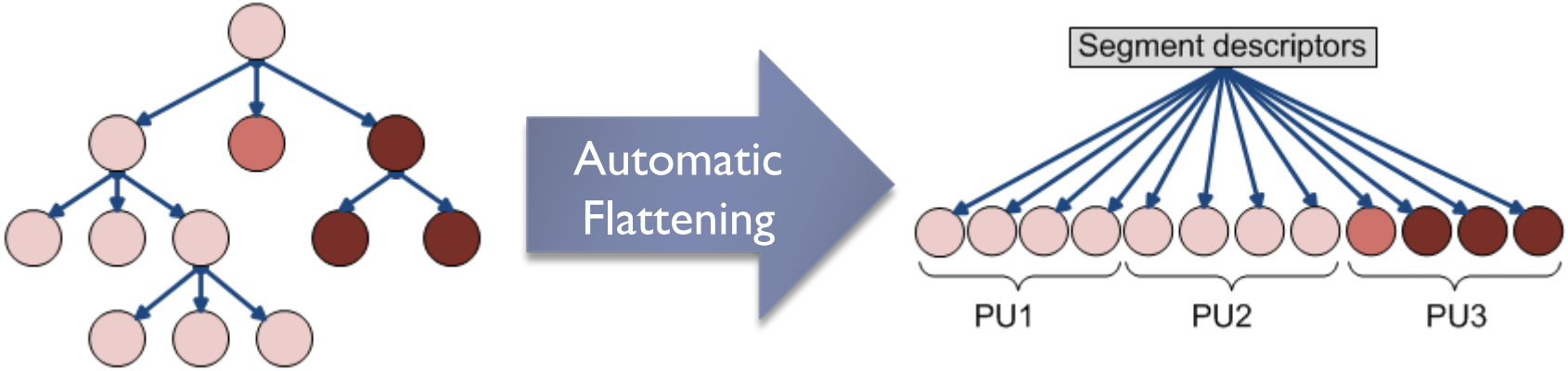
The Key Idea – flattening transformation



- 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



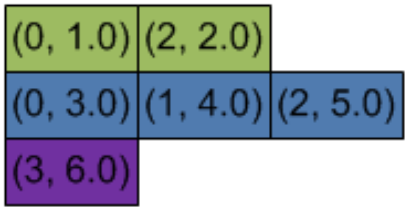
- The data might be irregular
- ill-balanced and not very parallel at top level
- The one we want to write

- regular after flattening
- Balanced chunking
- The one we want to run

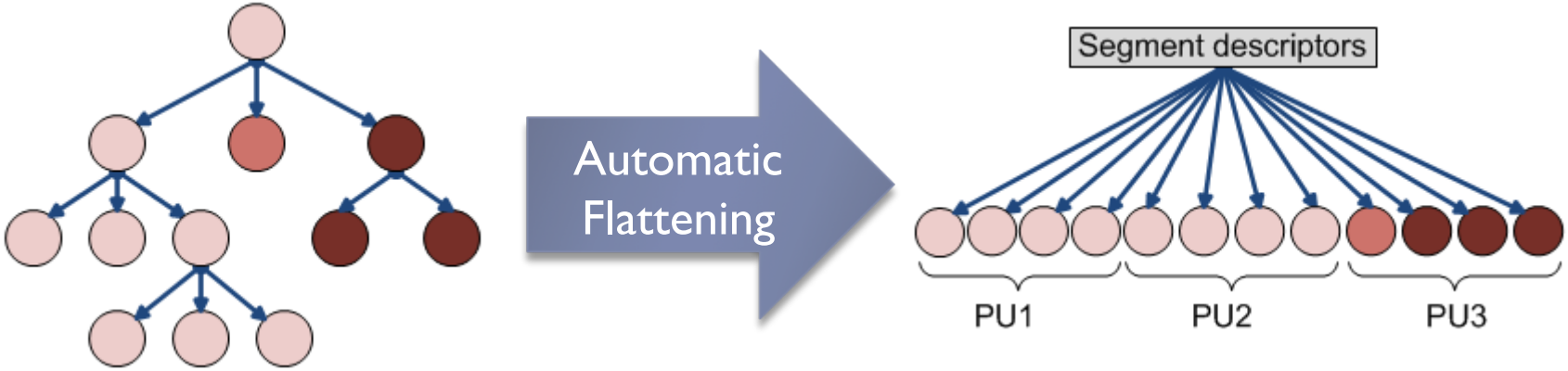
Sparse matrix

	0	1	2	3
0	1.0	0	2.0	0
1	3.0	4.0	5.0	0
2	0	0	0	6.0

Compressed row format



The Key Idea – flattening transformation



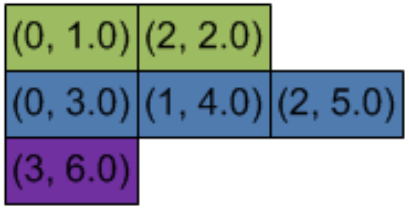
- The data might be irregular
- ill-balanced and not very parallel at top level
- The one we want to write

- regular after flattening
- Balanced chunking
- The one we want to run

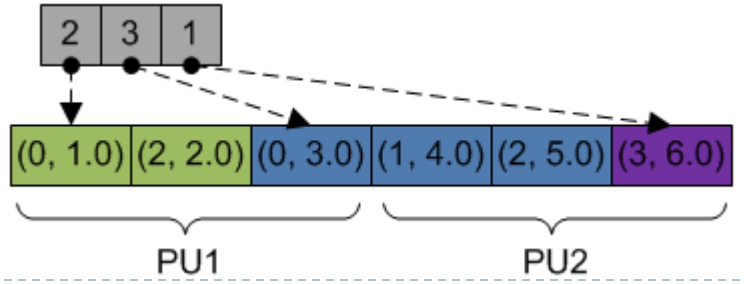
Sparse matrix

	0	1	2	3
0	1.0	0	2.0	0
1	3.0	4.0	5.0	0
2	0	0	0	6.0

Compressed row format



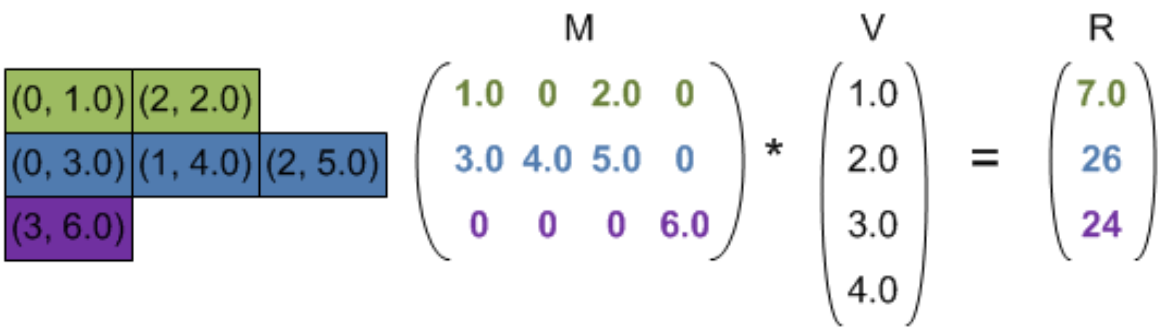
Flat representation



DSL example (application specific types)

In the DSL we can construct types

```
T = Unit | Int | Float | Boolean // base types
  | (T1,T2)                     // product of types
  | (T1+T2)                     // sum of types
  | PArray[T]                   // nested array
```




```
// Matrix in compressed row format (sparse matrix)
type SVector = PArray[(Int,Float)] // parallel array of products
type SMatrix = PArray[SVector]     // nested array of rows
type Vector = PArray[Float]        // dense vector
```

DSL Example (Sparse Matrix Vector Multiplication)


```
// Matrix in compressed row format
type SVector = PArray[(Int,Float)] // parallel array of pairs
type SMatrix = PArray[SVector]     // nested array of rows
type Vector = PArray[Float]        // dense vector
```

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

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

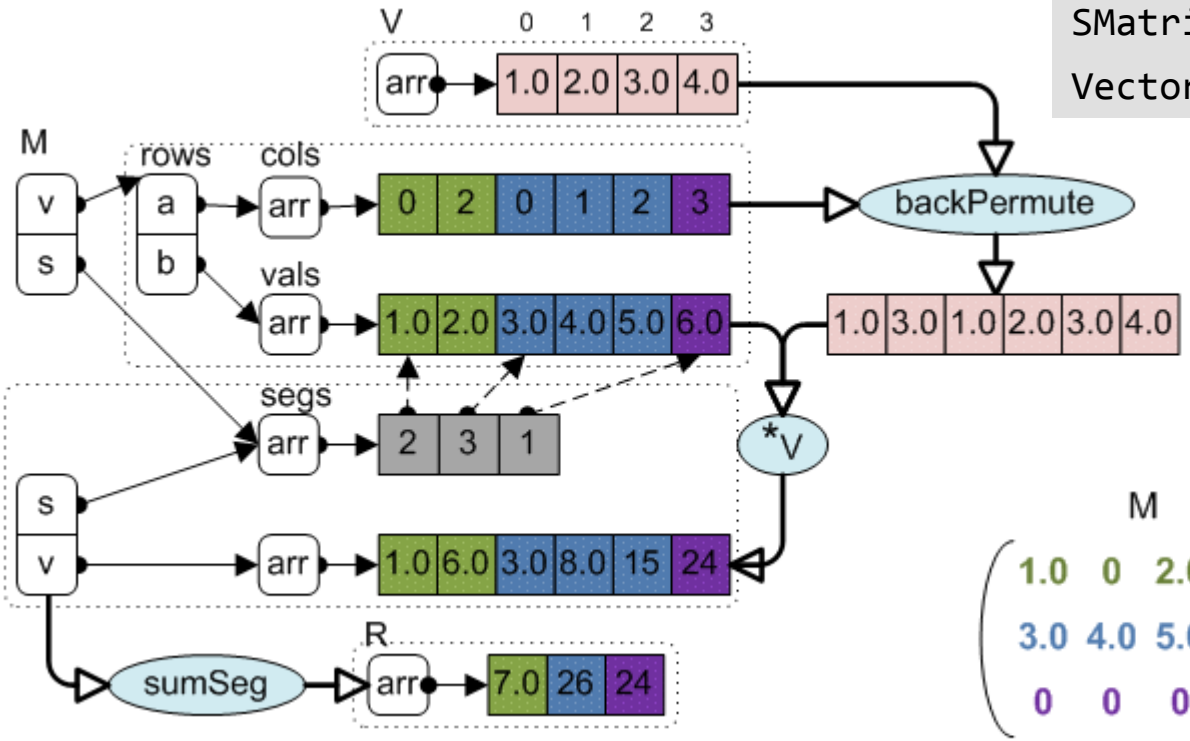
Inner parallelism

Outer parallelism

SMVM vectorized

```
def matrixVectorMul(m: SMatrix, v: Vector) = {
  val (cols,vals) = unzip(m.values)
  val products = backPermute(v, cols) |*| vals
  sumSeg(unconcat(products, m.segments))
}
```

```
SVector = PArray[(Int,Float)]
SMatrix = PArray[SVector]
Vector  = PArray[Float]
```



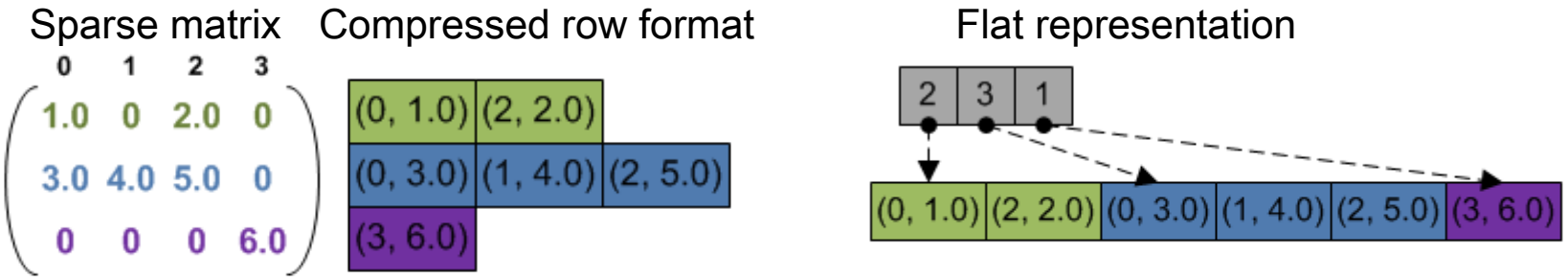
$$\underset{M}{\begin{pmatrix} 1.0 & 0 & 2.0 & 0 \\ 3.0 & 4.0 & 5.0 & 0 \\ 0 & 0 & 0 & 6.0 \end{pmatrix}} * \underset{V}{\begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{pmatrix}} = \underset{R}{\begin{pmatrix} 7.0 \\ 26 \\ 24 \end{pmatrix}}$$

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 a high-level declarative language (DSL)
- GOAL: **If a task is expressible in the DSL then it is automatically vectorizable and efficiently executable** (with asymptotic work-efficiency)
- Our DSL should interoperate with other DSLs and the host language
- Yet another tool in our toolbox

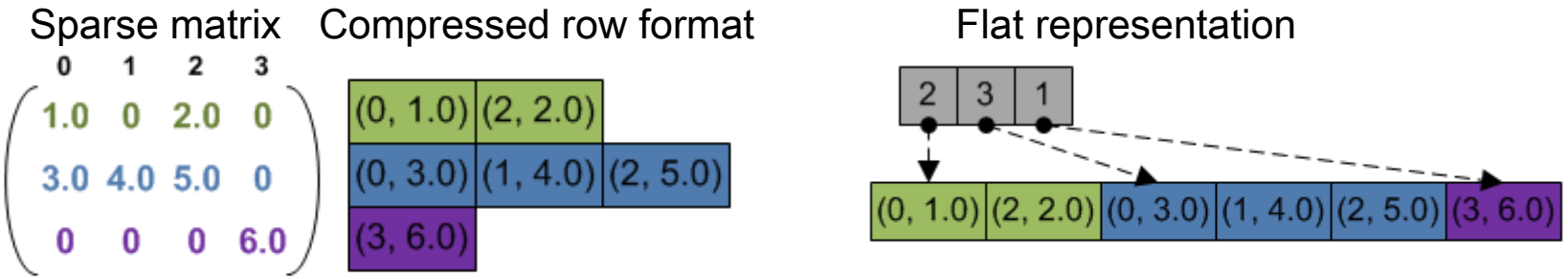
NDP domain is polytypic

- To support flattening transformation we need special data structures with internal representation that is different from a high level view



NDP domain is polytypic

- To support flattening transformation we need special data structures which internal representation is different from a high level view



- We want in the DSL to define application specific types

```
type SVector = PArray[(Int,Float)] // parallel array of products
type SMatrix = PArray[SVector]     // nested array of rows
```

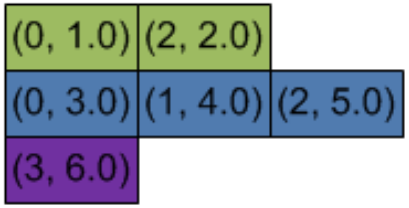
NDP domain is polytypic

- To support flattening transformation we need special data structures which internal representation is different from a high level view

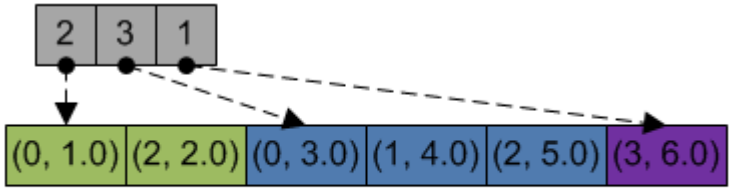
Sparse matrix

	0	1	2	3
	1.0	0	2.0	0
	3.0	4.0	5.0	0
	0	0	0	6.0

Compressed row format



Flat representation



- We want in the DSL to define application specific types

```
type SVector = PArray[(Int,Float)] // parallel array of products
type SMatrix = PArray[SVector]     // nested array of rows
```

- We want our implementation to work for any type in the family

```
T = Unit | Int | Float | Boolean
  | (T1,T2)
  | (T1+T2)
  | PArray[T]
```

Polytypic DSL

DSL	Monotypic (traditional)	Polytypic (data type generic)
Shallow embedding	✓ Ordinary types and functions ✓ Execution on the JVM	
Deep embedding		

Polytypic DSL

DSL	Monotypic (traditional)	Polytypic (data type generic)
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

DSL	Monotypic (traditional)	Polytypic (data type generic)
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

DSL	Monotypic (traditional)	Polytypic (data type generic)
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	✓ Type-indexed types and functions ✓ Staging + transform ✓ Execution on XXX by code generation

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

Polytypic DSL

DSL	Monotypic (traditional)	Polytypic (data type generic)
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	✓Type-indexed types and Polytypic Staging ✓Staging + transform ✓Execution Nested Data Parallelism code generation

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

Polytypic Staging

- Applies to polytypic DSL
- Uses generic programming to capture domain semantics
- Allows to implement deep embedding of the DSL
- Is based on practical tools (mainstream Scala language)
- Not limitied to NDP domain

Framework: Polymorphic Embedding of DSLs

```
type Rep[A]    // abstract type constructor of representations
trait PArray[A] {    // parallel array (to express parallelism)
  def length: Rep[Int]
  def map[R](f: Rep[A => R]): Rep[PArray[R]]
  def zip[B](b: Rep[PArray[B]]): Rep[PArray[(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
trait PArray[A] {    // parallel array (to express parallelism)
  def length: Rep[Int]
  def map[R](f: Rep[A => R]): Rep[PArray[R]
  def zip[B](b: Rep[PArray[B]]): Rep[PArra
       …
}
```

- The same code
- Two implementations
- Equivalent semantics

```
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 })
```

Shallow Embedding	Deep Embedding
`type Rep[A] = A`	`type Rep[A] = Exp[A]`
✓Should be simple, good for testing and debugging	✓Should generate an efficient code

Data types in Nested Data Parallelism

In the DSL we can construct types

```
T = Unit | Int | Float | Boolean // base types
  | (T1,T2)                     // product of types
  | (T1+T2)                     // sum of types
  | PArray[T]                   // nested array
```


```
// Matrix in compressed row format (sparse matrix)
type SVector = PArray[(Int,Float)] // parallel array of products
type SMatrix = PArray[SVector]     // nested array of rows
type Vector = PArray[Float]        // dense vector
```

PArray is a type-indexed type

Type-indexed data type is a data type that is constructed in a generic way from an argument data type. (Ralf Hinze)

Each type T is from the family of types

```
T = Unit | Int | Float | Boolean      // base types
    | (T1,T2)                         // product
    | (T1+T2)                         // sum
    | PArray[T]                       // nested array
```

PArray is a type-indexed type

Type-indexed data type is a data type that is constructed in a generic way from an argument data type. (Ralf Hinze)

Each type T is from the family of types

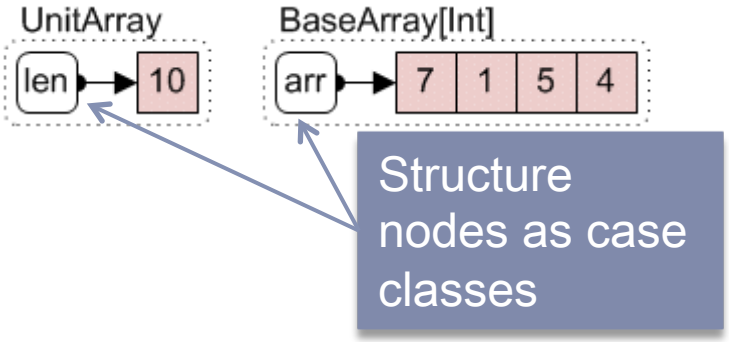
```
T = Unit | Int | Float | Boolean      // base types
    | (T1,T2)                         // product
    | (T1+T2)                         // sum
    | PArray[T]                       // nested array
```

Representation Transformation `RT: * -> *`

```
RT<PA[Unit]>  = UnitArray(len:Int)
RT<PA[T]>     = BaseArray(arr:Array[T]) if T - base type
RT<PA[(A,B)]> = PairArray(a:RT<PA[A]>, b:RT<PA[B]>)
RT<PA[PA[A]]> = NestedArray(
                    values: RT<PA[A]>,
                    segs  : RT<PA[(Int,Int)]>)
```

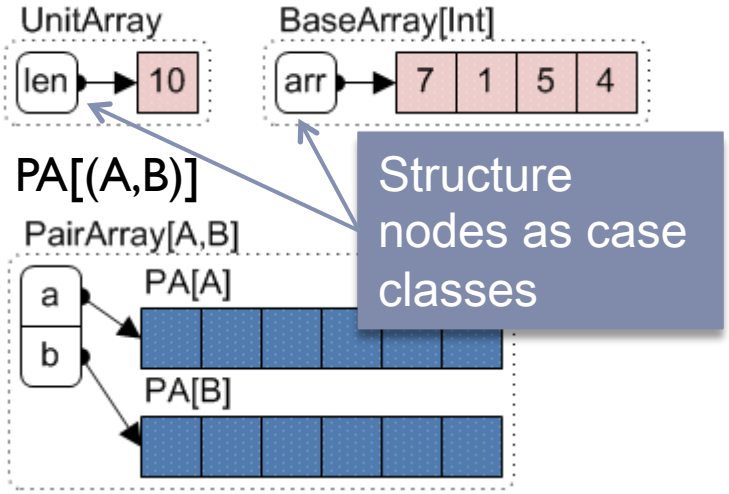
Data structures that support flattening

PA[Unit] PA[T] where T – base type



Data structures that support flattening

PA[Unit] PA[T] where T – base type

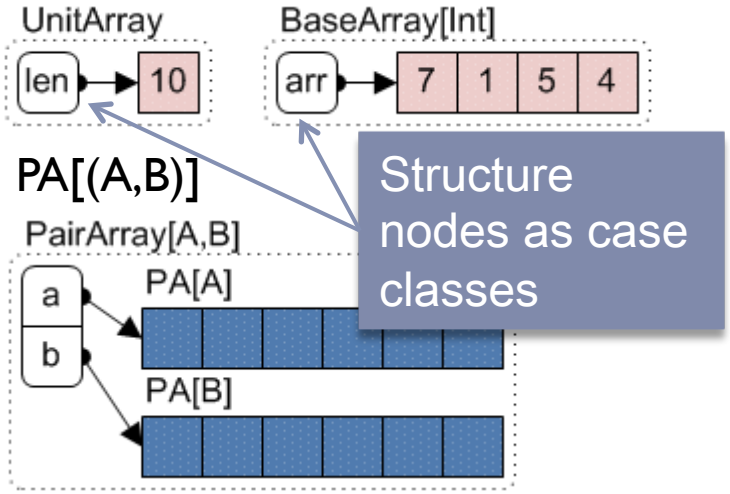


Constant time operations

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

Data structures that support flattening

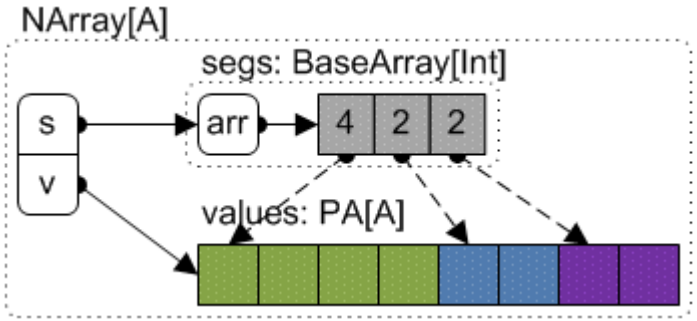
PA[Unit] PA[T] where T – base type



Constant time operations

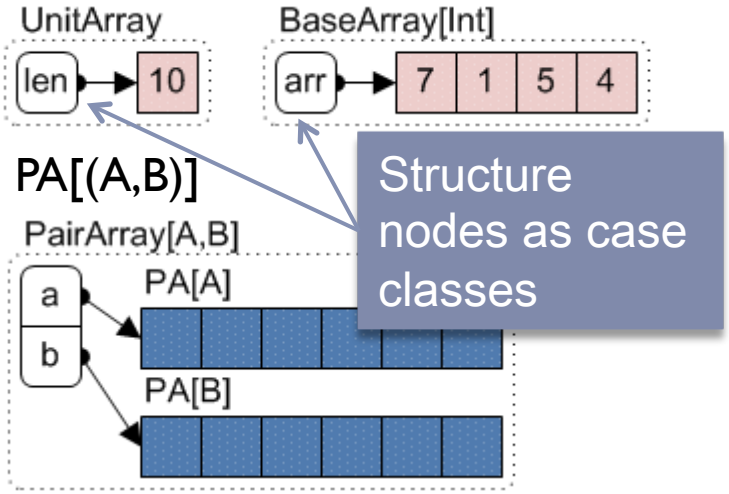
```
def zip[A,B](a:PA[A], b:PA[B]):PA[(A,B)]
    = PairArray(a, b)
```

PA[PA[A]]

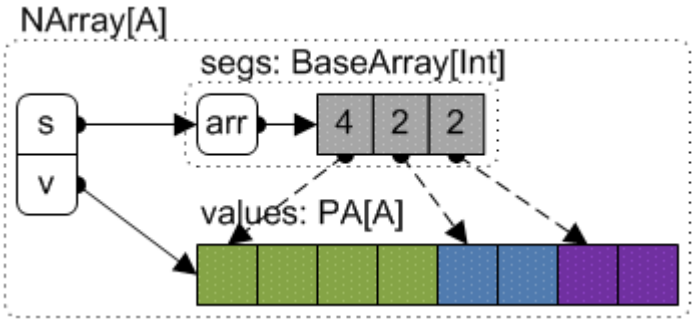


Data structures that support flattening

PA[Unit] PA[T] where T – base type



PA[PA[A]]



Constant time operations

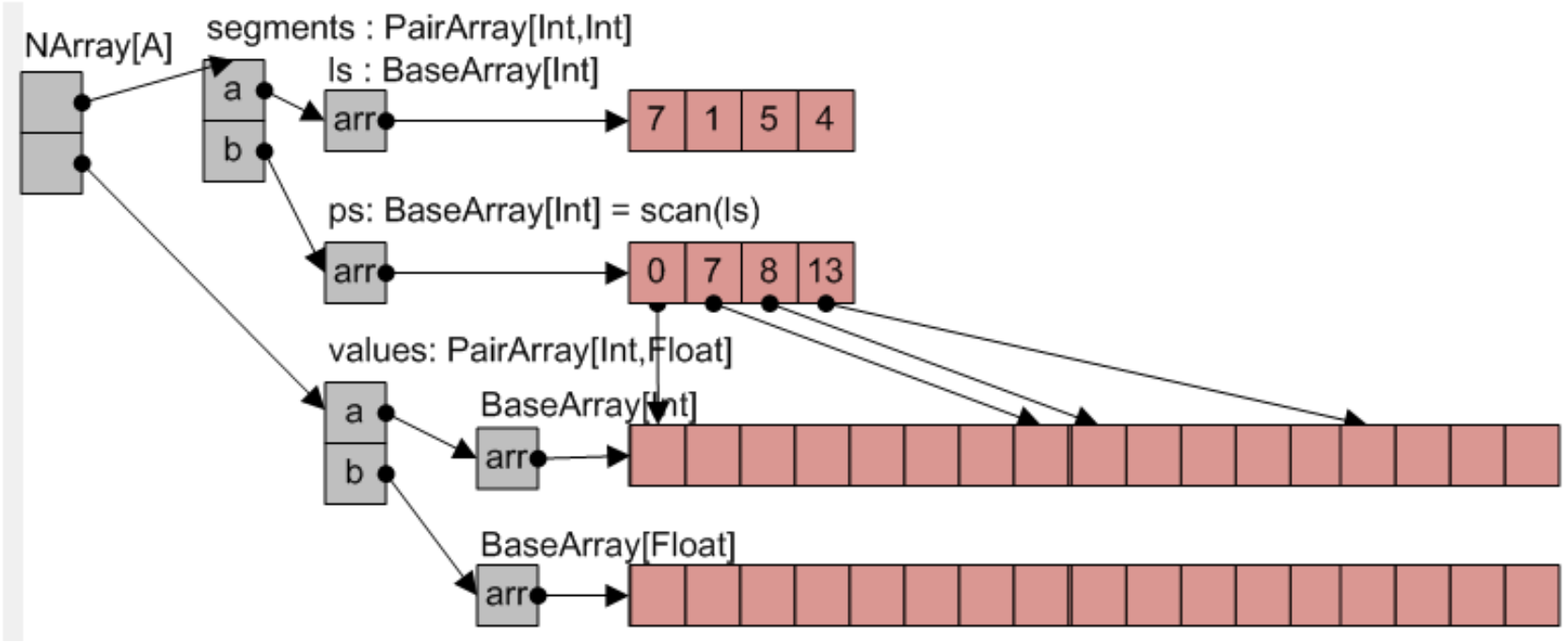
```
def zip[A,B](a:PA[A], b:PA[B]):PA[(A,B)]
    = PairArray(a, b)
def flatten[A](na: PA[PA[A]]): PA[A] =
    na match { case NArray(vs,_) => vs }
def nest[A,B](shape:PA[PA[A]],
                    vs:PA[B]): PA[PA[B]] =
  shape match {
   case NArray(_,segs)=> NArray(vs,segs)
  }
```

```
p^^(m: PA[PA[A]]): PA[PA[A]] =
  nest(m, p^(flatten(m)))
```

Matrix as type-indexed data type

```
type SVector = PArray[(Int,Float)]
type SMatrix = PArray[SVector]
```

RT<Matrix>



How flattening happens

Nested Code	Flattened Code
`p: A => B  // primitive`	`p^: PA[A] => PA[B] // p-lifted`
`g(as: PA[A]) = map p as` 	`g(as: PA[A]) = p^ as`

`h(m: PA[PA[A]]) = map g m` 	`h(m: PA[PA[A]])`
	`  = map g m`
	`  = map p^ m  // inlined g`
	`  = p^^(m)  // ???`

How flattening happens

Nested Code	Flattened Code
`p: A => B  // primitive`	`p^: PA[A] => PA[B] // p-lifted`
`g(as: PA[A]) = map p as` 	`g(as: PA[A]) = p^ as`

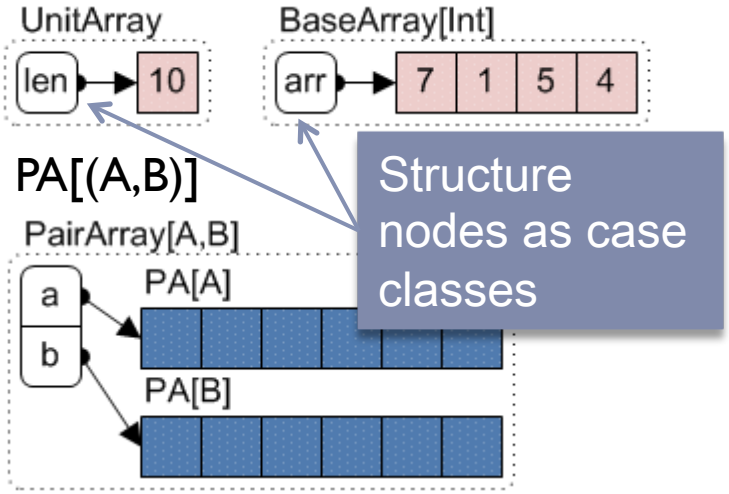
`h(m: PA[PA[A]]) = map g m` 	`h(m: PA[PA[A]])`
	`  = map g m`
	`  = map p^ m  // inlined g`
	`  = p^^(m)  // ???`

The key observation (we don't need p^^)

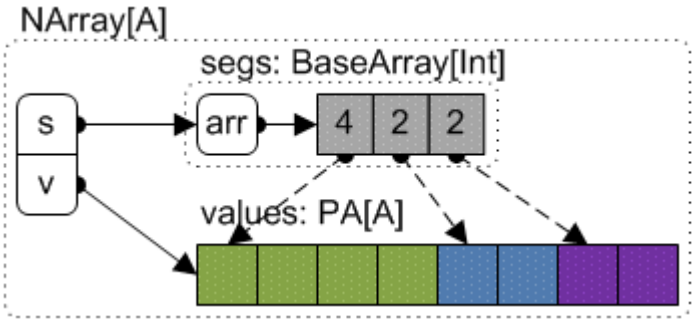
```
p^^ :: PA[PA[A]] -> PA[PA[A]]
p^^ m = nest(m, p^(flatten(m)))

flatten[A](nested: PA[PA[A]]): PA[A]
nest[A,B](shape: PA[PA[A]], values: PA[B]): PA[PA[B]]
```

Data structures that support flattening

PA[Unit] PA[T] where T – base type



PA[PA[A]]



Constant time operations

```
def zip[A,B](a:PA[A], b:PA[B]):PA[(A,B)]
    = PairArray(a, b)
def flatten[A](na: PA[PA[A]]): PA[A] =
    na match { case NArray(vs,_) => vs }
def nest[A,B](shape:PA[PA[A]],
                    vs:PA[B]): PA[PA[B]] =
  shape match {
   case NArray(_,segs)=> NArray(vs,segs)
  }
```

```
p^^(m: PA[PA[A]]): PA[PA[A]] =
  nest(m, p^(flatten(m)))
```

Related work

- This technique is not limited to Scala
- It was possible to implement shallow embedding in F#
- It may be interesting to consider deep embedding in F# using quasi-quotations
- The approach is not limited to NDP domain
- Hypothesis: The domain of reversive computations can be treated as polytypic and embedded as DSL in Scala using these techniques.

Conclusions

- It is useful to introduce a notion of polytypic DSL
- Nested data parallelism can be implemented in Scala as an embedded polytypic DSL
- To support flattening we need type-indexed data types from generic (polytypic) programming
- Polytypic Staging is a technique to implement **deep** embedding of polytypic DSLs

- Nested Data Parallelism is a "killer application" for this technique
- Could be interesting to consider **other polytypic domains**

References

[1] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. *Polymorphic embedding of DSLs*. GPCE '08

[2] Tiark Rompf, Martin Odersky. *Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs*. GPCE'10

[3] Bruno C.d.S. Oliveira, Jeremy Gibbons. *Scala for generic programmers*. WGP'08

Q&A

???