Peeling Abstraction Overhead from Functional Programs

Alexander Slesarenko (@avslesarenko)
Alexey Romanov (@alexey_r)
Maxim Gekk (@maxgekk)
Zotov Yuri (@zotov_yura)

Huawei Research, Russia

Scala Days Amsterdam, June 10, 2015

www.huawei.com

Huawei Technologies Co., Ltd.

HUAWEI

Agenda

- **Motivation**
- **A Framework for Domain-specific compilation**
- **How it works**
- **Discussion**

The problem

- Global properties of functional programs are usually better comparing to imperative ones.
- They are more predictable, robust, composable etc.
- Even performance and responsiveness can be better due to natural support of concurrency, parallelism, reactivity etc.

Roughly,

- 80% of the program is Ok.
- 20% need special care and performance improvement

There are **hot spots** that we need to take care of

Example 1: Matrix Vector Multiplication

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

Abstract data types

```
def mvm(m: Matr, v: Vec): Vec =
  Vec(m.rows.map(r => r.dot(v)))
```

Constructs a new vector from an array of values

Retrieves rows from the matrix as an array of vectors

Multiplys each row

What if this code is a part of a hotspot?

Abstraction overhead

Matrices: 10000 x 10000 elements

S$_m$ is matrix sparseness (% of zeros)

S$_v$ is vector sparseness

```
def mvm(m: Matr, v: Vec): Vec =
  Vec(m.rows.map(r => r.dot(v)))

val m = loadArrays("matrix.dat")
val v = loadArray("vector.dat")
val res = mvm(new DenseMatr(m), DenseVec(v))
```

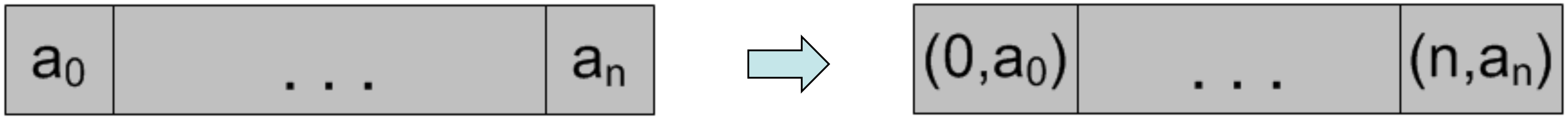
S_m	S_v	**dmdv**
0%	0%	11389
10%	10%	11408
50%	50%	12944
90%	90%	13093
99%	99%	13251
0%	50%	11483
50%	0%	12946
10%	90%	13907
90%	10%	14304

```
val nRows = m.length
var res = new Array[Double](nRows)
for (i <- 0 until nRows) {
  val row = m(i)
  val nCols = row.length
  var sum:Double = 0
  for (j <- 0 until nCols)
    sum += row(j) * v(j)
  res(i) = sum
}
```

S_m	S_v	**dmdv**
0%	0%	309
10%	10%	311
50%	50%	310
90%	90%	307
99%	99%	307
0%	50%	308
50%	0%	310
10%	90%	311
90%	10%	311

~ 40x performance improvement

Example 2: zipCollectionWithIndex




```
val C: Collection
val State: State[Int]  // State monad instance

def zipCollectionWithIndex[A](as: Coll[A]): Coll[(Int,A)] =
  State.eval(
    as.foldLeft(State.unit(C.empty[(Int, A)])){ case (acc, a) =>
      for { xs <- acc
             n <- State.get                    // get state
            _  <- State.set(n + 1)             // set new state
      } yield xs.append((n, a))
    }, 0)
```

Example 3: Compositional Application Architecture

The code we want to write

Algebraic DSLs

```
def prg[F[_]](implicit I: Interacts[F], A: Auths[F]): Free[F,Unit] = {
  import I._, A._;
  for {
    uid <- ask("What's your user ID?")
    pwd <- ask("Password, please.")
    u <- login(uid, pwd)
    b <- u.map(hasPermission(_, KnowSecret)).getOrElse(Return(false))
    _ <- if (b) {
      for { _ <- tell("Hi “ + uid) } yield ()
    } else {
      for { _ <- tell("Sorry “ + uid) } yield ()
    }
  } yield ()
}

type App[A] = Coproduct[Auth, Interact, A]
def runApp = prg[App].run(AuthId or InteractId)
```

Abstraction gives us a freedom of choice (change interpretation)

Source: Compositional Application Architecture With Reasonably Priced Monads (by Rúnar Bjarnason, Scala Days, Berlin, 2014)

Example 3: Compositional Application

The code we want to execute if it is a hot spot

```
def login: Unit = {
  println("What's your user ID?")
  val uid = readLine()
  println("Password, please.")
  val pwd = readLine()
  val ok = uid == "john.snow" && pwd == "Ghost”
  if (ok) {
    println("Hi " + uid)
  } else {
    println("Sorry " + uid)
  }
}
```

Agenda

- Motivation
- A Framework for Domain-specific compilation
- How it works
- Discussion

Why do bad things happen to good people?

- For performance of the hot spots we have to leave the FP paradise and go down to earth
- We **manually** do three kinds of things (usually bad things)

1. De-functionalization
   - remove lambdas, do inlining
   - change immutable to mutable
2. Specialization
   - specific data representation
   - specific mechanisms (caching, allocators, etc)
3. Acceleration
   - rewrite in C++ with JNI (or Unsafe) calls
   - offload to GPU
4. Other transformations
   - transform Array of Structures to Structure of Arrays

What’s the point?

The point is:

- We don’t really want to change the original code
- It is already in a good shape and has all the necessary abstractions
- It was designed by **us** (or by someone who is even smarter) with reusability, readability, maintainability and other abilities in mind

- Why we should want to change it at all?
- Just because of performance

Instead

- **teach the compiler** to generate an efficient code for a given hot spot
- **don’t ever change the code** which is already beautiful

Beautiful code

```
/** Matrix Vector Multiplication */

def mvm(m: Matr, v: Vec): Vec =
  Vec(m.rows.map(r => r.dot(v)))
```

Domain-specific compilation of hot spots

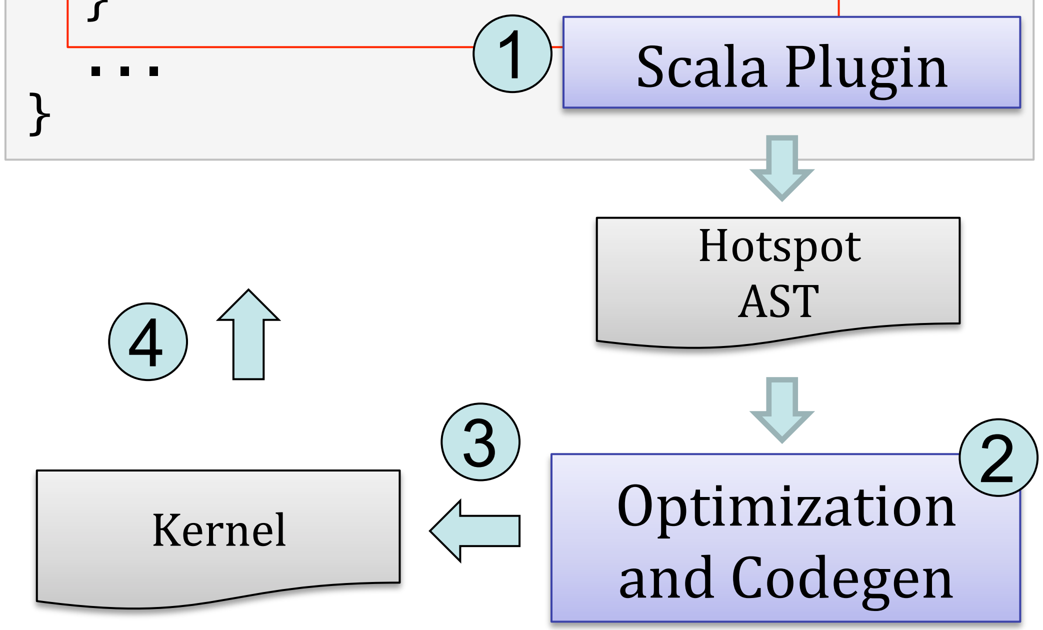
The key is to exploit properties of the hotspot's domain

Steps

1. Capture AST of the hotspot
2. Apply domain-specific optimizations
3. Generate efficient kernel
4. Replace original code with the kernel invocation

```
def mvm(m: Matrix,
        v: Vector): Vector =
  Vec(m.rows.map(r => r.dot(v)))

def main(args: Array[String]) = {
  ...
  for (i <- 0 to 1000000) {
    val v1 = mvm(m, vs(i))
           ...
  }
  ...
}
```



How we create all these optimizers for all possible domains?

Scalan – a framework for hotspot optimizers

Application Hot Spots

MVM	prg[F[_]]
zipWithIndex	etc.

Scala Compiler

Scala Plugin/Macros

Scalanizer	Code Virtualization

scalan-library

Linear Algebra	Graphs
Collections	FP

scalan-meta

Boilerplate Generator	Semantic Annotations

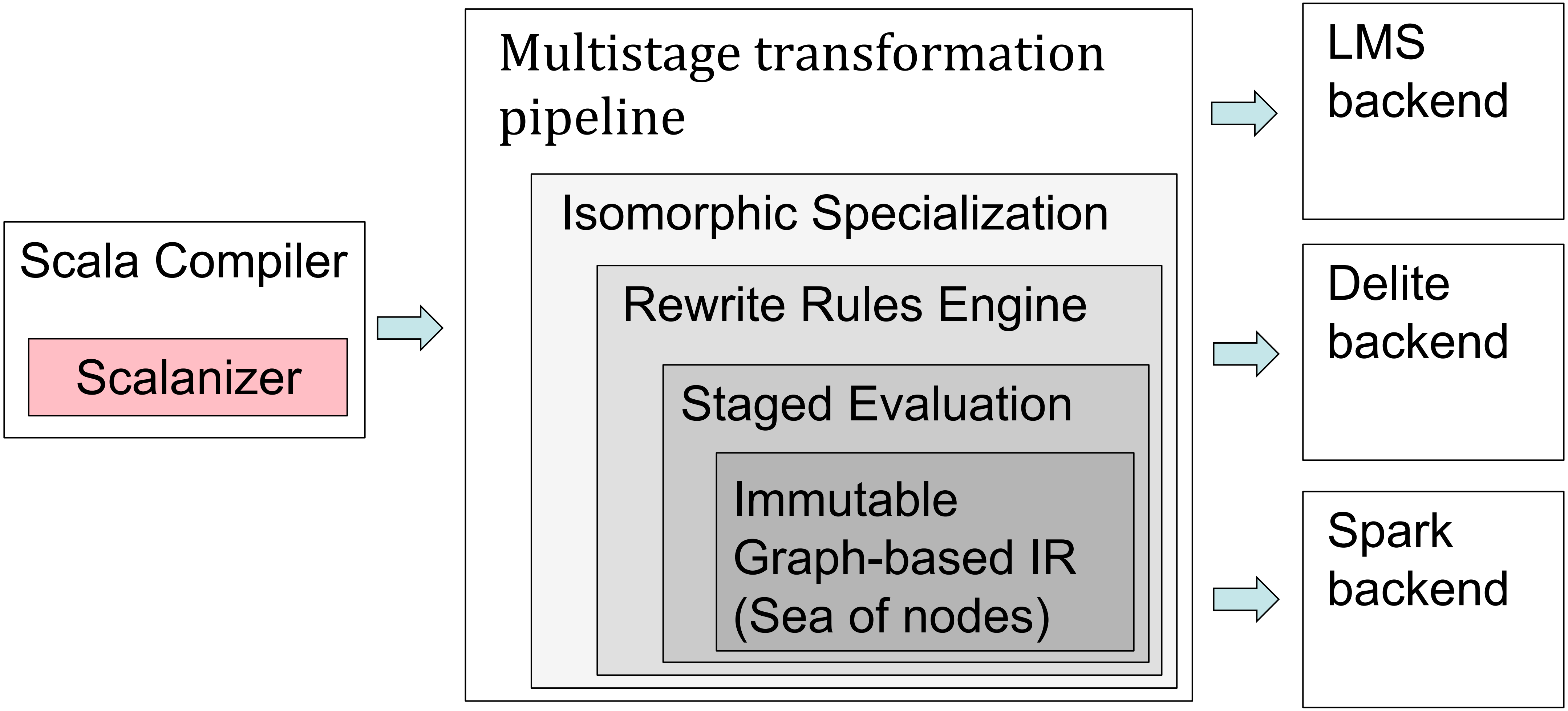
scalan-core

Common Immutable Graph-based IR	Graph-based Rewriting Engine	Isomorphic Specialization Engine	Multistage Transformation Pipeline

scalan-backends

Scalan-to-LMS Bridge	Scalan-to-Delite Bridge	Scala Codegen	C++ Codegen

Compilation pipeline of Scalan




```
// Rule: (if (c1) t1 else e1, if (c2) t2 else e2)
                    when c1 == c2 ==>  if (c1) (t1, t2) else (e1, e2)
// Rule: (if (c) t else e)._1 ==> if (c) t._1 else e._1
// Rule: (if (c) t else e)(arg) ==> if (c) t(arg) else e(arg)
```

DEMONSTRATION

Agenda

- **Motivation**
- **A Framework for Domain-specific compilation**
- **How it works**
- **Discussion**

What do we mean by saying domain-specific?

- Focus on semantics, NOT syntax

- Families of types with their operations, internal structure and properties

- Cross-domain interactions

Example domains

```
trait Vec[T] {    // abstract vector
  def length: Int
  def coords: Array[T]
  def dot(vec: Vec[T]): T
}

trait Matr[T] {   // abstract matrix
  def rows: Array[Vec[T]]
  def * (vec: Vec[T]): Vec[T] = {
    val vs: Array[Vec[T]] = rows
    Vec(vs.map(v => v.dot(vec)))
  }
}

def Vec[T](coords: Array[T]): Vec[T]
```

The goal:
For each hotspot extract a set of **relevant domain-specific artifacts** the hotspot depends on

Domain-specific artifacts

- Besides AST of the hotspot itself Scalanizer has to extract all relevant implementations in the domain(s)

- The implementations of abstract data types are given by subtyping

Example domain implementations

```
class DenseVec[T](val coords: Array[T])
  extends Vec[T] {
  def length = coords.length
  def dot(vec: Vec[T]) = vec match {
    case dv: DenseVec[T] =>
      sum(coords |*| dv.coords)
    case sv: SparseVec[T] =>
      sum(sv.values |*| coords(sv.indices))
  }
}

class SparseVec[T](
      val indices: Array[Int],
      val values: Array[T],
      val length: Int) extends Vec[T] {
  ...
}

class DenseMatr[T](
      val rows: Array[Vec[T]]) extends Matr[T]{
  ...
}
```

Optimization techniques supported by Scalan

1. De-functionalization
   - ✔ remove lambdas, do inlining
   - change immutable to mutable
2. Specialization
   - ✔ change specific data representation
   - use specific mechanisms (caching, allocators, etc)
3. Acceleration
   - ✔ rewrite in C++ with JNI (or Unsafe) calls
   - offload to GPU
4. Other transformations
   - ✔ rewrite rules
   - ✔ transform Array of Structures to Structure of Arrays
   - ✔ flattening (for Nested Data Parallelism)

Central idea behind Scalan

```
def mvm(m: Matr, v: Vec): Vec =
  Vec(m.rows.map(r => r.dot(v)))
```

Domain A
(Linear Algebra)

Automatic cross-domain translation

```
def dmdv(
  m: Array[Array[Double]],
  v: Array[Double]): Array[Double] =
    m.map(row => sum(row |*| v))
```

Domain B
(Arrays)

Optimizing Compiler (LMS)

Runtime
(JVM,C++)

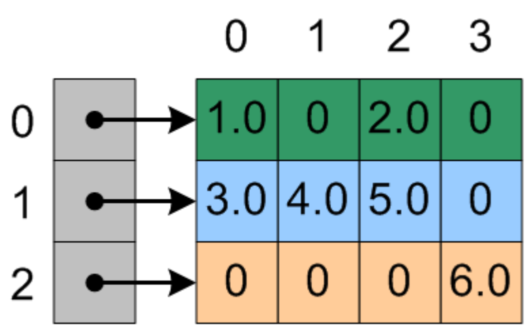
Questions:

1. Is it generic enough to handle all programs in A?
2. What about cross-domain programs?
3. Is it possible to have multilevel translations?

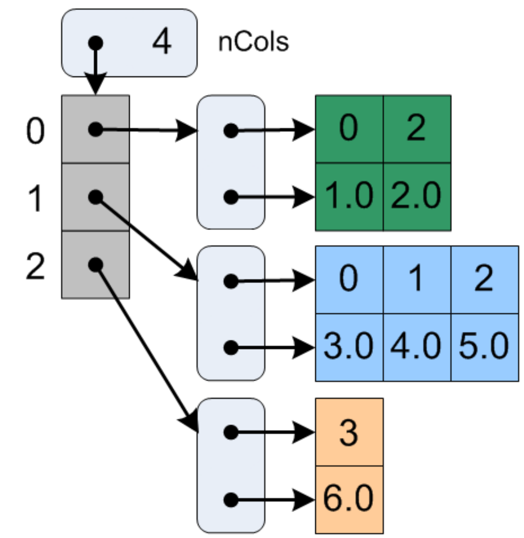
Selecting the best data structure for an algorithm

```scala
def mvm(m: Matr, v: Vec): Vec =
  Vec(m.rows.map(r => r.dot(v)))
```

Dense Matrix

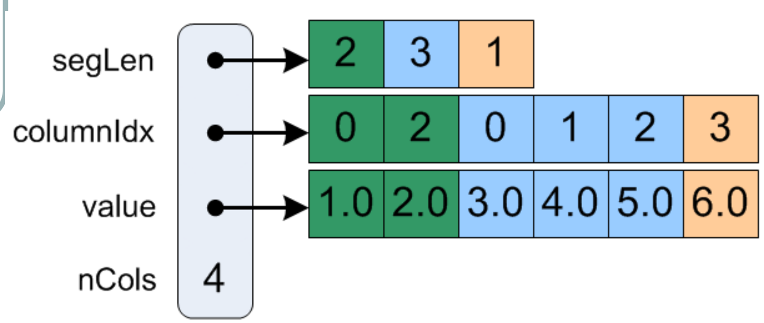


Sparse Matrix



Matrix

Flat Sparse Matrix



Why does it matter?

Consider matrix 10^4 x 10^4

$\mathbf{S_m}$ - matrix sparseness (% of zeros)

$\mathbf{S_v}$ - vector sparseness

S_m	S_v	dmdv	dmsv	smdv	smsv
0%	0%	309	354	366	760
10%	10%	311	323	332	1002
50%	50%	310	202	187	924
90%	90%	307	104	42	172
99%	99%	307	18	8	18
0%	50%	308	198	373	1134
50%	0%	310	359	187	986
10%	90%	311	118	335	497
90%	10%	311	323	42	345

Execution time in milliseconds

Best variant for given data sparseness

How do you know this is a bad choice?

The solution is based on specialization and profiling

```
def mvm(m: Matr, v: Vec): Vec = {
  Vec(m.rows.map(r => r.dot(v)))
}
```

- abstract domain
- concrete domain

⇩ specialize, run and measure performance

```
def dmdv(m: Array[Array[Double]], v: Array[Double]): Array[Double] =
    m.map(row => sum(row |*| v))

def dmsv(m: Array[Array[Double]], v: (Array[Int], Array[Double], Int)): Array[Double] = {
  val (indices, values, _) = v
  m.map(row => sum(row(indices) |*| values))}

def smsv(m: Array[ (Array[Int],Array[Double],Int) ],
         v: (Array[Int],Array[Double],Int)): Array[Double] = {
  val (indices, values, _) = v
  m.map((is,(vs,_)) => dotProductSV(is, vs, indices, values))}

def smdv(m: Array[(Array[Int], (Array[Double], Int))], v: Array[Double]): Array[Double] =
  m.map(r => {
    val (indices, values, _) = r
    sum(values |*| v(indices))
  })
```

Implementation-specific primitives:

- map
- sum
- |*|
- dotProductSV

Can we automate such a specialization?

What kind of specialization we need?

✔ Isomorphic Specialization

■ abstract domain
■ concrete domain

$P: A \to B,\ A \approx_A A',\ B \approx_B B'$

⇩

$P': A' \to B'$
such that
the diagram commutes
and
P' is more concrete
than P

$$\begin{tikzcd} A \arrow[r, "P"] & B \arrow[d, "\approx_B.from"] \\ A' \arrow[u, "\approx_A.to"] \arrow[r, "P'"'] & B' \end{tikzcd}$$

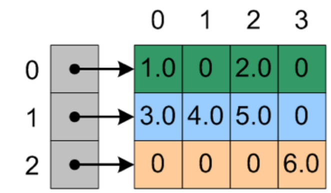
Isomorphic specialization is a systematic transformation of original program which removes all class based abstractions.

It is implemented as a set of graph rewriting rules and thus it can be easily combined with other transformations and facilitate them

Isomorphic Specialization for MVM

- abstract domain
- concrete domain

Dense Matrix



P: A -> B

⇓

P′: A′ -> B′

$$
\begin{tikzcd}
\texttt{mvm}: (\texttt{Matr}, \arrow[d, leftrightarrow, "\text{iso} \approx"'] & \texttt{Vec}) \arrow[d, leftrightarrow, "\text{iso} \approx"'] & \longrightarrow \texttt{Vec} \arrow[d, leftrightarrow, "\text{iso} \approx"'] \\
\textbf{dmdv}: (\text{Array[Array]}, & \text{Array}) & \longrightarrow \text{Array}
\end{tikzcd}
$$

$$
\begin{tikzcd}
A \arrow[r, "P"] & B \arrow[d, "\approx \ \mathit{from}"] \\
A' \arrow[u, "\approx \ \mathit{to}"] \arrow[r, "P'"'] & B'
\end{tikzcd}
$$

$$
\begin{tikzcd}
(\texttt{Matr},\texttt{Vec}) \arrow[r, "\texttt{mvm}"] & \texttt{Vec} \arrow[d, "\approx \ \texttt{from}"] \\
(\texttt{Array[Array]},\texttt{Array}) \arrow[u, "\approx \ \texttt{to}"'] \arrow[r, "\textbf{dmdv}"'] & \texttt{Array}
\end{tikzcd}
$$

“Specializable by design” abstractions

Vectors Domain

```
Array[T]
```

≈ iso

```
case class DenseVec[T](coords: Array[T])
  extends Vec[T]
```

```
trait Vec[T] {
  def length: Int
  def coords: Array[T]
  def dot(vec: Vec[T]): T
}
```

```
case class SparseVec[T](
  indices: Array[Int],
  values: Array[T],
  length: Int) extends Vec[T]
```

≈ iso

```
(Array[Int], Array[T], Int)
```

Matrix Domain

```
Array[Vec[T]]
```

≈ iso

```
case class DenseMatr[T](
  rows: Array[Vec[T]])
  extends Vec[T]
```

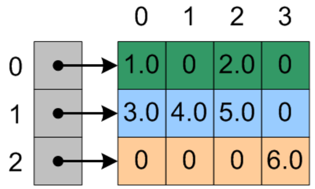
```
trait Matr[T] {
  def rows: Array[Vec[T]]
}
```

```
case class SparseMatr[T](
  rows: Array[(Array[Int], Array[T])],
  nCol: Int) extends Vec[T]
```

≈ iso

```
(Array[(Array[Int], Array[T])], Int)
```

Cross-domain translation in a pipeline




```
def mvm(m: Matr, v: Vec): Vec =
  Vec(m.rows.map(r => r.dot(v)))
```

Dense Matrix ≈ iso ⟷ Array[Array[T]]

Isomorphic Specialization into immutable arrays

```
def dmdv(m: Array[Array[Double]], v: Array[Double]): Array[Double] =
  m.map(row => sum(row |*| v))
```

Array language compilation with loop fusion, deforestation etc. using LMS (Lightweight Modular Staging)

```
def dmdv(m: Array[Array[Double]],
         v: Array[Double]): Array[Double] = {
  val nRows = m.length
  var res = new Array[Double](nRows)
  for (i <- 0 until nRows) {
    val row = m(i)
    val nCols = row.length
    var sum:Double = 0
    for (j <- 0 until nCols)
      sum += row(j) * v(j)
    res(i) = sum
  }
}
```

Wrap-Apply-Unwrap pattern

```
def mvm(m: Matr, v: Vec): Vec =
  Vec(m.rows.map(r => r.dot(v)))

def dmdv(m: Array[Array[Double]],
         v: Array[Double]): Array[Double]= {
  // wrap
  val dm = new DenseMatr(
             m.map(r => new DenseVec(r)))
  val dv = new DenseVec(v)

  val vres: Vec = mvm(dm, dv) // apply

  vres.coords    // unwrap
}
```

In abstract context we don't know how Matr[T] and Vec[T] are implemented

In concrete use case we execute abstract algorithm with concrete implementations in 3 steps:

1) Wrap data
2)Apply logic
3)Unwrap data

$$\begin{tikzcd}
(\mathtt{Matr},\mathtt{Vec}) \arrow[r, "\mathtt{mvm}"] & \mathtt{Vec} \arrow[d, "\approx \text{ unwrapper}", "\mathtt{from}"'] \\
(\mathtt{Array[Array]},\mathtt{Array}) \arrow[u, "\text{wrapper } \approx", "\mathtt{to}"'] \arrow[r, "\mathtt{dmdv}"'] & \mathtt{Array}
\end{tikzcd}$$

We have an instance of isomorphic specialization

Isomorphic Specialization by Staged Evaluation

```
def dmdv_S(m: Array[Array[Double]],
           v: Array[Double]) = SE[{
  val dm = new DenseMatr(
             m.map(r => new DenseVec(r)))
  val dv = new DenseVec(v)
  val vres = mvm(dm, dv)
  vres.coords
}]
```

Staged Evaluation(SE) with graph-based rewriting

```
def dmdv_S(m: Array[Array[Double]],
         v: Array[Double]) =
    m.map(row => sum(row |*| v))
```

Isomorphic Specialization by Staged Evaluation

$$SE[\![\_]\!] : Term \rightarrow List\ \partial\mathcal{E} \rightarrow Dag \rightarrow Dag\langle Addr\rangle$$

$$\begin{array}{lcl}
SE[\![v']\!]\ (d\ \overline{v}\Box e'\overline{e} :: \mathcal{R})\ \Delta & \mapsto & SE[\![e']\!](d\ \overline{v}v'\Box\overline{e} :: \mathcal{R})\Delta \\
SE[\![d\ e_0\overline{e}]\!]\ \mathcal{R}\ \Delta & \mapsto & SE[\![e_0]\!]\ (d\ \Box\overline{e} :: \mathcal{R})\ \Delta \\
SE[\![l]\!]\ \mathcal{R}\ \Delta & \mapsto & SE[\![v]\!]\ \mathcal{R}\ \Delta'\ \text{where}\ \Delta'\langle v\rangle = \Delta \leftarrow l \\
SE[\![\textbf{case}\ e\ \textbf{of}\ \{p_i \rightarrow e_i\}]\!]\ \mathcal{R}\ \Delta & \mapsto & SE[\![e]\!]\ (\textbf{case}\ \Box\ \textbf{of}\ \{p_i \rightarrow e_i\} :: \mathcal{R})\ \Delta \\
SE[\![e_1\ e_2]\!]\ \mathcal{R}\ \Delta & \mapsto & SE[\![e_1]\!]\ (\Box\ e_2 :: \mathcal{R})\ \Delta \\
SE[\![v']\!]\ (d\ \overline{v}\Box :: \mathcal{R})\ \Delta & \mapsto & SE[\![v]\!]\ \mathcal{R}\ \Delta'\ \text{where}\ \Delta'\langle v\rangle = \Delta \leftarrow d\ \overline{v}v'
\end{array}$$

$$\begin{array}{lcl}
SE[\![v']\!](\textbf{case}\ \Box\ \textbf{of}\ \{p_i\ \overline{x_i} \rightarrow e_i\} :: \mathcal{R})\ \Delta_0 & \mapsto & SE[\![v]\!]\ \mathcal{R}\ \Delta'\ \text{where} \\
& & \quad (\Delta_i\langle\overline{\alpha_i}\rangle \quad = \Delta_{i-1} \leftarrow \overline{x_i})^n;\ \Delta'_0 = \Delta_n \\
& & \quad (\Delta'_i\langle\beta_i\rangle \quad = SE[\![[\overline{\alpha_i}/\overline{x_i}]e_i]\!]\ \epsilon\ \Delta'_{i-1})^n \\
& & \quad \Delta'\langle v\rangle \quad = \Delta'_n \leftarrow \textbf{case}\ v'\ \textbf{of}\ \{p_i\ \alpha^{n_i} \rightarrow \beta_i\}
\end{array}$$

$$\begin{array}{lcl}
SE[\![v']\!](\Box\ e_2 :: \mathcal{R})\ \Delta & \mapsto & SE[\![e_2]\!]\ (v'\ \Box :: \mathcal{R})\ \Delta \\
SE[\![v_2]\!](v_1\ \Box :: \mathcal{R})\ \Delta & \mapsto & SE[\![v]\!]\ \mathcal{R}\ \Delta\ \text{where}\ \Delta'\langle v\rangle = \Delta \leftarrow app\ v_1 v_2
\end{array}$$

$$\begin{array}{lcl}
SE[\![\lambda x.e]\!]\mathcal{R}\ \Delta & \mapsto & SE[\![v]\!]\ \mathcal{R}\ \Delta'\ \text{where} \\
& & \quad \Delta_1\langle\alpha\rangle \quad = \Delta \leftarrow x \\
& & \quad \Delta_2\langle\beta\rangle \quad = SE[\![[\alpha/x]e]\!]\ \epsilon\ \Delta_1 \\
& & \quad \Delta'\langle v\rangle \quad = \Delta_2 \leftarrow \lambda\alpha.\beta
\end{array}$$

$$\begin{array}{lcl}
SE[\![e.\mathtt{f}]\!]\ \mathcal{R}\ \Delta & \mapsto & SE[\![e]\!]\ (\Box.\mathtt{f} :: \mathcal{R})\ \Delta \\
SE[\![v']\!](\Box.\mathtt{f} :: \mathcal{R})\ \Delta & \mapsto & SE[\![v]\!]\ \mathcal{R}\ \Delta'\ \text{where}\ \Delta'\langle v\rangle = \Delta \leftarrow v'.\mathtt{f}
\end{array}$$

$$\begin{array}{lcl}
SE[\![v']\!]((v_0 : \sigma).m(\overline{w}\Box) :: \mathcal{R})\ \Delta & \mapsto & SE[\![v]\!]\ \mathcal{R}\ \Delta'\ \text{where} \\
& & \quad \Delta_1\langle\mu\rangle \quad = \Delta \leftarrow \sigma.m \\
& & \quad \Delta'\langle v\rangle \quad = \Delta_1 \leftarrow mcall(v_0, \mu, \overline{w} :: v')
\end{array}$$

$$SE[\![v]\!]\ \epsilon\ \Delta \quad \mapsto \quad \Delta\langle v\rangle$$

Figure 16: Staged Evaluation (call-by-value)

Agenda

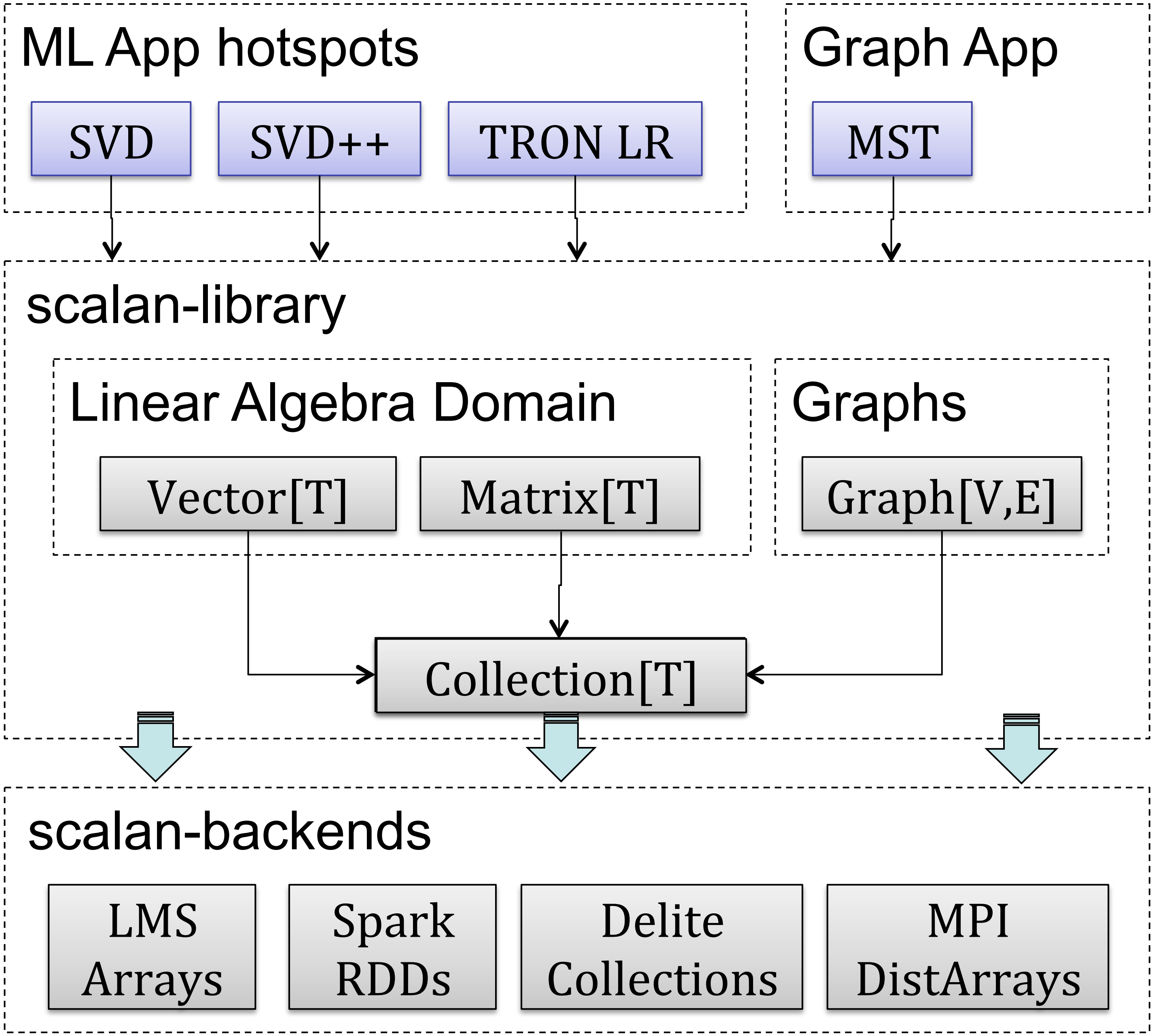
- ❑ **Motivation**
- ❑ **A Framework for Domain-specific compilation**
- ❑ **How it works**
- ❑ **Discussion**

How we use Scalan: Typical Application architecture

Write algorithms using general purpose abstractions packaged as Scala libraries

Each abstraction has alternative implementations in terms of APIs of backends

Each backend provides API and domain-specific optimizing compiler/ runtime



Existing Implementations

Functional kernels implemented in Scalan and then specialized into various backends

Kernel		Backends			
Domain	**Name**	**LMS Arrays**	**Spark RDDs**	**Delite Collection**	**MPI**
Machine Learning	Logistic Regression, Trust Region Newton Method (150 LoC)	✔			✔
Linear Algebra	Matrix Vector Multiplication (1 LoC)	✔	✔	✔	✔
	SVD (100 LoC)	✔	✔		
	SVD++(300 LoC)	✔	✔		
Graphs	MST (20 LoC)	✔			
Optimization	TabuSearch (20 LoC)	✔	✔		
Functional Programming	Monad combinators (sequence, traverse, etc)	✔			
	Monads: State, Reader, Free, FreeState	✔			
	Compositional Application Architecture	✔			

Projects that may benefit

- Scala
- Spark
- Functional streams (Scalaz-stream)
- Functional lenses (Monocle)
- Meerkat parsers

Limitations

1. Referential transparency (no operations with effects)
2. Limited recursion support now (tail recursion soon)

Effects can be introduces explicitly using Applicative or Monad (IO, State, etc.)
LMS-style reified effects are also supported, but they better used on the last stages of the compiler

Get involved

1. Check out project at **github.com/scalan**
2. Ask questions on Google Group
   https://groups.google.com/d/forum/scalan
3. Follow us on twitter
4. Scalanize your favorite domain