First-class Isomorphic Specialization by Staged Evaluation

Alexander Slesarenko
Alexander Filippov
Alexey Romanov

WGP'14, August 31, 2014, Gothenburg, Sweden

Huawei Technologies Co., Ltd.

Agenda

- **Isomorphic Specialization in a Nutshell**
- **Why it matters**
- **First-class Isomorphisms**
- **How it works**

Program Specialization

Partial Evaluation

P(x,y), x = const

⇩

P'(y)
for all y: ***P'***(y) = P(x,y)

Supercompilation

$P(x),\ x \in D$

⇩

$\boldsymbol{P'}(x) \wedge x \in D' \subset D$
for all $x \in D'$: $\boldsymbol{P'}(x) = P(x)$

Isomorphic Specialization

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

⇩

$\boldsymbol{P'}: X' \to Y'$
such that the diagram commutes

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

P' is better in some sence

Example: **M**atrix **V**ector **M**ultiplication

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

Abstract data types

```
def mvm(m: Matr[Float], vec: Vec[Float]): Vec[Float] = {
  val rs: Array[Vec[Float]] = m.rows
  Vec(rs.map(r => r.dot(vec)))
}
```

Assume we can retrieve rows from the matrix as an array of vectors

We can construct a new vector from an array of values

when we map an Array we create a new Array

What kind of language we use here? Looks like Scala…

Embedded Domain Specific Languages

```
trait Array[T] {  // abstract array interface
  def length: Int
  def map[R](f: T => R): Array[R]
}

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

trait Matr[T] {   // abstract matrix interface
  def rows: Array[Vec[T]]
}

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

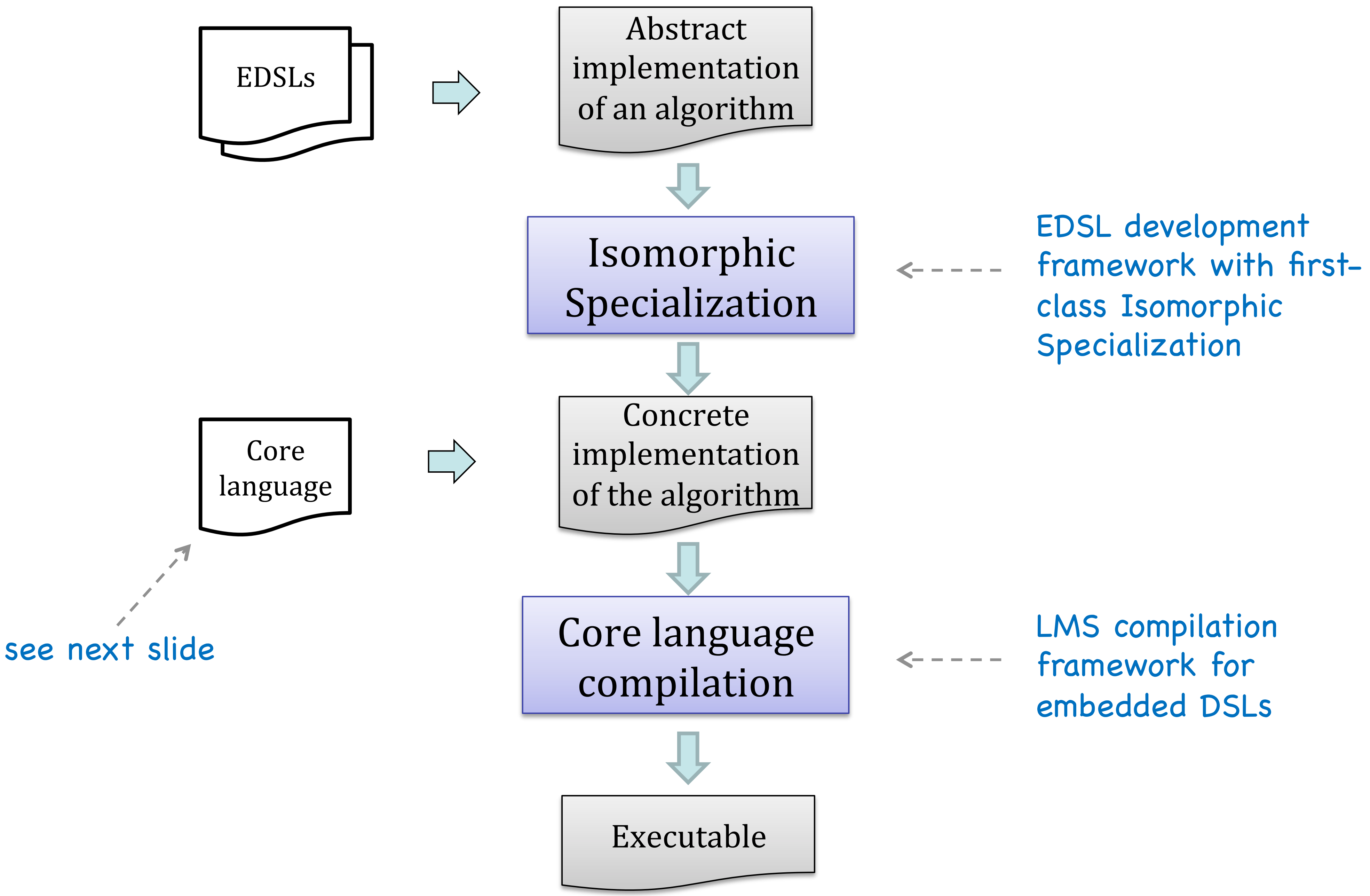
Examples:

```
val average = sum(xs) / xs.length
val squares = xs.map(x => x * x)

val x = v.coords(i)
val s = v1.dot(v2)

m.rows.map(v => v.dot(vec))
```

this languages can grow as a domain specific library in Scala (EDSL)

```
def mvm(m: Matr[T], vec: Vec[T]): Vec[T] = {
  val vs: Array[Vec[T]] = m.rows
  Vec(vs.map(v => v.dot(vec)))
}
```

Outline of the method



Core language with immutable arrays

You can select any language which works for you

In the Core language we can construct the following types

```
T = Unit | Int | Float | Boolean // base types
  | (T1,T2)                      // pair of types
  | Either[T1,T2]                // sum of types
  | Array[T]                     // array of values of the type T
```

```
trait Array[T] {
  def length: Int
  def apply(index: Int): T // get element at index
  def apply(indices: Array[Int]): Array[T]
  def map[R](f: T => R): Array[R]
  def filter(p: T => Boolean): Array[T]
  def zip[U](other: Array[U]): Array[(T,U)]
  def |*| (other: Array[T]): Array[T] // element-wise op
}

def range(start: Int, len: Int): Array[Int]
def sum[T](arr: Array[T]): T
def unzip[T,U](pairs: Array[(T,U)]): (Array[T],Array[U])
def dotSV[T](
      indices1: Array[Int], values1: Array[T],
      indices2: Array[Int], values2: Array[T]): T
```

Now, if we have a compiler from the
Core language into something executable

then we can implement abstract data
types using the Core language.

Let's look at the implementation

MVM using dense representations

```
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 DenseMatr[T](val rows: Array[DenseVec[T]]) extends Matr[T]

def Vec(coords: Array[T]): Vec[T] = new DenseVec(coords)

def mvm(m: Matr[T], vec: Vec[T]): Vec[T] =
  Vec(m.rows.map(v => v.dot(vec)))

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

Implements interface

Use core language types and primitives

Allow mixed types

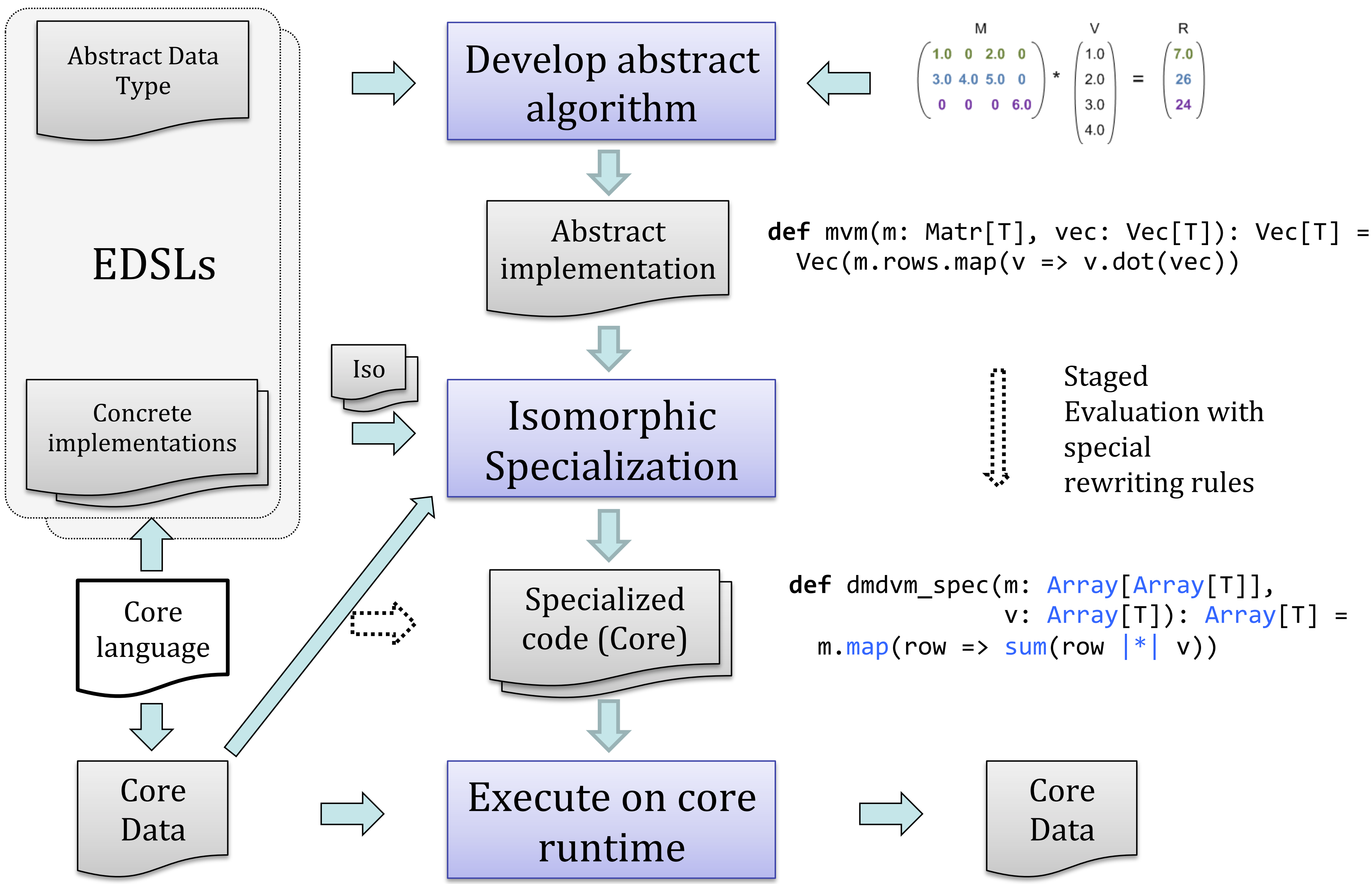
default vector is dense

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

In order to execute using core data we do three steps:

1) wrap
2) apply
3) extract

Outline of the method (2)



Because of staging, specialization can happen at runtime

Specialization by Staged Evaluation

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

⇩ Use Matr and Vec abstract data type

```
def mvm(m: Matr[T], vec: Vec[T]): Vec[T] =
  Vec(m.rows.map(v => v.dot(vec)))
```

⇩ Staged Evaluation with graph rewriting rules

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

Staged Evaluation with rewriting

```
val dv1 = new DenseVec[Double](v1)      // class DenseVec[T](val coords: Array[T])
val dv2 = new DenseVec[Double](v2)
SE[sum(dv1.coords |*| dv2.coords)]
                                        // Ctx[new C(v_1,…,v_n).f_i] → Ctx[v_i]
==> SE[sum(v1 |*| dv2.coords)]
==> SE[sum(v1 |*| v2)]
==> sum(v1 |*| v2)
```

Every field is also a constructor argument

```
val m: Array[Array[T]]
SE[m.map(r => new DenseVec(r)).map(dv => dv.coords)]
             // RW[as.map(f).map(g)] → as.map(a => g(f(a)))
             // f = (r => new DenseVec(r))
             // g = (dv => dv.coords)
==> SE[m.map(r => new DenseVec(r).coords)]
             // Ctx[new C(v_1,…,v_n).f_i] → Ctx[v_i]
==> SE[m.map(r => r)]
             // RW[as.map(a => a)] → as
==> m
```

Take It Home

1. **D**evelop an algorithm using abstract data types of embedded DSLs

2. **I**mplement abstract data types using a functional core language with an efficient compile

3. **S**pecialize an abstract code into the core language

Agenda

- ❑ **Isomorphic Specialization in a Nutshell**
- ❑ **Why it matters**
- ❑ **First-class Isomorphisms**
- ❑ **How it works**

Generated executable code

```
def mvm(m: Matr[T], vec: Vec[T]): Vec[T] =
  Vec(m.rows.map(v => v.dot(vec)))
```

Isomorphic Specialization

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

Core language compilation with loop fusion, deforestation etc.

```
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
    }
    res
}
```

Why it matters

All matrices: 10^4 x 10^4 elements
$\mathbf{S_m}$ is matrix sparseness (% of zeros)
$\mathbf{S_v}$ is vector sparseness

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

Execution time in milliseconds running as Scala program without abstraction elimination and optimizations

1) Isomorphic Specialization produces version in Core language with immutable arrays
2) LMS compiler optimizes purely functional array operations (deforestation, loop fusion etc.)

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

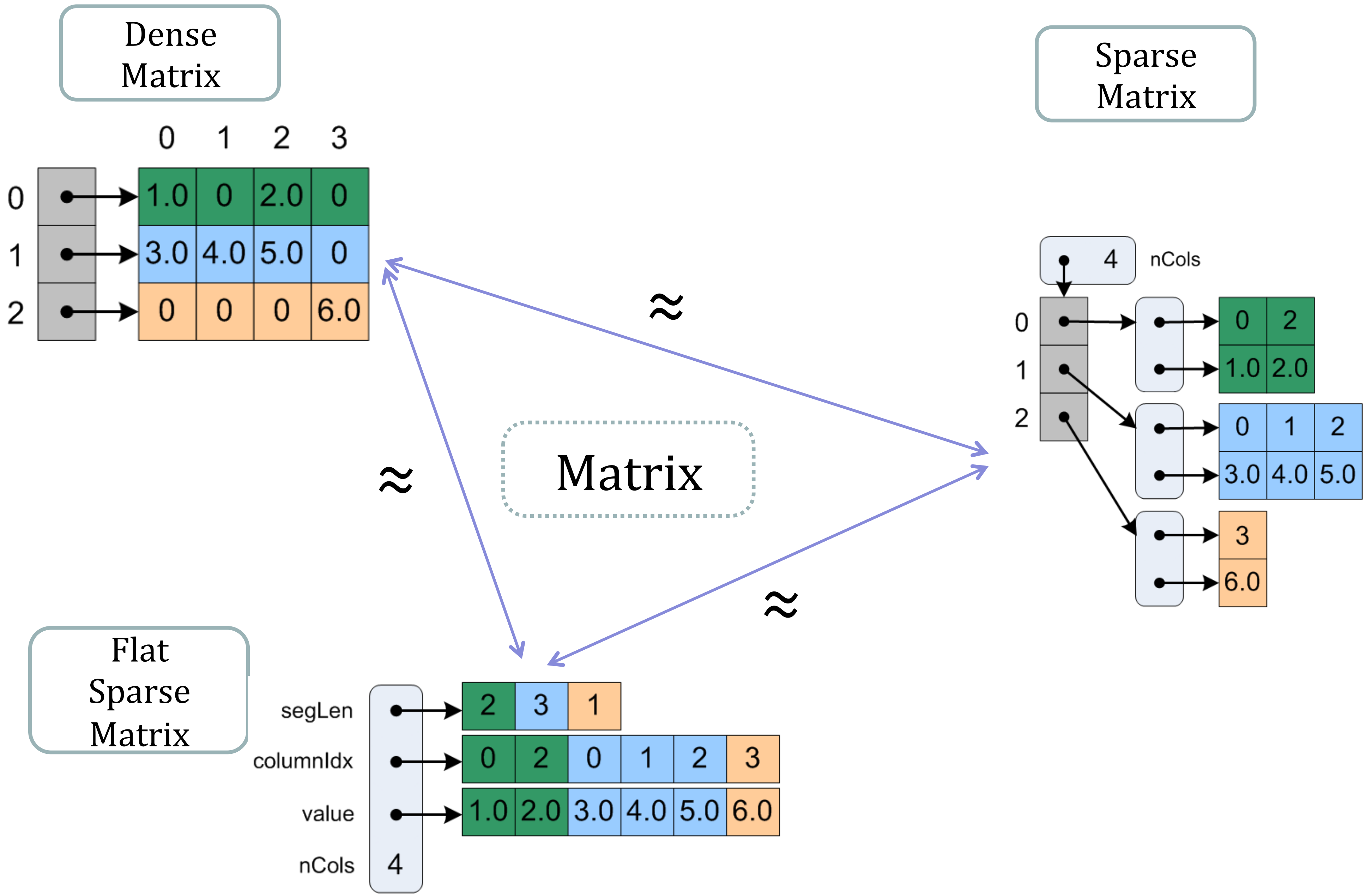
Execution time in milliseconds

~ 40x combined performance improvement

Agenda

- ❑ **Isomorphic Specialization in a Nutshell**
- ❑ **Why it matters**
- ❑ **First-class Isomorphisms**
- ❑ **How it works**

Isomorphic representations of data



First-class Isomorphisms

Vectors DSL

Array[T]

≈ (iso)

```
class DenseVec[T](val coords: Array[T])
  extends Vec[T]
{ ... }
```

```
trait Vec[T] {
  def length: Int
  def coords: Array[T]
  ...
}
```

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

≈ (iso)

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

Matrix DSL

Array[Array[T]]

≈ (iso)

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

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

```
class SparseMatr[T](say about why
  val rows: Array[(Array[Int],Array[T])],
  val nCol: Int) extends Matr[T]
{ ... }
```

≈ (iso)

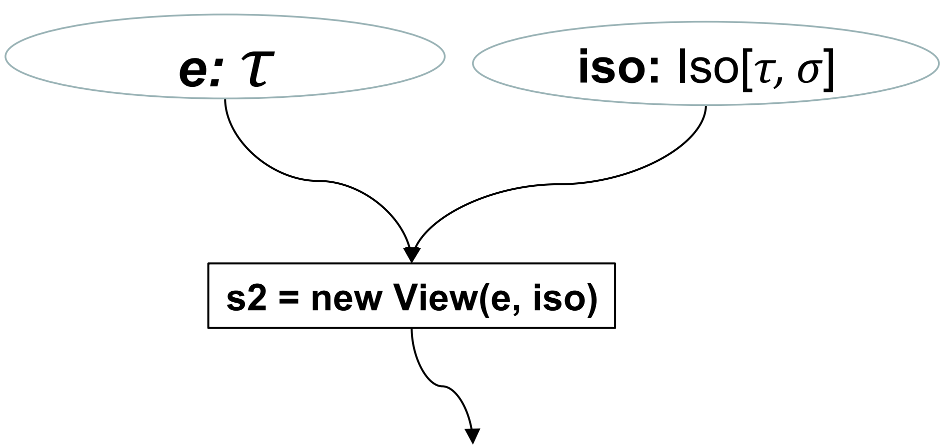
(Array[(Array[Int],Array[T])], Int)

Generic composition of isomorphisms

IDEA: Let's build an isomorphisms for each constructor of the Core language

```
T = Unit | Int | Float | Boolean
  | (T1,T2)
  | Either[T1,T2]
  | Array[T]
  | T1 => T2
```

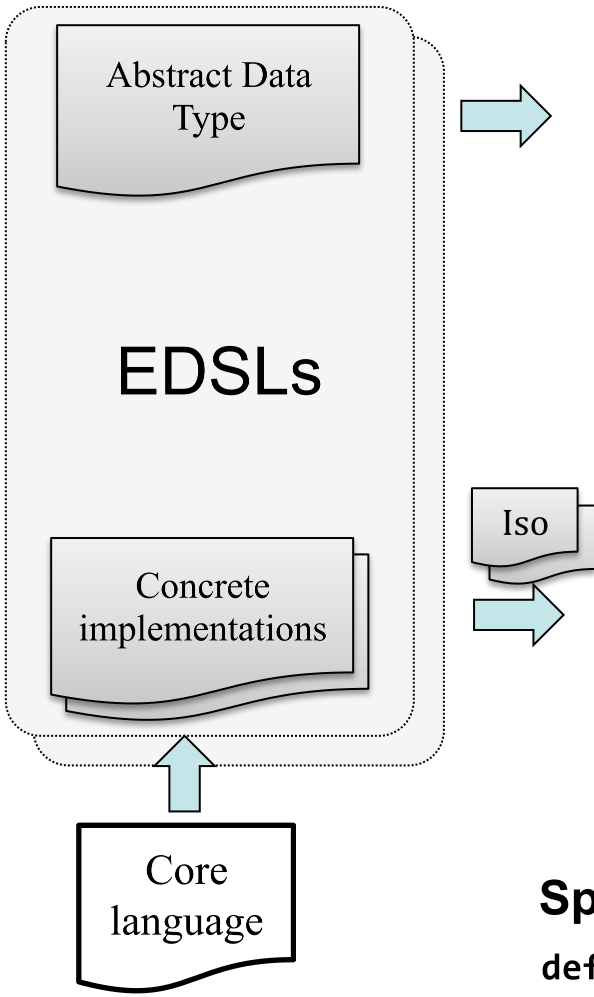
$$\frac{\Gamma \vdash e : \tau, iso : \mathtt{Iso}[\tau, \sigma]}{\Gamma \vdash e \lhd iso : \sigma}$$




```
class Iso×[A1, A2, B1, B2](
    val iso1: Iso[A1, B1], val iso2: Iso[A2, B2])
  extends Iso[A1 × A2, B1 × B2] {
  def to(a:A1 × A2) = (iso1.to(fst(a)), iso2.to(snd(a)))
  def from(b:B1 × B2)= (iso1.from(fst(b)), iso2.from(snd(b)))
}
class Iso+[A1, A2, B1, B2](
    val iso1: Iso[A1, B1], val iso2: Iso[A2, B2])
  extends Iso[A1 + A2, B1 + B2] {
  def to(a:A1 + A2) = case a of {
    l·a1 → l·iso1.to(a1); r·a2 → r·iso2.to(a2)
  }
  def from(b:B1 + B2) = case b of {
    l·b1 → l·iso1.from(b1); r·b2 → r·iso2.from(b2)
  }
}
class Iso_arr[A, B](val iso: Iso[A, B])
  extends Iso[Array[A],Array[B]] {
  def to(as: Array[A]) = as.map(iso.to)
  def from(bs: Array[B]) = bs.map(iso.from)
}
class Iso→[A1, A2, B1, B2](
    val iso1: Iso[A1, B1], val iso2: Iso[A2, B2])
  extends Iso[A1 → A2, B1 → B2] {
  def to(f:A1 → A2) = b ⇒ iso2.to(f(iso1.from(b)))
  def from(g:B1 → B2)= a ⇒ iso2.from(g(iso1.to(a)))
}
```

NOTE: These isomorphisms are defined in the same language and thus they are first class citizents in the framework

Alternative specialized versions




```
def mvm(m: Matr[T], vec: Vec[T]): Vec[T] =
  Vec(m.rows.map(v => v.dot(vec)))
```

Isomorphic Specialization

Specialization 1

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

Specialization 2

```
def smsvm_spec(
  m: (Array[(Array[Int], Array[T])], Int),
  v: (Array[Int],Array[T],Int)): Array[T] =
{
  val (rows, nCol) = m
  rows.map((is,vs) =>
     dotSV(is, vs, v._1, v._2))
}
```

Why it matters

All matrices: 10^4 x 10^4 elements
S_m is matrix sparseness (% of zeros)
S_v is vector sparseness

S_m	S_v	**dmdv**	**dmsv**	**smdv**	**smsv**
0%	0%	11389	14740	14827	53348
10%	10%	11408	13326	13253	44376
50%	50%	12944	7443	7428	21788
90%	90%	13093	1682	1546	3548
99%	99%	13251	227	167	280
0%	50%	11483	7466	14758	27987
50%	0%	12946	14852	7462	42471
10%	90%	13907	1659	14029	9728
90%	10%	14304	14581	1608	27586

Execution time in milliseconds running as Scala program.

Without abstraction elimination and optimizations

1) Isomorphic Specialization produces version in Core language with immutable arrays
2) LMS compiler optimizes purely functional array operations (deforestation, loop fusion etc.)

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

How do you know this is bad choice?

~ 20x - 40x performance improvement

A Remarkable Property of Isomorphic Specialization

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

⇩ *IS*

$\mathbf{P'}: X' \to Y'$
such that
the diagram
commutes

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

A, B – domain types

A′, B′ – Core language types

$\approx_A.to$ *– wrapper*

$\approx_B.from$ *– extractor*

Our conjecture (we haven't proved it formally):

For all $P: A \to B, \quad A \approx_A A', \; B \approx_B B'$ *there exists* $\mathbf{P'}: X' \to Y'$
such that $P' = SE_{RW}[\, \approx_B.from \circ P \circ \approx_A.to \,]$ *and* $P' \in Core$

It holds for many non-trivial examples.

Agenda

- ❑ Isomorphic Specialization in a Nutshell
- ❑ Why it matters
- ❑ First-class Isomorphisms
- ❑ How it works

Staged method invocation with graphs

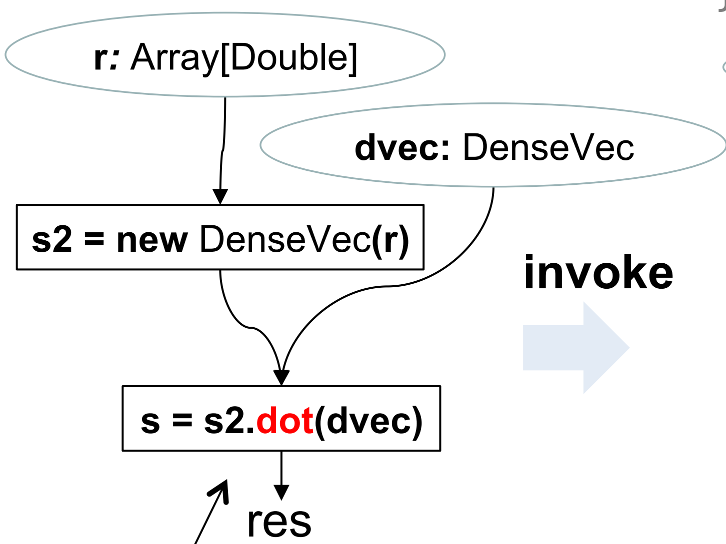
IDEA: let's implement semantics of virtual method call at staging time using IR nodes as class instances

During staging of mvm we have the following redex

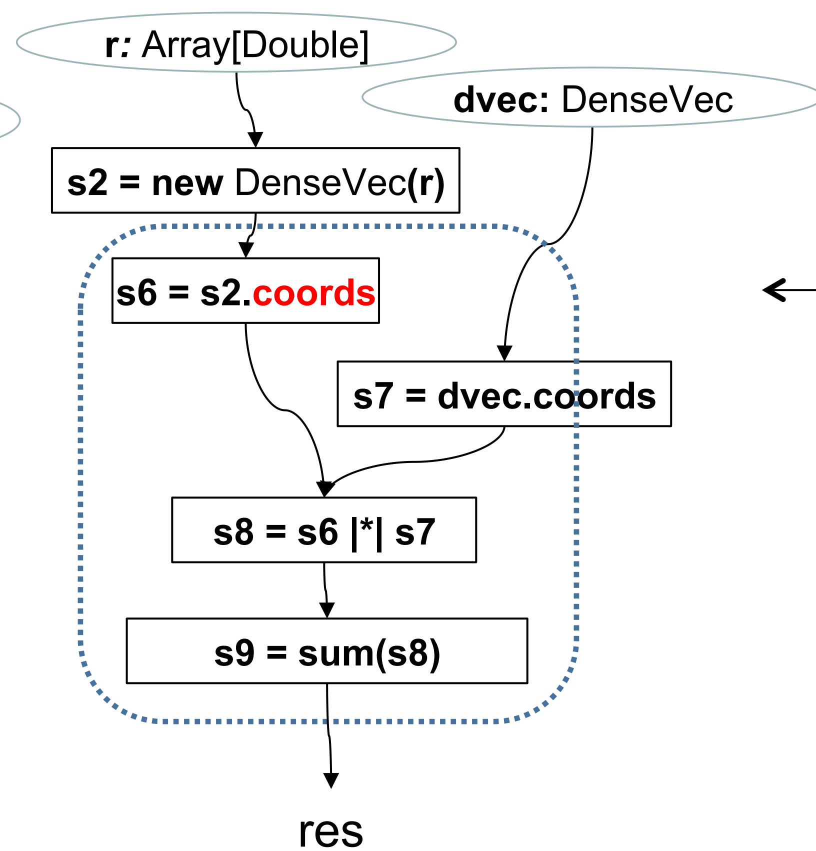
```
val r: Array[Double]
val dvec: DenseVec
new DenseVec(r).dot(dvec)
```

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

Method selected dynamically with respect to semantics of virtual method call of FJ language



Invocation is possible because s2 is defined and we can invoke the method of the concrete instance (which is the node of the graph).



Pattent matching on types during staged invocation. Even for undefined symbols.

Invocation is NOT possible. This method call is reified in the graph, whereas if invocation IS possible then the node is replaced with invocation result and the graph is rewired.

Take it home

1. **D**evelop an algorithm using abstract data types of embedded DSLs

2. **I**dentify isomorphic representations in the Domain

3. **I**mplement abstract data types using a Core language and isomorphic representations

4. **G**enerate representation-specific implementations in the Core Language

5. **U**se optimizing compiler of the Core language

6. **C**heck it out (https://github.com/scalan)

Thank you

github.com/scalan