How to select the best graph representation for a given task

Alexander Slesarenko
Alexander Filippov
Alexey Romanov
Yuri Zotov

GraphHPC’15, March 5, Moscow
Let’s consider the task

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

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.

We can retrieve rows from the matrix as an array of vectors.

Abstract data types

\[
\text{def mvm}(m: \text{Matr}, \text{vec: Vec}): \text{Vec} = \\
\{ \\
\text{Vec}(m.\text{rows}.\text{map}(r => r.\text{dot}(\text{vec})))
\}
\]
The problem: which representation is better?

Dense Matrix

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1.0 & 0 & 2.0 & 0 \\
1 & 3.0 & 4.0 & 5.0 & 0 \\
2 & 0 & 0 & 0 & 6.0 \\
\end{array}
\]

Sparse Matrix

Flat Sparse Matrix

- \text{segLen}:
  - 2
- \text{columnIdx}:
  - 3
  - 0
  - 2
  - 0
  - 1
  - 2
  - 3
- \text{value}:
  - 1.0
  - 2.0
  - 3.0
  - 4.0
  - 5.0
  - 6.0
- \text{nCols}:
  - 4

\text{nCols}:
- 4
Why it is a problem?

Consider matrix $10^4 \times 10^4$

$S_m$ - matrix sparseness (% of zeros)

$S_v$ - vector sparseness

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

Execution time in milliseconds

How do you know this is a bad choice?
The solution

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

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

**Automatically: generate, run and measure performance**

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

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

def smsvm(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 smdvm(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`
Two stages of compilation

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

Isomorphic Specialization into Core language

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

Core language compilation with loop fusion, deforestation etc.

```scala
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
}
```
First-class Isomorphic Specialization by Staged Evaluation

Alexander Slesarenko  
Shannon Laboratory, Huawei Technologies, Moscow, Russia  
alexander.slesarenko@huawei.com

Alexander Filippov  
Shannon Laboratory, Huawei Technologies, Moscow, Russia  
filippov.alexander@huawei.com

Alexey Romanov  
Shannon Laboratory, Huawei Technologies, Moscow, Russia  
alexey.romanov@huawei.com

Abstract

The state of the art approach for reducing complexity in software development is to use abstraction mechanisms of programming languages such as modules, types, higher-order functions etc. and develop high-level frameworks and domain-specific abstractions. Abstraction mechanisms, however, along with simplicity, introduce also execution overhead and often lead to significant performance degradation. Avoiding abstractions in favor of performance, on the other hand, increases code complexity and cost of maintenance.

We develop a systematic approach and formalized framework for implementing software components with a first-class specialization capability. We show how to extend a higher-order functional language with abstraction mechanisms carefully designed to provide automatic and guaranteed elimination of abstraction overhead.

We propose staged evaluation as a new method of program staging and show how it can be implemented as zipper-based traversal of program terms where one-hole contexts are generically constructed from the abstract syntax of the language.

We show how generic programming techniques together with staged evaluation lead to a very simple yet powerful method of isomorphic specialization which utilizes first-class definitions of isomorphisms between data types to provide guarantee of abstraction elimination.

Abstraction mechanisms (such as module systems, classes, interfaces, etc.). These mechanisms are often used to create domain-specific languages (DSLs) which allow a higher level of abstraction for programs in a given domain (e.g. Spark [33] can be considered as a DSL for distributed programming).

However, these mechanisms generally also introduce execution overhead (often called abstraction regret [4, 21] or abstraction penalty) and the trade-off between abstraction and performance is often difficult.

Modern advances in compilation techniques, such as just-in-time compilation and whole program optimization generally can’t eliminate the overhead completely and don’t scale well with the size of the program.

A recent trend is development of DSL-centric frameworks where abstractions can be introduced and software can be built without abstraction penalty [5, 23, 24], though it may require development of special tools [3].

In such frameworks, DSL compilers allow mapping of problemspecific abstractions directly to low-level architecture-specific programming models such as [12, 20]. However, the development of DSLs is difficult by itself, and adding a compilation stage considerably increases this difficulty.

While compiling DSLs is a promising approach, we believe that
MST

```scala
def MST_prim(g: Graph, startFront: Front): Coll[Int] = {
  def stopCondition(front: Front, tree: Coll[Int]) =
    (g.outEdgesOf(front).length === 0)

  def doStep(front: Front, tree: Int) = {
    val outEdges = g.outEdgesOf(front)
    val (_, (minFrom, minTo)) = outEdges
      .map(e => (e.value, (e.fromId, e.toId)))
      .reduce(MinWeightMonoid)
    (front.append(minTo), tree.update(minTo, minFrom))
  }

  val initTree = replicate(g.vertexNum, UNVISITED)
  val (_,resTree) = from(startFront, initTree)
    .until(stopCondition)(doStep)
  return resTree
}

Let's do the same trick with MST
```
Which Graph representation is better?

```scala
trait Graph {
  def vertexNum: Int
  def outDegrees: Coll[Int]
  def neighbors: Coll[Int]
  def nonZeroEdgeValues: Coll[Double]
  def edgeValues: Coll[Double]
}
```
Which Front representation is better?

```
trait Front {
  def contains(v: Int): Boolean
  def append(v: Int): Front
  def items: Coll[Int]
}
```
## Specialized versions of MST

<table>
<thead>
<tr>
<th><strong>MST.scala</strong></th>
<th>FlatAdjacencyList</th>
<th>AdjacencyList</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitsAndArray</td>
<td>MST\textsubscript{11}.scala</td>
<td>MST\textsubscript{12}.scala</td>
<td>...</td>
</tr>
<tr>
<td>HashSet</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BitsAndList</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>MST.cpp</strong></th>
<th>FlatAdjacencyList</th>
<th>AdjacencyList</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitsAndArray</td>
<td>MST\textsubscript{11}.cpp</td>
<td>MST\textsubscript{12}.cpp</td>
<td>...</td>
</tr>
<tr>
<td>HashSet</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BitsAndList</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Specialized versions of MST

<table>
<thead>
<tr>
<th></th>
<th>MST.scala</th>
<th>FlatAdjacencyList</th>
<th>AdjacencyList</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitsAndArray</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>HashSet</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>BitsAndList</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>MST.cpp</th>
<th>FlatAdjacencyList</th>
<th>AdjacencyList</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitsAndArray</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>HashSet</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>BitsAndList</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>
Evaluation (Scala, JVM, RMAT, scale 10, JMH)

<table>
<thead>
<tr>
<th>MST.scala</th>
<th>FlatAdjacencyList Graph</th>
<th>AdjacencyMatrix Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>density, %</td>
<td>BitsAndArray</td>
<td>BitsAndList</td>
</tr>
<tr>
<td>0,39</td>
<td>79,01</td>
<td>73,58</td>
</tr>
<tr>
<td>0,78</td>
<td>158,85</td>
<td>161,25</td>
</tr>
<tr>
<td>1,56</td>
<td>232,62</td>
<td>227,48</td>
</tr>
<tr>
<td>3,13</td>
<td>280,84</td>
<td>290,60</td>
</tr>
<tr>
<td>6,25</td>
<td>330,87</td>
<td>354,82</td>
</tr>
<tr>
<td>12,50</td>
<td>389,83</td>
<td>429,32</td>
</tr>
<tr>
<td>25,00</td>
<td>439,71</td>
<td>480,34</td>
</tr>
<tr>
<td>50,00</td>
<td>493,87</td>
<td>538,96</td>
</tr>
<tr>
<td>100,00</td>
<td>550,48</td>
<td>589,43</td>
</tr>
</tbody>
</table>
### Evaluation (JNI/C++, RMAT, scale 10, JMH)

<table>
<thead>
<tr>
<th>MST.scala</th>
<th>FlatAdjacencyList Graph, BitsAndArray Front</th>
<th>AdjacencyMatrix Graph, BitsAndArray Front</th>
</tr>
</thead>
<tbody>
<tr>
<td>density, %</td>
<td>JNI/C++</td>
<td>JNI/Scala, JVM</td>
</tr>
<tr>
<td>0,39</td>
<td>58,93</td>
<td>79,01</td>
</tr>
<tr>
<td>0,78</td>
<td>115,15</td>
<td>158,85</td>
</tr>
<tr>
<td>1,56</td>
<td>151,77</td>
<td>232,62</td>
</tr>
<tr>
<td>3,13</td>
<td>192,11</td>
<td>280,84</td>
</tr>
<tr>
<td>6,25</td>
<td>221,15</td>
<td>330,87</td>
</tr>
<tr>
<td>12,50</td>
<td>255,74</td>
<td>389,83</td>
</tr>
<tr>
<td>25,00</td>
<td>279,78</td>
<td>439,71</td>
</tr>
<tr>
<td>50,00</td>
<td>319,85</td>
<td>493,87</td>
</tr>
<tr>
<td>100,00</td>
<td>344,02</td>
<td>550,48</td>
</tr>
</tbody>
</table>

**Graphs:**
- **Left:** Adjacency List (adjlist-cpp vs adjlist-scala)
- **Right:** Adjacency Matrix (adjmatrix-cpp vs adjmatrix-scala)
Method summary

1. Develop an algorithm using domain-specific abstract data types (e.g. Graph, Front, Collection, etc.)

2. Identify *isomorphic* representations of domain objects (e.g. AdjListGraph, AdjMatrGraph, etc.)

3. Implement domain-specific interfaces using concrete representations (e.g. AdjListGraph implements Graph, etc.) and primitives of the Core language

4. Automatically specialize the algorithm with respect to all the alternative data representations

5. Compile specializations into target platform (Java, C++) using domain-specific compilation and select the best one.
Further reading


2. A. Slesarenko, Lightweight Polytypic Staging of DSLs in Scala, META’12


5. LMS (http://scala-lms.github.io/)

Download these and related Scalan publications on my home page (http://pat.keldysh.ru/~slesarenko/)
Join us or get involved

1. Check out source code at github.com/scalan

2. Ask questions on Google Group
   https://groups.google.com/d/forum/scalan

3. Follow us on twitter (@avslesarenko, @alexey_r)
Thank you

github.com/scalan