Language and Compiler Support for
Data-Centric Optimizations

Vlad Ureche, EPFL <vlad.ureche@epfl.ch>

① Data Representation Problem

In high-level languages, such as Scala, developers write their data structures using generic components from the library:

Library Class

```
class Vector[+A] extends Sequence[A] with ... {
  ...
}
```

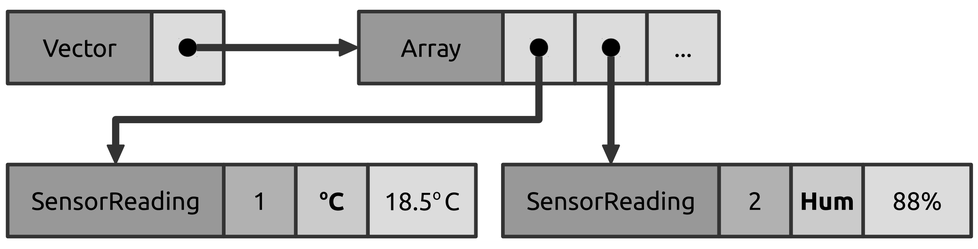
Library components are freely mixed with custom data structures. For example, with objects storing sensor readings:

Source Code

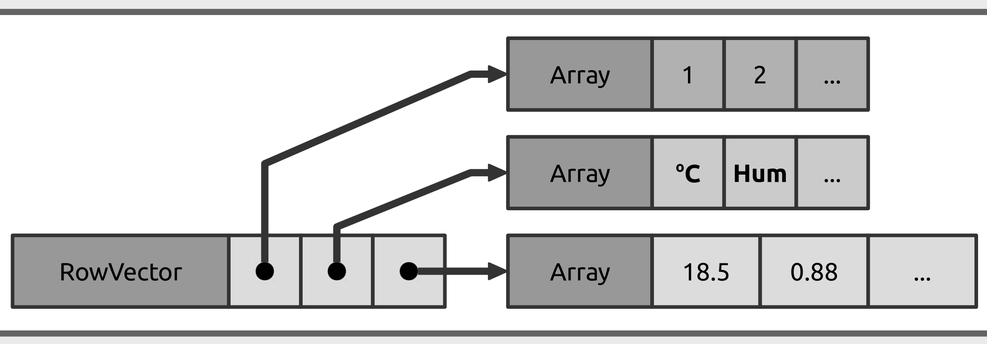
```
case class SensorReading(timestamp: Int,
                         sensor: Int,
                         value: Double)
```

Programmers appreciate the ability to mix data structures, as it increases productivity. Yet, without realizing, they give up performance, as the mixed data structures have suboptimal memory representations.

In our example, traversing a `Vector[SensorReading]` object requires a pointer dereference for each element:



Most programmers can immediately give a better layout:



But, currently, there is no way for them to modify the data representation, as it is fixed by the compiler.

② Challenges

Optimizing the data representation is difficult:

Productivity Transforming the code by hand is an option, but it is tedious, error prone and harms long-term maintenance.

Context dependency The best layout for a piece of data depends on how it's going to be manipulated and where it's going to be stored. Only the programmer has this information.

Open world assumption New code, which is not aware of the optimized representation, can loaded at any time, thus introducing inconsistencies. Contrarily, most DSLs assume a closed world: only the predefined data structures can be used in the program.

Combined, these three problems make optimizing the data representation very difficult.

③ Data-centric Optimizations

Data-centric Optimizations overcome the challenges:

Productivity Our technique extends the Scala compiler to allow transforming the data representation as part of the compilation pipeline, based on type system information. Thus, programmers can freely mix and match their data structures.

Context dependency Since the programmer is in the unique position of deciding the best data layout, we allow them to define it directly in Scala, without any special API:

Optimized Data Structure

```
class RowVector(timestamps: Array[Int],
                sensors: Array[Int],
                values: Array[Double])
```

and to instruct the compiler how to use it:

Transformation Description Object

```
object RowOpt extends Transformation { ... }
```

Open world assumption We can enclose a scope where the transformation occurs. Inside, the code uses the optimized representation, while outside, as soon as the value leaks, it is converted to the original encoding:

Source Code

```
transform(RowOpt) {
  def avgTemp(reads: Vector[...]): Double = ...
}
```

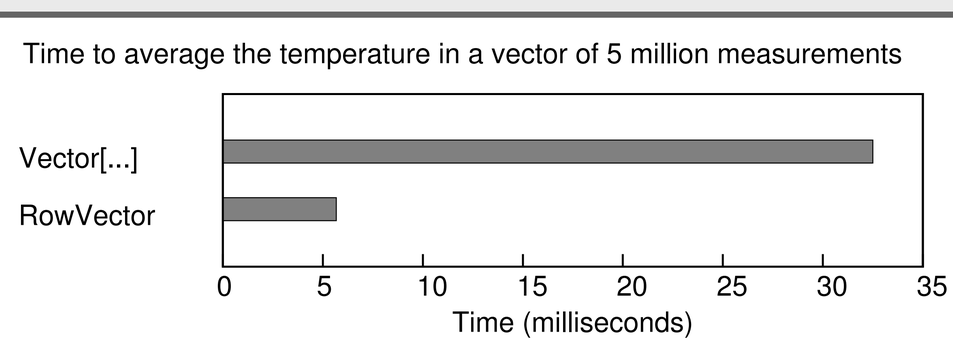
④ Composability

Transformation scopes can compose (communicate using the optimized data layout) across class boundaries and even across separate compilation. If instructed, the compiler can warn when expensive data transformations are necessary:

```
warning: When calling method avgTemp, the argument
'data' needs to be converted to the 'RowVector'
representation, which may incur some overhead:
      avgTemp(data)
              ^
```

By wrapping the code shown by the warning in the `transform(RowOpt){...}` scope, the slowdown is avoided, as both the caller and callee will use the optimized layout.

⑤ Benchmarks



More benchmarks, showing speedups of up to 20x are shown on the project website: `scala-ildl.org`