Функциональные программы с производительностью императивных

Dmitry Bushev @4e6

www.huawei.com

rifvrn, September 11, 2015

Huawei Technologies Co., Ltd.

HUAWEI

Outline

- ➔ Motivation
- ➔ Compilers, Scala
- ➔ Scalan domain-specific compiler
- ➔ Example

Motivation

What language you recall first, when think about performance?

Motivation

What language you recall first, when think about performance?

`C, C++ ?`

Motivation

What language you recall first, when think about performance?

`C, C++, Java, C# ?`

Motivation

What language you recall first, when think about performance?

```
C, C++, Java, C#, Scala, F# ?
```

Motivation

What language you recall first, when think about performance?

C, C++, Java, C#, Scala, F#, Python, Ruby ?

Motivation

What language you recall first, when think about performance?

`C, C++, Java, C#, Scala, F#, Python, Ruby ?`

Motivation

What language you recall first, when think about performance?

`C, C++, Java, C#, Scala, F#, Python, Ruby ?`

(not only) functional languages overhead:

→ Operates with **objects**, not primitives

- Allocations
- Boxing
- GC pressure

→ Also things like **virtual calls** and other cockroaches

Motivation

What language you recall first, when think about performance?

`C, C++, Java, C#, Scala, F#, Python, Ruby ?`

Programming in functional language:

➔ we don't think about performance

Motivation

What language you recall first, when think about performance?

C, C++, Java, C#, Scala, F#, Python, Ruby ?

Programming in functional language:

→ we don't think about performance

- and we should not!

Motivation

What language you recall first, when think about performance?

`C, C++, Java, C#, Scala, F#, Python, Ruby ?`

Programming in functional language:

→ we don't think about performance

- and we should not!
  - but if we do ...

Compilers for the rescue!

Optimizing Compilers

General purpose optimizations:

➔ loop fusion

Optimizing Compilers

General purpose optimizations:

- ➔ loop fusion
- ➔ loop unrolling

Optimizing Compilers

General purpose optimizations:

→ loop fusion

→ loop unrolling

→ if expression optimization

Optimizing Compilers

General purpose optimizations:

- ➔ loop fusion
- ➔ loop unrolling
- ➔ if expression optimization
- ➔ function inlining

Optimizing Compilers

General purpose optimizations:

→ loop fusion

→ loop unrolling

→ if expression optimization

→ function inlining

→ tail call elimination

→ … (a lot, lot more)

Scala Compiler

`scalac` compiler optimizations:

➔ create objects when it is really necessary

Scala Compiler

`scalac` compiler optimizations:

- ➔ create objects when it is really necessary
- ➔ function inlining

Scala Compiler

`scalac` compiler optimizations:

- ➔ create objects when it is really necessary
- ➔ function inlining
- ➔ tail recursion

Scala Compiler

`scalac` compiler optimizations:

- ➔ create objects when it is really necessary
- ➔ function inlining
- ➔ tail recursion
- ➔ tries to help JVM

Scala Compiler

`scalac` compiler optimizations:

- ➔ create objects when it is really necessary
- ➔ function inlining
- ➔ tail recursion
- ➔ tries to help JVM
- ➔ … and lots of other tricky things

see Lucas Rytz' JVM Backend and Optimizer in Scala 2.12

Range.foreach problem

- `Range.foreach` is much slower than equivalent while loop

```
(0 until 10000) foreach { i => ... }
```

vs

```
var i = 0
while (i < 10000) { ... }
```

- Why?

Range.foreach problem

- `Range.foreach` is much slower than equivalent while loop

```
(0 until 10000) foreach { i => ... }    vs    var i = 0
                                              while (i < 10000) { ... }
```

- Why?

```
class Range {
  def foreach(f: Int => Unit) =
    { var i = start; while(i < end) { f.apply(i); i += step } }
}
```

- f.apply(i) is a virtual call, not inlined because called with different f's.

Range.foreach problem

- `Range.foreach` is much slower than equivalent while loop

```
(0 until 10000) foreach { i => ... }    vs    var i = 0
                                              while (i < 10000) { ... }
```

- Why?

```
class Range {
  def foreach(f: Int => Unit) =
    { var i = start; while(i < end) { f.apply(i); i += step } }
}
```

- `f.apply(i)` is a virtual call, not inlined because called with different f's.
- Improved in **Scala 2.12** by new optimizer
- Can be improved by macro libraries, see Scala-Blitz

Range.foreach problem

- `Range.foreach` is much slower than equivalent while loop

```
(0 until 10000) foreach { i => ... }    vs    var i = 0
                                              while (i < 10000) { ... }
```

- Why?

```
class Range {
  def foreach(f: Int => Unit) =
    { var i = start; while(i < end) { f.apply(i); i += step } }
}
```

- f.apply(i) is a virtual call, not inlined because called with different f's.
- Improved in **Scala 2.12** by new optimizer
- Can be improved by macro libraries, see Scala-Blitz

Can we do better?

Domain-Specific Optimizations

High-level **idea**:

Domain-Specific Optimizations

High-level **idea**:

➔ We know our domain

Domain-Specific Optimizations

High-level **idea**:

➔ We know our domain

➔ We can pass domain knowledge to compiler

Domain-Specific Optimizations

High-level **idea**:

➔ We know our domain

➔ We can pass domain knowledge to compiler

➔ Compiler will do all the dirty work

Domain-Specific Optimizations

High-level **idea**:

- ➔ We know our domain
- ➔ We can pass domain knowledge to compiler
- ➔ Compiler will do all the dirty work
- ➔ Profit!

Domain-Specific Optimizations

High-level **idea**:

- ➔ We know our domain
- ➔ We can pass domain knowledge to compiler
- ➔ Compiler will do all the dirty work
- ➔ Profit!

Looks good to me

Example Domains

Domain	Abstractions
→ Collections	→ List, Set, Map, …
→ Linear Algebra	→ Matrix, Vector, Row
→ Machine Learning	→ Model, Regression, Feature
→ SQL	→ DataSource, Query
→ Graphs	→ Vertex, Edge
→ Your Domain	

Example Domains

Domain	Abstractions
→ Collections	→ List, Set, Map, …
→ Linear Algebra	→ Matrix, Vector, Row
→ Machine Learning	→ Model, Regression, Feature
→ SQL	→ DataSource, Query
→ Graphs	→ Vertex, Edge
→ Your Domain	

But we need a Domain-Specific Compiler…

Scalan

A framework for development of domain-specific compilers in Scala.

Definitions:

- Domain

  Set of abstractions and functions on top of core language

- Core language

  Set of basic primitives, that we know how to compile

- Backend

  Compile core language to **executable kernels**

How it Works

➔ Define domain

Implement your abstractions in terms of core language

Program with domain objects

How it Works

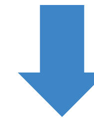
- ➔ Define domain

  Implement your abstractions in terms of core language

- ➔ Define a core language

  Currently we have a simple functional language with immutable arrays

Program with domain objects



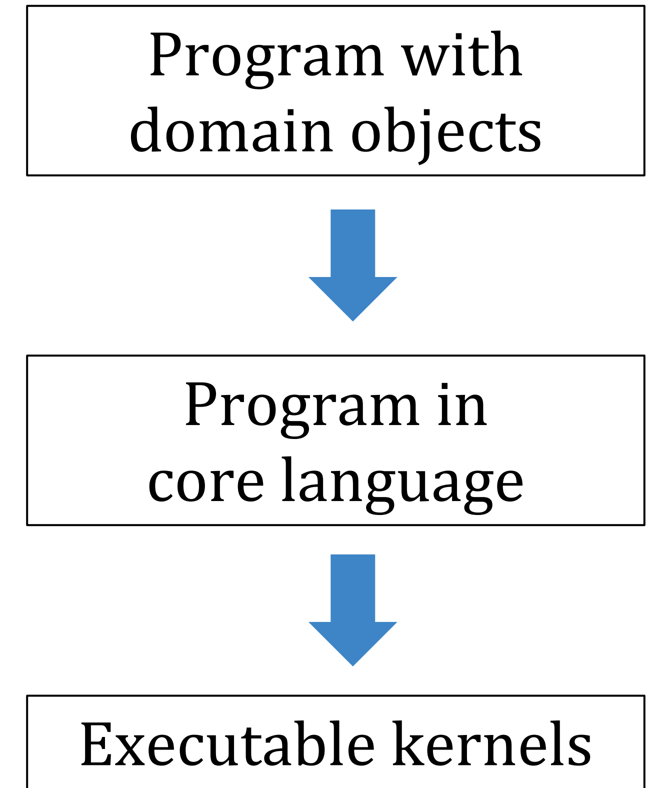
Program in core language

How it Works

- ➔ Define domain

  Implement your abstractions in terms of core language
- ➔ Define a core language

  Currently we have a simple functional language with immutable arrays
- ➔ Implement backend for core language



Scalan: Core Language

Functional language with immutable arrays

```
T = Unit | Int | Float | Boolean        // base types
  | (T,T)                               // product of types
  | Either[T,T]                         // sum of types
  | Array[T]                            // array of values T

trait Array[T] {
  def length: Int
  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 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])
...
```

Domain: Linear Algebra

→ Define domain interfaces (Matr, Vec) and implementations

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

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

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

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

→ There is a relationship (isomorphism) between constructor and domain object.

Matrix Vector Multiplication (MVM)

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

abstract data types

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

construct a new vector from an array of values

retrieve rows from the matrix as an array of vectors

when we map an Array we create a new Array

Domain-Specific Compilation

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

Domain-Specific Compilation

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

Compilation
into Core language

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

Domain-Specific Compilation

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

Compilation into Core language

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

Compilation to backend (JVM)

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

Scalan: Implemented Backends

- We can compile **Scalan** to
  - JVM
  - C++
  - C++ NDP, Nested Data Parallelism Runtime
  - MPI
  - Apache Spark
- Only JVM and C++ backends are open-source (so far)

Scalan: Implemented Domains

Domain	Abstractions
→ Linear Algebra	→ Matrix, Vector
→ Graphs	→ Vertex, Edge
→ Collections	→ Seq, List, Map, ...

These are open-source

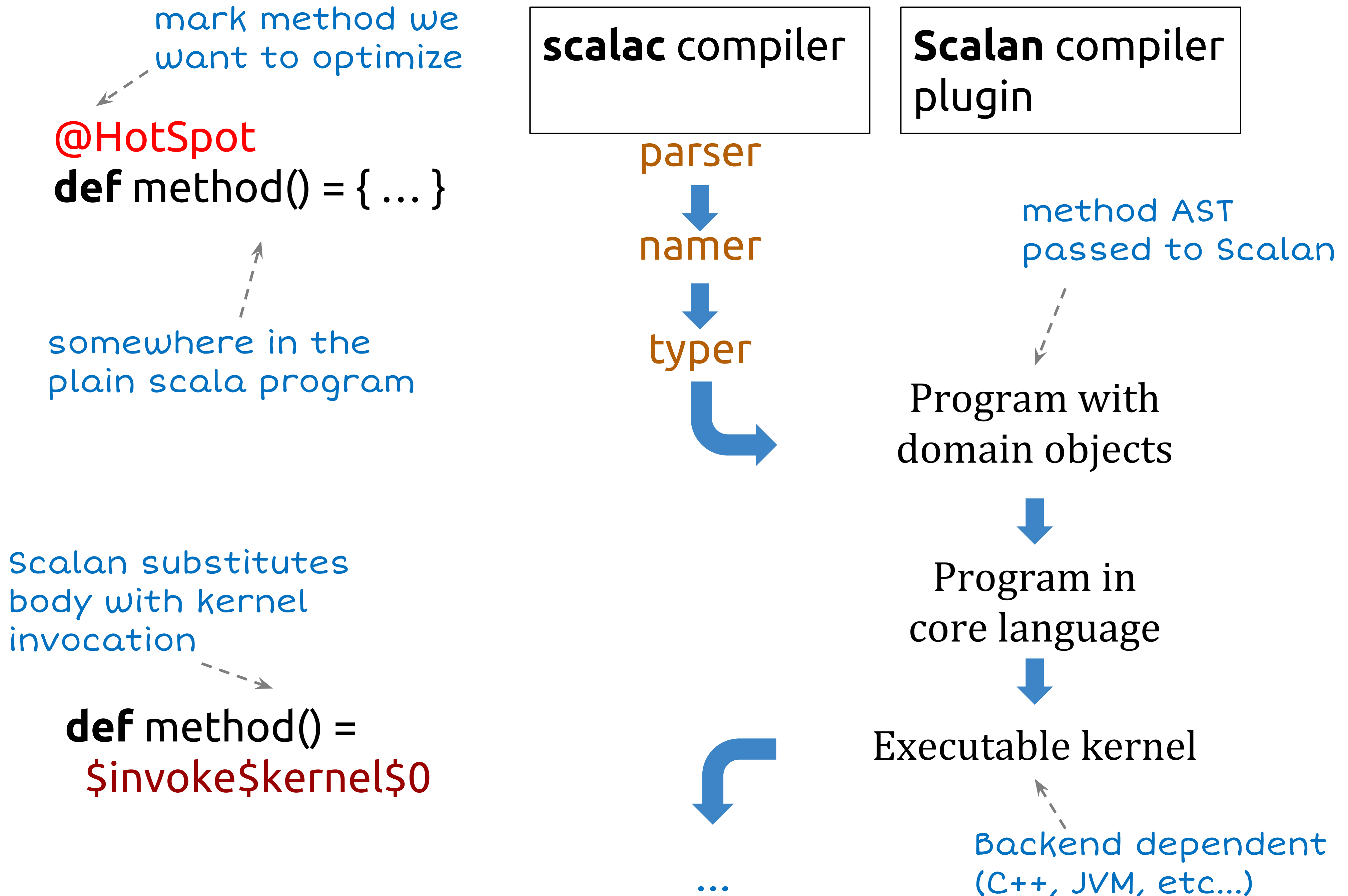
Scalan: Implemented Domains

Domain	Abstractions
→ Linear Algebra	→ Matrix, Vector
→ Graphs	→ Vertex, Edge
→ Collections	→ Seq, List, Map, ...

These are open-source

You can create your own!

Scalan: HotSpot Optimization



Summary

- ➔ Compilers do the hard work for you, to make your life easier
- ➔ General purpose optimizations
- ➔ Domain-specific optimizations
- ➔ Scalan domain-specific compiler
  - Domain
  - Core language
  - Executable kernels
- ➔ Domain knowledge is passed through constructors. There is a relationship (isomorphism) between constructor and domain object.

More Information

- The project is available at github.com/scalan/scalan
- Twitter: @avslesarenko, @alexey_r, @4e6, @maxgekk.
- Scala Days Amsterdam, June 2015. Program Functionally, Execute Imperatively: Peeling abstraction overhead from functional programs (slides/video)

More Information

- The project is available at github.com/scalan/scalan
- Twitter: @avslesarenko, @alexey_r, @4e6, @maxgekk.
- Scala Days Amsterdam, June 2015. Program Functionally, Execute Imperatively: Peeling abstraction overhead from functional programs (slides/video)

Thank you!
Questions?