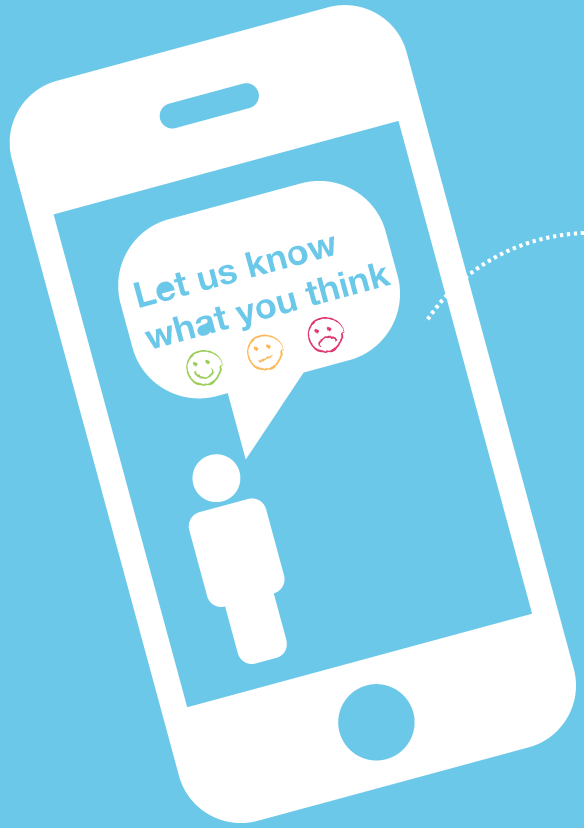
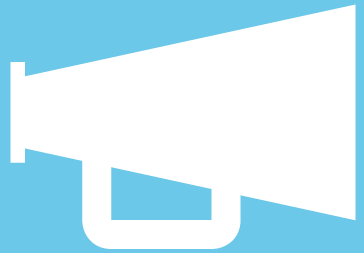
Scalan: a reasonably typed meta-programming framework in Scala

Alexander Slesarenko

SCALA DAYS BERLIN 2016

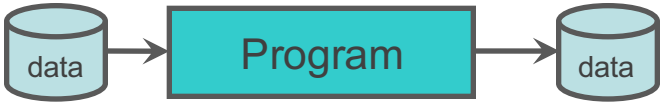




Please use the Scala Days app to rate sessions.

Join the conversation #scaladays

Meta-programming



Why meta-programming?

- avoid overhead of abstractions
- automatic code optimizations
- generate boilerplate
- develop DSLs
- tweak syntax
- etc.

Meta-programming is typically hard

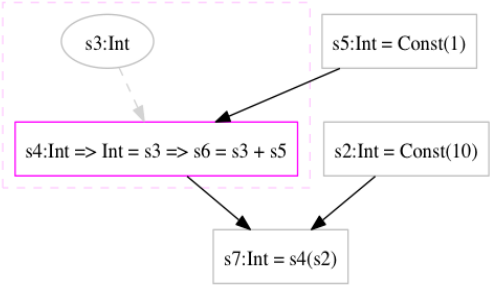
Why meta-programming is hard?

- Intermediate Representation (IR) is typically defined by Abstract Syntax
- The data has complex recursive structure
- All operations also has to be recursive

$\mathcal{T} \ni \tau$	$::=$	$\texttt{Unit} \mid \texttt{Int}$	base types
	$\mid$	$(\tau_1 \times \tau_2)$	binary product type
	$\mid$	$(\tau_1 + \tau_2)$	sum type
	$\mid$	$(\tau_1 \to \tau_2)$	function type
	$\mid$	$\texttt{Array}[\tau]$	array type
$Term \ni e$	$::=$	$l \mid x : \tau$	integer literals and vars
	$\mid$	$\lambda(x : \tau).e \parallel e_1 e_2$	functions
	$\mid$	$k\,\overline{e}$	constructors
	$\mid$	$\delta\,\overline{e}$	primitives
	$\mid$	$\textbf{case}\ e\ \textbf{of}\ \{\, p_i \to e_i \,\}$	case analysis
k	$::=$	$() \mid (\_, \_)$	unit and pair
	$\mid$	$\texttt{l}[\tau_1, \tau_2] \cdot \_ \mid \texttt{r}[\tau_1, \tau_2] \cdot \_$	injections
δ	$::=$	$\texttt{fst}\ \_ \mid \texttt{snd}\ \_$	projections
	$\mid$	$\_ \oplus \_$	binary operations
p	$::=$	$l \mid k\,\overline{x}$	patterns
v	$::=$	$l \mid k\,\overline{v} \mid \lambda(x : \tau).e$	values

The hard part is to manipulate such data structures in semantically correct way

How Scalan can help?




```
trait Collection[T] {
  def length: Rep[Int]
  …
}
trait Vector[T] {
  def items: Collection[T]
  …
}
trait Matrix[T] {
  def *(v: Vec[T]): Vec[T]
  …
}
```

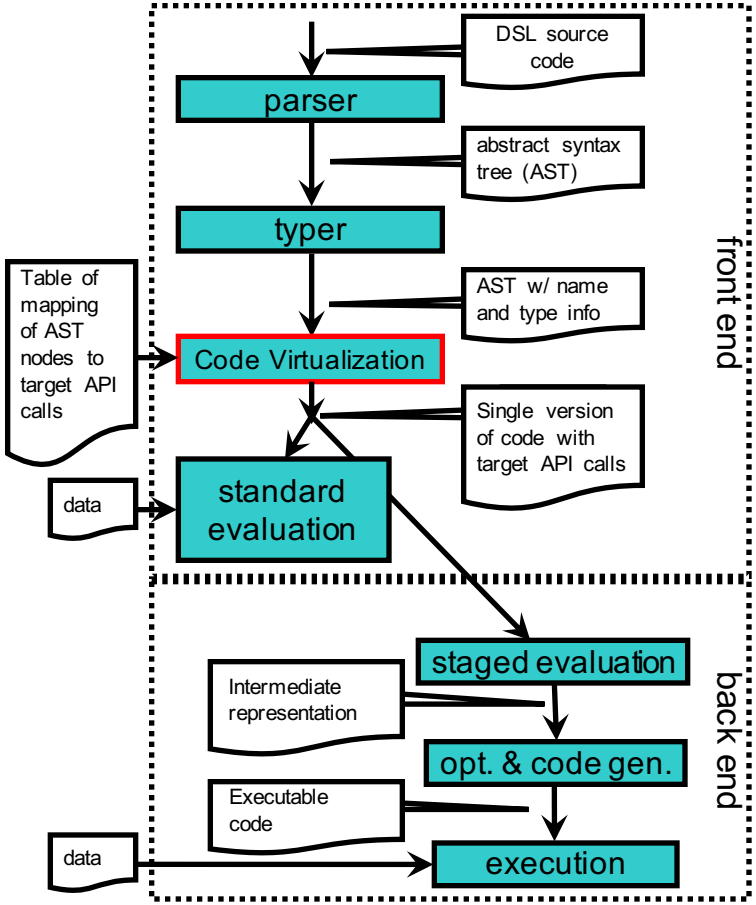
- DAG based intermediate representation (IR)
  - unified and extensible data structure
  - immutable and higher-order (HOAS)
  - configurable visualization (Graphviz)
- Composable and generic representation for
  - functions (lambdas) as DAGs
  - virtualized user-defined types (UDTs)
  - domain-specific isomorphisms
  - domain-specific converters
- Composable and generic meta-operators (later in slides)

Basic Machinery: Code Virtualization

```
def mvm(matr: Matr[T], vec: Vec[T]): Vec[T] =
  new DenseVec(
    matr.rows.map(
      { r: Vec[T] => r.dot(vec)}
    ))
```

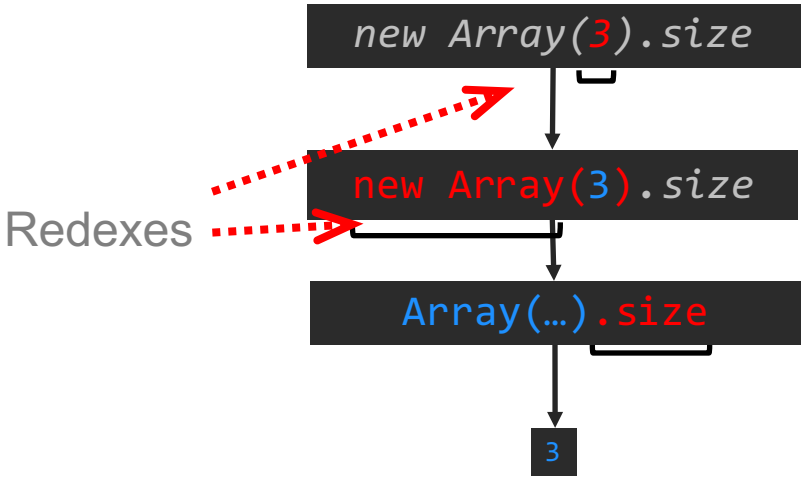
```
def mvm[T](matr: Rep[Matr[T]], vec: Rep[Vec[T]])
  (implicit emT: Elem[T]): Rep[Vec[T]] =
  DenseVec.apply(
    matr.rows.map(
      fun { r: Rep[Vec[T]] => r.dot(vec) }))
```

1) Process an abstract syntax tree of a source code
2) Associate nodes of the AST with calls of a virtualized API
3) Generate a virtualized code, which contain only the calls of the virtualized API.

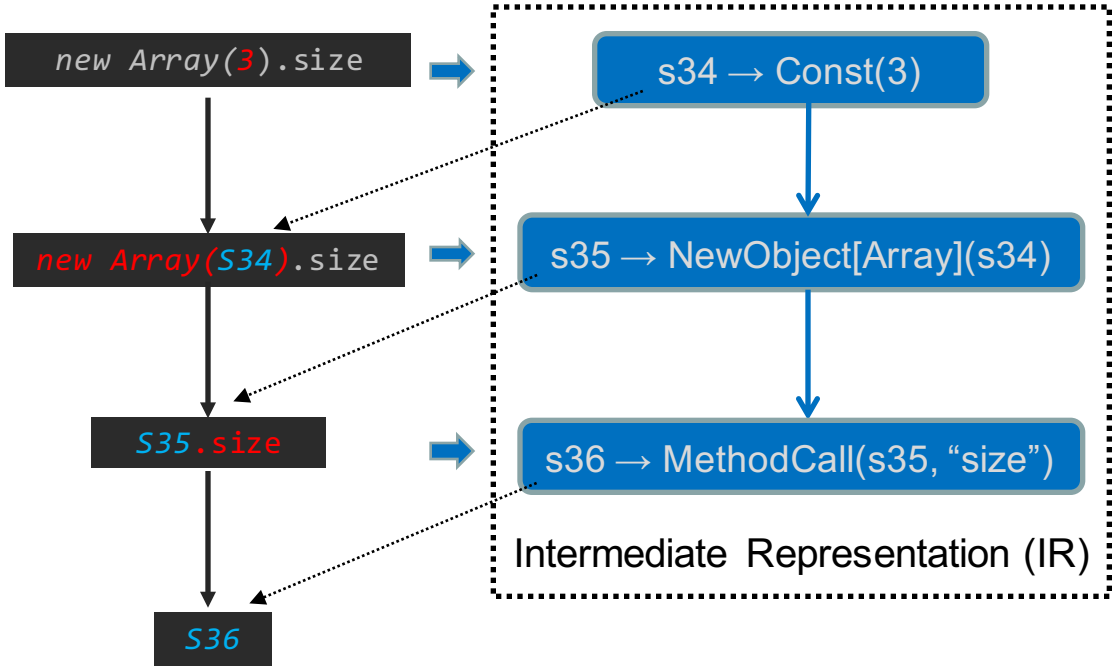


Basic Machinery: Standard Evaluation

Standard evaluation



Basic Machinery: Staged evaluation



Codegen

```
{
  val s34 = 3
  val s35 = new Array(s34)
  val s36 = s35.size
  s36
}
```

```
rule(∀n =>
  new Array(n).size ➔ n
)
```

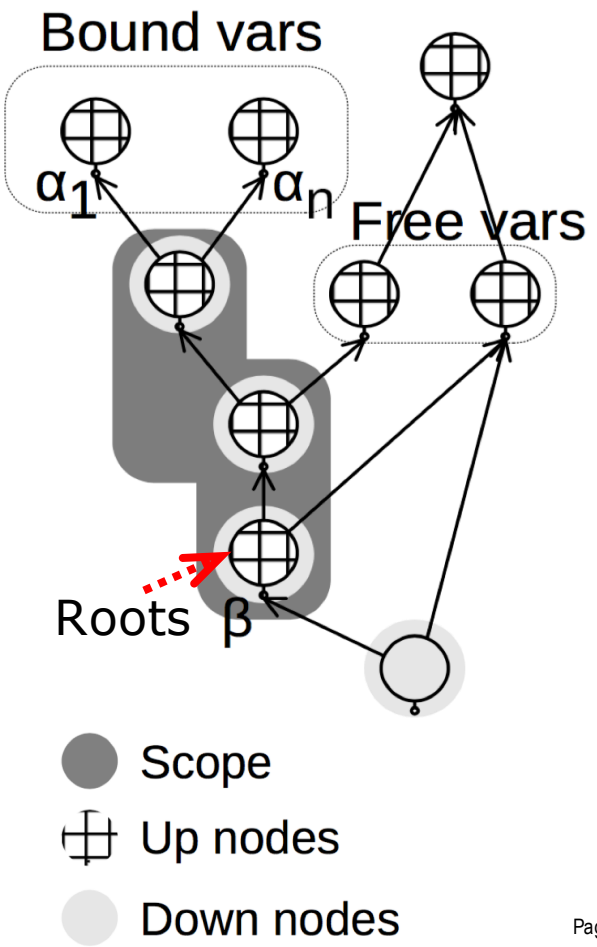
```
{
  val s34 = 3
  s34
}
```

- Staged evaluation can be understood as a *self-reproducing process*
- When a program is stage-evaluated it reproduces itself in a graph-based IR

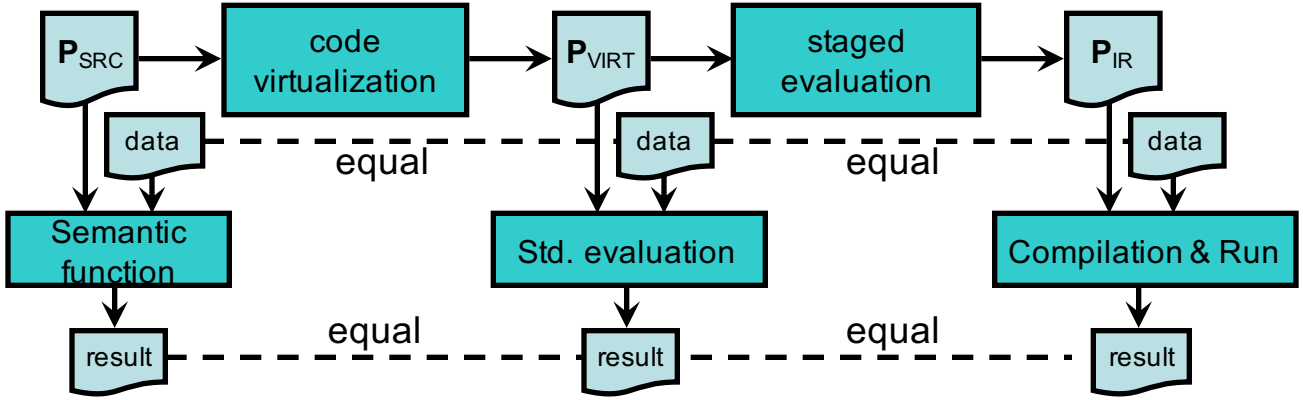
Basic Machinery: Representation of Lambdas

```
trait AstGraph {
  val boundVars: List[Sym[_]]
  val roots: List[Sym[_]]
  lazy val freeVars: Set[Sym[_]]
  lazy val schedule: Seq[TableEntry[_]]
  …
}
```

```
class Lambda[A, B](
  val f: Option[Sym[A] => Sym[B]]
  val x: Sym[A])(val y: Sym[B] = f(x),
  val mayInline: Boolean)
  extends BaseDef[A => B] with AstGraph
{
  val boundVars = List(x)
  val roots = List(y)
  …
}
```



Semantics of the Virtualized Code



Virtualized code is semantically connected to the source code in two ways:

1) Code Virtualization produces $\mathbf{P_{VIRT}}$ which is equivalent to Semantic Function with respect to Std. Evaluation

2) Staged Evaluation produces graph-based IR $\mathbf{P_{IR}}$, which is equivalent to the Std. Evaluation with respect to Compilation & Run

While $\mathbf{P_{VIRT}}$ can be executed directly (as JVM bytecode) Compiler is required to generate an executable machine code from $\mathbf{P_{IR}}$

Transformations

$\mathbf{P_{SRC}}$ – source code of the program

$\mathbf{P_{VIRT}}$ = CodeVirtualization($\mathbf{P_{SRC}}$)

$\mathbf{P_{IR}}$ = StagedEvaluation($\mathbf{P_{VIRT}}$)

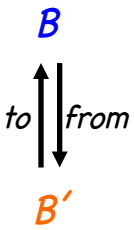
Property

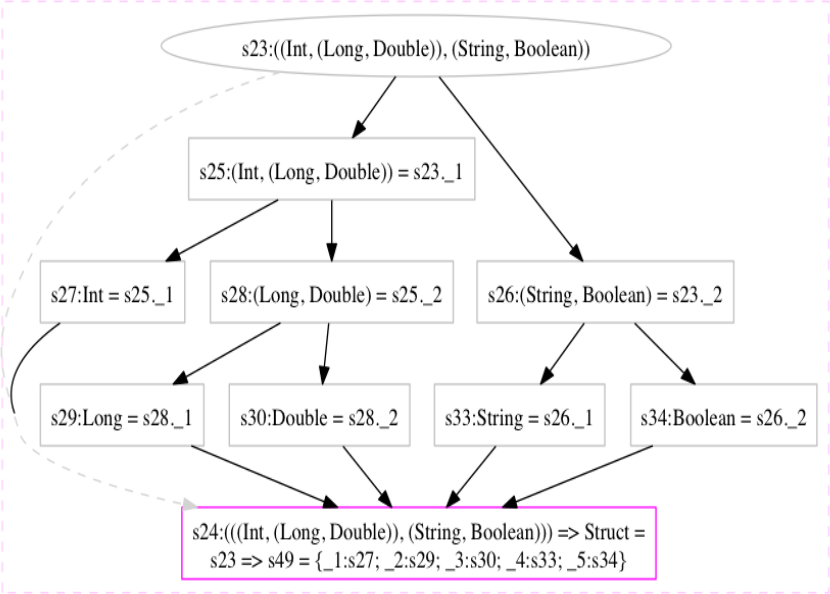
$\mathbf{P_{SRC}}$(data) = $\mathbf{P_{VIRT}}$(data) = $\mathbf{P_{IR}}$(data)

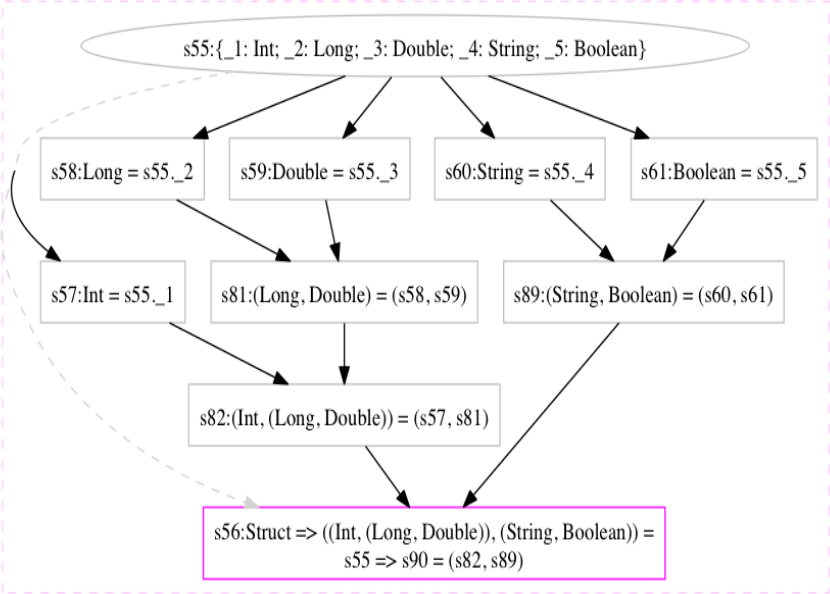
DEMONSTRATION
(1)

Basic Machinery: Generic meta-operators

- Isomorphisms







Basic Machinery: Generic meta-operators

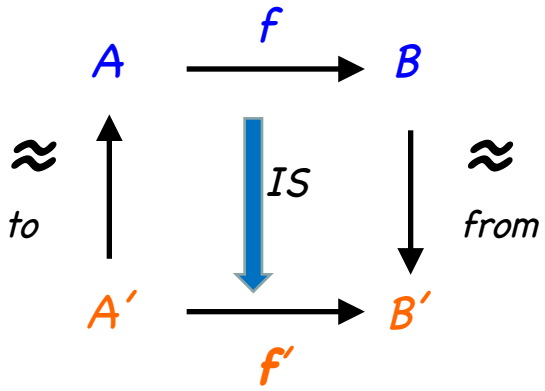
- **Converters**

```
class SparseMatrix
```

```
class DenseMatrix
```

Basic Machinery: Generic meta-operators

- **Isomorphic Specialization**

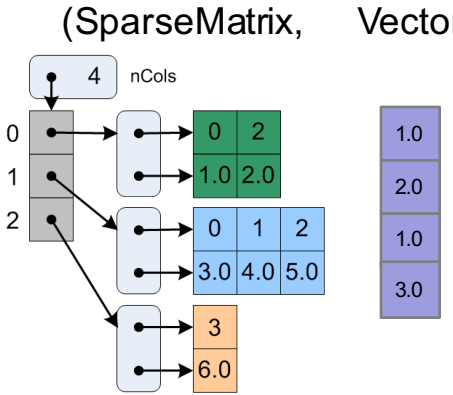


$f' = SE[to >> f >> from]$

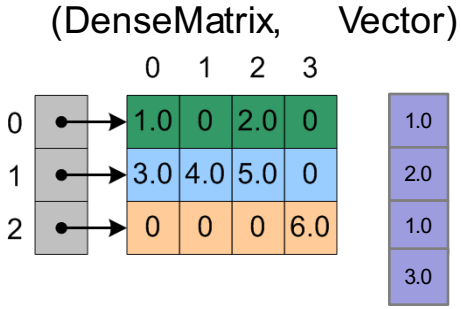
DEMONSTRATION
(2)

Converters for free

```
val Some(converter) = getConverter(
  element[(SparseMatrix[Double],Vector[Double])],
  element[(DenseMatrix[Double],Vector[Double])]
)
```



converter



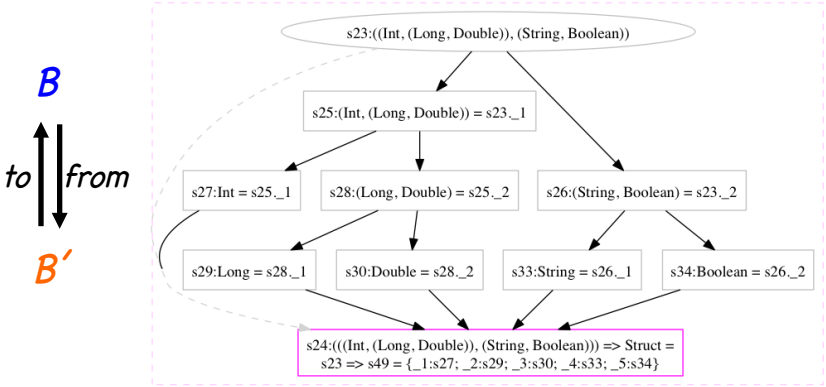
converter >>

```
lazy val mvm = fun {
  p: Rep[(Matrix[Double], Vector[Double])] =>
    val Pair(m, v) = p
    DenseVector(m.rows.map { r => r dot v })
}
```

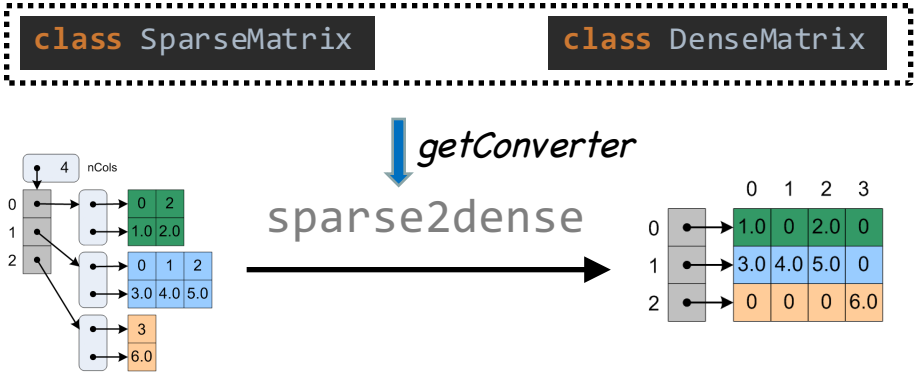
DEMONSTRATION
(3)

Generic meta-operators

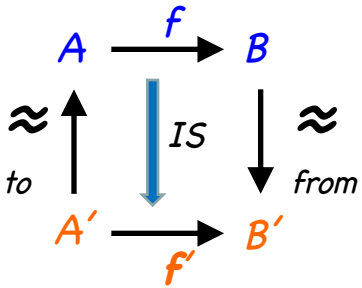
- **Isomorphisms**



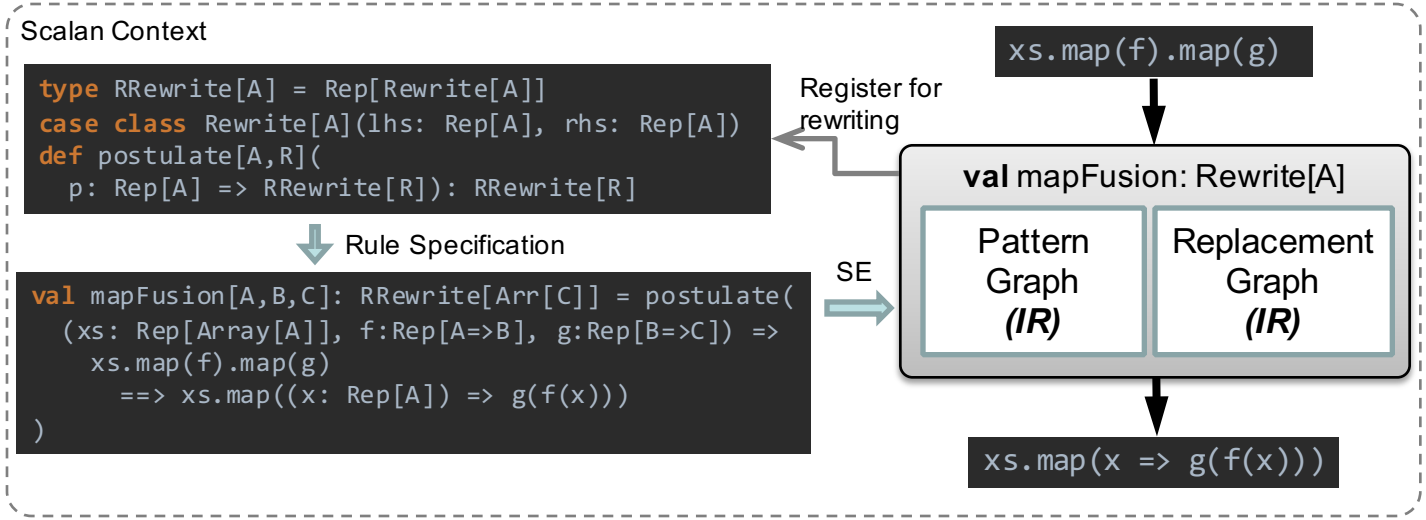
- **Converters**



- **Isomorphic Specialization**

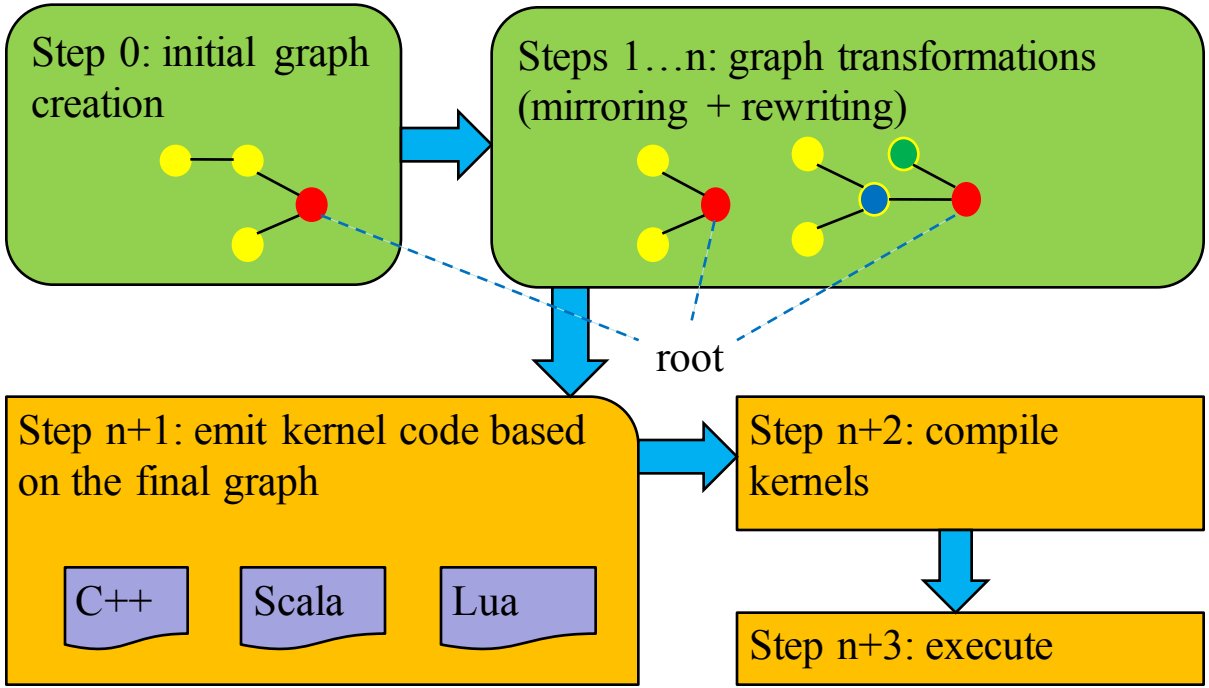


First-class Rewriting Rules



- Scalan compiler works as a provider of the pattern and replacement graphs (Rewrite rule components)
- It performs a staged evaluation of the set of rules and creates the IRs
- Rewrite rule component keeps for each pattern graph a corresponding replacement graph

Multi-stage Compilation Pipeline



Get involved

- Read the post about Scalan's Meta-programming Idioms (https://github.com/scalan/scalan.github.io/blob/master/idioms.md)
- Check out the project at https://github.com/scalan/scalan
- Ask questions on Google Group https://groups.google.com/d/forum/scalan
- Follow us on twitter (@avslesarenko, @alexey_r)
- Code for the demo: https://github.com/scalan/scalan-starter
- Program Functionally, Execute Imperatively: Peeling abstraction overhead from functional programs. Scala Days Amsterdam, June 2015(slides/video)
- Scalan publications: http://pat.keldysh.ru/~slesarenko/

Thank you

HUAWEI

SCALA DAYS BERLIN 2016

Please

Remember to rate this session

Thank you!

Join the conversation #scaladays


SCALA DAYS
BERLIN 2016