Optimizing Data Structures in High-Level Programs:

New Directions for Extensible Compilers based on Staging

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown,
Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda,
Kunle Olukotun, Martin Odersky

How should we build compilers?

Principles of Compiler Design

Alfred V. Aho
Jeffrey D. Ullman

Productivity: Generalization, Abstraction

Programs and Languages

Hardware

Performance: Specialization, Concretization

A Linear Algebra Library

```
abstract class Vector[T:Numeric] {
  val data: Array[T]
  def +(that: Vector[T]) =
    Vector.fromArray(data.zipWith(that.data)(_ + _))
}
object Vector {
  def fromArray[T:Numeric](a: Array[T]) =
    new Vector { val data = a }
  def zeros[T:Numeric](n: Int) =
    Vector.fromArray(Array.fill(n)(i => zero[T]))
}
abstract class Matrix[T:Numeric] { … }
case class Complex(re: Double, im: Double) {
  def +(that: Complex) = Complex(re + that.re, im + that.im)
  def *(that: Complex) = …
}
implicit object ComplexIsNumeric extends Numeric[Complex] { … }
```

User Program

```
def diag(k: Int, n: Int) =
  k * Matrix.identity(n)

val m1 = (v1+v2).trans * (v1+v2)
val m2 = diag(2, m1.numRows)

if (scale) println(m1*m2)
else println(m1)
```

Elegant and high level, but is it fast?

The compiler / VM will figure out how to run it fast

(wishful thinking)

No it doesn’t: 10 to 100x slower than optimized code with arrays and loops!

(hard reality)

Many productivity features don’t perform well

- **Problem 1: abstraction penalty**

- Problem 2: compiler lacks semantic knowledge

Abstractio

```
abstract class Vector[T:Numeric] {
  val data: Array[T]
  def +(that: Vector[T]) =
    Vector.fro  rray(data.zipWith(that.data)(_ + _))
}
object Vector
  def fromArray[                 Array[T]) =
    new Vector { val data = a }
  def zeros[T:Numeric](n: Int) =
    Vector.fromArray(Array.fill(n)(i => zero[T]))
}
abstract class Matrix[T:Numeric] { … }
case class Complex(re: Double, im: Double) {
  def +(that: Complex) = Complex(re + that.re, im + that.im)
  def *(that: Complex) = …
}
implicit object ComplexIsNu            Numeric[Complex] { … }
```

Type Classes

Indirection

Closures (and megamorphic call sites)

Object allocations

Idea: Let’s use Macros or Staging!

compose program fragments

programmatically remove abstraction

Lightweight Modular Staging (LMS)

- Use a type constructor Rep[T] to delay evaluation of expressions to the next (generated) stage
- Lift operations from type T to type Rep[T], generating code to apply the operation later
- Expressions of type T are evaluated immediately and become constants in generated code
- Maintain evaluation order within a stage (unlike syntactic quasi-quotation)

Tiark Rompf, Martin Odersky. Commun. ACM 55(6): 121-130 (2012)

Example: Vectors

```
abstract class Vector[T:Numeric] {
  val data: Array[T]
  def +(that: Vector[T]) =
    Vector.fromArray(data.zipWith(that.data)(_ + _))
}

object Vector {
  def fromArray[T:Numeric](a: Array[T]) =
    new Vector { val data = a }

  def zeros[T:Numeric](n: Int) =
    Vector.fromArray(Array.fill(n)(i => zero[T]))
}
```

Example: Vector

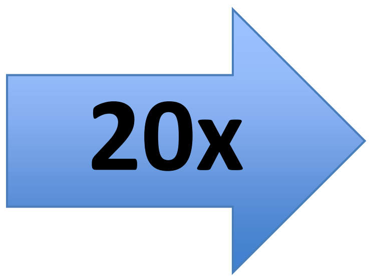
```
abstract class Vector[T:Numeric] {
  val data: Rep[Array[T]]
  def +(that: Vector[T]) =
    Vector.fromArray(data.zipWith(that.data)(_ + _))
}

object Vector {
  def fromArray[T:Numeric](a: Rep[Array[T]]) =
    new Vector { val data = a }

  def zeros[T:Numeric](n: Rep[Int]) =
    Vector.fromArray(Array.fill(n)(i => zero[T]))
}
```

Example: Array

```
implicit class ArrayOps[T](a: Rep[Array[T]]) {

  def zipWith[U,V](b: Rep[Array[U]])(f: (Rep[T],Rep[U]) => Rep[V]) =
   Array.fill(min(a.length,b.length))(i => f(a(i), b(i))

}

object Array {
  def fill[T](n: Size)(f: Rep[Int] => Rep[T]) = {
    val r = NewArray[T](n)
    var i: Rep[Int] = 0     // staged variable
    while (i < n) {         // staged loop
      r(i) = f(i)
      i += 1
    }
    r
  }
}
```

Example: Matrix

```
val m = Matrix.rand(500, 100)
val n = Matrix.rand(100, 500)
m * n
```




```
var x27 = 500 * 500
var x28 = new Array[Double](x27)
var x29: Int = 0
while (x29 < 500) {
  var x30: Int = 0
  while (x30 < 500) {
    var x31: Int = 0
    while (x31 < 100) {
      ...
      x31 += 1
    }
    var x46 = ()
    x46
    x30 += 1
  }
  var x47 = ()
  x47
  x29 += 1
}
```

Victory?

- Problem 1: abstraction penalty
  - Staging
- **Problem 2: compiler lacks semantic knowledge**

Compiler Lacks Semantic Knowledge

```
def diag(k: Int, n: Int) = k * Matrix.identity(n)

val m1 = (v1+v2).trans * (v1+v2)
val m2 = diag(2, m1.numRows)

if (scale) println(m1*m2)      // m1*(k*id) = k*m1*id = k*m1
else println(m1)               // no need to compute m2
```

Limitations of Staging / Macros

- Want to treat matrices as symbolic entities with algebraic laws
- `m*ident` expanded into arrays / loops before reaching the compiler
  - Too late to perform symbolic simplification!

Extend compiler with high-level semantic knowledge

Extensible Compilers

- Vector/Matrix operations as IR nodes
- Optimization pass to simplify m*ident => m
- Another pass to expand operations into loops
- Usual limitations:
  - heavyweight
  - IR-to-IR transformers much lower level, harder to express than with macros / staging
  - Phase ordering problems between new and existing optimizations

- Problem 1: abstraction penalty
  - Staging
- Problem 2: compiler lacks semantic knowledge
  - Extensible compilers
- Neither solution alone is sufficient!

Use staging in intermediate languages!

Stage away abstractions *after* applying symbolic rewrites, CSE, etc!

A staged interpreter is a program transformer

Instead of Tree => Tree:

Tree => staged code that computes a Tree

Not all Transformations are Alike

- Lowerings
  - e.g., vector/matrix ops --> loops over arrays
  - Have a natural ordering
  - Can be profitably arranged in separate passes
  - Easy to solve with staged interpreters
- Optimizations
  - No clear ordering, prone to phase ordering problems
  - Must be combined for maximum effectiveness (optimistic assumptions)
  - Should be applied exhaustively before lowering takes place
- Should optimize, lower, optimize, lower, …
  until lowest-level representation is reached

How to combine optimizations?

Rewriting using smart constructors for IR nodes:
The only problem is loops

Speculative Rewriting

- Apply all possible transformations optimistically
  - Ignore loop-carried dependencies, etc.
- If an assumption is violated, throw away transformed result and start again
- Repeat until fixed point is reached

```
var x = 7
var c = 0

while (c < 10) {
  if (x < 10) print("!")
  else x = c
  print(x)
  print(c)
  c += 1
}
```

```
var x = 7
var c = 0

while (true) {
  print("!")
  print(7)
  print(0)
  c = 1
}
```

```
var x = 7 //dead
var c = 0

while (c < 10) {
  print("!")
  print(7)
  print(c)
  c += 1
}
```

See: Lerner, Grove, Chambers (POPL'2002); Supercompilation: Turchin, Klimov, ….

Example: Matrix

```
trait MatrixExp extends BaseExp {
  trait Matrix[T]

  case class MatrixTimes[T:Numeric](a: Rep[Matrix[T]], b: Rep[Matrix[T]])
    extends Def[Matrix[T]]
  case class MatrixPlus[T:Numeric](a: Rep[Matrix[T]], b: Rep[Matrix[T]])
    extends Def[Matrix[T]]

  def infix_*[T:Numeric](a: Rep[Matrix[T]], b: Rep[Matrix[T]]) =
    reflect(MatrixTimes(a,b))
  def infix_+[T:Numeric](a: Rep[Matrix[T]], b: Rep[Matrix[T]]) =
    reflect(MatrixPlus(a,b))
}


trait MatrixExpOpt extends MatrixExp {
  override def infix_+[T:Numeric](a: Rep[Matrix[T]], b: Rep[Matrix[T]]) =
    (a,b) match {
      case (Def(MatrixTimes(a1,b)), Def(MatrixTimes(a2,c))) if a1 == a2 =>
        a1 * (b + c)   // A*B+A*C => A*(B+C)
      case _ => super.infix_+(a,b)
    }
}
```

Example: Matrix

```
trait MatrixExpLower extends MatrixExp {

  def matrixTimesImpl[T](a: Rep[Matrix[T]], b: Rep[Matrix[T]]) = {
    val res = MatrixNew(a.rows, b.cols)
    for (i <- 0 until a.rows) {
      for (j <- 0 until b.cols) {
        for (k <- 0 until a.rows)
          res(i, j) += a(i, k) * b(k, j)
      }
    }
    res
  }

  override def onCreate[T](sym: Rep[T], rhs: Def[T]) = rhs match {
    case MatrixTimes(a,b) => atPhase(lowering) { matrixTimesImpl(a,b) }
    case _ => super.onCreate(sym,rhs)
  }
}
```

What we have achieved:

- CSE, DCE on matrix operations done by LMS-Core compiler
- Added custom rewrite: A*B+A*C => A*(B+C)
  - Rewrites compose!
- Added custom lowering: MatrixTimes => loops
  - Implemented as a staged method

- Uniform low-level loop abstraction
  - fusion and data parallelism

Loop Fusion

```
def square(x: Rep[Double]) = x*x

def mean(xs: Rep[Vector[Double]]) =
      xs.sum / xs.length

def variance(xs: Rep[Vector[Double]]) =
      xs.map(square) / xs.length - square(mean(xs))


val v1 = Vector.fill(n) { i => 1 }
val v2 = Vector.fill(n) { i => 2*i }
val v3 = Vector.fill(n) { i => v1(i) + v2(i) }

val m = mean(array3)
val v = variance(array3)

println(m)
println(v)
```

```
// begin reduce x47,x51,x11
  var x47 = 0
  var x51 = 0
  var x11 = 0
  while (x11 < x0) {
     val x44 = 2.0*x11
     val x45 = 1.0+x44
     val x50 = x45*x45
     x47 += x45
     x51 += x50
     x11 += 1
  }
// end reduce
val x48 = x47/x0
val x49 = println(x48)
val x52 = x51/x0
val x53 = x48*x48
val x54 = x52-x53
val x55 = println(x54)
```

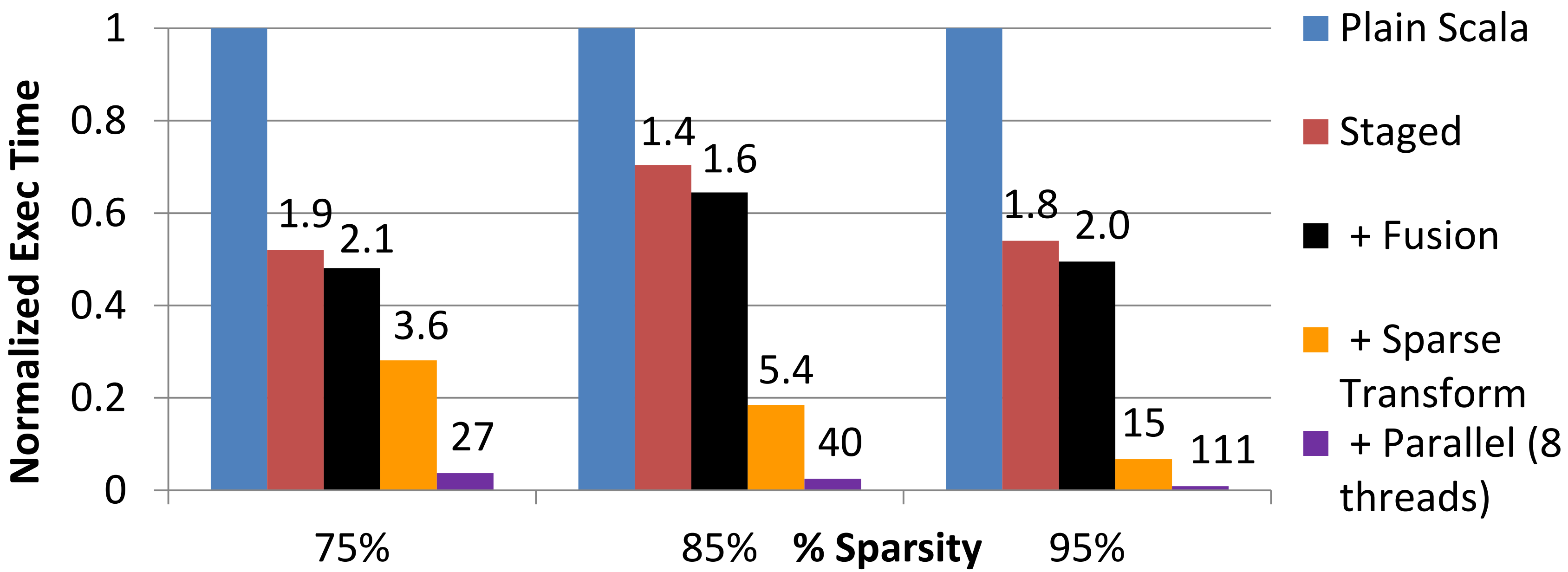
3+1+(1+1) = 6 traversals, 4 arrays

1 traversal, 0 arrays

Evaluation

Linear Algebra / Machine Learning

```
def preferences(ratings: Rep[Matrix[Int]], sims: Rep[Matrix[Double]]) = {
  sims.mapRowsToVector { testProfile =>
    val num = sum(0, ratings.numRows) {
      i => testProfile(ratings(i,1))*ratings(i,2) }
    val den = sum(0, ratings.numRows) {
      i => abs(testProfile(ratings(i,1))) }
    num/(den+1)
}}
```



Regular Expressions

```
def convertNFAtoDFA(flag: Boolean, state: NIO): DIO = {
  val cstate = canonicalize(state)
  dfa_trans(flag) { c: Rep[Char] => exploreNFA(cstate, c) {
    convertNFAtoDFA
  }
}}
convertNFAtoDFA(false, findAAB())
```

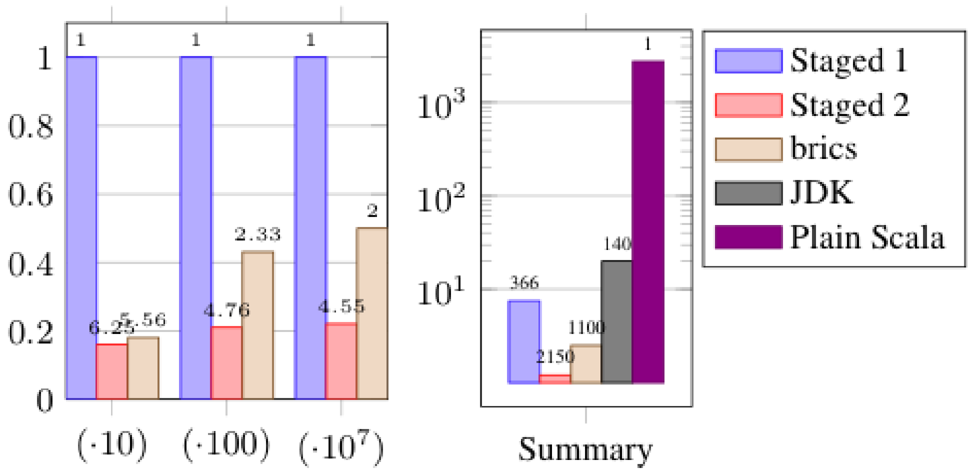
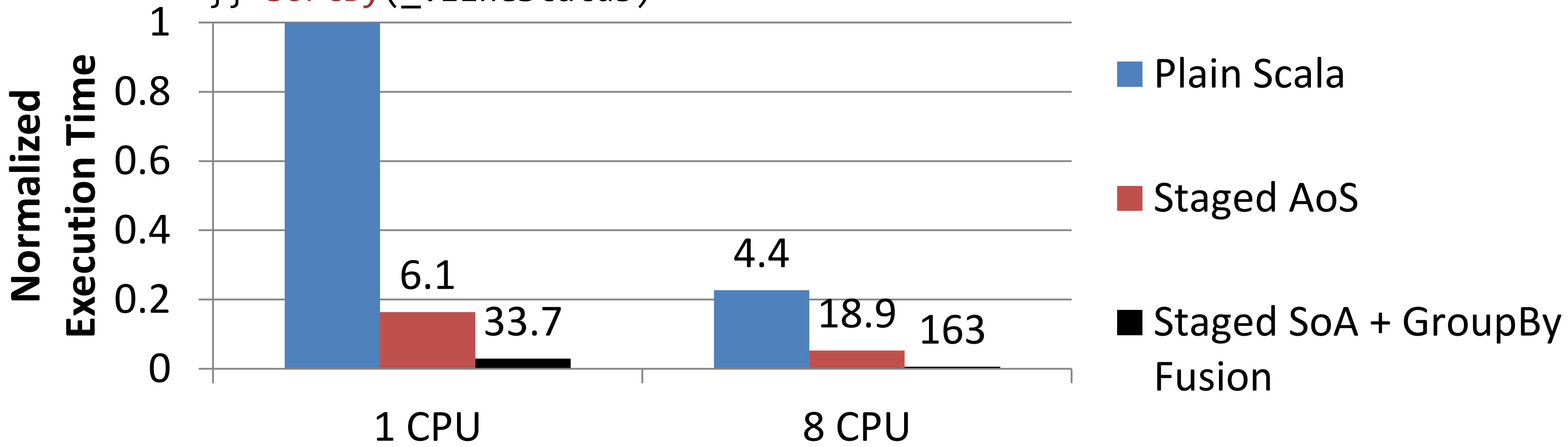


Figure 13. Regexp Benchmark. The first graph shows the relative execution time of matching a respectively 10, 100, 10^7 long input string of the form A+B on the regular expression .*AAB. The second graph summarizes the relative performance over many different inputs and regular expressions.

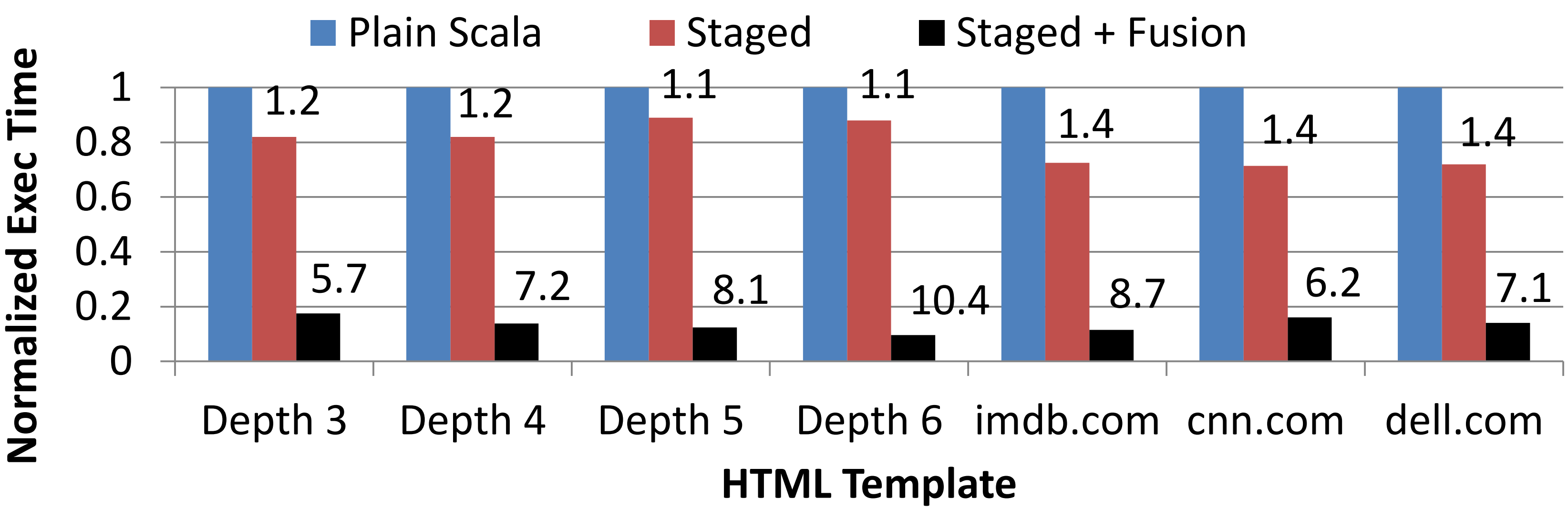
Collections and Queries

```
// lineItems: Array[LineItem]
val q = lineItems filter (_.l_shipdate <=
Date(‘‘19981201’’)).
  groupBy (_.l_linestatus) map { case (key,g) => new Record {
    val lineStatus = g.key
    val sumQty = g.map(_.l_quantity).sum
    val sumDiscountedPrice =
      g.map(r => r.l_extendedprice*(1.0-r.l_discount)).sum
    val avgPrice = g.map(_.l_extendedprice).sum / g.size
    val countOrder = g.size
  }} sortBy(_.lineStatus)
```



String Templating

```
def link(uri: Rep[String], name: Rep[String]) =
  List(“<a href=‘”, uri, “’>”, name, “</a”)

def renderItem(i: Rep[Item]) = List(“<li><ul>”) ++
  i.subitems.flatMap(link(i.name, i.link)) ++ List(“</ul></li>”)

def usersTable(items: Rep[List[Item]]) = List(“<ul>”) ++
  items.flatMap(renderItem) ++ List(“</ul>”)
```



Evaluation Summary

Order of magnitude speedups on a variety of high-level programs by:

- fusing collection operations
- changing data layout
- applying (generic and specific) optimizations on high-level objects

Key Take-Aways:

1. Compilers need to make sense of high-level, domain-specific abstractions
2. Many different techniques (staging, extensibility, speculative rewriting, fusion): We really need to combine all of them to achieve good results!

scala-lms.github.com

Backup Slides

Key Take-Aways:

- Optimizations should be combined
  - Avoid pessimistic assumptions
  - Avoid phase ordering problems
  - Speculative rewriting: generic solution for forward DF
- Lowering transforms should be separate passes
  - Apply high-level optimizations exhaustively before switching representations (e.g. Matrix/Vector to arrays and loops)
- Staged IR interpreters as IR to IR transformers:
  - Programmatically remove abstraction overhead at all intermediate stages
  - Simplify implementation

Expression Templates

- Purely frontend approach
- Not integrated with DCE, CSE
- Optimization horizon restricted to extent of compound expression

Rewriting Frameworks

- Graphs vs trees
- Dependency information
- “model transformation all the way down” similar to our approach to lowering
- But we want to combine optimizations, not layer them
- Interpretation is simpler than transformation!

Fusion

- Horizontal and vertical
- Includes flatMap and groupBy
- Not restricted to scope of single expression; only one resulting loop here:

```
def calcSum() = array.sum
def calcCount() = array.filter(_ > 0).count
println("sum: " + calcSum())
println("avg: " + (calcSum() / calcCount())")
```

Lisp/Scheme

- Also pervasive use of macros in compilation
- Which implementation:
  - Supports an open set of algebraic rewrites for vector/matrix operations without phase ordering problems?
  - Reuses generic CSE, DCE etc on vectors and matrices?
  - Can apply AOS to SOA transforms?

Partial Evaluation of Interpreters

- Earlier work on program transformation by partial evaluation
- Different techniques
- Arbitrary compiler optimisations, not just constant folding
- Arbitrary computation at staging/specialization time to remove abstraction overhead
- Strong guarantees about shape of residual code (Rep[T] vs T types)

EOF