Lightweight Modular Staging

Tiark Rompf, Martin Odersky
EPFL

Arvind Sujeeth, Hassan Chafi, Kevin Brown,
HyoukJoong Lee, Kunle Olukotun
Stanford University

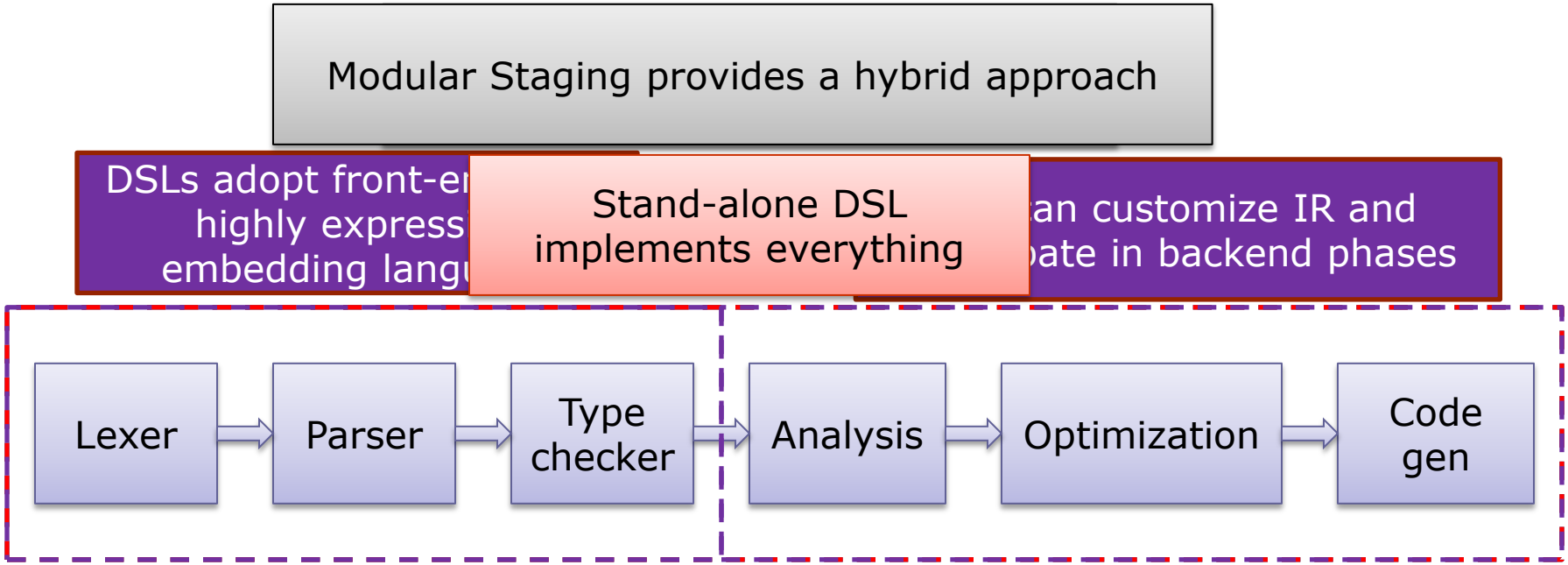
Outline

- Later lectures will discuss general DSL implementation strategies
- This lecture is about how we do things in Scala and Delite
  - Help you get a quick start on your projects
  - Explore the code base

Goal: embedded parallel DSL

- We want to be able to:
  - Build an intermediate representation (IR) of user programs
  - Analyze and optimize the IR
  - Generate parallel code
    - Scala, C/C++, CUDA, …
- …all without working too hard

Modular Staging Approach



Typical Compiler

GPCE'10: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs

How do you build an IR at runtime?

- Metaprogramming
  - C++ expression templates
  - C# expression trees
  - Haskell templates
  - MetaOCaml staging constructs
  - …
- Our approach is LMS
  - **Lightweight**: uses just Scala's type system
  - **Modular**: pick and choose how to represent nodes, what optimizations to apply, and which generators to use at runtime
  - **Staging:** a program that writes other (optimized) programs

Strategy

- Programs usually operate on concrete types (Int, Matrix, List, etc.)
- Instead, we'll use an **abstract placeholder** to represent types
  - Rep[T]
- Why?
  - What happens when you try to operate on a Rep[T]?

Looking closer at Rep[T]

- Rep[T] is an **abstract type constructor**
- We can define any concrete type constructor we want

```
trait StringRep extends Base {
    type Rep[T] = String
}
```

- But strings aren't that useful. What if we had a type that represented an Expression?

```
trait ExpRep extends Base {
    type Rep[T] = Exp[T]
}
```

Defining an IR

```
trait Expressions {
     // constants/symbols (atomic)
    abstract class Exp[T]
    case class Const[T](x: T) extends Exp[T]
    case class Sym[T](n: Int) extends Exp[T]

    // operations (composite, defined in subtraits)
    abstract class Def[T]

    // additional members for managing encountered definitions
    def findOrCreateDefinition[T](rhs: Def[T]): Sym[T]

    implicit def toExp[T](d: Def[T]): Exp[T] = findOrCreateDefinition(d)
  }

trait Base {
  type Rep[T] // abstract
}

trait BaseExp extends Base {
  type Rep[T] = Exp[T]
}
```

Using Rep[T]

- val x: Rep[Int]
- val y = x + 5

- "+" is not defined on Rep[Int]!
- But we can define it to be anything we want

Extending Rep[T]

```
trait IntOps extends Base {
  def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int]
}
```

- Now if x and y are both Rep[Int], x + y will be translated by the compiler to:
  - infix_+(x,y)
- But we still haven't defined the implementation for infix_+

Extending Rep[T]

```
trait IntOps extends Base {
   def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int]
}

trait IntOpsExp extends BaseExp {
  case class IntPlus(x: Exp[Int], y: Exp[Int])
    extends Def[Int]

  def infix_+(x: Exp[Int], y: Exp[Int]) = IntPlus(x,y)
}
```

- We just built an IR node!

But how do we get these Reps?

- If we start from an existing type (Int, List, etc.)
  - We can **lift** those types into the Rep world using an implicit conversion

```
// since we are starting with an already
// constructed instance, it is a constant at
// the time it is injected into the IR
implicit def unit(x: T) = Const(x)
```

But how do we get these Reps?

- If we start from a type that we made up
  - We can provide a factory method to return a Rep

```
trait MatrixOps extends Base {
    object Matrix {
      def apply[T](numRows: Rep[Int], numCols: Rep[Int])
          = matrix_new(numRows, numCols)
    }

    def matrix_new[T](m: Rep[Int], n: Rep[Int]): Rep[Matrix[T]]
}

trait MatrixOpsExp extends BaseExp {
  case class MatrixNew[T](m: Exp[Int], n: Exp[Int])
    extends Def[Matrix[T]]

  def matrix_new[T](m: Exp[Int], n: Exp[Int]) = MatrixNew(x,y)
}
```

So where are we now?

- We defined Rep[T], and one useful kind of Rep, Exp[T], representing an IR node
- We showed how to construct instances of Rep[T]
- We showed how to override methods on Reps to do anything we want
  - And we used this to construct an IR node representing the operation
- Everything is well-typed!

Let's look at an application

```
object MyApplication extends MatrixOps {
    def main(args: Array[String]) {
        val x = Matrix[Int](10,20) // Rep[Matrix[Int]]
        println(x)
    }
}
```

And run it…

```
error: polymorphic expression cannot be instantiated to expected type;
found   : [T(in method apply)]Example1OpsExp.this.MatrixNew[T(in method
    apply)]
required: Example1OpsExp.this.Rep[Matrix[T(in method matrix_new)]]
def matrix_new[T:Manifest](x: Exp[Int], y: Exp[Int]) = MatrixNew(x,y)
```

What the hell?

Debugging is painful – we're working on it

Fixing compile errors

```
trait MatrixOpsExp extends BaseExp {
  case class MatrixNew[T](m: Exp[Int], n: Exp[Int])
    extends Def[Matrix[T]]

  def matrix_new[T](m: Exp[Int], n: Exp[Int]) =
   MatrixNew[T](m,n)
}
```

Alternatively,

```
 def matrix_new[T](m: Exp[Int], n: Exp[Int]):
   Exp[Matrix[T]] = MatrixNew(m,n)
```

New error

```
error: type mismatch;
found   : Int(10)
required: Example1.Rep[Int]
val a = Matrix[Int](10,20)
```

- What happened?
  - We forgot to include the implicit that lifts Ints to Rep[Int]!
- Let's try again

Try #2

```
object MyApplication extends MatrixOps with LiftNumeric {
    def main(args: Array[String]) {
        val x = Matrix[Int](10,20) // Rep[Matrix[Int]]
        println(x)
    }
}
```

- Almost there…

error: object creation impossible, since:
method matrix_new in trait Example1Ops of type [T](x: Example1.Rep[Int],y: Example1.Rep[Int])(implicit evidence$2: Manifest[T])Example1.Rep[Matrix[T]] is not defined
method unit in trait Base of type [T](x: T)(implicit evidence$2: Manifest[T])Example1.Rep[T] is not defined
object Example1 extends Example1Ops with LiftNumeric {

MatrixOps is abstract

- It doesn't define Rep or matrix_new
- We need to use MatrixOpsExp

```
object MyApplicationRunner extends MyApplication with
    MatrixOpsExp {
    def main(args: Array[String]) { run() }
}

trait MyApplication extends MatrixOps with LiftNumeric {
    def run() {
      val x = Matrix[Int](10,20) // Rep[Matrix[Int]]
      println(x)
    }
}
```

Success!

- Our tiny embedded program compiles
- What happens when we run it?

```
Sym(0)
Process finished with exit code 0
```

- Exciting…

A slightly more complicated example

```
trait MyApplication extends MatrixOps with LiftNumeric {
  def run() {
    val x0 = Matrix[Int](10,10) // Rep[Matrix[Int]]
    val b = x0*x0*x0*x0*1
    println(b)
  }
}
```

- We need to add Matrix*Matrix and Matrix*Int nodes, or this will fail with a compile error:

```
error: value * is not a member of
  Example2.this.Rep[Example2.this.Matrix[Int]]
  val b = x0*x0*x0*x0*1
```

Adding to MatrixOps

```
trait MatrixOps extends Base {
    object Matrix {
      def apply[T](numRows: Rep[Int], numCols: Rep[Int])
          = matrix_new(numRows, numCols)
    }
    def infix_*[T](x: Rep[Matrix[T]], y: Rep[Matrix[T]]) = matrix_times(x,y)
    def infix_*[T](x: Rep[Matrix[T]], y: Rep[Int]) = matrix_times_scalar(x,y)

    def matrix_new[T](m: Rep[Int], n: Rep[Int]): Rep[Matrix[T]]
    def matrix_times[T](x: Rep[Matrix[T]], y: Rep[Matrix[T]]): Rep[Matrix[T]]
    def matrix_times_scalar[T](x: Rep[Matrix[T]], y: Rep[Int]): Rep[Matrix[T]]
}

trait MatrixOpsExp extends BaseExp {
  case class MatrixNew[T](m: Exp[Int], n: Exp[Int]) extends Def[Matrix[T]]
  case class MatrixTimes[T](x: Exp[Matrix[T]], y: Exp[Matrix[T]]) extends Def[Matrix[T]]
  case class MatrixTimesScalar[T](x: Exp[Matrix[T]], y: Exp[Int]) extends Def[Matrix[T]]

  def matrix_new[T](m: Exp[Int], n: Exp[Int]) = MatrixNew[T](x,y)
  def matrix_times[T](x: Exp[Matrix[T]], y: Exp[Matrix[T]]) = MatrixTimes(x,y)
  def matrix_times_scalar[T](x: Exp[Matrix[T]], y: Exp[Int]) = MatrixTimesScalar(x,y)
}
```

Good to go

- Hit “compile”…

```
error: double definition:
method infix_*:[T](x: Example2Ops.this.Rep[Example2Ops.this.Matrix[T]],y:
    Example2Ops.this.Rep[Int])(implicit evidence$3:
    Manifest[T])Example2Ops.this.Rep[Example2Ops.this.Matrix[T]] and
method infix_*:[T](x: Example2Ops.this.Rep[Example2Ops.this.Matrix[T]],y:
    Example2Ops.this.Rep[Example2Ops.this.Matrix[T]])(implicit evidence$2:
    Manifest[T])Example2Ops.this.Rep[Example2Ops.this.Matrix[T]] at line 10
have same type after erasure: (x: java.lang.Object,y: java.lang.Object,implicit
    evidence$3: scala.reflect.Manifest)java.lang.Object
def infix_*[T:Manifest](x: Rep[Matrix[T]], y: Rep[Int]) = matrix_times_scalar(x,y)
```

- Generics strikes again
  - Types are erased, so method signatures are identical

Fighting type erasure

```
trait MatrixOps extends Base with OverloadHack {
    object Matrix {
      def apply[T](numRows: Rep[Int], numCols: Rep[Int])
          = matrix_new(numRows, numCols)
    }
    def infix_*[T](x: Rep[Matrix[T]], y: Rep[Matrix[T]])(implicit o:
    Overloaded1) = matrix_times(x,y)
    def infix_*[T](x: Rep[Matrix[T]], y: Rep[Int])(implicit o:
    Overloaded2) = matrix_times_scalar(x,y)
}
```

- Seriously? "OverloadHack"?
  - At least only the DSL authors (you guys) see it, and not the users…

Almost no magic

```
trait OverloadHack {
  class Overloaded1
  class Overloaded2
  class Overloaded3
  class Overloaded4
  etc...

  implicit val overloaded1 = new Overloaded1
  implicit val overloaded2 = new Overloaded2
  implicit val overloaded3 = new Overloaded3
  implicit val overloaded4 = new Overloaded4
  etc…
}
```

- Force the compiler to distinguish the method types by attaching a different implicit parameter to each signature

Run again

```
Sym(4)
Process finished with exit code 0
```

- Still not that enlightening
- We can override println to peek under the covers (see examples posted online), and we get:

```
MatrixTimesScalar(MatrixTimes(MatrixTimes(MatrixTimes(Mat
    rixNew(Const(10),Const(10)),MatrixNew(Const(10),Const(
    10))),MatrixNew(Const(10),Const(10))),MatrixNew(Const(
    10),Const(10))),Const(1))
Process finished with exit code 0
```

- This is a textual representation of our program
  - -a graph in text form

Another way of looking at it

```
Sym(4) = MatrixTimesScalar(Sym(3),Const(1))
Sym(3) = MatrixTimes(Sym(2),Sym(0))
Sym(2) = MatrixTimes(Sym(1),Sym(0))
Sym(1) = MatrixTimes(Sym(0),Sym(0))
Sym(0) = MatrixNew(Const(10),Const(10))
Sym(0) = MatrixNew(Const(10),Const(10))
Sym(0) = MatrixNew(Const(10),Const(10))
Sym(0) = MatrixNew(Const(10),Const(10))

Process finished with exit code 0
```

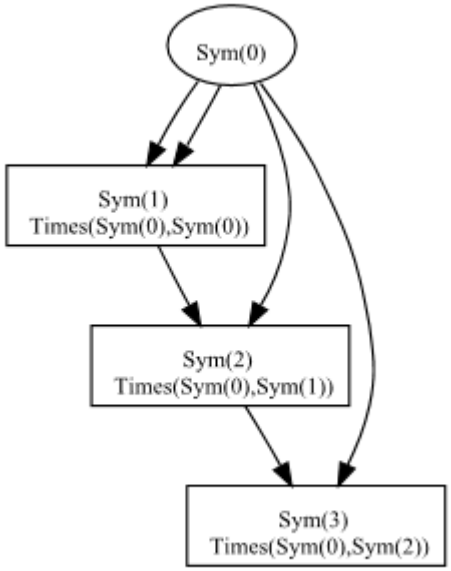
Another way of looking at it

- Here is another (nicer) view of the same thing

$$x_0 * x_0 * x_0 * x_0 * 1$$

$$\Downarrow$$

$$x_1 = x_0 * x_0$$
$$x_2 = x_0 * x_1$$
$$x_3 = x_0 * x_2$$



Details (1)

- How do you override operations in the language that aren't method calls?
  - var x = 5
  - x == y
  - while (true) { foo() }
  - if (foo) bar() else foobar()
- Turn them into methods!
- This is exactly what the scala-virtualized compiler does
  - It also adds those nice infix_ methods we've been using to handle operations on Reps
- Now the dsl author can override __equals(x: Rep[Any], y: Rep[Any]), etc.

Details (2)

- I’ve left off Manifest implicit parameters in all the previous examples for brevity
- Manifests are objects that carry around run-time type information
- They are instantiated automatically by compiler, and provided everywhere they are required as an implicit parameter
- We use them to keep the type information for all the symbols we create, despite erasure
- Very useful – you’ll see them everywhere

So we have an IR. Now what?

- How do I optimize the IR?
- How do I generate code?
- How do we handle control dependencies and side effects?

Optimizations

- Common subexpression elimination (CSE)
- Dead code elimination (DCE)
- Domain-specific pattern rewrites
- Loop hoisting & fusing (pretty involved – we won't talk about this now, but come ask questions if you're interested)

CSE

- Pretty simple
- Let's take a closer look at our IR trait (Expressions.scala)

```
protected implicit def toAtom[T:Manifest](d: Def[T]): Exp[T] = {
  findOrCreateDefinition(d).sym
}

 def findOrCreateDefinition[T:Manifest](d: Def[T]): TP[T] =
   findDefinition[T](d).getOrElse {
     createDefinition(fresh[T], d)
   }
```

CSE

```
trait MatrixOpsExp extends BaseExp {
  case class MatrixNew[T](m: Exp[Int], n: Exp[Int])
    extends Def[Matrix[T]]

  def matrix_new[T](m: Exp[Int], n: Exp[Int]) =
   MatrixNew[T](x,y)
}
```

- The return type of **matrix_new** is Exp[Matrix[T]]
- but MatrixNew[T](x,y) returns a Def[Matrix[T]]
- The compiler inserts the implicit **toAtom** conversion to create a symbol for the Def[T] and return a Sym[T]
  - If the symbol already exists, it is reused

DCE

- We essentially get it "for free" with our IR
- Notice that we don't have a traditional AST or CFG like representation
  - These representations are more closely tied to the original program
  - Can be good and bad
  - The good is that we are not over-constrained: we only care about true dependencies

DCE

- Our IR is similar to a **program-dependence graph** (PDG)
- We figure out a node's dependencies by following links in the IR
- If there is code that the result of a block didn't depend on, it is never found

DCE example

```
def run() = {
  val a = Matrix(10,10)
  val b = a*10
  val c = a*20
  println(c)
}
```

- B is dead code: it is never returned and never printed
- When we schedule this block, we will follow C's dependencies and find A, but not B

Domain-specific pattern rewriting

- A simple but powerful form of optimization
- Consider adding a MatrixPlus IR node

```
trait MatrixOpsExp extends BaseExp {
  case class MatrixPlus[T](x: Exp[Matrix[T]], y: Exp[Matrix[T]])
    extends Def[Matrix[T]]

  def matrix_plus[T](x: Exp[Matrix[T]], y: Exp[Matrix[T]]) =
   MatrixPlus(x,y)
}
```

- Normally we just construct the node with its arguments
- But we can use pattern matching to find special cases

Domain-specific pattern rewriting

```
trait MatrixOpsExp extends BaseExp {
  case class MatrixPlus[T](x: Exp[Matrix[T]], y: Exp[Matrix[T]])
    extends Def[Matrix[T]]

  def matrix_plus[T](x: Exp[Matrix[T]], y: Exp[Matrix[T]]) =
    MatrixPlus(x,y)
}

trait MatrixOpsExpOpt extends MatrixOpsExp {
  override def matrix_plus[T](x: Exp[Matrix[T]], y: Exp[Matrix[T]]) =
  (x,y) match {
    case (MatrixZero(m,n),b) => b
    case (a,MatrixZero(m,n)) => a
    case _ => super.matrix_plus(x,y)
  }
}
```

Can match on arbitrary patterns and perform arbitrary simplifications!

Code generation

- Time to produce something we can actually execute
- The LMS library provides a set of basic code generation facilities
  - Handles scheduling
    - tracking node dependencies, coming up with a correct program order
    - takes care of a lot of hairy details (handling nested scopes, etc.)
- All you (the DSL author) has to do is define code generators for your IR nodes

A simple code generator

```
trait Example3Codegen extends ScalaGenBase {
  val IR: Example3OpsExp
  import IR._

  def emitNode(sym: Sym[Any], rhs: Def[Any])(implicit stream:
    PrintWriter): Unit = rhs match {
    case MatrixNew((m,n) => emitValDef(sym, "new MatrixImpl(" +
    quote(m) + "," + quote(n) + ")")
    case MatrixTimes(x,y) => emitValDef(sym, quote(x) + " * " +
    quote(y))
    case MatrixTimesScalar(x,y) => emitValDef(sym, quote(x) + " *
    " + quote(y))
    case _ => super.emitNode(sym, rhs)
  }
}
```

A simple code generator, piece by piece

```
trait Example3ScalaGen extends ScalaGenBase {
  val IR: Example3OpsExp
  import IR._
```

- IR is a *path-dependent type*
- The code generator traits are not part of the same object as the IR – but they all need to agree on the same type Rep[T]
  - Reason: code generators for different targets should be kept separate (Scala, CUDA, C, etc.)

```
object Example3Runner extends Example3 with
    Example3OpsExp
object Example3Generator extends Example3ScalaGen {
    val IR = Example3Runner }
```

A simple code generator, piece by piece

```
override def emitNode(sym: Sym[Any], rhs: Def[Any])(implicit
  stream: PrintWriter): Unit = rhs match {
```

- GenericCodegen provides a default emitNode implementation that should be overridden by the generators for each node type
- GenericCodegen calls emitNode for each node after scheduling, in the correct order
  - If no-one implements it (it chains all the way back to the base class) it will throw a runtime exception

A simple code generator, piece by piece

```
case MatrixNew((m,n) => emitValDef(sym, "new MatrixImpl(" +
    quote(m) + "," + quote(n) + ")")
```

- Matching on nodes is normal Scala pattern matching
- The supplied string is exactly what will get written out in the generated file
- *emitValDef()* is a helper function defined for each target generator to declare a constant
- *quote()* is a helper function that returns a symbol's unique id (e.g. x13)
- Later on (in the Delite lecture) we will show how the Delite framework handles most of the code generation duties for you

Invoking code generation

```
object Example3Generator extends Example3ScalaGen {
  val IR = Example3Runner
}

object Example3Runner extends Example3 with
  Example3OpsExp {

  def main(args: Array[String]) {
    Example3Generator.emitSource((x: Rep[Unit]) => run(),
  "Application", new PrintWriter(System.out))
  }
}
```

Hit run… not bad!

```
/*****************************************
  Emitting Generated Code
*****************************************/
class Application extends ((Unit)=>(Unit)) {
def apply(x0:Unit): Unit = {
val x1 = new MatrixImpl(10,10)
val x2 = x1 * x1
val x3 = x2 * x1
val x4 = x3 * x1
val x5 = x4 * 1
val x6 = println(x5)
x6
}
}
/*****************************************
  End of Generated Code
*****************************************/

Process finished with exit code 0
```

This code almost works..

- Except for this “MatrixImpl” thing that doesn’t exist anywhere yet
- We don’t lift data structures into the IR (yet)
- So you generate calls to the constructor of a concrete class that you’ve defined somewhere
- Field accesses too..
  - case MatrixNumRows(x) => emitValDef(quote(x) + “.numRows”)

Data structures

- Just about everything besides construction and field access should not be defined in the real data structure
  - Instead, implement these as IR methods
  - Ex. MatrixApply(n) and MatrixUpdate(n,y) generate ".apply(n)" and ".update(n,y)"
  - But MatrixPlus can be implemented in terms of MatrixNew, MatrixApply, and MatrixUpdate, instead of being defined inside "MatrixImpl"
  - Anything that is emitted as a method call is a blackbox in the IR, and cannot be optimized
  - This will make more sense once you start playing with the code…

One other sneaky detail

- We've been overlooking it so far, but for Rep[Matrix[T]] to be a proper type, there has to be a type Matrix[T] somewhere
- If you look inside the examples, you'll see: class Matrix[T] // placeholder
- You need to define these "Interface" classes that contain the types that data structures used in generated code are expected to have

Data structure example

```
// Interfaces, to be used from generated code!

trait Matrix[T] {
  def numRows: Int
  def numCols: Int

  def apply(m: Int, n: Int): T
  def update(m: Int, n: Int, y: T): Unit
}


// Concrete data structure, to be used from generated code!

class MatrixImpl[T:Manifest](val numRows: Int, val numCols: Int) extends
    Matrix[T] {
  val _data = new Array[T](numRows*numCols)

  def apply(m: Int, n: Int) = _data(m*numCols + n)
  def update(m: Int, n: Int, y: T) { _data(m*numCols+n) = y }
}
```

Syms and friends

- For more complex IR nodes (usually those with nested blocks), you have to help the scheduler out
- **syms** is a method that finds dependencies: the default implementation is to grab every field in a case class
- **boundsyms** is used for lambdas: anything that is a bound sym will not be (and should not be) scheduled before the lambda, since it is only used inside

Syms and friends (2)

- for (i <- 0 until 100) { … }

```
trait BaseGenRangeOps extends GenericNestedCodegen {
  val IR: RangeOpsExp
  import IR._

  override def syms(e: Any): List[Sym[Any]] = e match {
    case RangeForeach(start, end, i, body) =>
     syms(start):::syms(end):::syms(body)
    case _ => super.syms(e)
  }

  override def boundSyms(e: Any): List[Sym[Any]] = e match {
    case RangeForeach(start, end, i, y) => i :: effectSyms(y)
    case _ => super.boundSyms(e)
  }
}
```

- What would happen if we left these out?

We're almost there…

- Just one small problem left to deal with

Side effects

- …which happens to be an amazing can of worms

Control dependencies and effects

- Side effects introduce a new set of ordering constraints on the IR
- They are very problematic in general
  - If A **may be** an alias for B, then every write to A must be treated as if it were also a write to B
  - Unless you can prove uniqueness, almost everything becomes serialized
  - Optimizations like code motion (op fusing, etc.) become impossible to apply

Restricting effects

- LMS takes a pragmatic, DSL-focused approach:
  - Let's not try to deal with arbitrary effects
  - But effects are still very useful, so we don't want to be fundamentalist
  - Restrict rather than disallow
- Rules
  - All symbols that might be mutated must be explicitly marked mutable by the DSL author
  - Nested mutable objects are not allowed
    - **var** x = y if y is mutable
    - **val** v = Vector(Vector(1,2,3,4)) if both vectors are mutable
  - Mutable objects cannot alias
    - a(i) = b if both a and b are mutable
    - (note that a(i) = b.clone is fine)

Tracking effects

- The DSL author is responsible for marking effectful operations
- LMS provides an API for doing so
  - reflectMutable // marks a symbol as mutable
  - reflectWrite // marks a write to a mutable symbol (if the symbol is not mutable, will print an error!)
  - reflectEffect // marks a general side-effect (e.g. println). All effects are totally ordered!

Effects example

```
trait Example4OpsExp extends Example4Ops with EffectExp {
  case class MatrixNew[T:Manifest](x: Exp[Int], y: Exp[Int]) extends
    Def[Matrix[T]]
  case class MatrixApply[T:Manifest](x: Exp[Matrix[T]], m: Exp[Int], n:
    Exp[Int]) extends Def[T]
  case class MatrixUpdate[T:Manifest](x: Exp[Matrix[T]], m: Exp[Int],
    n: Exp[Int], y: Exp[T]) extends Def[Unit]
  case class MatrixPrint[T:Manifest](x: Exp[Matrix[T]]) extends
    Def[Unit]


  def matrix_new[T:Manifest](x: Exp[Int], y: Exp[Int]) =
    reflectMutable(MatrixNew[T](x,y))
  def matrix_apply[T:Manifest](x: Exp[Matrix[T]], m: Exp[Int], n:
    Exp[Int]) = MatrixApply(x,m,n)
  def matrix_update[T:Manifest](x: Exp[Matrix[T]], m: Exp[Int], n:
    Exp[Int], y: Exp[T]) = reflectWrite(x)(MatrixUpdate(x,m,n,y))
  def matrix_print[T:Manifest](x: Exp[Matrix[T]]) =
    reflectEffect(MatrixPrint(x))
}
```

That's it

- Look for the examples in the slides online
- Good luck on your projects!
- Questions?

LANGUAGE VIRTUALIZATION

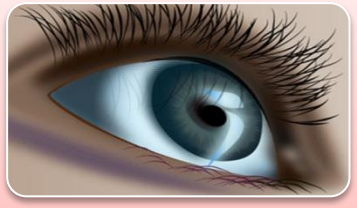
Onward! `10: Language Virtualization for Heterogeneous Parallel Computing

(see class website)

Language Virtualization

- A host language is **virtualizable** if it allows the implementation of embedded DSLs that are virtually indistinguishable from a stand-alone language
- Most of the power of a standalone language, with much less work

Embedding Language Requirements



Expressiveness

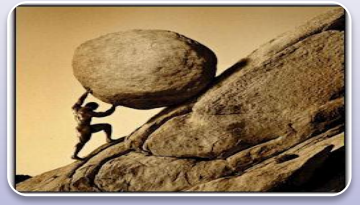
- Encompasses syntax, semantics and general ease of use for domain experts

Performance

- Embedded language must me amenable to extensive static and dynamic analysis, optimization and code generation

Safety

- Preserve type safety of embedded language
- Optimizations can be applied safely



Modest Effort

- Virtualization is only useful if it reduces effort to embed high performance DSL

Expressiveness

- OOP allowed higher level of abstractions
  - Add your own types and define operations on them
  - But how about custom type interaction with language features
- Overload all relevant embedding language constructs

```
for (x <- elems if x % 2 == 0) p(x)
```

maps to

```
elems.withFilter(x => x % 2 == 0).foreach(x => p(x))
```

- DSL developer can control how loops over domain collection should be represented and executed by implementing withFilter and foreach for their DSL type

Expressiveness

- Need to apply similar techniques to all other relevant constructs of the embedding language (for example)

```
if (cond) something else somethingElse
```

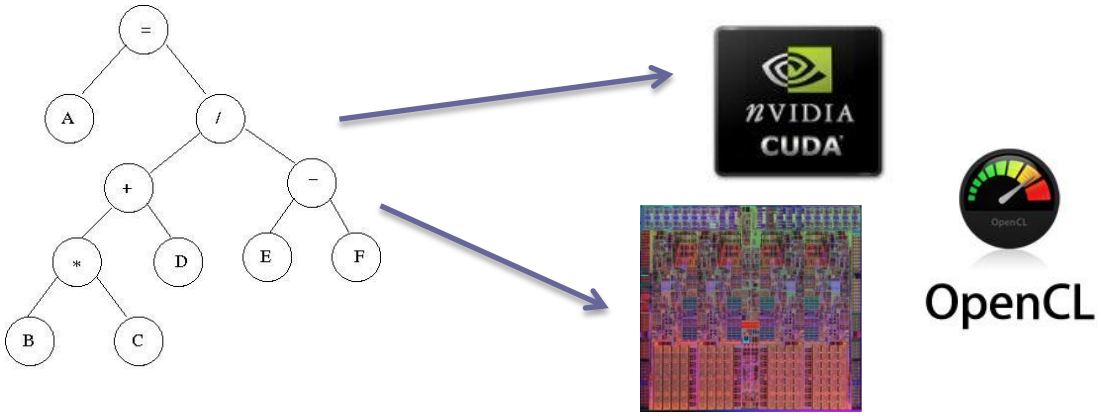
maps to

```
__ifThenElse(cond, something, somethingElse)
```

- DSL developer can control the meaning of conditionals by providing overloaded variants specialized to DSL types

Performance

- Requires the ability to support optimization and code generation in embedded DSLs
  - Implies that embedded programs must be available in some lifted intermediate representation
  - Customizing IR allows for domain-specific optimization and heterogeneous code generation



Safety

- Typed DSL should be embedded in a typed embedding language
- Plain AST-like representations would allow DSL program to get access to part of their own structure which in addition to being unsafe, can render optimizations unsound

```
def foo(x: Exp[Int]) = {
  val y = x + 1
  if (y.isInstanceOf[Plus])
    doNothing
  else
    killKittens
}
```

- Invoking foo(2) allows us to optimize program and calculate y during compile time
  - Unsound if program can access the DSL's AST

Modest Effort

- Lifting each new DSL that uses slightly different IR violates Effort criterion
- Need a DSL embedding infrastructure
- Provide building blocks of common DSL functionality
  - IR, analysis, optimizations, code generation