ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Scala *for* MULTICORE

PART 2: *Parallel Collections and Parallel DSLs*

Philipp HALLER, STANFORD UNIVERSITY AND EPFL

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

RESOURCES ONLINE AT:
http://lamp.epfl.ch/~phaller/upmarc

Scala *for* MULTICORE

PART 2: *Parallel Collections and Parallel DSLs*

Philipp HALLER, STANFORD UNIVERSITY AND EPFL

BUT FIRST,

Let's pick up from where we left off yesterday...

Goal of Scala Actors

Programming system for Erlang-style actors that:

- offers high scalability on mainstream platforms;
- integrates with thread-based code;
- provides safe and efficient message passing.

Goal of Scala Actors

Programming system for Erlang-style actors that:

- ✓ offers high scalability on mainstream platforms;
- ✓ integrates with thread-based code;
- ⊛ provides safe and efficient message passing.

Safe and Efficient Message Passing.

It's possible to produce data races with actors:

Safe and Efficient Message Passing.

It's possible to produce data races with actors:

- Pass **a reference to a mutable object** in a message

Safe and Efficient Message Passing.

It's possible to produce data races with actors:

- Pass **a reference to a mutable object** in a message
- Two actors **accessing the same mutable object** can lead to data races

Safe and Efficient Message Passing.

It's possible to produce data races with actors:

- Pass **a reference to a mutable object** in a message
- Two actors **accessing the same mutable object** can lead to data races

How do we prevent that?

Safe and Efficient Message Passing.

It's possible to produce data races with actors:

- Pass **a reference to a mutable object** in a message
- Two actors **accessing the same mutable object** can lead to data races

How do we prevent that?

- Make sure mutable objects are **unusable after they have been sent**

Safe and Efficient Message Passing.

It's possible to produce data races with actors:

- Pass **a reference to a mutable object** in a message
- Two actors **accessing the same mutable object** can lead to data races

How do we prevent that?

- Make sure mutable objects are **unusable after they have been sent**
- Use a compiler plugin to **check whether a variable is unusable**

The Compiler Plugin.

The compiler plugin checks at which point a variable is transferred and becomes unusable.

The Compiler Plugin.

The compiler plugin checks at which point a variable is transferred and becomes unusable.

This check is done in two steps:

I. MARK variables that we want to send in messages.

- Add `@unique` annotation to their type.

The Compiler Plugin.

The compiler plugin checks at which point a variable is transferred and becomes unusable.

This check is done in two steps:

1. MARK variables that we want to send in messages.
   - Add `@unique` annotation to their type.
2. Go through program and track which variables are USABLE/UNUSABLE.
   - Run additional type checking phase on program

Extending Type Checking.

An annotated variable

```
val buf: ArrayBuffer[Int] @unique
```

has a **type guarded with a capability:**

```
buf: ρ▻ArrayBuffer[Int]
```

Extending Type Checking.

An annotated variable

```
val buf: ArrayBuffer[Int] @unique
```

has a **type guarded with a capability:**

```
buf: ρ▻ArrayBuffer[Int]
```

Key idea:

A variable with guarded type is **only usable when its capability is available**

Tracking Capabilities.

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
  someActor ! buf
  buf.remove(0)
}
```

Tracking Capabilities.

LOCAL VARIABLES: **CAPABILITIES:**

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
  someActor ! buf
  buf.remove(0)
}
```

Tracking Capabilities.

LOCAL VARIABLES: **CAPABILITIES:**

∅

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
  someActor ! buf
  buf.remove(0)
}
```

Tracking Capabilities.

LOCAL VARIABLES: **CAPABILITIES:**

∅
∅

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
  someActor ! buf
  buf.remove(0)
}
```

Tracking Capabilities.

LOCAL VARIABLES:

`sum: Int`

CAPABILITIES:

∅

∅

∅

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
  someActor ! buf
  buf.remove(0)
}
```

Tracking Capabilities.

LOCAL VARIABLES:	CAPABILITIES:
	∅
	∅
`sum: Int`	∅
`sum: Int buf: ρ▻Buffer[Int]`	ρ

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
  someActor ! buf
  buf.remove(0)
}
```

Tracking Capabilities.

LOCAL VARIABLES:	CAPABILITIES:
	∅
	∅
`sum: Int`	∅
`sum: Int buf: ρ▻Buffer[Int]`	ρ
`sum: Int buf: ρ▻Buffer[Int]`	ρ

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
  someActor ! buf
  buf.remove(0)
}
```

Tracking Capabilities.

LOCAL VARIABLES:	CAPABILITIES:
	∅
	∅
`sum: Int`	∅
`sum: Int buf: ρ▻Buffer[Int]`	ρ
`sum: Int buf: ρ▻Buffer[Int]`	ρ
`sum: Int buf: ρ▻Buffer[Int]`	ρ

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
  someActor ! buf
  buf.remove(0)
}
```

Tracking Capabilities.

LOCAL VARIABLES:	CAPABILITIES:
	∅
	∅
sum: Int	∅
sum: Int buf: ρ▻Buffer[Int]	ρ
sum: Int buf: ρ▻Buffer[Int]	ρ
sum: Int buf: ρ▻Buffer[Int]	ρ
sum: Int buf: ρ▻Buffer[Int]	∅

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
  someActor ! buf
  buf.remove(0)
}
```

Tracking Capabilities.

LOCAL VARIABLES:	CAPABILITIES:
	∅
	∅
`sum: Int`	∅
`sum: Int buf: ρ▻Buffer[Int]`	ρ
`sum: Int buf: ρ▻Buffer[Int]`	ρ
`sum: Int buf: ρ▻Buffer[Int]`	ρ
`sum: Int buf: ρ▻Buffer[Int]`	∅

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
  someActor ! buf
  buf.remove(0)
}
```

Error: buf has type ρ▻`Buffer[Int]` but capability ρ is not available

```
  buf.remove(0)
  ^
```

Tracking Capabilities.

LOCAL VARIABLES:	CAPABILITIES:
	∅
	∅
`sum: Int`	∅
`sum: Int buf: ρ▻Buffer[Int]`	ρ
`sum: Int buf: ρ▻Buffer[Int]`	ρ
`sum: Ir`	
`sum: In`	

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
```

The extended type checker ensures mutable objects are no longer accessed after they have been consumed.

Error: buf has type ρ▻`Buffer[Int]` but capability ρ is not available

```
  buf.remove(0)
  ^
```

Tracking Capabilities.

LOCAL VARIABLES:	CAPABILITIES:
	∅
	∅
`sum: Int`	∅
`sum: Int buf: ρ▻Buffer[Int]`	ρ
`sum: Int buf: ρ▻Buffer[Int]`	ρ
`sum: In`	
`sum: In`	

```
actor {
  var sum = 2 + 3
  val buf: Buffer[Int]@unique =
    new ArrayBuffer[Int]
  buf += sum
```

The extended type checker ensures mutable objects are no longer accessed after they have been consumed.

THUS,
Uniqueness types can be used to ensure actors are isolated.

^

Implementation and Experience.

Plug in for Scala compiler

- Erases capabilities and `capturedBy` for code generation

Practical experience:

	size [LOC]	changes [LOC]	property checked
mutable collections	2046	60	collections self-contained
partest	4182	61	actor isolation
ray tracer	414	18	actor isolation

External vs. Separate Uniqueness

EXTERNAL UNIQUENESS

- No external aliases
- No unique method receivers
- Deep/full encapsulation

[*Clarke, Wrigstad 2003; Müller, Rudich 2007; Clarke et al. 2008*]

SEPARATE UNIQUENESS

THIS TALK.

- Local external aliases
- Unique method receivers (self transfer)
- Full encapsulation

Goal of Scala Actors?

REVISITED. (AGAIN)

Programming system for Erlang-style actors that:

- ✓ offers high scalability on mainstream platforms;
- ✓ integrates with thread-based code;
- ⊛ provides safe and efficient message passing.

Goal of Scala Actors?

REVISITED. (AGAIN)

Programming system for Erlang-style actors that:

- ✓ offers high scalability on mainstream platforms;
- ✓ integrates with thread-based code;
- ✓ provides safe and efficient message passing.

CAPABILITIES FOR UNIQUENESS

- Lightweight pluggable type system.
- Race-freedom through actor isolation.

[Haller and Odersky. **Capabilities for uniqueness and borrowing,** ***Proc. ECOOP, 2010***]

[Haller. **Isolated Actors for Race-Free Concurrent Programming,** ***PhD Thesis, 2010***]

Summary: Actors

- Scalable Erlang-style actors
- Integration of thread-based and event-based programming
- Used in large-scale production systems
- Lightweight uniqueness types for actor isolation

Parallel Collections

NEW!
in 2.9!

Collections?

THE COLLECTIONS MENTALITY:

Collections are literally collections of data elements, which you can perform operations on.

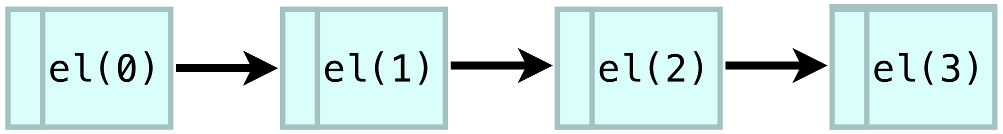
Collections?

THE COLLECTIONS MENTALITY:

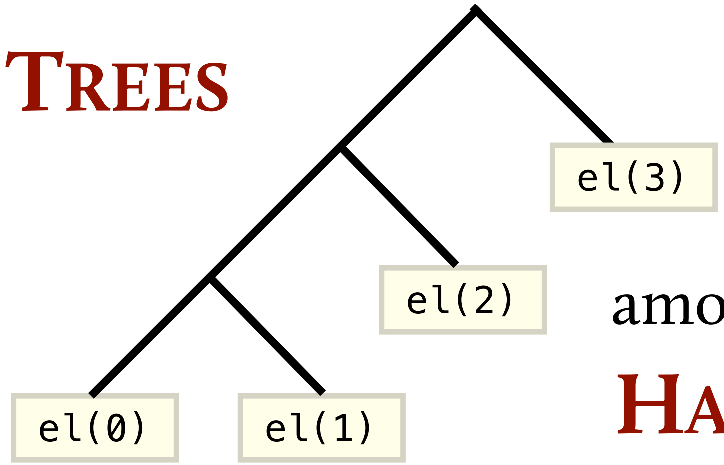
Collections are literally collections of data elements, which you can perform operations on.

A collection can be represented by
any data structure, like:

LINKED LISTS



TREES



amongst others...

HASH MAPS, RED-BLACK TREES, ETC.

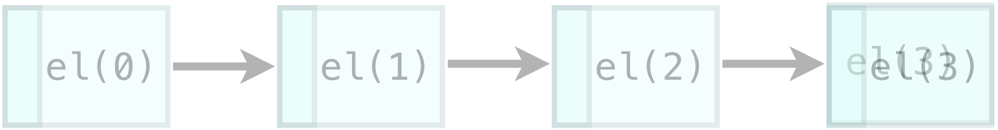
Collections?

THE COLLECTIONS MENTALITY:

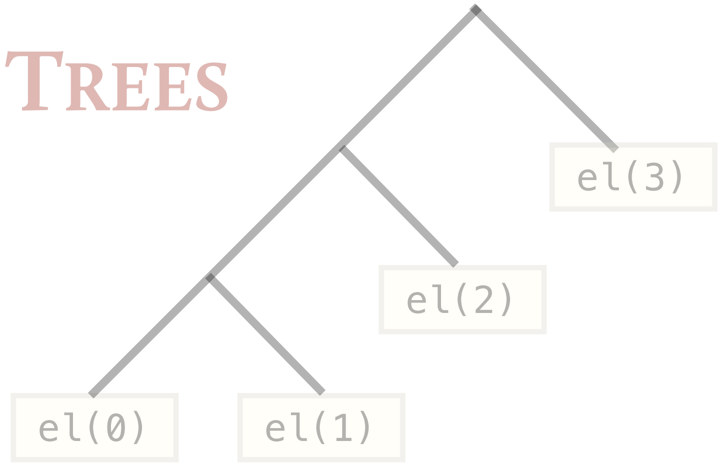
Collections are literally collections of data elements, which you can perform operations on.

A collection can be represented by any data structure, like:

LINKED LISTS



TREES



Each of which has a set of operations you can perform on it:

Menu

operations of the day:

`map`
`foreach`
`forall`
`groupBy`
`reduce`
`count`
`indexOf`
`sorted`

Collections?

Say you have *some* collection:

```
val myCollection: List[Int] = List(1,2,3,4,5)
```

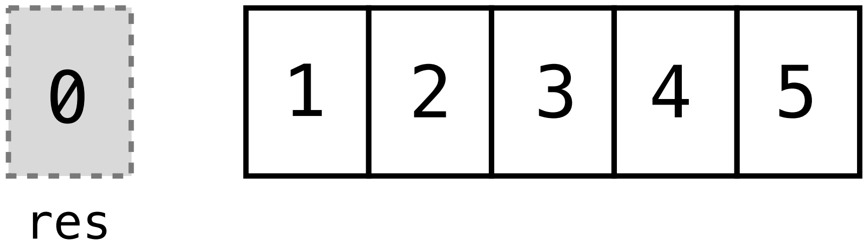
Collections?

Say you have *some* collection:

```
val myCollection: List[Int] = List(1,2,3,4,5)
```

We can perform an operation on that collection:

```
myCollection.foldLeft(0)((a,b) => a+b)
```



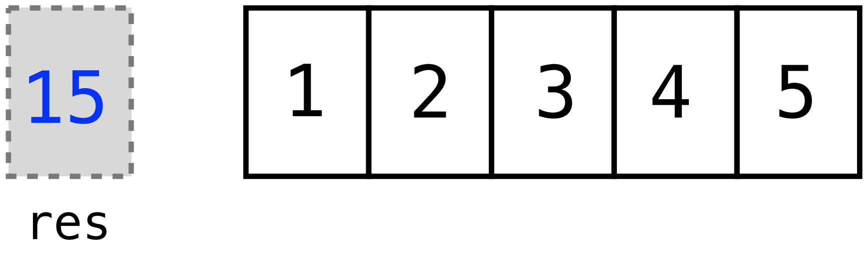
Collections?

Say you have *some* collection:

```
val myCollection: List[Int] = List(1,2,3,4,5)
```

We can perform an operation on that collection:

```
myCollection.foldLeft(0)((a,b) => a+b)
```



Collections Hierarchy.

Collections are organized in two packages.

Collections Hierarchy.

Collections are organized in two packages.

scala.collection.**mutable**

scala.collection.**immutable**

Collections Hierarchy.

Collections are organized in two packages.

scala.collection.**mutable**

Can change, add, or remove elements in place **as a side effect**

scala.collection.**immutable**

Collections Hierarchy.

Collections are organized in two packages.

scala.collection.**mutable**

Can change, add, or remove elements in place **as a side effect**

scala.collection.**immutable**

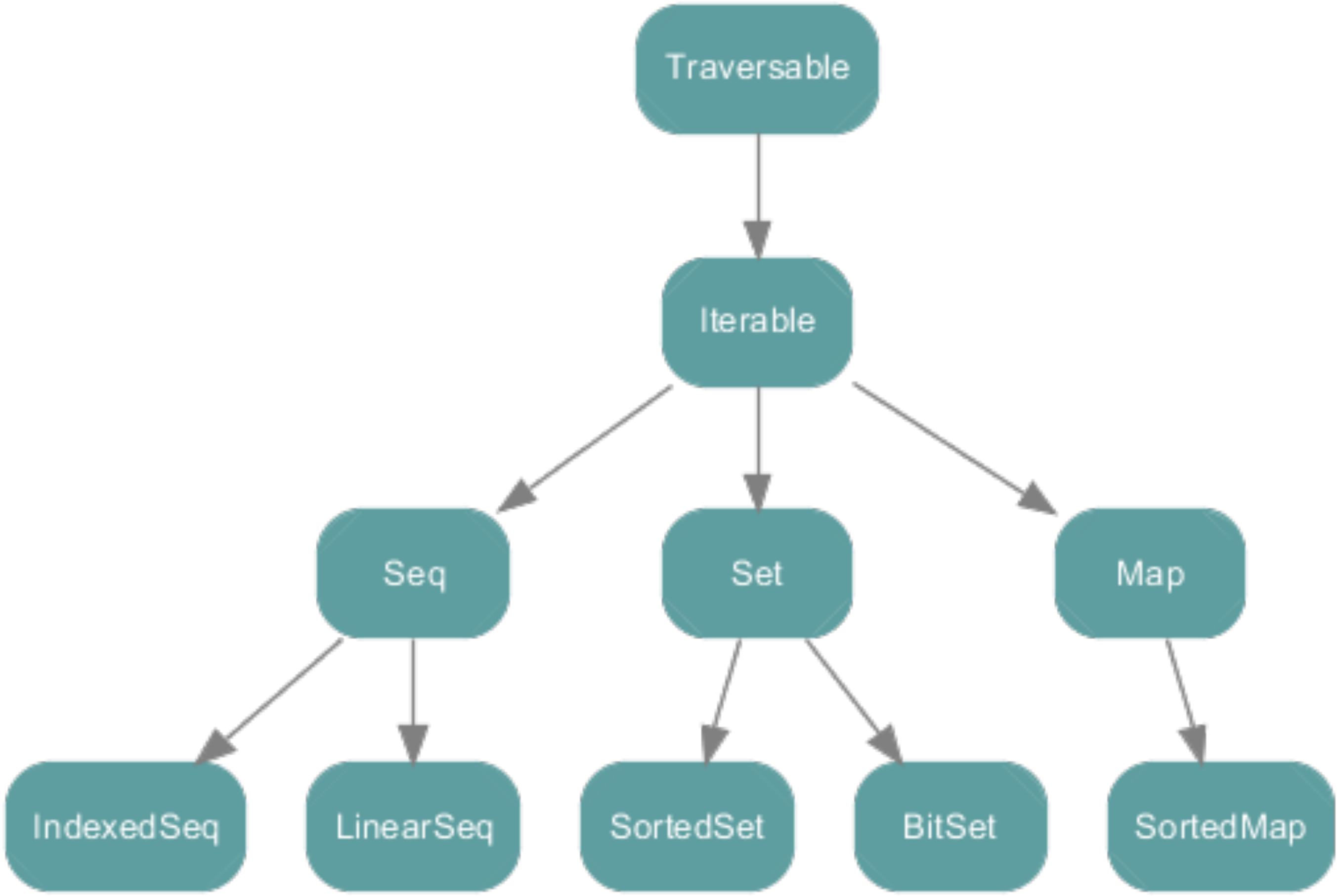
Methods that transform an immutable collection **return a new collection** and leave the old collection unchanged

Collections Hierarchy.

Collections are organized in two packages.

scala.collection.**mutable** scala.collection.**immutable**

Abstract classes in scala.**collection**



Parallel Collections

Scala 2.9 introduces *Parallel Collections*, based on the idea that many operations can safely be performed in parallel.

Parallel Collections

Scala 2.9 introduces *Parallel Collections*, based on the idea that many operations can safely be performed in parallel.

Just add `.par`

And the same operation is performed in parallel:

```
myCollection.par.foldLeft(0)((a,b) => a+b)
```

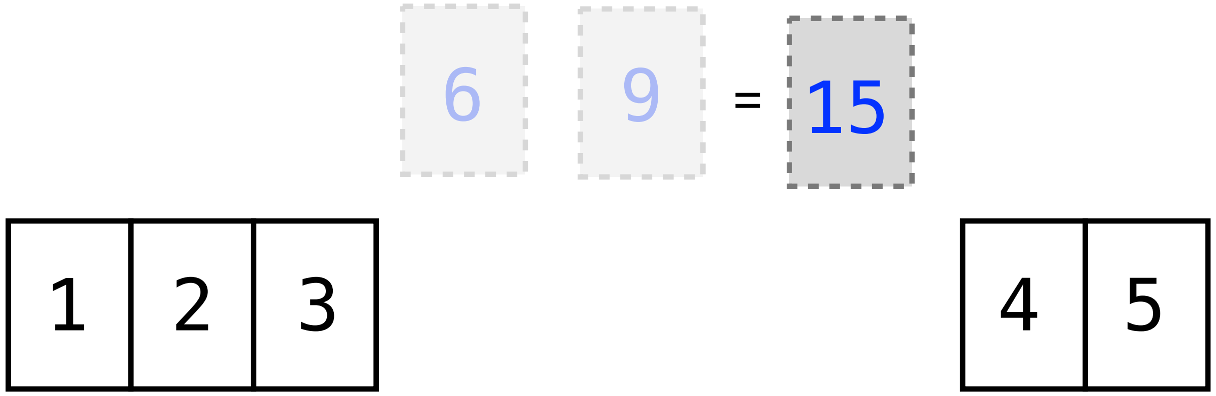
Parallel Collections

Scala 2.9 introduces *Parallel Collections*, based on the idea that many operations can safely be performed in parallel.

Just add `.par`

And the same operation is performed in parallel:

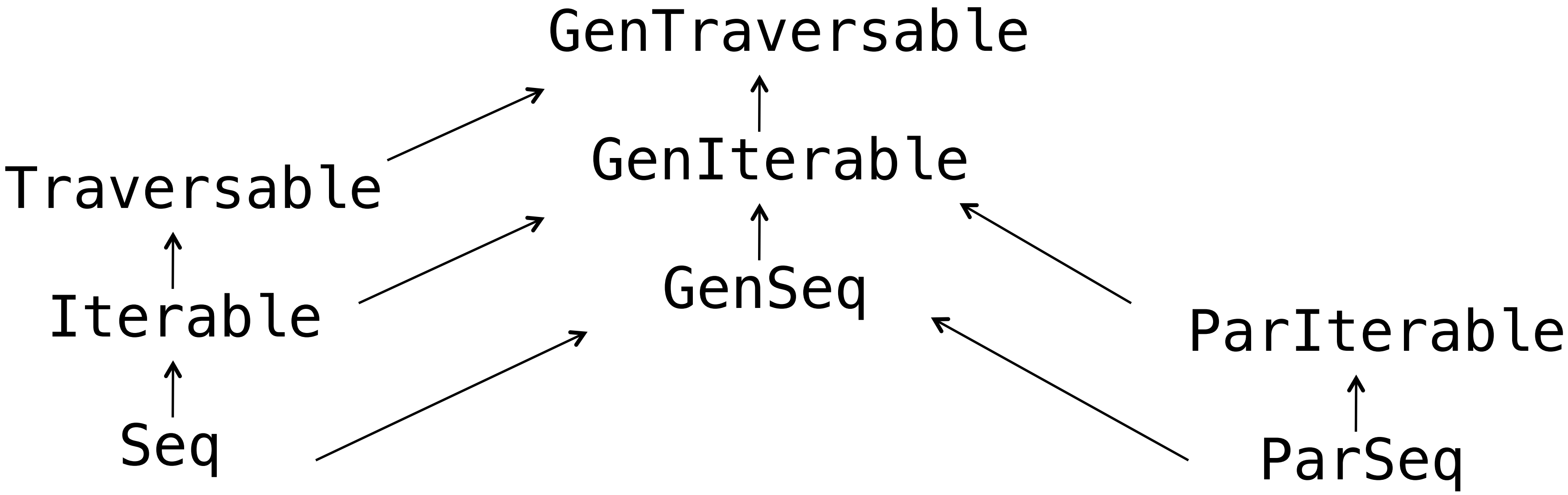
```
myCollection.par.foldLeft(0)((a,b) => a+b)
```



.`par`

- **New method** added to regular collections
- Returns a **parallel version of the collection** pointing to the same underlying data
- Use .`seq` to go back to the sequential collection
- Parallel sequences, maps, and sets defined in separate hierarchy

Parallel Collections Hierarchy


GenTraversable

GenIterable

GenSeq

Traversable

Iterable

Seq

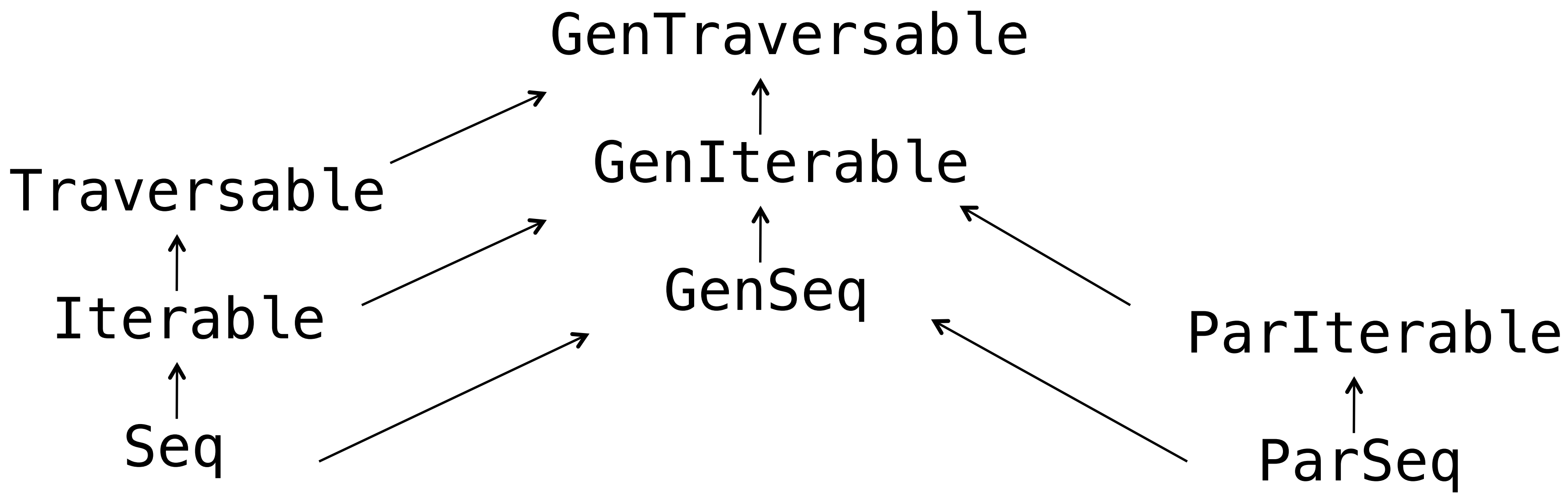
ParIterable

ParSeq

Parallel Collections Hierarchy

Immutable parallel collections:

`ParRange`
`ParVector`
`ParHashMap`
`ParHashSet`



`GenTraversable`

`GenIterable`

`GenSeq`

`Traversable`

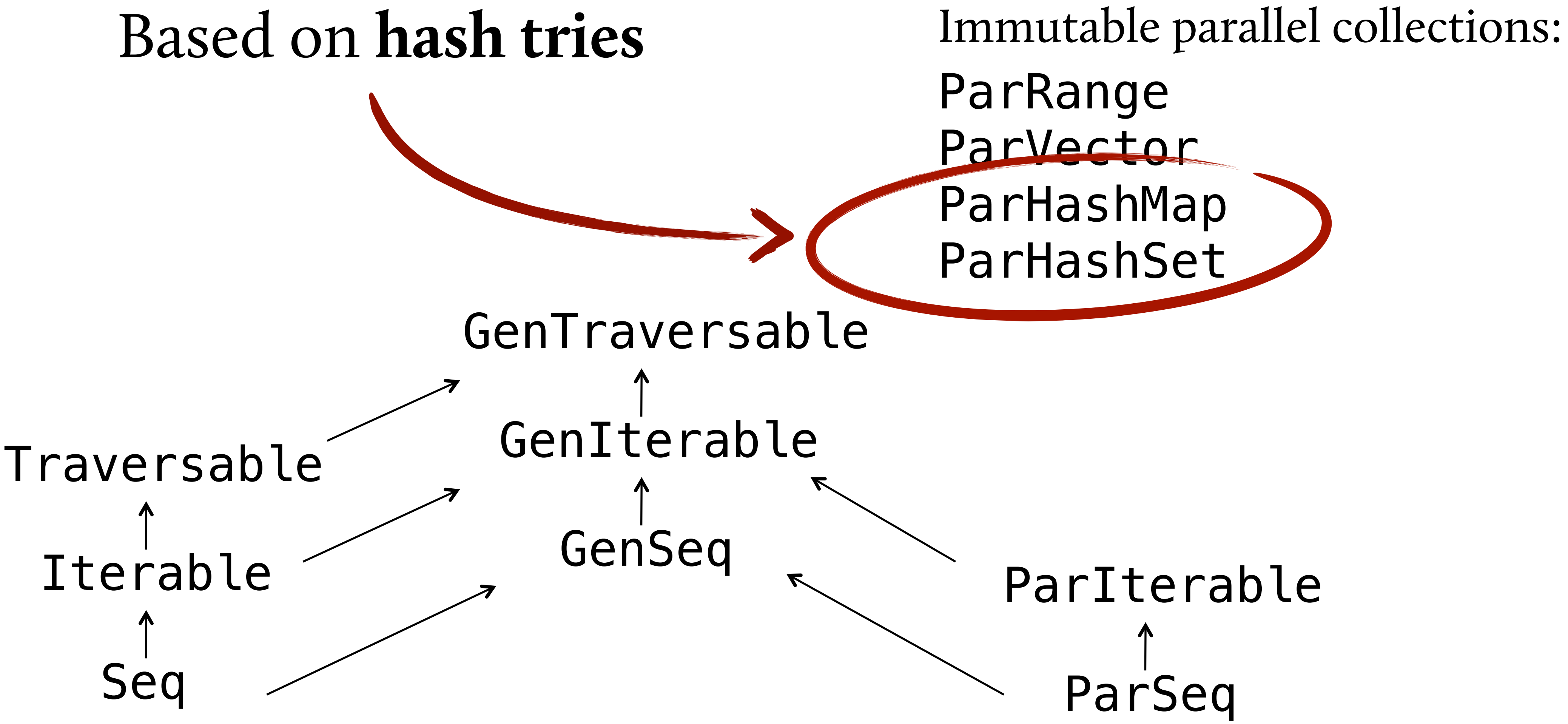
`Iterable`

`Seq`

`ParIterable`

`ParSeq`

Parallel Collections Hierarchy

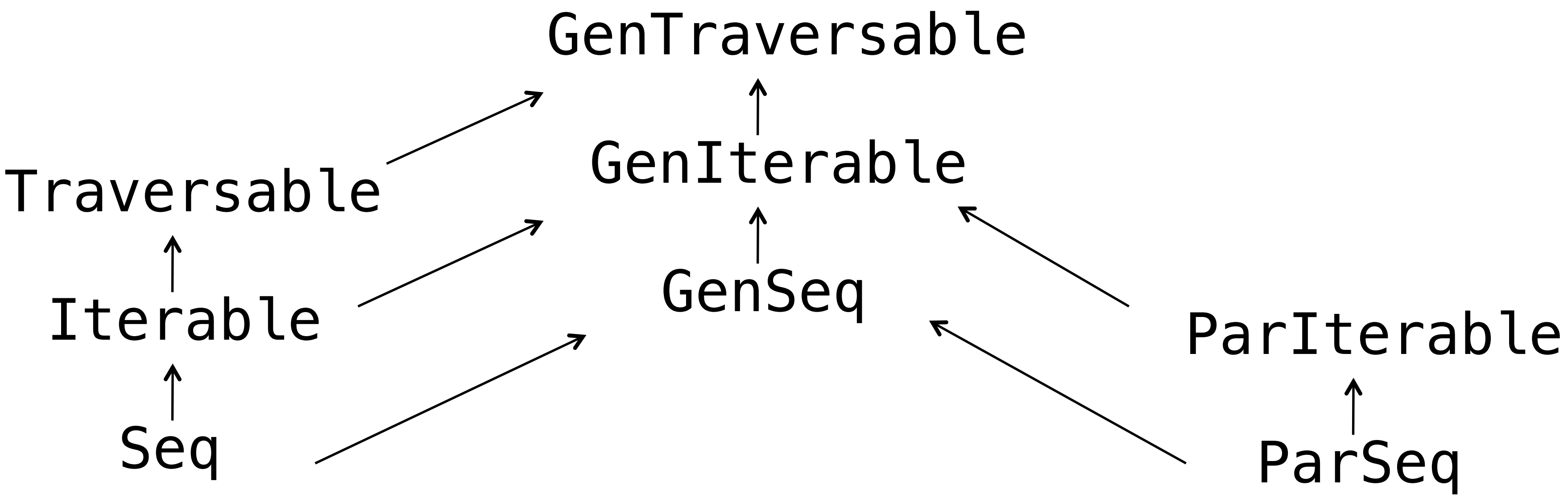


Based on **hash tries**

Immutable parallel collections:

`ParRange`
`ParVector`
`ParHashMap`
`ParHashSet`

Parallel Collections Hierarchy



GenTraversable

GenIterable

GenSeq

Traversable

Iterable

Seq

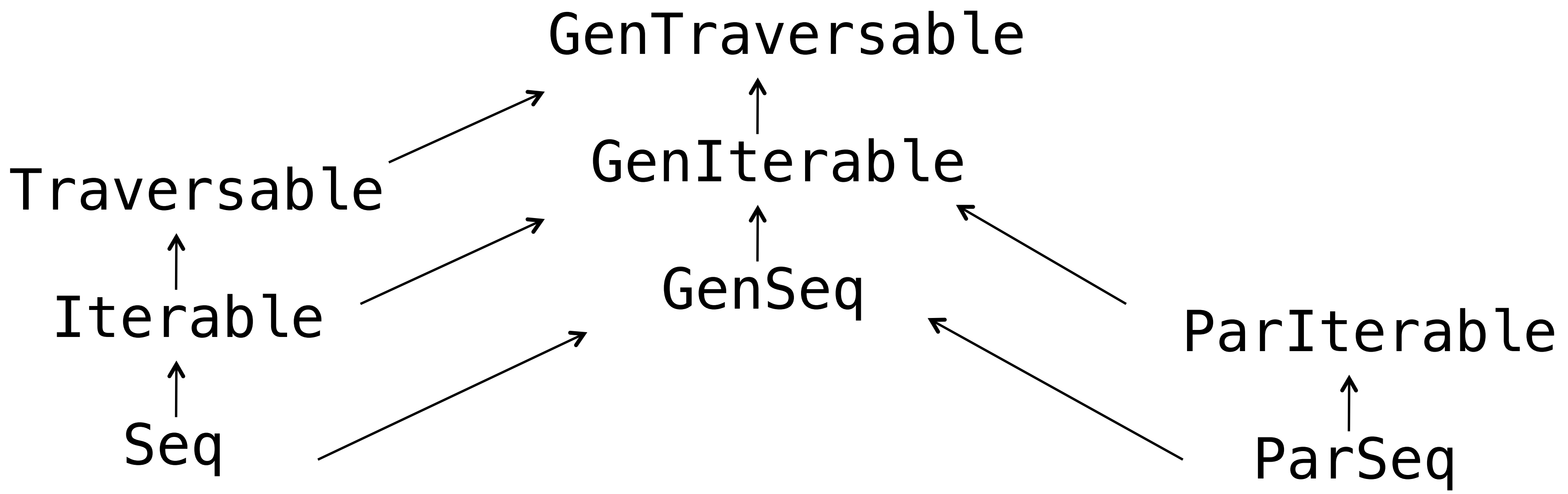
ParIterable

ParSeq

Parallel Collections Hierarchy

Mutable parallel collections:

`ParArray`
`ParHashMap`



GenTraversable

GenIterable

GenSeq

Traversable

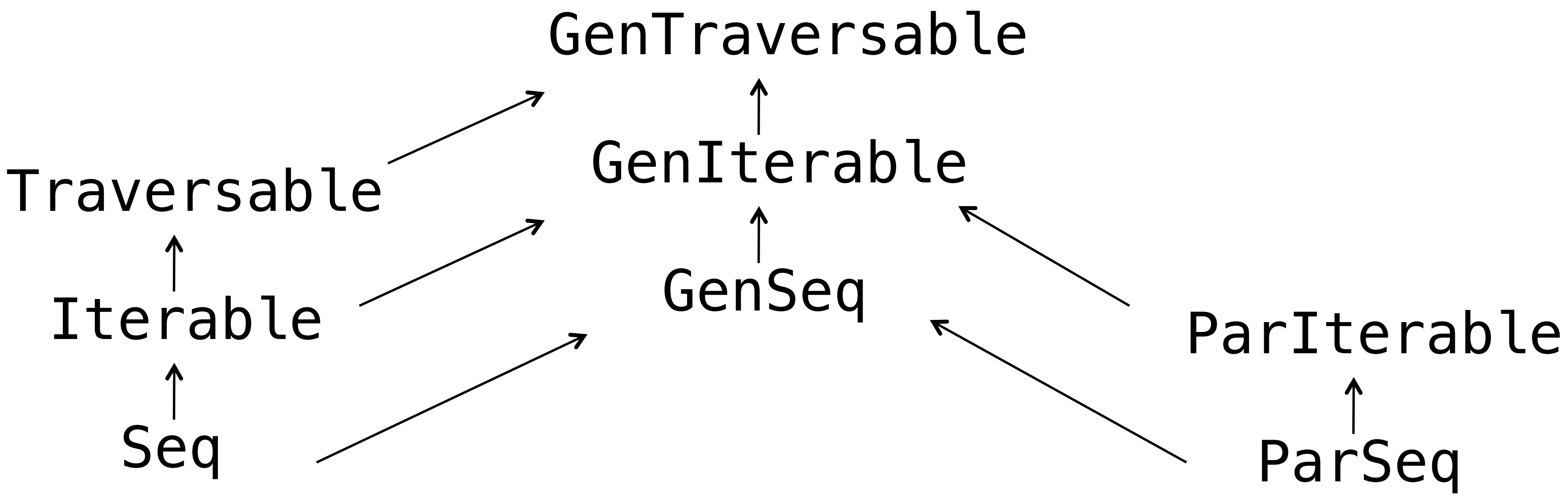
Iterable

Seq

ParIterable

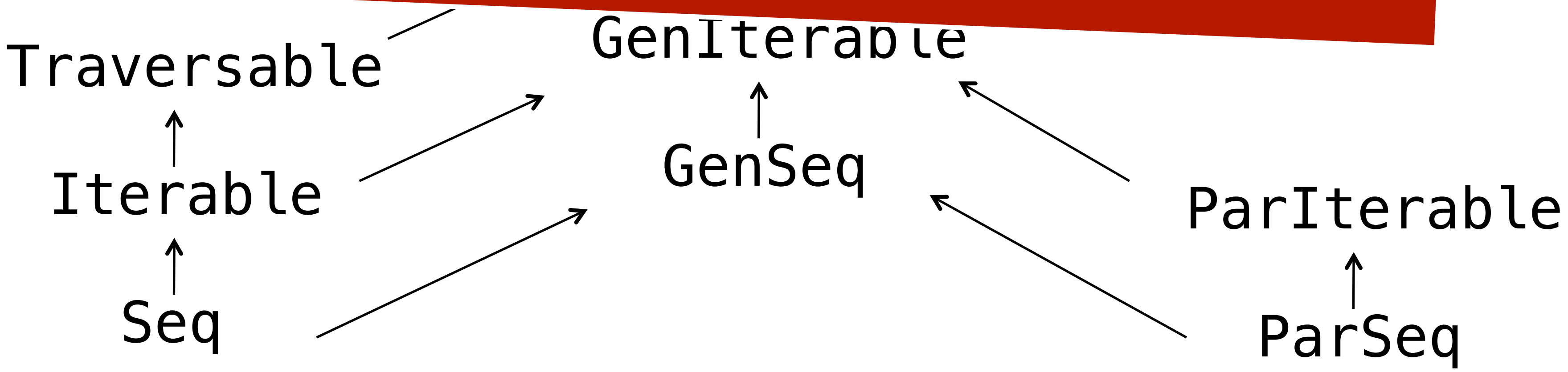
ParSeq

Parallel Collections Hierarchy



Parallel Collections Hierarchy

Why isn't a `ParSeq` a `Seq`?



`Traversable`

`Iterable`

`Seq`

`GenIterable`

`GenSeq`

`ParIterable`

`ParSeq`

Seq vs. ParSeq

```
def nonEmpty(sq: Seq[String]) = {
  val res = new mutable.ArrayBuffer[String]()
  for (s <- sq) {
    if (s.nonEmpty) res += s
  }
  res
}
```

Seq vs. ParSeq

```
def nonEmpty(sq: ParSeq[String]) = {
  val res = new mutable.ArrayBuffer[String]()
  for (s <- sq) {
    if (s.nonEmpty) res += s
  }
  res
}
```

Seq vs. ParSeq

```
def nonEmpty(sq: ParSeq[String]) = {
  val res = new mutable.ArrayBuffer[String]()
  for (s <- sq) {
    if (s.nonEmpty) res += s
  }
  res
}
```

Seq vs. ParSeq

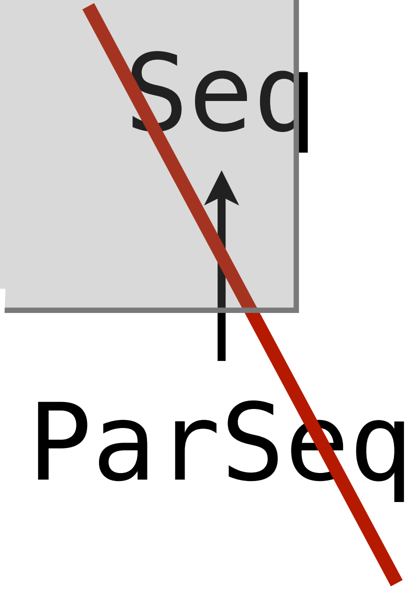
```
def nonEmpty(sq: ParSeq[String]) = {
  val res = new mutable.ArrayBuffer[String]()
  for (s <- sq) {
    if (s.nonEmpty) res += s
  }
  res
}
```

Side effect!
ArrayBuffer's += is not atomic!

Seq vs. ParSeq

```
def nonEmpty(sq: ParSeq[String]) = {
  val res = new mutable.ArrayBuffer[String]()
  for (s <- sq) {
    if (s.nonEmpty) res += s
  }
  res
}
```

Side effect!
ArrayBuffer’s += is not atomic!



~~Seq~~

~~ParSeq~~

Implementing Parallel Collections.

Implementing Parallel Collections.

GOAL: define operations in terms of *a few common abstractions*

- Typically, in terms of a foreach method or iterators
- However, their sequential nature makes these approaches **ill-suited for parallel execution!**

Implementing Parallel Collections.

GOAL: define operations in terms of *a few common abstractions*

- Typically, in terms of a foreach method or iterators
- However, their sequential nature makes these approaches **ill-suited for parallel execution!**

INSTEAD: **abstractions for splitting and combining**

- Split collection into non-trivial partition
- Iterate over disjunct subsets in parallel
- Combine partial results computed in parallel

Splitters and Combiners.

Splitters and Combiners.

- A splitter is an iterator that can be recursively split into disjoint iterators:

```
trait Splitter[T] extends Iterator[T] {
  def split: Seq[Splitter[T]]
}
```

Splitters and Combiners.

- A splitter is an iterator that can be recursively split into disjoint iterators:

```
trait Splitter[T] extends Iterator[T] {
  def split: Seq[Splitter[T]]
}
```

- A combiner combines partial results
  - The combine method returns a combiner containing the union of its argument elements
  - Results from different tasks are combined in a tree-like manner

```
trait Combiner[T, Coll] extends Builder[T, Coll] {
  def combine(other: Combiner[T, Coll]): Combiner[T, Coll]
}
```

Implementing ParArray.

SPLITTERS

- Hold a reference to the array and iteration bounds
- Divide the iteration range into two equal parts

Implementing ParArray.

SPLITTERS

- Hold a reference to the array and iteration bounds
- Divide the iteration range into two equal parts

```
class ArraySplitter[T](a: Array[T], start: Int, end: Int)
  extends Splitter[T] {

  def split = Seq(
    new ArraySplitter(a, start, (start + end) / 2),
    new ArraySplitter(a, (start + end) / 2, end))
}
```

Implementing ParArray.

COMBINERS

Implementing ParArray.

COMBINERS

- The final array size is not known in advance
  - The result array must be **constructed lazily**

Implementing ParArray.

COMBINERS

- The final array size is not known in advance
  - The result array must be **constructed lazily**
- Maintain elements in linked list of buffers

Implementing ParArray.

COMBINERS

- The final array size is not known in advance
  - The result array must be **constructed lazily**
- Maintain elements in linked list of buffers
- The result method allocates the array, and copies the chunks into the array in parallel

Implementing ParArray.

COMBINERS

- The final array size is not known in advance
  - The result array must be **constructed lazily**

- Maintain elements in linked list of buffers
- The result method allocates the array, and copies the chunks into the array in parallel

```
class ArrayCombiner[T] extends Combiner[T, ParArray[T]] {
  val chunks = LinkedList[Buffer[T]]() += Buffer[T]()
  def +=(elem: T) = chunks.last += elem
  def combine(that: ArrayCombiner[T]) = chunks append that.chunks
  def result = exec(new Copy(chunks,
                    new Array[T](chunks.fold(0)(_+_.size))))
}
```

Summary.

Summary.

- Simple transition from regular collections to parallel collections (“just add .`par`!”)
  - If access patterns aren’t inherently sequential

Summary.

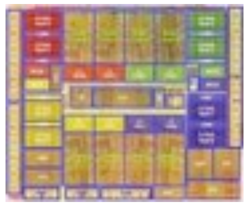
- Simple transition from regular collections to parallel collections ("just add `.par`!")
  - If access patterns aren't inherently sequential
- Parallel collection types do not extend sequential collection types
  - To avoid breaking existing code with side effects

Summary.

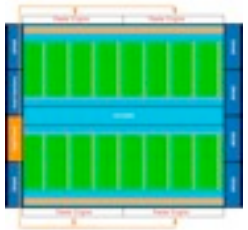
- Simple transition from regular collections to parallel collections ("just add `.par`!")
  - If access patterns aren't inherently sequential
- Parallel collection types do not extend sequential collection types
  - To avoid breaking existing code with side effects
- Parallel collections are implemented in terms of **splitters and combiners**
  - Parallel collections must provide efficient implementations of those

Heterogeneous Parallel DSLs

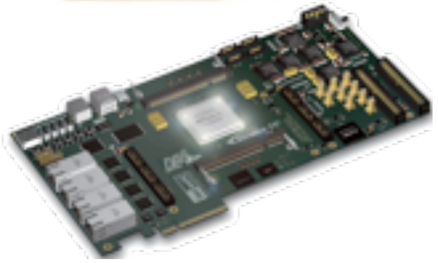
Heterogeneous Parallel Programming

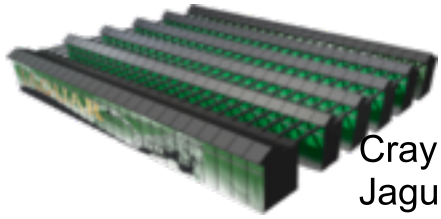


Sun
T2



Nvidia
Fermi

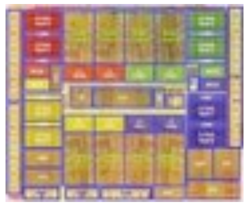




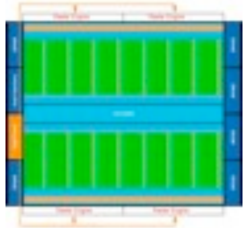
Cray
Jaguar

Heterogeneous Parallel Programming

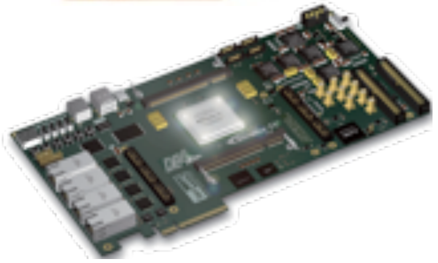
Pthreads
OpenMP

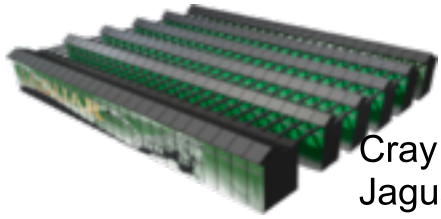


Sun
T2



Nvidia
Fermi

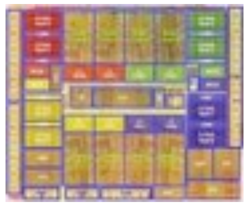




Cray
Jaguar

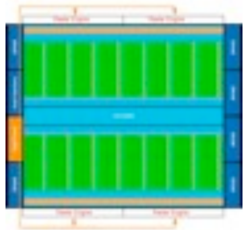
Heterogeneous Parallel Programming

Pthreads
OpenMP

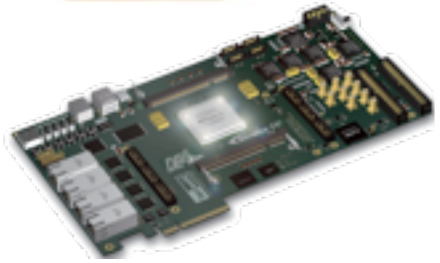


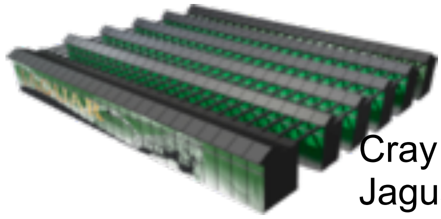
Sun
T2

CUDA
OpenCL



Nvidia
Fermi

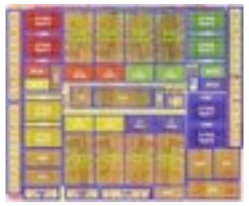




Cray
Jaguar

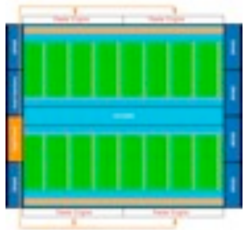
Heterogeneous Parallel Programming

Pthreads
OpenMP



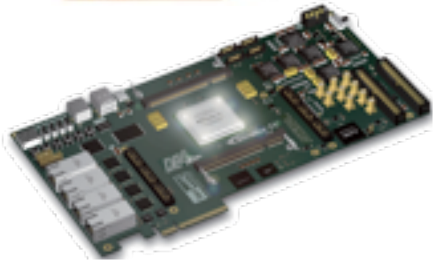
Sun
T2

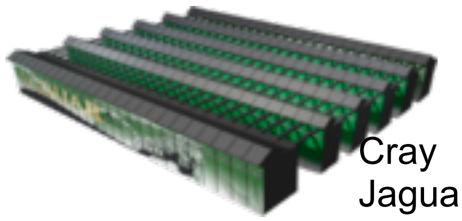
CUDA
OpenCL



Nvidia
Fermi

Verilog
VHDL

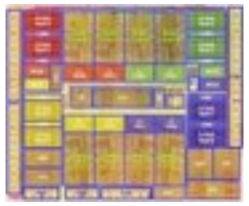




Cray
Jaguar

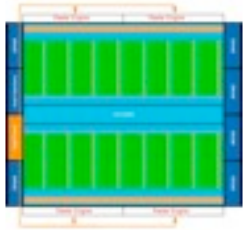
Heterogeneous Parallel Programming

Pthreads
OpenMP



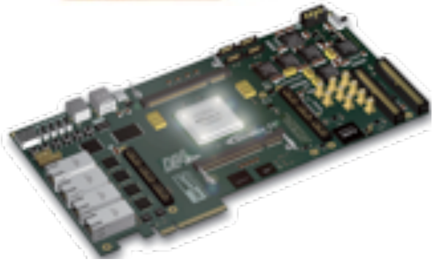
Sun
T2

CUDA
OpenCL

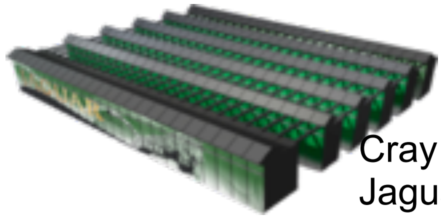


Nvidia
Fermi

Verilog
VHDL



MPI



Cray
Jaguar

Heterogeneous Parallel Programming

Applications

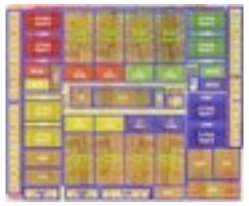
Scientific Engineering

Virtual Worlds

Personal Robotics

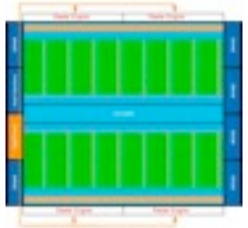
Data informatics

Pthreads
OpenMP



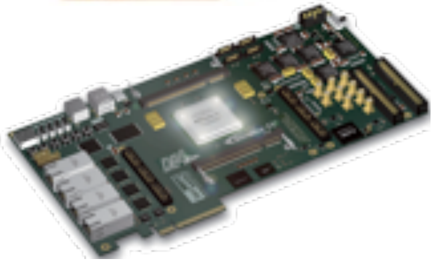
Sun
T2

CUDA
OpenCL

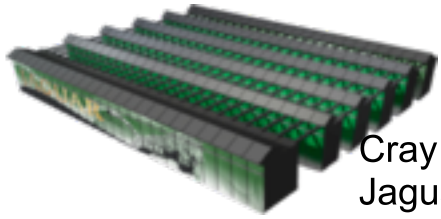


Nvidia
Fermi

Verilog
VHDL



MPI



Cray
Jaguar

Too many different programming models

Hypothesis and New Problem

Q: Is it possible to write one program and run it on all these targets?

Hypothesis and New Problem

Q: Is it possible to write one program and run it on all these targets?

HYPOTHESIS: Yes, but need domain-specific languages

THOUGH, IT'S QUITE DIFFICULT TO CREATE DSLS USING CURRENT METHODS.

Current DSL Development Approaches

Stand-alone DSLs

- Can include extensive optimizations
- Enormous effort to develop to a sufficient degree of maturity
  - Compiler, optimizations
  - Tooling (IDEs, debuggers, ...)
- Interoperation between multiple DSLs very difficult
- Examples: MATLAB, SQL

Current DSL Development Approaches

Purely embedded DSLs (“just a library”)

- Easy to develop (can reuse full host language)
- Easier to learn DSL
- Can combine multiple DSLs in one program
- Can share DSL infrastructure among several DSLs
- **Hard to optimize using domain knowledge**

We need to do better.

We need to do better.

GOAL:

Develop embedded DSLs that perform as well as stand-alone ones.

We need to do better.

GOAL:

Develop embedded DSLs that perform as well as stand-alone ones.

INTUITION: General-purpose languages should be designed with DSL embedding in mind.

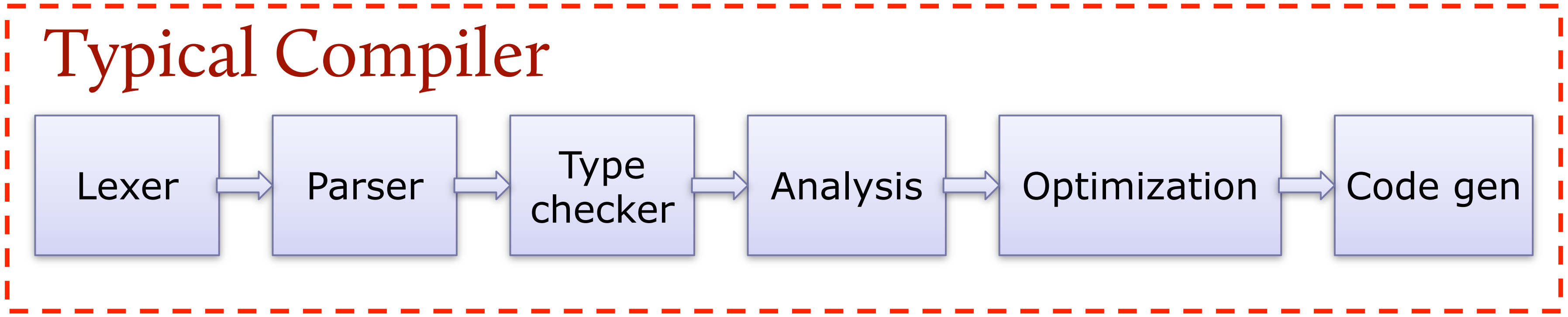
Lightweight Modular Staging.

Typical Compiler

Lexer → Parser → Type checker → Analysis → Optimization → Code gen

Lightweight Modular Staging.

Embedded DSL gets it all for free,
but can't change any of it

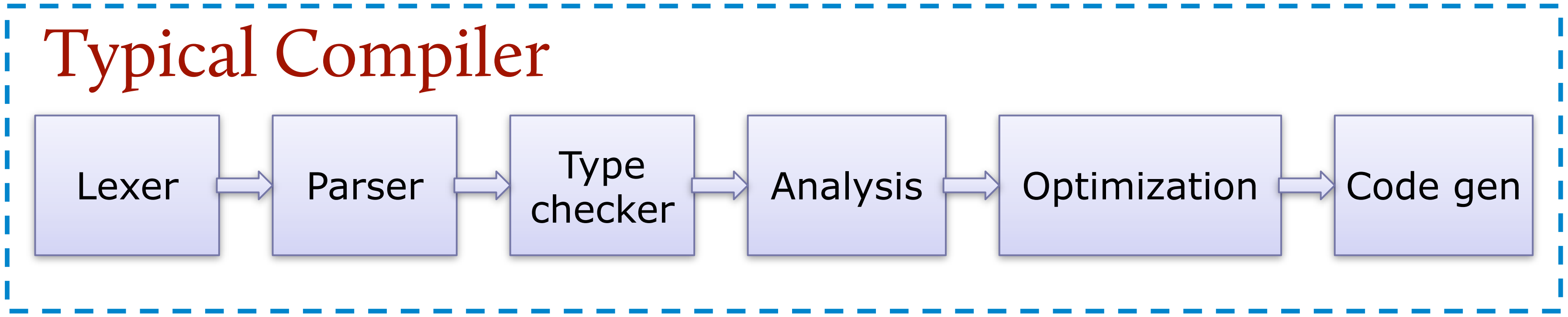


Typical Compiler

Lexer → Parser → Type checker → Analysis → Optimization → Code gen

Lightweight Modular Staging.

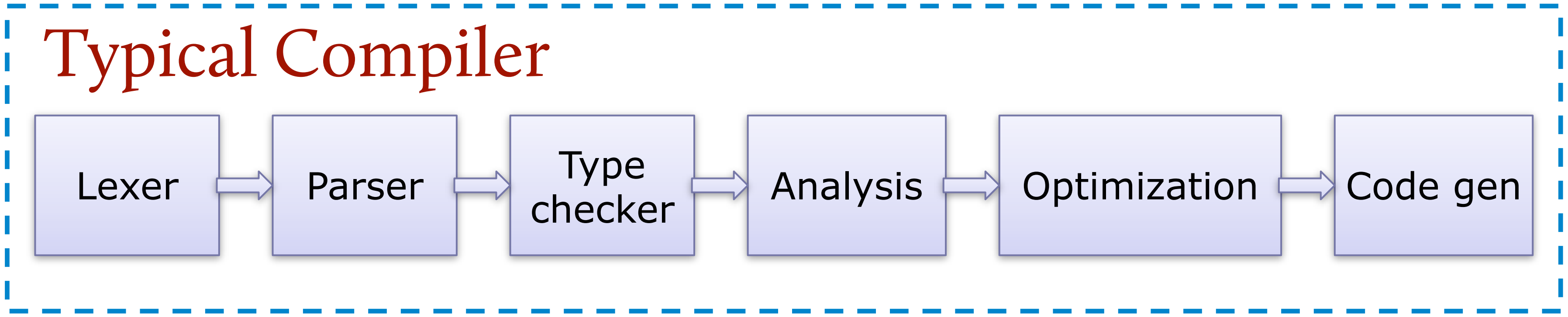
Stand-alone DSL
implements everything


Typical Compiler

Lexer → Parser → Type checker → Analysis → Optimization → Code gen

Lightweight Modular Staging.

Modular Staging provides a hybrid approach

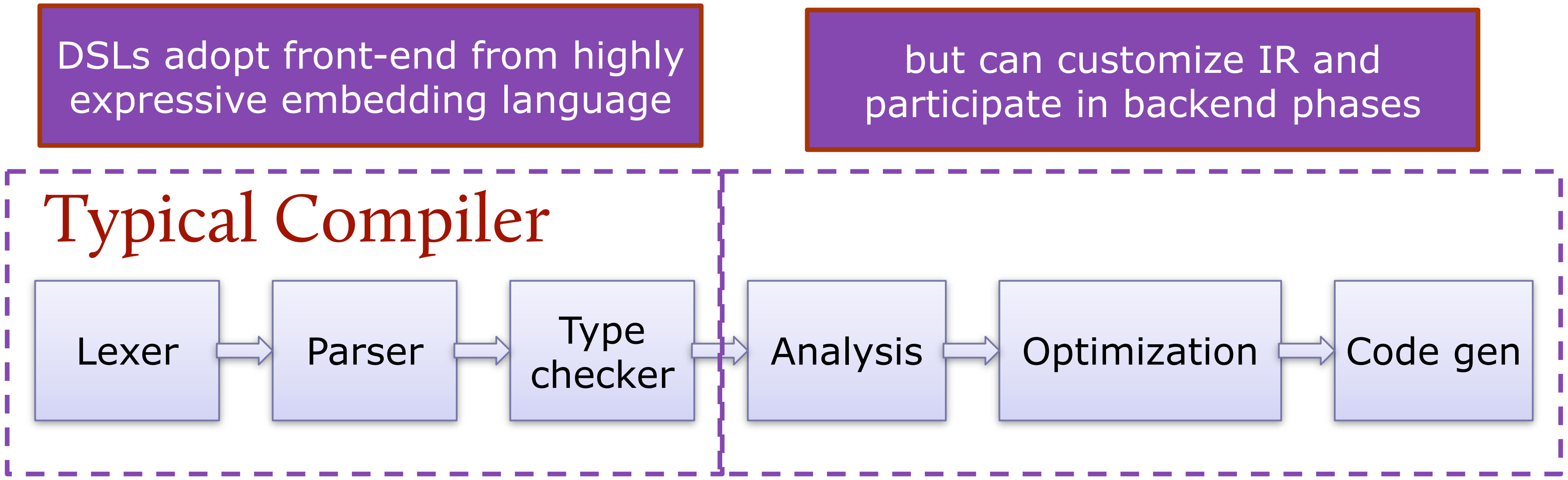


Typical Compiler

Lexer → Parser → Type checker → Analysis → Optimization → Code gen

Lightweight Modular Staging.

Modular Staging provides a hybrid approach



DSLs adopt front-end from highly expressive embedding language

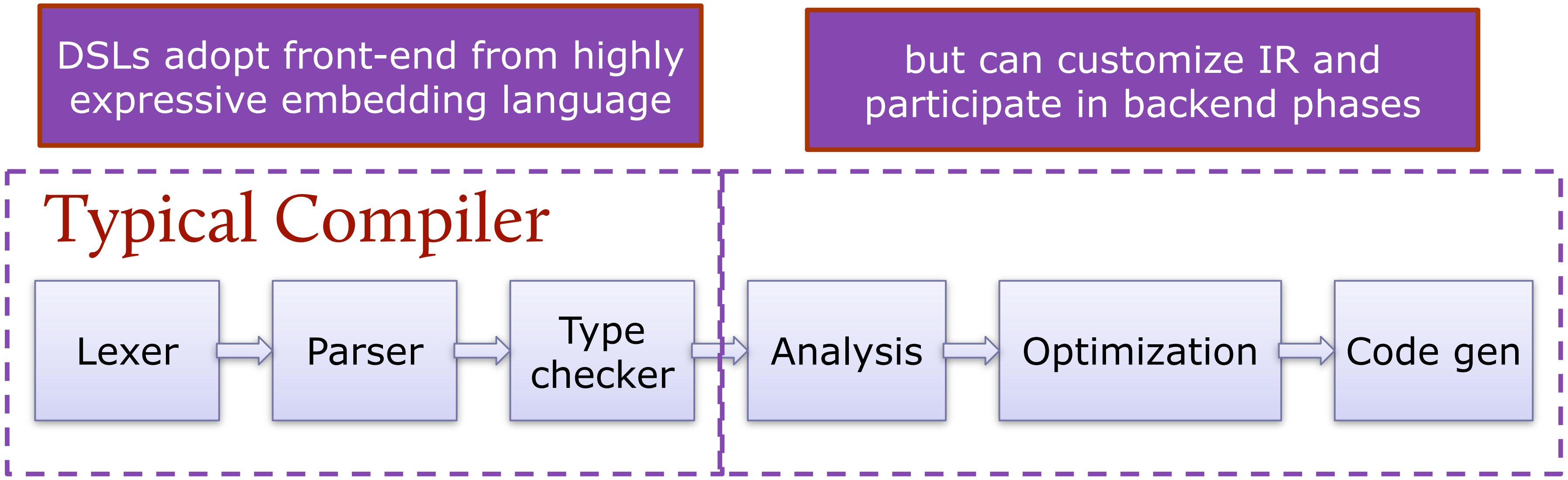
but can customize IR and participate in backend phases

Typical Compiler

Lexer → Parser → Type checker → Analysis → Optimization → Code gen

Lightweight Modular Staging.

Modular Staging provides a hybrid approach


DSLs adopt front-end from highly expressive embedding language

but can customize IR and participate in backend phases

Typical Compiler

Lexer → Parser → Type checker → Analysis → Optimization → Code gen

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

Linear Algebra Example.

```
object TestMatrix {

  def example(a: Matrix, b: Matrix, c: Matrix, d: Matrix) = {
    val x = a*b + a*c
    val y = a*c + a*d
    println(x+y)
  }
}
```

Targeting heterogeneous HW requires changing

- how data is represented
- how operations are implemented

Abstracting Matrices

Use abstract type constructor

- Do not fix a specific implementation, yet
- Operations work on abstract matrices

```
  type Rep[T]

  def infix_+(x: Rep[Matrix], y: Rep[Matrix]): Rep[Matrix]

  def example(a: Rep[Matrix], b: Rep[Matrix], c: Rep[Matrix],
d: Rep[Matrix]) = {
    val x = a*b + a*c
    val y = a*c + a*d
    println(x+y)
  }
```

IMPLEMENTATION DOESN’T CHANGE!

Lifting Scala Constants

Want to reuse Scala constants when operating on abstract data types:

```
val v: Rep[Vector[Double]]
v * 2
```

Possible approach: `v * intConst(2)`

- where `def intConst(x: Int): Rep[Int]`
- adds noise
- would be required also for more complex constants

Lifting Scala Constants

Want to reuse Scala constants when operating on abstract data types:

```
val v: Rep[Vector[Double]]
v * 2
```

Possible approach: `v * intConst(2)`

- where `def intConst(x: Int): Rep[Int]`
- adds noise
- would be required also for more complex constants

Demands parameters of type `Rep[Vector[Int]]` and `Rep[Int]`!

Lifting Scala Constants.

- **OUR APPROACH:** introduce:

  `implicit def intToRep(x: Int): Rep[Int]`
- Implicitly applied by compiler if `Rep[Int]` required, but `Int` found, and `intToRep` in scope
- No syntactic noise added to user programs
- Works not only for primitives

Staging.

Programming using only `Rep[Matrix]`, `Rep[Vector]` etc. allows different implementations for `Rep`

EXAMPLE: expression trees

```
abstract class Exp[T]
case class Const[T](x: T) extends Exp[T]
case class Symbol[T](id: Int) extends Exp[T]
abstract class Op[T]
```

Matrix implementation:

```
type Rep[T] = Exp[T]

def infix_+(x: Exp[Matrix], y: Exp[Matrix]) =
    new PlusOp(x, y)

class PlusOp(x: Exp[Matrix], y: Exp[Matrix])
      extends DeliteOpZip[Matrix]
```

Staging.

Programming using only `Rep[Matrix]`, `Rep[Vector]` etc. allows different implementations for `Rep`

EXAMPLE: expression trees

```
abstract class Exp[T]
case class Const[T](x: T) extends Exp[T]
case class Symbol[T](id: Int) extends Exp[T]
abstract class Op[T]
```

Matrix implementation:

```
type Rep[T] = Exp[T]

def infix_+(x: Exp[Matrix], y: Exp[Matrix]) =
    new PlusOp(x, y)

class PlusOp(x: Exp[Matrix], y: Exp[Matrix])
      extends DeliteOpZip[Matrix]
```

The Delite DSL Framework

- Provides IR with parallel execution patterns

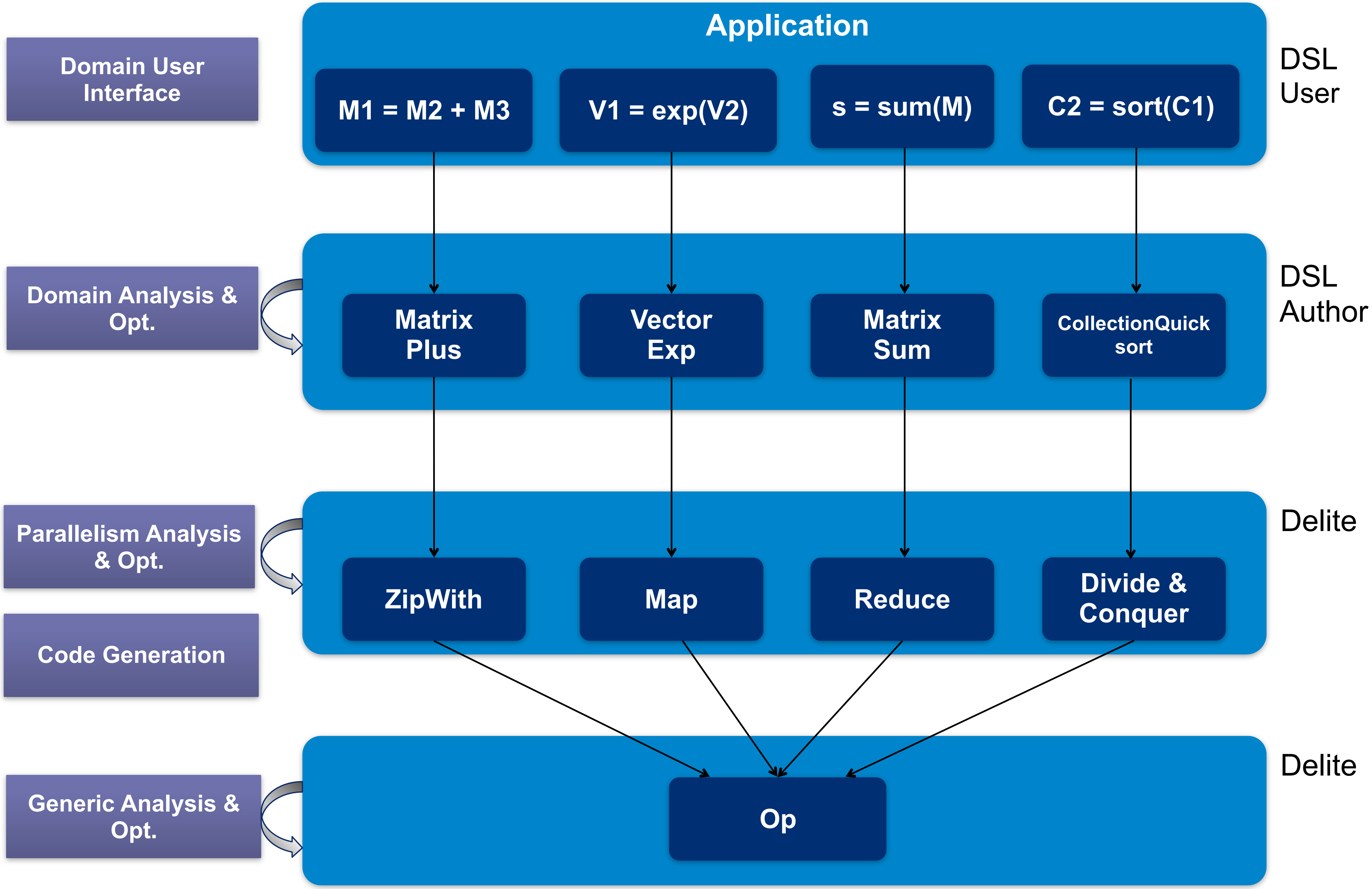
  EXAMPLE: `DeliteOpZip[T]`

- Parallel optimization of IR graph
- Compiler framework with support for heterogeneous hardware platforms
- DSL extends parallel operations

  EXAMPLE: `class Plus extends DeliteOpZip[Matrix]`

- Domain-specific analysis and optimization

The Delite IR Hierarchy



Domain User Interface

Application

M1 = M2 + M3

V1 = exp(V2)

s = sum(M)

C2 = sort(C1)

DSL User

Domain Analysis & Opt.

Matrix Plus

Vector Exp

Matrix Sum

CollectionQuicksort

DSL Author

Parallelism Analysis & Opt.

Code Generation

ZipWith

Map

Reduce

Divide & Conquer

Delite

Generic Analysis & Opt.

Op

Delite

Delite Ops

- Encode parallel execution patterns
  - **EXAMPLE:** data-parallel: map, reduce, zip, ...
- Delite provides implementations of these patterns for multiple hardware targets
  - **EXAMPLE:** multicore, GPU
- DSL developer maps each domain operation to the appropriate pattern

Optimization: Loop Fusion

Reduces loop overhead and improves locality

- Elimination of temporary data structures
- Communication through registers

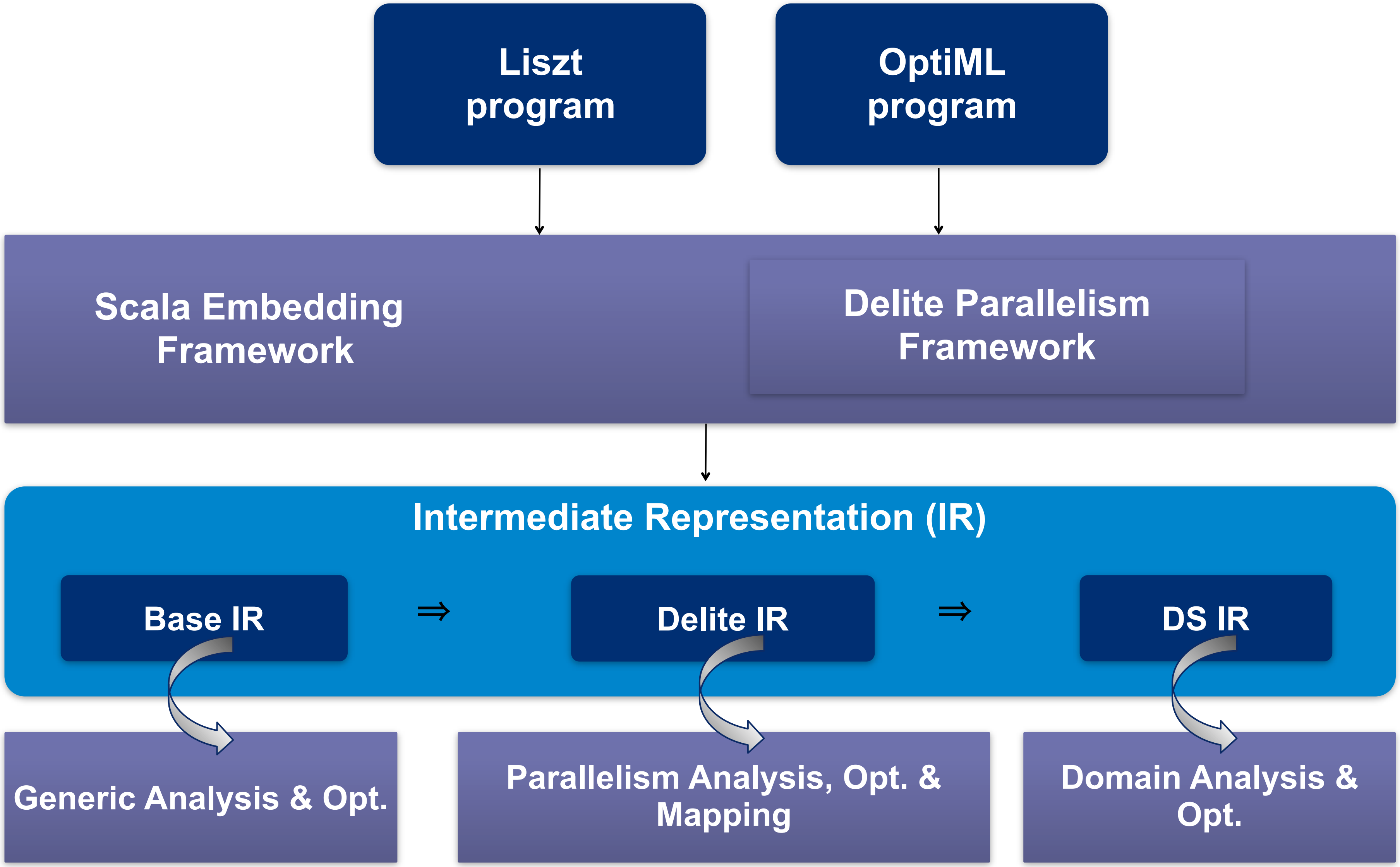
Fuse both dependent and side-by-side operations

- Fused operations can have multiple outputs

ALGORITHM: Fuse two loops if,

- `size(loop1) == size(loop2)`
- No dependencies exist that would require an impossible schedule when fused (e.g., C depends on B depends on A => cannot fuse C and A)

Delite DSL Compilers.



Liszt program

OptiML program

Scala Embedding Framework

Delite Parallelism Framework

Intermediate Representation (IR)

Base IR ⇒ Delite IR ⇒ DS IR

Generic Analysis & Opt.

Parallelism Analysis, Opt. & Mapping

Domain Analysis & Opt.

Delite: Conclusions.

- Need to simplify the process of developing DSLs for parallelism.
- Need programming languages to be designed for flexible embedding.
- Lightweight modular staging allows for powerful embedded DSLs.
- Delite provides a framework for adding parallelism.

THANK YOU.

Questions?

PhD Tips: Writing Papers

- Best help to earn you a PhD
  - But can earn PhD without a conference paper if practical contribution is worthwhile (in Europe)
- Follow Simon Peyton-Jones' advice on how to write a paper (it's motivating, too: write paper about any idea, no matter how small)

Submitting to a Conference

- Paper(s) accepted at conferences (as opposed to workshops) are best way to ensure you graduate soon
  - Acceptance at big conference (PLDI, POPL, OOPSLA, ECOOP) will earn PhD without any doubt (if you and advisor are authors)
  - But, second tier conference fine places to publish papers, too: actors paper with most impact appeared at a second class conference
  - Use deadlines to drive work (to some extent)

Doing Research

- Worst mistake: **Not** spending **1 hour per day** thinking really hard about your most important problem
    - Without **any** distractions
    - While working on an implementation, hard to make room for 1 hour **not at the computer**
    - Important to have deep thinking time that is not required to produce immediate result
- Balance between reading papers and thinking/programming yourself
    - Reading **the right papers** carefully most important