Arc

An MLIR Dialect for Data Analytics

PLDS' 20 Workshop
March 5, 2020

Presented by **Klas Segeljakt** <klasseg@kth.se>

Joint work with
Frej Drejhammar <frej.drejhammar@ri.se>
Khoa Dinh <khoad@kth.se>

- Outline
    - **Introduction & Problem**
    - **Related Work**
    - **Arc & MLIR**
    - **Summary**

Introduction & Problem

Introduction & Problem

Introduction & Problem

[Q1]: What does a programming language for **data analytics** definitely need?

Introduction & Problem

[Q1]: What does a programming language for **data analytics** definitely need?
[A1]: Data types

Introduction & Problem

[Q1]: What does a programming language for **data analytics** definitely need?

[A1]: Data types

[Q2]: Which **data types**?

Introduction & Problem

[Q1]: What does a programming language for **data analytics** definitely need?
[A1]: Data types

[Q2]: Which **data types**?
[A2]: Tables, **Arrays, Streams, Graphs, …**

Introduction & Problem

[Q1]: What does a programming language for **data analytics** definitely need?
[A1]: Data types

[Q2]: Which **data types**?
[A2]: Tables, **Arrays, Streams, Graphs, …**

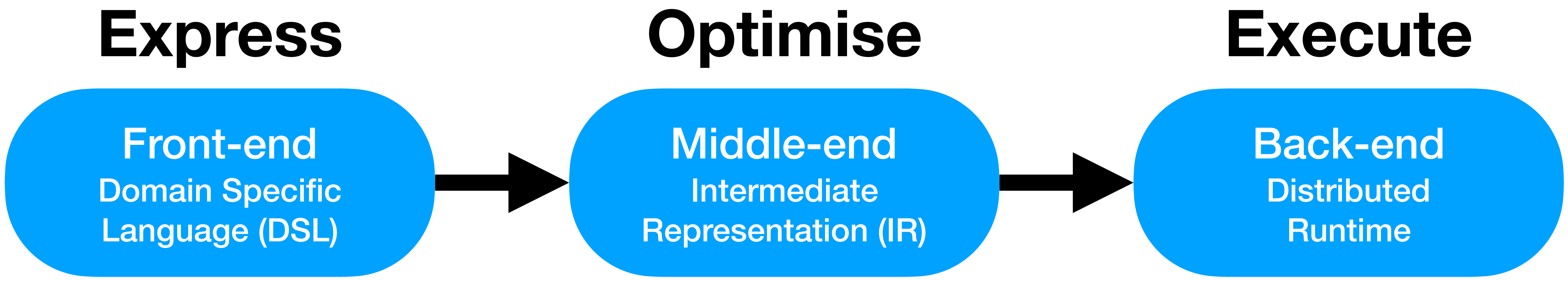
[Q3]: Can **[A1]** be "combined" under one **system** and **programming model**?

Introduction & Problem

Introduction & Problem

Express **Optimise** **Execute**

Introduction & Problem



Introduction & Problem

Express

Front-end
Domain Specific Language (DSL)

Challenges:

Optimise

Middle-end
Intermediate Representation (IR)

Challenges:

Execute

Back-end
Distributed Runtime

Challenges:

Introduction & Problem

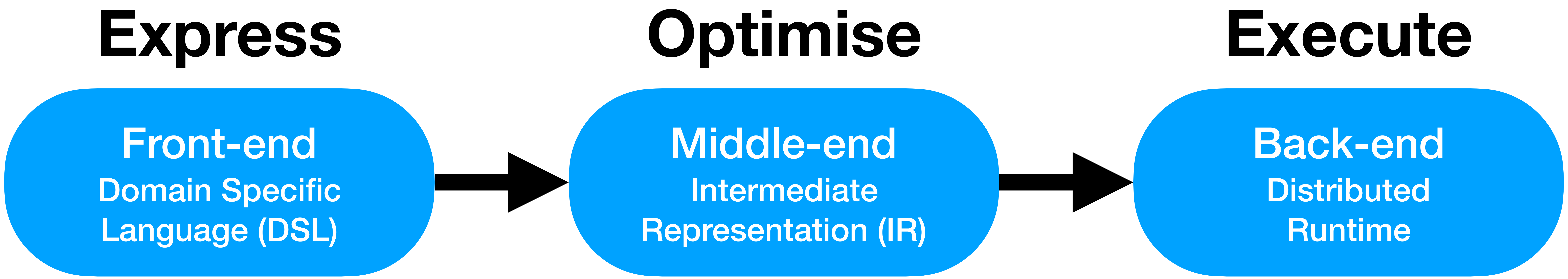
Express	Optimise	Execute
Front-end Domain Specific Language (DSL)	**Middle-end** Intermediate Representation (IR)	**Back-end** Distributed Runtime

Challenges:
- Abstraction level
- Syntax & type system
- Tooling & Integration

Challenges:

Challenges:

Introduction & Problem



Challenges:
- Abstraction level
- Syntax & type system
- Tooling & Integration

Challenges:
- Progressive lowering
- Algebraic simplification
- Instruction selection
- Cost model

Challenges:

Introduction & Problem

Express		Optimise		Execute
Front-end Domain Specific Language (DSL)	→	**Middle-end** Intermediate Representation (IR)	→	**Back-end** Distributed Runtime

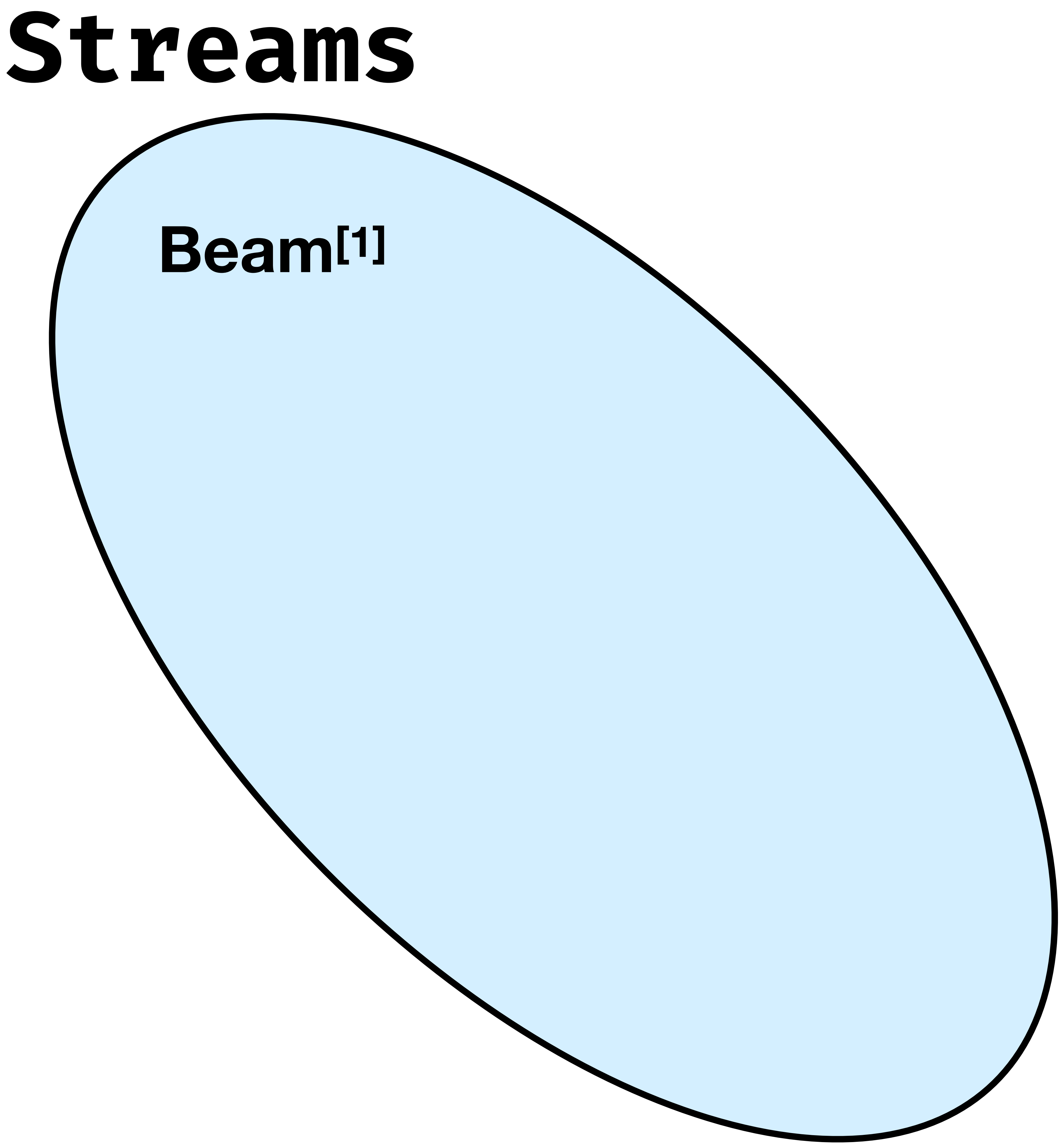
Challenges:
- Abstraction level
- Syntax & type system
- Tooling & Integration

Challenges:
- Progressive lowering
- Algebraic simplification
- Instruction selection
- Cost model

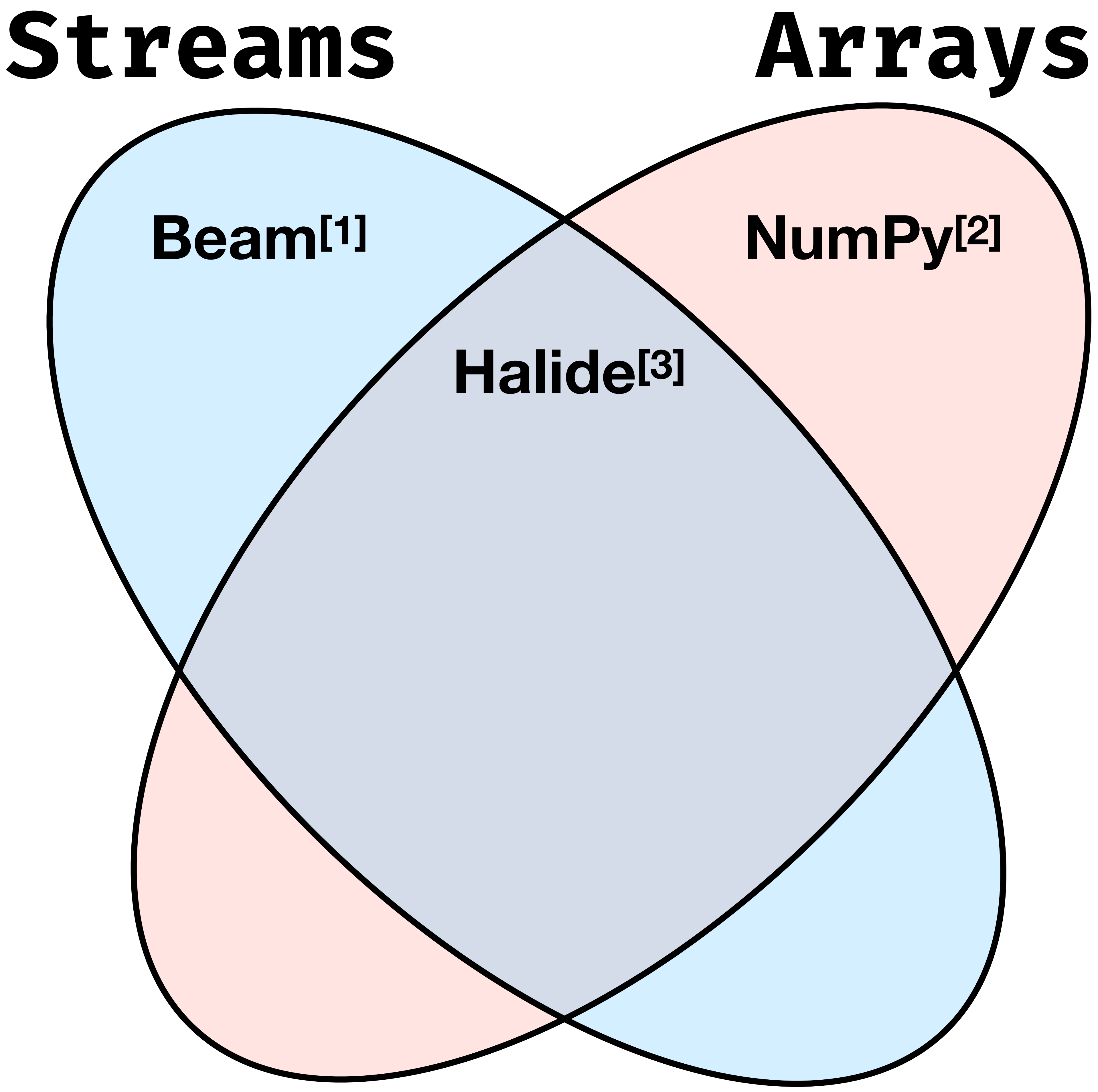
Challenges:
- State management
- Scheduling
- Failure recovery
- Profiling & Debugging

Related Work

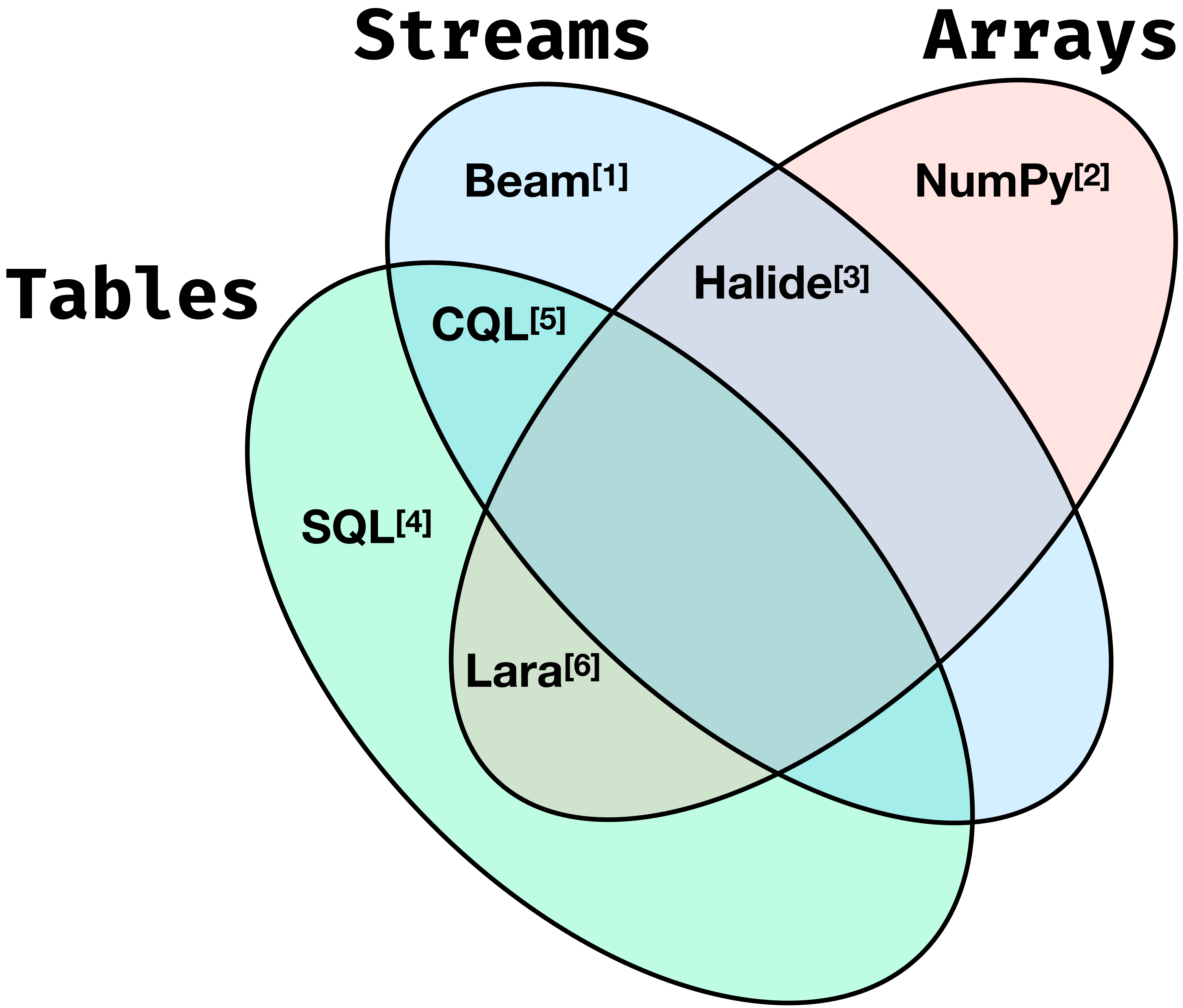
Related Work



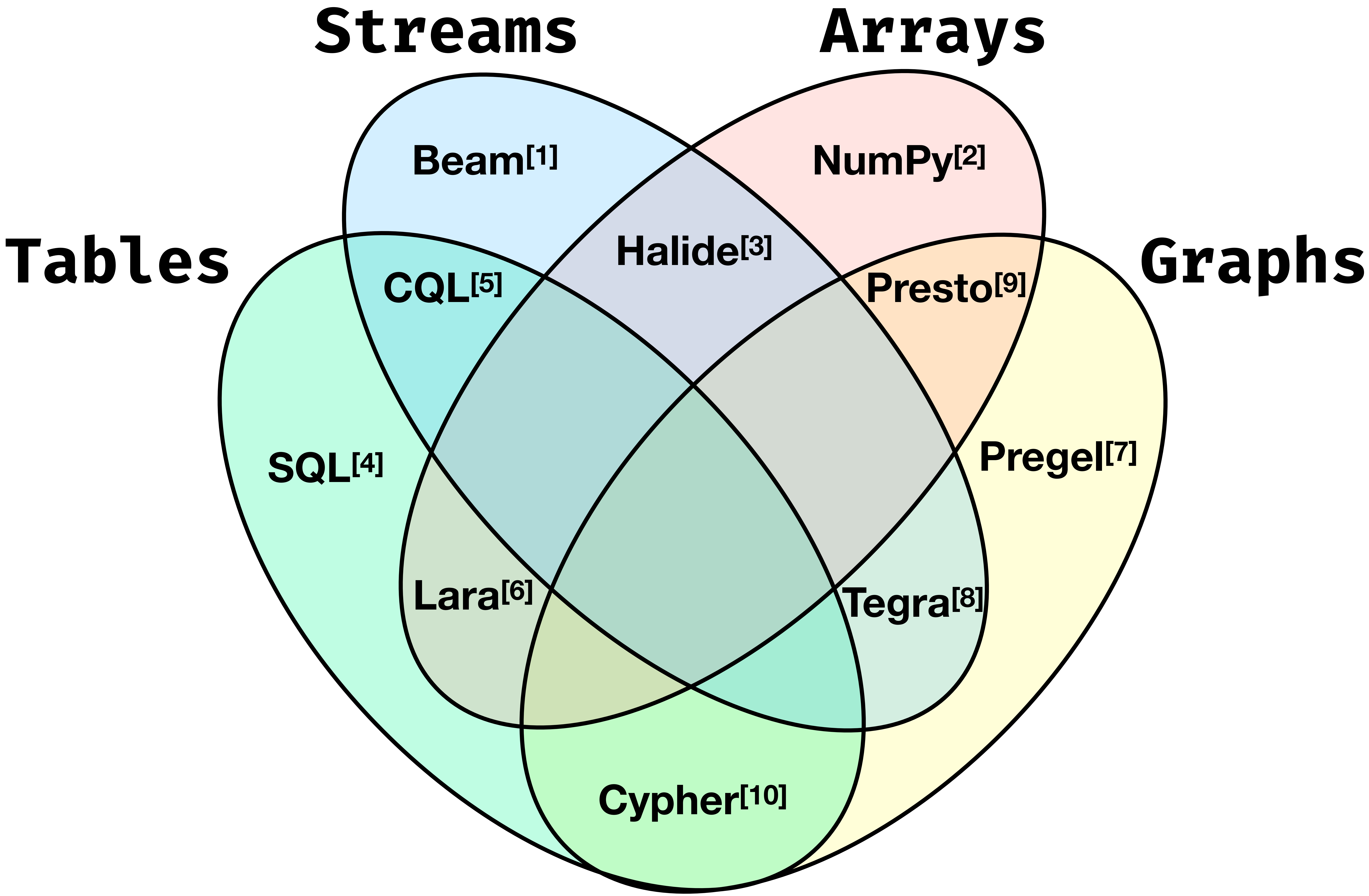
Related Work



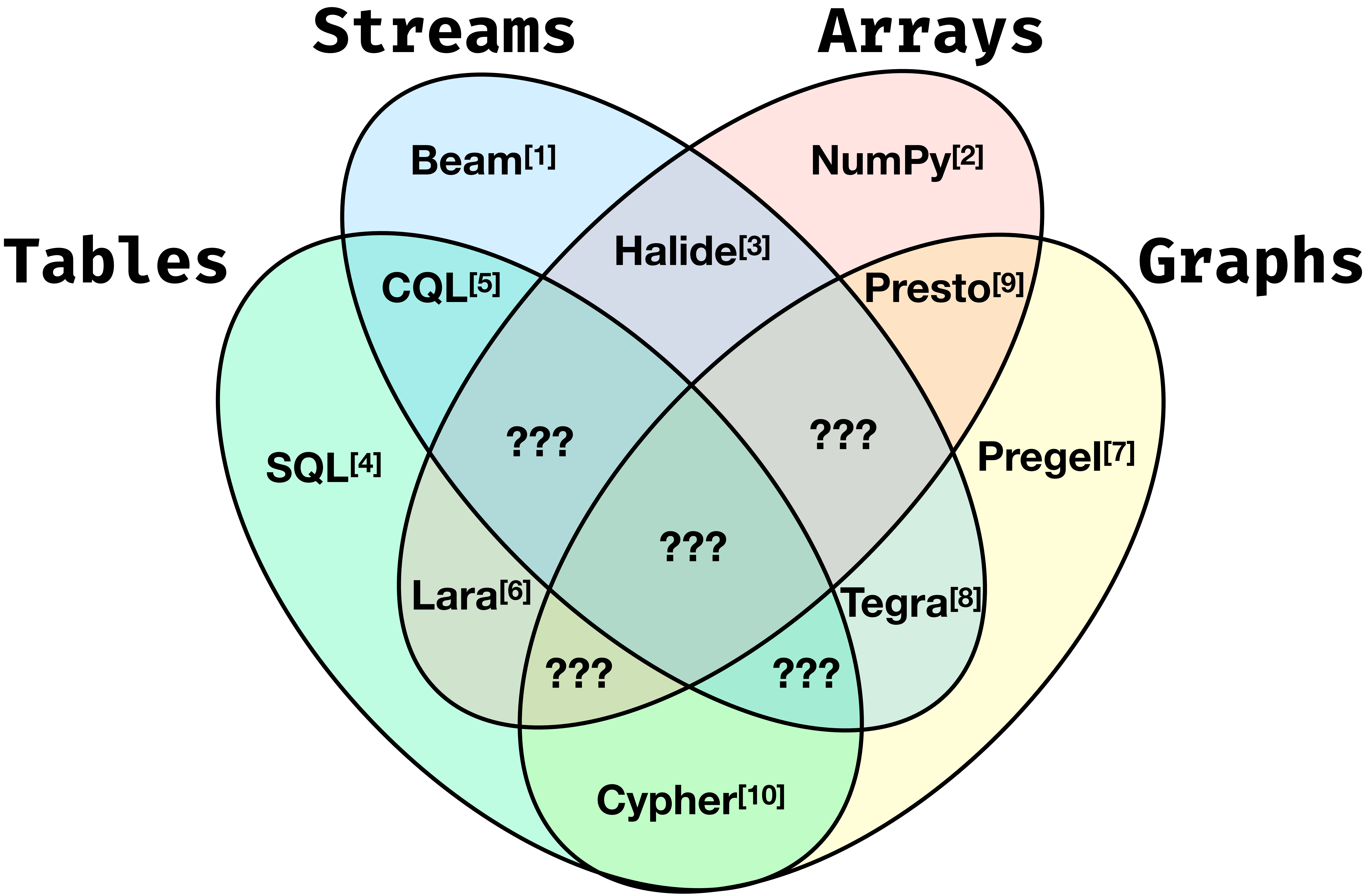
Related Work



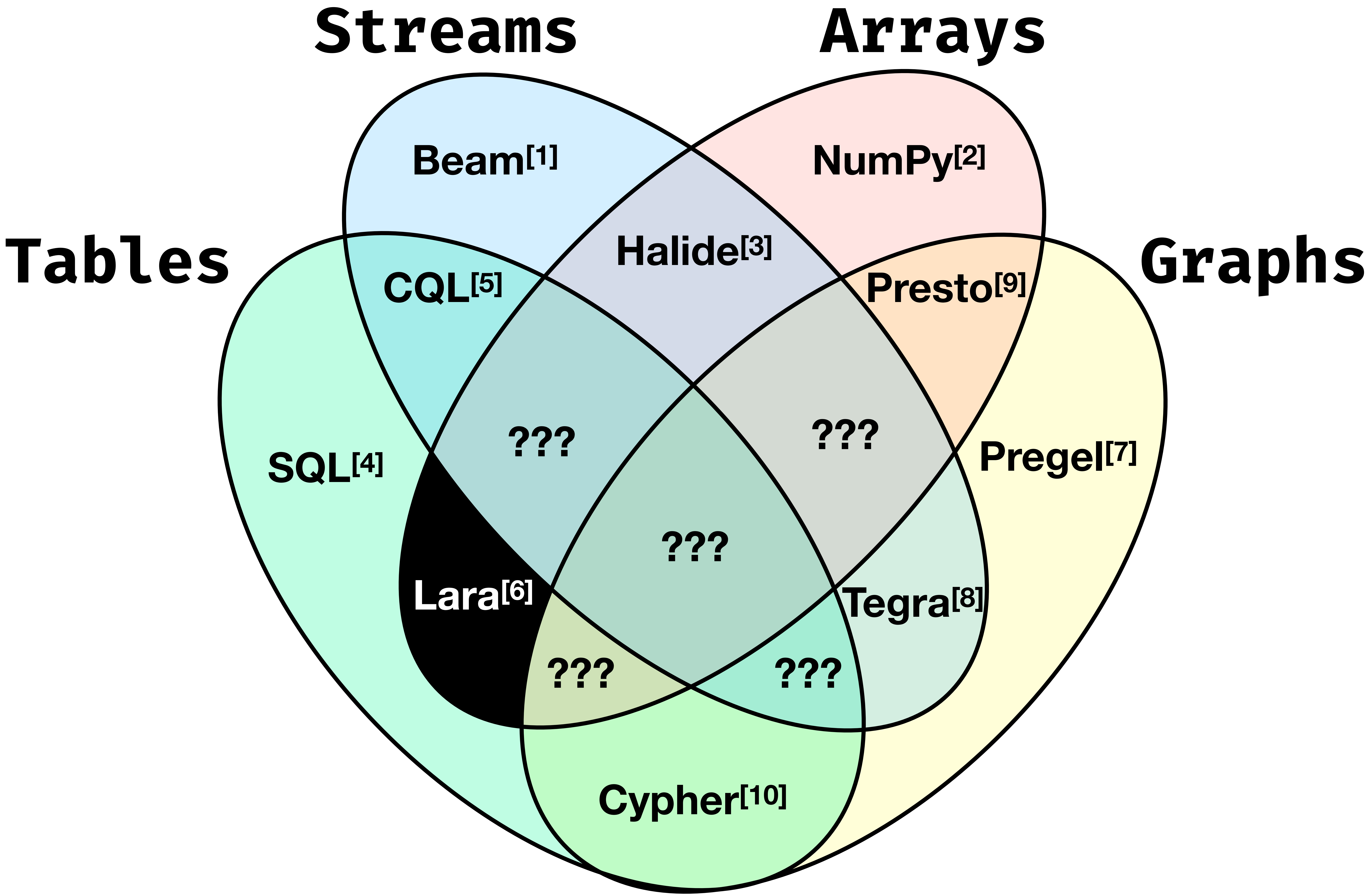
Related Work



Related Work



Related Work



What is Lara?

What is Lara?

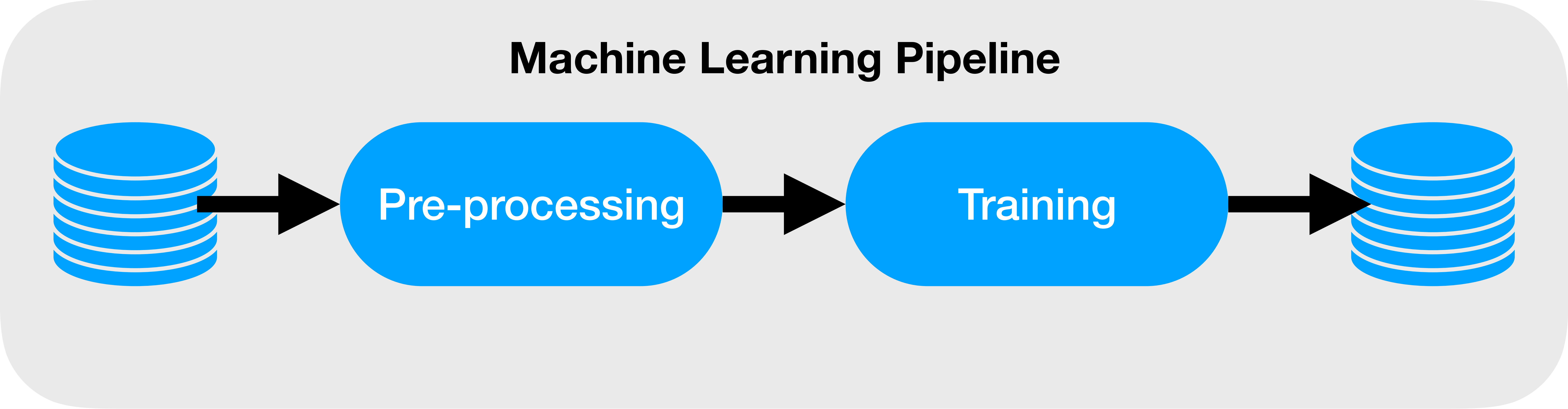
An Intermediate Representation for Optimizing Machine Learning Pipelines

Andreas Kunft[*] Asterios Katsifodimos[**] Sebastian Schelter[†]
Sebastian Breß[‡*] Tilmann Rabl[+] Volker Markl[‡*]

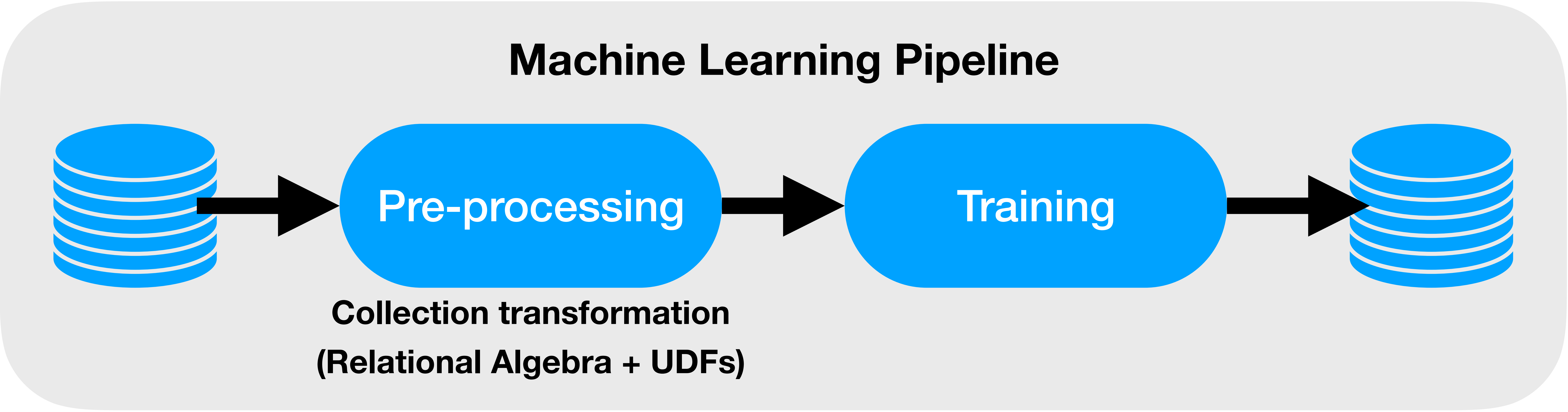
[*]TU Berlin [**]Delft University of Technology [†]New York University [‡]DFKI [+]HPI, Universität Potsdam

Where did Lara come from?

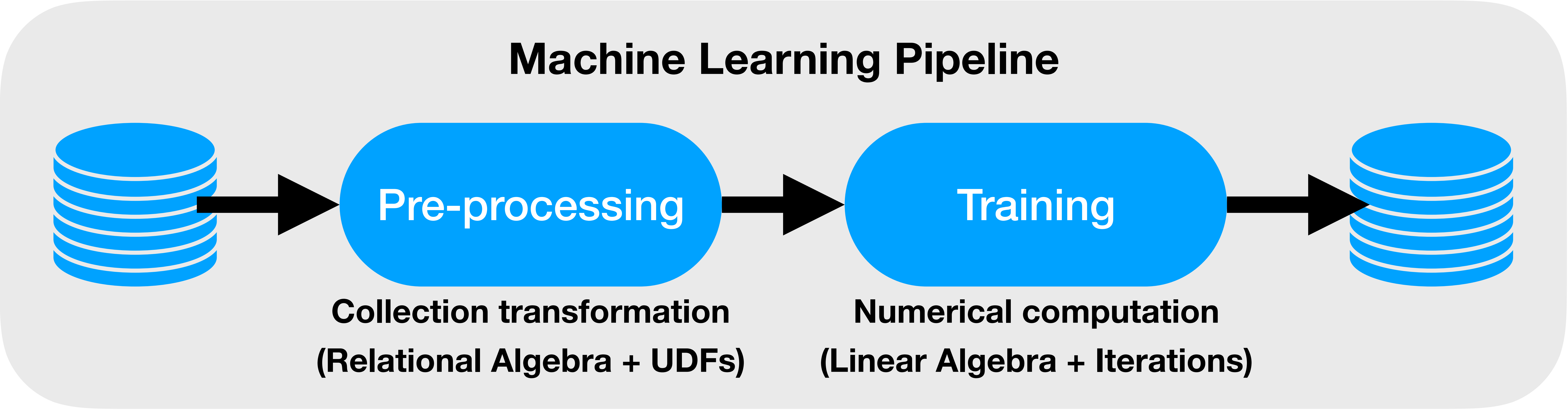
Where did Lara come from?



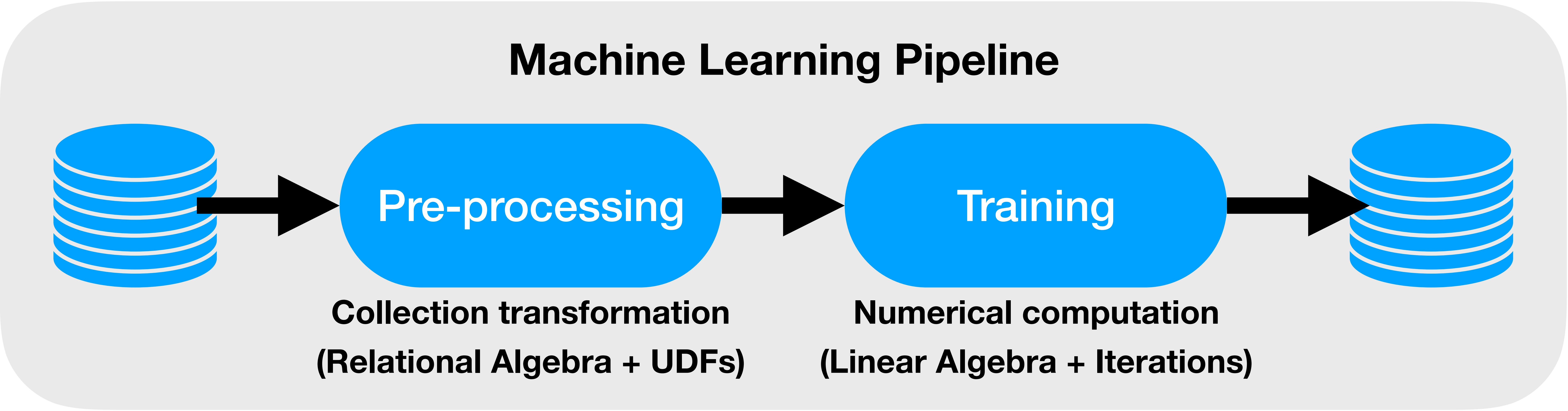
Where did Lara come from?



Where did Lara come from?

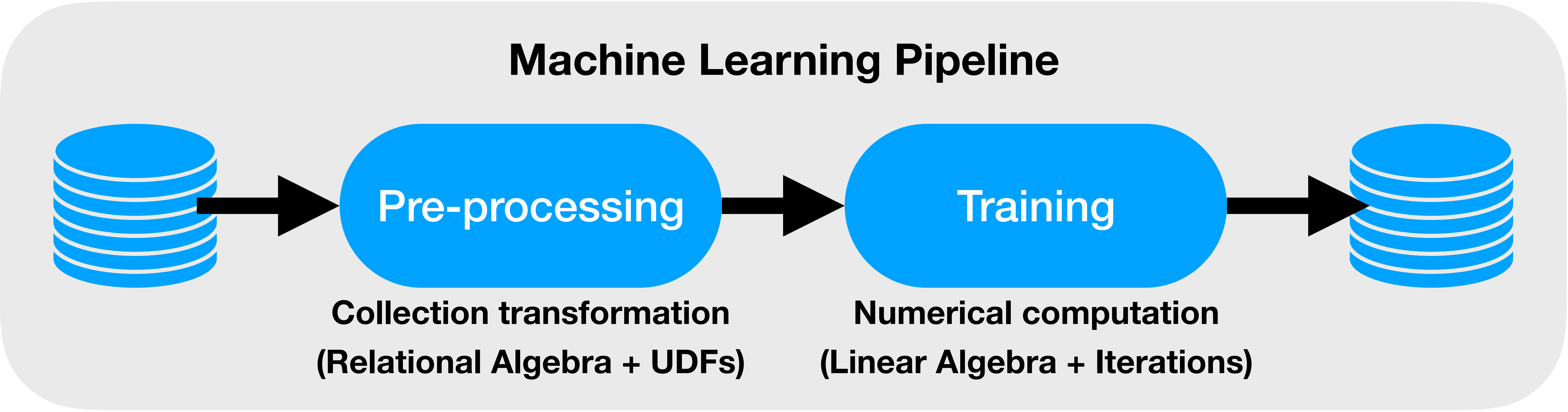


Where did Lara come from?



Problem? Current systems used for ML are tailored to either **processing** (Spark, Flink, etc.) or **training** (TensorFlow, SystemML, etc.)

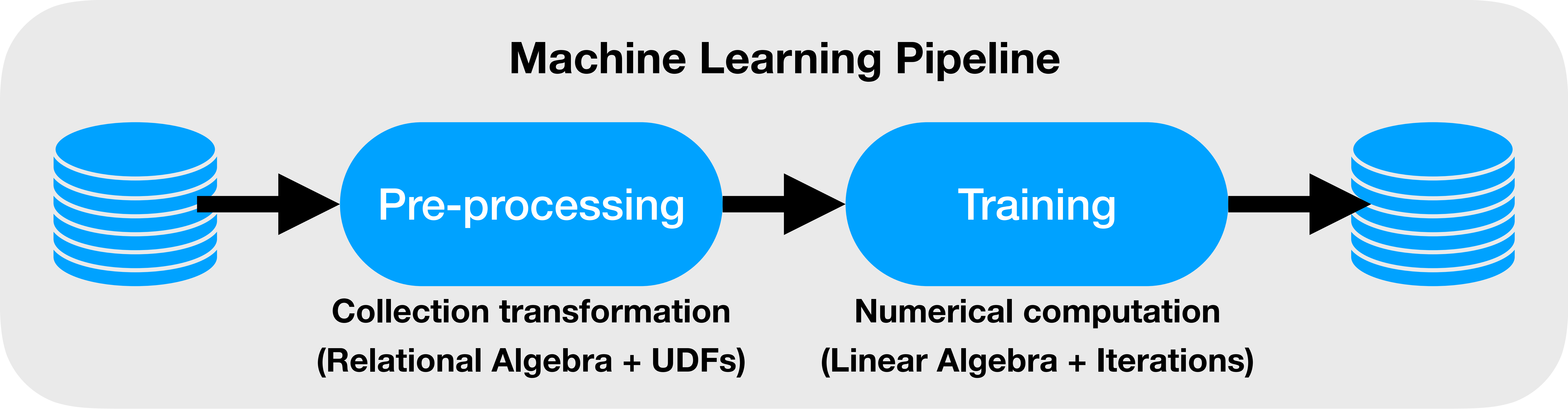
Where did Lara come from?



Problem? Current systems used for ML are tailored to either **processing** (Spark, Flink, etc.) or **training** (TensorFlow, SystemML, etc.)

➡No holistic optimisation

Where did Lara come from?



Problem? Current systems used for ML are tailored to either **processing** (Spark, Flink, etc.) or **training** (TensorFlow, SystemML, etc.)

➡No holistic optimisation

Solution? Lara: A language for **L**inear **A**lgebra and **R**elational **A**lgebra

How does Lara work?

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

- `DataBag A ::= Multiset A`
- `Vector A  ::= Set (ℕ, A)`
- `Matrix A  ::= Set ((ℕ, ℕ), A)`

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):
- **`DataBag`** `A ::=` **`Multiset`** `A`
- **`Vector`** `A ::=` **`Set`** `(ℕ, A)`
- **`Matrix`** `A ::=` **`Set`** `((ℕ, ℕ), A)`

Operations:
- For-comprehensions
- Linear algebra functions

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

- `DataBag A ::= Multiset A`
- `Vector A  ::= Set (ℕ, A)`
- `Matrix A  ::= Set ((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- `DataBag (ℕ, Vector A) → Matrix A`
- `DataBag A → (A → (ℕ, ℕ)) → Matrix A`

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

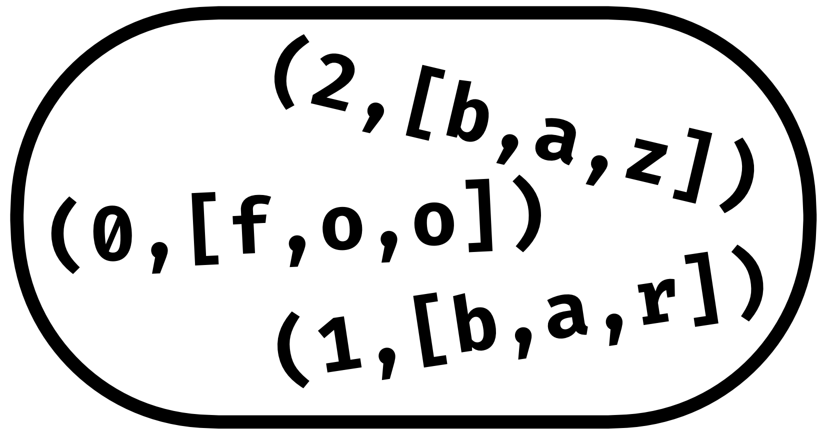
- **DataBag** A ::= **Multiset** A
- **Vector** A ::= **Set** (ℕ, A)
- **Matrix** A ::= **Set** ((ℕ, ℕ), A)

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- **DataBag** (ℕ, **Vector** A) → **Matrix** A
- **DataBag** A → (A → (ℕ, ℕ)) → **Matrix** A



DataBag (ℕ, **Vector** A)

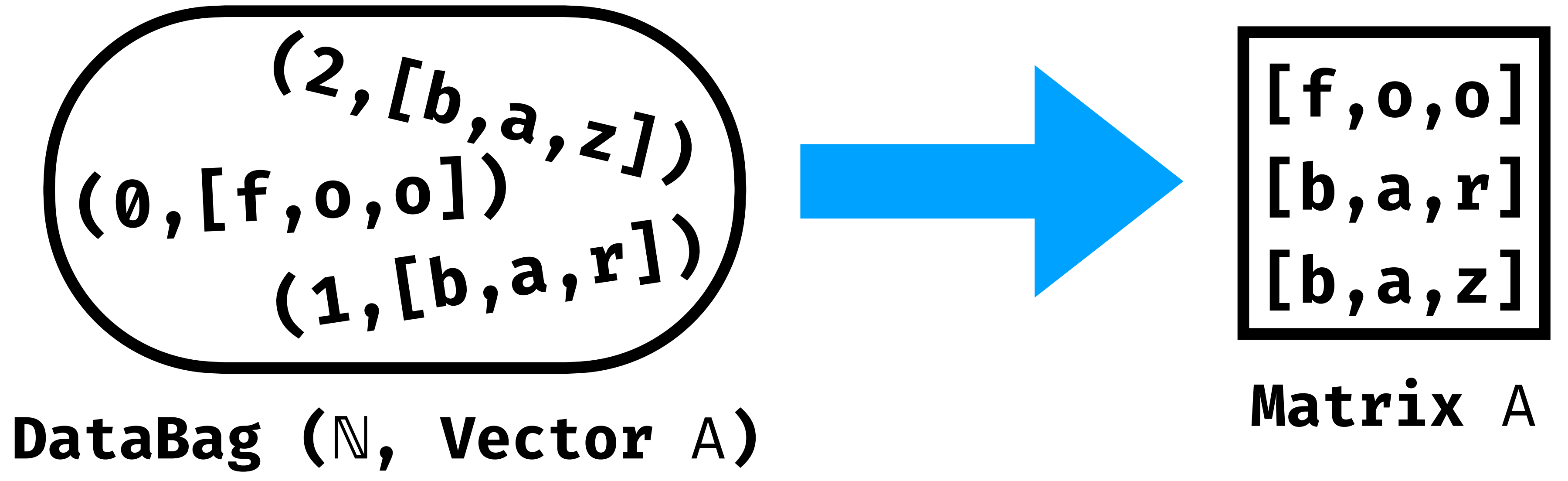
How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):
- **`DataBag`** `A ::= ` **`Multiset`** `A`
- **`Vector`** `A ::= ` **`Set`** `(ℕ, A)`
- **`Matrix`** `A ::= ` **`Set`** `((ℕ, ℕ), A)`

Operations:
- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:
- **`DataBag`** `(ℕ,` **`Vector`** `A) →` **`Matrix`** `A`
- **`DataBag`** `A → (A → (ℕ, ℕ)) →` **`Matrix`** `A`



How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

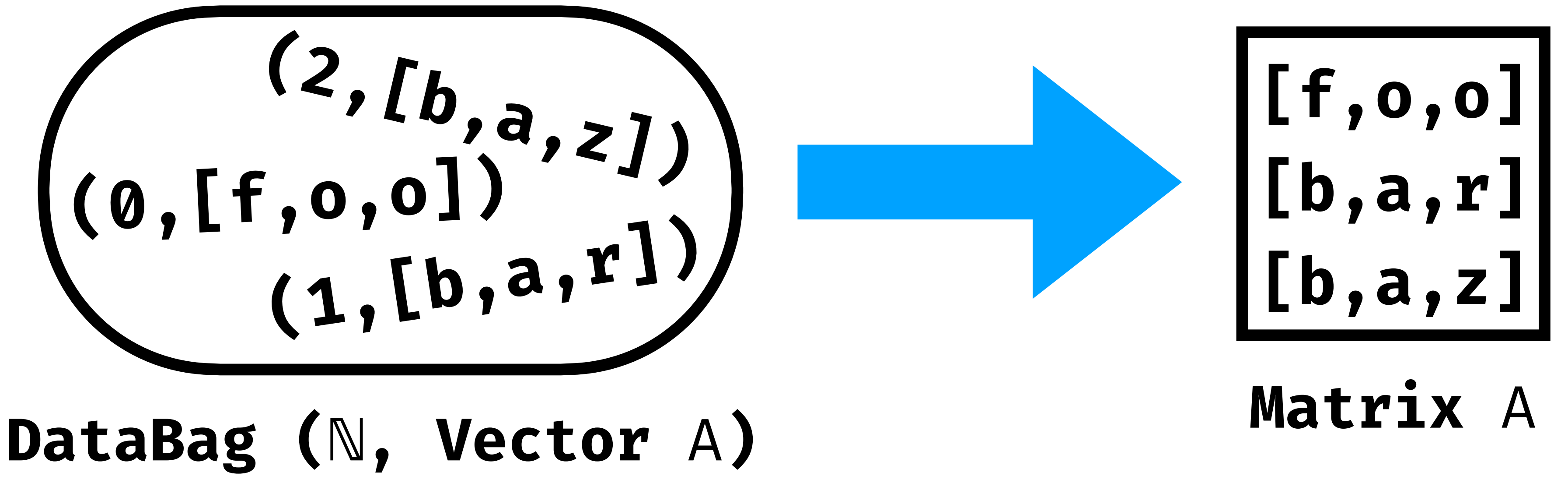
- **DataBag** A ::= **Multiset** A
- **Vector** A ::= **Set** (ℕ, A)
- **Matrix** A ::= **Set** ((ℕ, ℕ), A)

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- **DataBag** (ℕ, **Vector** A) → **Matrix** A
- **DataBag** A → (A → (ℕ, ℕ)) → **Matrix** A



Lara also has an **IR** with two **"views"**.

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

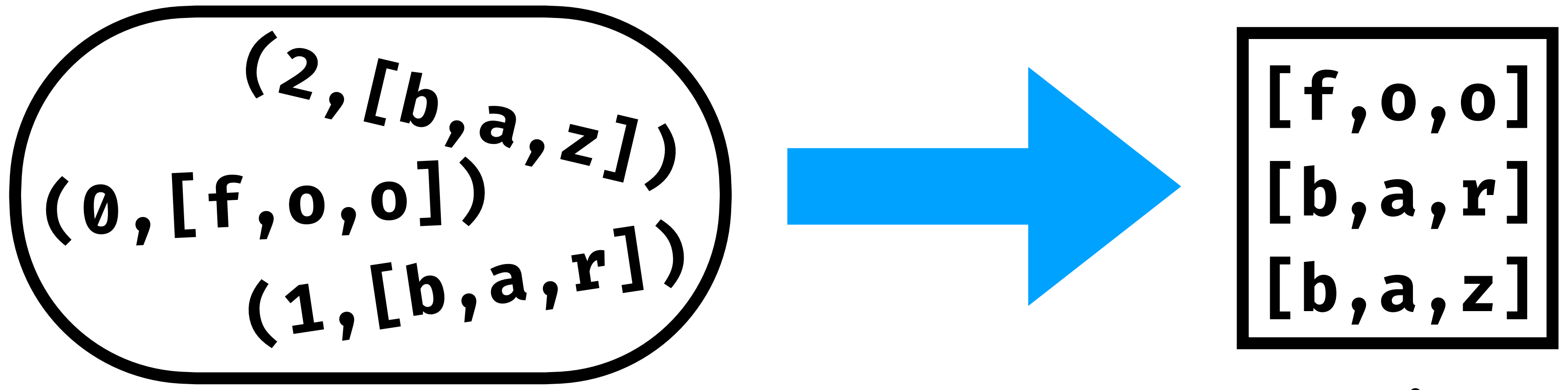
- `DataBag A ::= Multiset A`
- `Vector A  ::= Set (ℕ, A)`
- `Matrix A  ::= Set ((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- `DataBag (ℕ, Vector A) → Matrix A`
- `DataBag A → (A → (ℕ, ℕ)) → Matrix A`



`DataBag (ℕ, Vector A)`   `Matrix A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

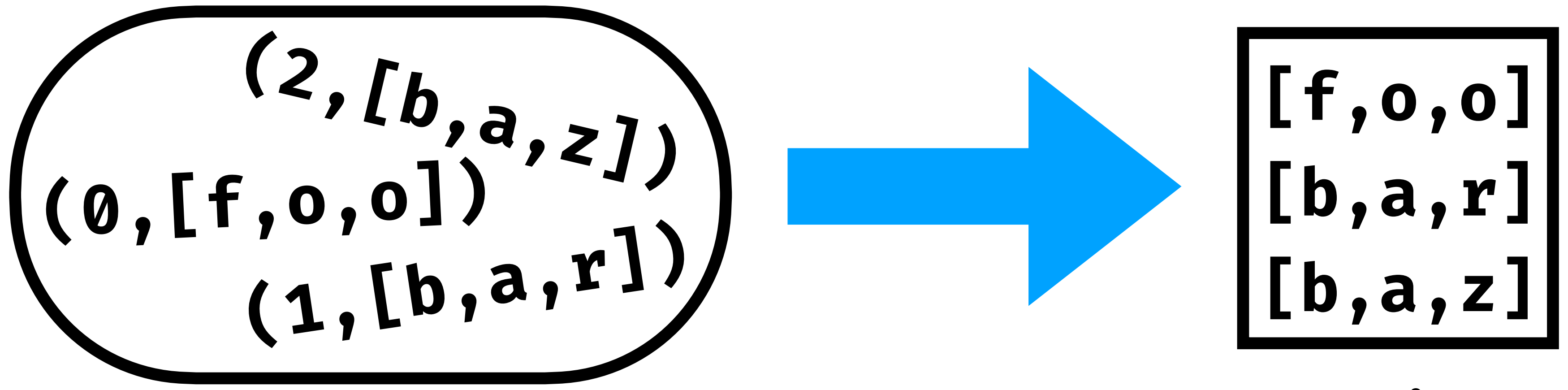
- `DataBag A ::= Multiset A`
- `Vector A  ::= Set (ℕ, A)`
- `Matrix A  ::= Set ((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- `DataBag (ℕ, Vector A) → Matrix A`
- `DataBag A → (A → (ℕ, ℕ)) → Matrix A`



`DataBag (ℕ, Vector A)`

`Matrix A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

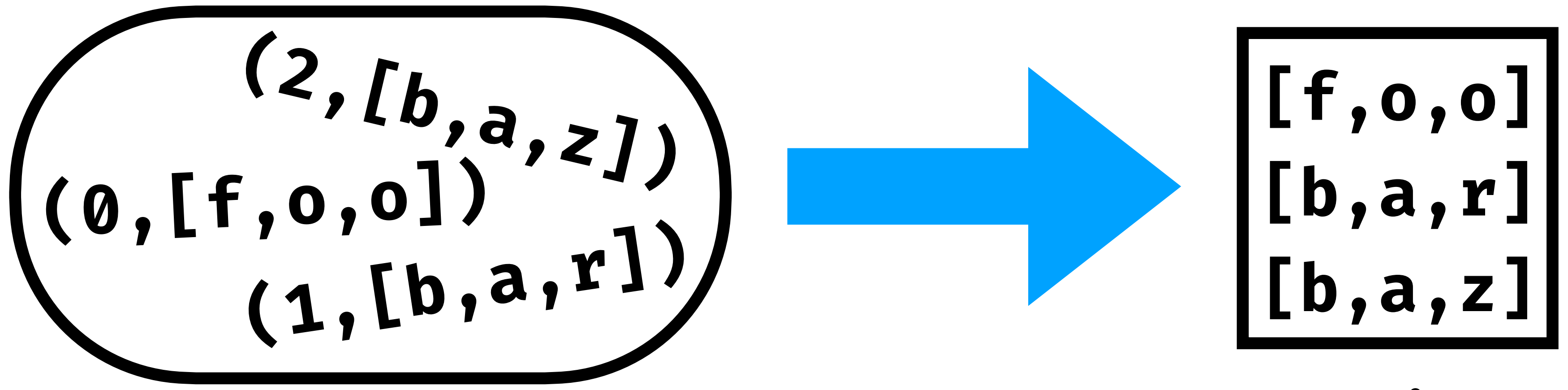
- `DataBag A ::= Multiset A`
- `Vector A ::= Set (ℕ, A)`
- `Matrix A ::= Set ((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- `DataBag (ℕ, Vector A) → Matrix A`
- `DataBag A → (A → (ℕ, ℕ)) → Matrix A`



`DataBag (ℕ, Vector A)`

`Matrix A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$$\mathbf{x} + \mathbf{y}$$

Element-wise
vector addition

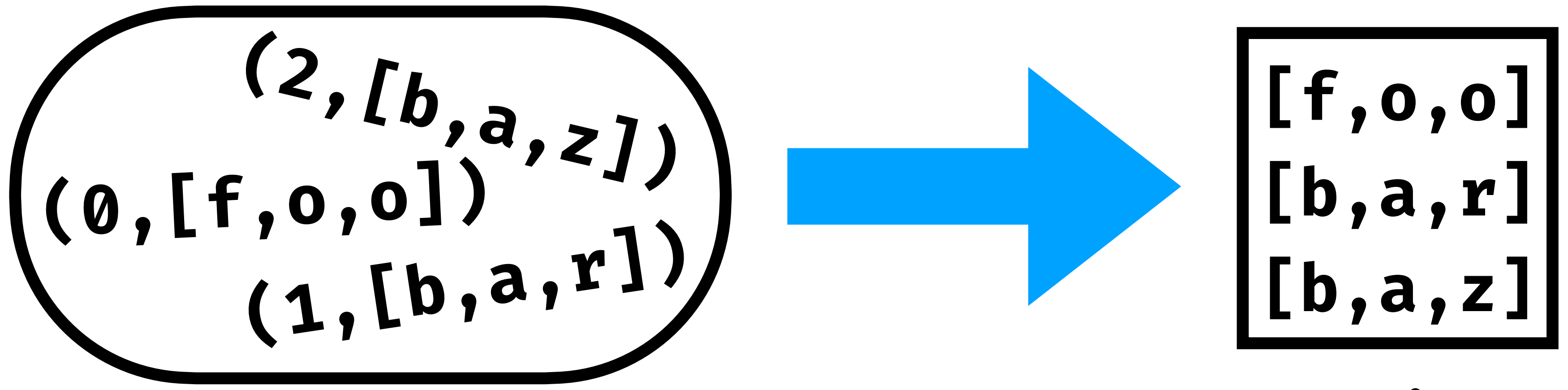
How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):
- `DataBag A ::= Multiset A`
- `Vector A  ::= Set (ℕ, A)`
- `Matrix A  ::= Set ((ℕ, ℕ), A)`

Operations:
- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:
- `DataBag (ℕ, Vector A) → Matrix A`
- `DataBag A → (A → (ℕ, ℕ)) → Matrix A`



DataBag (ℕ, Vector A)

Matrix A

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$\mathbf{x} + \mathbf{y}$

Element-wise vector addition

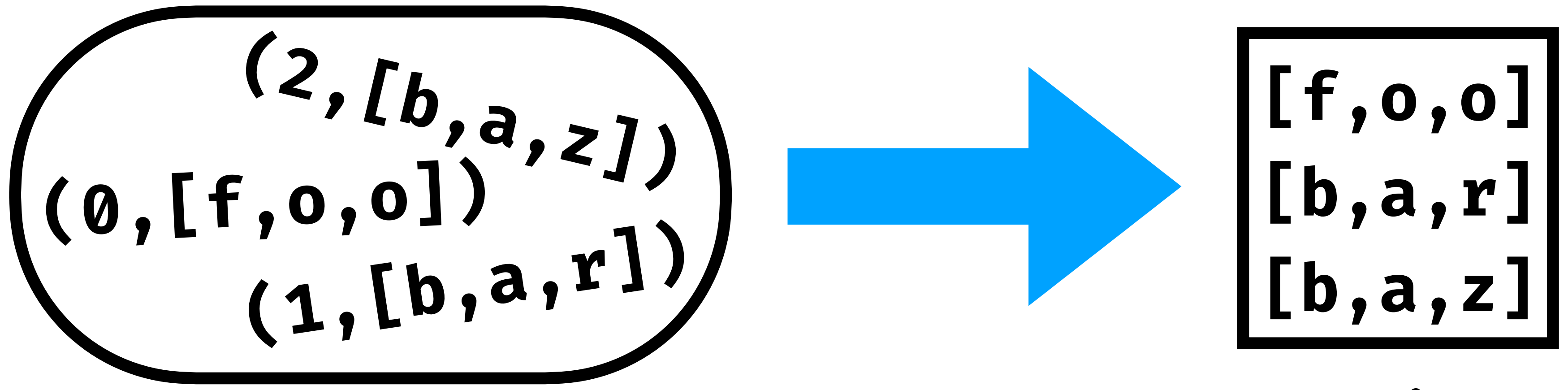
How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):
- `DataBag A ::= Multiset A`
- `Vector A ::= Set (ℕ, A)`
- `Matrix A ::= Set ((ℕ, ℕ), A)`

Operations:
- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:
- `DataBag (ℕ, Vector A) → Matrix A`
- `DataBag A → (A → (ℕ, ℕ)) → Matrix A`



`DataBag (ℕ, Vector A)`

`Matrix A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$\mathbf{x} + \mathbf{y}$

Element-wise vector addition

```
for {
  (idx_x, val_x) ← x
  (idx_y, val_y) ← y
  if idx_x == idx_y
} yield (idx_x, val_x + val_y)
```

For-comprehension

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

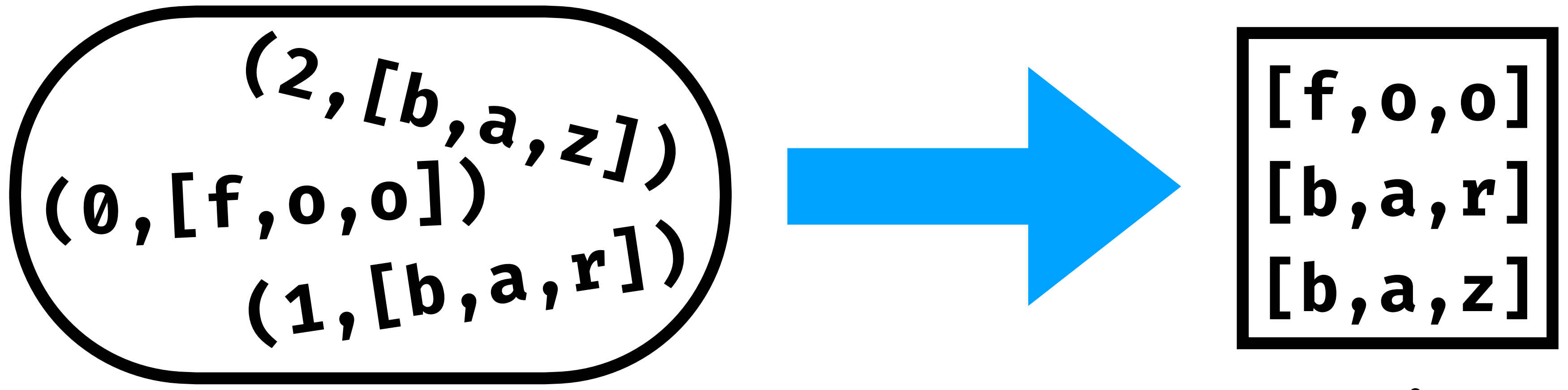
- **`DataBag`** `A ::=` **`Multiset`** `A`
- **`Vector`** `A ::=` **`Set`** `(ℕ, A)`
- **`Matrix`** `A ::=` **`Set`** `((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- **`DataBag`** `(ℕ,` **`Vector`** `A) →` **`Matrix`** `A`
- **`DataBag`** `A → (A → (ℕ, ℕ)) →` **`Matrix`** `A`



`DataBag (ℕ, Vector` `A)` **`Matrix`** `A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$\mathbf{x} + \mathbf{y}$ ➡

```
for {
  (idx_x, val_x) ← x
  (idx_y, val_y) ← y
  if idx_x == idx_y
} yield (idx_x, val_x + val_y)
```

Element-wise vector addition

For-comprehension

A **Combinator View** that enables **Domain-specific** opts.

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

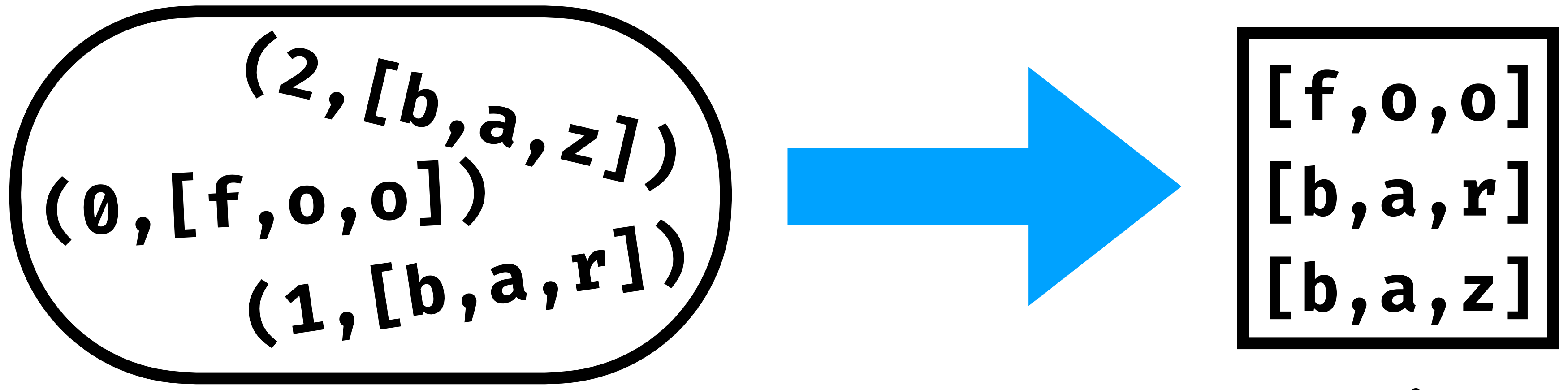
- **`DataBag`** `A ::=` **`Multiset`** `A`
- **`Vector`** `A ::=` **`Set`** `(ℕ, A)`
- **`Matrix`** `A ::=` **`Set`** `((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- **`DataBag`** `(ℕ,` **`Vector`** `A) →` **`Matrix`** `A`
- **`DataBag`** `A → (A → (ℕ, ℕ)) →` **`Matrix`** `A`



`DataBag (ℕ, Vector` `A)`

`Matrix` `A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$\mathbf{x} + \mathbf{y}$

```
for {
  (idx_x, val_x) ← x
  (idx_y, val_y) ← y
  if idx_x == idx_y
} yield (idx_x, val_x + val_y)
```

Element-wise vector addition

For-comprehension

A **Combinator View** that enables **Domain-specific** opts.
➡ **Instruction selection** and **Linear Algebra rewrites**

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

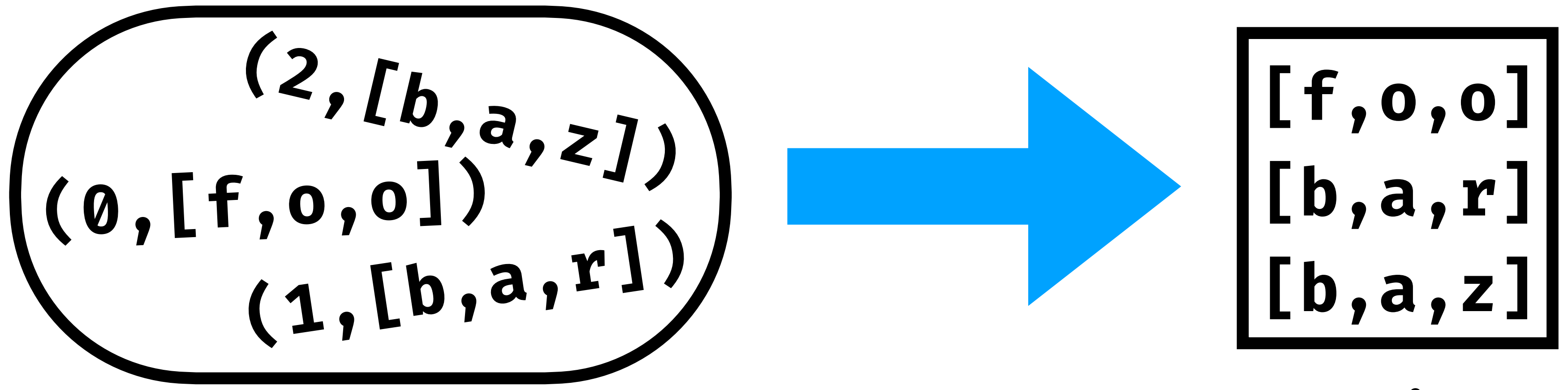
- **`DataBag`** `A ::=` **`Multiset`** `A`
- **`Vector`** `A ::=` **`Set`** `(ℕ, A)`
- **`Matrix`** `A ::=` **`Set`** `((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- **`DataBag`** `(ℕ,` **`Vector`** `A) →` **`Matrix`** `A`
- **`DataBag`** `A → (A → (ℕ, ℕ)) →` **`Matrix`** `A`



`DataBag (ℕ, Vector` `A)` **`Matrix`** `A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$$\mathbf{x} + \mathbf{y} \;\Rightarrow$$

```
for {
  (idx_x, val_x) ← x
  (idx_y, val_y) ← y
  if idx_x == idx_y
} yield (idx_x, val_x + val_y)
```

Element-wise vector addition

For-comprehension

A **Combinator View** that enables **Domain-specific** opts.
➡ **Instruction selection** and **Linear Algebra rewrites**

$$\mathbf{X}^\mathrm{T}\mathbf{X}$$

Gram matrix calculation

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

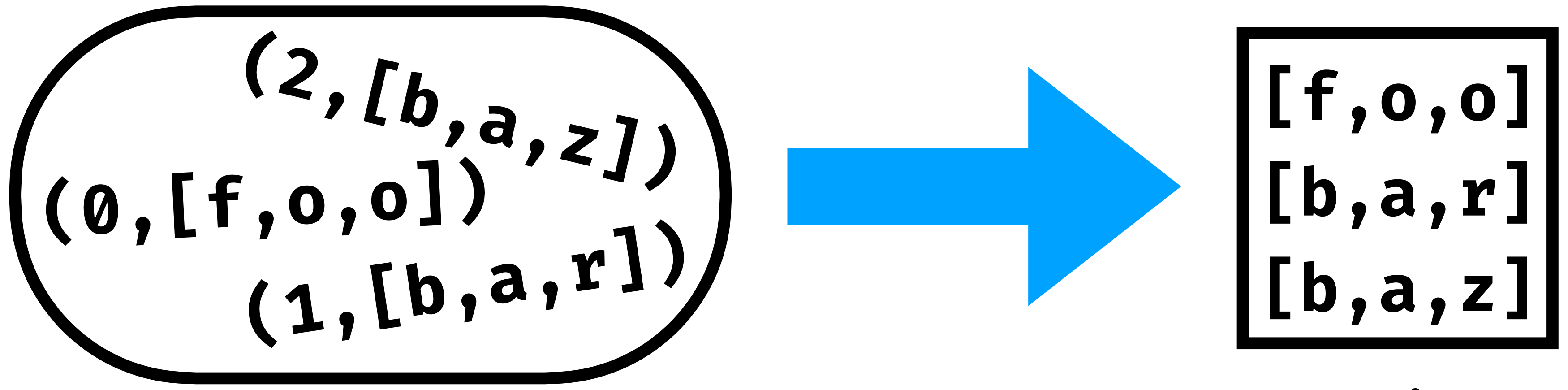
- `DataBag A ::= Multiset A`
- `Vector A ::= Set (ℕ, A)`
- `Matrix A ::= Set ((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- `DataBag (ℕ, Vector A) → Matrix A`
- `DataBag A → (A → (ℕ, ℕ)) → Matrix A`



`DataBag (ℕ, Vector A)` `Matrix A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$\mathbf{x} + \mathbf{y}$ →

```
for {
  (idx_x, val_x) ← x
  (idx_y, val_y) ← y
  if idx_x == idx_y
} yield (idx_x, val_x + val_y)
```

Element-wise vector addition

For-comprehension

A **Combinator View** that enables **Domain-specific** opts.
➡ **Instruction selection** and **Linear Algebra rewrites**

$\mathbf{X}^{\mathrm{T}}\mathbf{X}$ →

Gram matrix calculation

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

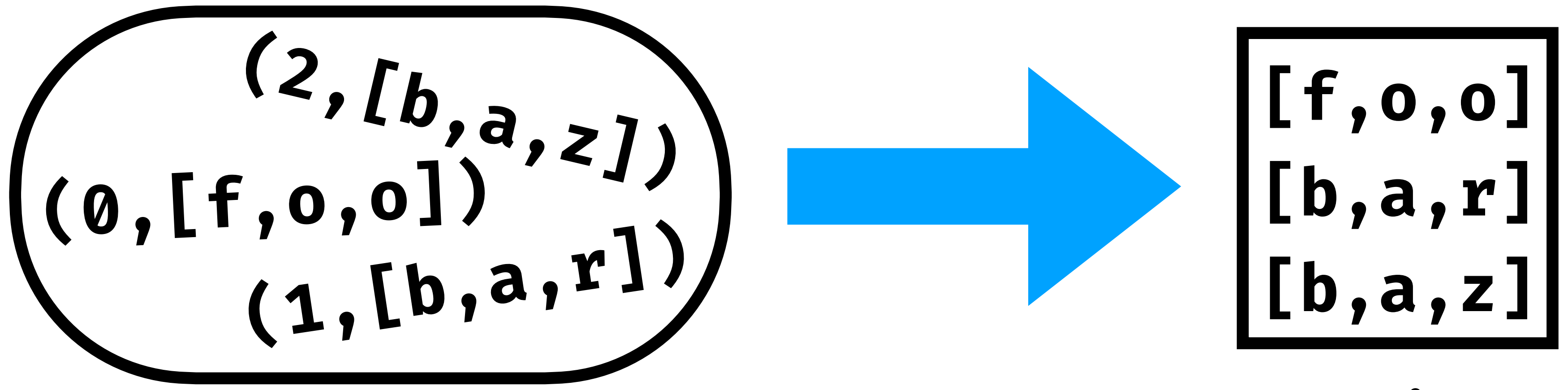
- **`DataBag`** `A ::=` **`Multiset`** `A`
- **`Vector`** `A ::=` **`Set`** `(ℕ, A)`
- **`Matrix`** `A ::=` **`Set`** `((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- **`DataBag`** `(ℕ,` **`Vector`** `A) →` **`Matrix`** `A`
- **`DataBag`** `A → (A → (ℕ, ℕ)) →` **`Matrix`** `A`



`DataBag` `(ℕ,` **`Vector`** `A)`

`Matrix` `A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$\mathbf{x} + \mathbf{y}$ ➡

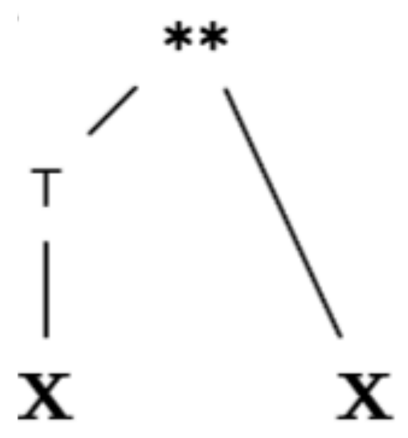
```
for {
  (idx_x, val_x) ← x
  (idx_y, val_y) ← y
  if idx_x == idx_y
} yield (idx_x, val_x + val_y)
```

Element-wise vector addition

For-comprehension

A **Combinator View** that enables **Domain-specific** opts.
➡ **Instruction selection** and **Linear Algebra rewrites**

$\mathbf{X}^{\mathrm{T}}\mathbf{X}$ ➡



Gram matrix calculation

Operator tree

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

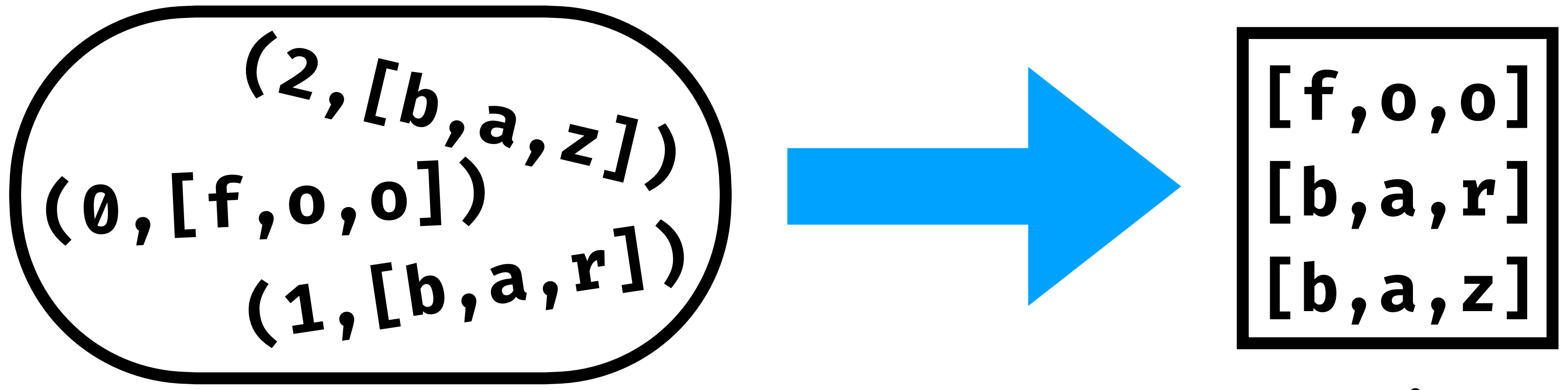
- `DataBag A ::= Multiset A`
- `Vector A ::= Set (ℕ, A)`
- `Matrix A ::= Set ((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- `DataBag (ℕ, Vector A) → Matrix A`
- `DataBag A → (A → (ℕ, ℕ)) → Matrix A`



DataBag (ℕ, Vector A)

Matrix A

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$$\mathbf{x} + \mathbf{y}$$

Element-wise vector addition

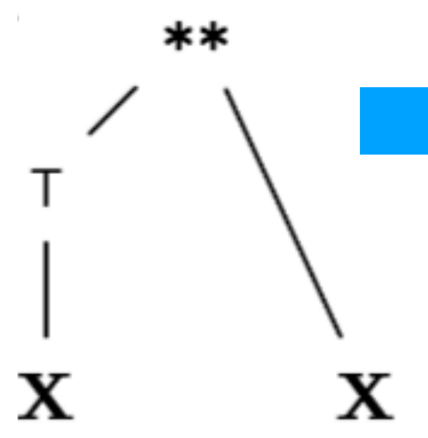
```
for {
  (idx_x, val_x) ← x
  (idx_y, val_y) ← y
  if idx_x == idx_y
} yield (idx_x, val_x + val_y)
```

For-comprehension

A **Combinator View** that enables **Domain-specific** opts.
➡ **Instruction selection** and **Linear Algebra rewrites**

$$\mathbf{X}^{\mathrm{T}}\mathbf{X}$$

Gram matrix calculation



Operator tree

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

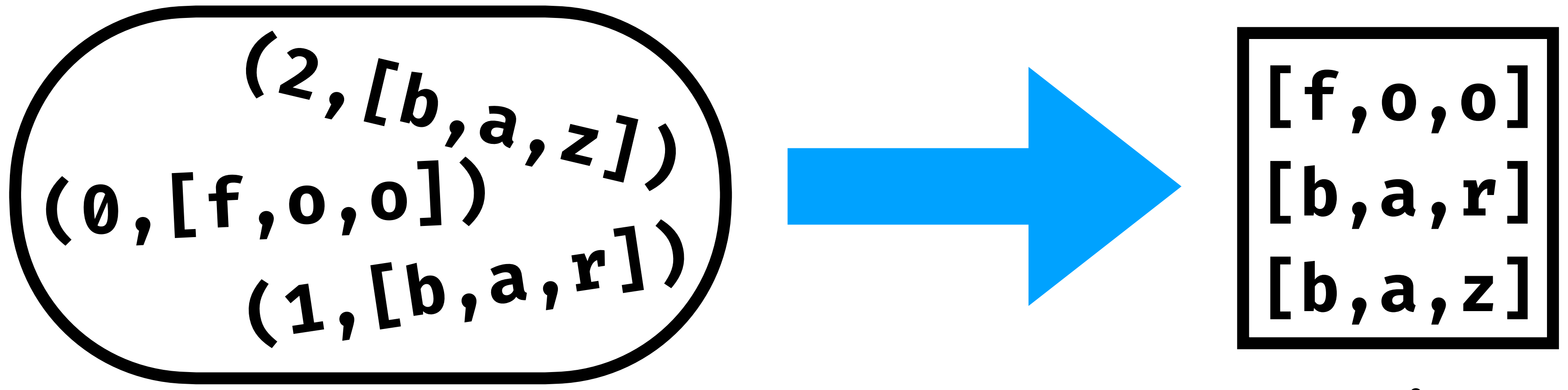
- `DataBag A ::= Multiset A`
- `Vector A  ::= Set (ℕ, A)`
- `Matrix A  ::= Set ((ℕ, ℕ), A)`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- `DataBag (ℕ, Vector A) → Matrix A`
- `DataBag A → (A → (ℕ, ℕ)) → Matrix A`



`DataBag (ℕ, Vector A)`

`Matrix A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$\mathbf{x} + \mathbf{y}$

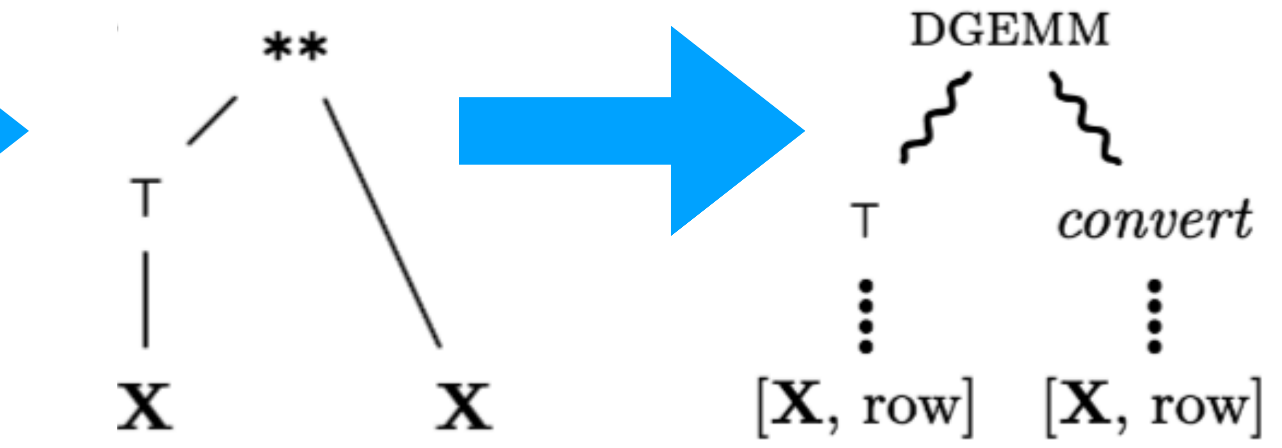
```
for {
  (idx_x, val_x) ← x
  (idx_y, val_y) ← y
  if idx_x == idx_y
} yield (idx_x, val_x + val_y)
```

Element-wise vector addition

For-comprehension

A **Combinator View** that enables **Domain-specific** opts.
➡ **Instruction selection** and **Linear Algebra rewrites**

$\mathbf{X}^T\mathbf{X}$



Gram matrix calculation

Operator tree

How does Lara work?

Lara has a Quotation Based **DSL** embedded in **Scala**.

Types (Monads):

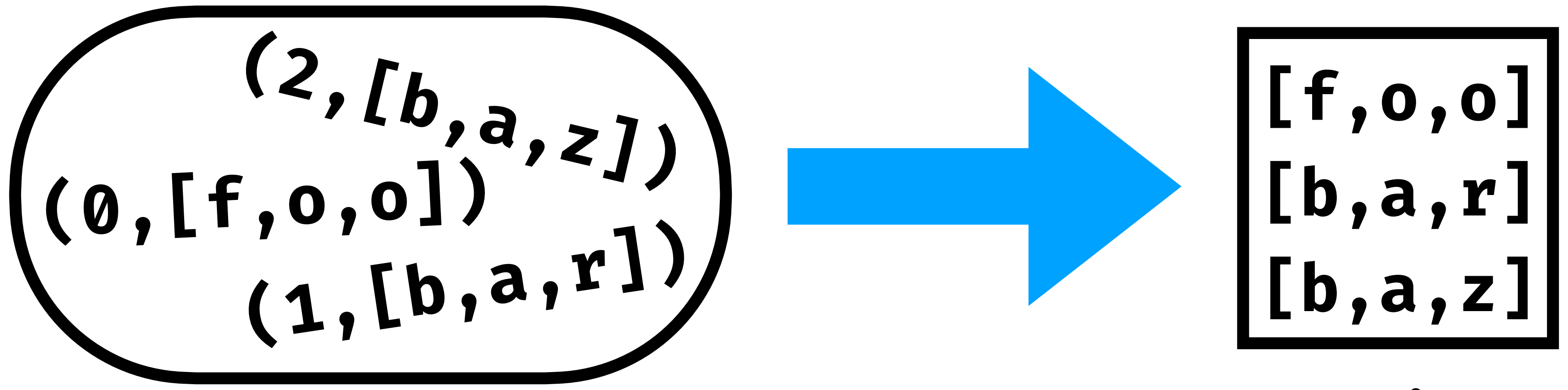
- `DataBag A ::= Multiset A`
- `Vector A ::= Set (ℕ, A)`
- `Matrix A ::= Set ((ℕ, ℕ), A)`
- `Stream A ::= Infinite Multiset A`

Operations:

- For-comprehensions
- Linear algebra functions

Type conversions track **provenance**:

- `DataBag (ℕ, Vector A) → Matrix A`
- `DataBag A → (A → (ℕ, ℕ)) → Matrix A`



`DataBag (ℕ, Vector A)`

`Matrix A`

Lara also has an **IR** with two **"views"**.

A **Monadic View** that enables **Dataflow** optimisations
➡ **Operator Fusion** and **Operator Pushdown**

$\mathbf{x} + \mathbf{y}$ ➡

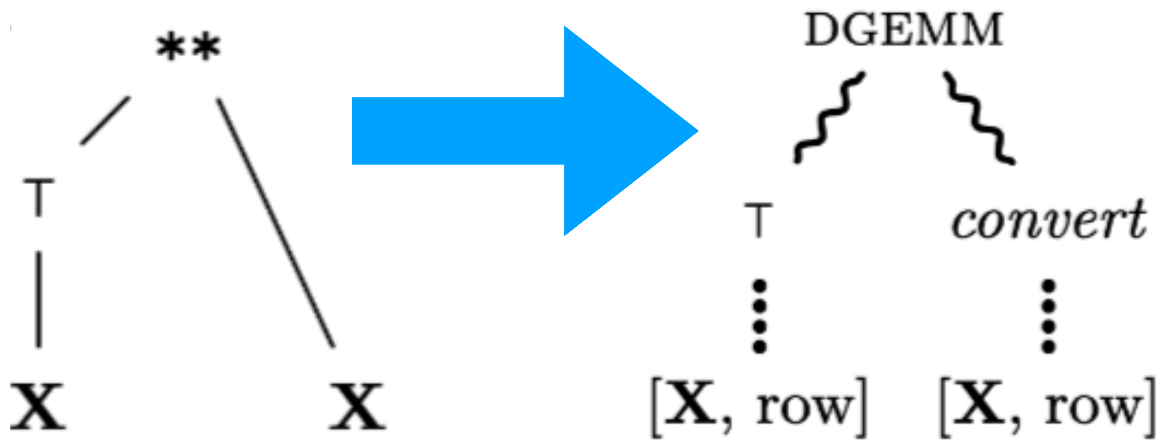
```
for {
  (idx_x, val_x) ← x
  (idx_y, val_y) ← y
  if idx_x == idx_y
} yield (idx_x, val_x + val_y)
```

Element-wise vector addition

For-comprehension

A **Combinator View** that enables **Domain-specific** opts.
➡ **Instruction selection** and **Linear Algebra rewrites**

$\mathbf{X}^{\mathrm{T}}\mathbf{X}$



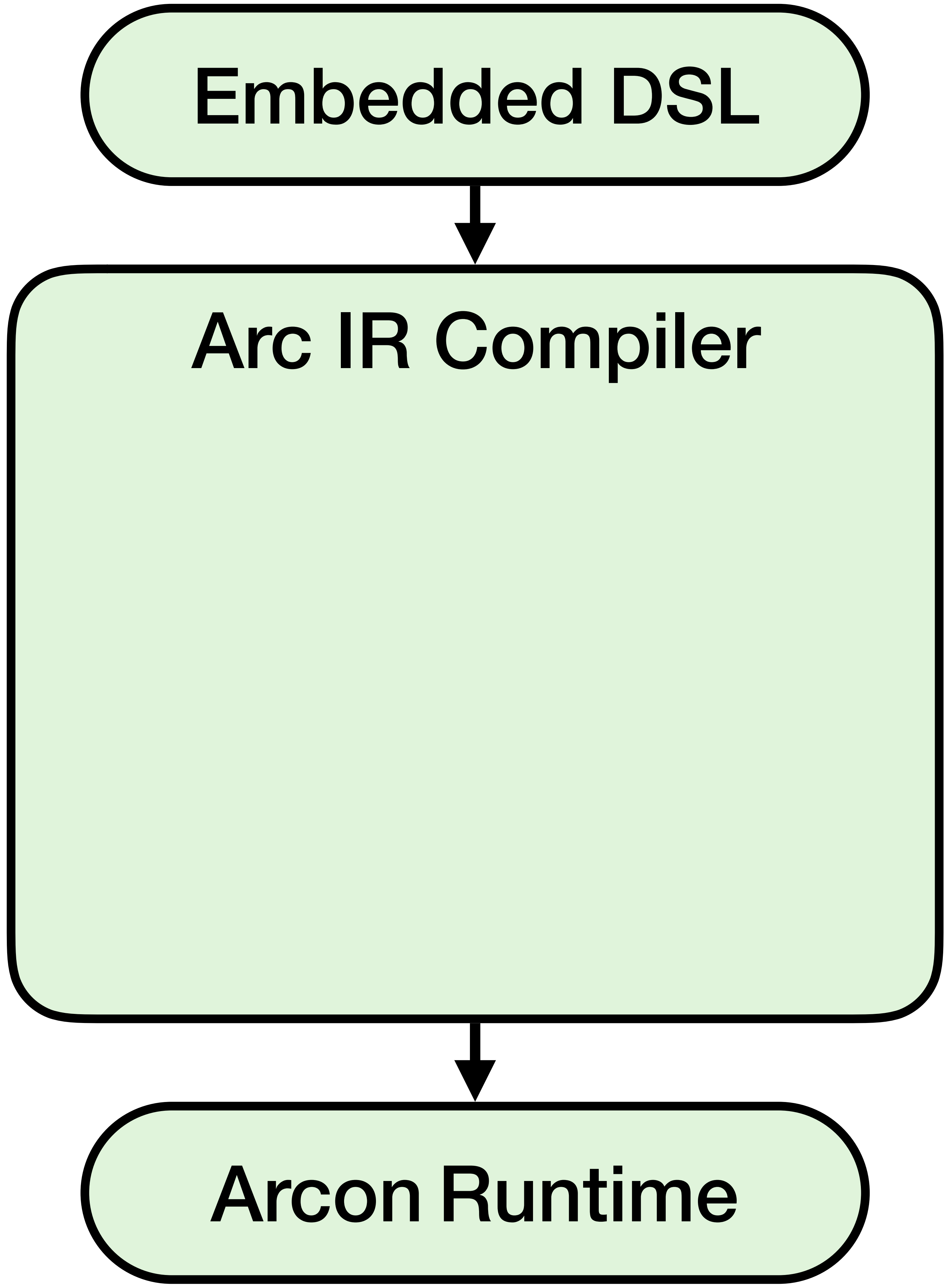
Gram matrix calculation

Operator tree

Arc - Compiler Approach

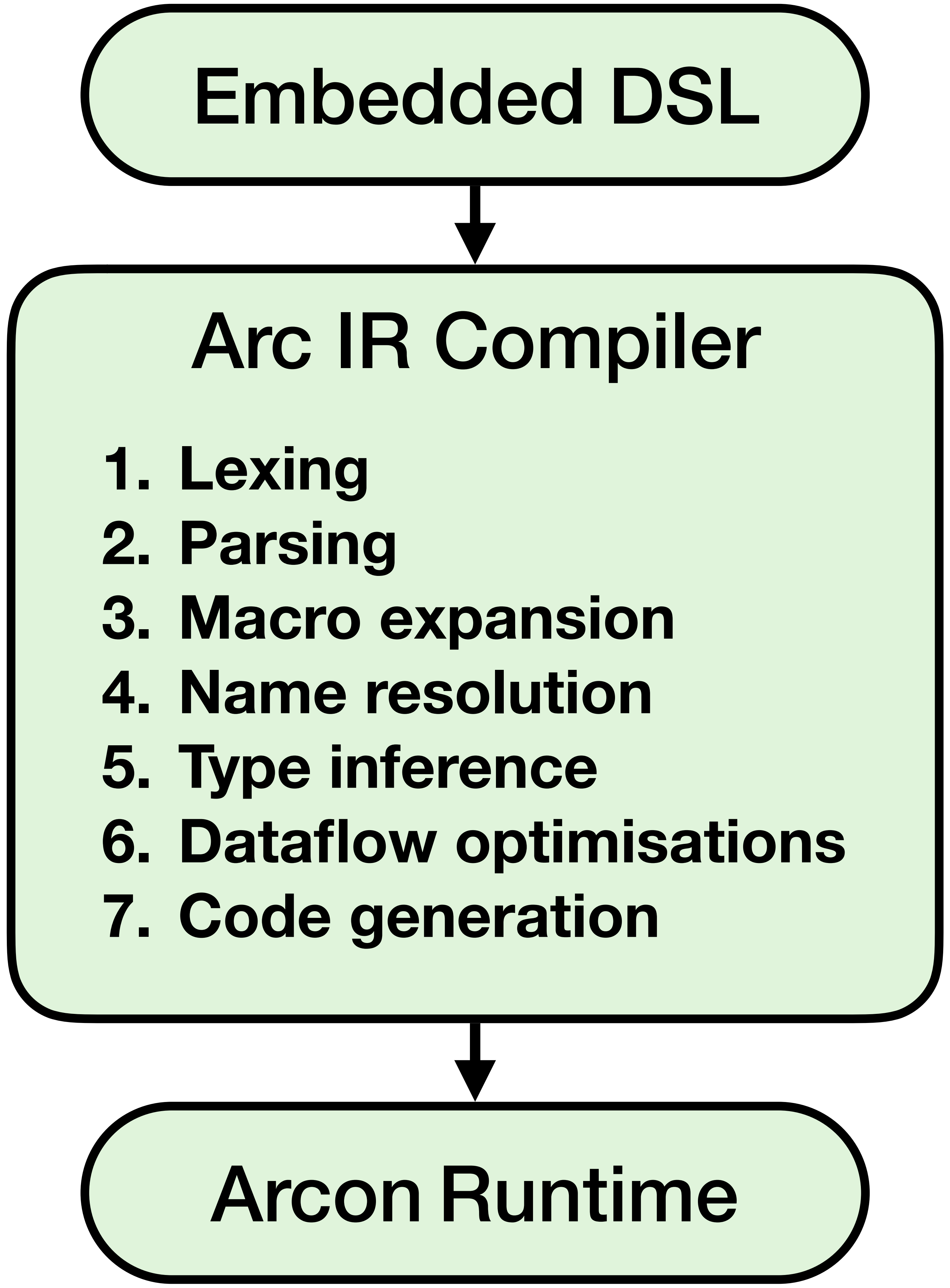
Arc - Compiler Approach

Original approach



Arc - Compiler Approach

Original approach



Arc - Compiler Approach

Original approach

Embedded DSL

↓

Arc IR Compiler

1. **Lexing**
2. **Parsing**
3. **Macro expansion**
4. **Name resolution**
5. **Type inference**
6. **Dataflow optimisations**
7. **Code generation**

↓

Arcon Runtime

Current approach

Embedded DSL

↓

MLIR (Multi-Level IR)

? ? ? ? ?

? ? ? ? ?

? ? ? ? ? ?

↓

Arcon Runtime

What is MLIR (Multi-Level IR)?

What is MLIR (Multi-Level IR)?

MLIR: A Compiler Infrastructure for the End of Moore's Law

Chris Lattner * — Google
Mehdi Amini — Google
Uday Bondhugula — IISc
Albert Cohen — Google
Andy Davis — Google

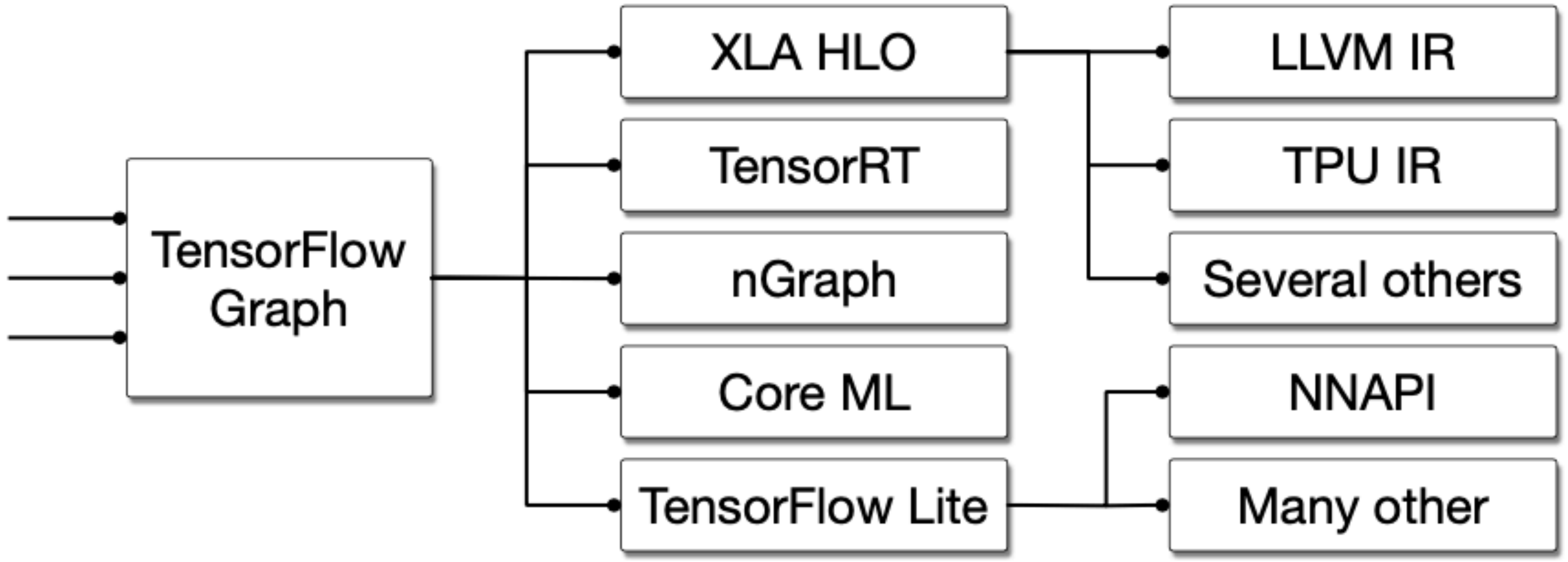
Jacques Pienaar — Google
River Riddle — Google
Tatiana Shpeisman — Google
Nicolas Vasilache — Google

Oleksandr Zinenko — Google

Where did MLIR come from?

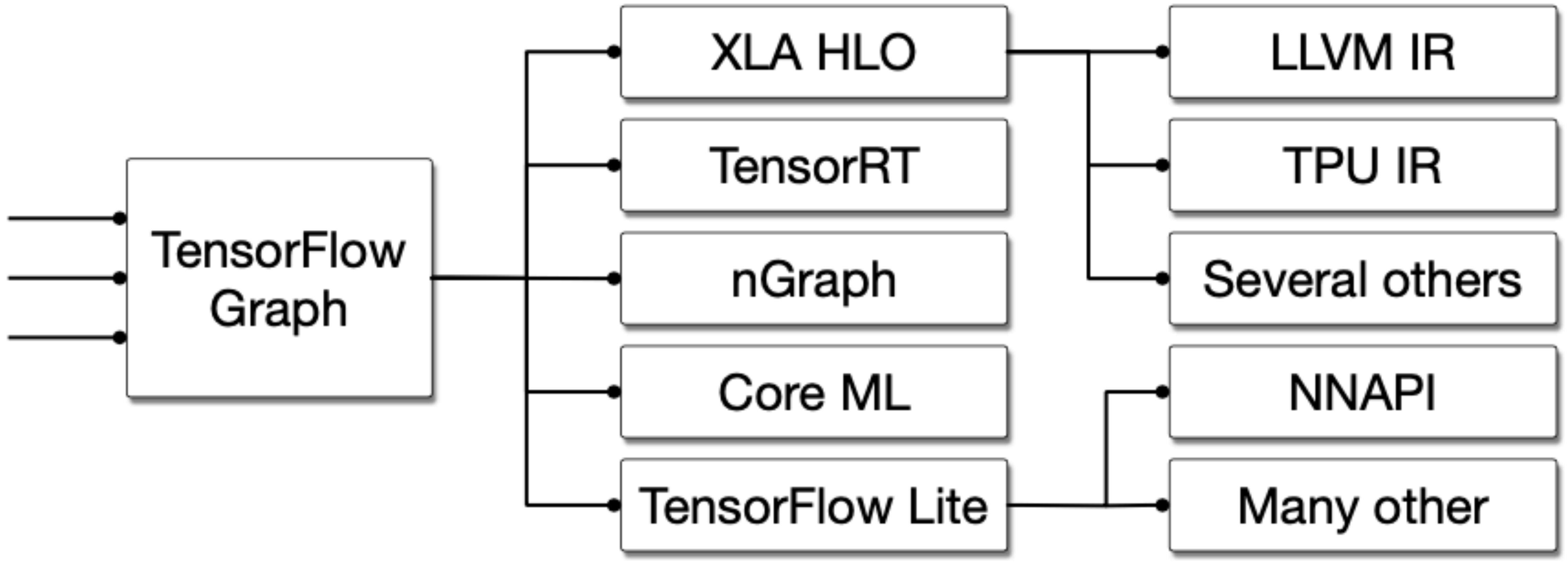
Where did MLIR come from?

Compiler infrastructure of
Google's **TensorFlow**



Where did MLIR come from?

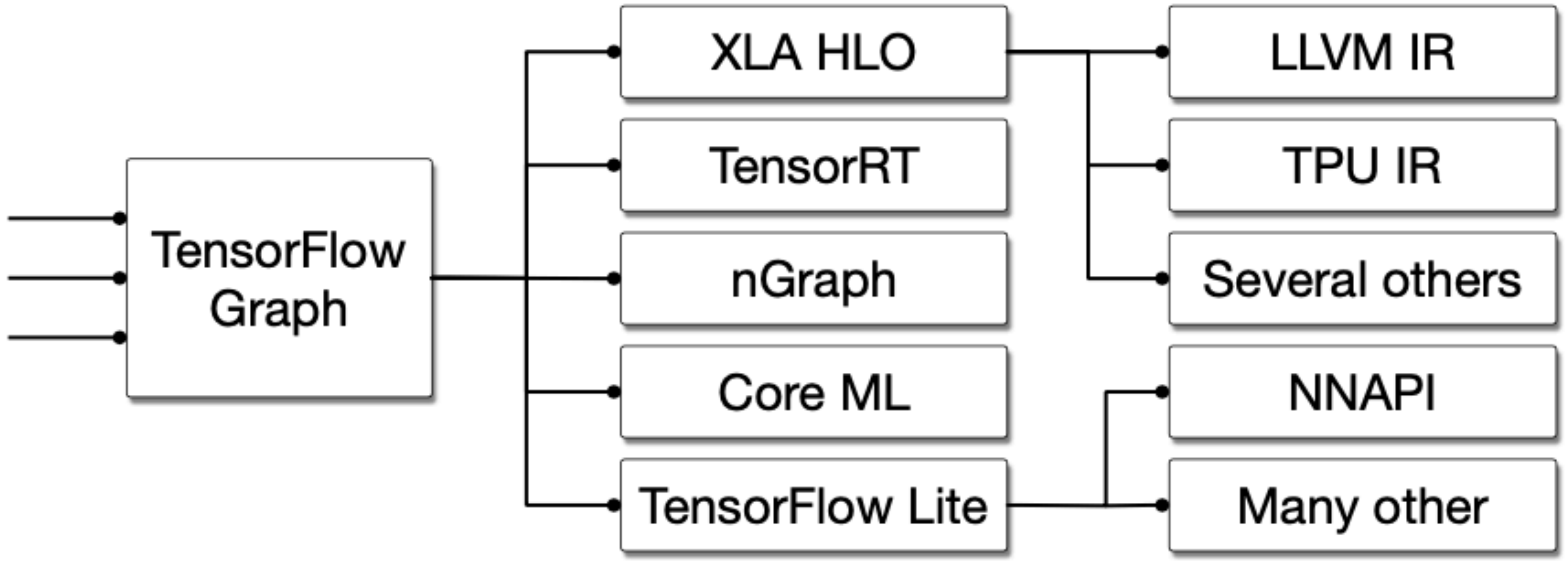
Compiler infrastructure of
Google's **TensorFlow**



Problem: No re-use between IR compilers

Where did MLIR come from?

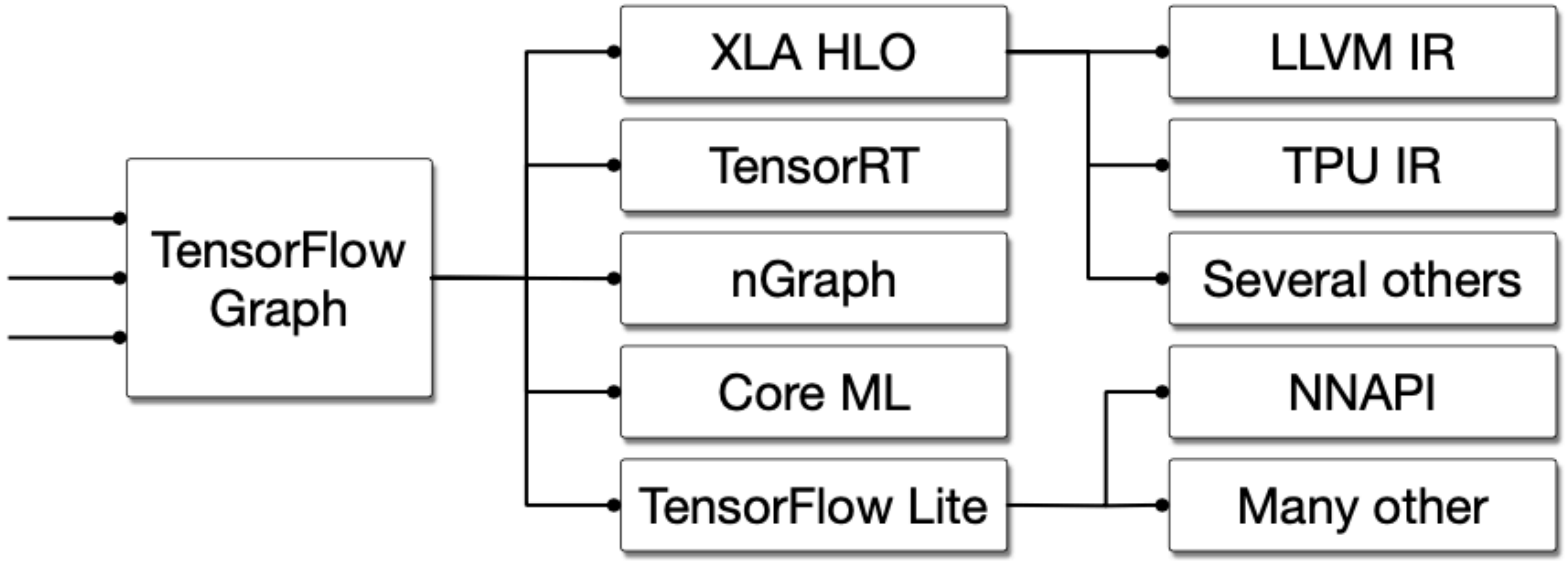
Compiler infrastructure of
Google's **TensorFlow**



Problem: No re-use between IR compilers
➡ A compiler infrastructure like LLVM's is needed

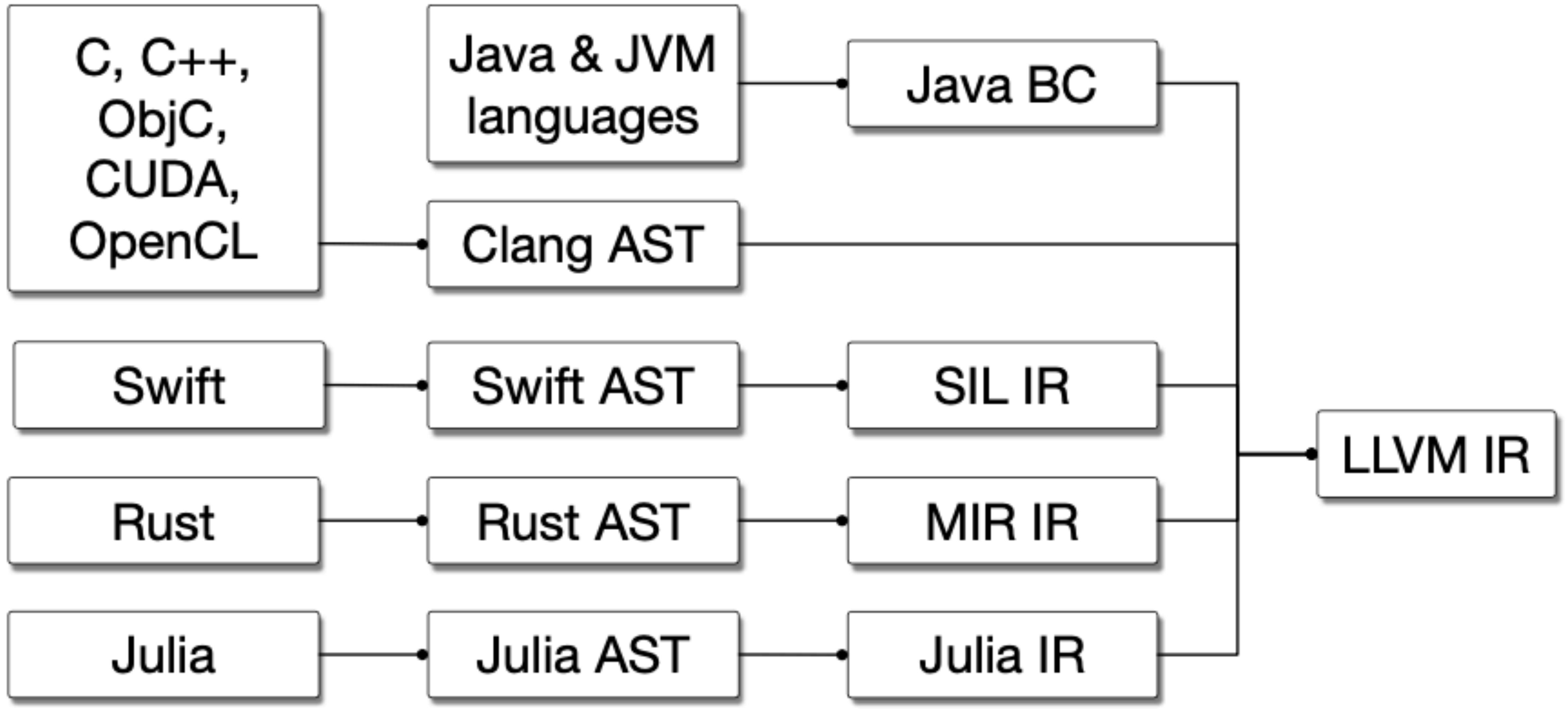
Where did MLIR come from?

Compiler infrastructure of Google's **TensorFlow**



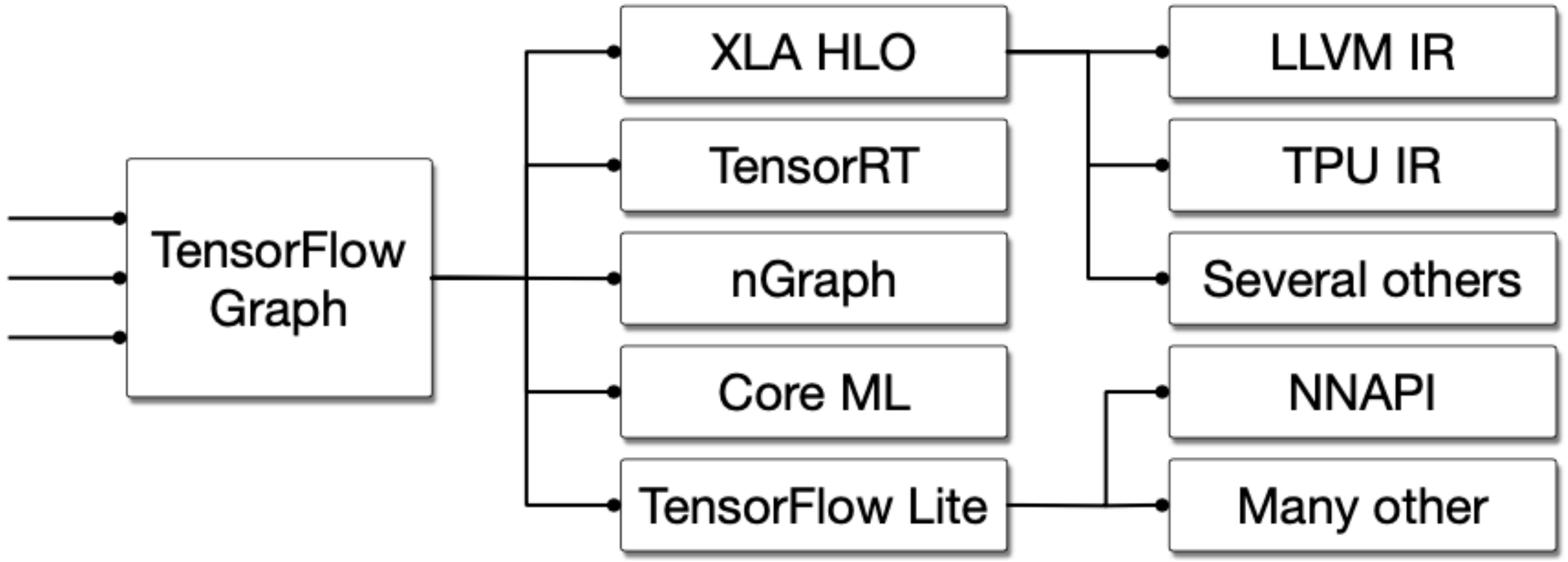
Problem: No re-use between IR compilers
➡ A compiler infrastructure like LLVM's is needed

Compiler infrastructure of languages using **LLVM**



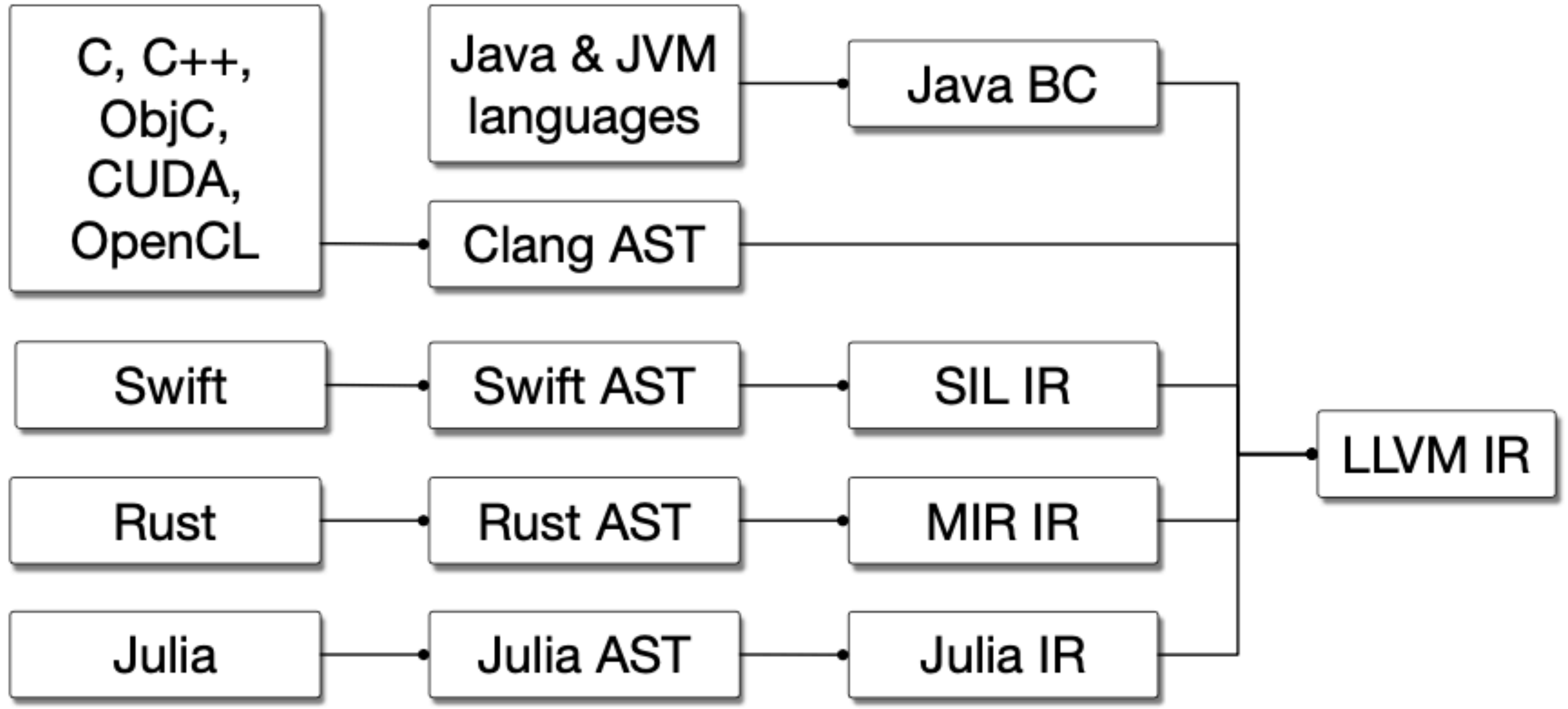
Where did MLIR come from?

Compiler infrastructure of Google's **TensorFlow**



Problem: No re-use between IR compilers
➡ A compiler infrastructure like LLVM's is needed

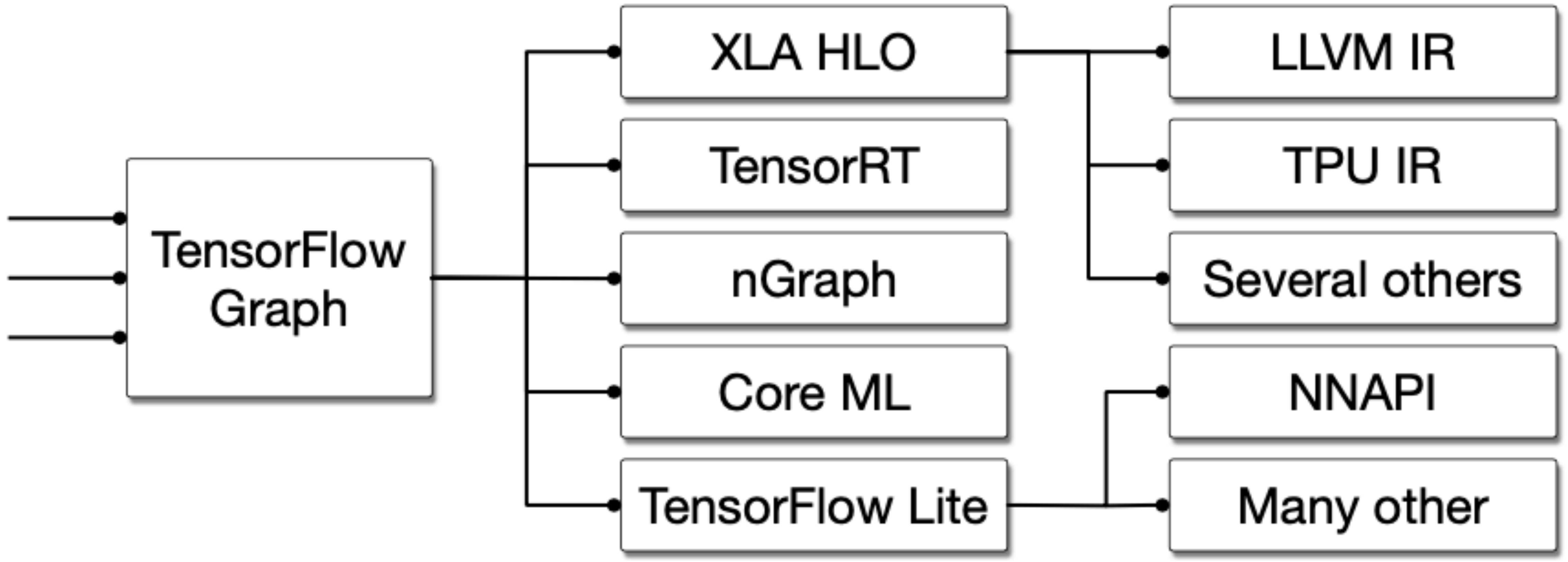
Compiler infrastructure of languages using **LLVM**



Problem: LLVM is locked to one level of abstraction

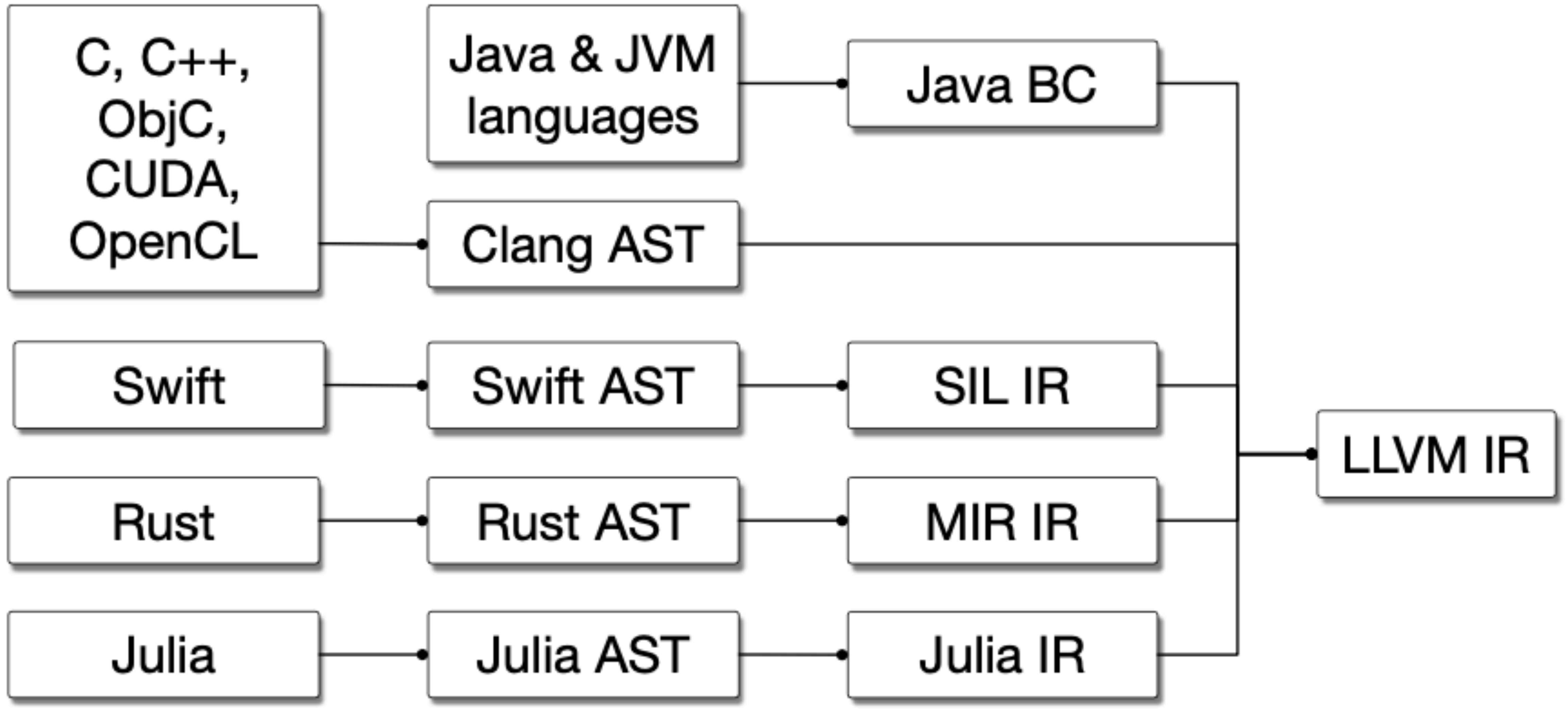
Where did MLIR come from?

Compiler infrastructure of
Google's **TensorFlow**



Problem: No re-use between IR compilers
➡ A compiler infrastructure like LLVM's is needed

Compiler infrastructure of
languages using **LLVM**



Problem: LLVM is locked to one level of abstraction
➡ There is a need for a more general solution

How does MLIR work?

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

MLIR handles *parsing*, *type checking/ inference*, *line-number tracking, ...*

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

MLIR handles *parsing*, *type checking/ inference*, *line-number tracking, ...*

Additionally, MLIR provides **tooling** for *testing, parallel compilation, documentation*, *CLI, ...*

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

MLIR handles *parsing*, *type checking/ inference*, *line-number tracking, ...*

Additionally, MLIR provides **tooling** for *testing, parallel compilation, documentation*, *CLI, ...*

Builtin dialects (14 total):

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

MLIR handles *parsing*, *type checking/ inference*, *line-number tracking, ...*

Additionally, MLIR provides **tooling** for *testing, parallel compilation, documentation*, *CLI, ...*

Builtin dialects (14 total):

- **`std`** – Basic types and arithmetics

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

MLIR handles *parsing*, *type checking/ inference*, *line-number tracking, ...*

Additionally, MLIR provides **tooling** for *testing, parallel compilation, documentation*, *CLI, ...*

Builtin dialects (14 total):

- **`std`** - Basic types and arithmetics
- **`loop`** - Ordinary loops

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

MLIR handles *parsing*, *type checking/ inference*, *line-number tracking, ...*

Additionally, MLIR provides **tooling** for *testing, parallel compilation, documentation*, *CLI, ...*

Builtin dialects (14 total):

- **`std`** - Basic types and arithmetics
- **`loop`** - Ordinary loops
- **`affine`** - Polyhedral loops

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

MLIR handles *parsing*, *type checking/ inference*, *line-number tracking, ...*

Additionally, MLIR provides **tooling** for *testing, parallel compilation, documentation*, *CLI, ...*

Builtin dialects (14 total):

- **`std`** - Basic types and arithmetics
- **`loop`** - Ordinary loops
- **`affine`** - Polyhedral loops
- **`linalg`** - Linear algebra

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

MLIR handles *parsing*, *type checking/ inference*, *line-number tracking, ...*

Additionally, MLIR provides **tooling** for *testing, parallel compilation, documentation*, *CLI, ...*

Builtin dialects (14 total):

- **`std`** - Basic types and arithmetics
- **`loop`** - Ordinary loops
- **`affine`** - Polyhedral loops
- **`linalg`** - Linear algebra
- **`gpu`** - CUDA & OpenCL abstraction

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

MLIR handles *parsing*, *type checking/ inference*, *line-number tracking, ...*

Additionally, MLIR provides **tooling** for *testing, parallel compilation, documentation*, *CLI, ...*

Builtin dialects (14 total):

- **`std`** - Basic types and arithmetics
- **`loop`** - Ordinary loops
- **`affine`** - Polyhedral loops
- **`linalg`** - Linear algebra
- **`gpu`** - CUDA & OpenCL abstraction
- **`llvm`** - LLVM-IR code generation

How does MLIR work?

MLIR is an **IR** which can be extended with new "**dialects**"

Dialects define *operations*, *types*, *type constraints*, *rewrite rules*, *lowerings*, ...

Dialects adhere to the same **meta-syntax** and **meta-semantics**
➡ Dialects can **coexist** in the same program

MLIR handles *parsing*, *type checking/ inference*, *line-number tracking, ...*

Additionally, MLIR provides **tooling** for *testing, parallel compilation, documentation, CLI, ...*

Builtin dialects (14 total):

- **`std`** - Basic types and arithmetics
- **`loop`** - Ordinary loops
- **`affine`** - Polyhedral loops
- **`linalg`** - Linear algebra
- **`gpu`** - CUDA & OpenCL abstraction
- **`llvm`** - LLVM-IR code generation

MLIR - Meta-Syntax/Semantics

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"`

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION ::= "DIALECT.OP_NAME"(ARGS) ........ ...... ...... ......`

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) ......   ......   ......`

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS}      ......      ......`

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE       ::= !DIALECT.TYPE_NAME<TYPES>`

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE       ::= !DIALECT.TYPE_NAME<TYPES>`

Meta-semantics: All values are **SSA**, **typed**, **scoped** (and so on)

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE       ::= !DIALECT.TYPE_NAME<TYPES>`

Meta-semantics: All values are **SSA**, **typed**, **scoped** (and so on)

Example:

Compute the
max of two values

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE       ::= !DIALECT.TYPE_NAME<TYPES>`

Meta-semantics: All values are **SSA**, **typed**, **scoped** (and so on)

Example:

Compute the
max of two values

```
c = a > b
max = if c {
  a
} else {
  b
}
```

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE ::= !DIALECT.TYPE_NAME<TYPES>`

Meta-semantics: All values are **SSA**, **typed**, **scoped** (and so on)

Example:

Compute the
max of two values

```
c = a > b
max = if c {
  a
} else {
  b
}
```

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE       ::= !DIALECT.TYPE_NAME<TYPES>`

Meta-semantics: All values are **SSA**, **typed**, **scoped** (and so on)

Example:

Compute the
max of two values

```
c = a > b
max = if c {
  a
} else {
  b
}
```

```
%c = "arc.greater_than"(%a, %b)
```

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE       ::= !DIALECT.TYPE_NAME<TYPES>`

Meta-semantics: All values are **SSA**, **typed**, **scoped** (and so on)

Example:

Compute the
max of two values

```
c = a > b
max = if c {
  a
} else {
  b
}
```

```
%c = "arc.greater_than"(%a, %b) : (!arc.int, !arc.int) → (!arc.bool)
```

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE ::= !DIALECT.TYPE_NAME<TYPES>`

Meta-semantics: All values are **SSA**, **typed**, **scoped** (and so on)

Example:

Compute the
max of two values

```
c = a > b
max = if c {
  a
} else {
  b
}
```

```
%c = "arc.greater_than"(%a, %b) : (!arc.int, !arc.int) → (!arc.bool)
%max = "arc.if"(%c)
```

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE       ::= !DIALECT.TYPE_NAME<TYPES>`

Meta-semantics: All values are **SSA**, **typed**, **scoped** (and so on)

Example:

Compute the
max of two values

```
c = a > b
max = if c {
  a
} else {
  b
}
```

```
%c = "arc.greater_than"(%a, %b) : (!arc.int, !arc.int) → (!arc.bool)
%max = "arc.if"(%c) ({
   ...
}, {
   ...
}) : (arc.bool) → (!arc.int)
```

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION  ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE       ::= !DIALECT.TYPE_NAME<TYPES>`

Meta-semantics: All values are **SSA**, **typed**, **scoped** (and so on)

Example:

Compute the
max of two values

```
c = a > b
max = if c {
  a
} else {
  b
}
```

```
%c = "arc.greater_than"(%a, %b) : (!arc.int, !arc.int) → (!arc.bool)
%max = "arc.if"(%c) ({
  "arc.yield"(%a)
}, {
  "arc.yield"(%b)
}) : (arc.bool) → (!arc.int)
```

MLIR - Meta-Syntax/Semantics

Meta-syntax (with some parts optional):

- `ASSIGNMENT ::= %VAR_NAME = OPERATION`
- `OPERATION ::= "DIALECT.OP_NAME"(ARGS) (REGIONS) {ATTRS} : (TYPES) → (TYPES)`
- `TYPE ::= !DIALECT.TYPE_NAME<TYPES>`

Meta-semantics: All values are **SSA**, **typed**, **scoped** (and so on)

Example:

Compute the
<u>max of two values</u>

```
c = a > b
max = if c {
  a
} else {
  b
}
```

```
%c = "arc.greater_than"(%a, %b) : (!arc.int, !arc.int) → (!arc.bool)
%max = "arc.if"(%c) ({
  "arc.yield"(%a) : (!arc.int) → (!arc.int)
}, {
  "arc.yield"(%b) : (!arc.int) → (!arc.int)
}) : (arc.bool) → (!arc.int)
```

MLIR - Defining Operations

MLIR - Defining Operations

The **structure** of each operation is defined using an high-level **DSL**, while **details** are implemented in **C++**

MLIR - Defining Operations

The **structure** of each operation is defined using an high-level **DSL**, while **details** are implemented in **C++**

Example:

MLIR - Defining Operations

The **structure** of each operation is defined using an high-level **DSL**, while **details** are implemented in **C++**

Example:

MLIR - Defining Operations

The **structure** of each operation is defined using an high-level **DSL**, while **details** are implemented in **C++**

Example:

```
def GreaterThanOp : Op<"greater_than", [...........................]> {









}
```

MLIR - Defining Operations

The **structure** of each operation is defined using an high-level **DSL**, while **details** are implemented in **C++**

Example:

```
def GreaterThanOp : Op<"greater_than", [............................]> {
  let summary     = "greater than operation";
  let description = [{ Returns true if $left is greater than $right }];




}
```

MLIR - Defining Operations

The **structure** of each operation is defined using an high-level **DSL**, while **details** are implemented in **C++**

Example:

```
def GreaterThanOp : Op<"greater_than", [............................]> {
  let summary     = "greater than operation";
  let description = [{ Returns true if $left is greater than $right }];
  let arguments   = (ins AnyType:$left, AnyType:$right);







}
```

MLIR - Defining Operations

The **structure** of each operation is defined using an high-level **DSL**, while **details** are implemented in **C++**

Example:

```
def GreaterThanOp : Op<"greater_than", [............................]> {
  let summary     = "greater than operation";
  let description = [{ Returns true if $left is greater than $right }];
  let arguments   = (ins AnyType:$left, AnyType:$right);
  let results     = (outs BoolType:$output);





}
```

MLIR - Defining Operations

The **structure** of each operation is defined using an high-level **DSL**, while **details** are implemented in **C++**

Example:

```
def GreaterThanOp : Op<"greater_than", [ArgsAreSameType, NoSideEffect]> {
  let summary     = "greater than operation";
  let description = [{ Returns true if $left is greater than $right }];
  let arguments   = (ins AnyType:$left, AnyType:$right);
  let results     = (outs BoolType:$output);




}
```

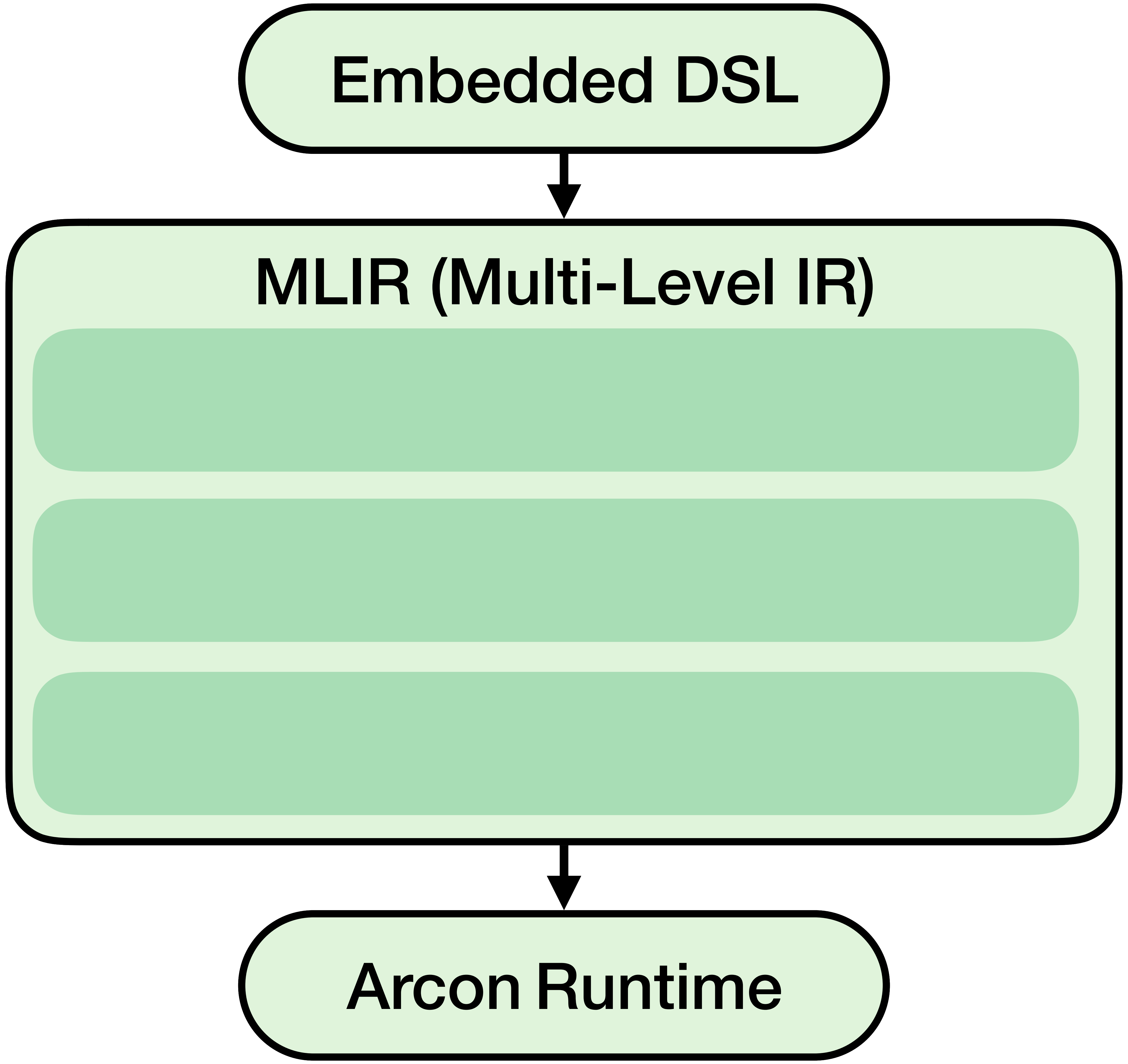
MLIR - Defining Operations

The **structure** of each operation is defined using an high-level **DSL**, while **details** are implemented in **C++**

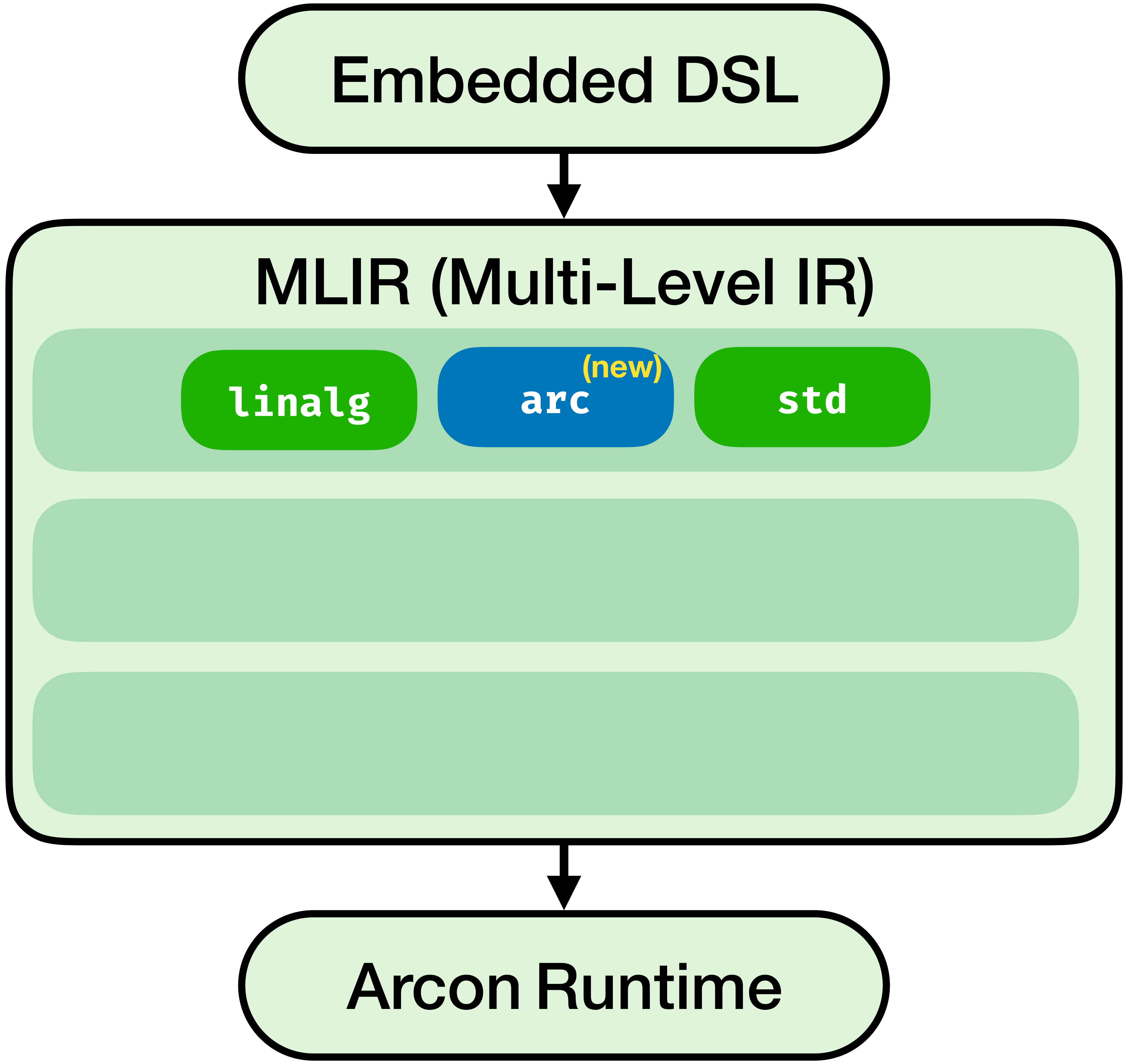
Example:

```
def GreaterThanOp : Op<"greater_than", [ArgsAreSameType, NoSideEffect]> {
  let summary     = "greater than operation";
  let description = [{ Returns true if $left is greater than $right }];
  let arguments   = (ins AnyType:$left, AnyType:$right);
  let results     = (outs BoolType:$output);
  // Optional
  let regions     = ...
  let verifier    = ...
  let parser      = ...
  let printer     = ...
}
```

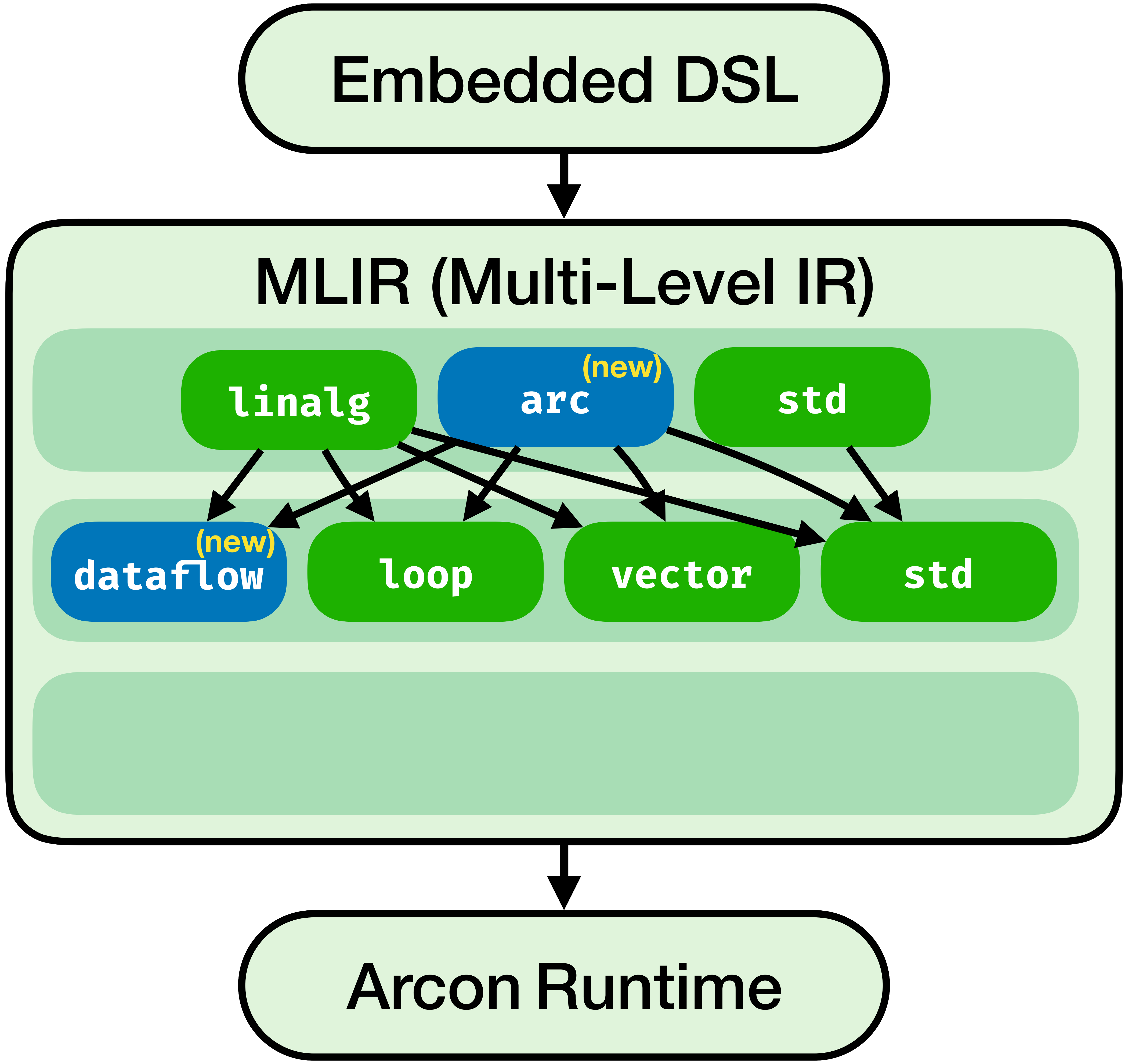
Current approach



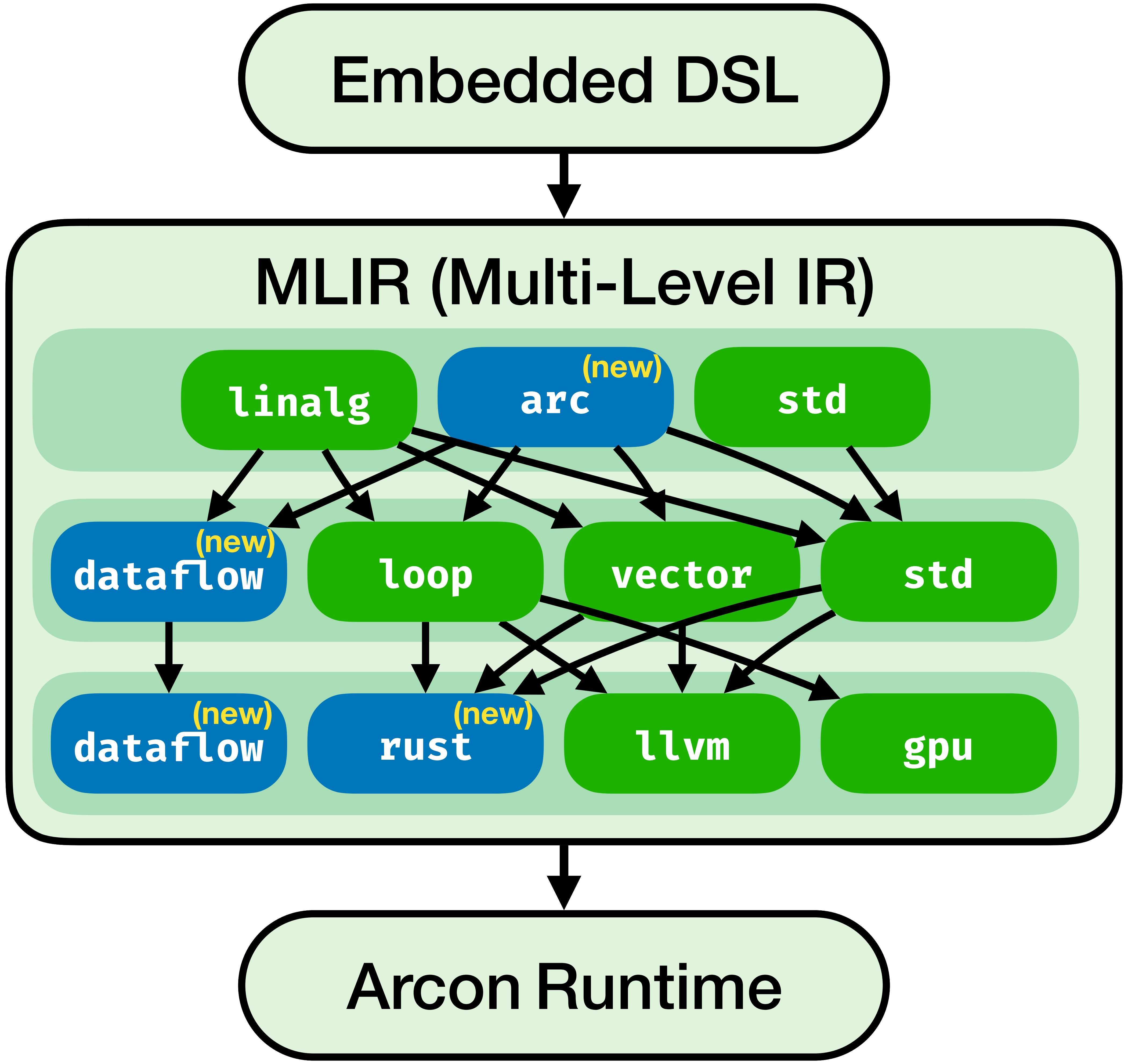
Current approach



Current approach

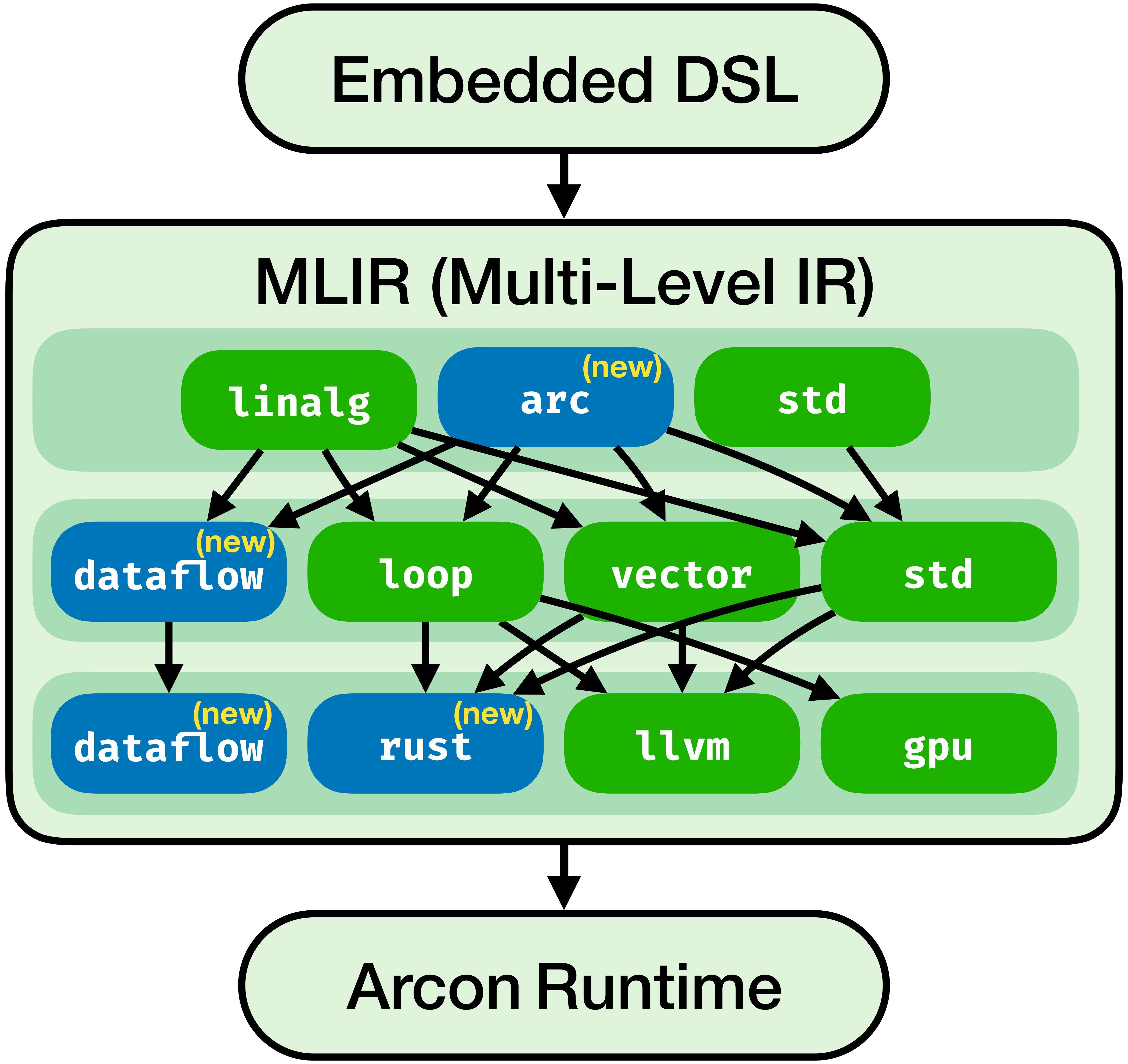


Current approach



Summary

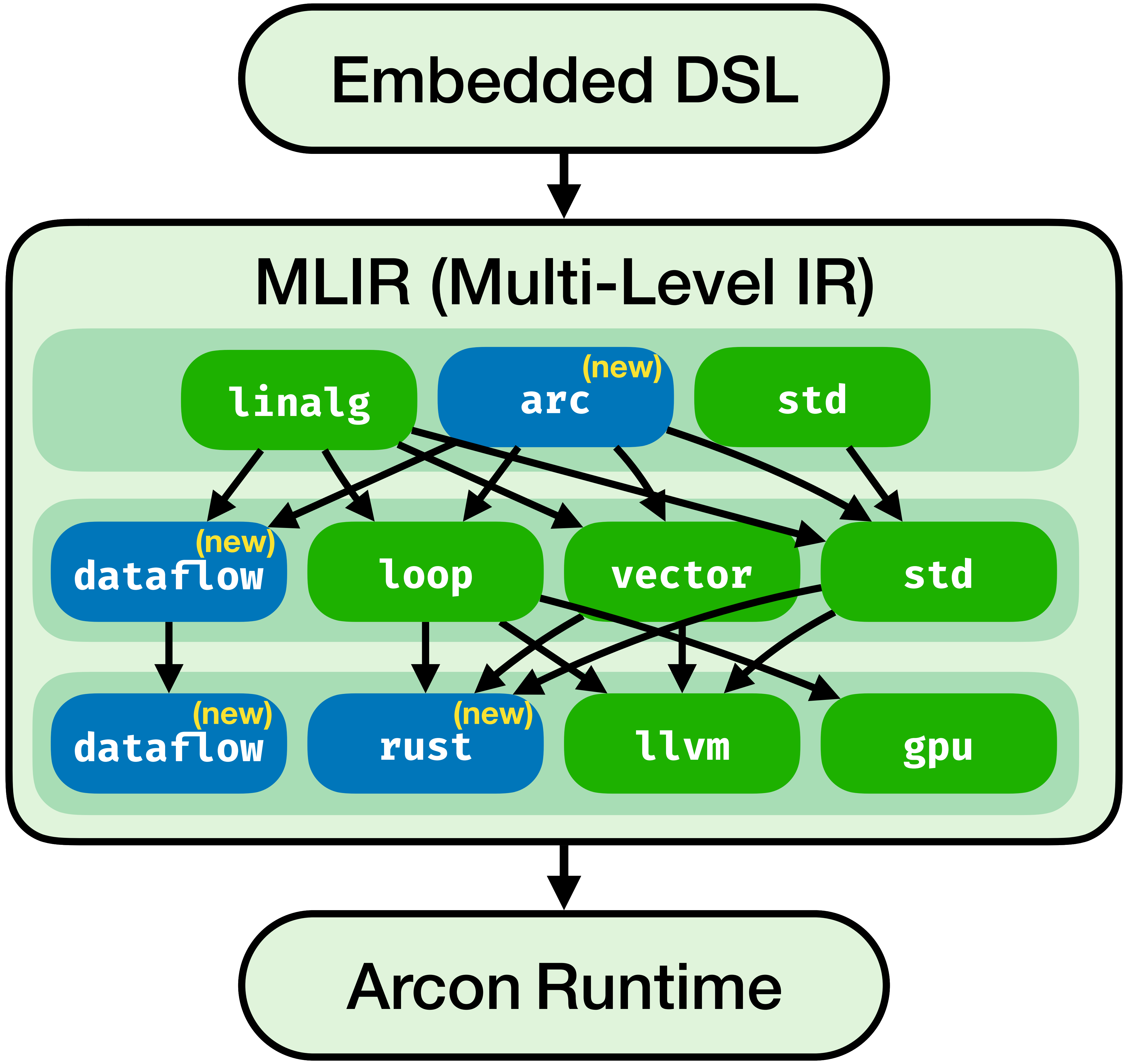
Current approach



Summary

Arc is a dialect in **MLIR** for data analytics that takes inspiration from **Lara**

Current approach

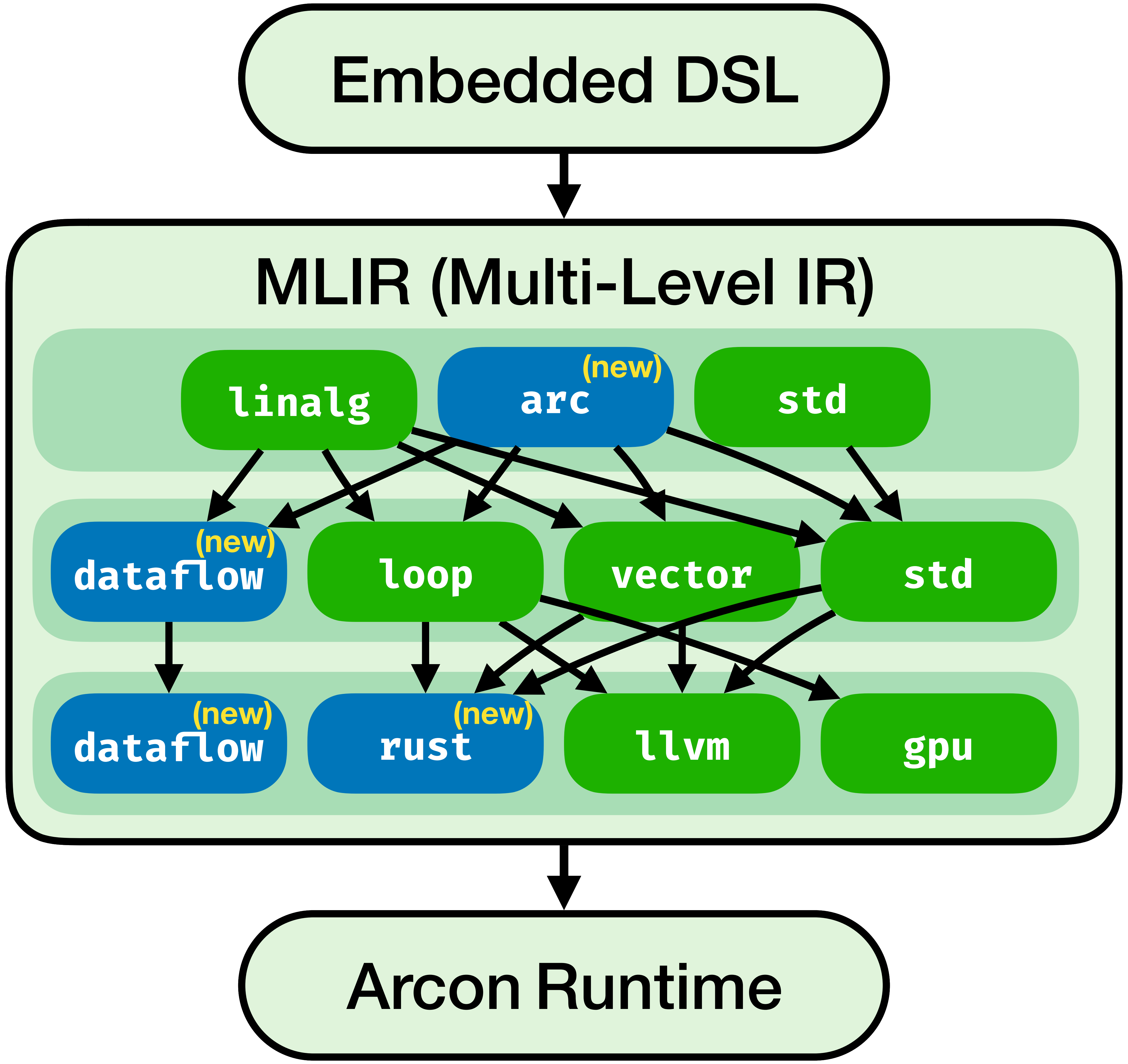


Summary

Arc is a dialect in **MLIR** for data analytics that takes inspiration from **Lara**

Arc aims to **extend Lara**'s model with support for **`Stream`** data types

Current approach



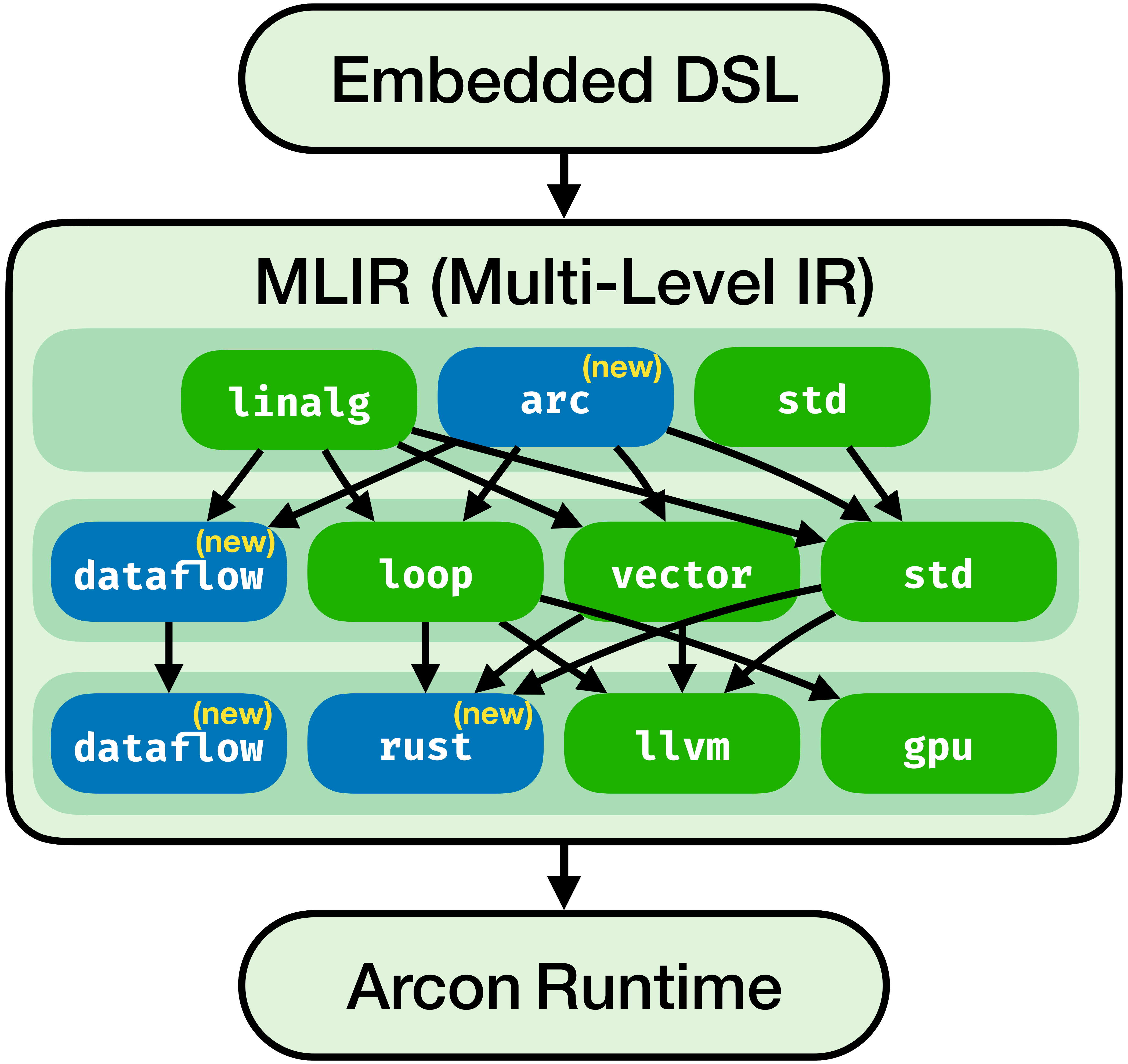
Summary

Arc is a dialect in **MLIR** for data analytics that takes inspiration from **Lara**

Arc aims to **extend Lara**'s model with support for **`Stream`** data types

Through MLIR, Arc can **reuse** existing compiler technology and widen its **scope**

Current approach



Summary

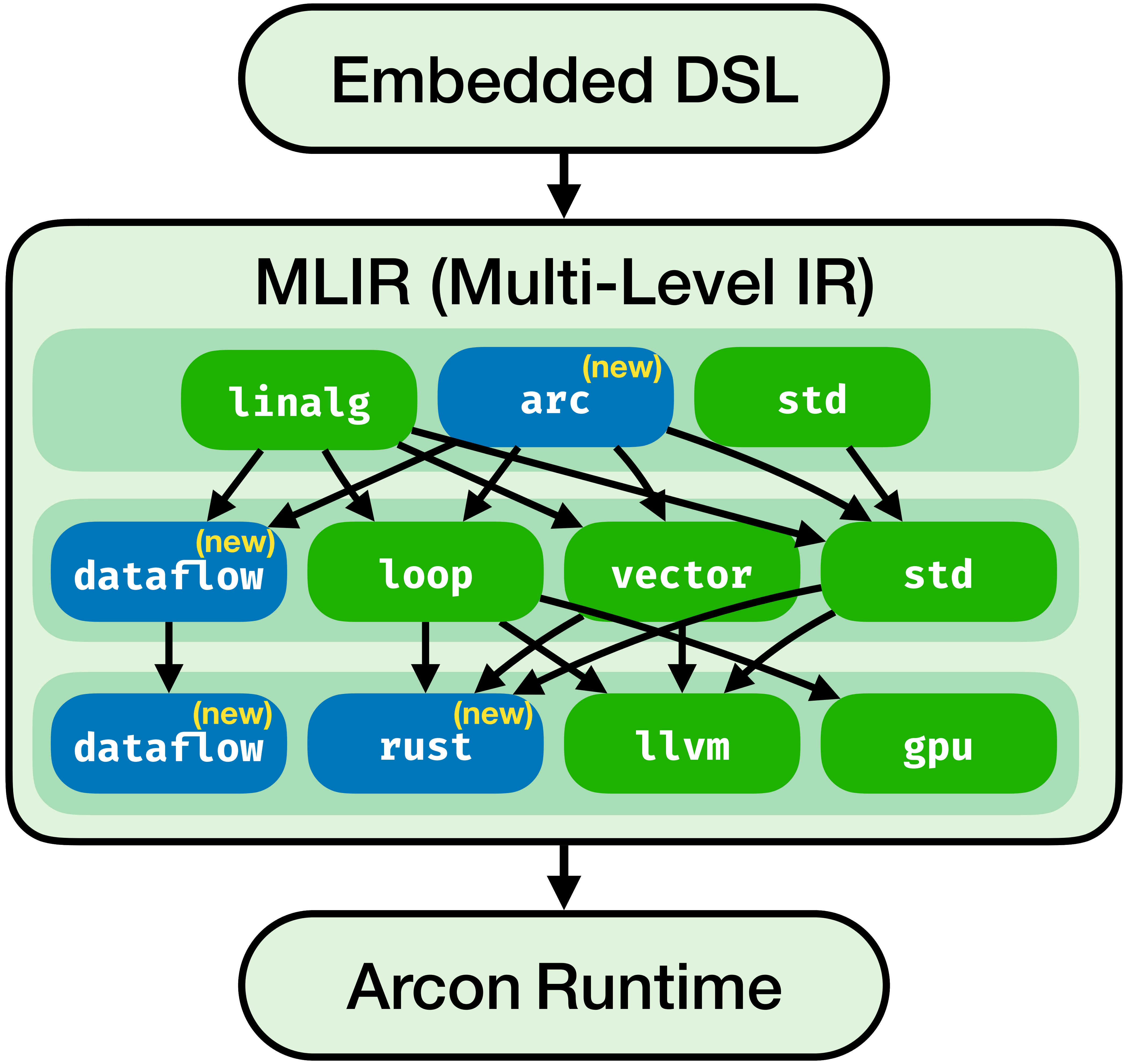
Arc is a dialect in **MLIR** for data analytics that takes inspiration from **Lara**

Arc aims to **extend Lara**'s model with support for **`Stream`** data types

Through MLIR, Arc can **reuse** existing compiler technology and widen its **scope**

Upcoming work: Embedded DSL Design

Current approach



Extra slides

Presenter: Klas Segeljakt <klasseg@kth.se>

Lara's DSL - Example

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

```
Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]
```

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {















}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>









  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
    // Ridge regression
  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
    // Ridge regression
  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

$$\hat{\mathbf{w}}_{ridge} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \left(\mathbf{y}_i - \mathbf{X}_i\mathbf{w}\right)^2 + \lambda \sum_{i=1}^{K} \mathbf{w}_i^2$$

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
    // Ridge regression
  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

$$\hat{\mathbf{w}}_{ridge} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \left(\mathbf{y}_i - \mathbf{X}_i\mathbf{w}\right)^2 + \lambda \sum_{i=1}^{K} \mathbf{w}_i^2$$

$$\hat{\mathbf{w}}_{ridge} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}(\mathbf{X}^{\mathrm{T}}\mathbf{y})$$

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
    // Ridge regression








  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

$$\hat{\mathbf{w}}_{ridge} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \left(\mathbf{y}_i - \mathbf{X}_i \mathbf{w}\right)^2 + \lambda \sum_{i=1}^{K} \mathbf{w}_i^2$$

$$\hat{\mathbf{w}}_{ridge} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I})^{-1}(\mathbf{X}^{\mathrm{T}}\mathbf{y})$$

$$\hat{\mathbf{w}}_{ridge} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I}) \setminus (\mathbf{X}^{\mathrm{T}}\mathbf{y})$$

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
    // Ridge regression
  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

$$\hat{\mathbf{w}}_{ridge} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \left(\mathbf{y}_i - \mathbf{X}_i\mathbf{w}\right)^2 + \lambda \sum_{i=1}^{K} \mathbf{w}_i^2$$

$$\left.\begin{aligned}\hat{\mathbf{w}}_{ridge} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}(\mathbf{X}^{\mathrm{T}}\mathbf{y}) \\ \hat{\mathbf{w}}_{ridge} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I}) \setminus (\mathbf{X}^{\mathrm{T}}\mathbf{y})\end{aligned}\right\{$$

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
     // Ridge regression







  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

$$\hat{\mathbf{w}}_{ridge} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \left(\mathbf{y}_i - \mathbf{X}_i \mathbf{w}\right)^2 + \lambda \sum_{i=1}^{K} \mathbf{w}_i^2$$

$$\left.\begin{aligned} \hat{\mathbf{w}}_{ridge} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I})^{-1}(\mathbf{X}^{\mathrm{T}}\mathbf{y}) \\ \hat{\mathbf{w}}_{ridge} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I}) \setminus (\mathbf{X}^{\mathrm{T}}\mathbf{y}) \end{aligned}\right\}$$

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
    // Ridge regression
    val reg = Matrix.eye(X_train.nCols) * lambda
    val XtX = X_train.t ** X_train + reg
    val Xty = X_train.t ** y_train
    val w   = XtX \ Xty



  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

$$\hat{\mathbf{w}}_{ridge} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \left(\mathbf{y}_i - \mathbf{X}_i\mathbf{w}\right)^2 + \lambda \sum_{i=1}^{K} \mathbf{w}_i^2$$

$$\left.\begin{aligned} \hat{\mathbf{w}}_{ridge} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}(\mathbf{X}^{\mathrm{T}}\mathbf{y}) \\ \hat{\mathbf{w}}_{ridge} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I}) \setminus (\mathbf{X}^{\mathrm{T}}\mathbf{y}) \end{aligned}\right\}$$

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
    // Ridge regression
    val reg = Matrix.eye(X_train.nCols) * lambda
    val XtX = X_train.t ** X_train + reg
    val Xty = X_train.t ** y_train
    val w   = XtX \ Xty
    // Calculate mean squared error on test set


  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

$$\hat{\mathbf{w}}_{ridge} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \left(\mathbf{y}_i - \mathbf{X}_i \mathbf{w}\right)^2 + \lambda \sum_{i=1}^{K} \mathbf{w}_i^2$$

$$\left.\begin{aligned} \hat{\mathbf{w}}_{ridge} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I})^{-1}(\mathbf{X}^{\mathrm{T}}\mathbf{y}) \\ \hat{\mathbf{w}}_{ridge} &= (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I}) \setminus (\mathbf{X}^{\mathrm{T}}\mathbf{y}) \end{aligned}\right\\}$$

$$\textbf{MSE} = \frac{1}{N} \sum_{i=1}^{N} \left(\mathbf{y}_i - \mathbf{X}_i \mathbf{w}\right)^2$$

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
    // Ridge regression
    val reg = Matrix.eye(X_train.nCols) * lambda
    val XtX = X_train.t ** X_train + reg
    val Xty = X_train.t ** y_train
    val w   = XtX \ Xty
    // Calculate mean squared error on test set


  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

$$\hat{\mathbf{w}}_{ridge} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \left(\mathbf{y}_i - \mathbf{X}_i\mathbf{w}\right)^2 + \lambda \sum_{i=1}^{K} \mathbf{w}_i^2$$

$$\hat{\mathbf{w}}_{ridge} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}(\mathbf{X}^{\mathrm{T}}\mathbf{y})$$

$$\hat{\mathbf{w}}_{ridge} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I}) \setminus (\mathbf{X}^{\mathrm{T}}\mathbf{y})$$

$$\textbf{MSE} = \frac{1}{N} \sum_{i=1}^{N} \left(\mathbf{y}_i - \mathbf{X}_i\mathbf{w}\right)^2$$

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
    // Ridge regression
    val reg = Matrix.eye(X_train.nCols) * lambda
    val XtX = X_train.t ** X_train + reg
    val Xty = X_train.t ** y_train
    val w   = XtX \ Xty
    // Calculate mean squared error on test set


  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

Lara's DSL - Example

Description

Based on **numerical** and **categorical** data, train a **Ridge Regression** model with **3-fold cross-validation**, and **Mean Squared Error** as the loss function, to the predict number of future clicks on advertisements

$$\hat{\mathbf{w}}_{ridge} = \underset{\mathbf{w}}{\arg\min} \sum_{i=1}^{N} (\mathbf{y}_i - \mathbf{X}_i\mathbf{w})^2 + \lambda \sum_{i=1}^{K} \mathbf{w}_i^2$$

$$\hat{\mathbf{w}}_{ridge} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}(\mathbf{X}^{\mathrm{T}}\mathbf{y})$$

$$\hat{\mathbf{w}}_{ridge} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda\mathbf{I}) \setminus (\mathbf{X}^{\mathrm{T}}\mathbf{y})$$

$$\textbf{MSE} = \frac{1}{N} \sum_{i=1}^{N} (\mathbf{y}_i - \mathbf{X}_i\mathbf{w})^2$$

Input data: [L,N,N,N,N,N,N,N,N,N,N,C,C,C,C,C]

```
// Column 0 contains the target variable, columns 1-10 contain
// numerical and columns 11-15 contain categorical features
val dataset  = readAndClean("/path/to/data")
val encoded  = dummyEncode(dataset, 11 to 15)
val vectors  = concatNumericalFeatures(encoded, 1 to 10)
val features = concatVectors(vectors)
// y = 0: extract 1st column as target vector y
val (M, y)   = Matrix(features, y = 0)
val X        = Matrix.normalize(M, 1 to 10)
// Grid search over hyperparameter candidates
val regCandidates: Seq[Double] = // ...
for (lambda <- regCandidates) {
  // 3-fold cross-validation for the hyperparameter lambda
  val errors = ML.crossValidate(3, X, y) {
    (X_train, X_test, y_train, y_test) =>
    // Ridge regression
    val reg = Matrix.eye(X_train.nCols) * lambda
    val XtX = X_train.t ** X_train + reg
    val Xty = X_train.t ** y_train
    val w   = XtX \ Xty
    // Calculate mean squared error on test set
    val residuals = y_test - (X_test ** w)
    residuals.map(r => r * r).agg(_ + _) / y_test.size
  }
  // Print mean error for chosen hyperparameter
  println(errors.sum / k)
}
```

References

[1] Beam.apache.org. (2020). *Apache **Beam***. [online] Available at: https://beam.apache.org/ [Accessed 2 Mar. 2020].

[2] Walt, S.V.D., Colbert, S.C. and Varoquaux, G., 2011. The **NumPy** array: a structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), pp.22-30.

[3] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F. and Amarasinghe, S., 2013. **Halide**: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, *48*(6), pp.519-530.

[4] Date, C.J. and Darwen, H., 1993. *A Guide to the **SQL** Standard* (Vol. 3). Reading: Addison-wesley.

[5] Arasu, A., Babu, S. and Widom, J., 2006. The **CQL** continuous query language: semantic foundations and query execution. *The VLDB Journal*, *15*(2), pp.121-142.

[6] Kunft, A., Katsifodimos, A., Schelter, S., Breß, S., Rabl, T. and Markl, V., 2019. An intermediate representation for optimizing machine learning pipelines. *Proceedings of the VLDB Endowment*, *12*(11), pp.1553-1567.

[7] Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N. and Czajkowski, G., 2010, June. **Pregel**: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (pp. 135-146).

[8] Iyer, A.P., Pu, Q., Patel, K., Gonzalez, J.E. and Stoica, I., 2019. *TEGRA: Efficient ad-hoc analytics on time-evolving graphs*. Technical report.

[9] Venkataraman, S., Bodzsar, E., Roy, I., AuYoung, A. and Schreiber, R.S., 2013, April. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems* (pp. 197-210).

[10] Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P. and Taylor, A., 2018, May. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data* (pp. 1433-1445).