DLVM

DLVM

A Modern Compiler Framework for Neural Network DSLs

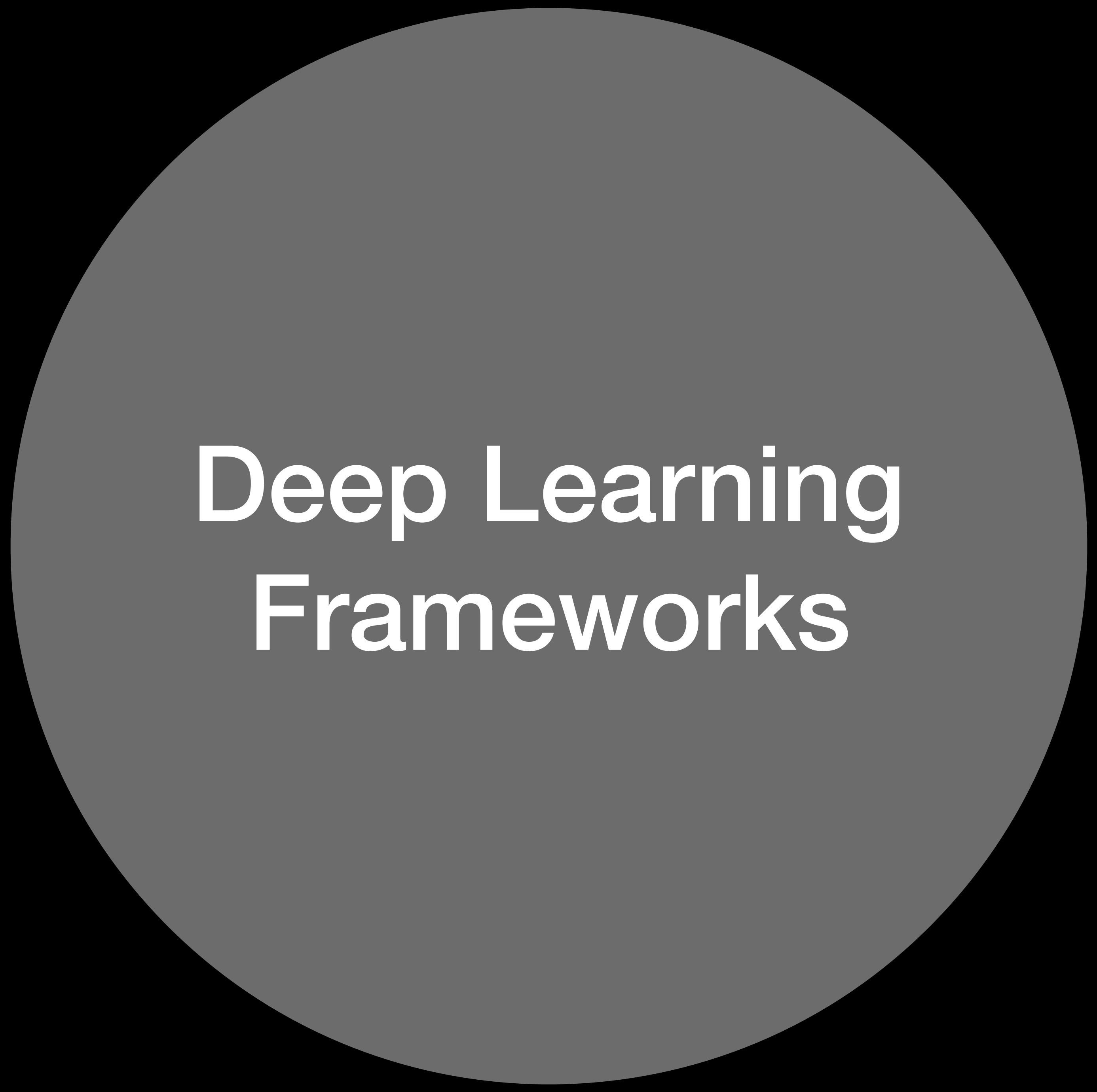
DLVM

A Modern Compiler Framework for Neural Network DSLs

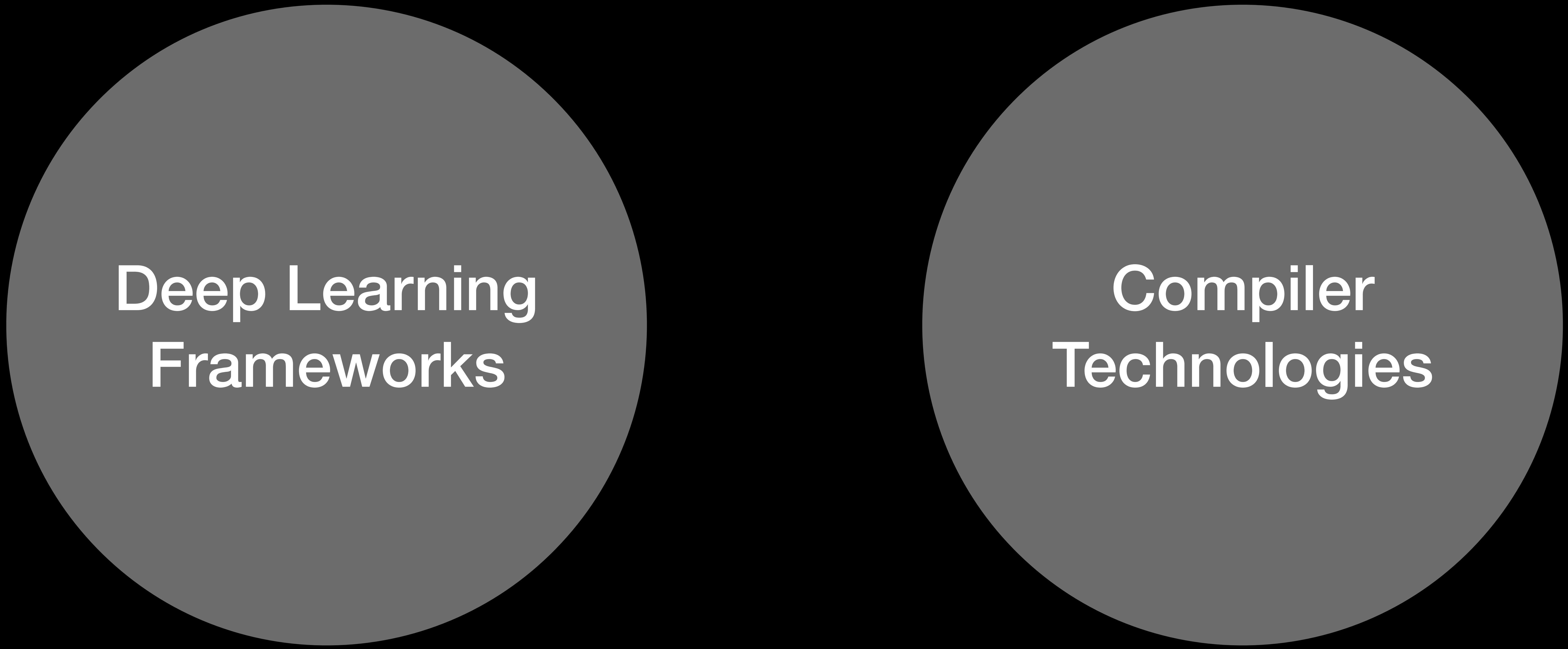
Richard Wei **Lane Schwartz** **Vikram Adve**
University of Illinois at Urbana-Champaign

1-2 years ago

1-2 years ago



1-2 years ago



Today

Compiler
Technologies

Today

Deep Learning
Compiler Technologies

Today

Deep Learning
Compiler Technologies

- Latte.jl
- XLA
- NNVM + TVM
- ONNX
- PyTorch JIT
- DLVM

Neural networks are programs

Typing

Compute

Optimizations

Static Analysis

Intermediate Representation

Neural networks are programs

Control Flow

Automatic Differentiation

Auto Vectorization

A New Compiler Problem

A New Compiler Problem

- Programs, not just a data flow graph

A New Compiler Problem

- Programs, not just a data flow graph
- Type safety

A New Compiler Problem

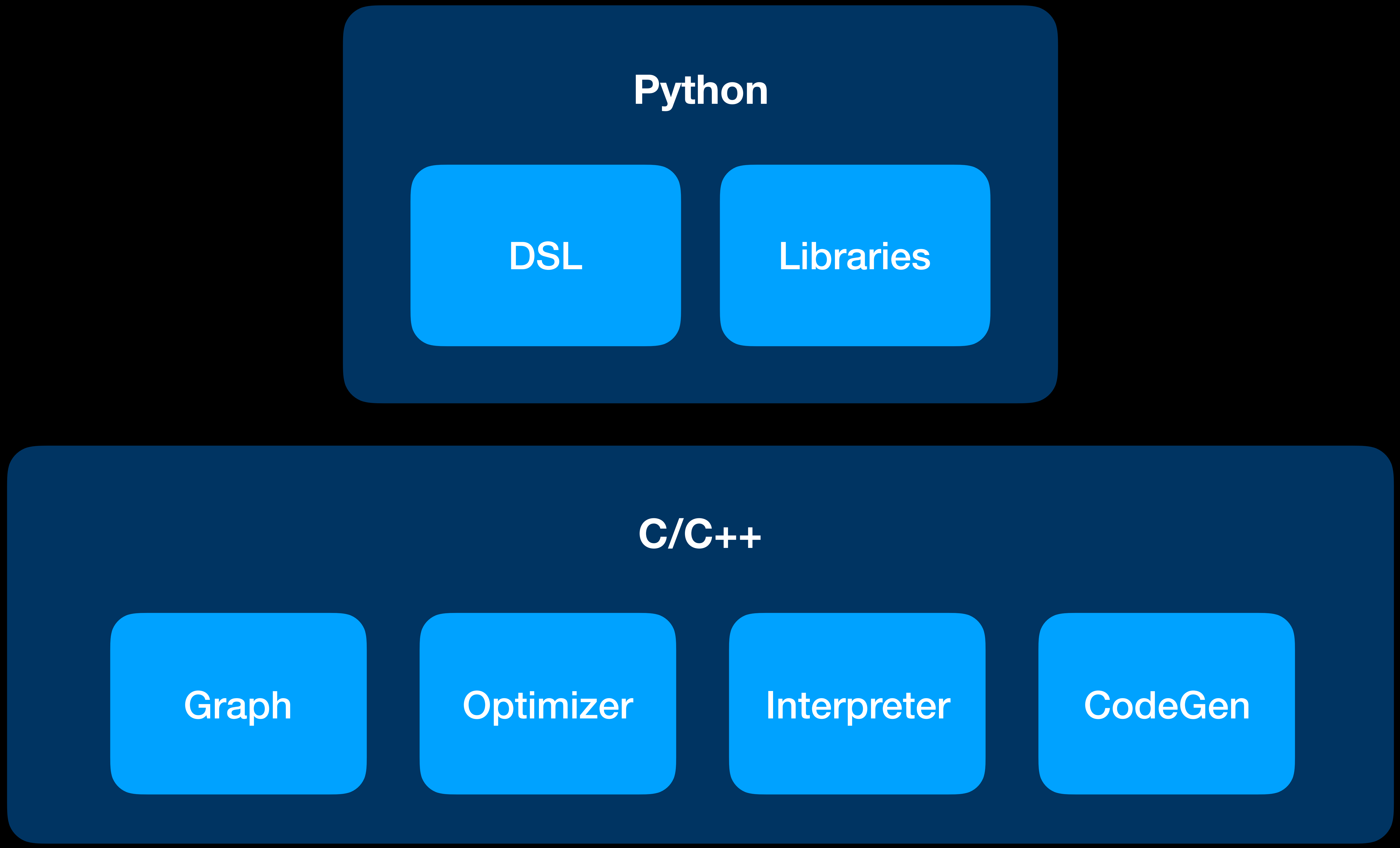
- Programs, not just a data flow graph
- Type safety
- Ahead-of-time AD

A New Compiler Problem

- Programs, not just a data flow graph
- Type safety
- Ahead-of-time AD
- Code generation

A New Compiler Problem

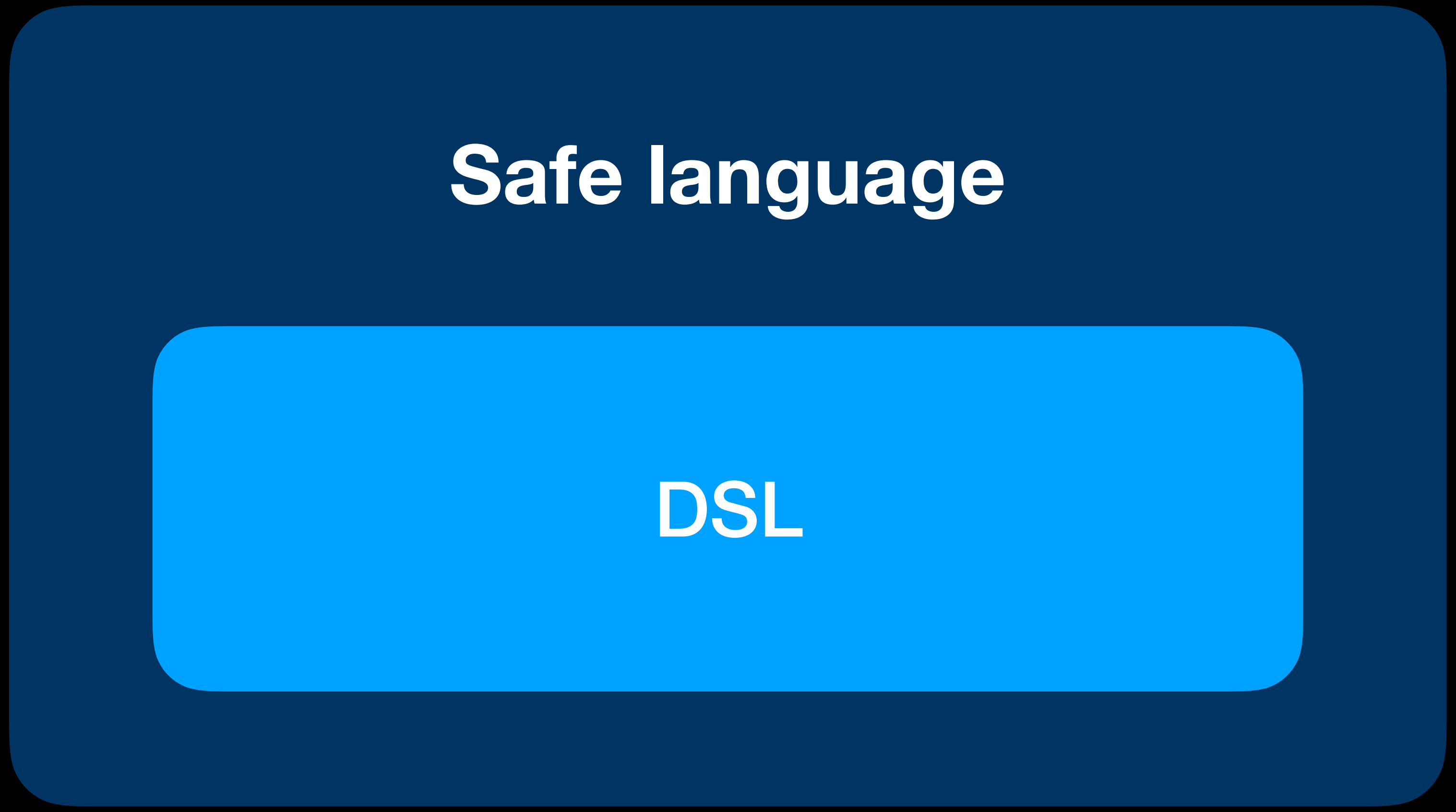
- Programs, not just a data flow graph
- Type safety
- Ahead-of-time AD
- Code generation
- Lightweight installation


Python
DSL
Libraries
C/C++
Graph
Optimizer
Interpreter
CodeGen

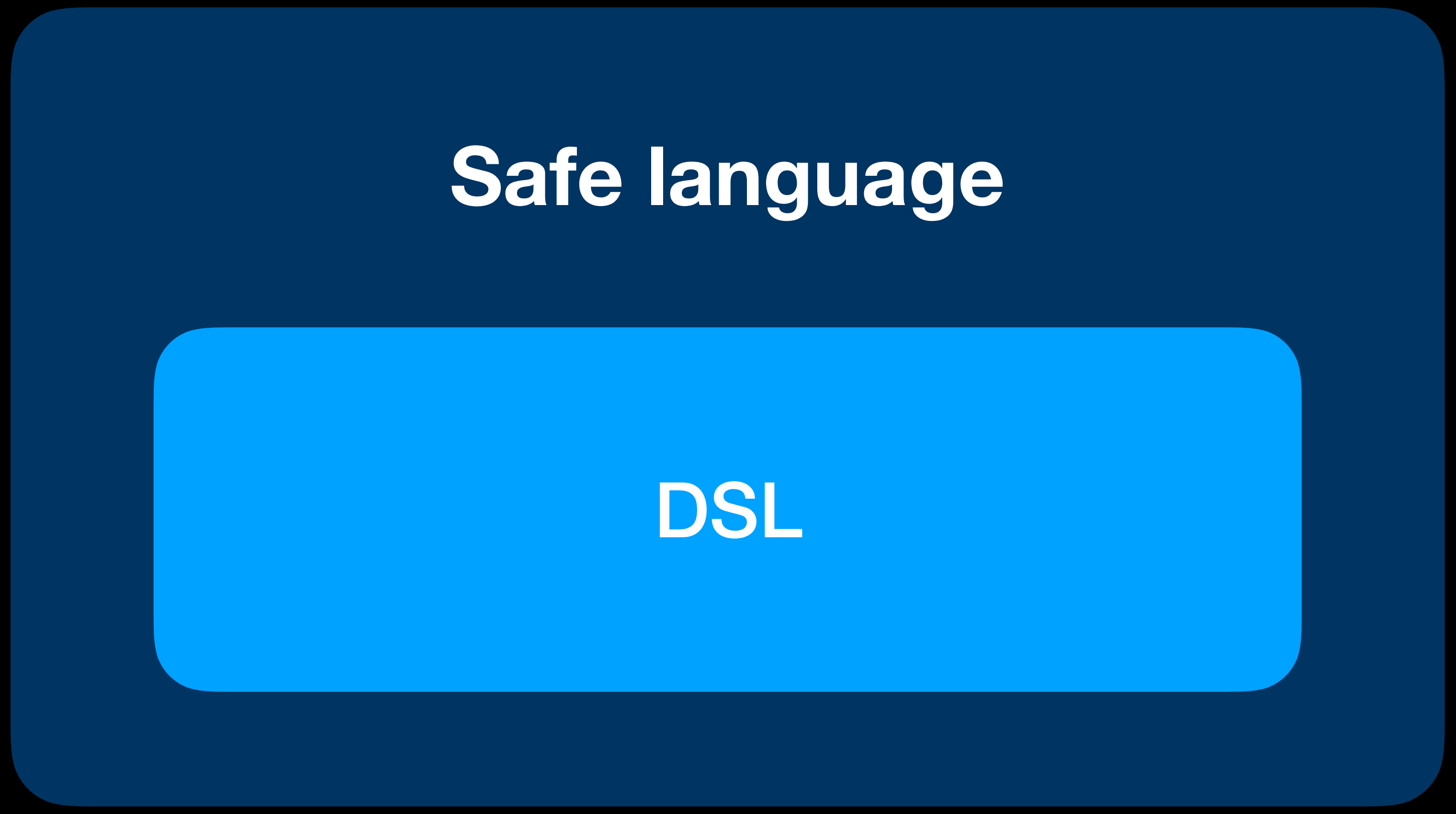
Python

Python

Safe language

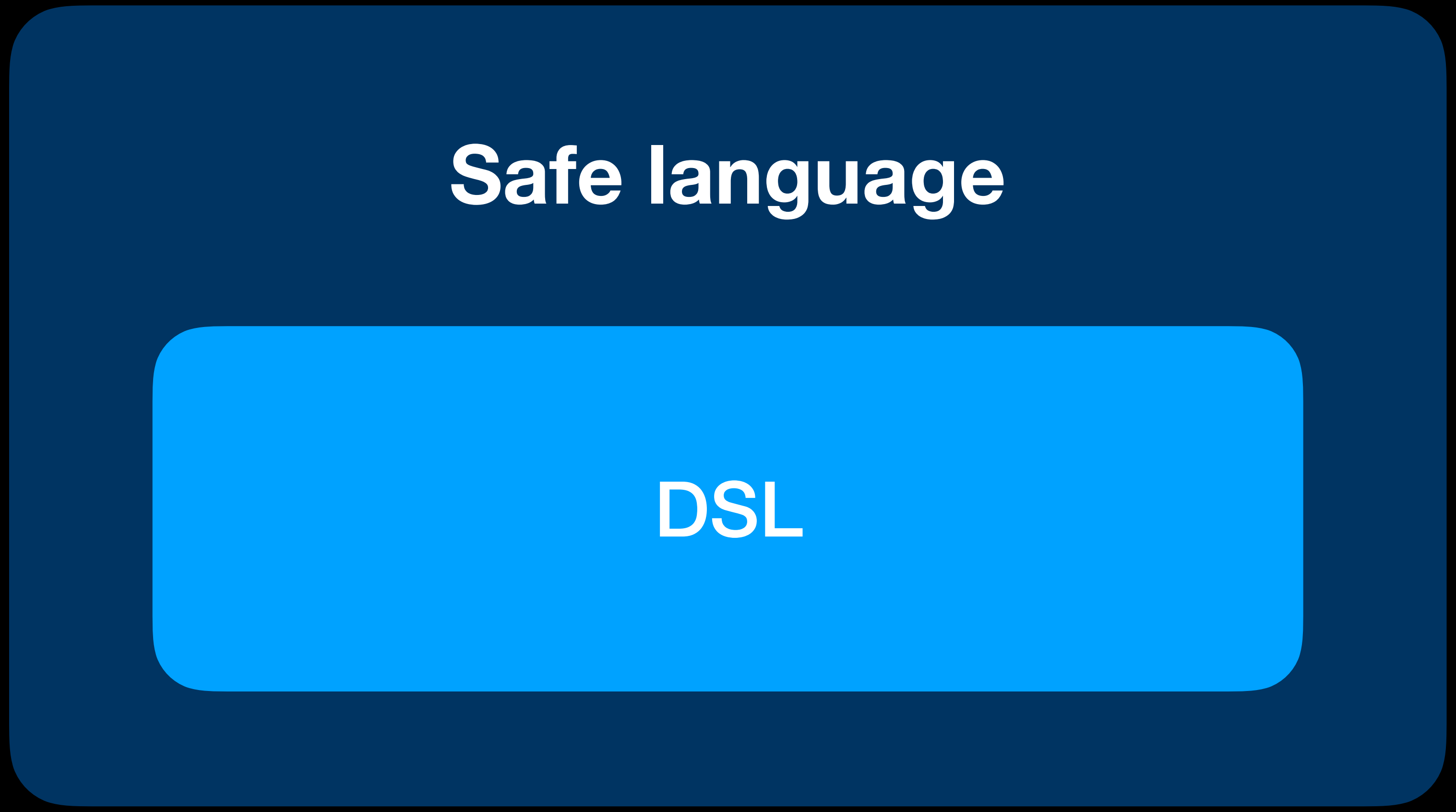


* Rompf, Tiark, and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs, 2010



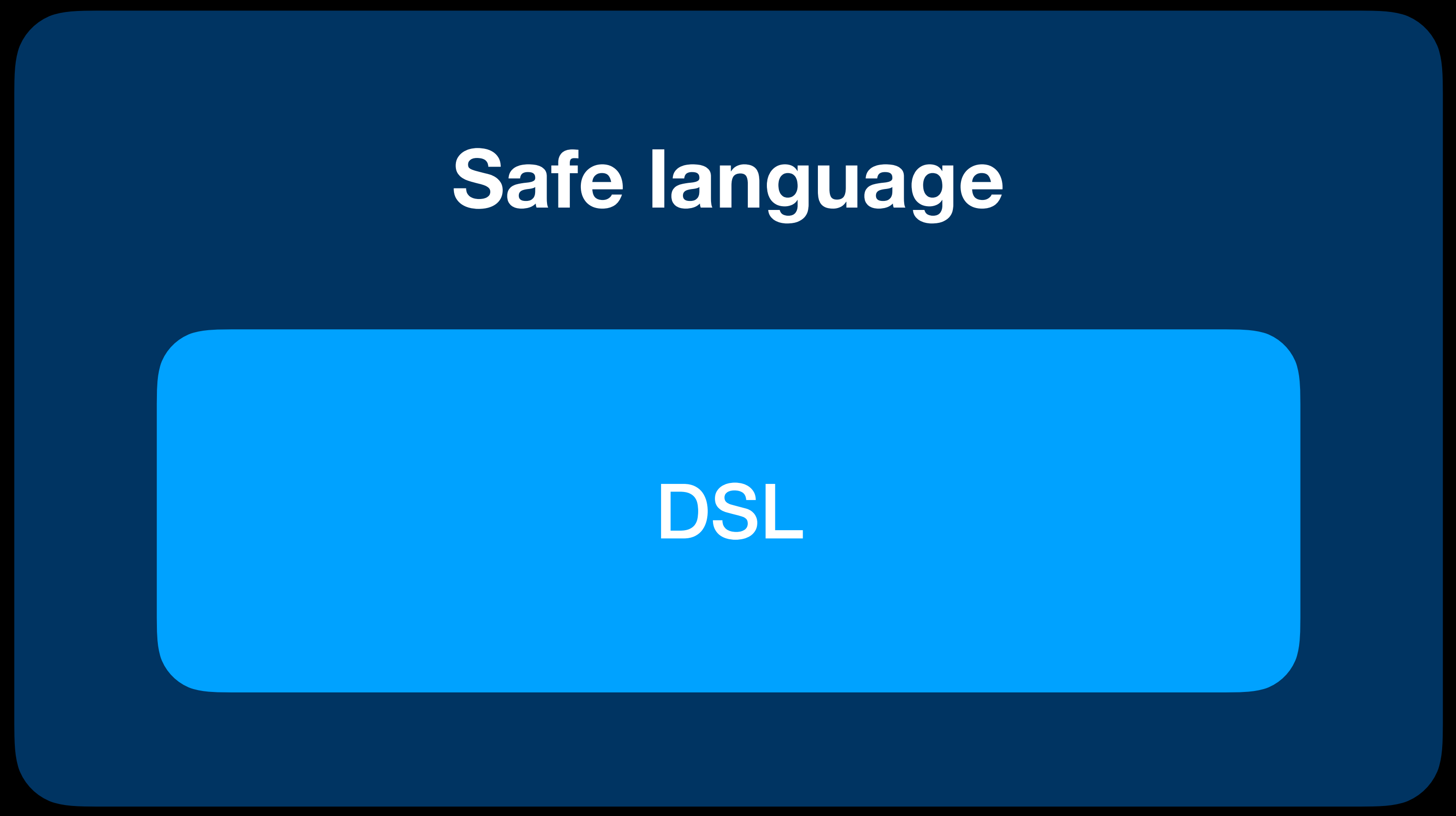
- NN as a host language function

* Rompf, Tiark, and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs, 2010



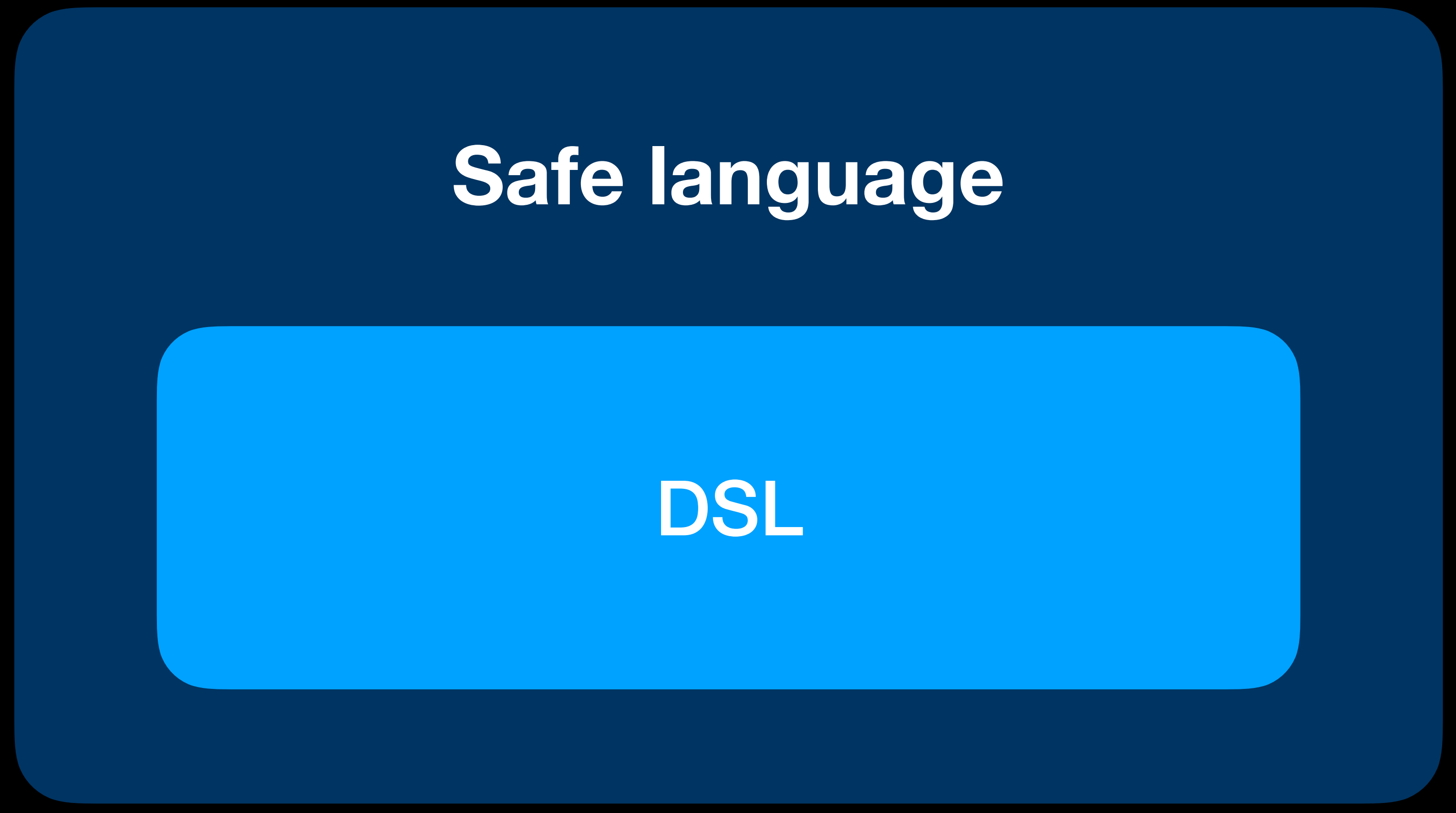
- NN as a host language function
- Type safety

* Rompf, Tiark, and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs, 2010



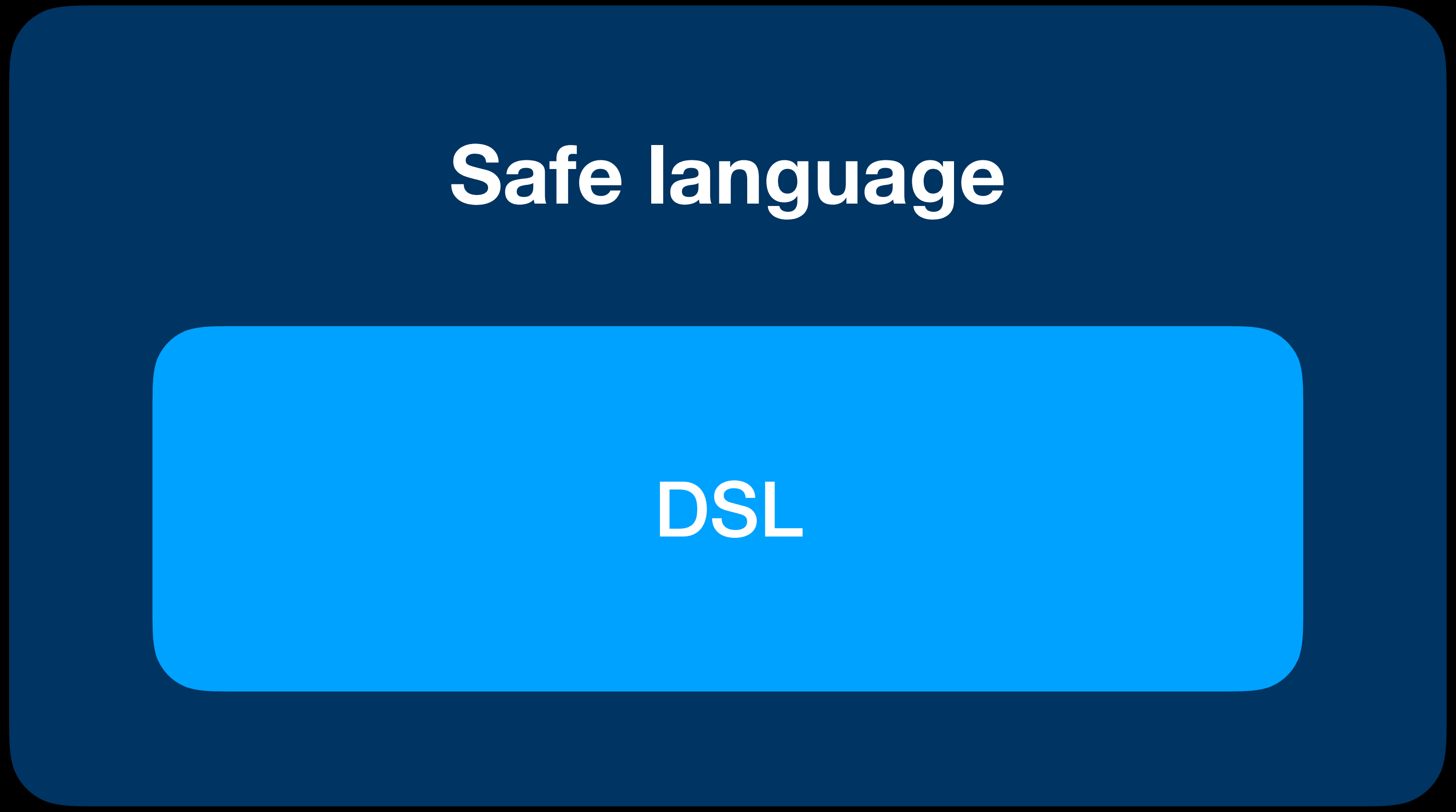
- NN as a host language function
- Type safety
- Naturalness

* Rompf, Tiark, and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs, 2010



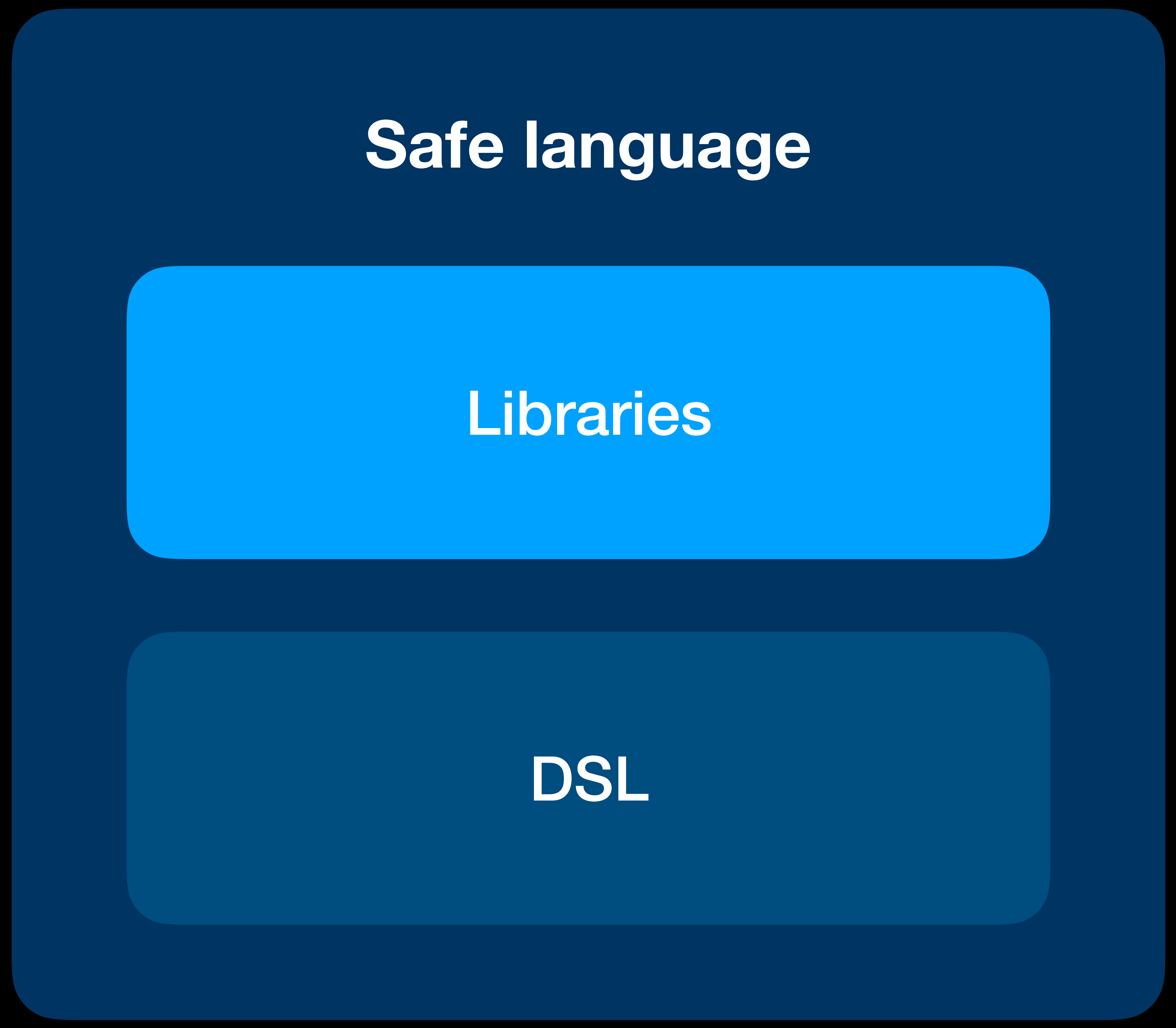
- NN as a host language function
- Type safety
- Naturalness
  - Lightweight modular staging*

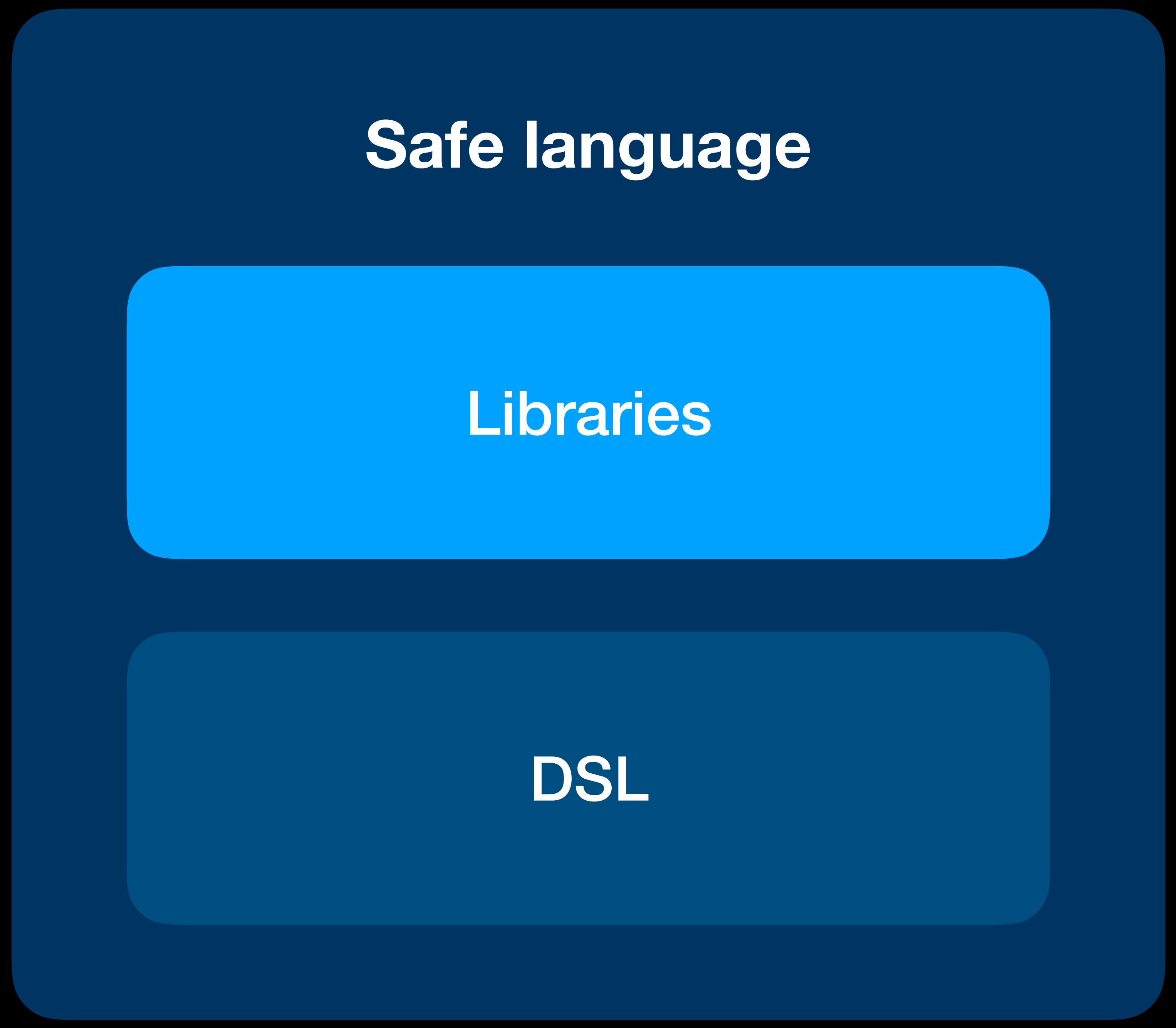
* Rompf, Tiark, and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs, 2010



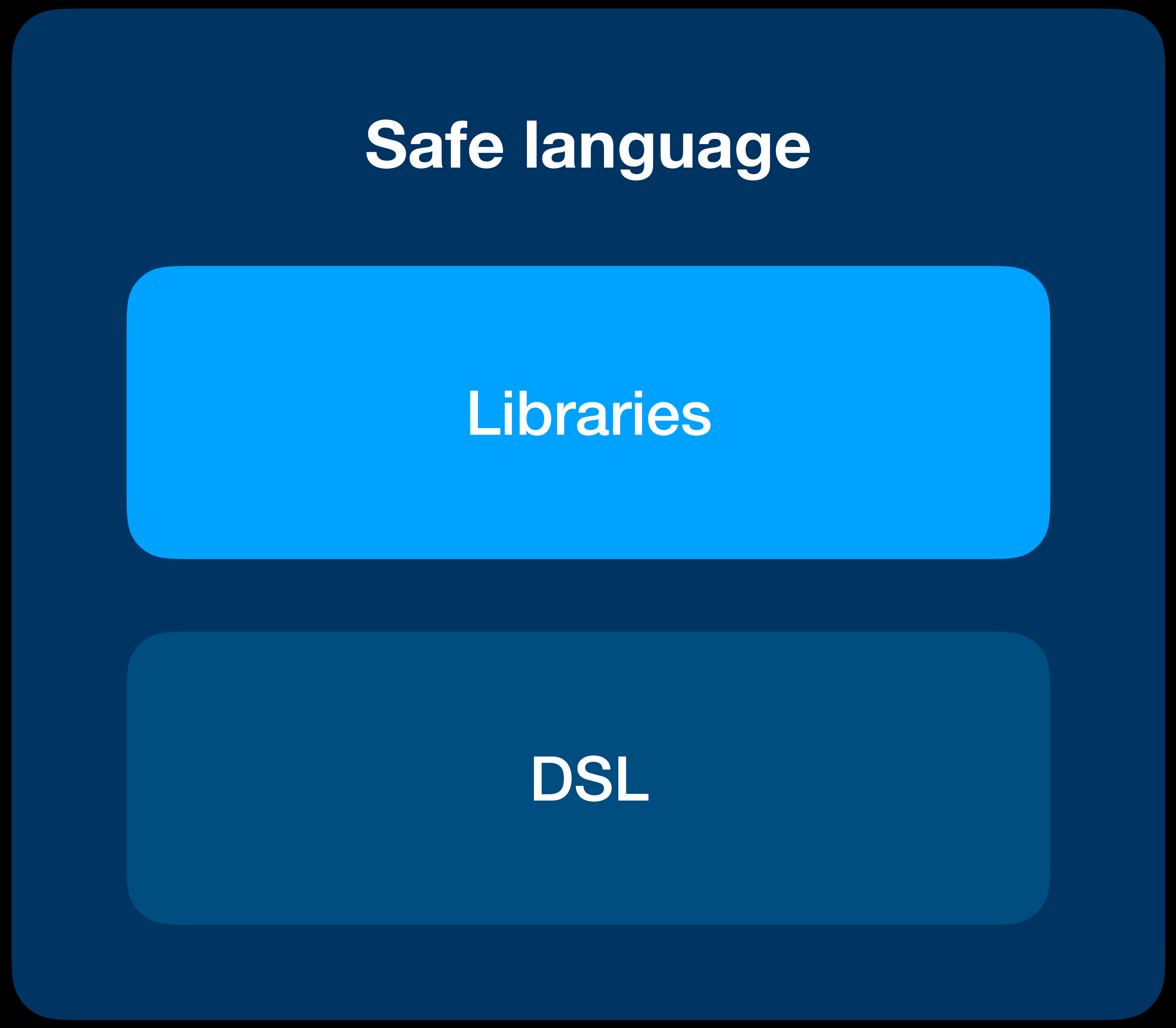
- NN as a host language function
- Type safety
- Naturalness
  - Lightweight modular staging*
  - Compiler magic

* Rompf, Tiark, and Martin Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs, 2010


Safe language
Libraries
DSL



- Trainer



- Trainer
- Layers

- Trainer
- Layers
- Application API

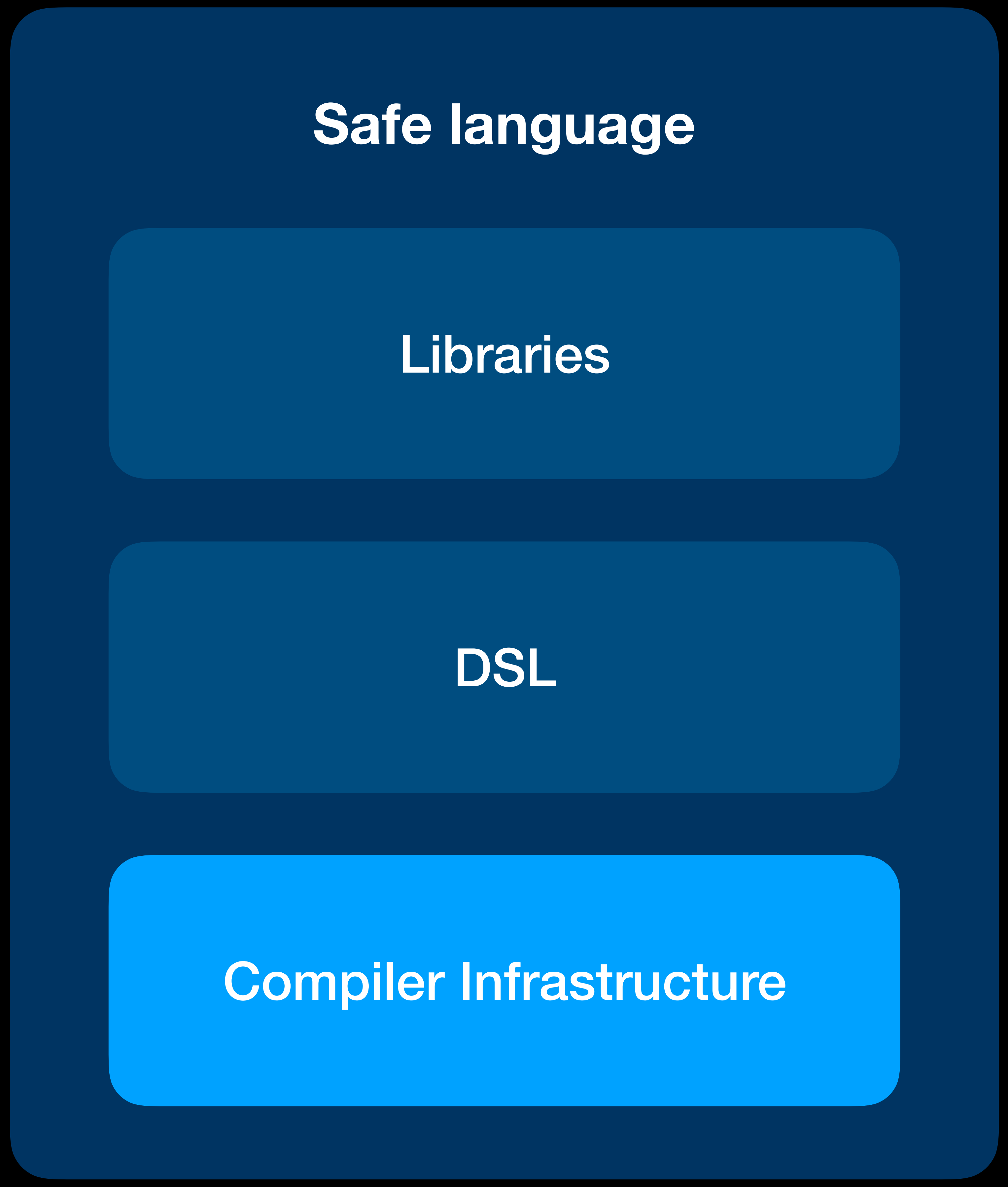

Safe language
Libraries
DSL
Compiler Infrastructure

- Generic linear algebra IR

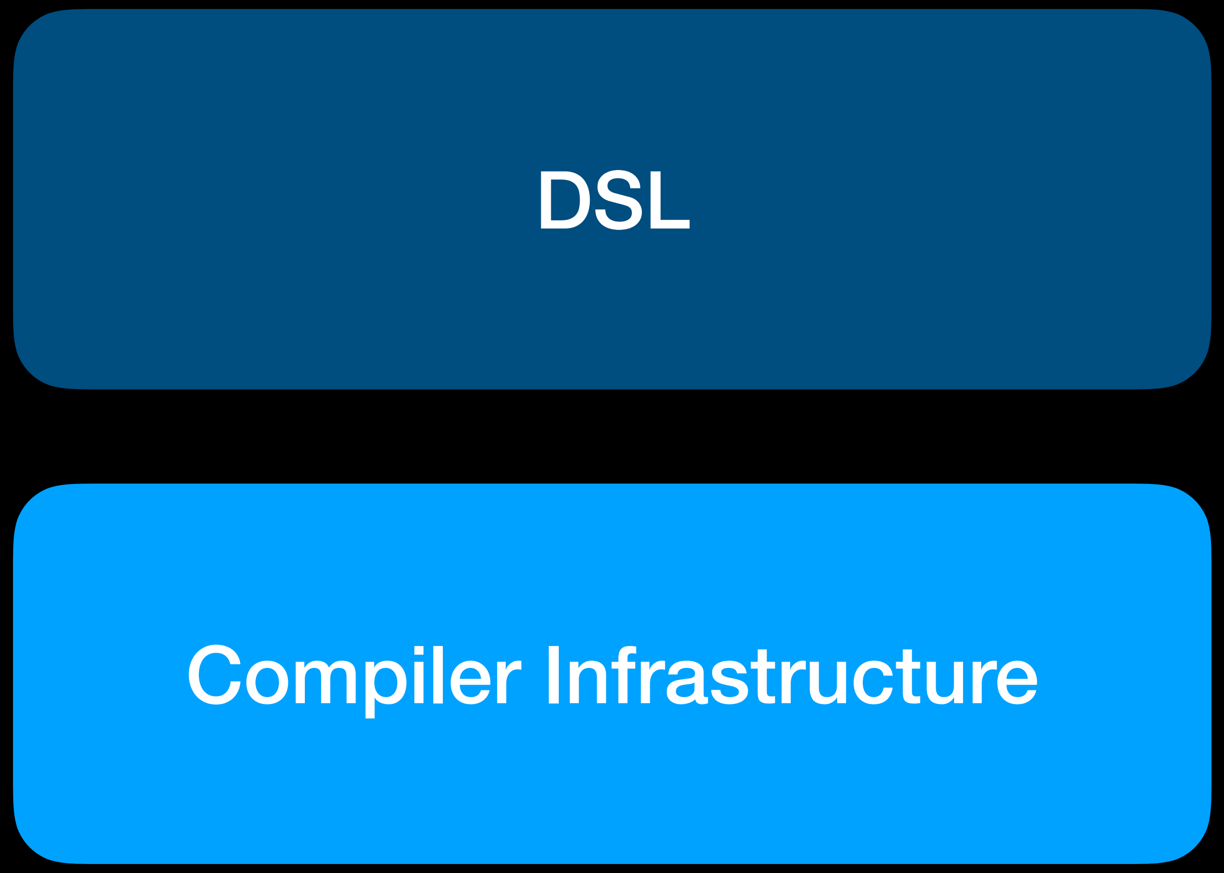
- Generic linear algebra IR
- Automatic differentiation

- Generic linear algebra IR
- Automatic differentiation
- Optimizations

- Generic linear algebra IR
- Automatic differentiation
- Optimizations
- Code generation



- Generic linear algebra IR
- Automatic differentiation
- Optimizations
- Code generation
- Runtime


DSL
Compiler Infrastructure

DLVM

DLVM

- Linear algebra IR

- Linear algebra IR
- Framework for building DSLs

DLVM

- Linear algebra IR
- Framework for building DSLs
- Automatic backpropagator

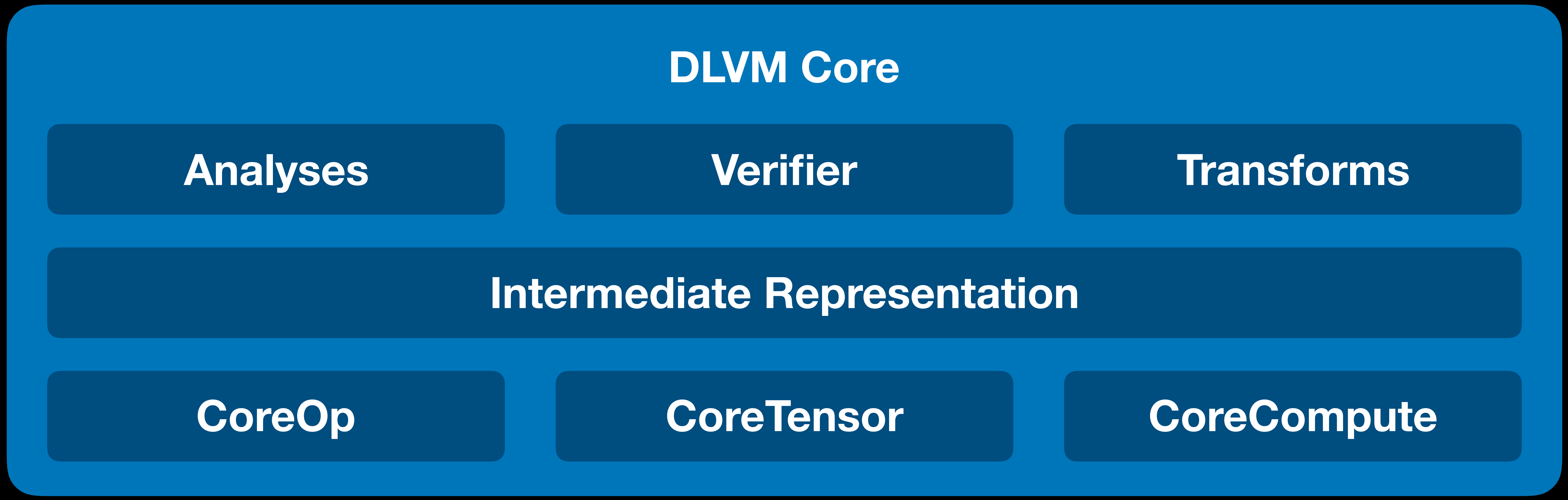
DLVM

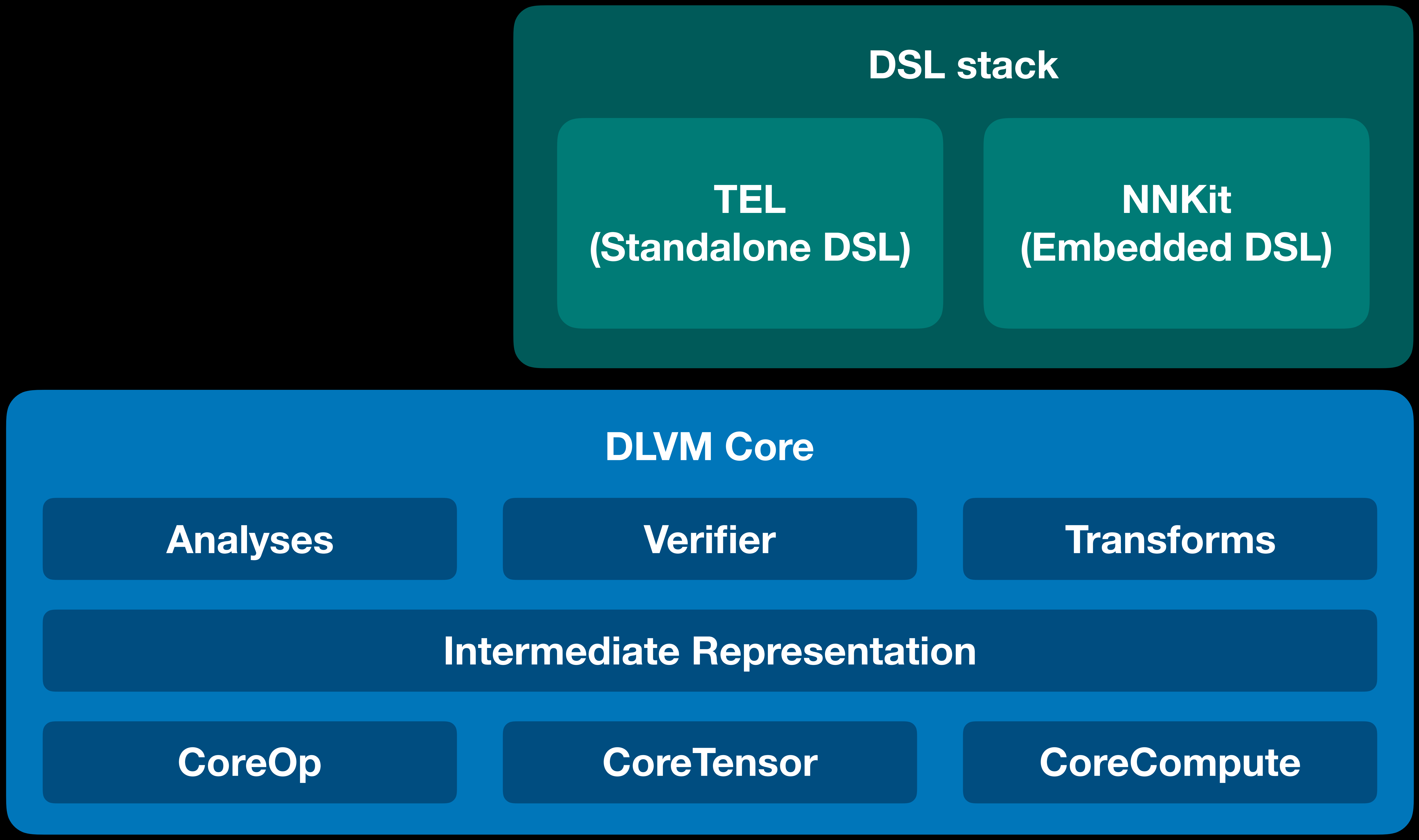
DLVM

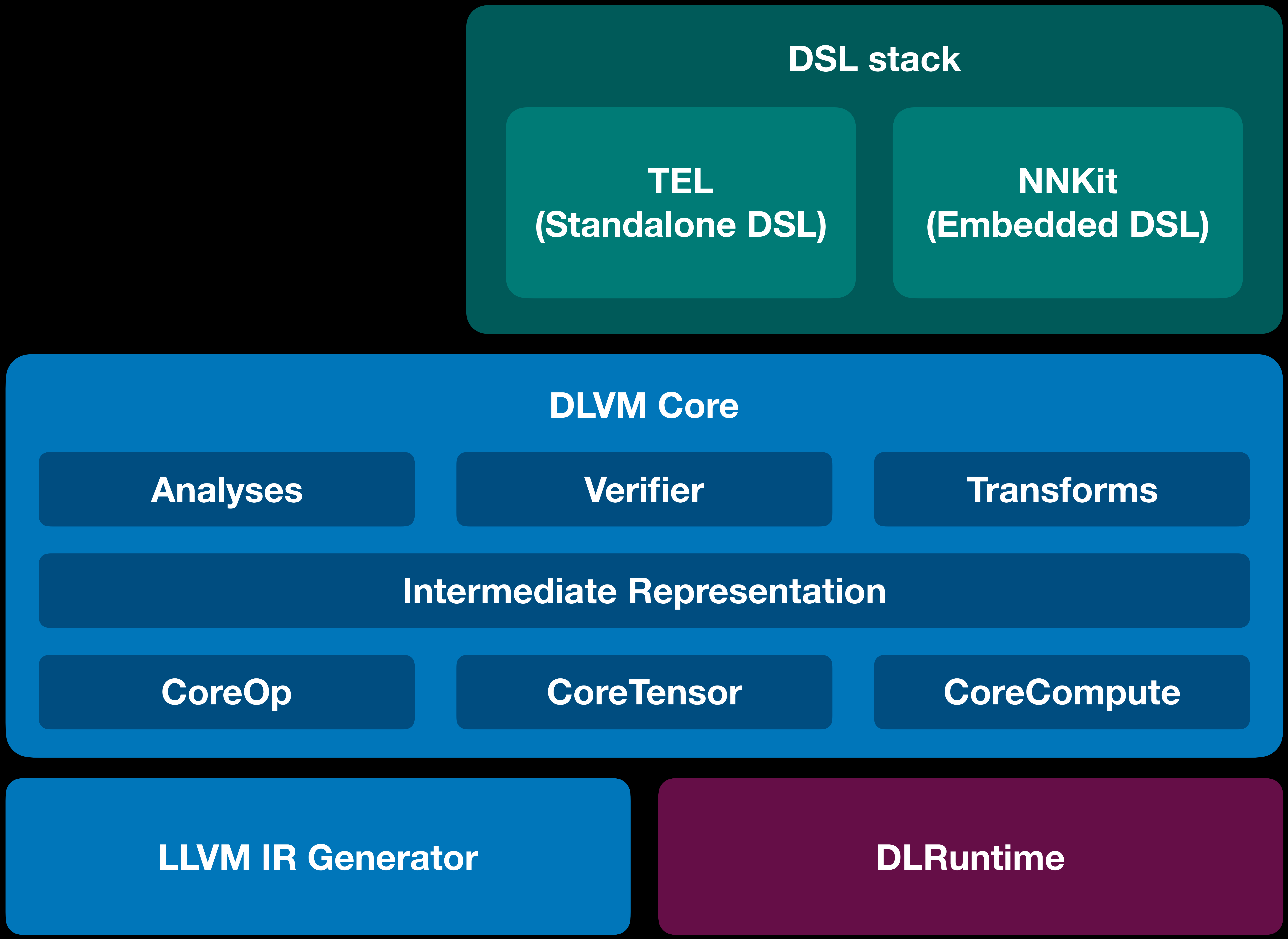
- Linear algebra IR
- Framework for building DSLs
- Automatic backpropagator
- Multi-stage optimizer

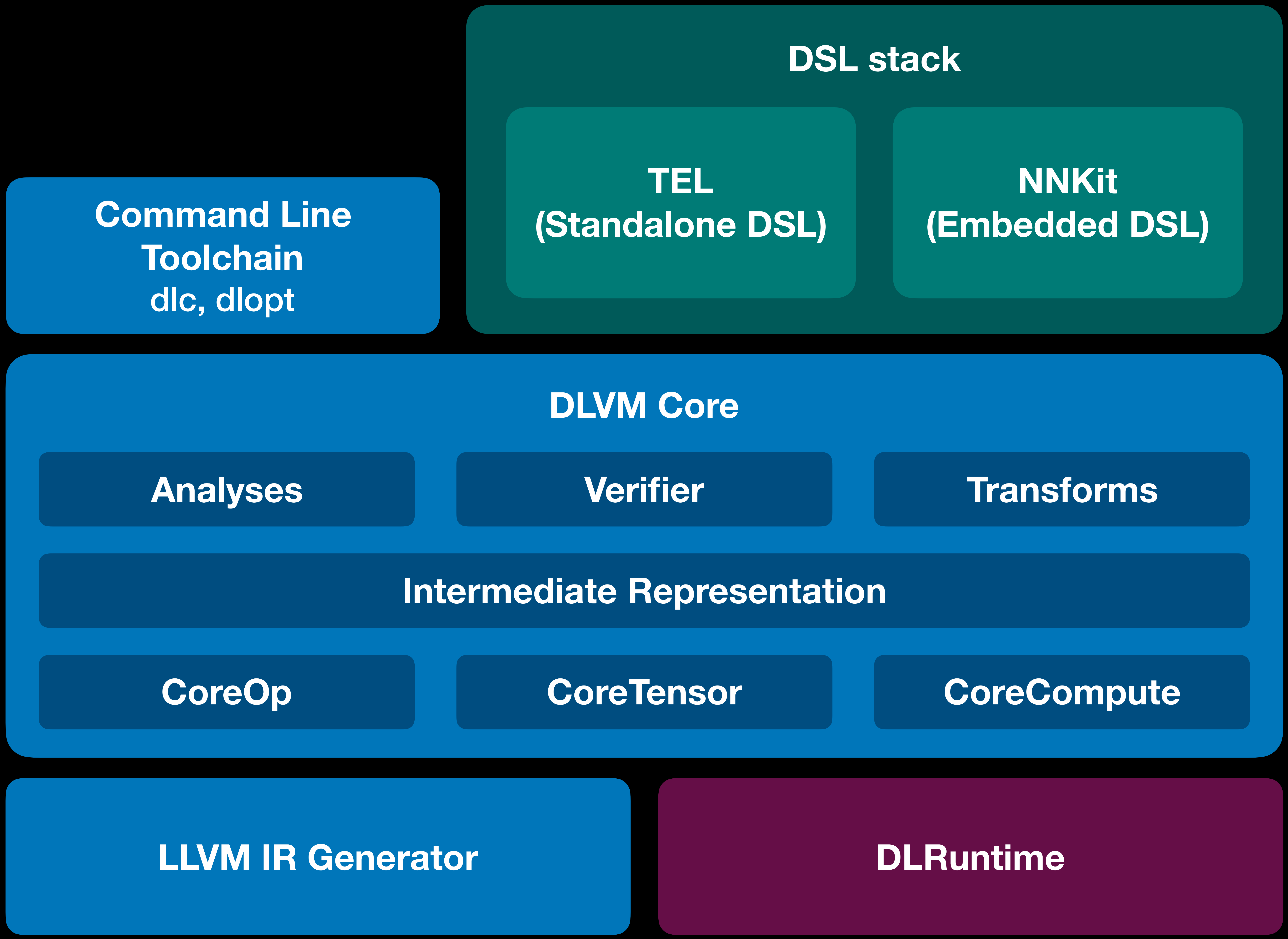
DLVM

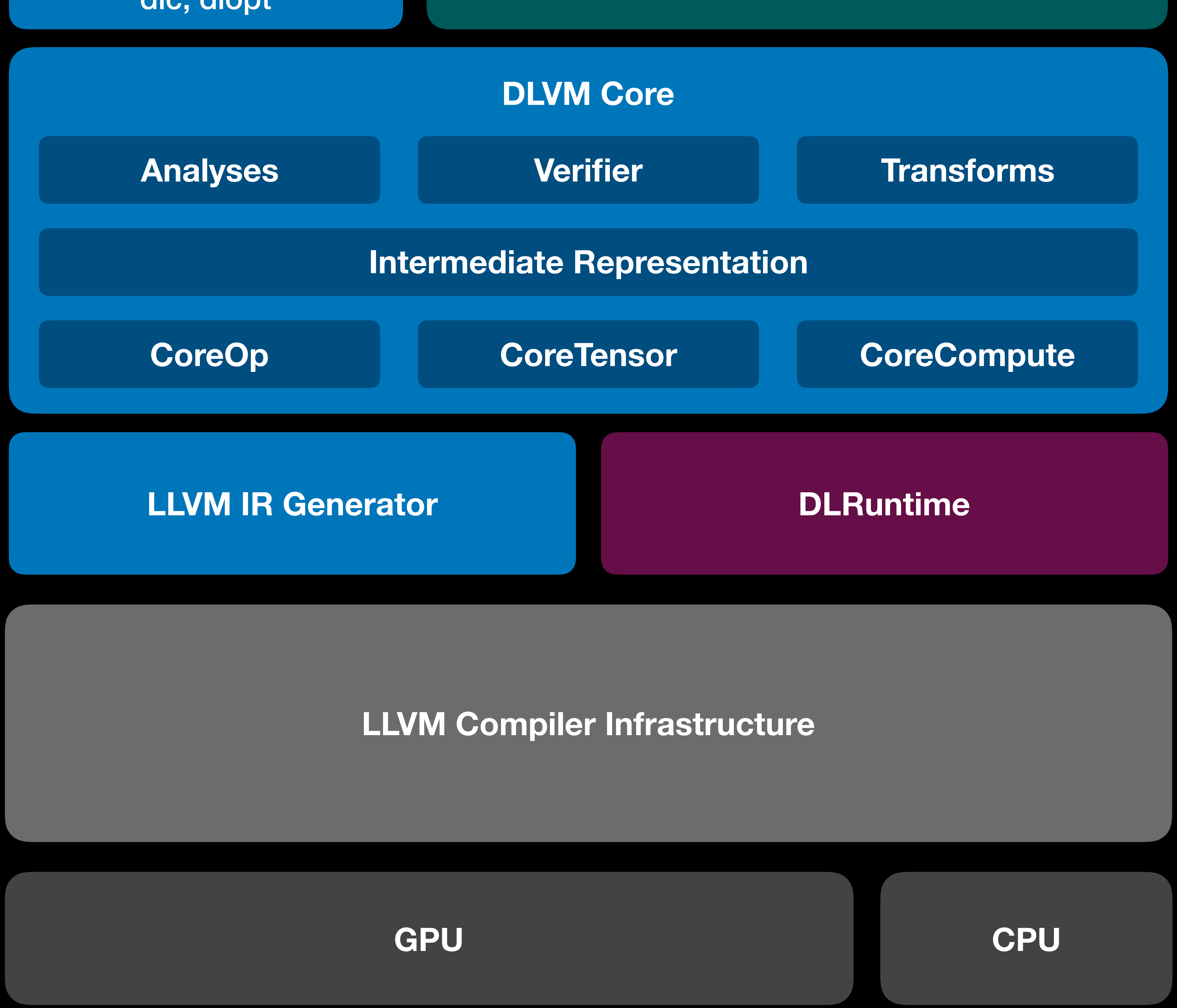
- Linear algebra IR
- Framework for building DSLs
- Automatic backpropagator
- Multi-stage optimizer
- Static code generator based on LLVM


DLVM Core
Analyses
Verifier
Transforms
Intermediate Representation
CoreOp
CoreTensor
CoreCompute


DSL stack
TEL
(Standalone DSL)
NNKit
(Embedded DSL)
DLVM Core
Analyses
Verifier
Transforms
Intermediate Representation
CoreOp
CoreTensor
CoreCompute


DSL stack
TEL
(Standalone DSL)
NNKit
(Embedded DSL)
DLVM Core
Analyses
Verifier
Transforms
Intermediate Representation
CoreOp
CoreTensor
CoreCompute
LLVM IR Generator
DLRuntime


DSL stack
TEL
(Standalone DSL)
NNKit
(Embedded DSL)
Command Line
Toolchain
dlc, dlopt
DLVM Core
Analyses
Verifier
Transforms
Intermediate Representation
CoreOp
CoreTensor
CoreCompute
LLVM IR Generator
DLRuntime


DLVM Core
Analyses
Verifier
Transforms
Intermediate Representation
CoreOp
CoreTensor
CoreCompute
LLVM IR Generator
DLRuntime
LLVM Compiler Infrastructure
GPU
CPU

Core Language: DLVM IR

Tensor Type

Rank	Notation	Descripton
0	`i64`	64-bit integer
1	`<100 x f32>`	float vector of size 100
2	`<100 x 300 x f64>`	double matrix of size 100x300
n	`<100 x 300 x ... x bool>`	rank-n tensor

First-class tensors

Domain-Specific Instructions

Kind	Example
Element-wise unary	tanh %a: <10 x f32>
Element-wise binary	power %a: <10 x f32>, %b: 2: f32
Dot	dot %a: <10 x 20 x f32>, %b: <20 x 2 x f32>
Concatenate	concatenate %a: <10 x f32>, %b: <20 x f32> along 0
Reduce	reduce %a: <10 x 30 x f32> by add along 1
Transpose	transpose %m: <2 x 3 x 4 x 5 x i32>
Convolution	convolve %a: <…> kernel %b: <…> stride %c: <…> …
Slice	slice %a: <10 x 20 x i32> from 1 upto 5
Random	random 768 x 10 from 0.0: f32 upto 1.0: f32
Select	select %x: <10 x f64>, %y: <10 x f64> by %flags: <10 x bool>
Compare	greaterThan %a: <10 x 20 x bool>, %b: <1 x 20 x bool>
Data type cast	dataTypeCast %x: <10 x i32> to f64

General-Purpose Instructions

Kind	Example
Function application	`apply %foo(%x: f32, %y: f32): (f32, f32) -> <10 x 10 x f32>`
Branch	`branch 'block_name(%a: i32, %b: i32)`
Conditional (if-then-else)	`conditional %cond: bool then 'then_block() else 'else_block()`
Shape cast	`shapeCast %a: <1 x 40 x f32> to 2 x 20`
Extract	`extract #x from %pt: $Point`
Insert	`insert 10: f32 to %pt: $Point at #x`
Allocate stack	`allocateStack $Point count 1`
Allocate heap	`allocateHeap $MNIST count 1`
Deallocate	`deallocate %x: *<10 x f32>`
Load	`load %ptr: *<10 x i32>`
Store	`store %x: <10 x i32> to %ptr: *<10 x i32>`
Copy	`copy from %src: *<10 x f16> to %dst: *<10 x f16> count 1: i64`

Instruction Set

Instruction Set

- Primitive math operators & general purpose operators

Instruction Set

- Primitive math operators & general purpose operators
- No **softmax, sigmoid**
  - Composed by primitive math ops

Instruction Set

- Primitive math operators & general purpose operators
- No **softmax, sigmoid**
  - Composed by primitive math ops
- No **min, max, relu**
  - Composed by **compare** & **select**

DLVM IR

DLVM IR

- Full static single assignment (SSA) form

DLVM IR

- Full static single assignment (SSA) form
    - Control flow graph (CFG) and basic blocks with arguments

DLVM IR

- Full static single assignment (SSA) form
  - Control flow graph (CFG) and basic blocks with arguments
- Custom type definitions

DLVM IR

- Full static single assignment (SSA) form
  - Control flow graph (CFG) and basic blocks with arguments
- Custom type definitions
- Modular architecture (module - function - basic block - instruction)

DLVM IR

- Full static single assignment (SSA) form
  - Control flow graph (CFG) and basic blocks with arguments
- Custom type definitions
- Modular architecture (module - function - basic block - instruction)
- Textual format & in-memory format

DLVM IR

- Full static single assignment (SSA) form
  - Control flow graph (CFG) and basic blocks with arguments
- Custom type definitions
- Modular architecture (module - function - basic block - instruction)
- Textual format & in-memory format
  - Built-in parser and verifier

DLVM IR

- Full static single assignment (SSA) form
  - Control flow graph (CFG) and basic blocks with arguments
- Custom type definitions
- Modular architecture (module - function - basic block - instruction)
- Textual format & in-memory format
  - Built-in parser and verifier
  - Robust unit testing via LLVM Integrated Tester (lit) and FileCheck

DLVM IR


Module
Function
Basic Block
Basic Block
Basic Block
Function
Basic Block
Basic Block
Basic Block
Type Definition

```
module "my_module" // Module declaration
stage raw          // Raw stage IR in the compilation phase

struct $Classifier {
    #w: <784 x 10 x f32>,
    #b: <1 x 10 x f32>,
}

type $MyClassifier = $Classifier
```

```
module "my_module" // Module declaration
stage raw          // Raw stage IR in the compilation phase

struct $Classifier {
    #w: <784 x 10 x f32>,
    #b: <1 x 10 x f32>,
}

type $MyClassifier = $Classifier

func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

```
module "my_module" // Module declaration
stage raw          // Raw stage IR in the compilation phase

struct $Classifier {
    #w: <784 x 10 x f32>,
    #b: <1 x 10 x f32>,
}

type $MyClassifier = $Classifier

func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    conditional true: bool then ‘b0() else 'b1()
'b0():
    return %0.1: <1 x 10 x f32>
‘b1():
    return 0: <1 x 10 x f32>
}
```

Transformations:
Differentiation & Optimizations

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @inference wrt 1, 2]
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @inference wrt 1, 2]
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

Differentiation Pass

Canonicalizes every gradient function declaration in an IR module

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

```
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):





}
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}


func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):

       Copy instructions from original function



}
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}


func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>




}
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}


func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>

        Generate adjoint code

}
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

```
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    %0.2 = transpose %x: <1 x 784 x f32>
    %0.3 = multiply %0.2: <1 x 784 x f32>, 1: f32
    return (%0.3: <1 x 10 x f32>, 1: f32): (<1 x 10 x f32>, f32)
}
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

```
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    %0.2 = transpose %x: <1 x 784 x f32>
    %0.3 = multiply %0.2: <1 x 784 x f32>, 1: f32
    return (%0.3: <1 x 10 x f32>, 1: f32): (<1 x 10 x f32>, f32)
}
```

Algebra Simplification Pass

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

```
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    %0.2 = transpose %x: <1 x 784 x f32>
    return (%0.2: <1 x 10 x f32>, 1: f32): (<1 x 10 x f32>, f32)
}
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

```
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    %0.2 = transpose %x: <1 x 784 x f32>
    return (%0.2: <1 x 10 x f32>, 1: f32): (<1 x 10 x f32>, f32)
}
```

Dead Code Elimination Pass

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}


func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>) {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = transpose %x: <1 x 784 x f32>
    return (%0.0: <1 x 10 x f32>, 1: f32): (<1 x 10 x f32>, f32)
}
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @inference from 0]
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
```

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @inference from 0]
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
```

Configurable gradient declaration

`from`: selecting which output to differentiate in tuple return

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @inference from 0 wrt 1, 2]
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

Configurable gradient declaration

`from`: selecting which output to differentiate in tuple return

`wrt`: with respect to arguments 1 & 2

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @inference from 0 wrt 1, 2 keeping 0]
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>, <1 x 10 x f32>)
```

Configurable gradient declaration

`from`: selecting which output to differentiate in tuple return

`wrt`: with respect to arguments 1 & 2

`keeping`: keeping original output

```
func @inference: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @inference from 0 wrt 1, 2 keeping 0 seedable]
func @inference_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>, <1 x 10 x f32>)
                    -> (<784 x 10 x f32>, <1 x 10 x f32>, <1 x 10 x f32>)
```

Configurable gradient declaration

`from`: selecting which output to differentiate in tuple return

`wrt`: with respect to arguments 1 & 2

`keeping`: keeping original output

`seedable`: allow passing in back-propagated gradients as seed

```
func @f: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

func @g: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = apply @f(%x, %w, %b): (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32>
    %0.1 = tanh %0.0: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

```
func @f: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

func @g: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = apply @f(%x, %w, %b): (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32>
    %0.1 = tanh %0.0: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @g wrt 1, 2]
func @g_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

```
func @f: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

func @g: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = apply @f(%x, %w, %b): (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32>
    %0.1 = tanh %0.0: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @g wrt 1, 2]
func @g_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

```
func @f: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

func @g: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = apply @f(%x, %w, %b): (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32>
    %0.1 = tanh %0.0: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @g wrt 1, 2]
func @g_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> (<784 x 10 x f32>, <1 x 10 x f32>)

[gradient @f wrt 1, 2 seedable]
func @f_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>, <1 x 10 x f32>)
             -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

```
func @f: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

func @g: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = apply @f(%x, %w, %b): (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32>
    %0.1 = tanh %0.0: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}

[gradient @g wrt 1, 2]
func @g_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> (<784 x 10 x f32>, <1 x 10 x f32>)

[gradient @f wrt 1, 2 seedable]
func @f_grad: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>, <1 x 10 x f32>)
            -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

Seed

Compilation Phases

Compilation Phases

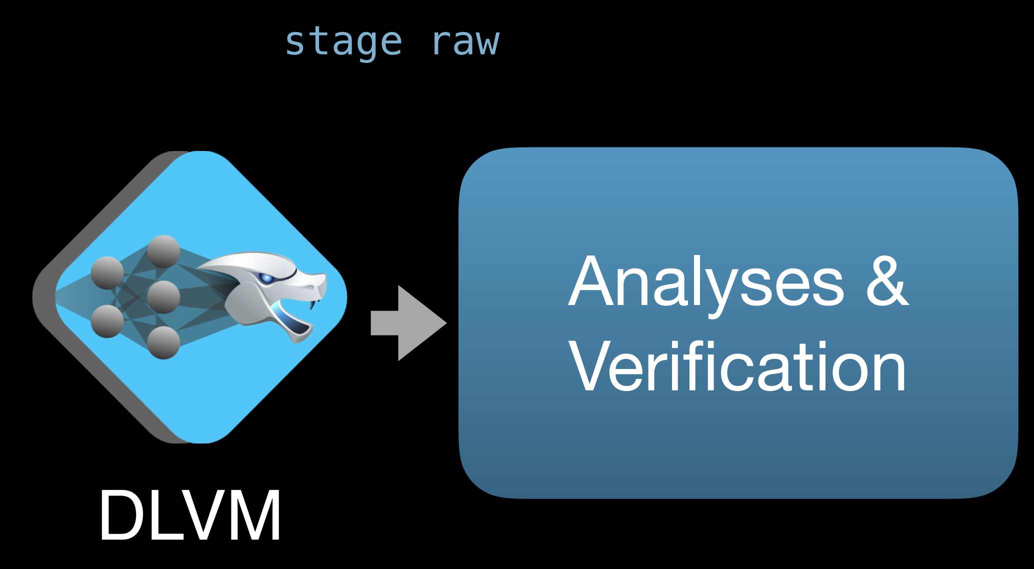
DLVM

Compilation Phases

`stage raw`

DLVM

Compilation Phases



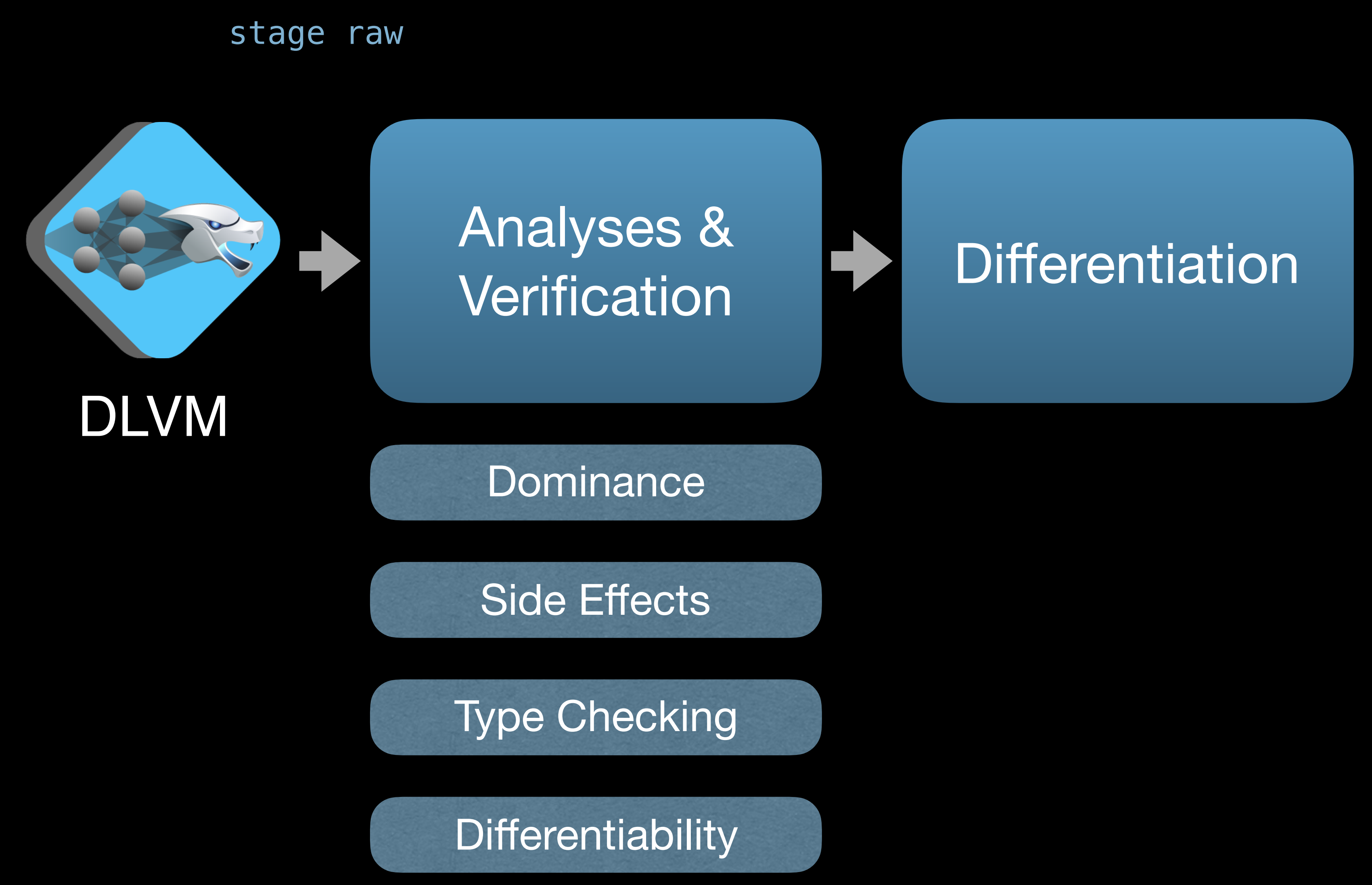
Compilation Phases

`stage raw`

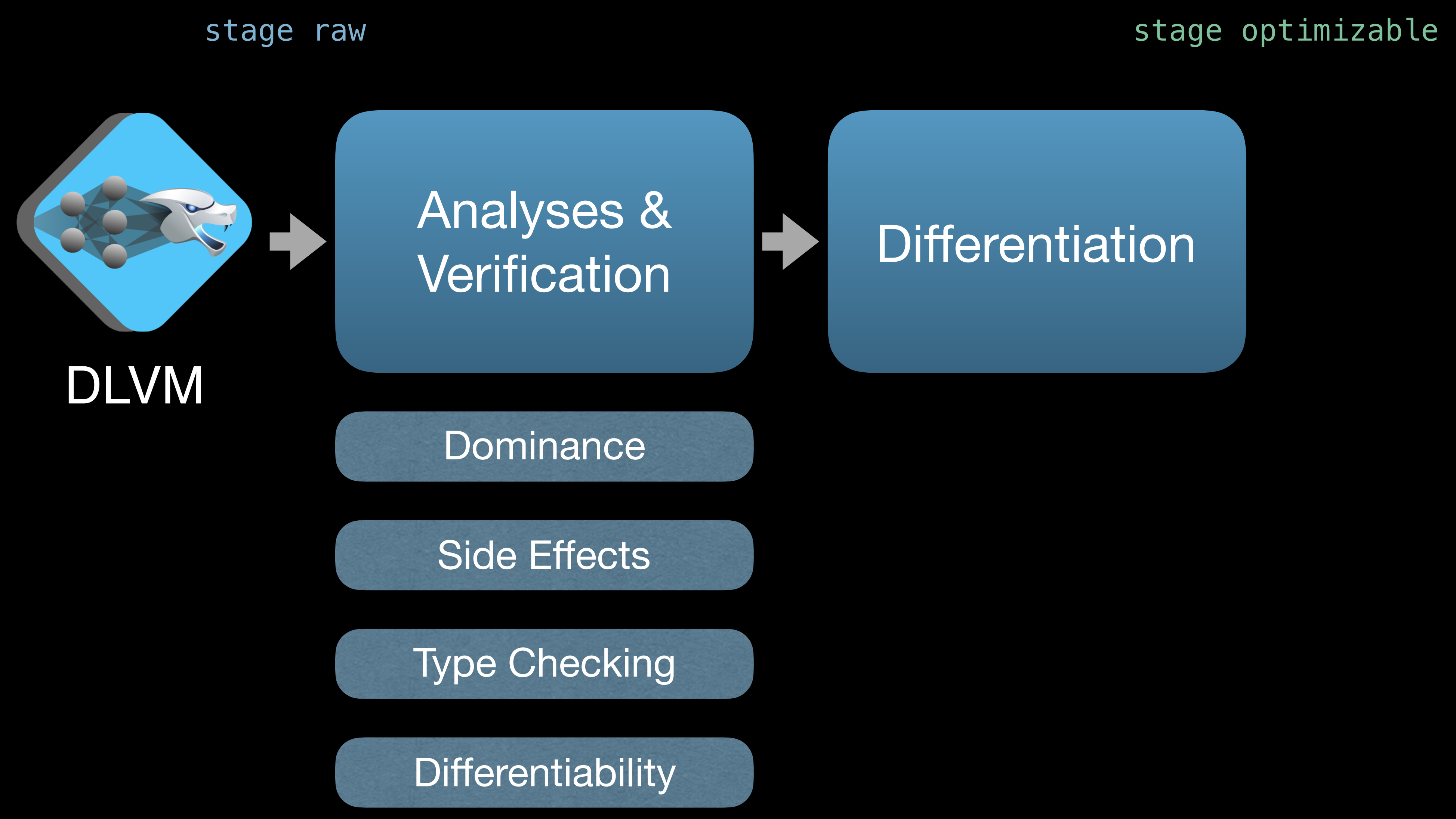
DLVM → Analyses & Verification

- Dominance
- Side Effects
- Type Checking
- Differentiability

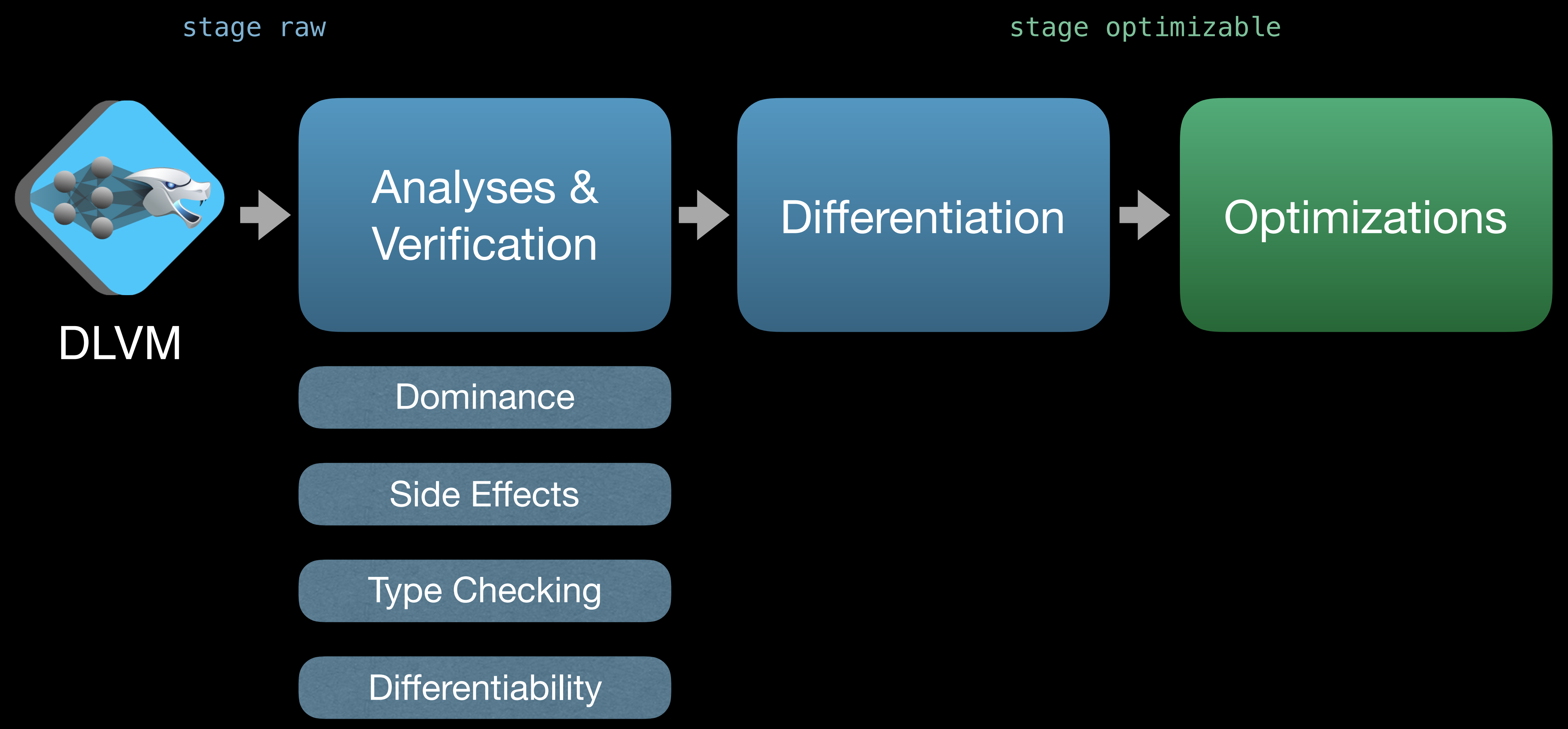
Compilation Phases



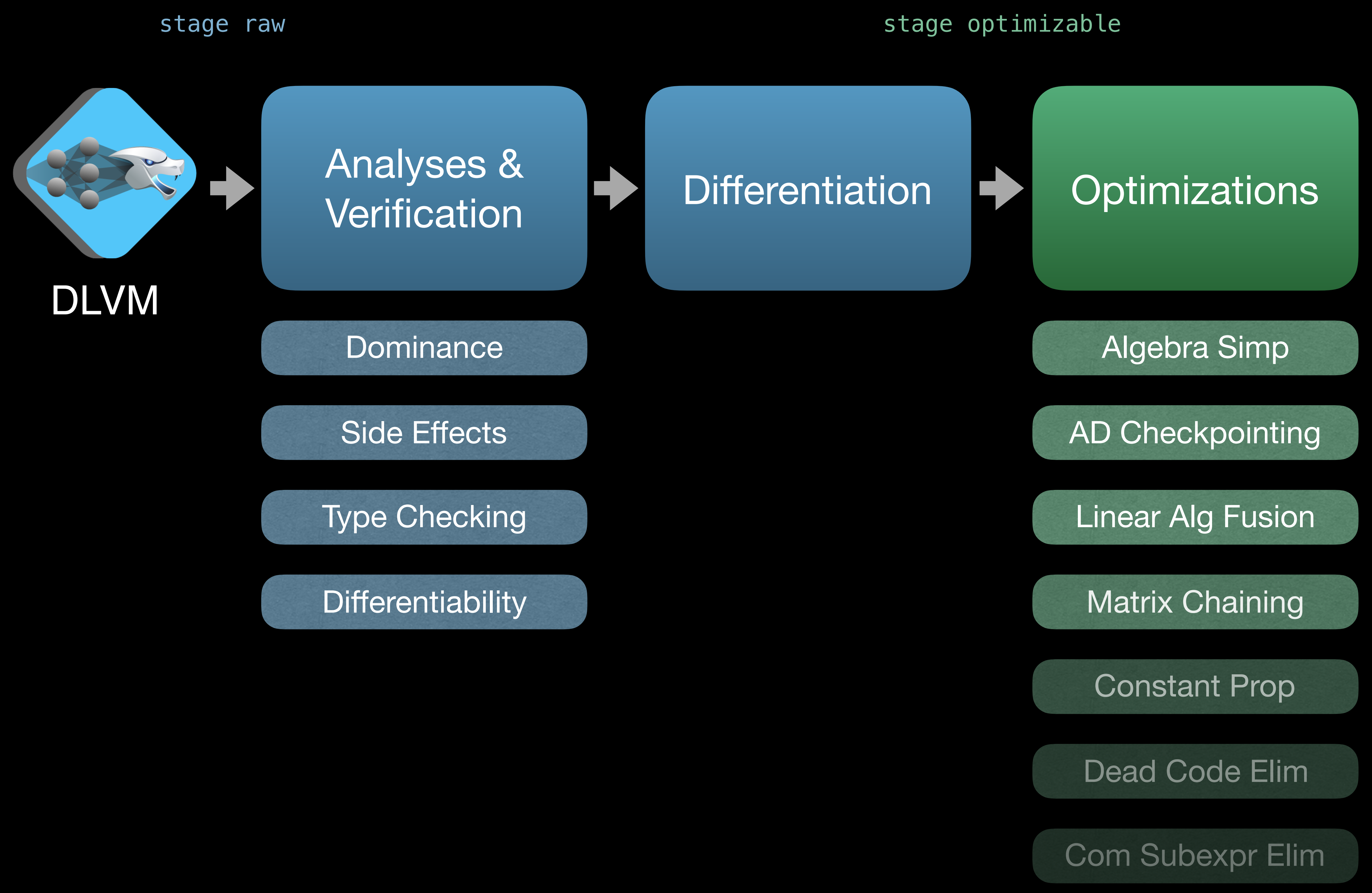
Compilation Phases



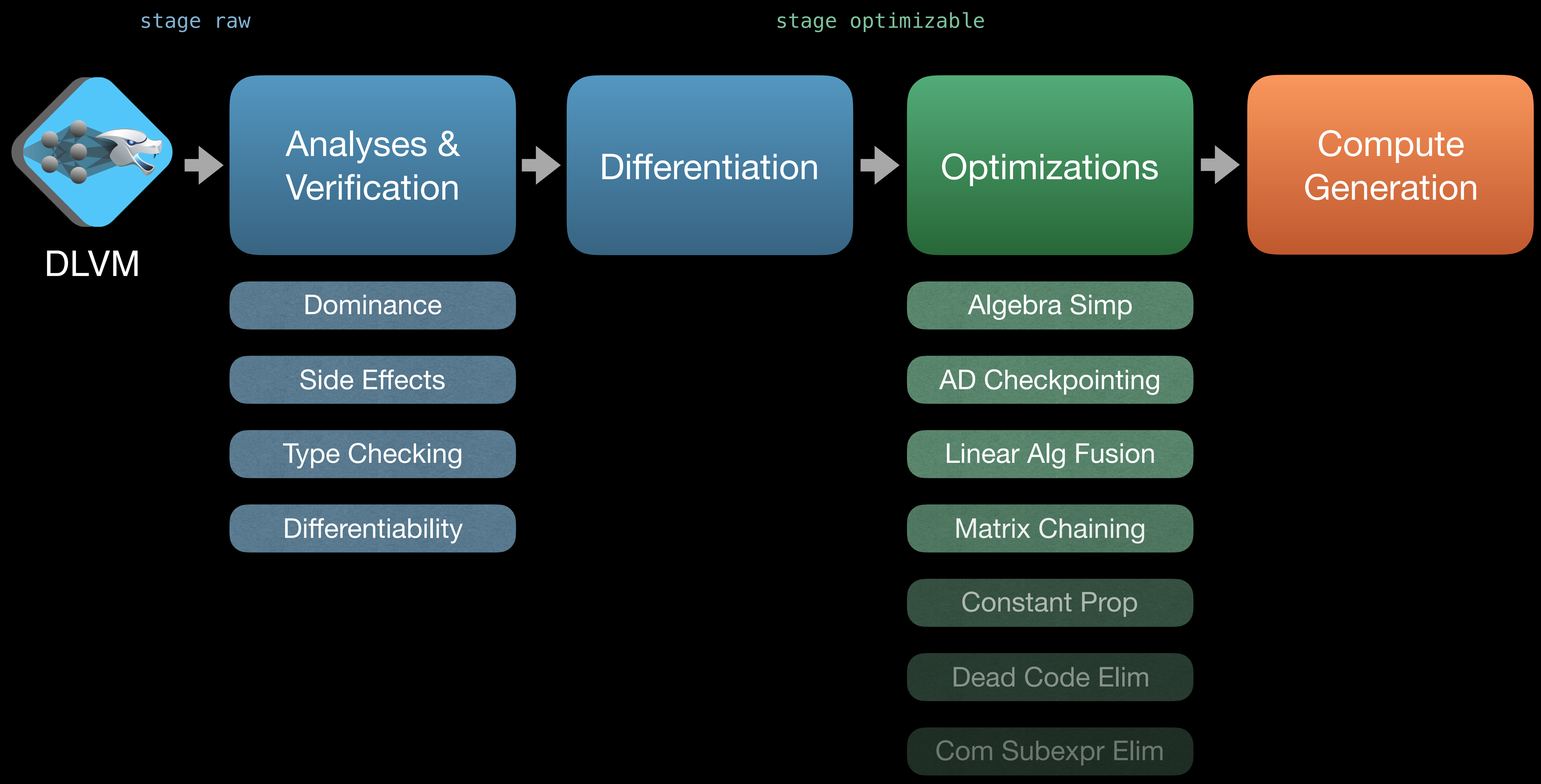
Compilation Phases



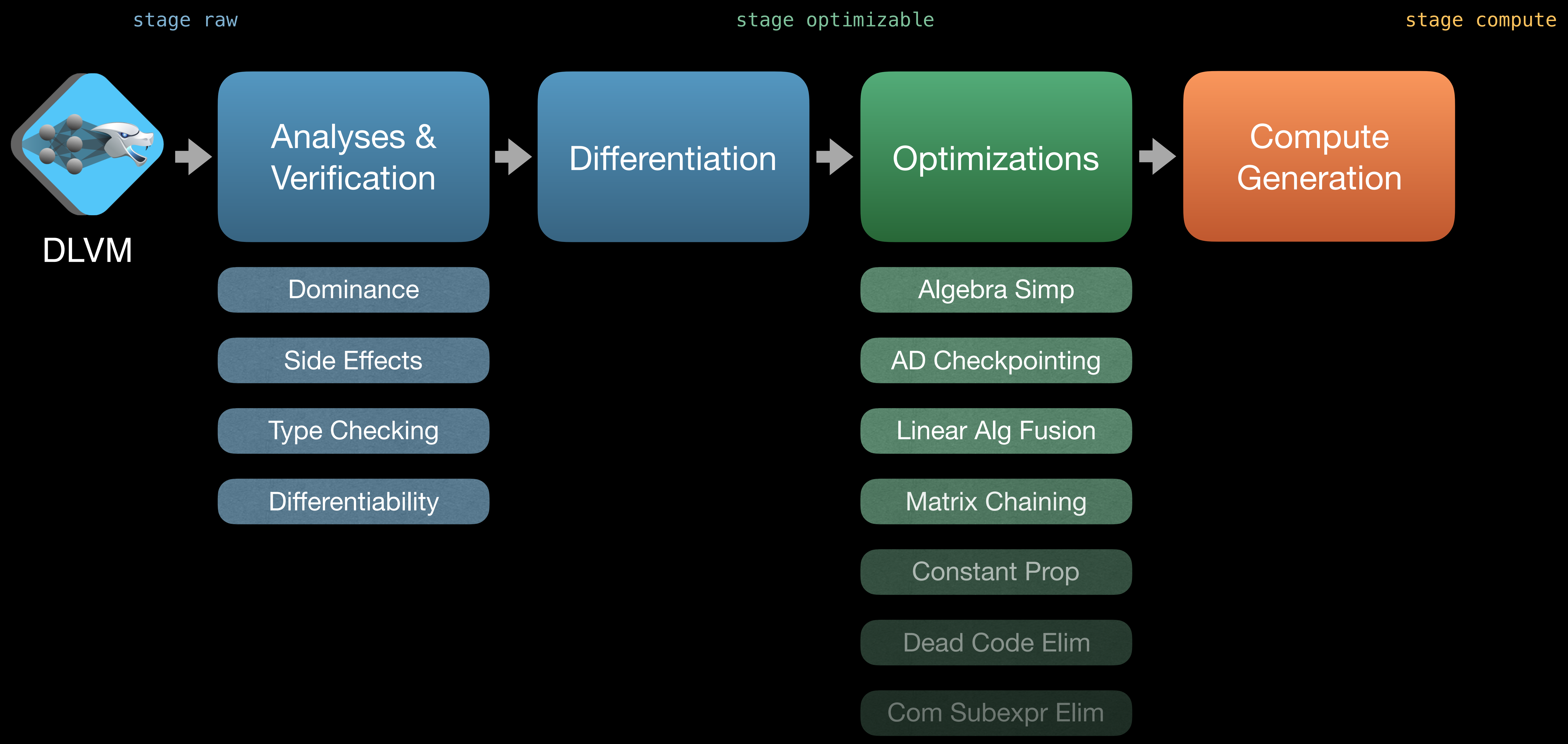
Compilation Phases



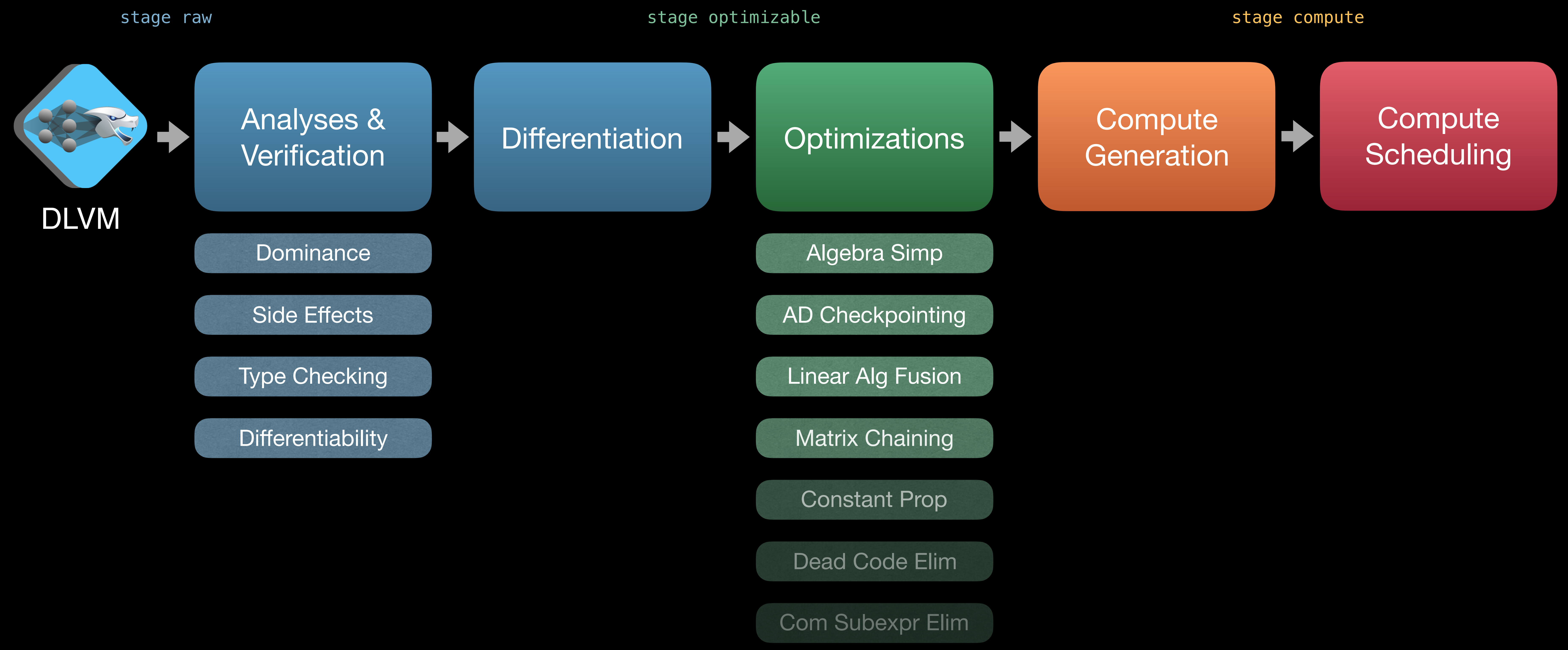
Compilation Phases



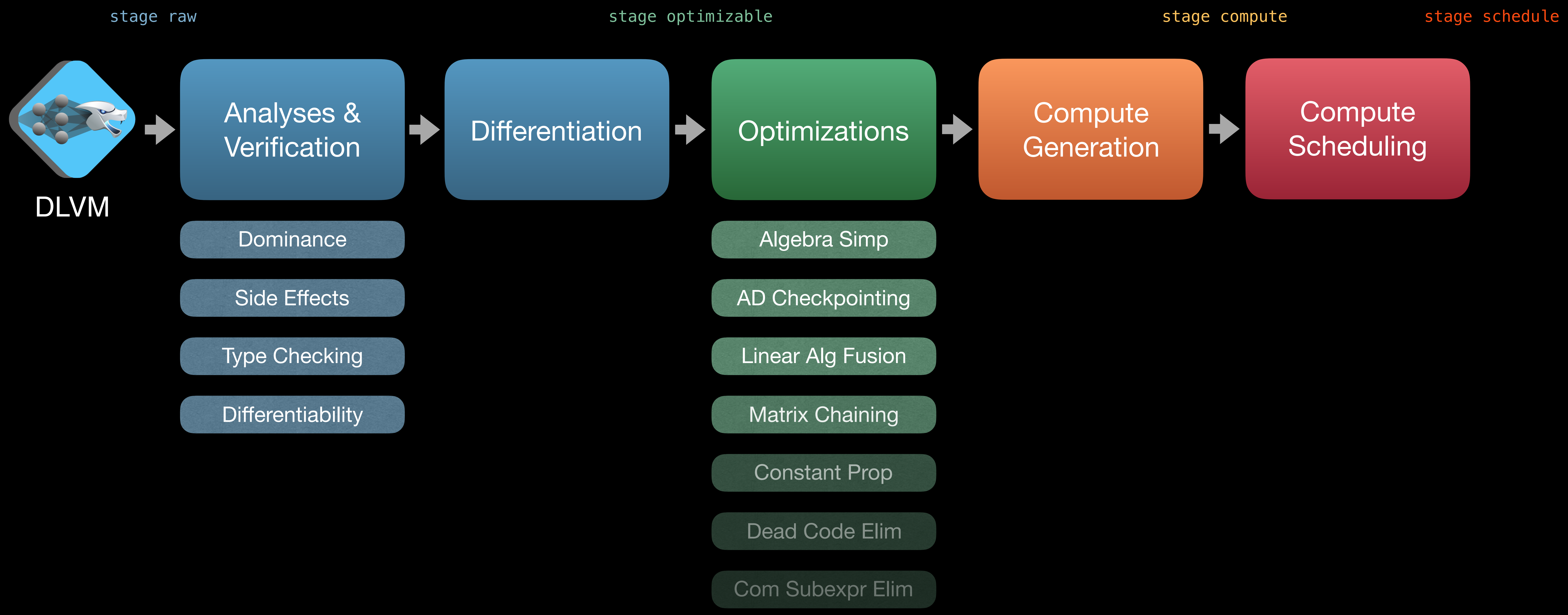
Compilation Phases



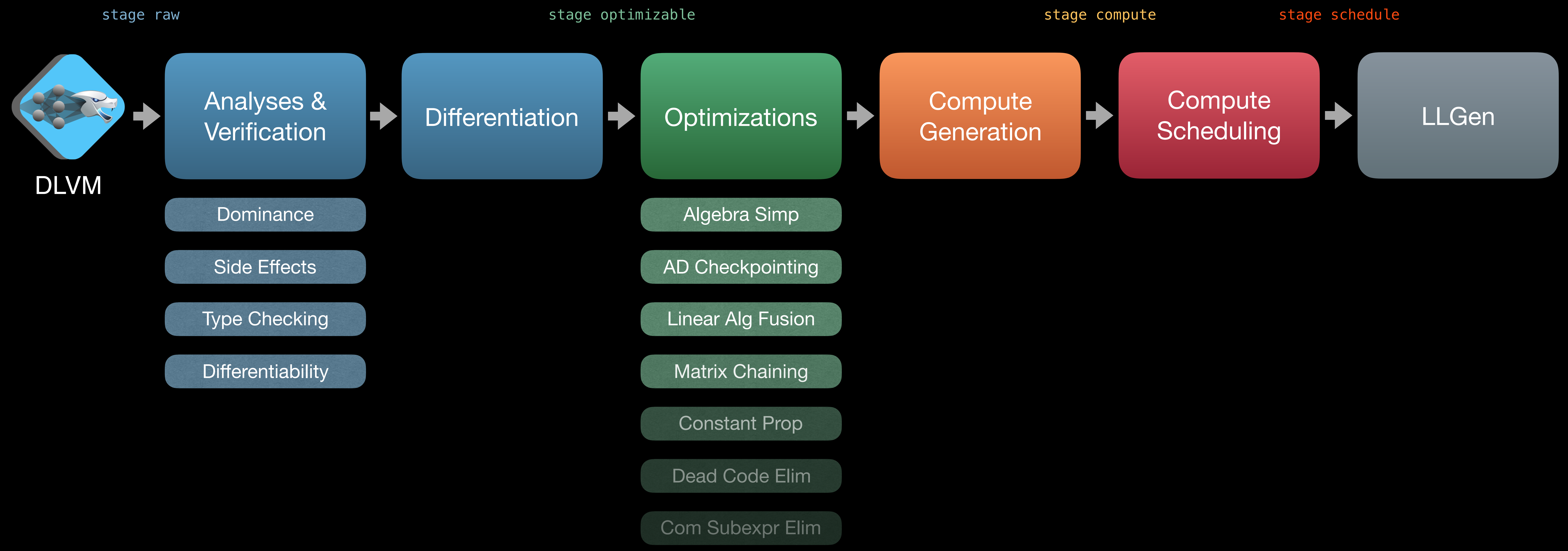
Compilation Phases



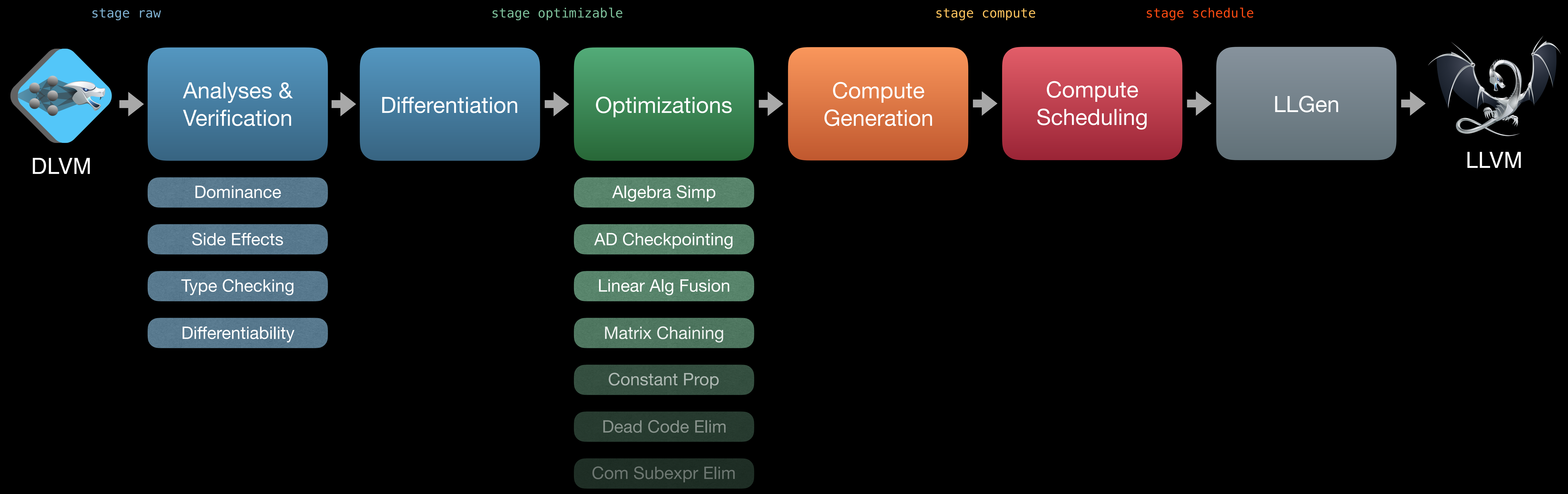
Compilation Phases



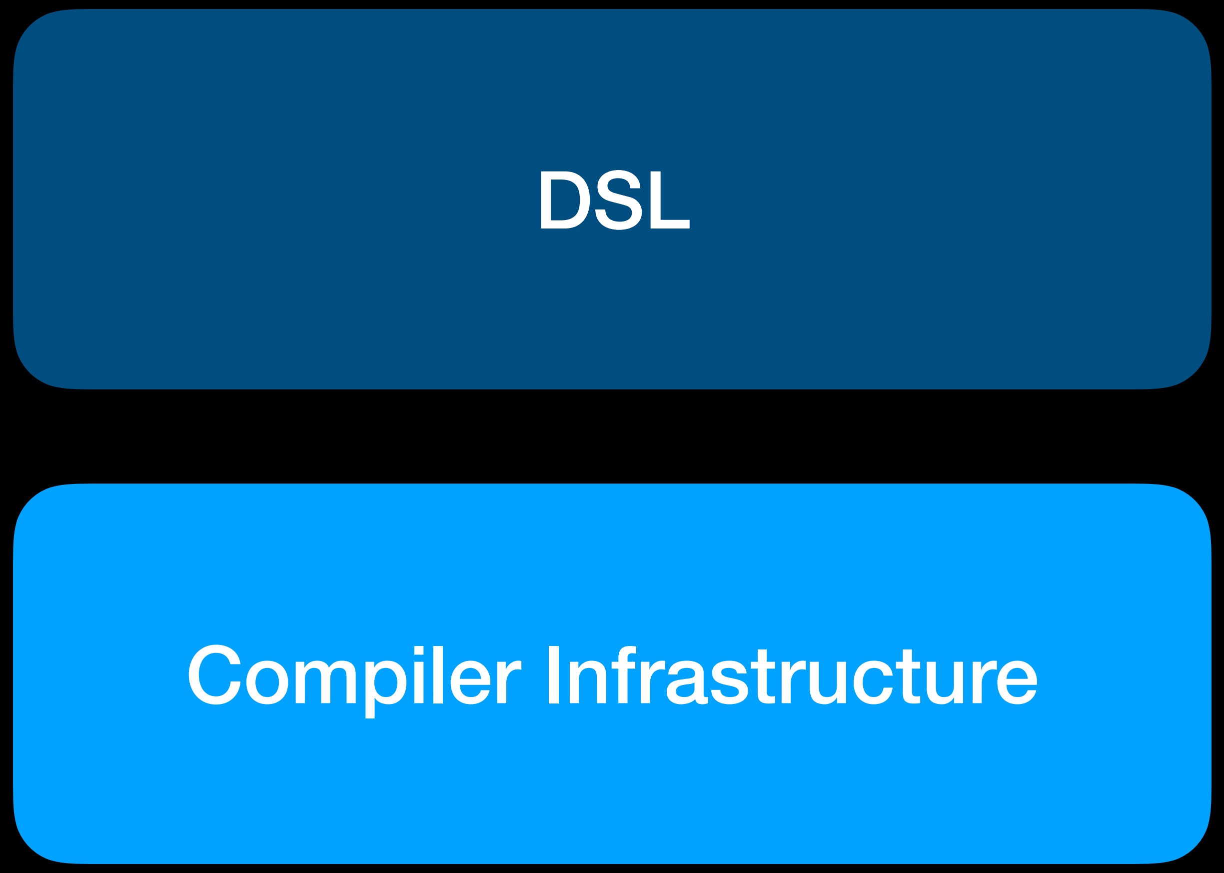
Compilation Phases

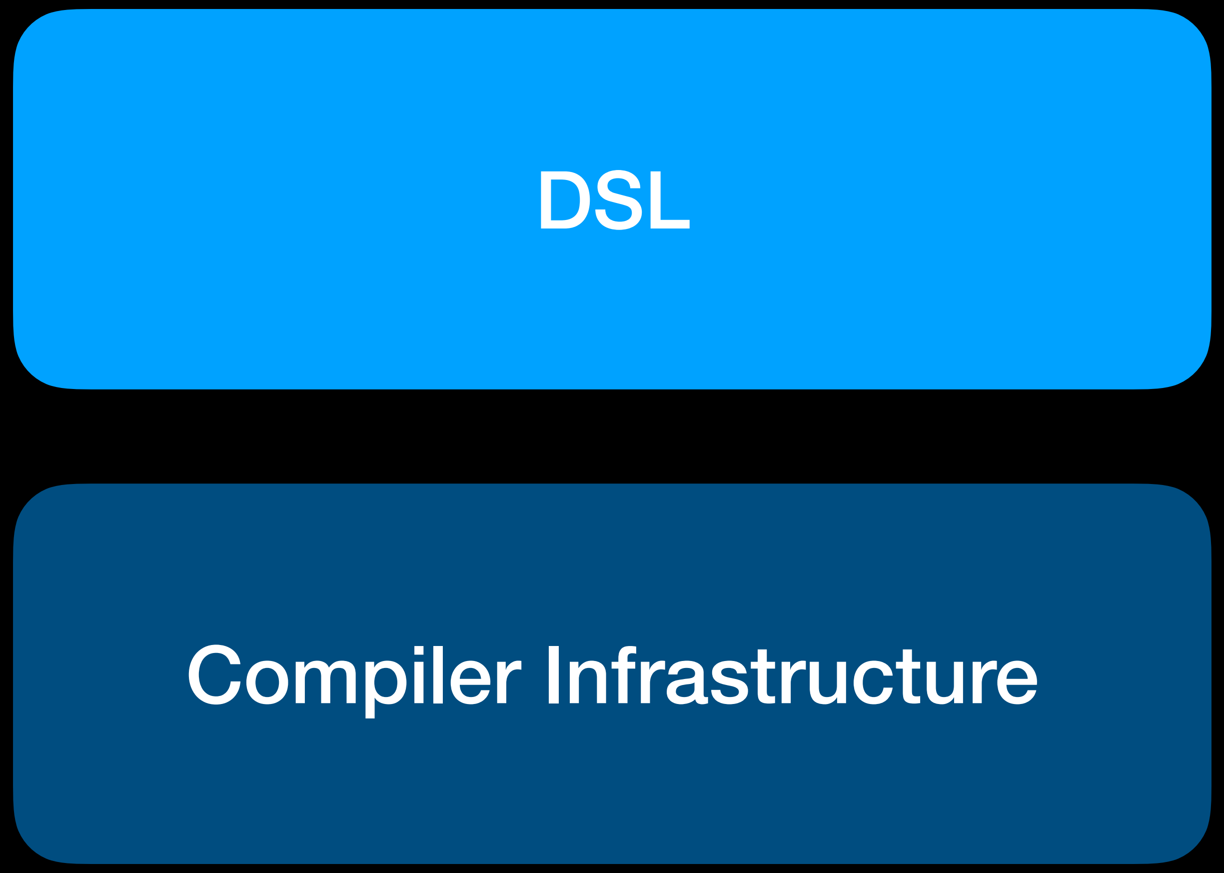


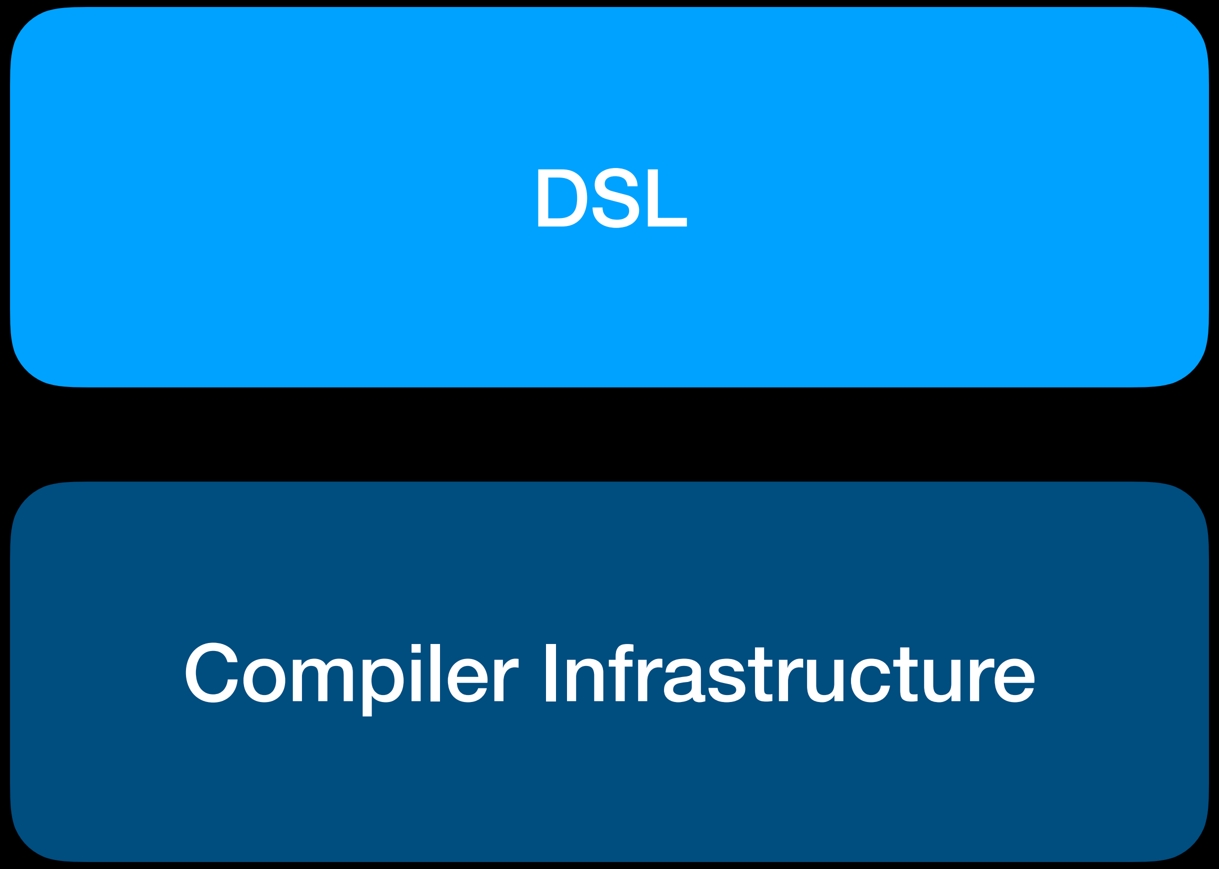
Compilation Phases

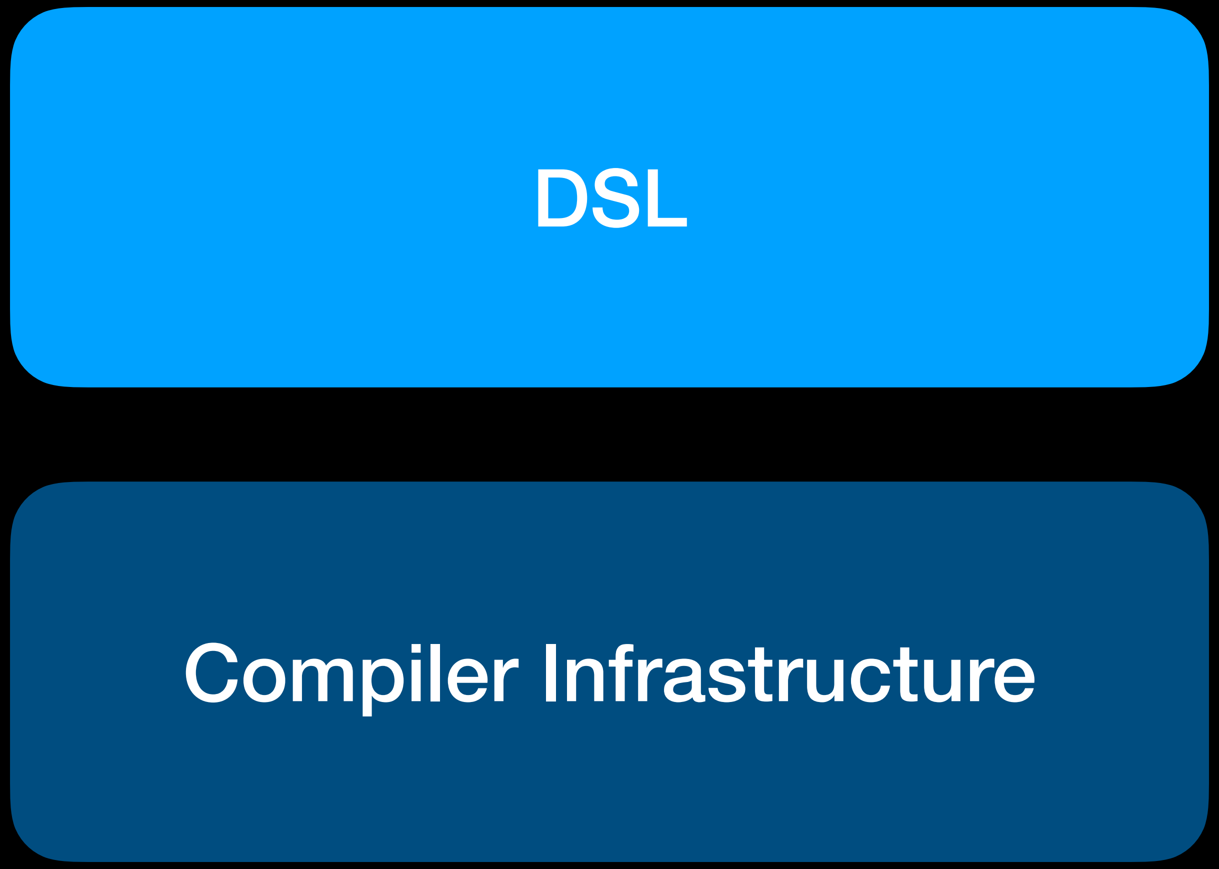


DLVM


DSL
Compiler Infrastructure


DSL
Compiler Infrastructure


DSL
Compiler Infrastructure



- NN as program, not a graph
- Static analysis
- Type safety
- Naturalness
  - Lightweight modular staging
  - Compiler magic

NNKit: Staged DSL in Swift

NNKit

NNKit

- It’s a prototype!

NNKit

- It’s a prototype!
- Tensor computation embedded in host language

NNKit

- It’s a prototype!
- Tensor computation embedded in host language
- Type safety

NNKit

- It’s a prototype!
- Tensor computation embedded in host language
- Type safety
- Generates DLVM IR on the fly

Language

Language

- Statically ranked tensors
  - `T, Tensor1D<T>, Tensor2D<T>, Tensor3D<T>, Tensor4D<T>`

Language

- Statically ranked tensors
  - `T, Tensor1D<T>, Tensor2D<T>, Tensor3D<T>, Tensor4D<T>`
- Type wrapper for staging – `Rep<Wrapped>`
  - `Rep<Float>, Rep<Tensor1D<Float>>, Rep<Tensor2D<T>>`

Language

- Statically ranked tensors
  - T, Tensor1D<T>, Tensor2D<T>, Tensor3D<T>, Tensor4D<T>
- Type wrapper for staging – Rep<Wrapped>
  - Rep<Float>, Rep<Tensor1D<Float>>, Rep<Tensor2D<T>>
- Operator overloading
  - func + <T: Numeric>(_: Rep<T>, _: Rep<T>) -> Rep<T>
  - func • (_: Rep<Tensor2D<T>>, _: Rep<Tensor2D<T>>) -> Rep<Tensor2D<T>>

Language

Language

- Lambda abstraction
  - `func lambda<T, U>(_ f: (Rep<T>) -> Rep<U>) -> Rep<(T) -> U>`

Language

- Lambda abstraction
  - `func lambda<T, U>(_ f: (Rep<T>) -> Rep<U>) -> Rep<(T) -> U>`
- Function application
  - `subscript<T, U>(arg: Rep<T>) -> Rep<U> where Wrapped == (T) -> U`
  - `subscript<T, U>(arg: T) -> U where Wrapped == (T) -> U`
    `// JIT DLVM IR`

Staged Evaluation

Staged Evaluation

```
Rep<(Float2D) -> Float2D>
```

Staged Evaluation

```
Rep<(Float2D) -> Float2D>
```

```
(Float2D) -> Float2D
```

Staged Evaluation

Staged Evaluation

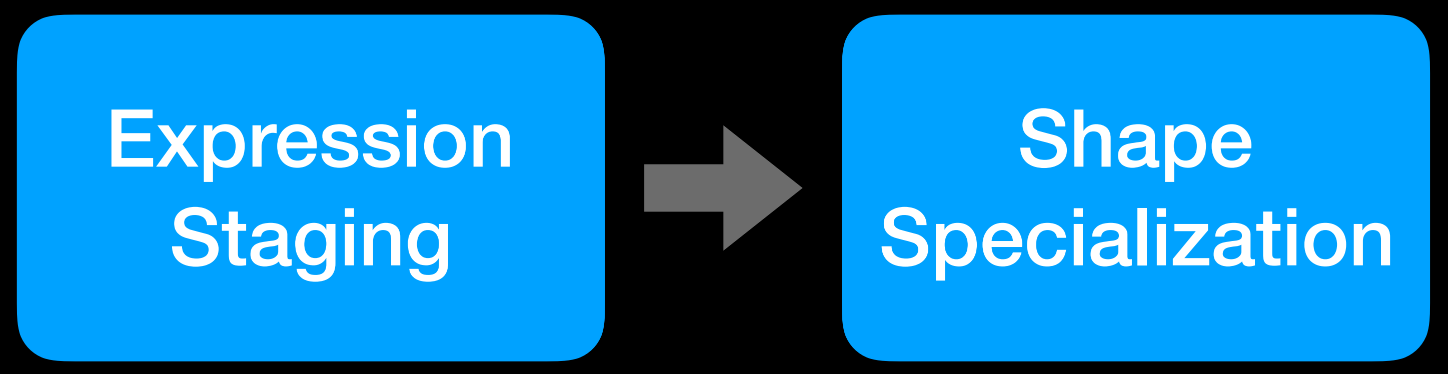
```
Rep<(Float2D) -> Float2D>
```

Staged Evaluation

Expression Staging

```
Rep<(Float2D) -> Float2D>
```

Staged Evaluation

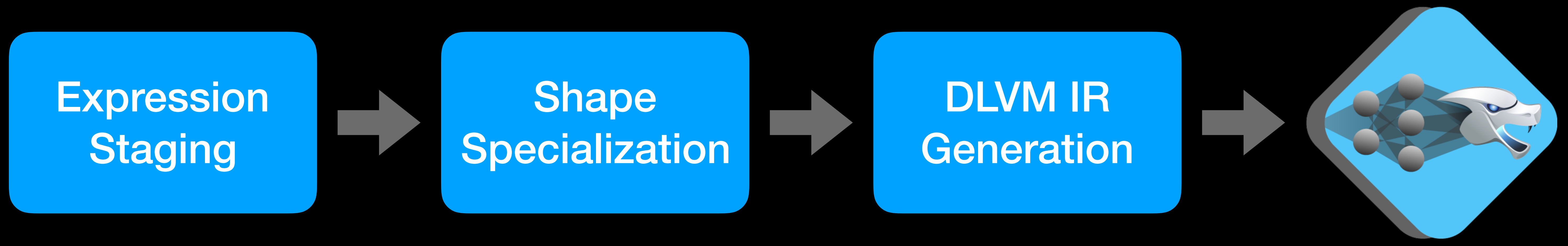



```
Rep<(Float2D) -> Float2D>
```

Staged Evaluation


```
Rep<(Float2D) -> Float2D>
```

Staged Evaluation




```
Rep<(Float2D) -> Float2D>
```

Staged Evaluation

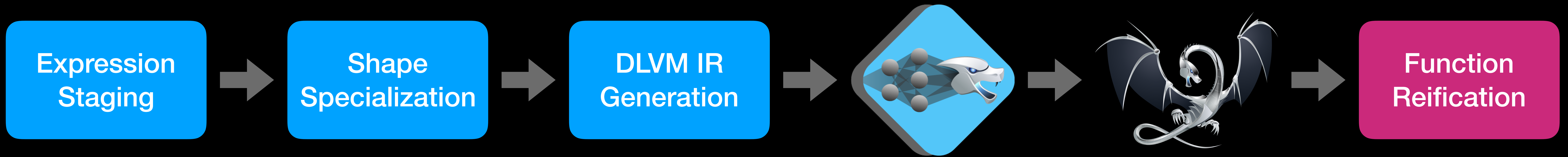

```
Rep<(Float2D) -> Float2D>
```

Staged Evaluation

Expression Staging → Shape Specialization → DLVM IR Generation → → → Function Reification

```
Rep<(Float2D) -> Float2D>
```

Staged Evaluation




```
Rep<(Float2D) -> Float2D>
```

```
(Float2D) -> Float2D
```

```
typealias Float2D = Tensor2D<Float>
```

```swift
typealias Float2D = Tensor2D<Float>

struct Parameters {
    var w: Float2D
    var b: Float2D
}
```

```swift
typealias Float2D = Tensor2D<Float>

struct Parameters {
    var w: Float2D
    var b: Float2D
}

let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
```

```
typealias Float2D = Tensor2D<Float>

struct Parameters {
    var w: Float2D
    var b: Float2D
}

let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }
```

```
typealias Float2D = Tensor2D<Float>

struct Parameters {
    var w: Float2D
    var b: Float2D
}

let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let params: Parameters = …
```

```
typealias Float2D = Tensor2D<Float>

struct Parameters {
    var w: Float2D
    var b: Float2D
}

let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let params: Parameters = …
let x: Float2D = [[0.0, 1.0]]
```

```
typealias Float2D = Tensor2D<Float>

struct Parameters {
    var w: Float2D
    var b: Float2D
}

let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let params: Parameters = …
let x: Float2D = [[0.0, 1.0]]
f[x, params.w, params.b]  // ==> result
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

f[x, w, b]
// x: 1x784, w: 784x10, b: 1x10
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

f[x, w, b]
// x: 1x784, w: 784x10, b: 1x10
```

```
func @f: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>) -> <1 x 10 x f32> {
'entry(%x: <1 x 784 x f32>, %w: <784 x 10 x f32>, %b: <1 x 10 x f32>):
    %0.0 = dot %x: <1 x 784 x f32>, %w: <784 x 10 x f32>
    %0.1 = add %0.0: <1 x 10 x f32>, %b: <1 x 10 x f32>
    return %0.1: <1 x 10 x f32>
}
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}

let ∇g = gradient(of: g, withRespectTo: (1, 2))
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}

let ∇g = gradient(of: g, withRespectTo: (1, 2))
// ∇g : Rep<(Float2D, Float2D, Float2D) -> (Float2D, Float2D)>
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}

let ∇g = gradient(of: g, withRespectTo: (1, 2))
// ∇g : Rep<(Float2D, Float2D, Float2D) -> (Float2D, Float2D)>

∇g[x, w, b] // ==> ( ∂g/∂w, ∂g/∂b )
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}

let ∇g = gradient(of: g, withRespectTo: (1, 2))
// ∇g : Rep<(Float2D, Float2D, Float2D) -> (Float2D, Float2D)>

∇g[x, w, b] // ==> ( ∂g/∂w, ∂g/∂b )

[gradient @f wrt 1, 2]
func @g: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
         -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}

let ∇g = gradient(of: g, withRespectTo: (1, 2))
// ∇g : Rep<(Float2D, Float2D, Float2D) -> (Float2D, Float2D)>

∇g[x, w, b] // ==> ( ∂g/∂w, ∂g/∂b )

[gradient @f wrt 1, 2]
func @g: (<1 x 784 x f32>, <784 x 10 x f32>, <1 x 10 x f32>)
         -> (<784 x 10 x f32>, <1 x 10 x f32>)
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}

let ∇g = gradient(of: g, withRespectTo: (1, 2))
// ∇g : Rep<(Float2D, Float2D, Float2D) -> (Float2D, Float2D)>

∇g[x, w, b] // ==> ( ∂g/∂w, ∂g/∂b )
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}

let ∇g = gradient(of: g, withRespectTo: (1, 2), seedable: true)
// ∇g : Rep<(Float2D, Float2D, Float2D, Float2D) -> (Float2D, Float2D)>
```

```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}

let ∇g = gradient(of: g, withRespectTo: (1, 2), seedable: true, keeping: (0))
// ∇g : Rep<(Float2D, Float2D, Float2D, Float2D) -> (Float2D, Float2D, Float2D)>
```

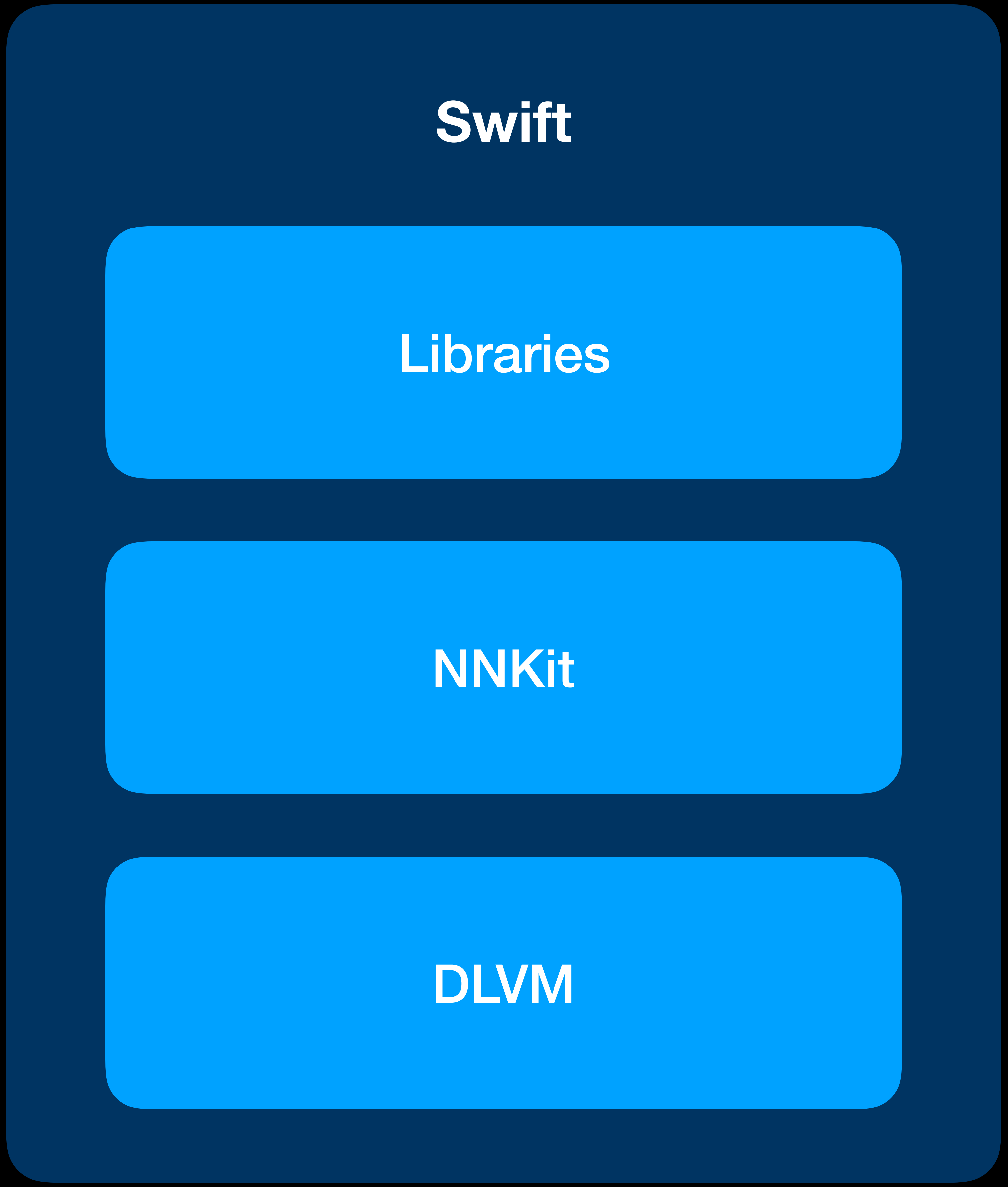
```
let f: Rep<(Float2D, Float2D, Float2D) -> Float2D> =
    lambda { x, w, b in
        x • w + b
    }

let g = lambda { x, w, b in
    let linear = f[x, w, b]
    return tanh(linear)
}

let ∇g = gradient(of: g, withRespectTo: (1, 2), seedable: true, keeping: (0))
// ∇g : Rep<(Float2D, Float2D, Float2D, Float2D) -> (Float2D, Float2D, Float2D)>

∇g[x, w, b, ∂h_∂g] // ==> ( ∂h/∂w, ∂h/∂b, g(x,w,b) )
```


Safe language
Libraries
DSL
Compiler Infrastructure


Swift
Libraries
NNKit
DLVM

Swift

Libraries

NNKit

DLVM

DLVM is written in Swift!

PL & Compilers + ML

PL & Compilers + ML

- Programs, not just a data flow graph

PL & Compilers + ML

- Programs, not just a data flow graph
- Type safety

PL & Compilers + ML

- Programs, not just a data flow graph
- Type safety
- Ahead-of-time AD

PL & Compilers + ML

- Programs, not just a data flow graph
- Type safety
- Ahead-of-time AD
- Code generation

dlvm.org

DLVM