Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients

William S. Moses

Valentin Churavy

wmoses@mit.edu

NeurIPS '20 Spotlight Talk

December 2020

Differentiation Is Key To Machine Learning

- Computing derivatives is key to many machine learning algorithms
- Existing approaches:
  - Rewrite all code in a differentiable DSL (PyTorch, TensorFlow, Taichi, etc)
  - Manually write gradient functions

Differentiation Is Key To Machine Learning

```
// C++ nbody simulator

void step(std::array<Planet> bodies, double dt) {
  vec3 acc[bodies.size()];
  for (size_t i=0; i<bodies.size(); i++) {
    acc[i] = vec3(0, 0, 0);
    for (size_t j=0; j<bodies.size(); j++) {
      if (i == j) continue;
      acc[i] += force(bodies[i], bodies[j]) /
                         bodies[i].mass;
    }
  }
  for (size_t i=0; i<bodies.size(); i++) {
    bodies[i].vel += acc[i] * dt;
    bodies[i].pos += bodies[i].vel * dt;
  }
}
```

```
// PyTorch rewrite of nbody simulator
import torch

def step(bodies, dt):
  acc = []
  for i in range(len(bodies)):
    acc.push(torch.zeros([3]))
    for j in range(len(bodies)):
      if i == j: continue
      acc[i] += force(bodies[i], bodies[j]) /
                         bodies[i].mass

  for i, body in enumerate(bodies):
    body.vel += acc[i] * dt
    body.pos += body.vel * dt
```

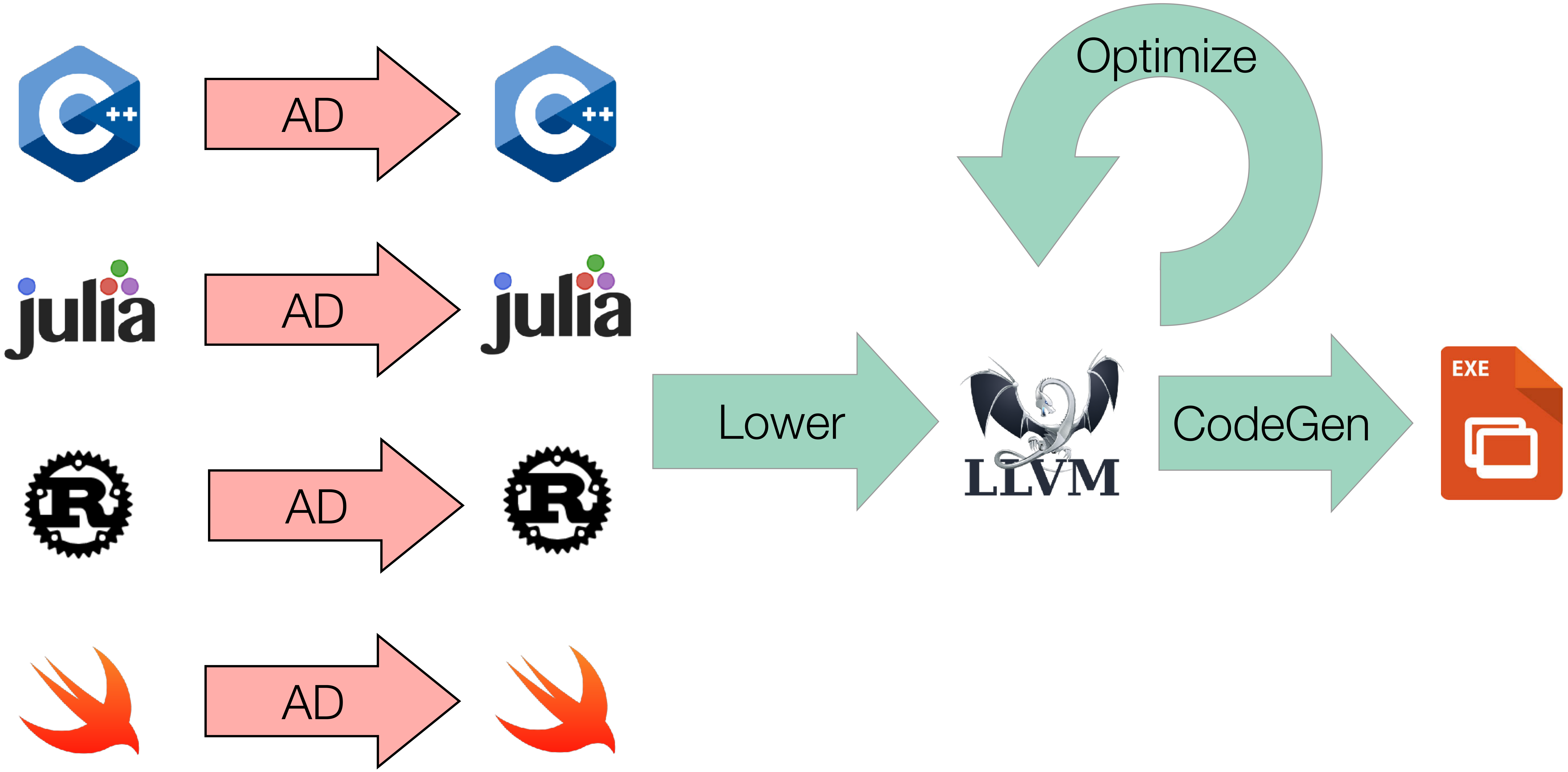
- Hinders application of ML to new domains
- Synthesizing gradients aims to close this gap

Conventional Wisdom: AD Only Feasible at High-Level

- Automatic Differentiation requires high level semantics to produce gradients
- Lack of high-level information can hinder performance of low-level AD
  - “AD is more effective in high-level compiled languages (e.g. Julia, Swift, Rust, Nim) than traditional ones such as C/C++, Fortran and LLVM IR […]” -Innes[1]

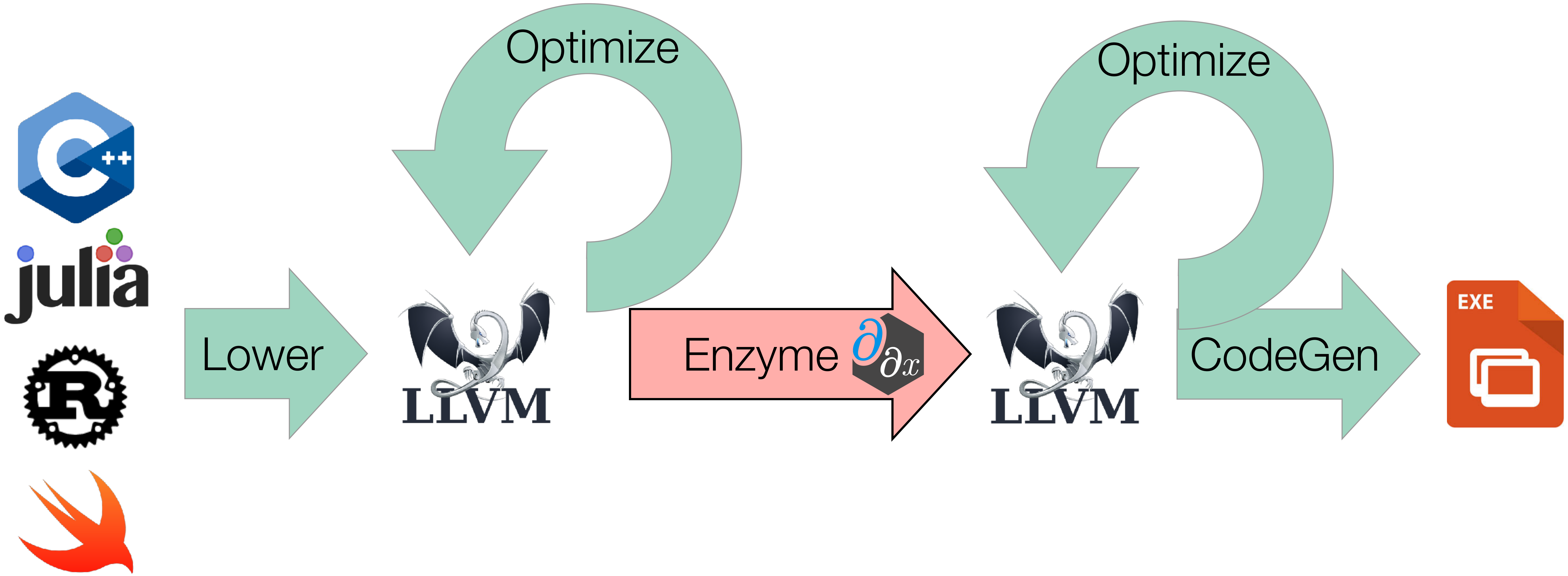
[1] Michael Innes. Don’t Unroll Adjoint: Differentiating SSA-Form Programs. arXiv preprint arXiv:1810.07951, 2018

Existing Automatic Differentiation Pipelines



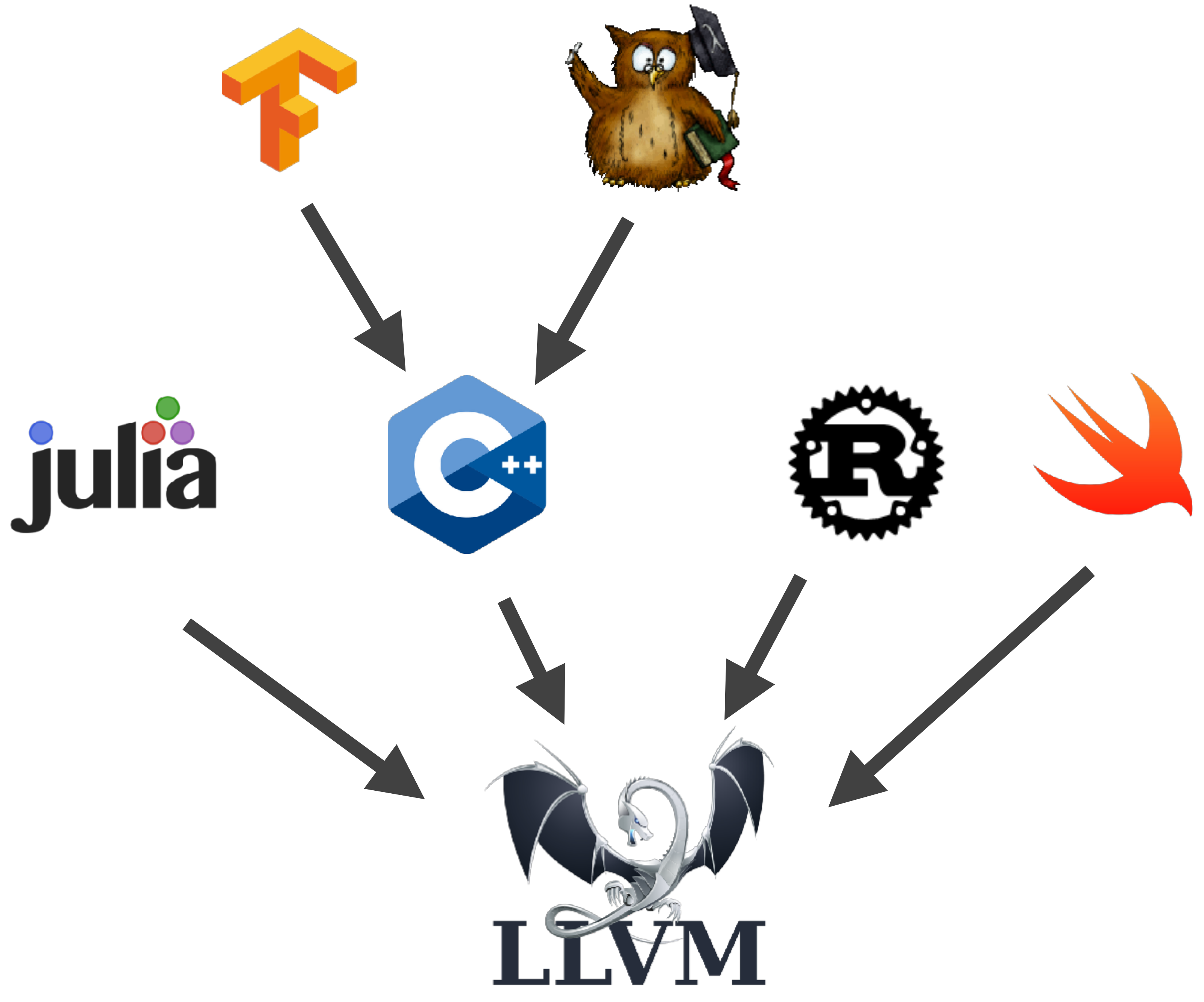
Enzyme Overturns Conventional Wisdom

- As fast or faster than state-of-the-art tools
  - Running after optimization enables a ***4.2x speedup***
- Necessary semantics for AD derived at low-level (with potential cooperation of frontend)



Why Does Enzyme Use LLVM?

- Generic low-level compiler infrastructure with many frontends
  - “Cross platform assembly”
- Well-defined semantics
- Large collection of optimizations and analyses



Case Study: Vector Normalization

```
//Compute magnitude in O(n)
double mag(double[] x);

//Compute norm in O(n^2)
void norm(double[] out, double[] in) {

  for (int i=0; i<n; i++) {
    out[i] = in[i] / mag(in);
  }
}
```

Case Study: Vector Normalization

```
//Compute magnitude in O(n)
double mag(double[] x);

//Compute norm in O(n)
void norm(double[] out, double[] in) {
  double res = mag(in);
  for (int i=0; i<n; i++) {
    out[i] = in[i] / res;
  }
}
```

Optimization & Automatic Differentiation

$O\left(n^2\right)$

```
for i=0..n {
  out[i] /= mag(in)
}
```

Optimize

$O\left(n\right)$

```
res = mag(in)
for i=0..n {
  out[i] /= res
}
```

AD

$O\left(n\right)$

```
d_res = 0.0
for i=n..0 {
  d_res += d_out[i]…
}
∇mag(d_in, d_res)
```

Optimization & Automatic Differentiation

$O\left(n^2\right)$

```
for i=0..n {
  out[i] /= mag(in)
}
```

Optimize

$O\left(n\right)$

```
res = mag(in)
for i=0..n {
  out[i] /= res
}
```

AD

$O\left(n\right)$

```
d_res = 0.0
for i=n..0 {
  d_res += d_out[i]…
}
∇mag(d_in, d_res)
```

$O\left(n^2\right)$

```
for i=0..n {
  out[i] /= mag(in)
}
```

AD

$O\left(n^2\right)$

```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

Optimization & Automatic Differentiation

$O\left(n^2\right)$

```
for i=0..n {
  out[i] /= mag(in)
}
```

Optimize

$O\left(n\right)$

```
res = mag(in)
for i=0..n {
  out[i] /= res
}
```

AD

$O\left(n\right)$

```
d_res = 0.0
for i=n..0 {
  d_res += d_out[i]…
}
∇mag(d_in, d_res)
```

$O\left(n^2\right)$

```
for i=0..n {
  out[i] /= mag(in)
}
```

AD

$O\left(n^2\right)$

```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

Optimize

$O\left(n^2\right)$

```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

Optimization & Automatic Differentiation

Differentiating after optimization can create ***asymptotically faster*** gradients!

$O\left(n^2\right)$

```
for i=0..n {
  out[i] /= mag(in)
}
```

Optimize

$O\left(n\right)$

```
res = mag(in)
for i=0..n {
  out[i] /= res
}
```

AD

$O\left(n\right)$

```
d_res = 0.0
for i=n..0 {
  d_res += d_out[i]…
}
∇mag(d_in, d_res)
```

$O\left(n^2\right)$

```
for i=0..n {
  out[i] /= mag(in)
}
```

AD

$O\left(n^2\right)$

```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

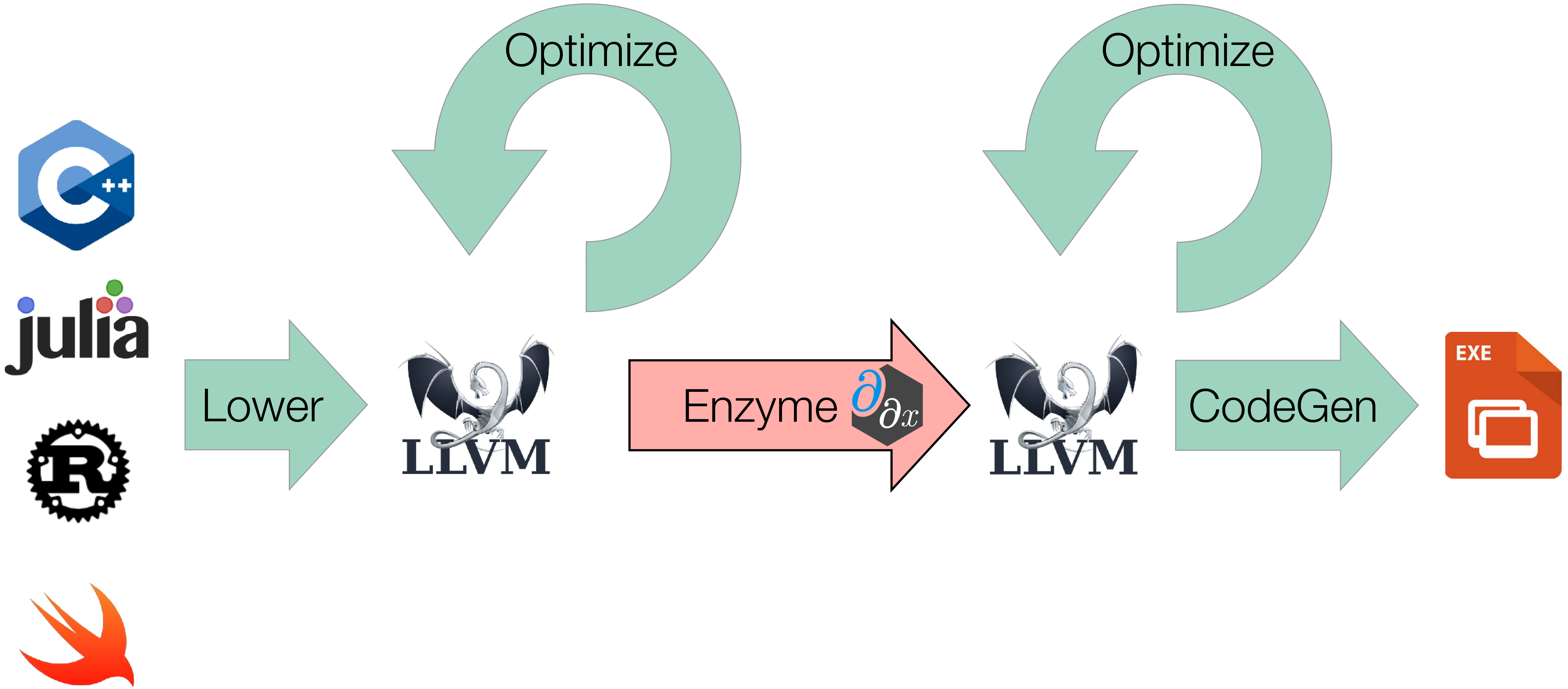
Optimize

$O\left(n^2\right)$

```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

Enzyme Approach

Performing AD at low-level lets us work on ***optimized*** code!



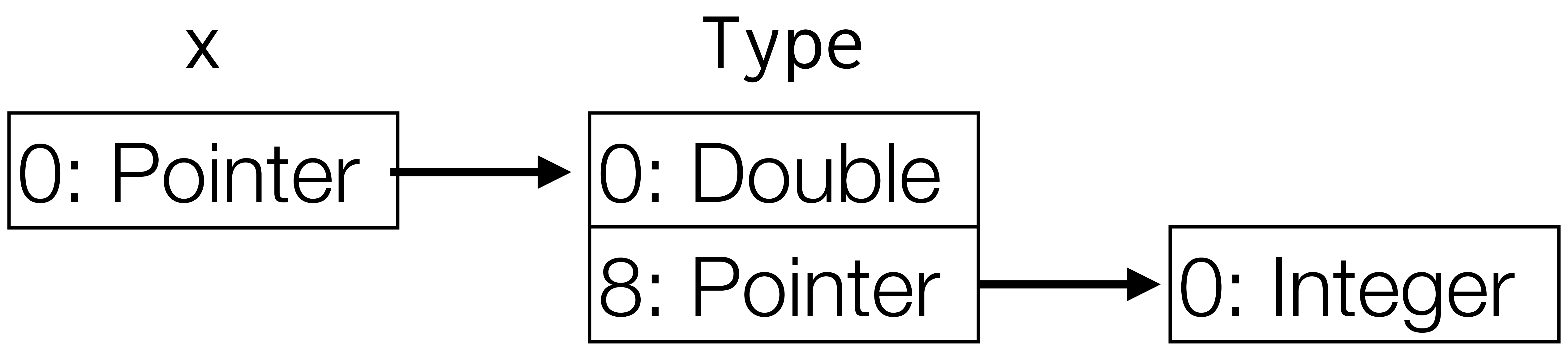
Challenges of Low-Level AD

- Low-level code lacks information necessary to compute adjoints

- ***Solution:*** Created new interprocedural analyses to derive information and optimize

```
struct Type {
  double;
  int*;
}

x = Type*;
```

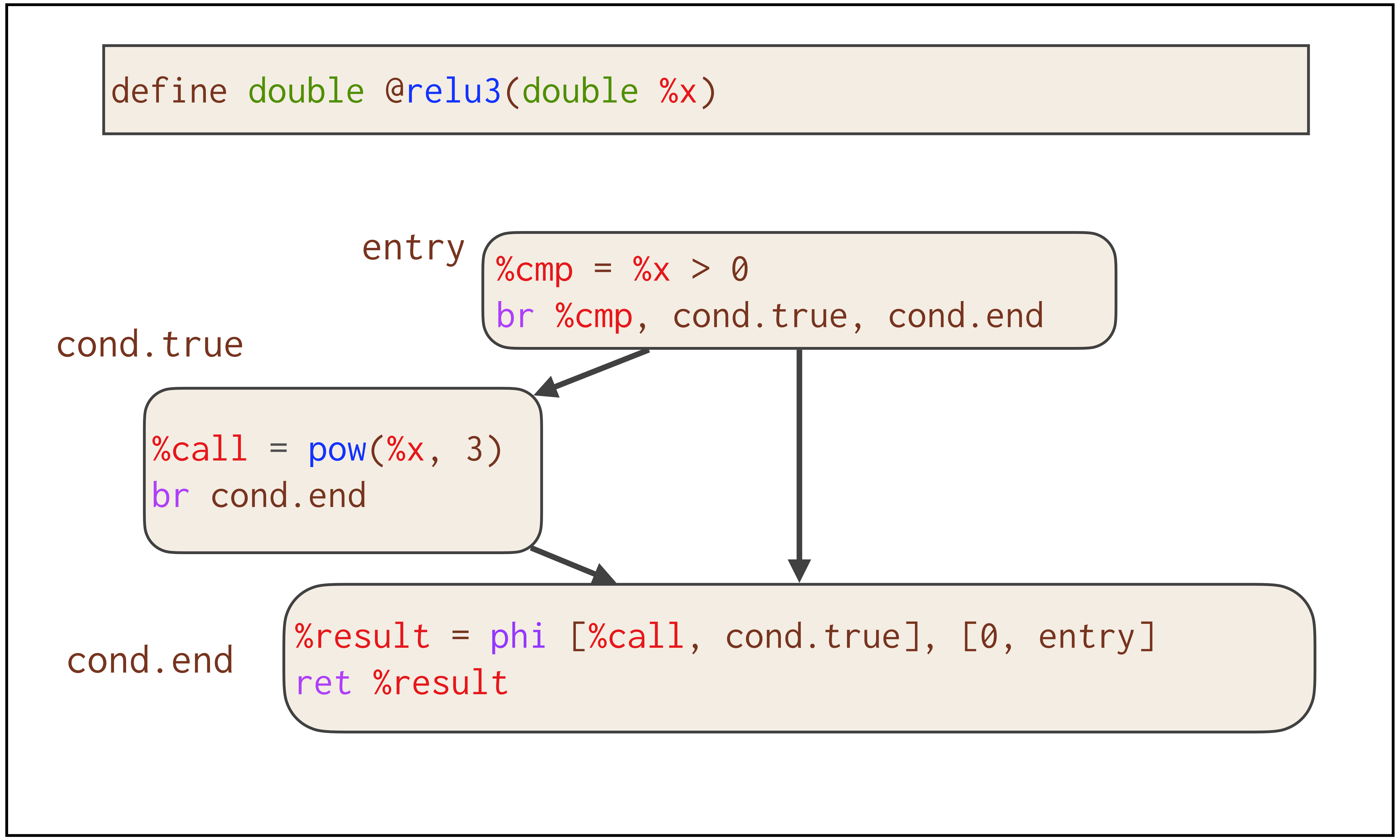


Case Study: ReLU3

C Source

```c
double relu3(double x) {
  double result;
  if (x > 0)
    result = pow(x, 3);
  else
    result = 0;
  return result;
}
```

LLVM

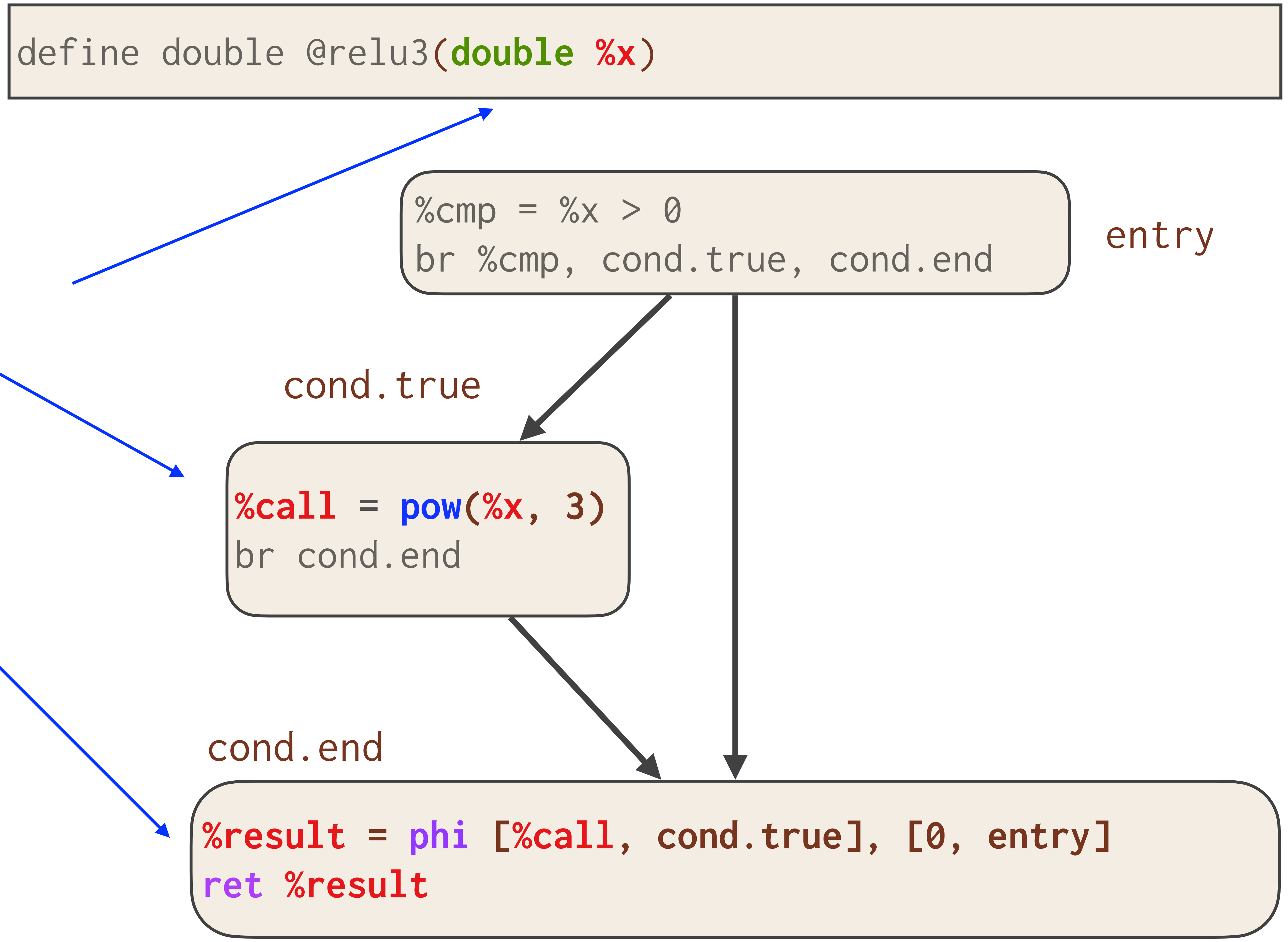


Enzyme Usage

```c
double diffe_relu3(double x) {
  return __enzyme_autodiff(relu3, x);
}
```

Case Study: ReLU3

Active Instructions



```
define double @diffe_relu3(double %x, double %differet)
```

entry

```
alloca %result' = 0.0
alloca %call'   = 0.0
alloca %x'      = 0.0
%cmp = %x > 0
br %cmp, cond.true, cond.end
```

Allocate & zero shadow memory for active values

cond.true

```
%call = pow(%x, 3)
br cond.end
```

cond.end

```
%result = phi [%call, cond.true], [0, entry]

; deleted return

%result' = 1.0
br reverse_cond.end
```

Compute adjoints for active instructions

```
define double @diffe_relu3(double %x, double %differet)
```

entry
```
alloca %result' = 0.0
alloca %call'   = 0.0
alloca %x'      = 0.0
%cmp = %x > 0
br %cmp, cond.true, cond.end
```

cond.true
```
%call = pow(%x, 3)
br cond.end
```

cond.end
```
%result = phi [%call, cond.true], [0, entry]

; deleted return

%result' = 1.0
br reverse_cond.end
```

reverse_cond.end
```
%tmp_res' = load %result'
%call' += if %x > 0 then %tmp_res' else 0
store %result' = 0.0
br %cmp, reverse_cond.true, reverse_entry
```

reverse_cond.true
```
%df = 3 * pow(%x, 2)
%tmp_call' = load %call
%x' += %df * %tmp_call'
store %call' = 0.0
br reverse_entry
```

reverse_entry
```
%0 = load %x'
ret %0
```

Compute adjoints for active instructions

```
define double @diffe_relu3(double %x, double %differet)
```

entry

```
alloca %result’ = 0.0
alloca %call’   = 0.0
alloca %x’      = 0.0
%cmp = %x > 0
br %cmp, cond.true, cond.end
```

cond.true

```
%call = pow(%x, 3)
br cond.end
```

cond.end

```
%result = phi [%call, cond.true], [0, entry]

; deleted return

%result’ = 1.0
br reverse_cond.end
```

reverse_cond.end

```
%tmp_res’ = load %result’
%call’ += if %x > 0 then %tmp_res’ else 0
store %result’ = 0.0
br %cmp, reverse_cond.true, reverse_entry
```

reverse_cond.true

```
%df = 3 * pow(%x, 2)
%tmp_call’ = load %call
%x’ += %df * %tmp_call’
store %call’ = 0.0
br reverse_entry
```

reverse_entry

```
%0 = load %x’
ret %0
```

```
define double @diffe_relu3(double %x)
```

Post Optimization

entry

```
%cmp = %x > 0
br %cmp, reverse_cond.true, reverse_entry
```

reverse_cond.true

```
%3 = 3 * pow(%x, 2)
br reverse_entry
```

reverse_entry

```
%0 = phi [%3, reverse_cond.true], [0, entry]
ret %0
```

Essentially the optimal hand-written gradient!

```
double diffe_relu3(double x) {
  double result;
  if (x > 0)
    result = 3 * pow(x, 2);
  else
    result = 0;
  return result;
}
```

PyTorch-Enzyme & TensorFlow-Enzyme

```python
import torch
from torch_enzyme import enzyme

# Create some initial tensor
inp = …

# Apply foreign function to tensor
out = enzyme("test.c", “f").apply(inp)

# Derive gradient
out.backward()
print(inp.grad)
```

```python
import tensorflow as tf
from tf_enzyme import enzyme

inp = tf.Variable(…)
# Use external C code as a regular TF op

out = enzyme(inp, filename=“test.c",
                  function=“f”)

# Results is a TF tensor
out = tf.sigmoid(out)
```

```c
// Input tensor + size, and output tensor
void f(float* inp, size_t n, float* out);

// diffe_dupnoneed specifies not recomputing the output
void diffef(float* inp, float* d_inp, size_t n, float* d_out) {
  __enzyme_autodiff(f, diffe_dup, inp, d_inp, n, diffe_dupnoneed, (float*)0, d_out);
}
```

Custom Derivatives & Multisource

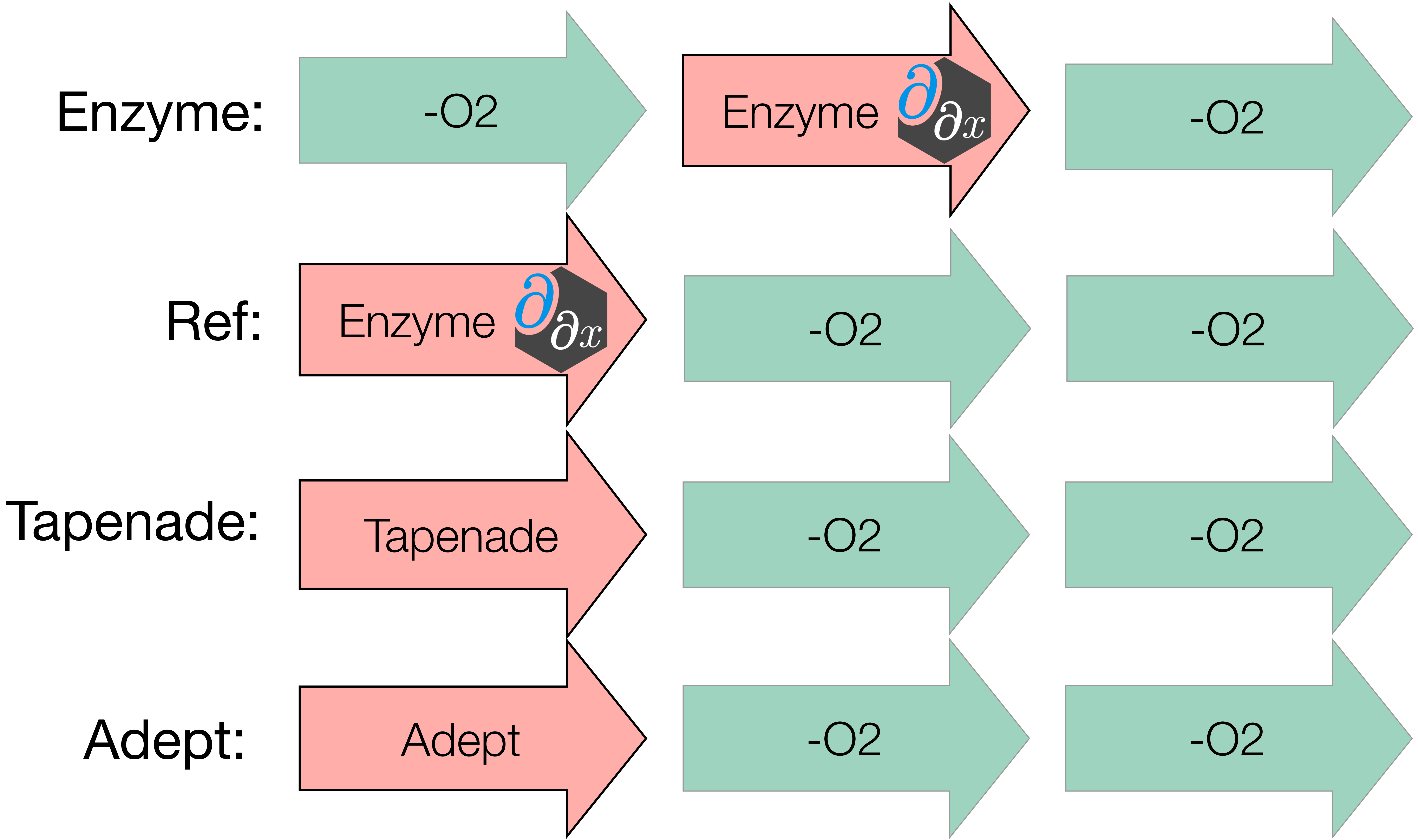
- One can specify custom forward/reverse passes of functions by attaching metadata

```
__attribute__((enzyme("augment", augment_func)))
__attribute__((enzyme("gradient", gradient_func)))
double func(double n);
```

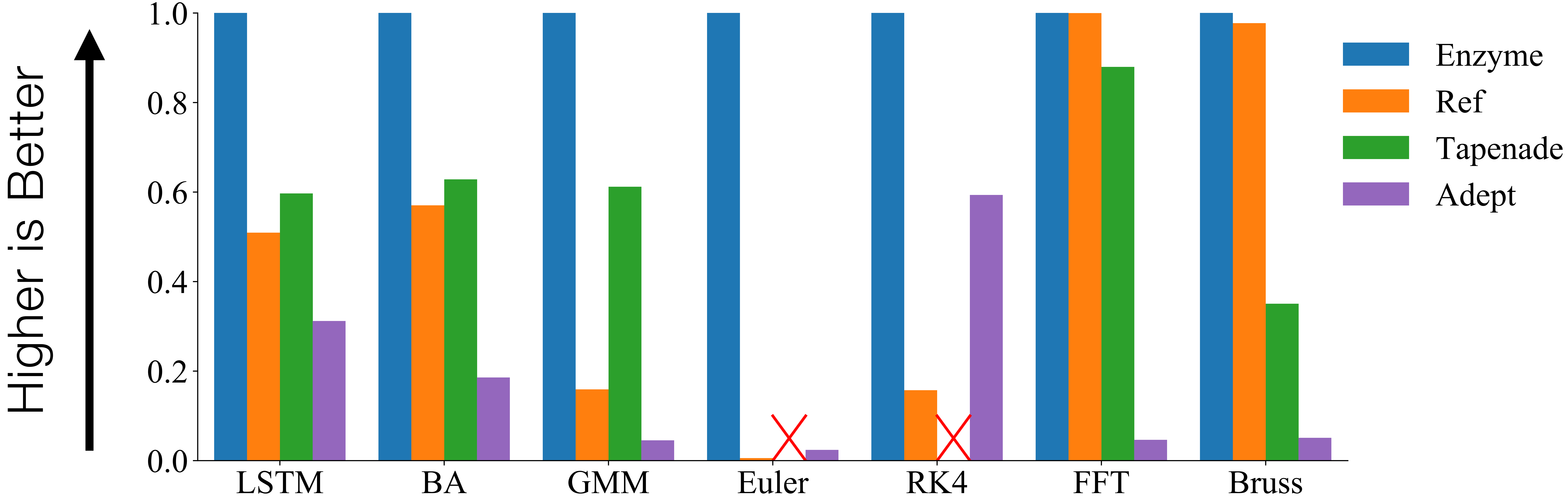
- Enzyme leverages LLVM's link-time optimization (LTO) & "fat libraries" to ensure that LLVM bitcode is available for all potential differentiated functions before AD

Experimental Setup

- Collection of benchmarks from Microsoft's ADBench suite and of technically interest



Speedup of Enzyme



Enzyme is ***4.2x faster*** than Reference!

Enzyme

- Tool for performing reverse-mode AD of statically analyzable LLVM IR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc)
- 4.2x speedup over AD before optimization
- State-of-the art performance with existing tools
- PyTorch-Enzyme & TensorFlow-Enzyme lets researchers use foreign code in ML workflow
- Open source (enzyme.mit.edu & join our mailing list)
- For more information come to our poster!

Acknowledgements

- Thanks to James Bradbury, Alex Chernyakhovsky, Hal Finkel, Laurent Hascoet, Paul Hovland, Jan Hueckelheim, Mike Innes, Tim Kaler, Charles Leiserson, Yingbo Ma, Chris Rackauckas, TB Schardl, Lizhou Sha, Yo Shavit, Dhash Shrivathsa, Nalini Singh, Miguel Young de la Sota, and Alex Zinenko

- William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DESC0019323.

- Valentin Churavy was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR0011-20-9-0016, and in part by NSF Grant OAC-1835443.

- This research was supported in part by LANL grant 531711. Research was sponsored by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000.

- The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government.

Enzyme

- Tool for performing reverse-mode AD of statically analyzable LLVM IR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc)
- 4.2x speedup over AD before optimization
- State-of-the art performance with existing tools
- PyTorch-Enzyme & TensorFlow-Enzyme lets researchers use foreign code in ML workflow
- Open source (enzyme.mit.edu & join our mailing list)
- For more information come to our poster!

END

Compiler Analyses Better Optimize AD

- Existing
- Alias analysis results that prove a function does not write to memory, we can prove that additional function calls do not need to be differentiated since they cannot impact the output
- Don’t cache equivalent values
- Statically allocate caches when a loop’s bounds can be determined in advance

Decomposing the “Tape”

- Performing AD on a function requires data structures to compute
    - All values necessary to compute adjoints are available [cache]
    - Place to store adjoints [shadow memory]
    - Record instructions [we are static]
- Creating these directly in LLVM allows us to explicitly specify their behavior for optimization, unlike approaches that call out to a library
- For more details look in paper

The “memcpy” Problem

```
void f(void* dst, void* src) {
  memcpy(dst, src, 8);
}
```

```
void grad_f(double* dst, double* dst’,
            double* src, double* src’) {
  // Forward Pass
  memcpy(dst, src, 8);

  // Reverse Pass
  src’[0] += dst’[0];
  dst’[0] = 0;
}
```

```
void grad_f(float* dst, float* dst’,
            float* src, float* src’) {
  // Forward Pass
  memcpy(dst, src, 8);

  // Reverse Pass
  src’[0] += dst’[0];
  dst’[0] = 0;
  src’[1] += dst’[1];
  dst’[1] = 0;
}
```