Enzyme: High-Performance Automatic Differentiation of LLVM

William S. Moses

Valentin Churavy

wmoses@mit.edu

LLVM Developers' Meeting

October 8, 2020

CSAIL

Automatic Differentiation

- Many algorithms require the derivatives of various functions
  - Machine learning (back-propagation, Bayesian inference, uncertainty quantification)
  - Scientific computing (modeling, simulation)
- When working with large codebases or dynamically-generated programs, manually writing derivative functions becomes intractable
- Community has developed tools to automatically create derivatives automatically

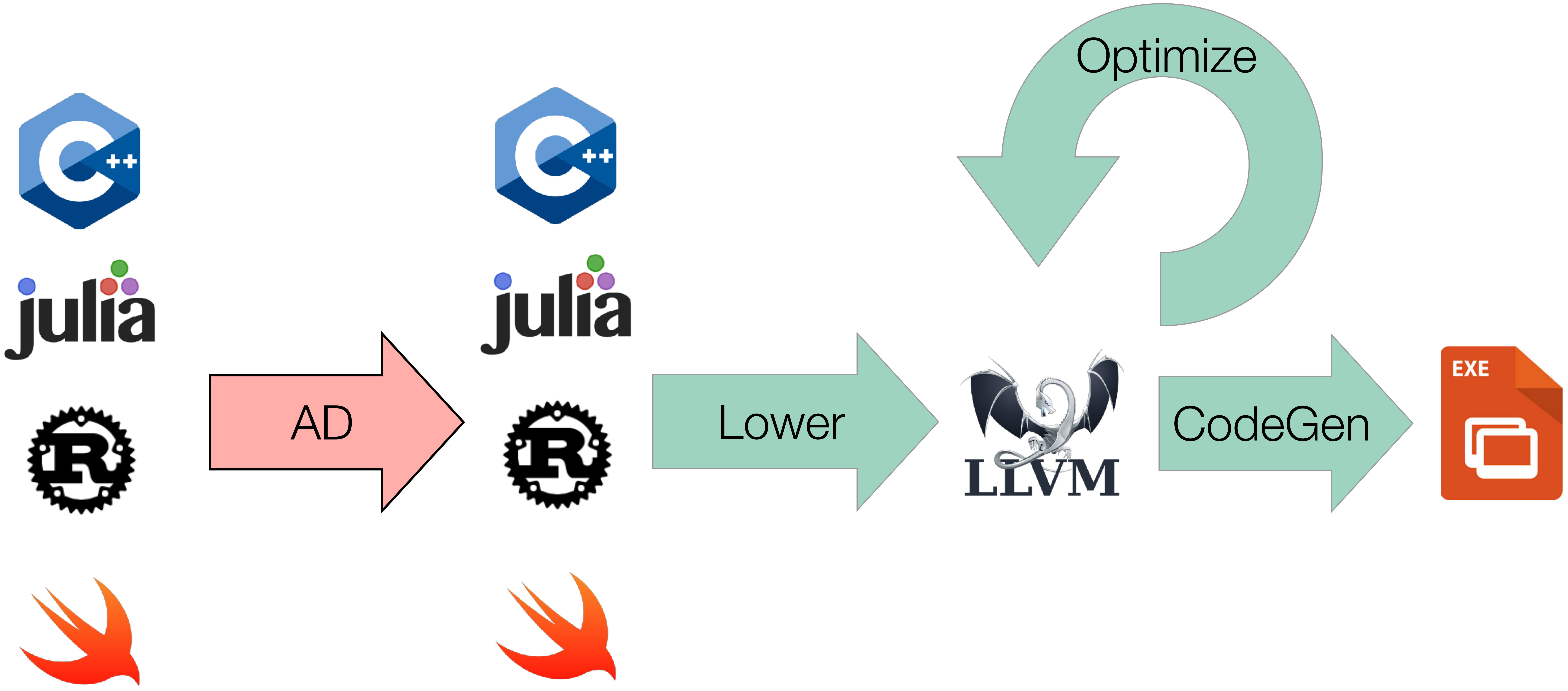
Existing AD Approaches

- Differentiable DSL (TensorFlow, PyTorch, DiffTaichi)
  - Provide a new language where all functions are differentiable
  - Requires rewriting everything in the DSL and the DSL must support all operations in original code
  - Fast if DSL matches original code well
- Operator overloading (Adept, JAX)
  - Provide differentiable versions of existing language constructs (double => adouble, np.sum => jax.sum)
  - May require writing to use non-standard utilities
  - Often dynamic: storing instructions/values to later be interpreted

Existing AD Approaches

- Source rewriting
  - Statically analyze program to produce a new gradient function in the source language
  - Requires all code to be available ahead of time
  - Difficult to use with external libraries

Existing AD Pipelines



Case Study: Vector Normalization

```
//Compute magnitude in O(n)
double mag(double* x, size_t n);

//Compute norm in O(n^2)
void norm(double* out, double* in, size_t n) {
 for(int i=0; i<n; i++) {
    out[i] = in[i]/mag(in, n);
  }
}
```

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

```
double mag(double* x, size_t n);

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

Loop Invariant Code Motion

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

```
double mag(double* x, size_t n);

void norm(double* out, double* in, size_t n) {
  double res = mag(in, n);

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

LICM then Differentiate

```
void dnorm(double* out, double* dout,
           double* in,  double* din, size_t n) {
  double res = mag(in, n);

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

  double d_res = 0;
  for(int i=0; i<n; i++) {
    dres += -in[i]*in[i]/res * dout[i];
    din[i] += dout[i]/res;
  }

  dmag(in, din, n, dres);
}
```

$O(n)$

$O(n)$

Just Differentiate

```
void dnorm(double* out, double* dout,
           double* in,  double* din, size_t n) {

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

  for(int i=0; i<n; i++) {
    double dres = -in[i]*in[i]/mag * dout[i];
    din[i] += dout[i]/mag;
    dmag(in, din, n, dres);
  }
}
```

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

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

Differentiate then LICM

```
void dnorm(double* out, double* dout,
           double* in,  double* din, size_t n) {

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

  for(int i=0; i<n; i++) {
    double dres = -in[i]*in[i]/res * dout[i];
    din[i] += dout[i]/res;
    dmag(in, din, n, dres);
  }
}
```

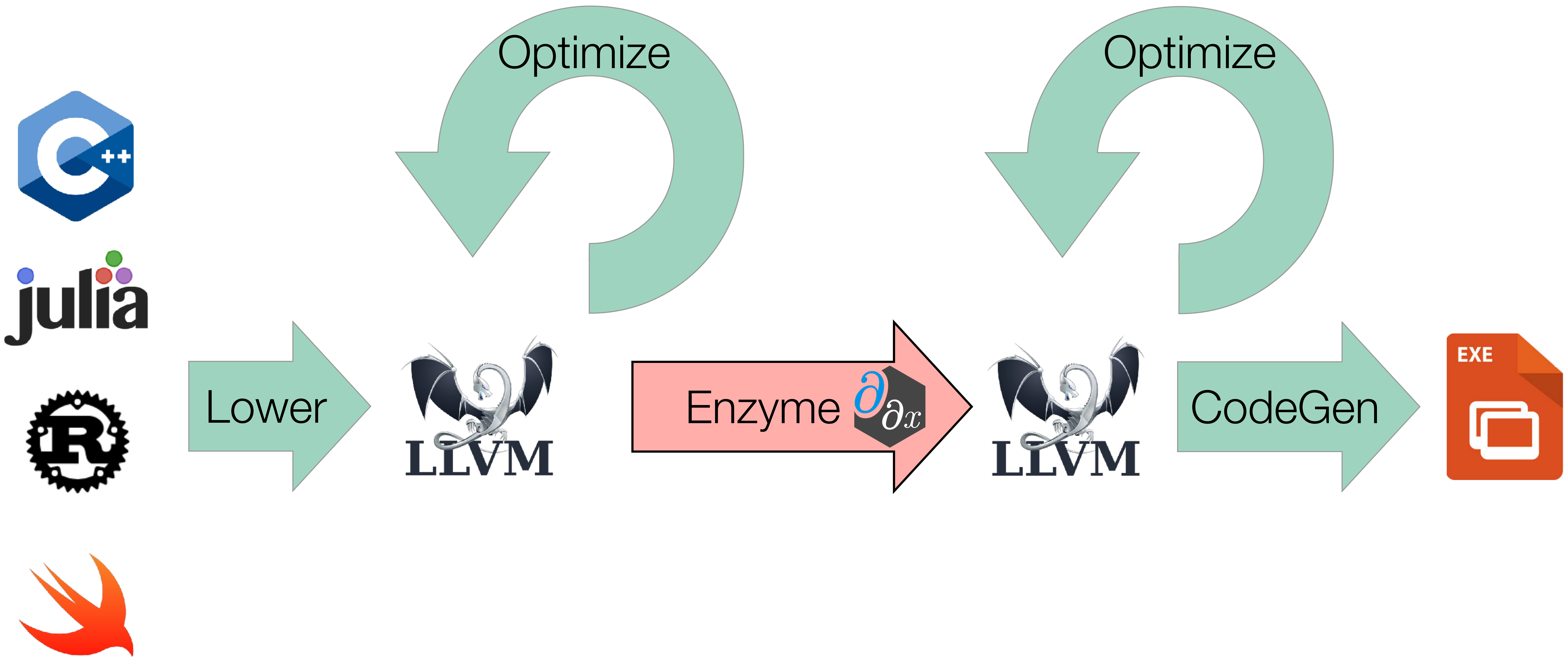
$O(n)$

$O(n^2)$

Can't LICM as `dmag` uses loop-local variable `dres`

Enzyme Approach

Perform AD on ***optimized*** programs!



How to Achieve Post-Optimization AD

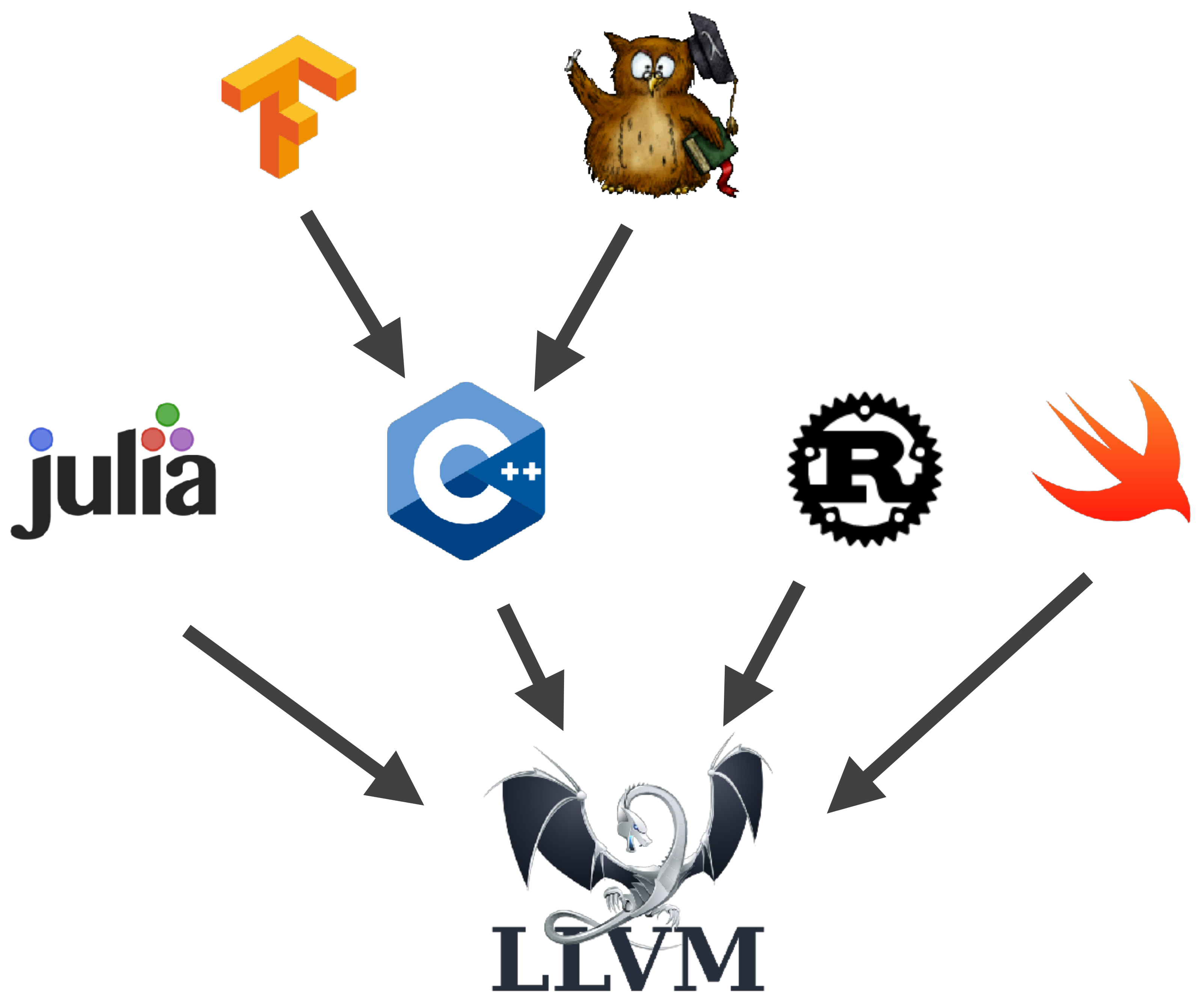
- Implement all optimizations in AD system
  - Embed a compiler into your AD
  - Rewrite all compiler analyzes and optimizations
- Perform AD on low-level post-optimization representation
  - Embed AD into your compiler
  - “AD is more effective in high-level compiled languages (e.g. Julia, Swift, Rust, Nim) than traditional ones such as C/C++, Fortran and LLVM IR […]” -Innes

Enzyme

- Reverse-mode source-rewriting AD plugin for statically analyzable LLVM IR
- 4.5x speedup over AD before optimization
- State-of-the art performance with existing tools
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc)
- PyTorch-Enzyme & TensorFlow-Enzyme packages let researchers use foreign code in their ML workflow
- Multisource AD & library support by leveraging LTO

Why LLVM?

- Generic low-level compiler infrastructure with many frontends
    - “Cross platform assembly”
- Well-defined semantics
- Large collection of optimizations
- Analysis passes can be used as utilities



Enzyme Differentiation Algorithm

- Type Analysis
- Activity Analysis
- Synthesize derivatives
  - Forward pass that mirrors original code
  - Reverse pass inverts instructions in forward pass (adjoints) to compute derivatives
- Optimize

The “memcpy” Problem

- Taking the derivative of operations such as memcpy memcpy depends on the type of the data being copied
    - e.g. one derivative for pointers, one for doubles, another for floats
- LLVM Types != C/C++ types

Case Study: Read Sum

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

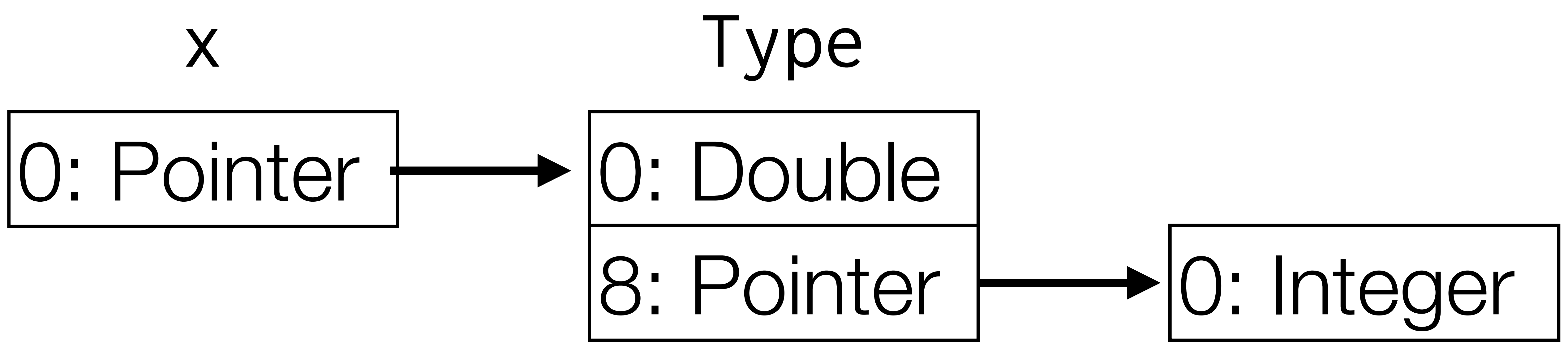
```
void grad_f(double* dst, double* dst',
            double* src, double* src') {
  // Forward Pass
  memcpy(dst, src, 8);

  // 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;
}
```

Type Analysis

- New interprocedural dataflow analysis that detects the underlying type of data
- Each value has a set of memory offsets : type

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

x = Type*;
```




```
types(x) = {[0]:Pointer, [0,0]:Double, [0,8]:Pointer, [0,8,0]:Integer}
```

Type Analysis

- Initialize type trees
  - Constant, TBAA, and known instructions
- Perform series of fixed-point updates
  - Each instruction has a type propagation rule

```
z{} = add x{0:Int}, y{0:Int}
              ↓
z{0:Int} = add x{0:Int}, y{0:Int}
```

- Provide a compile-time error if a necessary type cannot be deduced statically

Type Analysis

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
callee:
```

```
void callee(int* ptr) {
ptr:      {[0]:Pointer, [0,16]:Double, [0,24]:Int}
ptr2:     {[0]:Pointer, [0,0]:Double, [0,8]:Int}
loadtype: {[0]:Double}
ptr3:     {[0]:Pointer, [0,0]:Int}
cptr2:    {[0]:Pointer, [0,0]:Double, [0,8]:Int}
notype:   {[0]:Double}
cptr3:    {[0]:Pointer, [0,0]:Int}
```

Activity Analysis

- Determines what instructions could impact derivative computation
- Avoids taking meaningless or unnecessary derivatives (e.g. d/dx cpuid)
- Instruction is active iff it can propagate a differential value to its return or memory
- Build off of alias analysis & type analysis
  - E.g. all read-only function that returns an integer are inactive since they cannot propagate adjoints through the return or to any memory location

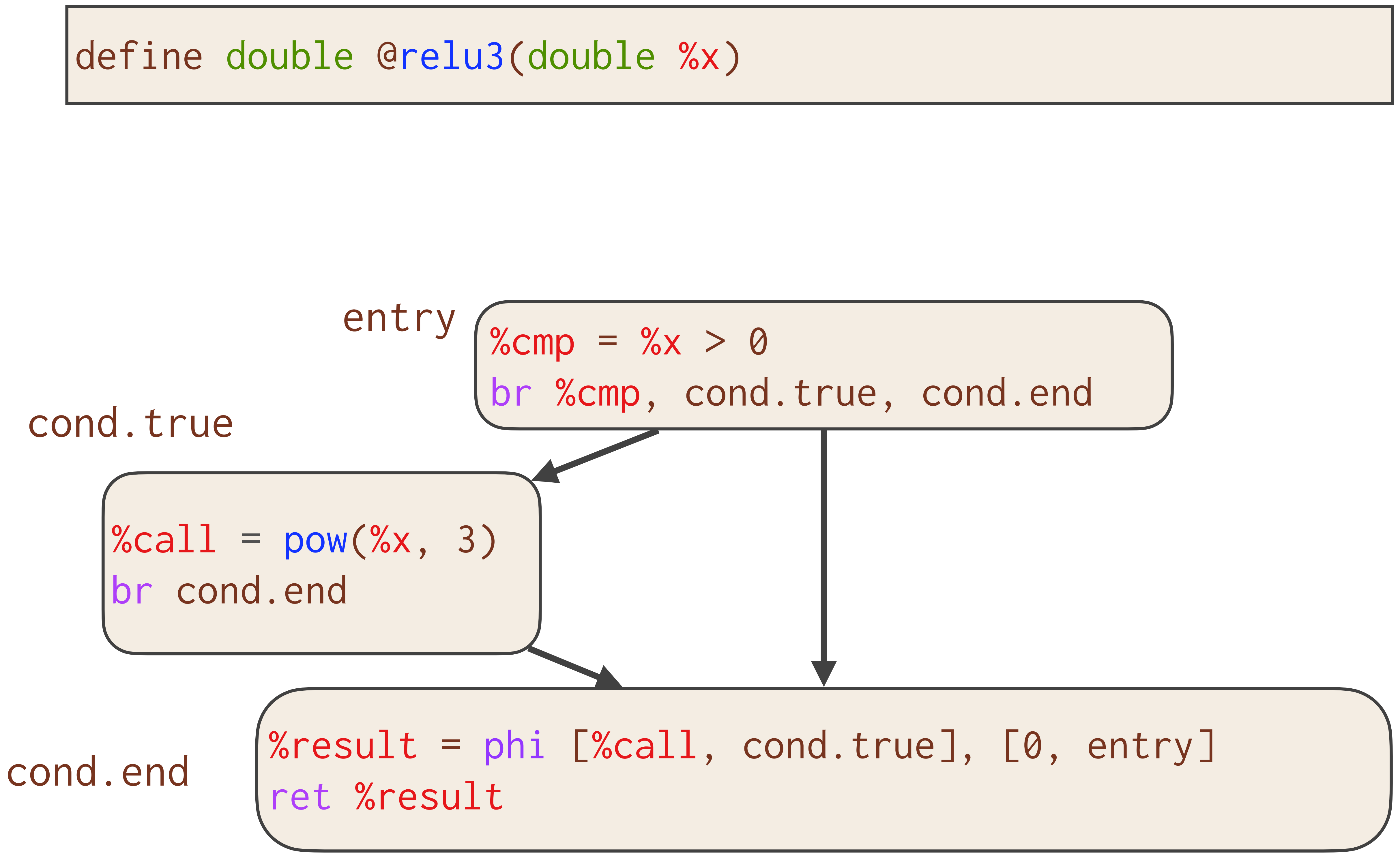
Shadow Memory

- Derivatives of values are stored in shadow allocations
- For all active values, allocate and zero shadow memory to store the derivative of all of its occurrences
- All data structures need to have a shadow data structure created
  - Enzyme will create shadow allocation/stores for structures created inside code being differentiated
  - Data structures passed as arguments will pass shadow arguments

Derivative Synthesis

- Initialize shadow memory
- For each BasicBlock BB:
  - For each Instruction I in reverse(BB):
    - Emit adjoint I, caching and reloading any necessary values from the forward pass

Case Study: ReLU3

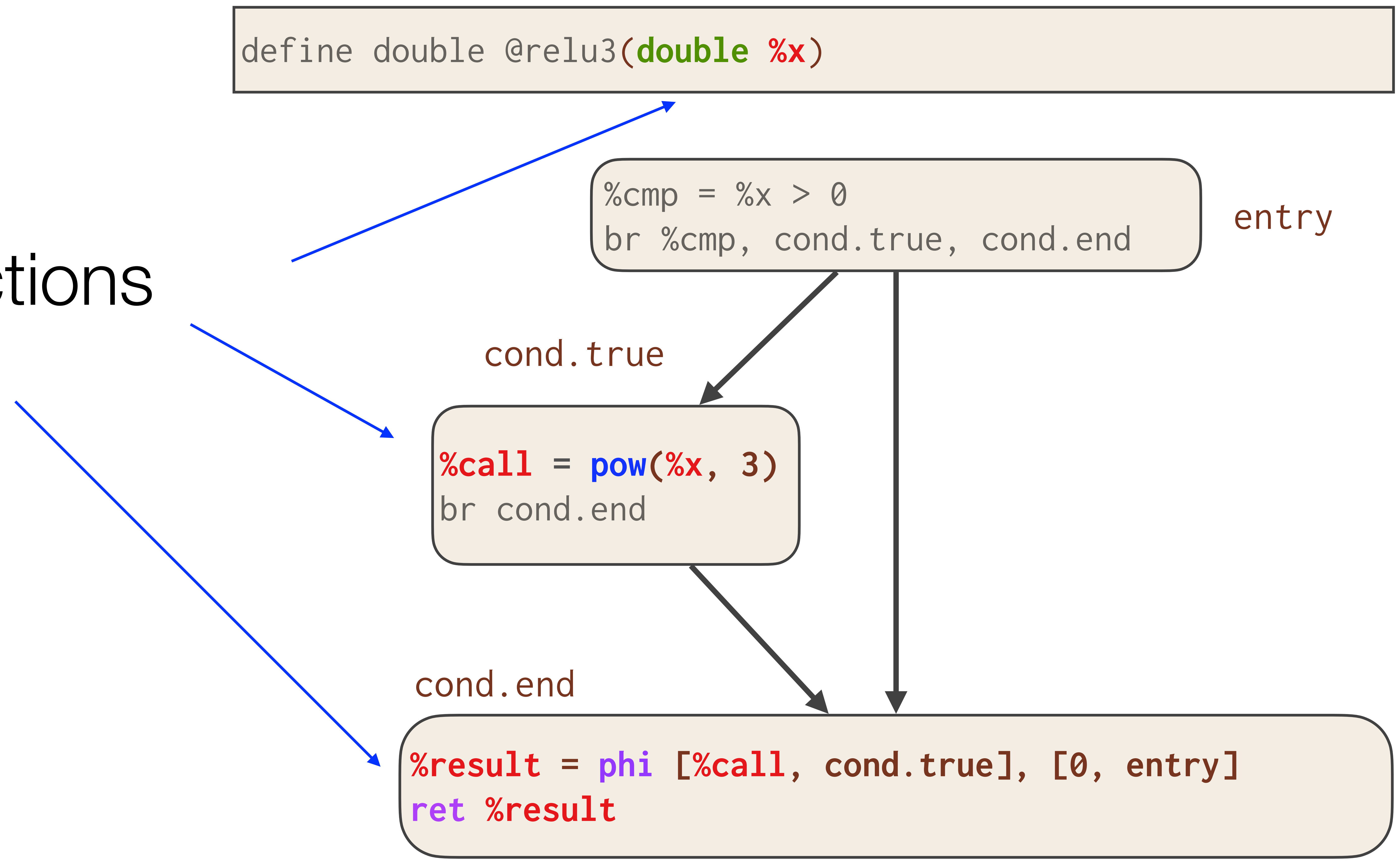



```c
double relu3(double x) {
  double result;
  if (x > 0)
    result = pow(x, 3);
  else
    result = 0;
  return result;
}
```

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

Case Study: ReLU-f

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 instructions

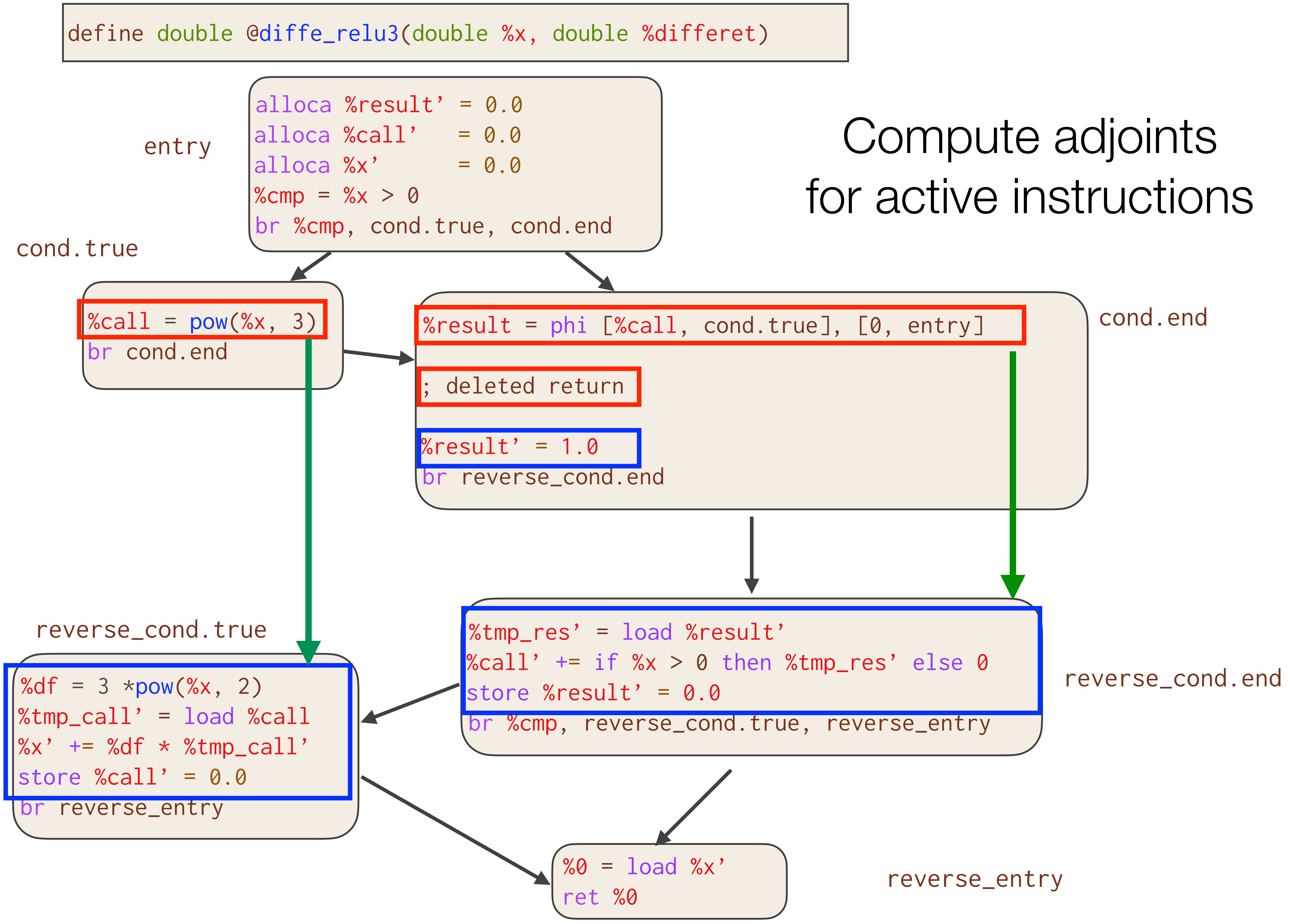
cond.true

```
%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)
```

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-compiled program!

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

Cache

- Adjoint instructions may require values from the forward pass
  - e.g. ∇(x * y) => x dy + y dx
- For all such values, allocate memory in the function header to store the value for use in the reverse pass
- Values computed inside loops are stored in an array indexed by the loop induction variable
  - Array allocated statically if possible; otherwise dynamically `realloc`’d

Case Study: Read Sum

```
double sum(double* x) {
  double total = 0;

  for(int i=0; i<10; i++)
    total += read() * x[i];

  return total;
}
```

```
void diffe_sum(double* x,
                    double* xp) {
  return
   __enzyme_autodiff(sum, x, xp);
}
```

```
define double @sum(double* %x)
```

entry

```
br for.body
```

for.body

```
%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
%0 = load %x[%i]
%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body
```

for.cleanup

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

Case Study: Read Sum

Active Variables

```
define double @sum(double* %x)
```

entry

```
br for.body
```

for.body

```
%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
%0 = load %x[%i]
%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body
```

for.cleanup

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

Case Study: Read Sum

Each register in the for loop represents a distinct active variable every iteration

```
define double @sum(double* %x)
```

entry

```
br for.body
```

for.body

```
%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
%0 = load %x[%i]
%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body
```

for.cleanup

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

Allocate & zero shadow memory per active value

```
define double @diffe_sum(double* %x, double* %xp)
```

entry

```
alloca %x'      = 0.0
alloca %total'  = 0.0
alloca %0'      = 0.0
alloca %mul'    = 0.0
alloca %add'    = 0.0
alloca %result' = 0.0

br for.body
```

for.body

```
%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
%0 = load %x[%i]
%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body
```

for.cleanup

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

```
define double @diffe_sum(double* %x, double* %xp)
```

Cache forward pass variables for use in reverse

entry

```
alloca %x'      = 0.0
alloca %total'  = 0.0
alloca %0'      = 0.0
alloca %mul'    = 0.0
alloca %add'    = 0.0
alloca %result' = 0.0
%call_cache = @malloc(10 x double)
br for.body
```

for.body

```
%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
store %call_cache[%i] = %call
%0 = load %x[%i]
%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body
```

for.cleanup

```
%result = phi [ %call, cond.true], [0, entry]
@free(%cache)
ret %result
```

After lowering & some optimizations

```
define void @diffe_sum(double* %x, double* %xp)
```

entry

```
%call_cache = @malloc(10 x double)
br for.body
```

for.body

```
%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
store %call_cache[%i] = %call
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, reversefor.body, for.body
```

reversefor.body

```
%i' = phi [ 9, for.body ], [ %i’.next, reversefor.body ]
%i’.next = %i' - 1
%cached_read = load %call_cache[%i’]
store %xp[%i’] = %cached_read + %xp[%i’]
%exit2 = %i = 0
br %exitcond, %exit2, reversefor.body
```

exit

```
@free(%cache)
ret
```

Case Study: Read Sum

```
define void @diffe_sum(double* %x, double* %xp)
```

entry

```
%call0 = @read()
store %xp[0] = %call0
%call1 = @read()
store %xp[1] = %call1
%call2 = @read()
store %xp[2] = %call2
%call3 = @read()
store %xp[3] = %call3
%call4 = @read()
store %xp[4] = %call4
%call5 = @read()
store %xp[5] = %call5
%call6 = @read()
store %xp[6] = %call6
%call7 = @read()
store %xp[7] = %call7
%call8 = @read()
store %xp[8] = %call8
%call9 = @read()
store %xp[9] = %call9
ret
```

After more optimizations

```
void diffe_sum(double* x, double* xp) {
   xp[0] = read();
   xp[1] = read();
   xp[2] = read();
   xp[3] = read();
   xp[4] = read();
   xp[5] = read();
   xp[6] = read();
   xp[7] = read();
   xp[8] = read();
   xp[9] = read();
}
```

Cache Optimizations

- By carefully caching in a form LLVM understands, existing optimization passes can optimize the memory away! [*]
- Further optimizations:
  - Use alias analysis to prove that recomputing an instruction is legal
  - Don't cache unnecessary values
  - Don't cache a value that already has already been cached elsewhere

[*] For dynamic loop optimizations, LLVM must understand semantics of `realloc`.

Function Calls

- Computing both forward and reverse pass in the same function allows further optimization and reduces memory usage
    - Enzyme uses Alias Analysis to detect legality of computing forward/reverse pass together
- Otherwise, Enzyme may need to modify forward pass to cache values needed by reverse pass

Indirect Function Calls

- Calls to functions that aren't known at compile time are dealt with by leveraging shadow memory
- The shadow of function pointers is defined to be a global containing the forward and reverse pass
- Thus taking the adjoint of an indirect function call simply requires extracting and calling the corresponding shadow callee

Custom Derivatives & Multisource

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

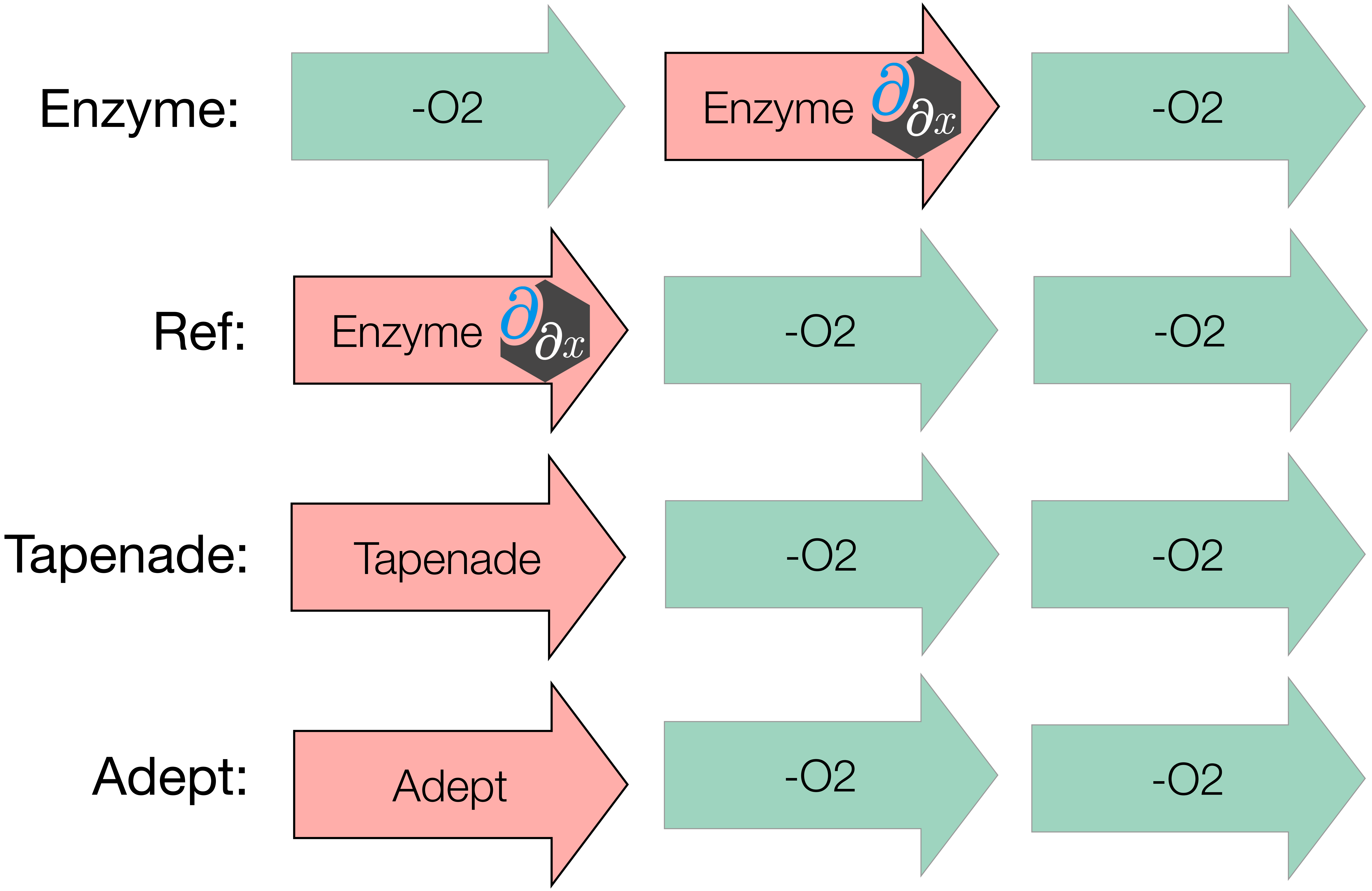
```
__attribute__((enzyme("augment", augment_func)))
__attribute__((enzyme("gradient", gradient_func)))
double func(double n);
```

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

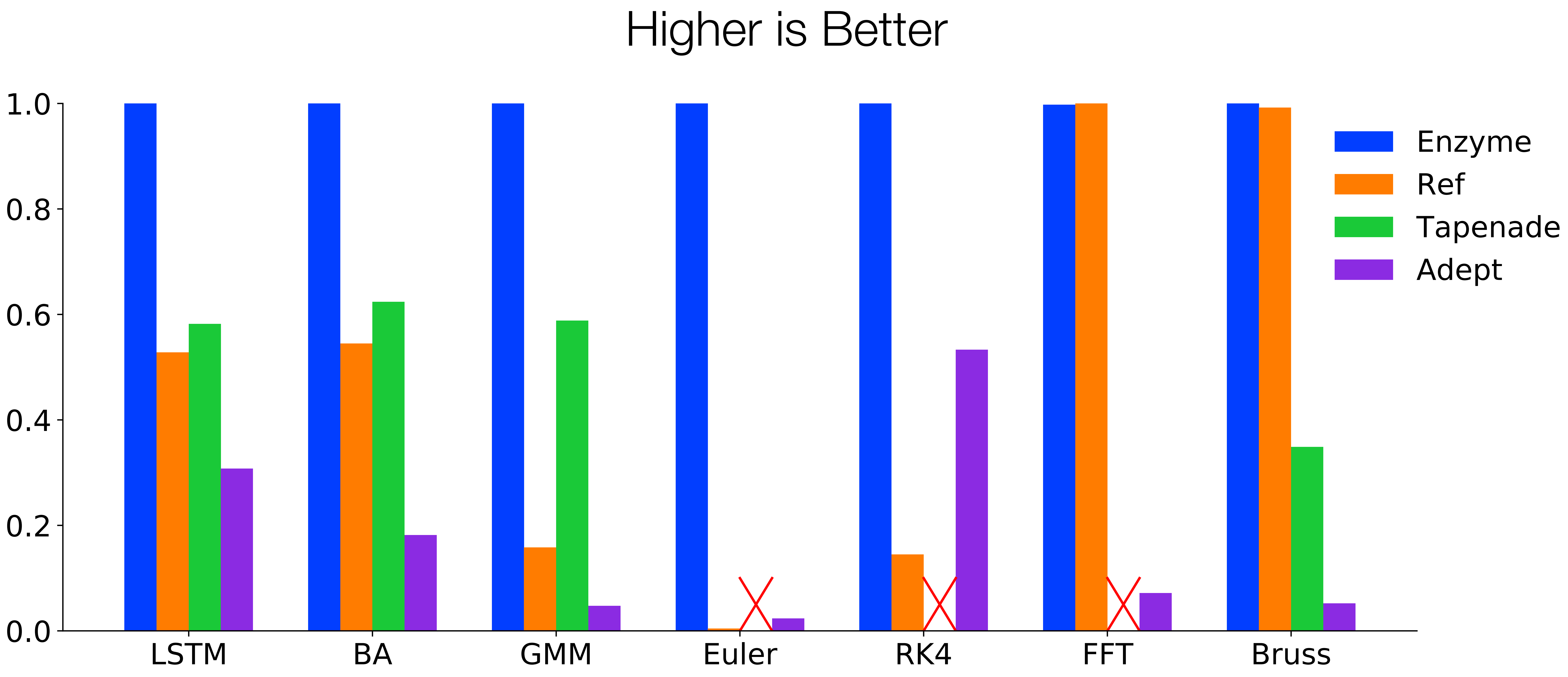
Evaluation

- Collection of benchmarks from Microsoft's ADBench suite and of technically interest
- Evaluated Enzyme, Reference, and the two fastest AD systems from ADBench (Tapenade, Adept)
- All programs run serially
- Quiesed Amazon c4.8xlarge (disabled turbo-boost; hyper-threading)

Experimental Setup



Relative Speedup



Speedup of 0.5 denotes program took twice as long as Speedup of 1.0

Runtime

	Enzyme	Ref	Tapenade	Adept
LSTM	**2.353**	4.458	4.042	7.645
BA	**0.424**	0.778	0.680	2.334
GMM	**0.073**	0.462	0.124	1.544
Euler	**0.161**	36.723	nan	6.851
RK4	**3.397**	23.442	nan	6.371
FFT	**0.183**	**0.182**	nan	2.538
Bruss	**0.181**	**0.182**	0.518	3.457

Enzyme is 4.5x faster than Ref!

ML Framework Integration

```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);
}
```

Conclusions

- AD on low-level IR can be fast
- Optimization before AD is crucial
- Enzyme provides high-performance cross-language AD
- Open-sourcing now (visit enzyme.mit.edu & join our mailing list)
  - Hope to upstream as LLVM project
- Future Work:
  - Parallelism, GPU AD
  - AD-specific optimizations

Acknowledgements

- Thanks to James Bradbury, Tim Kaler, Charles Leiserson, Yingbo Ma, Chris Rackauckas, TB Schardl, Yo Shavit, Dhash Shrivathsa, Nalini Singh, and Alex Zinenko

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

- 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.

Conclusions

- AD on low-level IR can be fast
- Optimization before AD is crucial
- Enzyme provides high-performance cross-language AD
- Open-sourcing now (visit enzyme.mit.edu & join our mailing list)
    - Hope to upstream as LLVM project
- Future Work:
    - Parallelism, GPU AD
    - AD-specific optimizations

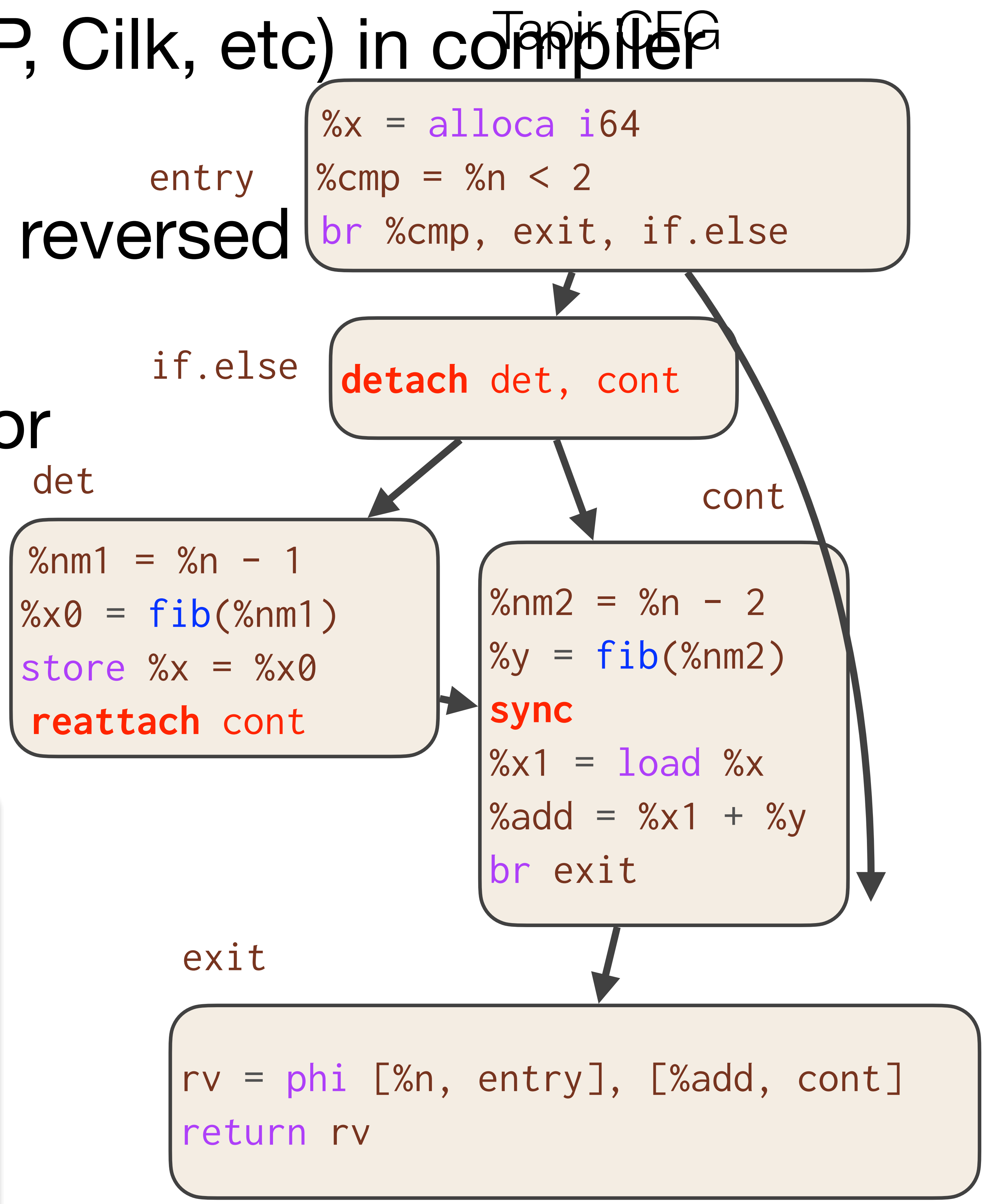
Backup Slides

Requirements & Performance Boosts

- Requirements
  - Enable TBAA (Type based alias analysis)
  - Strict Aliasing (no unions)
  - Disable exceptions
- Performance Boosts
  - Disable Loop Unrolling before AD
  - Disable Vectorization before AD

Future Work: Parallelism*

- Build off prior work [1] representing parallelism (OpenMP, Cilk, etc) in compiler
- Reverse pass can remain in parallel, with dependencies reversed
- Updates to adjoints in parallel tasks done with reducer or atomic add to prevent races




```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  x = spawn fib(n - 1);
  y = fib(n - 2);
  sync;
  return x + y;
}
```

[1] Tapir; Tao. B Schardl, **William S Moses**, Charles E. Leiserson; PPoPP 2017

[*] Work in progress — suggestions appreciated

Benchmarks

- LSTM: Long-short term memory model
- BA: Bundle analysis
- GMM: Gaussian mixture model
- Euler: Euler integration
- RK4: Runge-Kutta integration
- FFT: Fast Fourier transform
- Bruss: Brusselrator chemical simulation

Matrix Vector: Single Iteration

- LLVM Optimization Passe
- Constant (wrt inputs) detecti
-

```
#define N 20000
#define M 20000
#define ITERS 1
```

	Enzyme	Adept
Normal	1.119	0.0006
Forward	1.119	11.016
Forward +Reverse	1.210	13.445

Taylor Expand Log

- LLVM Optimization Passe
- Constant (wrt inputs) detection
-

```
tic adouble logger(adouble x) {
    adouble sum = 0;
  for(int i=1; i<=ITERS; i++) {
    sum += pow(x, i) / i;
  }
    return sum;
}

ger_and_gradient(double xin, double& xgrad) {
    adept::Stack stack;
    adouble x = xin;
    stack.new_recording();
    adouble y = logger(x);
    y.set_gradient(1.0);
    stack.compute_adjoint();
    xgrad = x.get_gradient();
    return y.value();
```

Taylor Expand Log (Julia)

- LLVM Optimization Passe
- Constant (wrt inputs) detection
-

$$f(x) = \sum_{i=1}^{N} \frac{x^i}{i} \approx -\log(1-x)$$

```
       #define ITERS 10000000
double logger(double x) {
  double sum = 0;

  for(int i=1; i<=ITERS; i++)
    sum += pow(x, i) / i;

  return sum;
}
```

```
function jl_f1(f::Float64)
  sum = 0 * f;
  for i = 1:10000000
    sum += f^i / i;
end
  return sum;
end
```

$$\frac{\partial}{\partial x} f(x) \approx \frac{1}{1-x}$$

$$\frac{\partial}{\partial x} f(x=0.5) \approx 2$$

```
       ; Enzyme derivative code
@show autodiff(fl_f1, 0.5)
@time autodiff(fl_f1, 0.5)
```

```
using Zygote
@show jl_f1'(0.5)
@time jl_f1'(0.5)
```

Taylor Expand Log

- LLVM Optimization Passe
- Constant (wrt inputs) detection
-

10000000 iterations

	Enzyme	Adept	Enzyme-Julia	Zygote-Julia	AutoGrad-Julia
Normal	3.74	3.72	3.82	3.82	3.82
Forward	3.74	4.56	3.82	3.82	3.82
Forward +Revers	3.90	4.65	3.95	44.694	896.30

LogSumExp

- LLVM Optimization Passe
- Constant (
-

```c
#define N 10000000
double logsumexp(double* x, size_t n) {
  double A = 0;
  for(int i=1; i < n; i++) {
    A = max(A, x[i]);
  }
  double sema = 0;
  for(int i=0; i < n; i++) {
    sema += max(x[i] - A);
  }
  return max(sema) + A;
}
```

```julia
function logsumexp(x::Array{Float64,1})
  A = maximum(x)
ema = exp.(x .- A)
sema = sum(ema)
  return log(sema) + A
end
```

Taylor Expand Log

- LLVM Optimization Passe
- Constant (wrt inputs) detection
-

10000000 iterations

	Enzyme	Adept
Normal	3.74	3.72
Forward	3.74	4.56
Forward +Reverse	3.90	4.65

LogSumExp

- LLVM Optimization Passe
- Constant (wrt inputs) detection
-

10000000 elements

	Enzyme	Adept
Normal	0.364	0.364
Forward	0.364	2.994
Forward +Reverse	0.605	3.836

Gradient Descent : Find Matrix by Gradient Descent

- LLVM Optimization Passe
- Constant (wrt inputs) detection
-

	Enzyme	Adept
Forward	4.731	25.606
Gradient Descent	22.672	133.354

Training Simple Neural Network

- LLVM Optimization Passe
- Constant (wrt inputs) det
-

Enzyme	Adept	Handwritte n
73.718	338.097	72.178

Picked first C MNIST Code on Github:

https://github.com/AndrewCarterUK/mnist-neural-network-plain-c

- 1-layer fully connected layer => softmax => cross-entropy loss
- Batch size 100
- 1000 iterations
- Learning rate 0.5

Case Study: Subcall

- Previous parallel IR's based on CFG's mo

```
double loadsq(double* x) {
  return x[0] * x[0];
}

void f(double* x) {
*x = loadsq(x);
}
```

```
define double @loadsq(double* %x)
```

entry

```
%val = load %x
%mul = %val * %val
ret %mul
```

```
define void @f(double* %x)
```

entry

```
%call = @loadsq(%x)
store %x = %call
ret
```

```
void diffe_f(double* x,
                 double* xp) {
  __enzyme_autodiff(f, x, xp);
}
```

Case S

```
double loadsq(double* x) {
  return x[0] * x[0];
}

void f(double* x) {
*x = loadsq(x);
}
```

```
define {double,double} @augment_loadsq(double* %x)
```

```
entry
    %val = load %x
    %mul = %val * %val
    ret {/*return val*/%mul,
         /*cache*/    %val}
```

```
define void @diffe_loadsq(double* %x, double* %x', double %diffe, double %cache)
```

```
entry
    %val = %cache // cannot reload as x changed
    %mul = %val * %val
    %mul' = %diffe
    %val' = 2 * %val * %mul'
    store %x' += %val'
```

Case Study

```
define {double,double} @augment_loadsq(double* %x)
```

entry

```
%val = load %x
%mul = %val * %val
ret {/*return val*/%mul,
     /*cache*/    %val}
```

```c
double loadsq(double* x) {
  return x[0] * x[0];
}

void f(double* x) {
*x = loadsq(x);
}
```

```
define void @diffe_loadsq(double* %x, double* %x', double %diffe, double %cache)
```

entry

```
%val = %cache // cannot reload as x changed
%mul = %val * %val
%mul' = %diffe
%val' = 2 * %val * %mul'
store %x' += %val'
```

```
define void @diffe_f(double* %x)
```

entry

```
{%call, %cache} = @augment_loadsq(%x)

store %x = %call

%call' = load %x'
store %x' = 0

@augment_loadsq(%x, %x', %call', %cache)
ret
```

Case Stud

```
define {double,double} @augment_loadsq(double* %x)

entry
  %val = load %x
  %mul = %val * %val
  ret {/*return val*/%mul,
       /*cache*/    %val}
```

```
double loadsq(double* x) {
  return x[0] * x[0];
}

void f(double* x) {
*x = loadsq(x);
}
```

```
define void @diffe_loadsq(double* %x’, double %diffe, double %cache)

entry
  store %x’ += 2 * %cache * %diffe
```

```
define void @diffe_f(double* %x)

entry
  {%call, %cache} = @augment_loadsq(%x)

  store %x = %call

  %call’ = load %x’
  store %x’ = 0

  @augment_loadsq(%x’, %call’, %cache)
  ret
```

Type Analysis: TODO REDO THIS

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
callee:
```

```
void callee(int* ptr) {
ptr:       {}
ptr2:      {}
loadtype:  {}
ptr3:      {}
cptr2:     {}
notype:    {}
cptr3:     {}
```

```
ptr2 =
indirect
```

```
ptr3 =
indirect
```

Load + Store Propagation

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
callee:
```

```
void callee(int* ptr) {
ptr:       {}
ptr2:      {[]:Pointer}
loadtype:  {}
ptr3:      {}
cptr2:     {[]:Pointer}
notype:    {}
cptr3:     {[]:Pointer}
```

```
ptr2 =
indirect
```

```
ptr3 =
indirect
```

TBAA Propagation

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

callee:

```
void callee(int* ptr) {
ptr:      {}
ptr2:     {[]:Pointer, [0]:Double}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer}
notype:   {}
cptr3:    {[]:Pointer, [0]:Int}
```

ptr2 = indirect

ptr3 = indirect

cptr3 => ptr

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
callee:
```

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer}
notype:   {}
cptr3:    {[]:Pointer, [0]:Int}
```

```
ptr2 =
indirect
```

```
ptr3 =
indirect
```

ptr => cptr2

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
callee:
```

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [8]:Int}
notype:   {}
cptr3:    {[]:Pointer, [0]:Int}
```

```
ptr2 =
indirect
```

```
ptr3 =
indirect
```

ptr2 Call IPO

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

callee:

```
void callee(int* ptr) {
ptr:       {[]:Pointer, [24]:Int}
ptr2:      {[]:Pointer, [0]:Double}
loadtype:  {[]:Double}
ptr3:      {}
cptr2:     {[]:Pointer, [8]:Int}
notype:    {}
cptr3:     {[]:Pointer, [0]:Int}
```

ptr2 = indirect

```
int* indirect(int* x, int idx) {
x:       {[]:Pointer, [24]:Int}
idx:     {[]:Int@2}
&x[idx]  {}
return   {[]:Pointer, [0]:Double}
```

ptr2 Call IPO - ret

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

callee:

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [8]:Int}
notype:   {}
cptr3:    {[]:Pointer, [0]:Int}
```

ptr2 = indirect

```
int* indirect(int* x, int idx) {
x:      {[]:Pointer, [24]:Int}
idx:    {[]:Int@2}
&x[idx] {[]:Pointer, [0]:Double}
return  {[]:Pointer, [0]:Double}
```

ptr2 Call IPO - x

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

callee:

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [8]:Int}
notype:   {}
cptr3:    {[]:Pointer, [0]:Int}
```

ptr2 = indirect

```
int* indirect(int* x, int idx) {
x:       {[]:Pointer, [16]:Double, [24]:Int}
idx:     {[]:Int@2}
&x[idx] {[]:Pointer, [0]:Double, [8]:Int}
return  {[]:Pointer, [0]:Double}
```

ptr2 Call IPO - x

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
callee:
```

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [8]:Int}
notype:   {}
cptr3:    {[]:Pointer, [0]:Int}
```

```
ptr2 = indirect
```

```
int* indirect(int* x, int idx) {
x:       {[]:Pointer, [16]:Double, [24]:Int}
idx:     {[]:Int@2}
&x[idx] {[]:Pointer, [0]:Double, [8]:Int}
return  {[]:Pointer, [0]:Double, [8]:Int}
```

ptr2 Call IPO

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

callee:

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [16]:Double, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double, [8]:Int}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [8]:Int}
notype:   {}
cptr3:    {[]:Pointer, [0]:Int}
```

ptr2 = indirect

```
int* indirect(int* x, int idx) {
x:       {[]:Pointer, [16]:Double, [24]:Int}
idx:     {[]:Int@2}
&x[idx] {[]:Pointer, [0]:Double, [8]:Int}
return  {[]:Pointer, [0]:Double, [8]:Int}
```

ptr2 Call IPO

- New interprocedural analysis that detects th callee: ype of data
- Perform s agat
- Each valu ype

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [16]:Double, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double, [8]:Int}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [8]:Int}
notype:   {}
cptr3:    {[]:Pointer, [0]:Int}
```

ptr2 = indirect

```
int* indirect(int* x, int idx) {
x:       {[]:Pointer, [16]:Double, [24]:Int}
idx:     {[]:Int@2}
&x[idx]  {[]:Pointer, [0]:Double, [8]:Int}
return   {[]:Pointer, [0]:Double, [8]:Int}
```

ptr => cptr2

- New interprocedural analysis that detects th

```
callee:
```

ype of data

- Perform s ... agat
- Each valu ... pe

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [16]:Double, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double, [8]:Int}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [0]:Double, [8]:Int}
notype:   {}
cptr3:    {[]:Pointer, [0]:Int}
```

cptr2 => notype

- New interprocedural analysis that detects th ype of data
- Perform s agat
- Each valu pe

```
callee:
```

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [16]:Double, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double, [8]:Int}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [0]:Double, [8]:Int}
notype:   {[]:Double}
cptr3:    {[]:Pointer, [0]:Int}
```

ptr3 Call IPO

- New interprocedural analysis that detects th
- Perform s
- Each valu

```
callee:
```

ype of data

agat

ype

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [16]:Double, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double, [8]:Int}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [0]:Double, [8]:Int}
notype:   {[]:Double}
cptr3:    {[]:Pointer, [0]:Int}
```

```
ptr3 = indirect
```

```
int* indirect(int* x, int idx) {
x:      {[]:Pointer, [16]:Double, [24]:Int}
idx:    {[]:Int@3}
&x[idx] {}
return  {}
```

ptr3 Call IPO - x

- New interprocedural analysis that detects th
- Perform s
- Each valu

callee:

ype of data

agat

ype

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [16]:Double, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double, [8]:Int}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [0]:Double, [8]:Int}
notype:   {[]:Double}
cptr3:    {[]:Pointer, [0]:Int}
```

ptr3 = indirect

```
int* indirect(int* x, int idx) {
x:      {[]:Pointer, [16]:Double, [24]:Int}
idx:    {[]:Int@3}
&x[idx] {[]:Pointer, [0]:Int}
return  {}
```

ptr3 Call IPO - return

- New interprocedural analysis that detects th
- Perform s
- Each valu

callee:

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

ype of data

agat

pe

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [16]:Double, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double, [8]:Int}
loadtype: {[]:Double}
ptr3:     {}
cptr2:    {[]:Pointer, [0]:Double, [8]:Int}
notype:   {[]:Double}
cptr3:    {[]:Pointer, [0]:Int}
```

ptr3 = indirect

```
int* indirect(int* x, int idx) {
x:       {[]:Pointer, [16]:Double, [24]:Int}
idx:     {[]:Int@3}
&x[idx] {[]:Pointer, [0]:Int}
return  {[]:Pointer, [0]:Int}
```

ptr3 Call IPO

- New interprocedural analysis that detects th … ype of data
- Perform s … agat
- Each valu … pe

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

callee:

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [16]:Double, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double, [8]:Int}
loadtype: {[]:Double}
ptr3:     {[]:Pointer, [0]:Int}
cptr2:    {[]:Pointer, [0]:Double, [8]:Int}
notype:   {[]:Double}
cptr3:    {[]:Pointer, [0]:Int}
```

ptr3 = indirect

```
int* indirect(int* x, int idx) {
x:       {[]:Pointer, [16]:Double, [24]:Int}
idx:     {[]:Int@3}
&x[idx] {[]:Pointer, [0]:Int}
return  {[]:Pointer, [0]:Int}
```

Done!

- New interprocedural analysis that detects th callee: ype of data
- Perform s agat
- Each valu ype

```
int* indirect(int* x, int idx) {
  return &x[idx];
}

void callee(int* ptr) {
  int* ptr2 = indirect(ptr, 2);
  double loadtype = *(double*)ptr2;
  int* ptr3 = indirect(ptr, 3);
  int* cptr2 = &ptr[2];
  int notype = *ctr2;
  int* cptr3 = &ptr[3];
  *((int64_t*)cptr3) = 100;
}
```

```
void callee(int* ptr) {
ptr:      {[]:Pointer, [16]:Double, [24]:Int}
ptr2:     {[]:Pointer, [0]:Double, [8]:Int}
loadtype: {[]:Double}
ptr3:     {[]:Pointer, [0]:Int}
cptr2:    {[]:Pointer, [0]:Double, [8]:Int}
notype:   {[]:Double}
cptr3:    {[]:Pointer, [0]:Int}
```

LLVM IR

LLVM represents each function as a **control-flow graph (CFG)** of **BasicBlocks**, containing lists of **Instructions**.

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  x = fib(n - 1);
  y = fib(n - 2);
  return x + y;
}
```