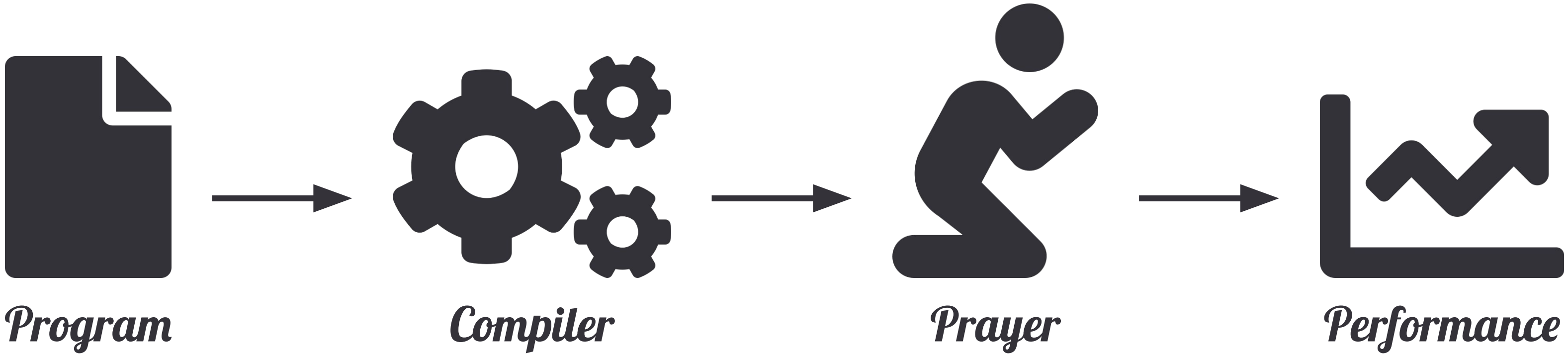
ELEVATE

A Language for Describing Optimization Strategies

Bastian Hagedorn[1] | Johannes Lenfers[1] | Thomas Koehler[2] | Sergei Gorlatch[1] | Michel Steuwer[2]
[1]University of Münster | [2]University of Glasgow

MOTIVATION

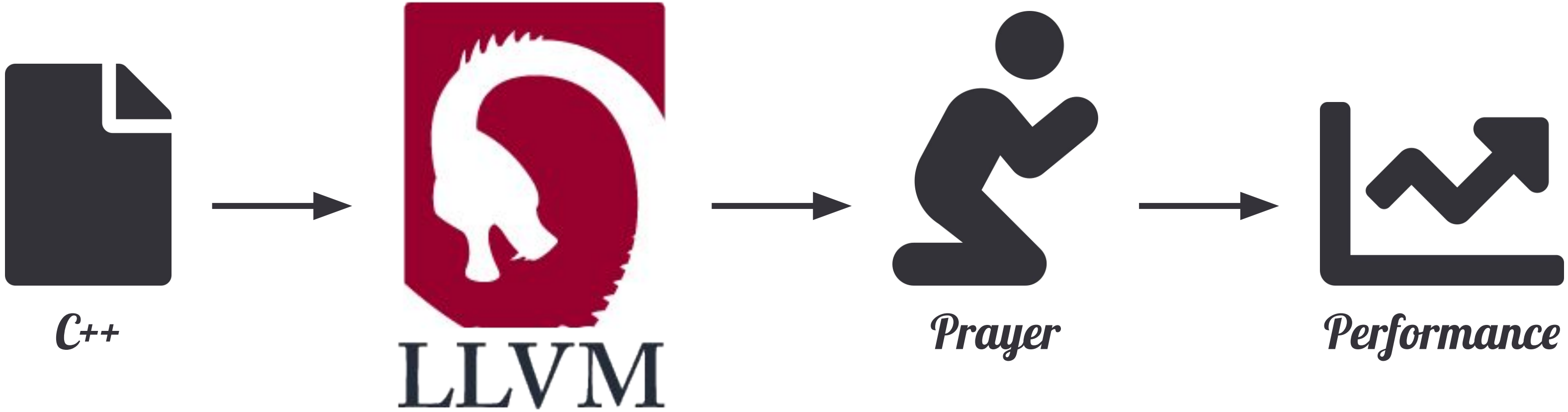
The Landscape of Optimizing Compilers



MOTIVATION

The Landscape of Optimizing Compilers

a General Purpose Compilers



MOTIVATION

The Landscape of Optimizing Compilers

a *General Purpose Compilers*

C++

LLVM

Code Generation Options

-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -Og, -O, -O4

Specify which optimization level to use:

- -O0 Means “no optimization”: this level compiles the fastest and generates the most debuggable code.
- -O1 Somewhere between -O0 and -O2.
- -O2 Moderate level of optimization which enables most optimizations.
- -O3 Like -O2, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).
- -Ofast Enables all the optimizations from -O3 along with other aggressive optimizations that may violate strict compliance with language standards.
- -Os Like -O2 with extra optimizations to reduce code size.
- -Oz Like -Os (and thus -O2), but reduces code size further.
- -Og Like -O1. In future versions, this option might disable different optimizations in order to improve debuggability.
- -O Equivalent to -O2.
- -O4 and higher

  Currently equivalent to -O3

"... in an <u>attempt</u> to make the program run faster"

MOTIVATION

The Landscape of Optimizing Compilers

Code Generation Options

-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -Og, -O, -O4

Specify which optimization level to use:

-O0 Means “no optimization”: this level compiles the fastest and generates the most debuggable code.

-O1 Somewhere between -O0 and -O2.

-O2 Moderate level of optimization which enables most optimizations.

-O3 Like -O2, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

-Ofast Enables all the optimizations from -O3 along with other aggressive optimizations that may violate strict compliance with language standards.

-Os Like -O2 with extra optimizations to reduce code size.

-Oz Like -Os (and thus -O2), but reduces code size further.

-Og Like -O1. In future versions, this option might disable different optimizations in order to improve debuggability.

-O Equivalent to -O2.

-O4 and higher

Currently equivalent to -O3

"Somewhere between -O0 *and* -O2*"*

MOTIVATION

The Landscape of Optimizing Compilers

a General Purpose Compilers

C++

LLVM

```
-targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-info -forceattrs -inferattrs -callsite-splitting -i
psccp -called-value-propagation -globalopt -domtree -mem2reg -deadargelim -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-fre
q -opt-remark-emitter -instcombine -simplifycfg -basiccg -globals-aa -prune-eh -inline -functionattrs -argpromotion -domtree -sroa -basi
caa -aa -memoryssa -early-cse-memssa -speculative-execution -basicaa -aa -lazy-value-info -jump-threading -correlated-propagation -simpl
ifycfg -domtree -aggressive-instcombine -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -libcall
s-shrinkwrap -loops -branch-prob -block-freq -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -pgo-memop-opt -basicaa -aa -loops -
lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa-veri
fication -lcssa -basicaa -aa -scalar-evolution -loop-rotate -licm -loop-unswitch -simplifycfg -domtree -basicaa -aa -loops -lazy-branch-
prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution -indvars -loop-idiom
-loop-deletion -loop-unroll -mldst-motion -phi-values -basicaa -aa -memdep -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -gvn -
phi-values -basicaa -aa -memdep -memcpyopt -sccp -demanded-bits -bdce -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark
-emitter -instcombine -lazy-value-info -jump-threading -correlated-propagation -basicaa -aa -phi-values -memdep -dse -loops -loop-simpli
fy -lcssa-verification -lcssa -basicaa -aa -scalar-evolution -licm -postdomtree -adce -simplifycfg -domtree -basicaa -aa -loops -lazy-br
anch-prob -lazy-block-freq -opt-remark-emitter -instcombine -barrier -elim-avail-extern -basiccg -rpo-functionattrs -globalopt -globaldc
e -basiccg -globals-aa -float2int -domtree -loops -loop-simplify -lcssa-verification -lcssa -basicaa -aa -scalar-evolution -loop-rotate
-loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-distribute -branch-prob -block-freq -scalar-evolution -basic
aa -aa -loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evolu
tion -aa -loop-accesses -loop-load-elim -basicaa -aa -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -d
omtree -loops -scalar-evolution -basicaa -aa -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -slp-vectorizer -opt-
remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution -loop-unroll -lazy-branch-prob -lazy-block-freq
-opt-remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution -licm -alignment-from-assumptions -strip-de
ad-prototypes -globaldce -constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -lcssa-verification -lcssa -basicaa -aa -sc
alar-evolution -branch-prob -block-freq -loop-sink -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instsimplify -div-rem-pairs -
simplifycfg -verify
```

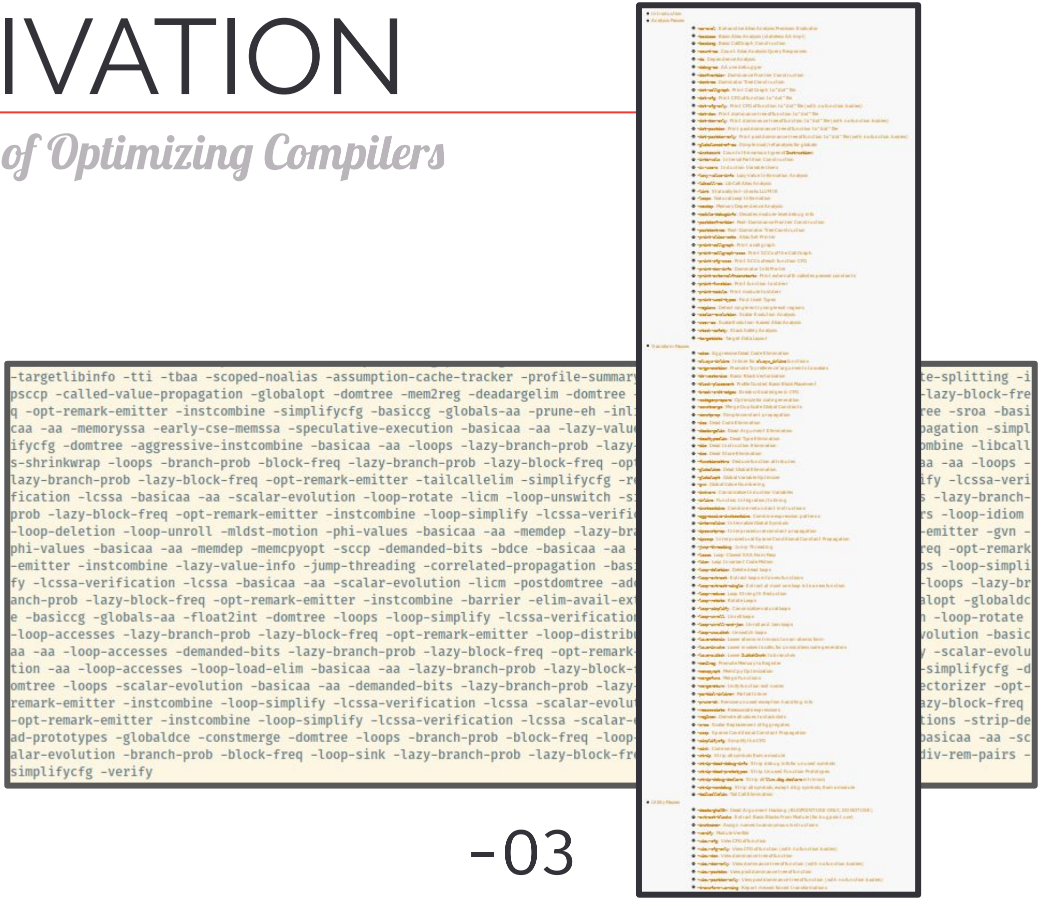
-O3

MOTIVATION

The Landscape of Optimizing Compilers

a General Purpose Compilers

C++



-O3

Compiler Passes

MOTIVATION

The Landscape of Optimizing Compilers

a General Purpose Compilers

LLVM

```
-targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summar
psccp -called-value-propagation -globalopt -domtree -mem2reg -deadargelim -domtree
q -opt-remark-emitter -instcombine -simplifycfg -basiccg -globals-aa -prune-eh -inl
caa -aa -memoryssa -early-cse-memssa -speculative-execution -basicaa -aa -lazy-valu
ifycfg -domtree -aggressive-instcombine -basicaa -aa -loops -lazy-branch-prob -lazy
s-shrinkwrap -loops -branch-prob -block-freq -lazy-branch-prob -lazy-block-freq -op
lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim -simplifycfg -r
fication -lcssa -basicaa -aa -scalar-evolution -loop-rotate -licm -loop-unswitch -s
prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa-verifi
-loop-deletion -loop-unroll -mldst-motion -phi-values -basicaa -aa -memdep -lazy-br
phi-values -basicaa -aa -memdep -memcpyopt -sccp -demanded-bits -bdce -basicaa -aa
-emitter -instcombine -lazy-value-info -jump-threading -correlated-propagation -bas
fy -lcssa-verification -lcssa -basicaa -aa -scalar-evolution -licm -postdomtree -ad
anch-prob -lazy-block-freq -opt-remark-emitter -instcombine -barrier -elim-avail-ex
e -basiccg -globals-aa -float2int -domtree -loops -loop-simplify -lcssa-verificatio
-loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-distrib
aa -aa -loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark
tion -aa -loop-accesses -loop-load-elim -basicaa -aa -lazy-branch-prob -lazy-block-
omtree -loops -scalar-evolution -basicaa -aa -demanded-bits -lazy-branch-prob -lazy
remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolu
-opt-remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar-
ad-prototypes -globaldce -constmerge -domtree -loops -branch-prob -block-freq -loop
alar-evolution -branch-prob -block-freq -loop-sink -lazy-branch-prob -lazy-block-fr
simplifycfg -verify
```

-O3

Compiler Passes

MOTIVATION

The Landscape of Optimizing Compilers

-O3

Compiler Passes

MOTIVATION

The Landscape of Optimizing Compilers

b Schedule-Based Compilers

Experts define how to optimize the program (*algorithm*) in a separate *schedule*

MOTIVATION

The Landscape of Optimizing Compilers

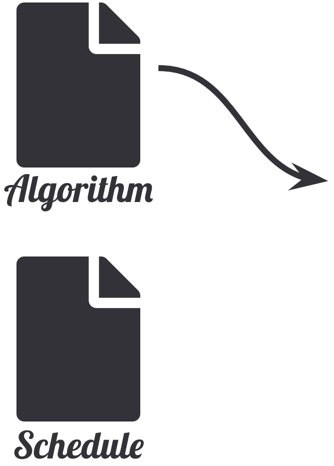
b *Schedule-Based Compilers*

Experts define how to optimize the program (*algorithm*) in a separate *schedule*

MOTIVATION

The Landscape of Optimizing Compilers

b Schedule-Based Compilers




```
Func prod(“prod");
RDom r(0, size);
prod(x, y) +=
A(x, r) * B(r, y);
out(x, y) = prod(x, y);
```

Matrix Multiplication

MOTIVATION

The Landscape of Optimizing Compilers

b Schedule-Based Compilers

Algorithm

```
Func prod(“prod");
RDom r(0, size);
prod(x, y) +=
A(x, r) * B(r, y);
out(x, y) = prod(x, y);
```

Matrix Multiplication

Schedule

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
     .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
     .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
     .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
     .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
     .unroll(xo).unroll(Ay);
```

Schedule for Nvidia GPUs

MOTIVATION

The Landscape of Optimizing Compilers

```
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
```

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
```

```
Func prod("prod");
RDom r(0, size);
prod(x, y) +=
A(x, r) * B(r, y);
out(x, y) = prod(x, y);
```

Matrix Multiplication

MOTIVATION

The Landscape of Optimizing Compilers

```
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
```

1 Schedules are ***much harder to write*** than algorithms

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi, yi, xio, xii, yii, xo, yo, x_pair, xiio, ty; RVar rxo, rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
```

```
Func prod("prod");
RDom r(0, size);
prod(x, y) +=
A(x, r) * B(r, y);
out(x, y) = prod(x, y);
```

Matrix Multiplication

MOTIVATION

The Landscape of Optimizing Compilers

```
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
```

1. Schedules are ***much harder to write*** than algorithms
2. Schedules and algorithms are ***not really separated***

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
```

```
Func prod("prod");
RDom r(0, size);
prod(x, y) +=
A(x, r) * B(r, y);
out(x, y) = prod(x, y);
```

Matrix Multiplication

MOTIVATION

The Landscape of Optimizing Compilers

```
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
```

1. Schedules are ***much harder to write*** than algorithms
2. Schedules and algorithms are ***not really separated***
3. The schedule "language" is a fixed API and ***not extensible***

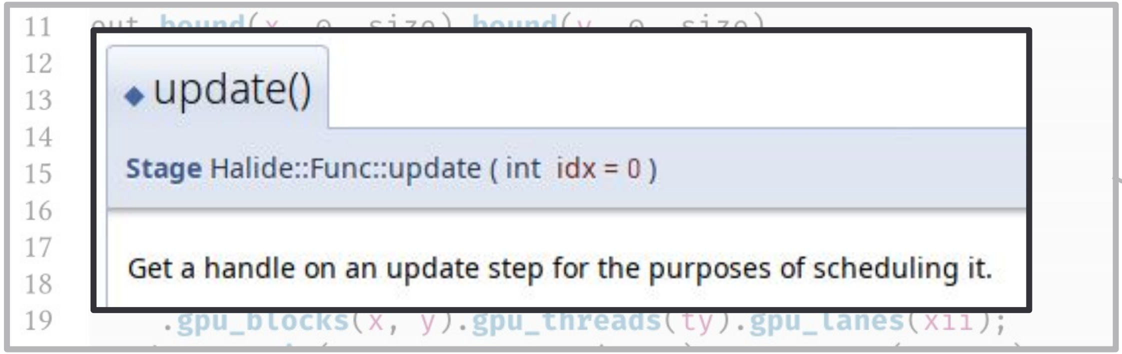
```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi, yi, xio, xii, yii, xo, yo, x_pair, xiio, ty; RVar rxo, rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
```

```
Func prod("prod");
RDom r(0, size);
prod(x, y) +=
A(x, r) * B(r, y);
out(x, y) = prod(x, y);
```

Matrix Multiplication

MOTIVATION

The Landscape of Optimizing Compilers



1. Schedules are ***much harder to write*** than algorithms
2. Schedules and algorithms are ***not really separated***
3. The schedule "language" is a fixed API and ***not extensible***
4. Schedule primitives are not intuitive and have ***unclear semantics***

```
// functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
```

```
Func prod("prod");
RDom r(0, size);
prod(x, y) +=
A(x, r) * B(r, y);
out(x, y) = prod(x, y);
```

Matrix Multiplication

MOTIVATION

The Landscape of Optimizing Compilers

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

Separable Convolution: Sobel Filter

```
Var x, y; Func out; Func in = BC::repeat_edge(input);
out(x, y) = (
  1.f * in(x-1,y-1) + 2.f * in(x,y-1) + 1.f * in(x+1,y-1) +
  2.f * in(x-1,y)   + 4.f * in(x,y)   + 2.f * in(x+1,y)   +
  1.f * in(x-1,y+1) + 2.f * in(x,y+1) + 1.f * in(x+1,y+1)
) * (1.f/16.f);
```

2D Convolution Algorithm

1. Schedules are ***much harder to write*** than algorithms
2. Schedules and algorithms are ***not really separated***
3. The schedule "language" is a fixed API and ***not extensible***
4. Schedule primitives are not intuitive and have ***unclear semantics***
5. Schedules are ***not expressive enough***

```
Var x,y; Func b_x,b_y,out; Func in=BC::repeat_edge(input);
b_y(x, y) =  in(x, y-1) + 2.f * in(x, y)  +  in(x, y+1);
b_x(x, y) = b_y(x-1, y) + 2.f * b_y(x, y) + b_y(x+1, y);
out(x, y) = b_x(x, y) * (1.f/16.f);
```

Separated Algorithm

ELEVATE

Desirable Properties of a Strategy Language

Wouldn't it be great…

if we could ***look behind the curtains*** of optimizing compilers and actually understand how optimizations are applied

ELEVATE

Desirable Properties of a Strategy Language

Wouldn't it be great...

if we could ***look behind the curtains*** of optimizing compilers and actually understand how optimizations are applied

to have a ***flexible*** way of specifying optimizations for your compiler and your programming language

ELEVATE

Desirable Properties of a Strategy Language

Wouldn't it be great...

- if we could ***look behind the curtains*** of optimizing compilers and actually understand how optimizations are applied
- to have a ***flexible*** way of specifying optimizations for your compiler and your programming language
- to build custom optimizations in an ***extensible*** language while avoiding to rely on fixed scheduling APIs

ELEVATE

Desirable Properties of a Strategy Language

Wouldn't it be great...

- if we could ***look behind the curtains*** of optimizing compilers and actually understand how optimizations are applied
- to have a ***flexible*** way of specifying optimizations for your compiler and your programming language
- to build custom optimizations in an ***extensible*** language while avoiding to rely on fixed scheduling APIs
- to have a ***scalable*** approach that competes with state-of-the-art solutions

ELEVATE

Desirable Properties of a Strategy Language

Wouldn't it be great…

if we could ***look behind the curtains*** of optimizing compilers and actually understand how optimizations are applied

A strategy language should be built with the same standards as a language describing computation

to build custom optimizations in an ***extensible*** language while avoiding to rely on fixed scheduling APIs

to have a ***scalable*** approach that competes with state-of-the-art solutions

STRATEGIES

Optimizing Programs like it's ~~1998~~ 2020

Visser et. al.: Building program optimizers with rewriting strategies (ICFP 1998)

ELEVATE

What actually is a "Strategy"?

A ***Strategy*** encodes a program transformation:

```
type Strategy[P] = P => RewriteResult[P]
```

ELEVATE

What actually is a "Strategy"?

A ***Strategy*** encodes a program transformation:

```
type Strategy[P] = P => RewriteResult[P]
```

A ***RewriteResult*** encodes its success or failure:

```
RewriteResult[P] = Success[P](p: P)
                 | Failure[P](s: Strategy[P])
```

ELEVATE

What actually is a "Strategy"?

A ***Strategy*** encodes a program transformation:

```
type Strategy[P] = P => RewriteResult[P]
```

A ***RewriteResult*** encodes its success or failure:

```
RewriteResult[P] = Success[P](p: P)
                 | Failure[P](s: Strategy[P])
```

Two naive generic strategies:

```
def   id[P]: Strategy[P] = (p:P) => Success(p)
def fail[P]: Strategy[P] = (p:P) => Failure(fail)
```

EXAMPLE

A Language-Specific Strategy

Let's encode an arithmetic simplification as a strategy: $x + 0 \rightarrow x$

EXAMPLE

A Language-Specific Strategy

Let's encode an arithmetic simplification as a strategy: $x + 0 \rightarrow x$

```
val p: ArithExpr = x + 0 // AST: Plus(Var("x"), 0)
```

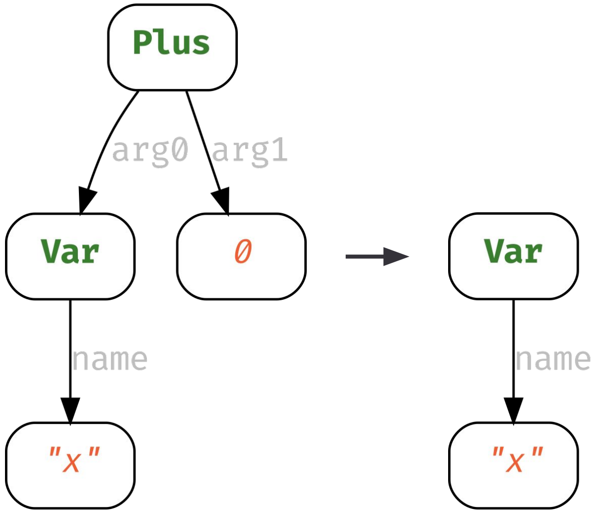
Our toy-example target DSL

EXAMPLE

A Language-Specific Strategy

Let's encode an arithmetic simplification as a strategy: $x + 0 \rightarrow x$

```
val p: ArithExpr = x + 0 // AST: Plus(Var("x"), 0)
```



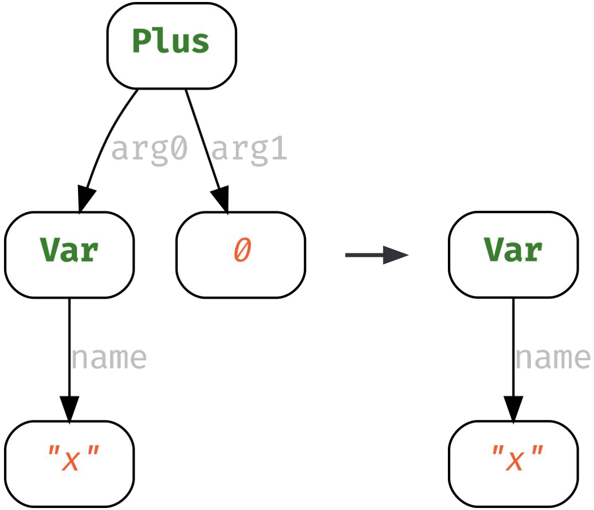
AST Transformation: $x + 0 \rightarrow x$

EXAMPLE

A Language-Specific Strategy

Let's encode an arithmetic simplification as a strategy: $x + 0 \rightarrow x$

```
val p: ArithExpr = x + 0 // AST: Plus(Var("x"), 0)
```



AST Transformation: $x + 0 \rightarrow x$

```
def plus0: Strategy[ArithExpr] =
```

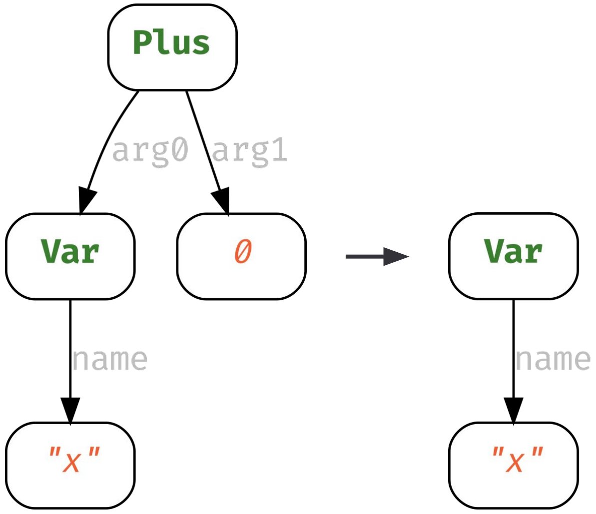
Simplification rule expressed in ELEVATE:

EXAMPLE

A Language-Specific Strategy

Let's encode an arithmetic simplification as a strategy: $x + 0 \rightarrow x$

```
val p: ArithExpr = x + 0 // AST: Plus(Var("x"), 0)
```



AST Transformation: $x + 0 \rightarrow x$

```
def plus0: Strategy[ArithExpr] =
  (p:ArithExpr) => p match {


  }
```

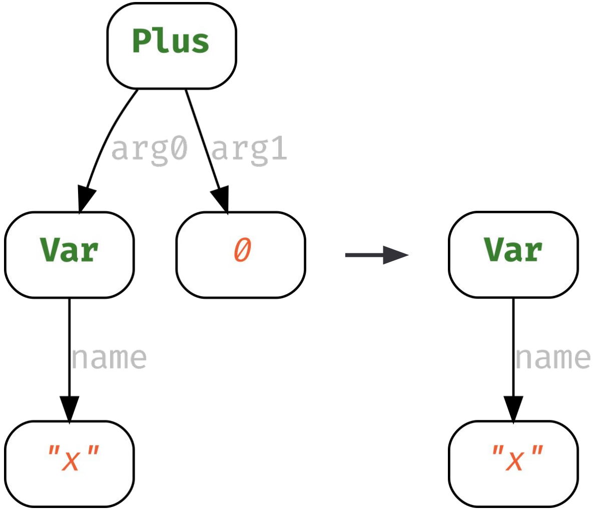
Simplification rule expressed in ELEVATE:

EXAMPLE

A Language-Specific Strategy

Let's encode an arithmetic simplification as a strategy: $x + 0 \rightarrow x$

```
val p: ArithExpr = x + 0 // AST: Plus(Var("x"), 0)
```



AST Transformation: $x + 0 \rightarrow x$

```
def plus0: Strategy[ArithExpr] =
  (p:ArithExpr) => p match {
    case Plus(Var(x),0) => Success( Var(x) )

  }
```

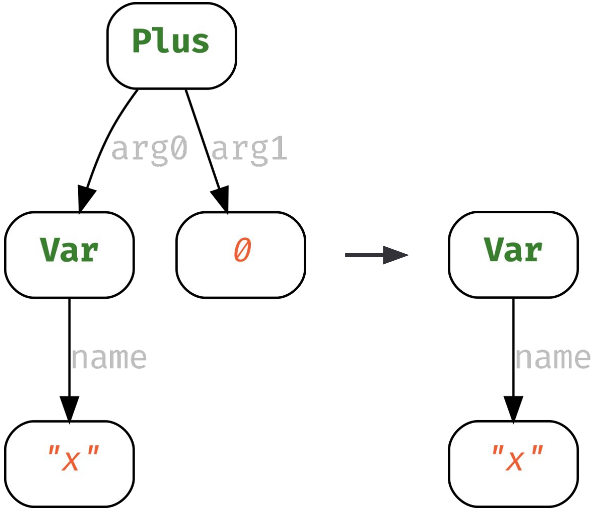
Simplification rule expressed in ELEVATE:

EXAMPLE

A Language-Specific Strategy

Let's encode an arithmetic simplification as a strategy: $x + 0 \to x$

```
val p: ArithExpr = x + 0 // AST: Plus(Var("x"), 0)
```



AST Transformation: $x + 0 \to x$

```
def plus0: Strategy[ArithExpr] =
  (p:ArithExpr) => p match {
    case Plus(Var(x),0) => Success( Var(x) )
    case _              => Failure( plus0 )
  }
```

Simplification rule expressed in ELEVATE:

EXAMPLE

A More Interesting Language-Specific Strategy

Let's encode another language specific strategy: $map(f) \circ map(g) \rightarrow map\ (f \circ g)$

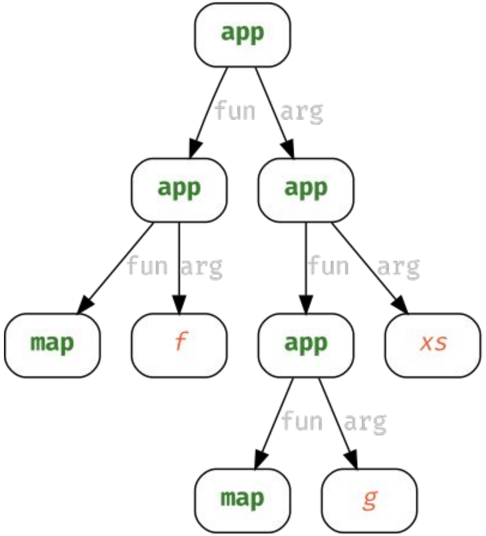
```
val p: Lift = fun(xs => map(f)(map(g)(xs)))
```

EXAMPLE

A More Interesting Language-Specific Strategy

Let's encode another language specific strategy: *map(f) ∘ map(g) → map (f ∘ g)*

```
val p: Lift = fun(xs => map(f)(map(g)(xs)))
```



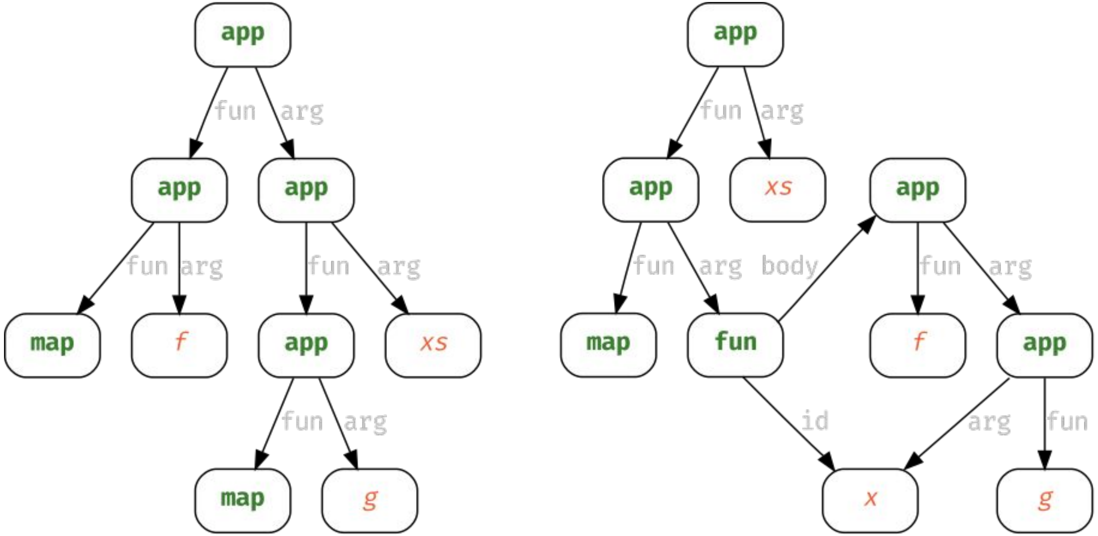
map(f)(map(g)(xs))

EXAMPLE

A More Interesting Language-Specific Strategy

Let's encode another language specific strategy: *map(f) ∘ map(g) → map (f ∘ g)*

```
val p: Lift = fun(xs => map(f)(map(g)(xs)))
```



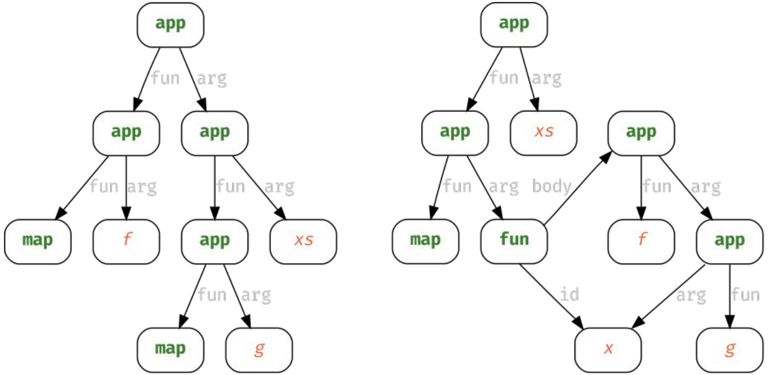
map(f)(map(g)(xs)) ⟶ map(fun(x ⇒ f(g(x))))(xs)

EXAMPLE

A More Interesting Language-Specific Strategy

Let's encode another language specific strategy: $map(f) \circ map(g) \rightarrow map\ (f \circ g)$

```
val p: Lift = fun(xs => map(f)(map(g)(xs)))
```



map(f)(map(g)(xs)) → map(fun(x ⇒ f(g(x))))(xs)

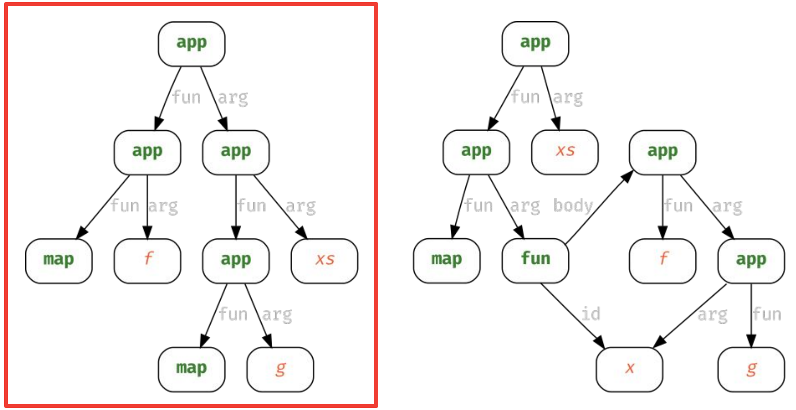
```
def mapFusion: Strategy[Lift] =
  (p:Lift) => p match {


  }
```

EXAMPLE

A More Interesting Language-Specific Strategy

Let's encode another language specific strategy: *map(f) ∘ map(g) → map (f ∘ g)*

```
val p: Lift = fun(xs => map(f)(map(g)(xs)))
```



map(f)(map(g)(xs)) → map(fun(x ⇒ f(g(x))))(xs)

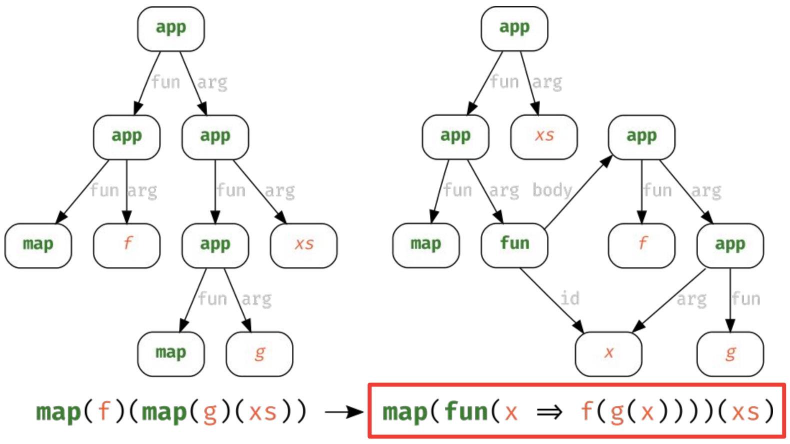
```
def mapFusion: Strategy[Lift] =
  (p:Lift) => p match {
    case app(app(map, f),
             app(app(map, g), xs))


  }
```

EXAMPLE

A More Interesting Language-Specific Strategy

Let's encode another language specific strategy: *map(f) ∘ map(g) → map (f ∘ g)*

```
val p: Lift = fun(xs => map(f)(map(g)(xs)))
```



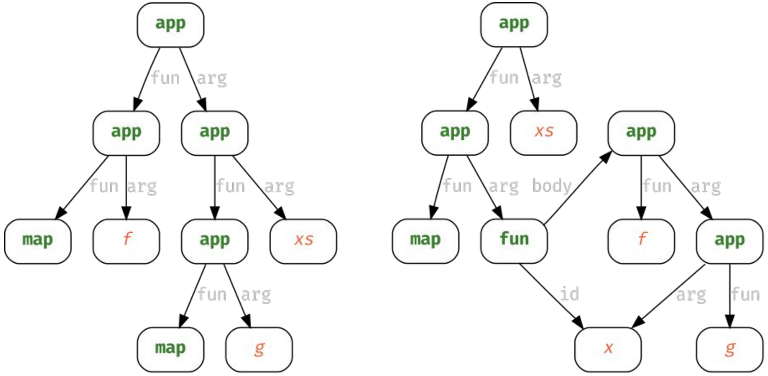

```
def mapFusion: Strategy[Lift] =
  (p:Lift) => p match {
    case app(app(map, f),
             app(app(map, g), xs)) =>
      Success( map(fun(x => f(g(x))))(xs) )

  }
```

EXAMPLE

A More Interesting Language-Specific Strategy

Let's encode another language specific strategy: $map(f) \circ map(g) \rightarrow map\,(f \circ g)$

```
val p: Lift = fun(xs => map(f)(map(g)(xs)))
```



map(f)(map(g)(xs)) ⟶ map(fun(x ⇒ f(g(x))))(xs)

```
def mapFusion: Strategy[Lift] =
  (p:Lift) => p match {
    case app(app(map, f),
             app(app(map, g), xs)) =>
      Success( map(fun(x => f(g(x))))(xs) )
    case _ => Failure( mapFusion )
  }
```

EXAMPLE

Program Transformations as Strategies

Essentially: *myTransformation*: *lhs* ⟶ *rhs*

↓ ELEVATE

```
def myTransformation: Strategy[MyLanguage] =
  (p:MyLanguage) => p match {
    case lhs => Success( rhs )
    case _   => Failure( myTransformation )
  }
```

COMBINATORS

How to Build More Powerful Strategies

The ***seq*** combinator applies two strategies in sequence

```
def seq[P]: Strategy[P] => Strategy[P] => Strategy[P] =
        fs => ss => p => fs(p) >>= ss
```

first strategy

second strategy

input program

apply strategy to program

iff successful apply *ss* to result
otherwise return Failure

COMBINATORS

How to Build More Powerful Strategies

The ***seq*** combinator applies two strategies in sequence

```
def seq[P]: Strategy[P] => Strategy[P] => Strategy[P] =
    fs => ss => p => fs(p) >>= ss
```

The ***lChoice*** combinator applies the second Strategy only if the first one failed

```
def lChoice[P]: Strategy[P] => Strategy[P] => Strategy[P] =
    fs => ss => p => fs(p) <|> ss(p)
```

first strategy

second strategy

input program

apply strategy to program

if lhs successful return that
otherwise apply second strategy

COMBINATORS

How to Build More Powerful Strategies

The ***seq*** combinator applies two strategies in sequence

```
def seq[P]: Strategy[P] => Strategy[P] => Strategy[P] =
  fs => ss => p => fs(p) >>= ss
```

The ***lChoice*** combinator applies the second Strategy only if the first one failed

```
def lChoice[P]: Strategy[P] => Strategy[P] => Strategy[P] =
  fs => ss => p => fs(p) <|> ss(p)
```

first strategy

input program

if lhs successful return that
otherwise apply second strategy

second strategy

apply strategy to program

We write `a ; b` for `seq(a, b)` and `a <+ b` for `lChoice(a, b)`

COMBINATORS

How to Build More Powerful Strategies

```
def seq[P]: Strategy[P] => Strategy[P] => Strategy[P] =
    fs => ss => p => fs(p) >>= ss
```

```
def lChoice[P]: Strategy[P] => Strategy[P] => Strategy[P] =
    fs => ss => p => fs(p) <|> ss(p)
```

The ***try*** combinator tries to apply a strategy and in case of Failure returns the input unchanged

```
def try[P]: Strategy[P] => Strategy[P] =
    s => p => (s <+ id)(p)
```

Observation: ***try*** never fails!

COMBINATORS

How to Build More Powerful Strategies

```
def seq[P]: Strategy[P] => Strategy[P] => Strategy[P] =
    fs => ss => p => fs(p) >>= ss
```

```
def lChoice[P]: Strategy[P] => Strategy[P] => Strategy[P] =
    fs => ss => p => fs(p) <|> ss(p)
```

The ***try*** combinator tries to apply a strategy and in case of Failure returns the input unchanged

```
def try[P]: Strategy[P] => Strategy[P] =
    s => p => (s <+ id)(p)
```

The ***repeat*** combinator applies a strategy until it's no longer applicable

```
def repeat[P]: Strategy[P] => Strategy[P] =
    s => p => try(s ; repeat(s))(p)
```

TRAVERSALS

Describing Precise Locations in the AST

Another simple Lift program: $(\, map(f) \circ map(g) \circ map(h)\,)(xs)$

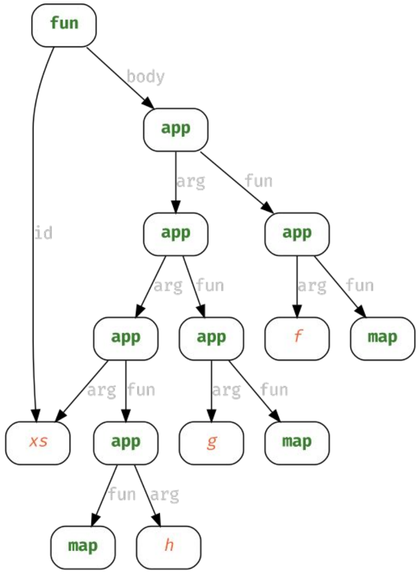
```
val threeMaps: Lift = fun(xs => map(f)(map(g)(map(h)(xs))))
```

TRAVERSALS

Describing Precise Locations in the AST

Another simple Lift program: (*map(f) ∘ map(g) ∘ map(h))(xs)*

```
val threeMaps: Lift = fun(xs => map(f)(map(g)(map(h)(xs))))
```



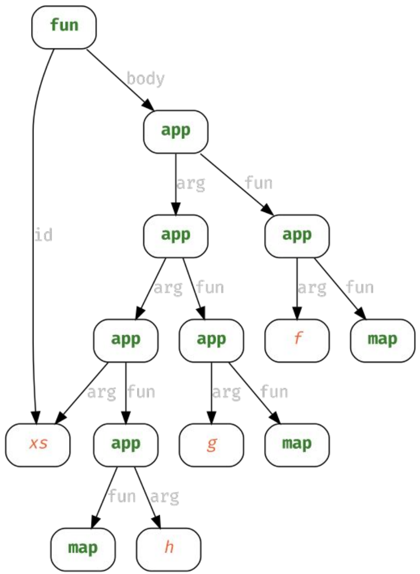

```
fun(xs ⇒ map(f)(map(g)(map(h)(xs))))
```

TRAVERSALS

Describing Precise Locations in the AST

Another simple Lift program: (*map(f)* ∘ *map(g)* ∘ *map(h)*)*(xs)*

```
val threeMaps: Lift = fun(xs => map(f)(map(g)(map(h)(xs))))
```



fun(xs ⇒ map(f)(map(g)(map(h)(xs))))

```
def mapFusion: Strategy[Lift] =
  (p:Lift) => p match {
    case app(app(map, f),
         app(app(map, g), xs)) =>
      Success( map(fun(x => f(g(x))))(xs) )
    case _ => Failure( mapFusion )
  }
```

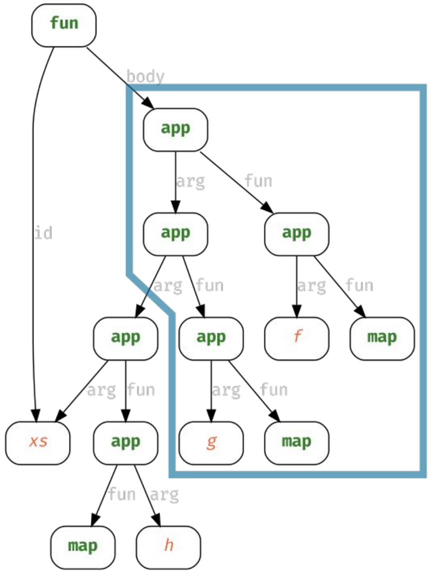
mapFusion: *map(f)* ∘ *map(g)* ⟶ *map (f* ∘ *g)*

TRAVERSALS

Describing Precise Locations in the AST

Another simple Lift program: (*map(f) ∘ map(g) ∘ map(h))(xs)*

```
val threeMaps: Lift = fun(xs => map(f)(map(g)(map(h)(xs))))
```



fun(xs ⇒ map(f)(map(g)(map(h)(xs))))

```
def mapFusion: Strategy[Lift] =
  (p:Lift) => p match {
    case app(app(map, f),
             app(app(map, g), xs)) =>
      Success( map(fun(x => f(g(x))))(xs) )
    case _ => Failure( mapFusion )
  }
```

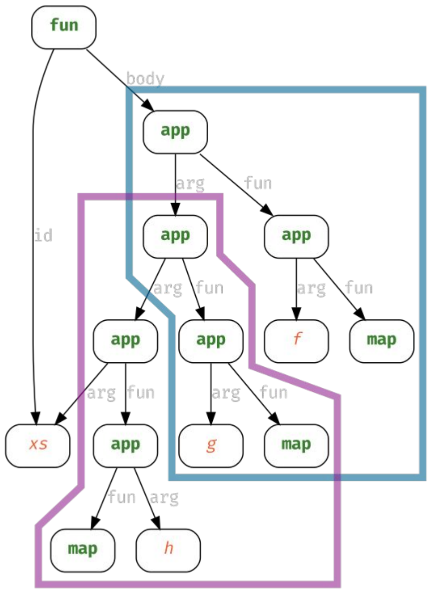
mapFusion: *map(f) ∘ map(g) → map (f ∘ g)*

TRAVERSALS

Describing Precise Locations in the AST

Another simple Lift program: (*map(f)* ∘ *map(g)* ∘ *map(h)*)*(xs)*

```
val threeMaps: Lift = fun(xs => map(f)(map(g)(map(h)(xs))))
```



fun(xs ⇒ map(f)(map(g)(map(h)(xs))))

```
def mapFusion: Strategy[Lift] =
  (p:Lift) => p match {
    case app(app(map, f),
             app(app(map, g), xs)) =>
      Success( map(fun(x => f(g(x))))(xs) )
    case _ => Failure( mapFusion )
  }
```

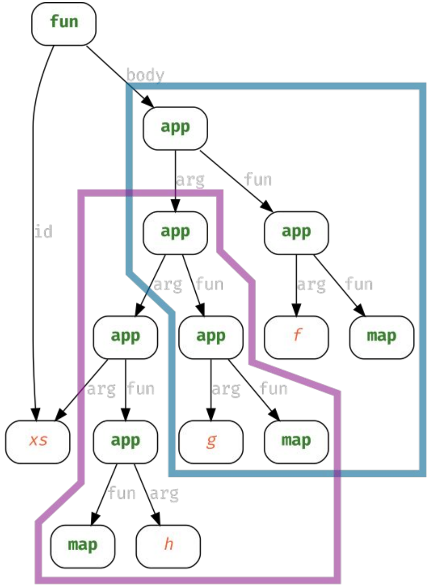
mapFusion: *map(f)* ∘ *map(g)* → *map (f* ∘ *g)*

TRAVERSALS

Describing Precise Locations in the AST

Another simple Lift program: (*map(f)* ∘ *map(g)* ∘ *map(h)*)*(xs)*

```
val threeMaps: Lift = fun(xs => map(f)(map(g)(map(h)(xs))))
```



fun(xs ⇒ map(f)(map(g)(map(h)(xs))))

```
def mapFusion: Strategy[Lift] =
  (p:Lift) => p match {
    case app(app(map, f),
             app(app(map, g), xs)) =>
      Success( map(fun(x => f(g(x))))(xs) )
    case _ => Failure( mapFusion )
  }
```

mapFusion: *map(f)* ∘ *map(g)* → *map (f* ∘ *g)*

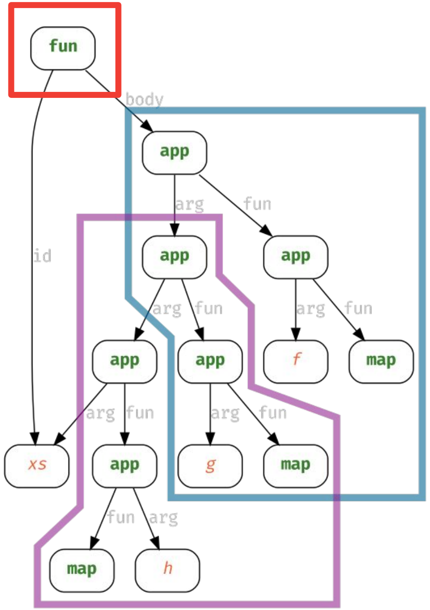
```
mapFusion(threeMaps) == Failure( mapFusion )
```

TRAVERSALS

Describing Precise Locations in the AST

Another simple Lift program: *(map(f) ∘ map(g) ∘ map(h))(xs)*

```
val threeMaps: Lift = fun(xs => map(f)(map(g)(map(h)(xs))))
```



fun(xs ⇒ map(f)(map(g)(map(h)(xs))))

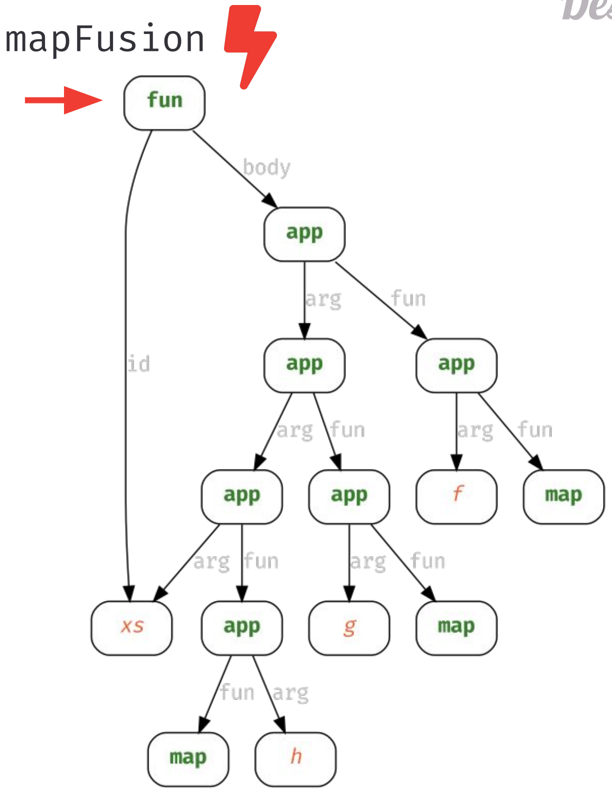
```
def mapFusion: Strategy[Lift] =
  (p:Lift) => p match {
    case app(app(map, f),
         app(app(map, g), xs)) =>
      Success( map(fun(x => f(g(x))))(xs) )
    case _ => Failure( mapFusion )
  }
```

mapFusion: *map(f) ∘ map(g) → map (f ∘ g)*

```
mapFusion(threeMaps) == Failure( mapFusion )
```

TRAVERSALS

Describing Precise Locations in the AST



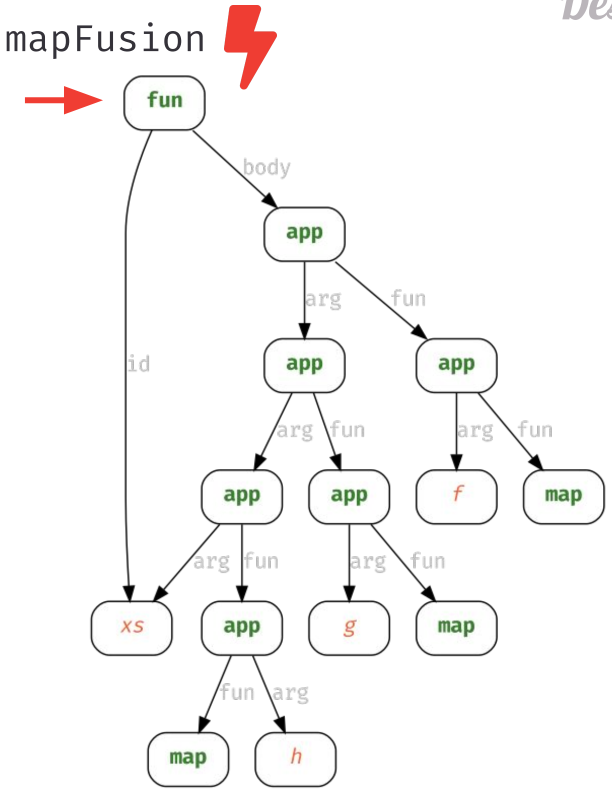
`fun(xs ⇒ map(f)(map(g)(map(h)(xs))))`

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

TRAVERSALS

Describing Precise Locations in the AST



fun(xs ⇒ **map**(f)(**map**(g)(**map**(h)(xs))))

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

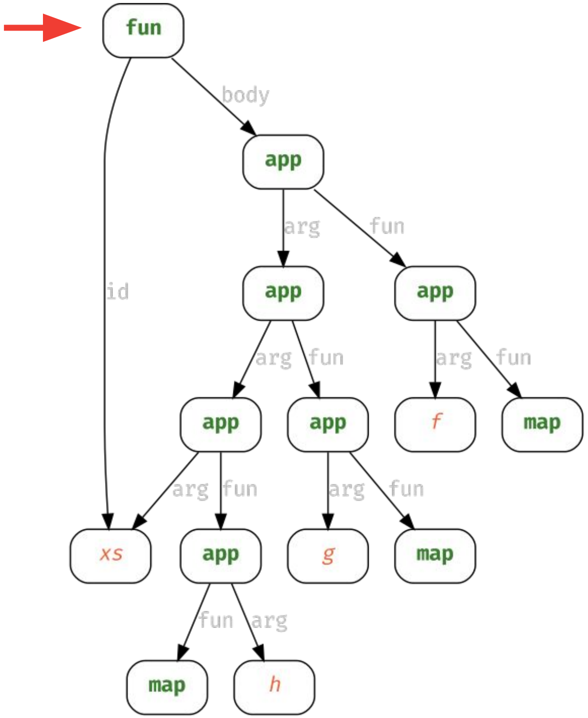
...but we can use ***generic one-level traversals*** to push strategy applications down the AST

```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

TRAVERSALS

Describing Precise Locations in the AST

`all(mapFusion)`



`fun(xs ⇒ map(f)(map(g)(map(h)(xs))))`

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals*** to push strategy applications down the AST

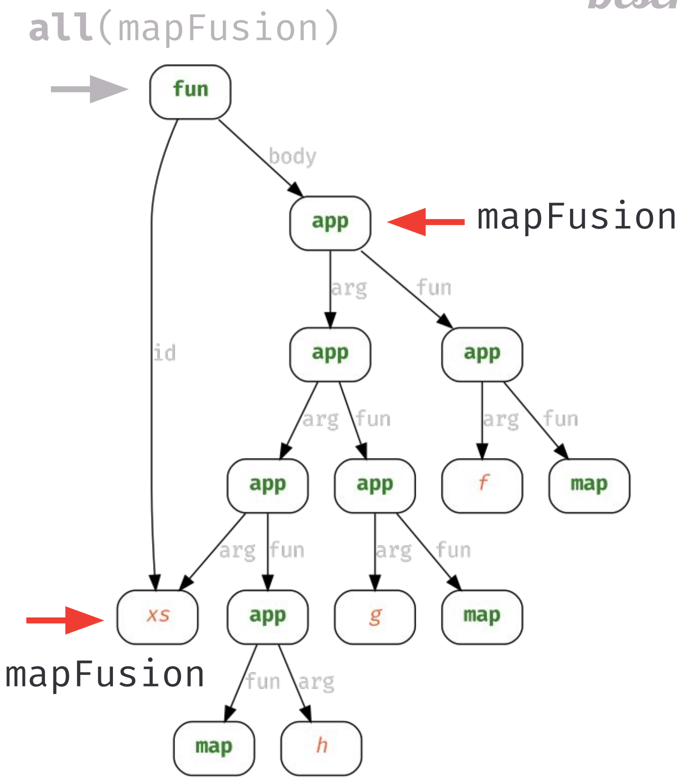
```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

Let's try...

```
all(mapFusion)(threeMaps)
```

TRAVERSALS

Describing Precise Locations in the AST



`fun(xs ⇒ map(f)(map(g)(map(h)(xs))))`

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals*** to push strategy applications down the AST

```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

Let's try...

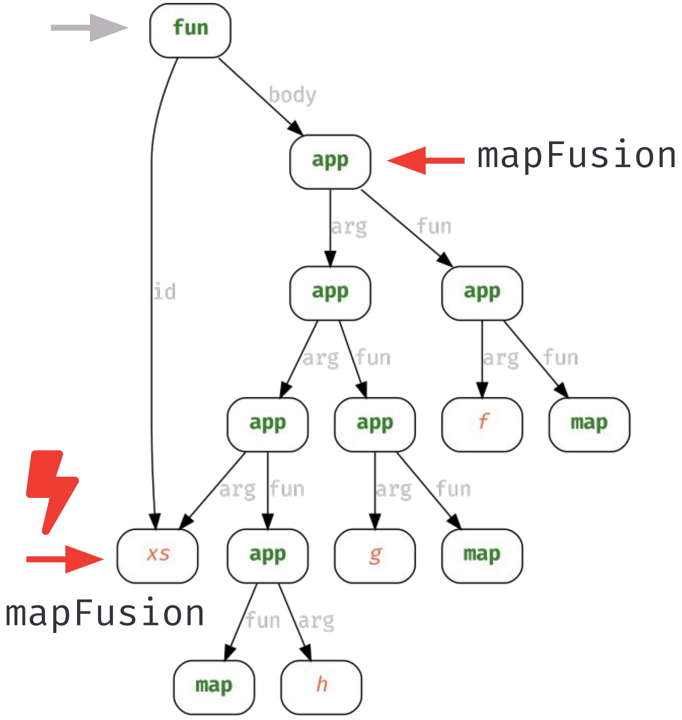
```
all(mapFusion)(threeMaps)
```

all fails if the strategy is not applicable to all children

TRAVERSALS

Describing Precise Locations in the AST

`all(mapFusion)`



`fun(xs ⇒ map(f)(map(g)(map(h)(xs))))`

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals***
to push strategy applications down the AST

```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

Let's try...

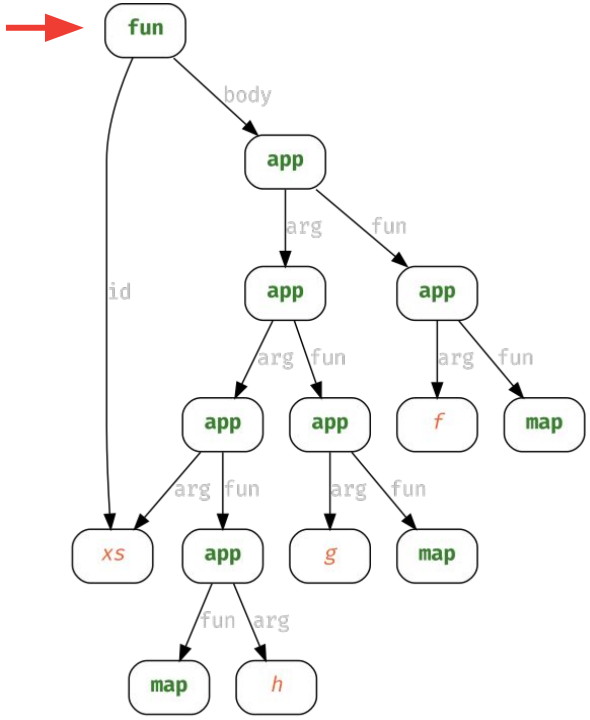
```
all(mapFusion)(threeMaps)
```

all fails if the strategy is not applicable to all children

TRAVERSALS

Describing Precise Locations in the AST

one(mapFusion)



`fun(xs ⇒ map(f)(map(g)(map(h)(xs))))`

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals*** to push strategy applications down the AST

```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

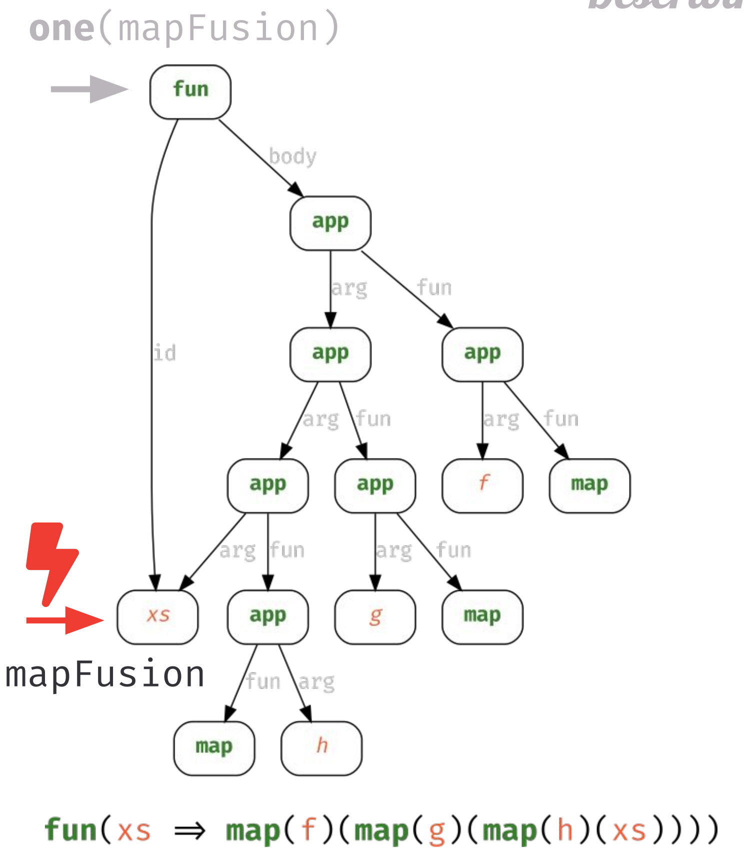
Let's try...

```
one(mapFusion)(threeMaps)
```

one fails if the strategy is not applicable to any child

TRAVERSALS

Describing Precise Locations in the AST



A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals*** to push strategy applications down the AST

```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

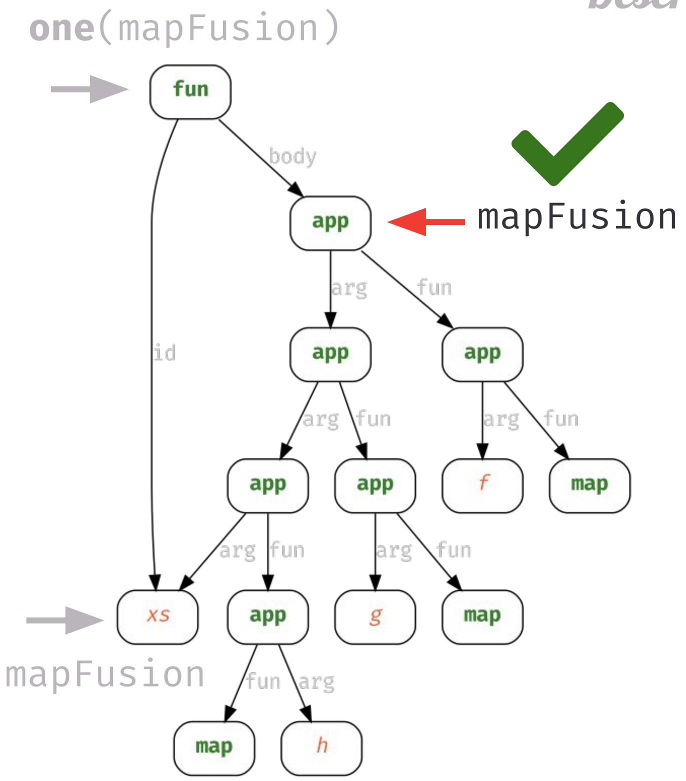
Let's try...

```
one(mapFusion)(threeMaps)
```

one fails if the strategy is not applicable to any child

TRAVERSALS

Describing Precise Locations in the AST



fun(xs ⇒ map(f)(map(g)(map(h)(xs))))

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals*** to push strategy applications down the AST

```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

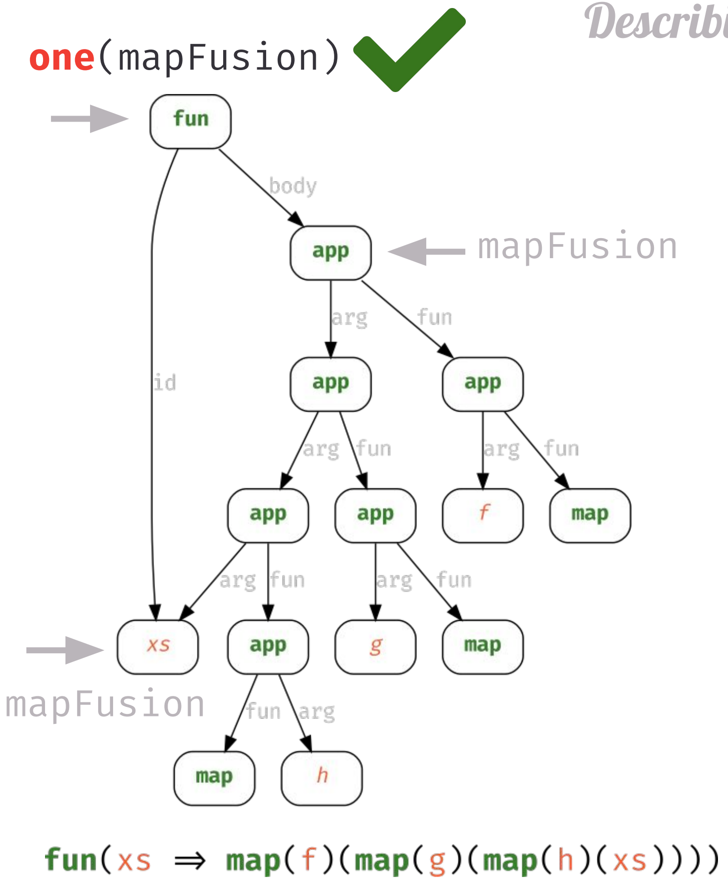
Let's try...

```
one(mapFusion)(threeMaps)
```

one fails if the strategy is not applicable to any child

TRAVERSALS

Describing Precise Locations in the AST



A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals*** to push strategy applications down the AST

```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

Let's try...

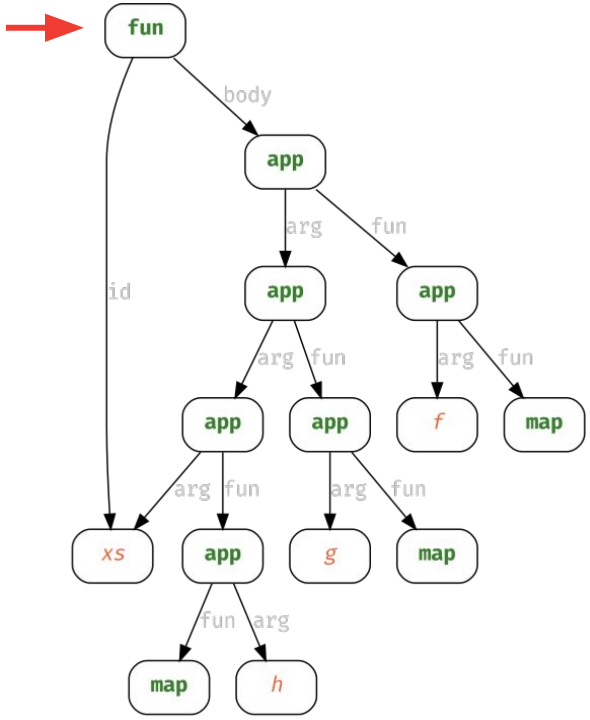
```
one(mapFusion)(threeMaps)
```

one fails if the strategy is not applicable to any child

TRAVERSALS

Describing Precise Locations in the AST

body(mapFusion)



fun(xs ⇒ map(f)(map(g)(map(h)(xs))))

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals*** to push strategy applications down the AST

```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

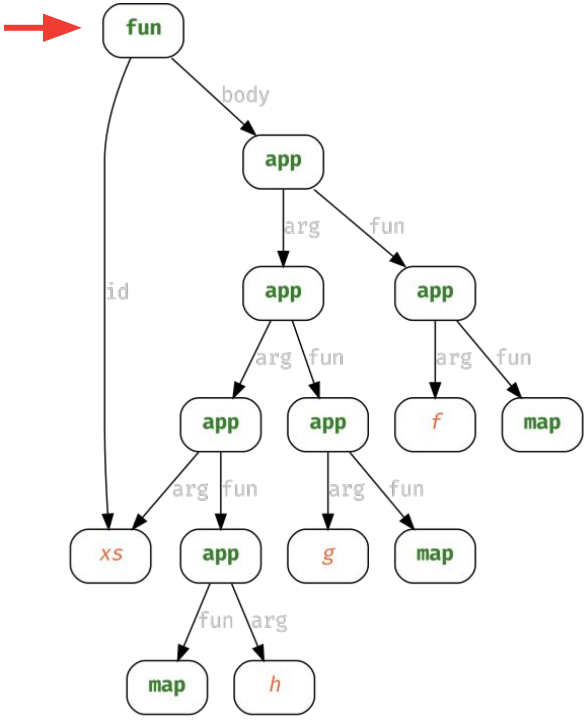
or we define our own ***domain-specific traversals***:

```
def body: Strategy[Lift] => Strategy[Lift] =
  s => p => p match {


                                            }
```

TRAVERSALS

Describing Precise Locations in the AST

body(mapFusion)



fun(xs ⇒ map(f)(map(g)(map(h)(xs))))

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals*** to push strategy applications down the AST

```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

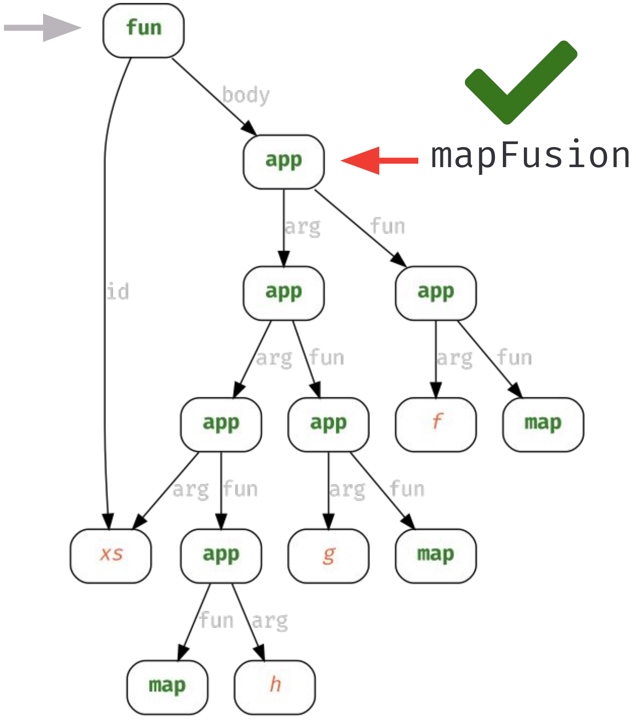
or we define our own ***domain-specific traversals***:

```
def body: Strategy[Lift] => Strategy[Lift] =
  s => p => p match {
    case fun(x,b) =>
      s(b).mapSuccess( nb => fun(x,nb) )
    case _ => Failure( body )                 }
```

TRAVERSALS

Describing Precise Locations in the AST

body(mapFusion)



fun(xs ⇒ map(f)(map(g)(map(h)(xs))))

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals*** to push strategy applications down the AST

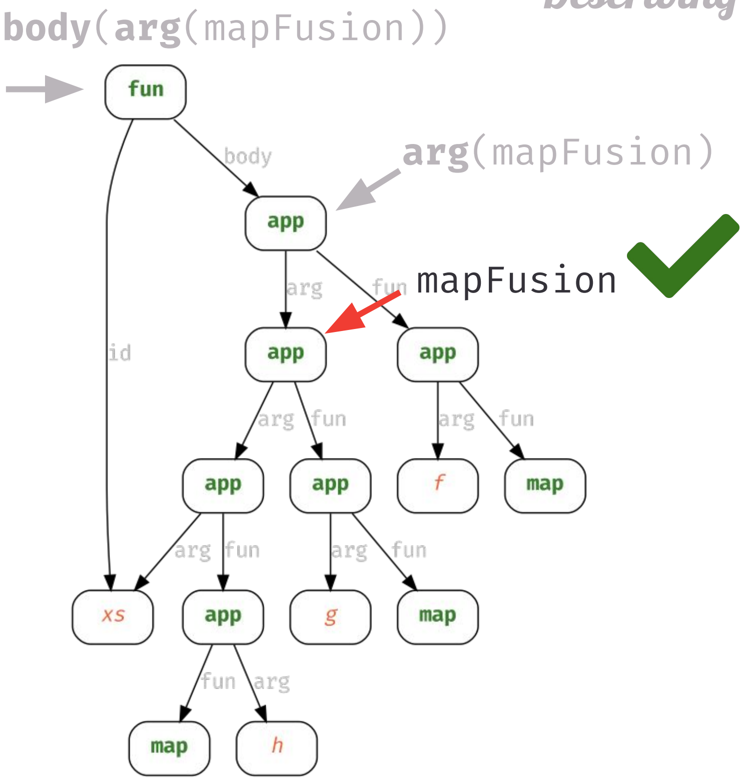
```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

or we define our own ***domain-specific traversals***:

```
def body: Strategy[Lift] => Strategy[Lift] =
  s => p => p match {
    case fun(x,b) =>
      s(b).mapSuccess( nb => fun(x,nb) )
    case _ => Failure( body )          }
```

TRAVERSALS

Describing Precise Locations in the AST



fun(xs ⇒ map(f)(map(g)(map(h)(xs))))

A strategy is generally always applied at the ***root*** of the AST

```
mapFusion(threeMaps)
```

...but we can use ***generic one-level traversals*** to push strategy applications down the AST

```
def all[P] : Strategy[P] => Strategy[P]
def one[P] : Strategy[P] => Strategy[P]
def some[P]: Strategy[P] => Strategy[P]
```

or we define our own ***domain-specific traversals***:

```
def arg: Strategy[Lift] => Strategy[Lift] =
  s => p => p match {
    case app(f,e) =>
      s(e).mapSuccess( ne => app(f,ne) )
    case _ => Failure( arg )                }
```

COMPLETE TRAVERSALS

Go Down More Than One Step

The ***topDown*** traversal traverses the tree until it finds a successful location

```
def topDown[P]: Strategy[P] => Strategy[P] =
    s => p => (s <+ one(topdown(s)))(p)
```

given strategy

input program

apply at root

lChoice

iff s failed, go down one level and try recursively again

COMPLETE TRAVERSALS

Go Down More Than One Step

The ***topDown*** traversal traverses the tree until it finds a successful location

```
def topDown[P]: Strategy[P] => Strategy[P] =
    s => p => (s <+ one(topDown(s)))(p)
```

```
def bottomUp[P]: Strategy[P] => Strategy[P] =
    s => p => (one(bottomUp(s)) <+ s)(p)
```

```
def allTopDown[P]: Strategy[P] => Strategy[P] =
    s => p => (s ; one(allTopDown(s)))(p)
```

COMPLETE TRAVERSALS

Go Down More Than One Step

The ***topDown*** traversal traverses the tree until it finds a successful location

```
def topDown[P]: Strategy[P] => Strategy[P] =
  s => p => (s <+ one(topDown(s)))(p)
```

```
def bottomUp[P]: Strategy[P] => Strategy[P] =
  s => p => (one(bottomUp(s)) <+ s)(p)
```

```
def allTopDown[P]: Strategy[P] => Strategy[P] =
  s => p => (s ; one(allTopDown(s)))(p)
```

or we could also ***normalize*** an AST

```
def normalize[P]: Strategy[P] => Strategy[P] =
  s => p => (repeat(topDown(s))(p)
```

RECAP

What have we seen so far?

With ELEVATE *we are able to…*

- to define ***language-specific transformations*** as strategies
- to compose strategies using ***generic strategy combinators***
- to describe precise locations in the AST using ***generic*** and ***language-specific one-step traversals***
- to compose one-step traversals to define ***whole-tree traversals*** including ***normalization***

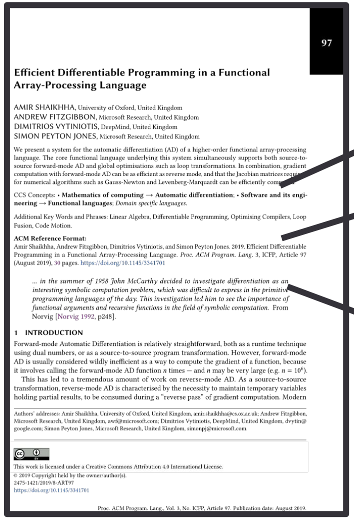
CASE STUDIES

Put it into Practice

AUTOMATIC DIFFERENTIATION

Optimizing F-Smooth using Elevate

ICFP'19: Efficient Differentiable Programming in a Functional Array-Processing Language



97

Efficient Differentiable Programming in a Functional Array-Processing Language

AMIR SHAIKHHA, University of Oxford, United Kingdom
ANDREW FITZGIBBON, Microsoft Research, United Kingdom
DIMITRIOS VYTINIOTIS, DeepMind, United Kingdom
SIMON PEYTON JONES, Microsoft Research, United Kingdom

We present a system for the automatic differentiation (AD) of a higher-order functional array-processing language. The core functional language underlying this system simultaneously supports both source-to-source forward-mode AD and global optimisations such as loop transformations. In combination, gradient computation with forward-mode AD can be as efficient as reverse mode, and that the Jacobian matrices requi[illegible] for numerical algorithms such as Gauss-Newton and Levenberg-Marquardt can be efficiently comp[illegible].

CCS Concepts: • **Mathematics of computing → Automatic differentiation**; • **Software and its engineering → Functional languages**; *Domain specific languages.*

Additional Key Words and Phrases: Linear Algebra, Differentiable Programming, Optimising Compilers, Loop Fusion, Code Motion.

ACM Reference Format:
Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient Differentiable Programming in a Functional Array-Processing Language. *Proc. ACM Program. Lang.* 3, ICFP, Article 97 (August 2019), 30 pages. https://doi.org/10.1145/3341701

> *... in the summer of 1958 John McCarthy decided to investigate differentiation as an interesting symbolic computation problem, which was difficult to express in the primitive programming languages of the day. This investigation led him to see the importance of functional arguments and recursive functions in the field of symbolic computation.* From Norvig [Norvig 1992, p248].

1 INTRODUCTION

Forward-mode Automatic Differentiation is relatively straightforward, both as a runtime technique using dual numbers, or as a source-to-source program transformation. However, forward-mode AD is usually considered wildly inefficient as a way to compute the gradient of a function, because it involves calling the forward-mode AD function n times — and n may be very large (e.g. $n = 10^6$).

This has led to a tremendous amount of work on reverse-mode AD. As a source-to-source transformation, reverse-mode AD is characterised by the necessity to maintain temporary variables holding partial results, to be consumed during a "reverse pass" of gradient computation. Modern

Authors' addresses: Amir Shaikhha, University of Oxford, United Kingdom, amir.shaikhha@cs.ox.ac.uk; Andrew Fitzgibbon, Microsoft Research, United Kingdom, awf@microsoft.com; Dimitrios Vytiniotis, DeepMind, United Kingdom, dvytin@google.com; Simon Peyton Jones, Microsoft Research, United Kingdom, simonpj@microsoft.com.

2475-1421/2019/8-ART97
https://doi.org/10.1145/3341701

Arbitrary F-Smooth expressions are differentiable

They achieve efficiency by ***rewriting*** differentiated code

"The strategy for applying rewrite rules can become tricky"

Use ELEVATE for optimizing F-Smooth programs

AUTOMATIC DIFFERENTIATION

Optimizing F-Smooth using Elevate

$$\texttt{length}\,(\texttt{build}\; e_0\; e_1) \quad \rightsquigarrow \quad e_0$$

(fun x -> e_0) e_1	$\rightsquigarrow$	let x = e_1 in e_0
let x = e_0 in e_1	$\rightsquigarrow$	$e_1[x \mapsto e_0]$
let x = e_0 in e_1	$\rightsquigarrow$	e_1 ($x \notin fvs(e_1)$)
let x = let y = e_0 in e_1 in e_2	$\rightsquigarrow$	let y = e_0 in let x = e_1 in e_2
let x = e_0 in let y = e_0 in e_1	$\rightsquigarrow$	let x = e_0 in let y = x in e_1
let x = e_0 in let y = e_1 in e_2	$\rightsquigarrow$	let y = e_1 in let x = e_0 in e_2
f(let x = e_0 in e_1)	$\rightsquigarrow$	let x = e_0 in f(e_1)

(a) λ-Calculus Rules

e + 0 = 0 + e	$\rightsquigarrow$	e
e * 1 = 1 * e	$\rightsquigarrow$	e
e * 0 = 0 * e	$\rightsquigarrow$	0
e + -e = e - e	$\rightsquigarrow$	0
$e_0 * e_1 + e_0 * e_2$	$\rightsquigarrow$	$e_0 * (e_1 + e_2)$

(b) Ring-Structure Rules

(build e_0 e_1)[e_2]	$\rightsquigarrow$	e_1 e_2
$\boxed{\text{length (build } e_0\ e_1) \rightsquigarrow e_0}$		

(c) Loop Fusion Rules

if true then e_1 else e_2	$\rightsquigarrow$	e_1
if false then e_1 else e_2	$\rightsquigarrow$	e_2
if e_0 then e_1 else e_1	$\rightsquigarrow$	e_1
if e_0 then e_1 else e_2	$\rightsquigarrow$	if e_0 then $e_1[e_0 \mapsto \text{true}]$ else $e_2[e_0 \mapsto \text{false}]$
f (if e_0 then e_1 else e_2)	$\rightsquigarrow$	if e_0 then f (e_1) else f (e_2)

(d) Conditional Rules

ifold f z 0	$\rightsquigarrow$	z
ifold f z n	$\rightsquigarrow$	ifold (fun a i -> f a (i+1)) (f z 0) (n - 1)
ifold (fun a i -> a) z n	$\rightsquigarrow$	z
ifold (fun a i -> if(i = e_0) then e_1 else a) z n	$\rightsquigarrow$	let a = z in let i = e_0 in e_1 *(if e_0 does not mention a or i)*

(e) Loop Normalisation Rules

fst (e_0, e_1)	$\rightsquigarrow$	e_0
snd (e_0, e_1)	$\rightsquigarrow$	e_1

(f) Tuple Normalisation Rules

ifold (fun a i -> (f_0 (fst a) i, f_1 (snd a) i)) (z_0, z_1) n $\rightsquigarrow$ (ifold f_0 z_0 n, ifold f_1 z_1 n)

(g) Loop Fission Rule

Fig. 8. Transformation Rules for $\widetilde{F}$. Even though none of these rules are AD-specific, the rules of Figure 8f and Figure 8g are more useful in the AD context.

F-Smooth Rewrite Rules

AUTOMATIC DIFFERENTIATION

Optimizing F-Smooth using Elevate

(fun x -> e_0) e_1	$\leadsto$	let x = e_1 in e_0
let x = e_0 in e_1	$\leadsto$	$e_1[x \mapsto e_0]$
let x = e_0 in e_1	$\leadsto$	e_1 ($x \notin fvs(e_1)$)
let x = let y = e_0 in e_1 in e_2	$\leadsto$	let y = e_0 in let x = e_1 in e_2
let x = e_0 in let y = e_0 in e_1	$\leadsto$	let x = e_0 in let y = x in e_1
let x = e_0 in let y = e_1 in e_2	$\leadsto$	let y = e_1 in let x = e_0 in e_2
f(let x = e_0 in e_1)	$\leadsto$	let x = e_0 in f(e_1)

(a) λ-Calculus Rules

e + 0 = 0 + e	$\leadsto$	e
e * 1 = 1 * e	$\leadsto$	e
e * 0 = 0 * e	$\leadsto$	0
e + -e = e - e	$\leadsto$	0
$e_0 * e_1 + e_0 * e_2$	$\leadsto$	$e_0 * (e_1 + e_2)$

(b) Ring-Structure Rules

(build e_0 e_1)[e_2]	$\leadsto$	e_1 e_2
length (build e_0 e_1)	$\leadsto$	e_0

(c) Loop Fusion Rules

if true then e_1 else e_2	$\leadsto$	e_1
if false then e_1 else e_2	$\leadsto$	e_2
if e_0 then e_1 else e_1	$\leadsto$	e_1
if e_0 then e_1 else e_2	$\leadsto$	if e_0 then $e_1[e_0 \mapsto$ true] else $e_2[e_0 \mapsto$ false]
f (if e_0 then e_1 else e_2)	$\leadsto$	if e_0 then f (e_1) else f (e_2)

(d) Conditional Rules

ifold f z 0	$\leadsto$	z
ifold f z n	$\leadsto$	ifold (fun a i -> f a (i+1)) (f z 0) (n - 1)
ifold (fun a i -> a) z n	$\leadsto$	z
ifold (fun a i -> if(i = e_0) then e_1 else a) z n	$\leadsto$	let a = z in let i = e_0 in e_1 *(if e_0 does not mention a or i)*

(e) Loop Normalisation Rules

fst (e_0, e_1)	$\leadsto$	e_0
snd (e_0, e_1)	$\leadsto$	e_1

(f) Tuple Normalisation Rules

ifold (fun a i -> (f_0 (fst a) i, f_1 (snd a) i)) (z_0, z_1) n	$\leadsto$	(ifold f_0 z_0 n, ifold f_1 z_1 n)

(g) Loop Fission Rule

Fig. 8. Transformation Rules for $\widetilde{\mathrm{F}}$. Even though none of these rules are AD-specific, the rules of Figure 8f and Figure 8g are more useful in the AD context.

F-Smooth Rewrite Rules

length (build e_0 e_1) $\leadsto$ e_0

ELEVATE

```
def lengthBuild: Strategy[FSmooth] =
  (p:FSmooth) => p match {
    case length(build(e0,e1) => Success( e0 )
    case _    => Failure( lengthBuild )
  }
```

AUTOMATIC DIFFERENTIATION

Optimizing F-Smooth using Elevate

(fun x -> e_0) e_1	$\rightsquigarrow$	let x = e_1 in e_0
let x = e_0 in e_1	$\rightsquigarrow$	$e_1[x \mapsto e_0]$
let x = e_0 in e_1	$\rightsquigarrow$	e_1 ($x \notin fvs(e_1)$)
let x = let y = e_0 in e_1 in e_2	$\rightsquigarrow$	let y = e_0 in let x = e_1 in e_2
let x = e_0 in let y = e_0 in e_1	$\rightsquigarrow$	let x = e_0 in let y = x in e_1
let x = e_0 in let y = e_1 in e_2	$\rightsquigarrow$	let y = e_1 in let x = e_0 in e_2
f(let x = e_0 in e_1)	$\rightsquigarrow$	let x = e_0 in f(e_1)

(a) λ-Calculus Rules

e + 0 = 0 + e	$\rightsquigarrow$	e
e * 1 = 1 * e	$\rightsquigarrow$	e
e * 0 = 0 * e	$\rightsquigarrow$	0
e + -e = e - e	$\rightsquigarrow$	0
$e_0 * e_1 + e_0 * e_2$	$\rightsquigarrow$	$e_0 * (e_1 + e_2)$

(b) Ring-Structure Rules

~~(build e_0 e_1)[e_2]~~	$\rightsquigarrow$	e_1 e_2
$\boxed{\text{length (build } e_0\ e_1)}$	$\rightsquigarrow$	e_0

(c) Loop Fusion Rules

if true then e_1 else e_2	$\rightsquigarrow$	e_1
if false then e_1 else e_2	$\rightsquigarrow$	e_2
if e_0 then e_1 else e_1	$\rightsquigarrow$	e_1
if e_0 then e_1 else e_2	$\rightsquigarrow$	if e_0 then $e_1[e_0 \mapsto$ true] else $e_2[e_0 \mapsto$ false]
f (if e_0 then e_1 else e_2)	$\rightsquigarrow$	if e_0 then f (e_1) else f (e_2)

(d) Conditional Rules

ifold f z 0	$\rightsquigarrow$	z
ifold f z n	$\rightsquigarrow$	ifold (fun a i -> f a (i+1)) (f z 0) (n - 1)
ifold (fun a i -> a) z n	$\rightsquigarrow$	z
ifold (fun a i -> if(i = e_0) then e_1 else a) z n	$\rightsquigarrow$	let a = z in let i = e_0 in e_1 *(if e_0 does not mention a or i)*

(e) Loop Normalisation Rules

fst (e_0, e_1)	$\rightsquigarrow$	e_0
snd (e_0, e_1)	$\rightsquigarrow$	e_1

(f) Tuple Normalisation Rules

ifold (fun a i -> (f_0 (fst a) i, f_1 (snd a) i)) (z_0, z_1) n	$\rightsquigarrow$	(ifold f_0 z_0 n, ifold f_1 z_1 n)

(g) Loop Fission Rule

Fig. 8. Transformation Rules for $\widetilde{F}$. Even though none of these rules are AD-specific, the rules of Figure 8f and Figure 8g are more useful in the AD context.

F-Smooth Rewrite Rules

$$\text{length (build } e_0\ e_1) \quad \rightsquigarrow \quad e_0$$

$\downarrow$ ELEVATE

```
def lengthBuild: Strategy[FSmooth] =
  (p:FSmooth) => p match {
    case length(build(e0,e1) => Success( e0 )
    case _     => Failure( lengthBuild )
  }
```

Example 5: Simplification: $(M^T)^T = M$

```
normalize(lengthBuild <+ …)((Mᵀ)ᵀ) = Success(M)
```

We are able to ***trace*** the rule applications: Here, 12 steps

AUTOMATIC DIFFERENTIATION

Optimizing F-Smooth using Elevate

(fun x -> e_0) e_1	⇝	let x = e_1 in e_0
let x = e_0 in e_1	⇝	$e_1[x \mapsto e_0]$
let x = e_0 in e_1	⇝	e_1 ($x \notin fvs(e_1)$)
let x = let y = e_0 in e_1 in e_2	⇝	let y = e_0 in let x = e_1 in e_2
let x = e_0 in let y = e_0 in e_1	⇝	let x = e_0 in let y = x in e_1
let x = e_0 in let y = e_1 in e_2	⇝	let y = e_1 in let x = e_0 in e_2
f(let x = e_0 in e_1)	⇝	let x = e_0 in f(e_1)

(a) λ-Calculus Rules

e + 0 = 0 + e	⇝	e
e * 1 = 1 * e	⇝	e
e * 0 = 0 * e	⇝	0
e + -e = e - e	⇝	0
$e_0 * e_1 + e_0 * e_2$	⇝	$e_0 * (e_1 + e_2)$

(b) Ring-Structure Rules

(build e_0 e_1)[e_2]	⇝	e_1 e_2
$\boxed{\text{length (build } e_0\ e_1) \rightsquigarrow e_0}$		

(c) Loop Fusion Rules

if true then e_1 else e_2	⇝	e_1
if false then e_1 else e_2	⇝	e_2
if e_0 then e_1 else e_1	⇝	e_1
if e_0 then e_1 else e_2	⇝	if e
f (if e_0 then e_1 else e_2)	⇝	if e

(d) Cond

ifold f z 0	⇝	z
ifold f z n	⇝	ifold (fun a i -> f a (i+1)) (f z 0) (n - 1)
ifold (fun a i -> a) z n	⇝	z
ifold (fun a i -> if(i = e_0) then e_1 else a) z n	⇝	let a = z in let i = e_0 in e_1 (*if e_0 does not mention a or i*)

(e) Loop Normalisation Rules

fst (e_0, e_1)	⇝	e_0
snd (e_0, e_1)	⇝	e_1

(f) Tuple Normalisation Rules

ifold (fun a i -> (f_0 (fst a) i, f_1 (snd a) i)) (z_0, z_1) n ⇝ (ifold f_0 z_0 n, ifold f_1 z_1 n)

(g) Loop Fission Rule

Fig. 8. Transformation Rules for $\widetilde{F}$. Even though none of these rules are AD-specific, the rules of Figure 8f and Figure 8g are more useful in the AD context.

F-Smooth Rewrite Rules

length (build e_0 e_1) ⇝ e_0

ELEVATE

```
def lengthBuild: Strategy[FSmooth] =
                                        0 )
```

Flexible: ELEVATE is able to implement and optimize existing rewrite systems

Example 5: Simplification: $(M^T)^T = M$

```
normalize(lengthBuild <+ …)((Mᵀ)ᵀ) = Success(M)
```

We are able to ***trace*** the rule applications: Here, 12 steps

IMAGE PROCESSING

Expressing Separable Convolution with Elevate and Lift

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

Separable Convolution: Sobel Filter

```
Var x, y; Func out; Func in = BC::repeat_edge(input);
out(x, y) = (
  1.f * in(x-1,y-1) + 2.f * in(x,y-1) + 1.f * in(x+1,y-1) +
  2.f * in(x-1,y)   + 4.f * in(x,y)   + 2.f * in(x+1,y)   +
  1.f * in(x-1,y+1) + 2.f * in(x,y+1) + 1.f * in(x+1,y+1)
) * (1.f/16.f);
```

Halide: 2D Convolution

no schedule for this optimization

```
Var x,y; Func b_x,b_y,out; Func in=BC::repeat_edge(input);
b_y(x, y) =  in(x, y-1) + 2.f * in(x, y)  +  in(x, y+1);
b_x(x, y) = b_y(x-1, y) + 2.f * b_y(x, y) + b_y(x+1, y);
out(x, y) = b_x(x, y) * (1.f/16.f);
```

Halide: Separated Convolution

IMAGE PROCESSING

Expressing Separable Convolution with Elevate and Lift

$$\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix} = \begin{bmatrix}1\\ 2\\ 1\end{bmatrix} \times \begin{bmatrix}-1 & 0 & 1\end{bmatrix}$$

Separable Convolution: Sobel Filter

```
Var x, y; Func out; Func in = BC::repeat_edge(input);
out(x, y) = (
  1.f * in(x-1,y-1) + 2.f * in(x,y-1) + 1.f * in(x+1,y-1) +
  2.f * in(x-1,y)   + 4.f * in(x,y)   + 2.f * in(x+1,y)   +
  1.f * in(x-1,y+1) + 2.f * in(x,y+1) + 1.f * in(x+1,y+1)
) * (1.f/16.f);
```

Halide: 2D Convolution

↓ ***no schedule for this optimization***

```
Var x,y; Func b_x,b_y,out; Func in=BC::repeat_edge(input);
b_y(x, y) =  in(x, y-1) + 2.f * in(x, y)  +  in(x, y+1);
b_x(x, y) = b_y(x-1, y) + 2.f * b_y(x, y) + b_y(x+1, y);
out(x, y) = b_x(x, y) * (1.f/16.f);
```

Halide: Separated Convolution

```
img |>
  pad2D(1) |>                 // boundary handling
    slide2D(3)(1) |>          // neighborhood creation
      map2D(fun(nbh =>        // 2D stencil computation
        dot(join(weights2d))(join(nbh))))))
```

Lift: 2D Convolution

```
img |>
  pad2D(1) |>
    slide2D(3)(1) |>
      map2D(fun(nbh => nbh |> // 2 1D stencils
        map(dot(weightsH)) |> map(dot(weightsV)) ))
```

Lift: Separated Convolution

IMAGE PROCESSING

Expressing Separable Convolution with Elevate and Lift

$$\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\end{bmatrix} = \begin{bmatrix}1\\ 2\end{bmatrix} \times \begin{bmatrix}-1 & 0 & 1\end{bmatrix}$$

```
def separateConv(w2d:Lift, wh:Lift, wv:Lift): Strategy[Lift] = p =>
 p match {
  case app(app(app(reduce, add), 0), app(app(map, mult),
        app(app(zip, app(join, w)), app(join, nbh)))) if w==w2d =>
            Success(nbh |> map(dot(wh)) |> dot(wv))
  case _ => Failure(separateConv)                                 }
```

```
  1.f * in(x-1,y-1) + 2.f * in(x,y-1) + 1.f * in(x+1,y-1) +
  2.f * in(x-1,y)   + 4.f * in(x,y)   + 2.f * in(x+1,y)   +
  1.f * in(x-1,y+1) + 2.f * in(x,y+1) + 1.f * in(x+1,y+1)
) * (1.f/16.f);
```

Halide: 2D Convolution

```
img |>
  pad2D(1) |>              // boundary handling
    slide2D(3)(1) |>       // neighborhood creation
      map2D(fun(nbh =>     // 2D stencil computation
        dot(join(weights2d))(join(nbh)))))
```

Lift: 2D Convolution

```
topDown(separateConv)(conv2D)
```

ELEVATE: Separate Convolution using Strategies

no schedule for this optimization

```
Var x,y; Func b_x,b_y,out; Func in=BC::repeat_edge(input);
b_y(x, y) =  in(x, y-1) + 2.f * in(x, y)  +  in(x, y+1);
b_x(x, y) = b_y(x-1, y) + 2.f * b_y(x, y) + b_y(x+1, y);
out(x, y) = b_x(x, y) * (1.f/16.f);
```

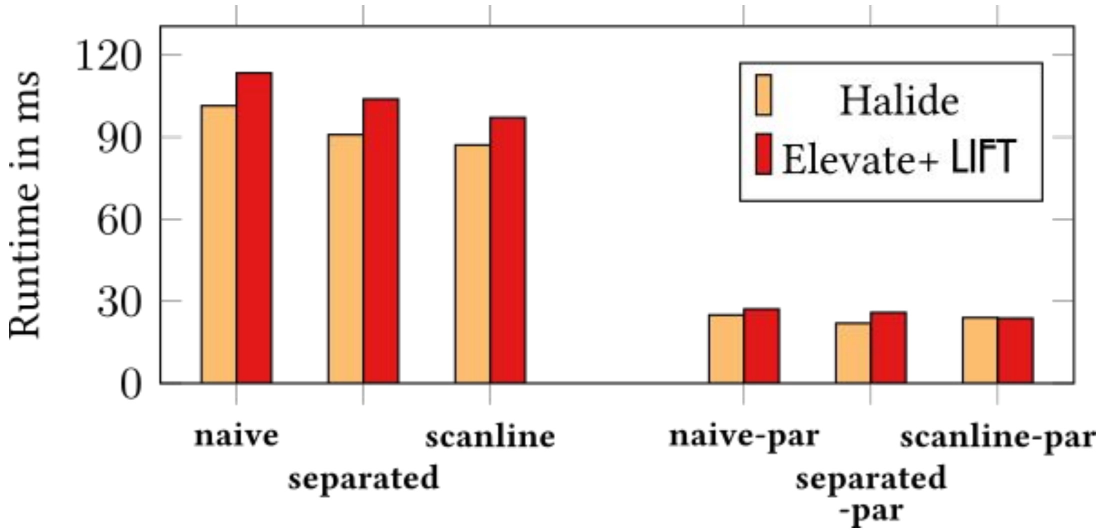
Halide: Separated Convolution

```
img |>
  pad2D(1) |>
    slide2D(3)(1) |>
      map2D(fun(nbh => nbh |> // 2 1D stencils
        map(dot(weightsH)) |> map(dot(weightsV)) ))
```

Lift: Separated Convolution

IMAGE PROCESSING

Expressing Separable Convolution with Elevate and Lift



Our strategies achieve the ***same trend*** in performance
→ they encode the ***same optimizations*** as described by the schedules

Lift: Separated Convolution

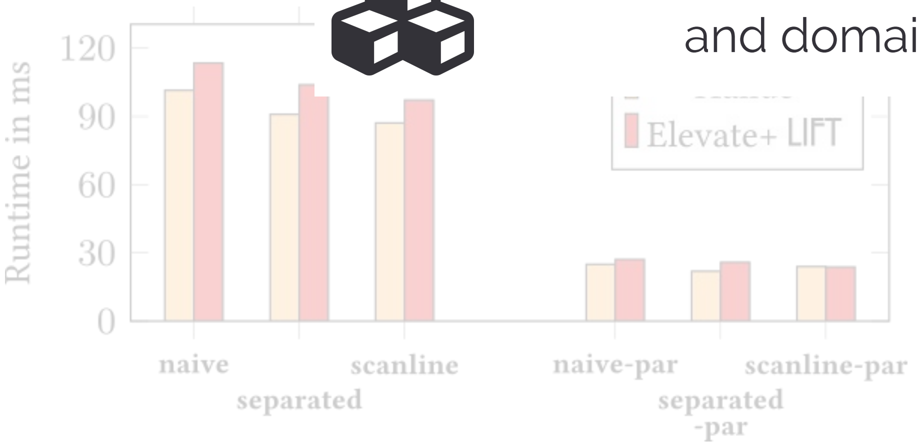
IMAGE PROCESSING

Expressing Separable Convolution with Elevate and Lift

```
def separateConv(w2d:Lift, wh:Lift, wv:Lift): Strategy[Lift] = p =>
 p match {
  case app(app(app(reduce, add), 0), app(app(map, mult),
        app(app(zip, app(join, w)), app(join, nbh)))) if w==w2d =>
            Success(nbh |> map(dot(wh)) |> dot(wv))
  case _ => Failure(separateConv)                                    }
```

```
img |>
  pad2D(1) |>                  // boundary handling
    slide2D(3)(1) |>           // neighborhood creation
      map2D(fun(nbh =>         // 2D stencil computation
        dot(join(weights2d))(join(nbh)))))
```

Extensible: ELEVATE allows to create your own custom and domain-specific optimizations



ELEVATE: Separate Convolution using Strategies

```
img |>
  pad2D(1) |>
    slide2D(3)(1) |>
      map2D(fun(nbh => nbh |> // 2 1D stencils
        map(dot(weightsH)) |> map(dot(weightsV)) ))
```

Lift: Separated Convolution

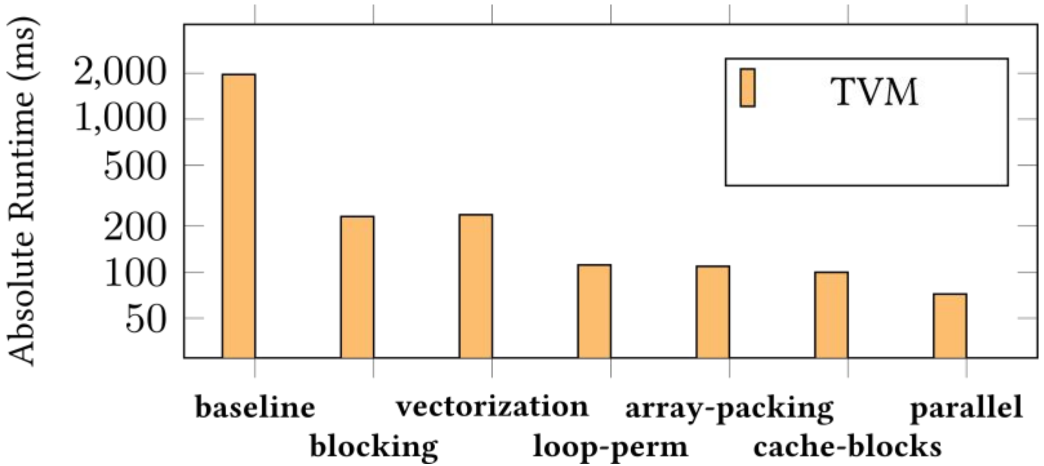
DEEP LEARNING

Implementing a Scheduling Language using Strategies

tvm ***Tutorial***: How to optimize GEMM

```
# Algorithm
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N), lambda x, y:
 tvm.sum(A[x, k] * B[k, y], axis=k),
 name='C')
```

TVM: Matrix Multiplication



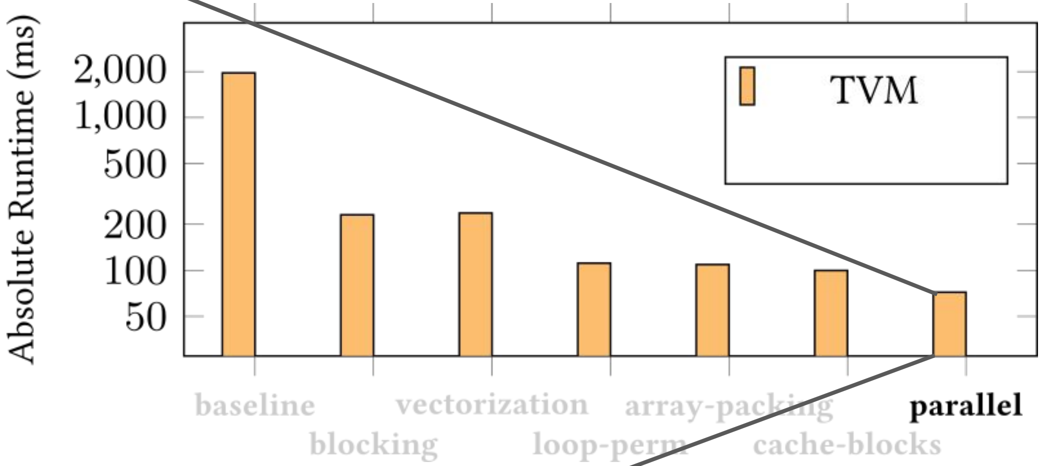
In this tutorial, we will demonstrate how to use TVM to optimize square matrix multiplication and achieve ***200 times faster*** than baseline by simply adding ***18 extra lines of code***.

DEEP LEARNING

Implementing a Scheduling Language using Strategies

```
# "parallel schedule
s = tvm.create_schedule(C.op)
CC = s.cache_write(C, 'global')

xo, yo, xi, yi = s[C].tile(
  C.op.axis[0], C.op.axis[1], bn, bn)

s[CC].compute_at(s[C], yo)

xc, yc = s[CC].op.axis

k, = s[CC].op.reduce_axis
ko, ki = s[CC].split(k, factor=4)
s[CC].reorder(ko, xc, ki, yc)
s[CC].unroll(ki)
s[CC].vectorize(yc)

s[C].parallel(xo)

x, y, z = s[packedB].op.axis
s[packedB].vectorize(z)
s[packedB].parallel(x)
```

Tutorial: How to optimize GEMM



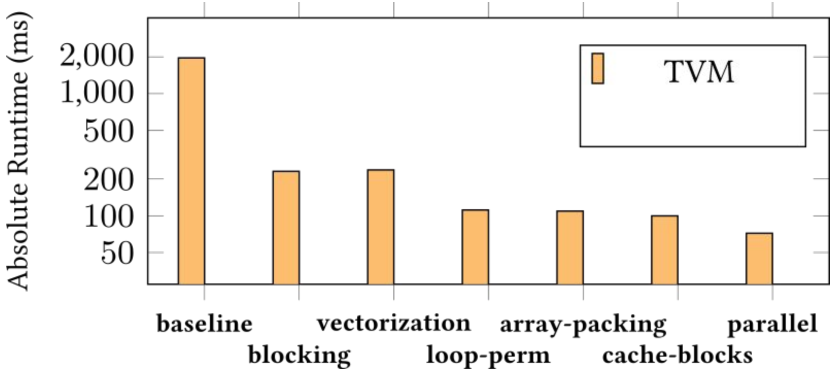
Some versions require to manually change the algorithm again!

DEEP LEARNING

Implementing a Scheduling Language using Strategies

```
# Algorithm
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N), lambda x, y:
 tvm.sum(A[x, k] * B[k, y], axis=k),
 name='C')
```

TVM: Matrix Multiplication

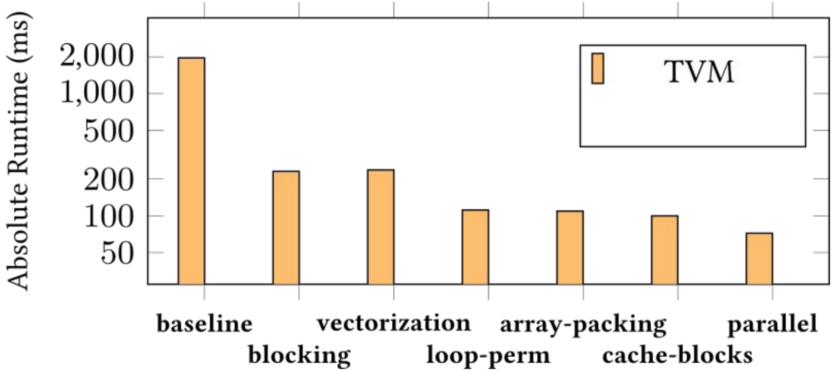


DEEP LEARNING

Implementing a Scheduling Language using Strategies

```
# Algorithm
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N), lambda x, y:
 tvm.sum(A[x, k] * B[k, y], axis=k),
 name='C')
```

TVM: Matrix Multiplication




```
val dot = fun((a,b) => zip(a,b) |> map(*) |> reduce(+,0))
val mm = fun(a :: M.K.float => fun(b :: K.N.float =>
  map( fun(arow => // iterating over M
    map( fun(bcol => // iterating over N
      dot(arow, bcol)  // iterating over K
    )(transpose(b))
  )(a)
```

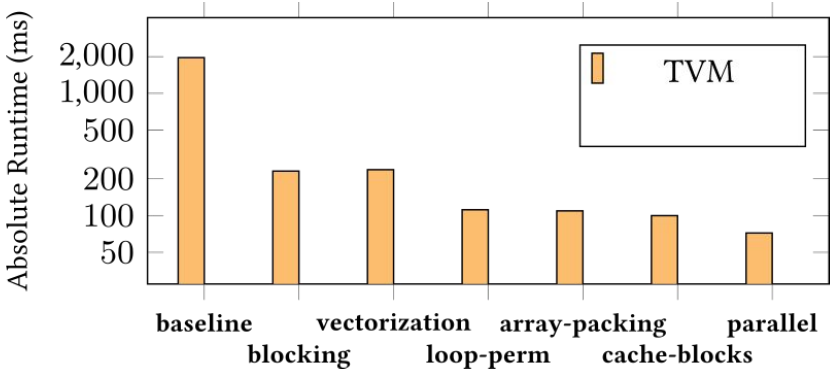
Lift: Matrix Multiplication

DEEP LEARNING

Implementing a Scheduling Language using Strategies

```
# Algorithm
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N), lambda x, y:
 tvm.sum(A[x, k] * B[k, y], axis=k),
 name='C')
```

TVM: Matrix Multiplication




```
val dot = fun((a,b) => zip(a,b) |> map(*) |> reduce(+,0))
val mm = fun(a :: M.K.float => fun(b :: K.N.float =>
  map( fun(arow => // iterating over M
    map( fun(bcol => // iterating over N
      dot(arow, bcol)  // iterating over K
    )(transpose(b))
  )(a)
```

Lift: Matrix Multiplication

```
(DFNF ;                          // normalize AST
 topDown(fuseMapReduce) ;        // loop-fusion
 lowerToC                        // lowering
)(mm)
```

"baseline" ELEVATE strategy

No implicit optimizations!

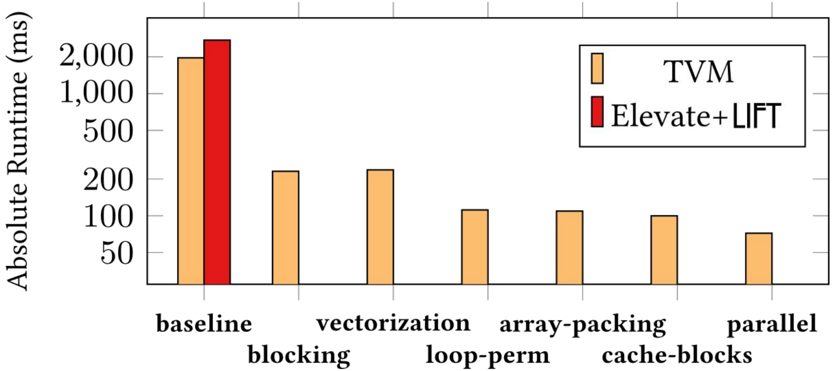
Every transformation is explicit and therefore customizable

DEEP LEARNING

Implementing a Scheduling Language using Strategies

```
# Algorithm
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N), lambda x, y:
 tvm.sum(A[x, k] * B[k, y], axis=k),
 name='C')
```

TVM: Matrix Multiplication




```
val dot = fun((a,b) => zip(a,b) |> map(*) |> reduce(+,0))
val mm = fun(a :: M.K.float => fun(b :: K.N.float =>
  map( fun(arow => // iterating over M
    map( fun(bcol => // iterating over N
      dot(arow, bcol)  // iterating over K
    )(transpose(b))
  )(a)
```

Lift: Matrix Multiplication

```
(DFNF ;                          // normalize AST
 topDown(fuseMapReduce) ;        // loop-fusion
 lowerToC                        // lowering
)(mm)
```

"baseline" ELEVATE strategy

No implicit optimizations!

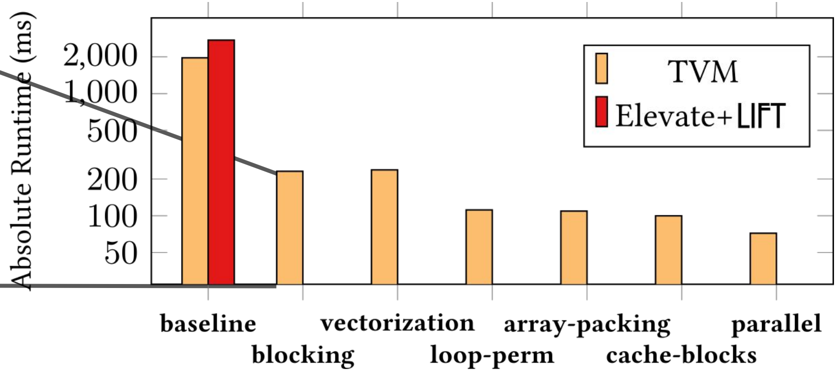
Every transformation is explicit and therefore customizable

DEEP LEARNING

Implementing a Scheduling Language using Strategies

```
# blocking version
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0],C.op.axis[1],32,32)
k,      = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)
s[C].reorder(xo, yo, ko, ki, xi, yi)
```

TVM: blocking schedule

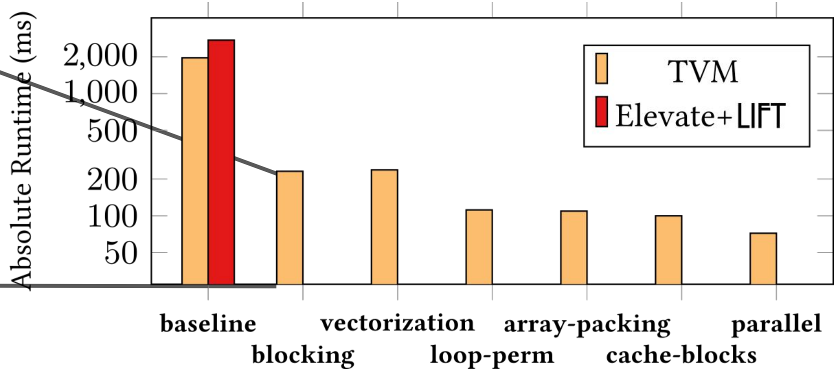


DEEP LEARNING

Implementing a Scheduling Language using Strategies

```
# blocking version
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0],C.op.axis[1],32,32)
k,       = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=4)
s[C].reorder(xo, yo, ko, ki, xi, yi)
```

TVM: blocking schedule

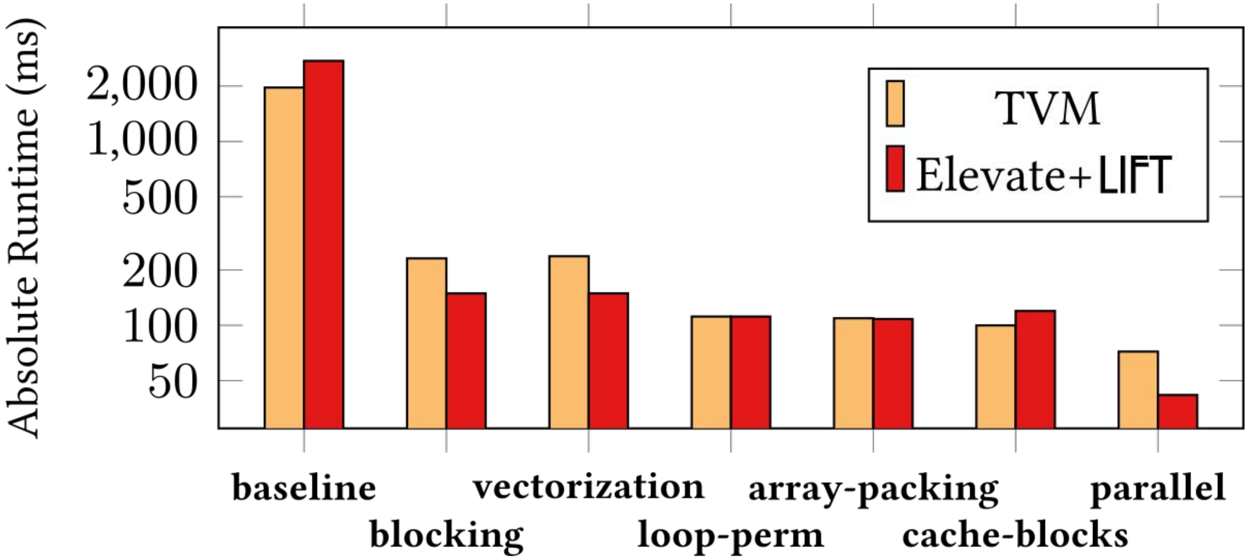



```
val blocking = (topDown(tile(32,32)) ;
                topDown( isReduce ; split(4) ) ;
                topDown(reorder(Seq(1,2,5,6,3,4))) )
(blocking ; lowerToC)(mm)
```

ELEVATE: blocking strategy

DEEP LEARNING

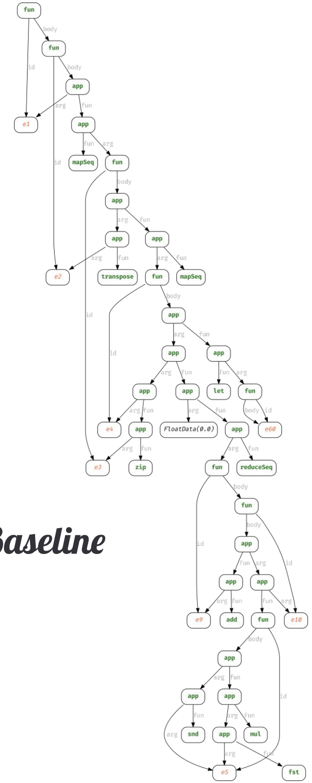
Implementing a Scheduling Language using Strategies



Our strategies achieve the ***same trend*** in performance
→ they encode the ***same optimizations*** as described by the schedules

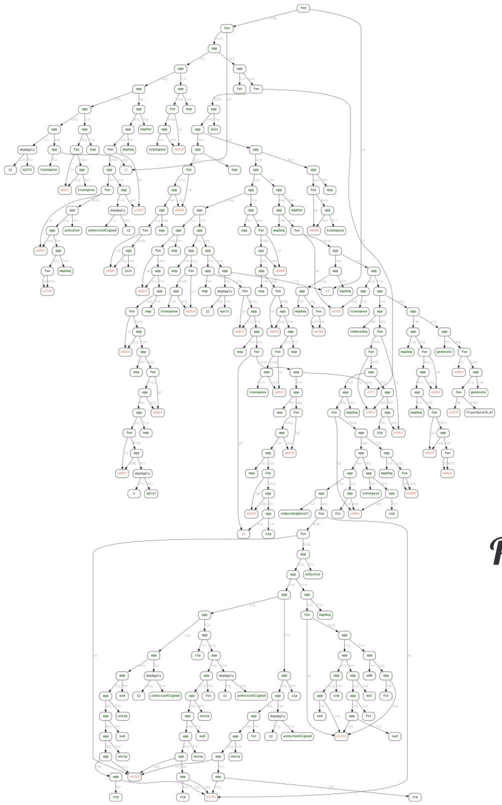
DEEP LEARNING

Implementing a Scheduling Language using Strategies



Baseline

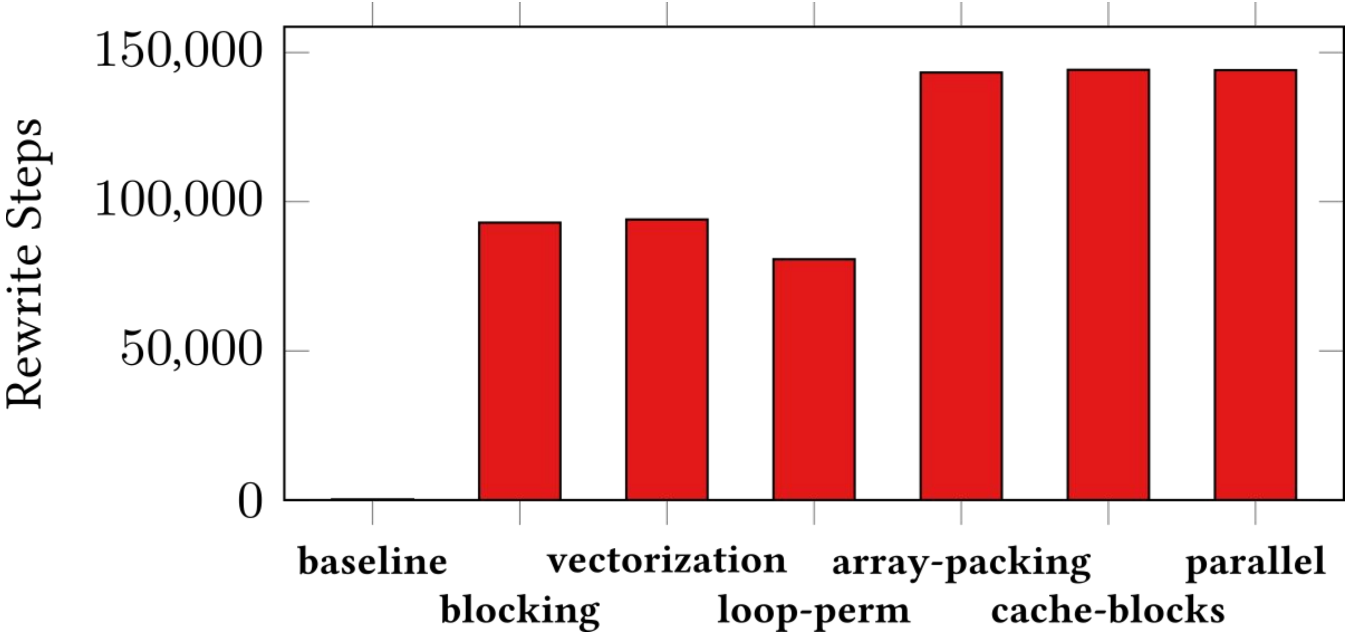
ELEVATE →



Parallel

DEEP LEARNING

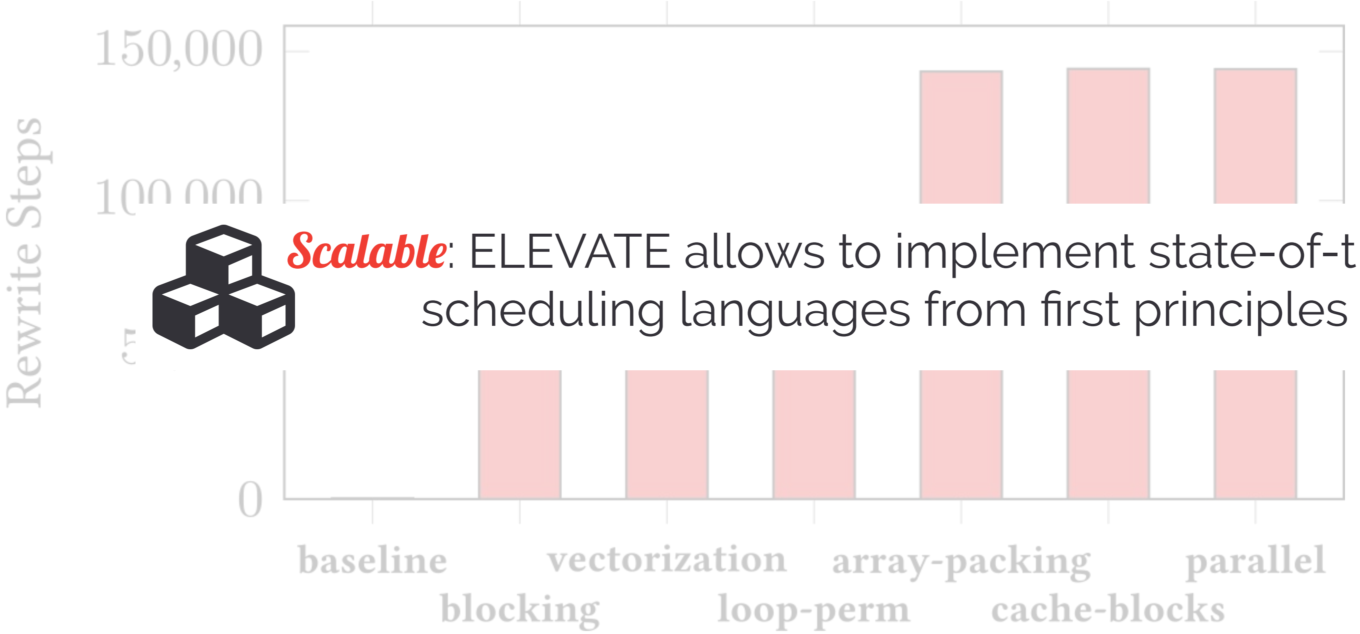
Implementing a Scheduling Language using Strategies



Rewriting the input program requires ***less than 60 seconds*** per version

DEEP LEARNING

Implementing a Scheduling Language using Strategies



Scalable: ELEVATE allows to implement state-of-the art scheduling languages from first principles

Rewriting the baseline version requires ***less than 60 seconds*** per version

FUTURE WORK

Would this be of interest for Myelin or related projects?

I'm in my "final" PhD year and ***I'm joining Nvidia*** afterwards!

ELEVATE is Open Source and ***publicly available***

I am ***available for collaborations now!*** Projects could start now and continue when I join NVIDIA

THANK YOU

for your attention!

ELEVATE is OpenSource: `github.com/elevate-lang`