University of Glasgow

Beta

ELEVATE *a language to write composable program optimisations*

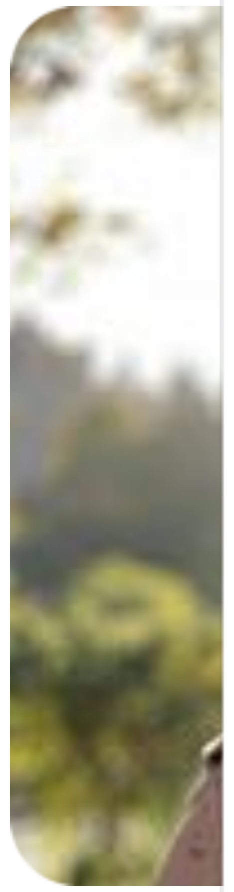
Michel Steuwer — *michel.steuwer@glasgow.ac.uk*

INSPIRING PEOPLE

Joined work with

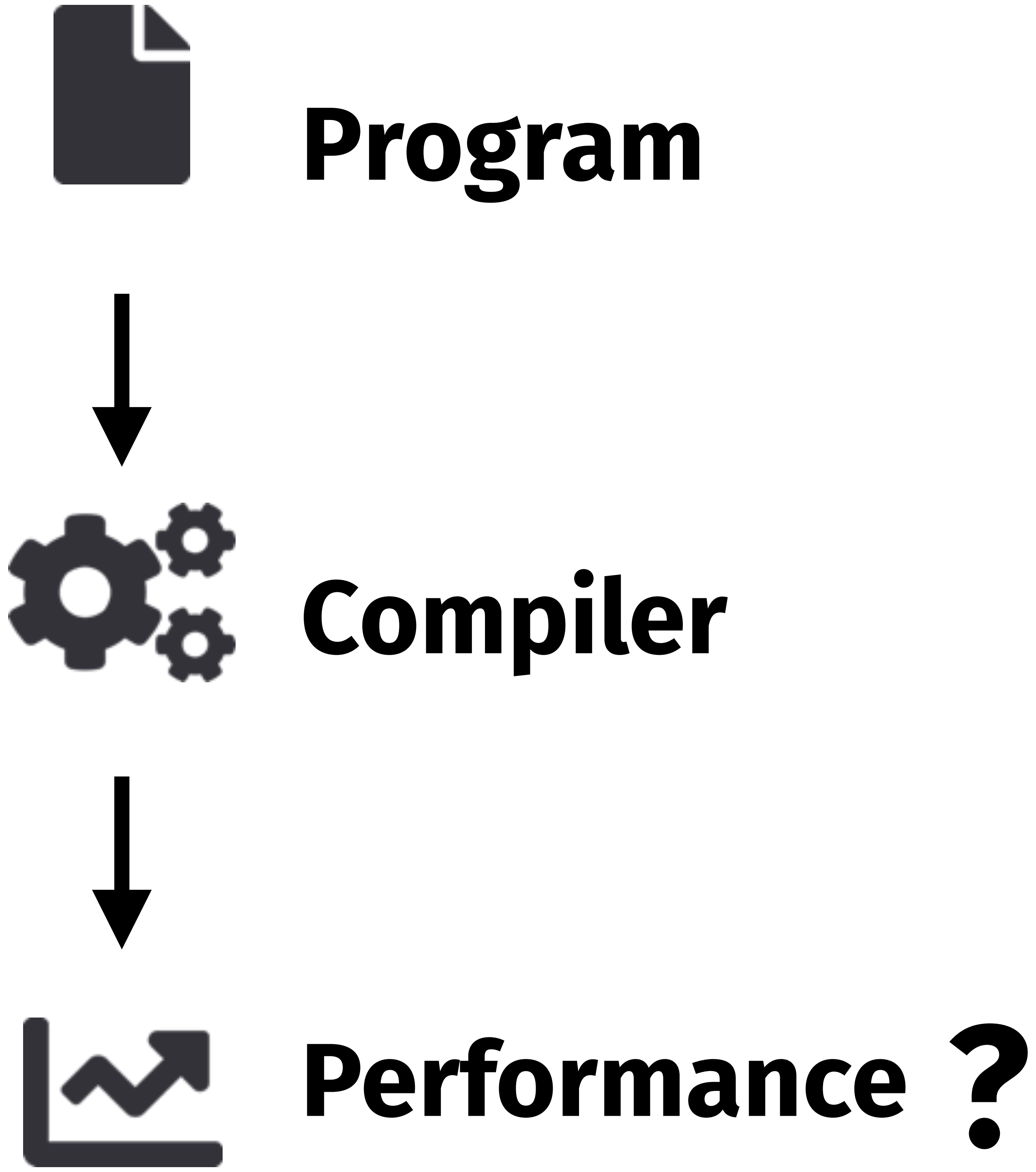
Bastian Hagedorn

https://bastianhagedorn.github.io



Currently at the Heidelberg Laureate Forum

How do we optimise programs today?



- Change the program manually
- Change compiler options

How do we optimise programs today?

Program

Compiler

Performance ?

```
for (i = 0; i < N; ++i) {
  for (j = 0; j < N; ++j){
    C[i][j] = 0;
    for (k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j]; } }
```

How do we optimise programs today?

Program

Compiler

Performance ?

```
for (i = 0; i < N; ++i) {
  for (j = 0; j < N; ++j){
    C[i][j] = 0;
    for (k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j]; } }
```

CPUs

- **Blocking / Tiling**
- **Exploit ILP**
- **Exploit locality**

```
for (ii = 0; ii < N; ii += ib) {
  for (kk = 0; kk < N; kk += kb) {
    for (j=0; j < N; j += 2) {
      for(i = ii; i < ii + ib; i += 2 ) {
        if (kk == 0)
          acc00 = acc01 = acc10 = acc11 = 0;
        else {
          acc00 = C[i + 0][j + 0];
          acc01 = C[i + 0][j + 1];
          acc10 = C[i + 1][j + 0];
          acc11 = C[i + 1][j + 1]; }
        for (k = kk; k < kk + kb; k++) {
          acc00 += A[k][j + 0] * B[i + 0][k];
          acc01 += A[k][j + 1] * B[i + 0][k];
          acc10 += A[k][j + 0] * B[i + 1][k];
          acc11 += A[k][j + 1] * B[i + 1][k];
        }
        C[i + 0][j + 0] = acc00;
        C[i + 0][j + 1] = acc01;
        C[i + 1][j + 0] = acc10;
        C[i + 1][j + 1] = acc11; } } } }
```

How do we optimise programs today?

Program

Compiler

Performance ?

```
for (i = 0; i < N; ++i) {
  for (j = 0; j < N; ++j){
    C[i][j] = 0;
    for (k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j]; } }
```

GPUs

```
kernel mm_amd_opt(global float * A, B, C,
                  int K, M, N) {
local float tileA[512]; tileB[512];

private float acc_0;       ...; acc_31;
private float blockOfB_0; ...; blockOfB_3;
private float blockOfA_0; ...; blockOfA_7;

int lid0 = local_id(0); lid1 = local_id(1);
int wid0 = group_id(0); wid1 = group_id(1);

for (int w1=wid1; w1<M/64; w1+=num_grps(1)) {
 for (int w0=wid0; w0<N/64; w0+=num_grps(0)) {

  acc_0 = 0.0f; ...; acc_31 = 0.0f;
  for (int i=0; i<K/8; i++) {
   vstore4(vload4(lid1*M/4+2*i*M+16*w1+lid0,A)
           ,16*lid1+lid0, tileA);
   vstore4(vload4(lid1*N/4+2*i*N+16*w0+lid0,B)
           ,16*lid1+lid0, tileB);
   barrier(...);

   for (int j = 0; j<8; j++) {
    blockOfA_0 = tileA[0+64*j+lid1*8];
    ... 6 more statements
    blockOfA_7 = tileA[7+64*j+lid1*8];
    blockOfB_0 = tileB[0 +64*j+lid0];
    ... 2 more statements
    blockOfB_3 = tileB[48+64*j+lid0];

    acc_0  += blockOfA_0 * blockOfB_0;
    acc_1  += blockOfA_0 * blockOfB_1;
    acc_2  += blockOfA_0 * blockOfB_2;
    acc_3  += blockOfA_0 * blockOfB_3;
    ... 24 more statements
    acc_28 += blockOfA_7 * blockOfB_0;
    acc_29 += blockOfA_7 * blockOfB_1;
    acc_30 += blockOfA_7 * blockOfB_2;
    acc_31 += blockOfA_7 * blockOfB_3;
   }
   barrier(...);

  }

  C[ 0+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_0;
  C[16+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_1;
  C[32+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_2;
  C[48+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_3;
  ... 24 more statements
  C[ 0+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_28;
  C[16+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_29;
  C[32+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_30;
  C[48+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_31;
} } }
```

AMD

```
// kernel __attribute((reqd_work_group_size(32, 8, 1)))
void KERNEL(const global float *restrict A, const global float *restrict B,
            const global float *restrict C, float alpha, float beta,
            global float *out, int K, int M, int N) {
  local float l_tmp_1[512];
  local float l_tmp_2[1024];
  float acc_1_1_425 = 0.0f;
  //  ... 31 more
  float p_tmp_1_1_457;
  //  ... 107 more
  int wg_id_1 = get_group_id(1);
  int wg_id_0 = get_group_id(0);
  for (int i = 0; i < (K / 8); i = (1 + i)) {
    int l_id_1 = get_local_id(1);
    for (int l_id_0 = get_local_id(0); (l_id_0 < 64); l_id_0 = (32 + l_id_0)) {
      l_tmp_1[(l_id_0 + (64 * l_id_1))] =
          (A[(l_id_0 + (8 * M * i) + (64 * wg_id_1) + (M * l_id_1))]);
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int l_id_0 = get_local_id(0); (l_id_0 < 128); l_id_0 = (32 + l_id_0)) {
      l_tmp_2[(l_id_0 + (128 * l_id_1))] =
          (B[(l_id_0 + (8 * N * i) + (128 * wg_id_0) + (N * l_id_1))]);
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int j = 0; (j < 8); j = (1 + j)) {
      p_tmp_1_1_457 = (l_tmp_1[(0 + (8 * get_local_id(1)) + (64 * j))]);
      p_tmp_1_2_458 = (l_tmp_1[(1 + (8 * get_local_id(1)) + (64 * j))]);
      //  ... 6 more
      p_tmp_2_1_465 = (l_tmp_2[(0 +  (128 * j) + get_local_id(0))]);
      p_tmp_2_2_466 = (l_tmp_2[(32 + (128 * j) + get_local_id(0))]);
      //  ... 2 more
      p_tmp_3_1_469 = p_tmp_1_1_457 * p_tmp_2_1_465;
      acc_1_1_425 = acc_1_1_425 + p_tmp_3_1_469;
      //  ... 31 more
    }
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
  }
  p_tmp_4_1_501 = acc_1_1_425 * alpha;
  p_tmp_5_1_533 = C[((64 * N * wg_id_1) + (8 * N * get_local_id(1)) +
                        (128 * wg_id_0) + get_local_id(0))] *
                     beta;
  out[((64 * N * wg_id_1) + (8 * N * get_local_id(1)) + (128 * wg_id_0) +
       get_local_id(0))] = p_tmp_4_1_501 + p_tmp_5_1_533;
  //  ... 31 more
}
```

Nvidia

```
kernel void mm(global float4* const A,
               global float4* const B,
               global float2* C, uint n) {
  uint i = get_global_id(0);
  uint j = get_global_id(1);
  uint nv4 = n >> 2;
  float4 ab = (float4)(0.0f);
  for (uint k = 0; k < nv4; ++k) {
    float4 a0 = A[ 2*i    *nv4+k];
    float4 a1 = A[(2*i+1)*nv4+k];
    float4 b0 = B[ 2*j    *nv4+k];
    float4 b1 = B[(2*j+1)*nv4+k];
    ab += (float4)(dot(a0, b0), dot(a0, b1),
                   dot(a1, b0), dot(a1, b1)); }
  uint ix = 2*i*(n>>1) + j;
  C[ix]            = ab.s01;
  C[ix + (n>>1)] = ab.s23; }
```

ARM

How do we optimise programs today?

Program

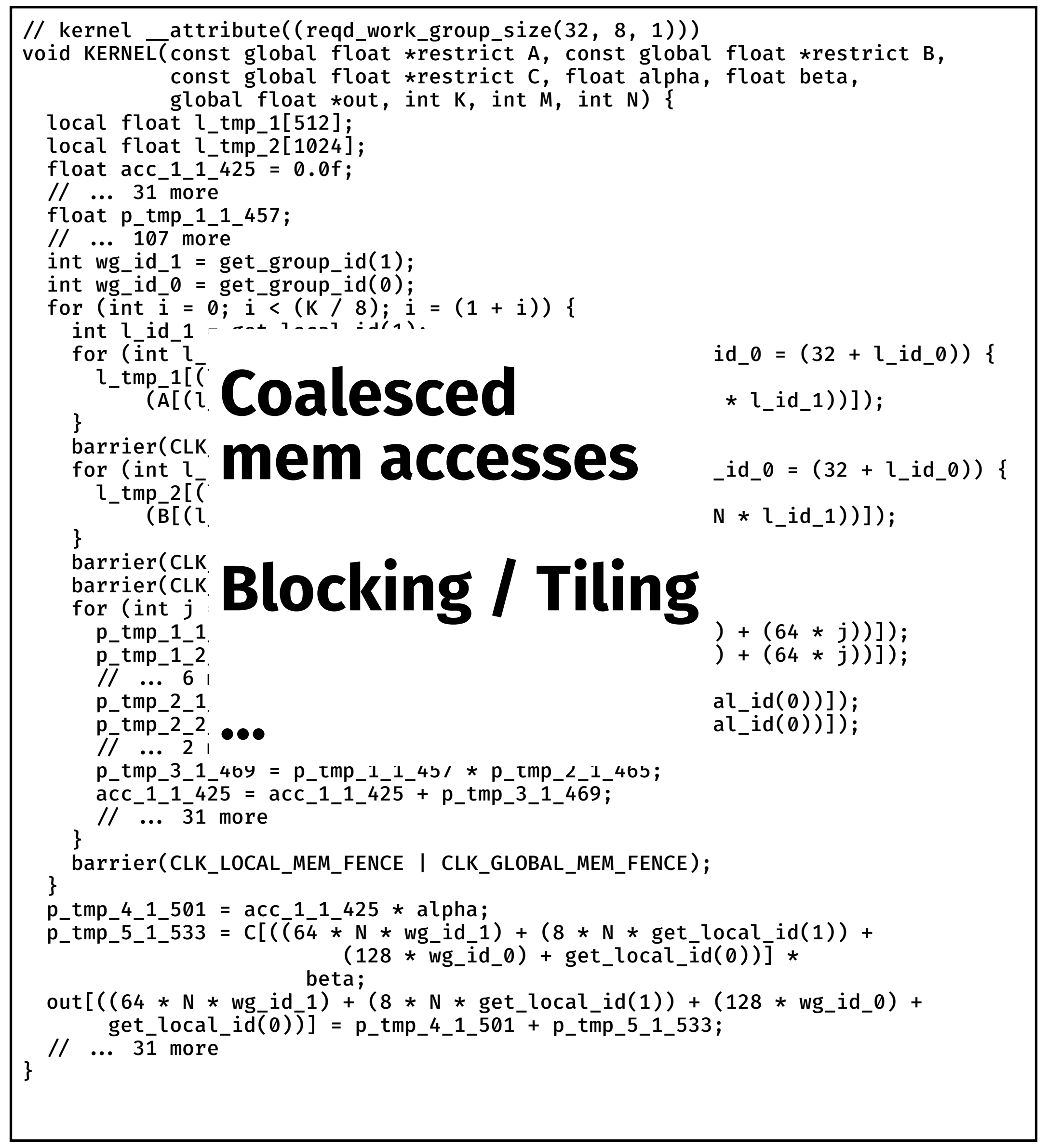
Compiler

Performance ?

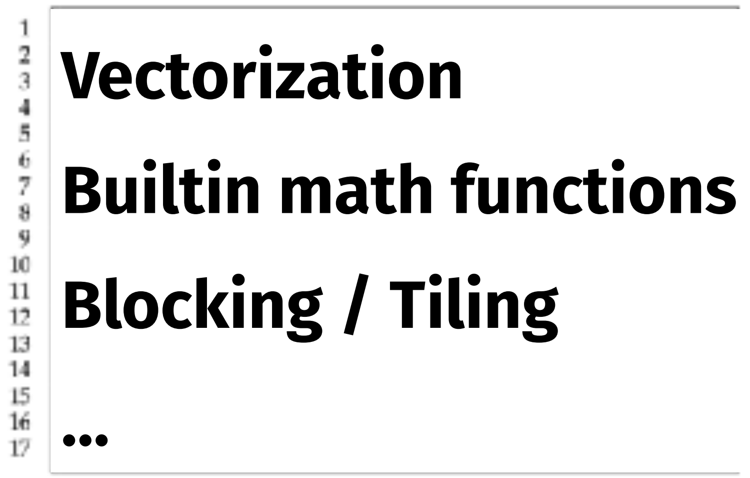
```
for (i = 0; i < N; ++i) {
  for (j = 0; j < N; ++j){
    C[i][j] = 0;
    for (k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j]; } }
```

GPUs

AMD



Nvidia



ARM

How do we optimise programs today?

Program

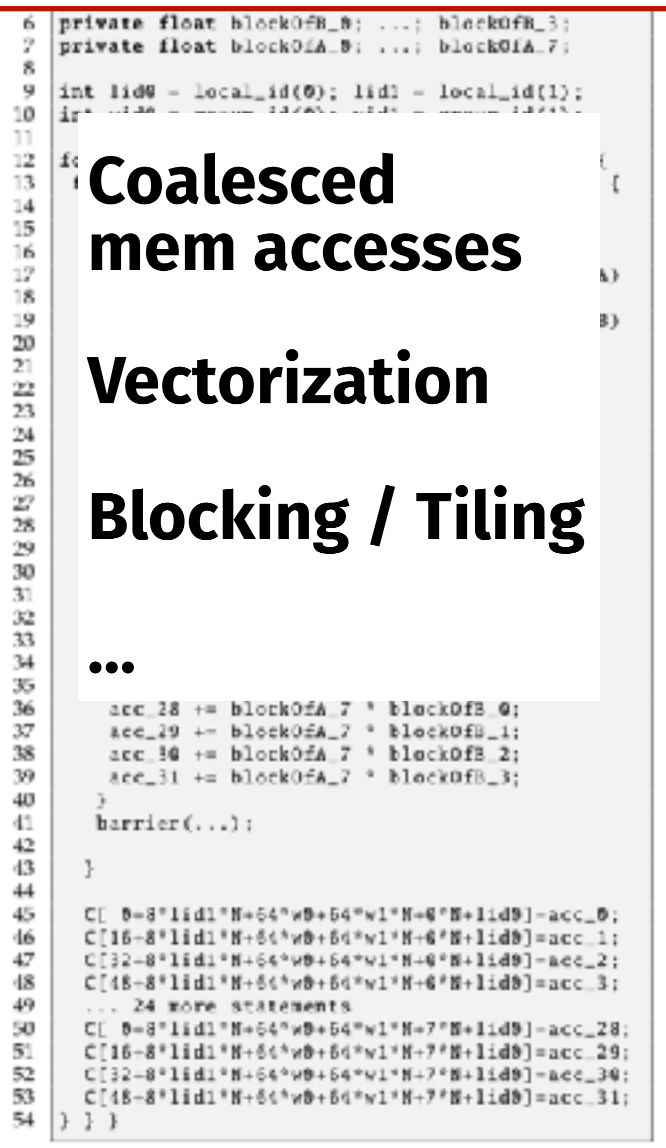
Compiler

Performance ?

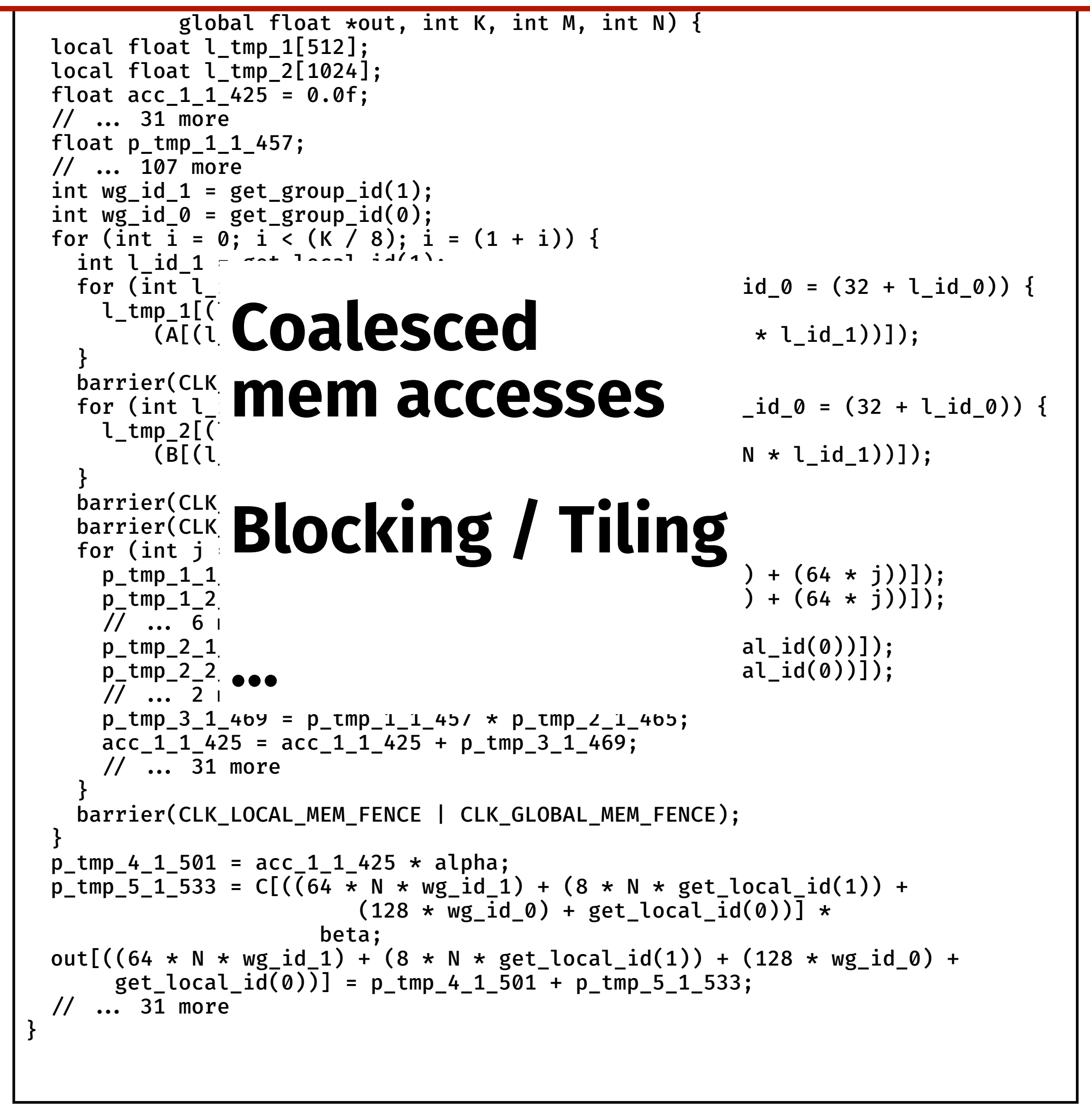
```
for (i = 0; i < N; ++i) {
  for (j = 0; j < N; ++j){
    C[i][j] = 0;
    for (k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j]; } }
```

GPUs

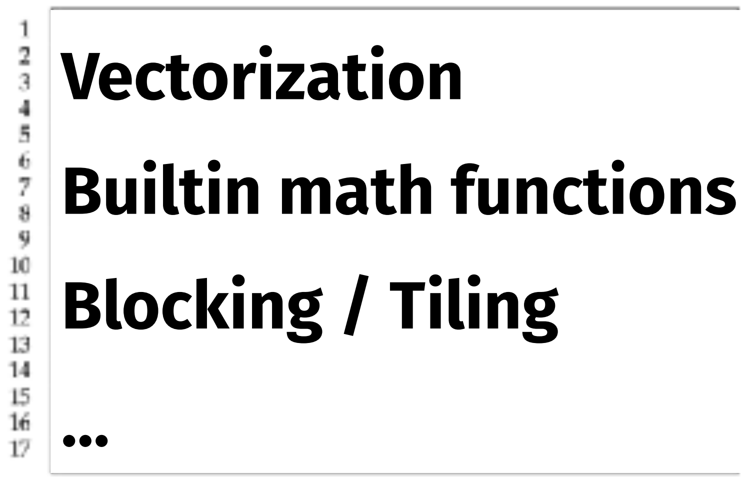
Unsustainable to re-optimise for every new architecture ⇒ No performance portability



AMD

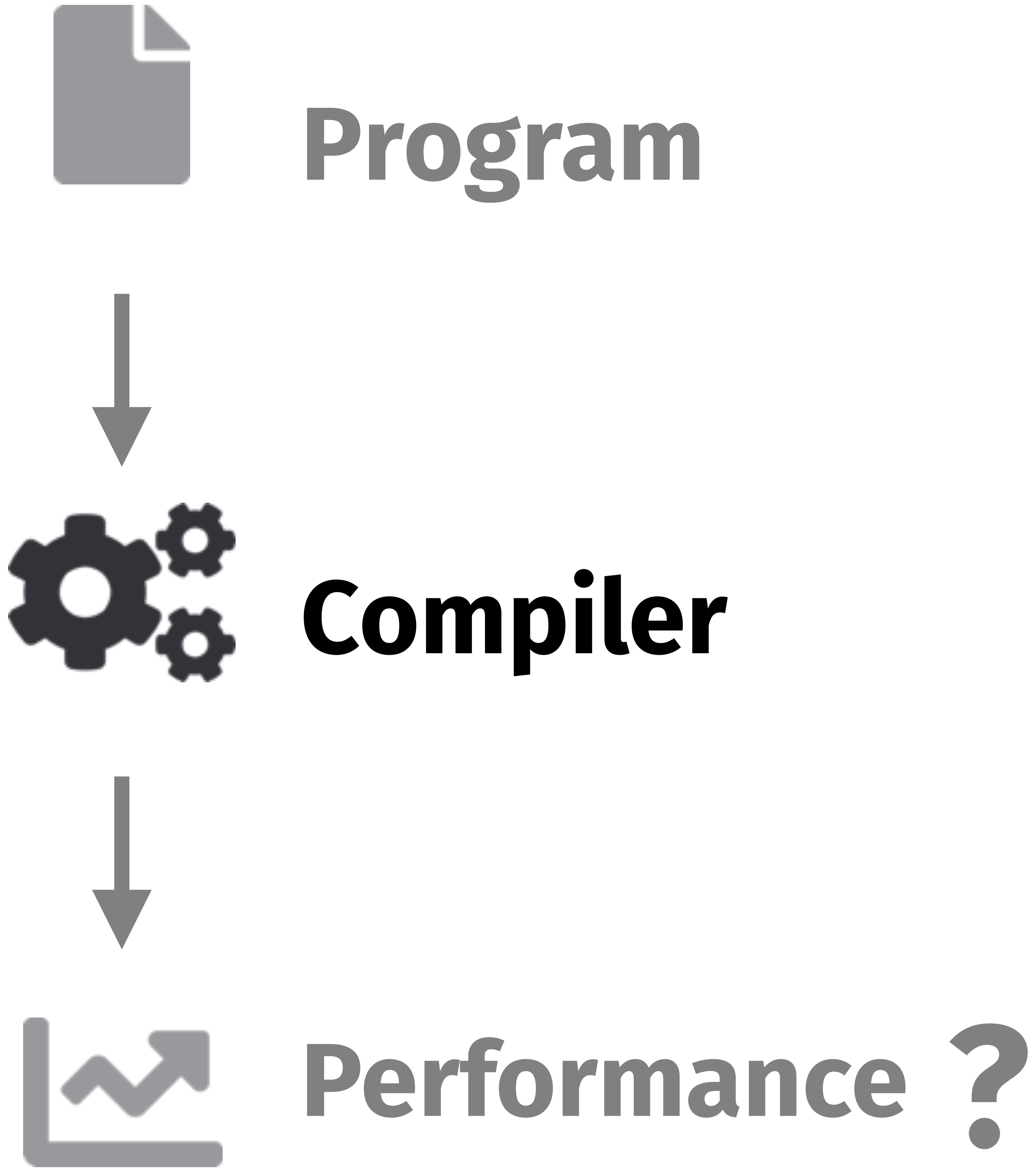


Nvidia

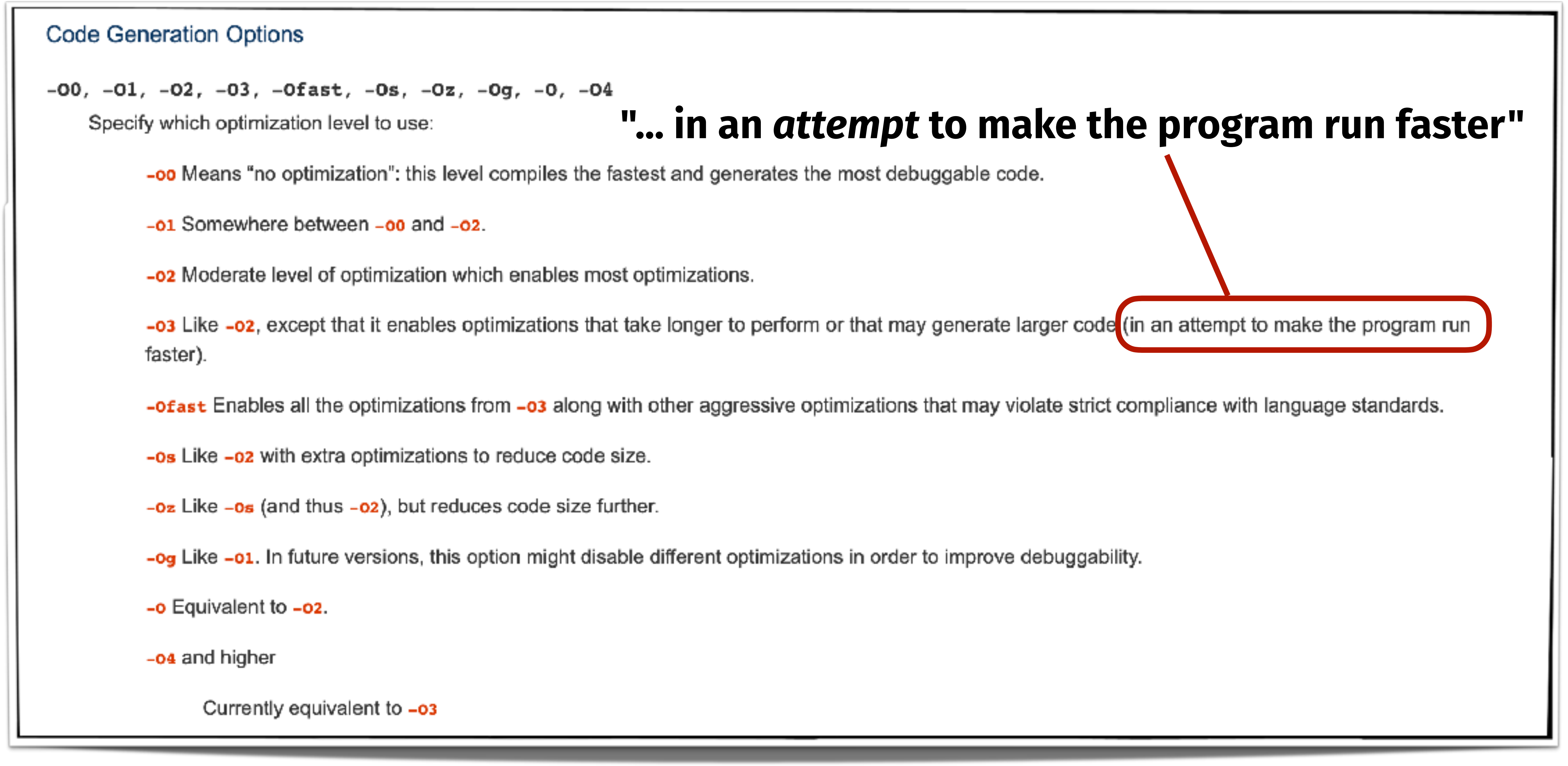


ARM

How do we optimise programs today?



From the LLVM manual:



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.

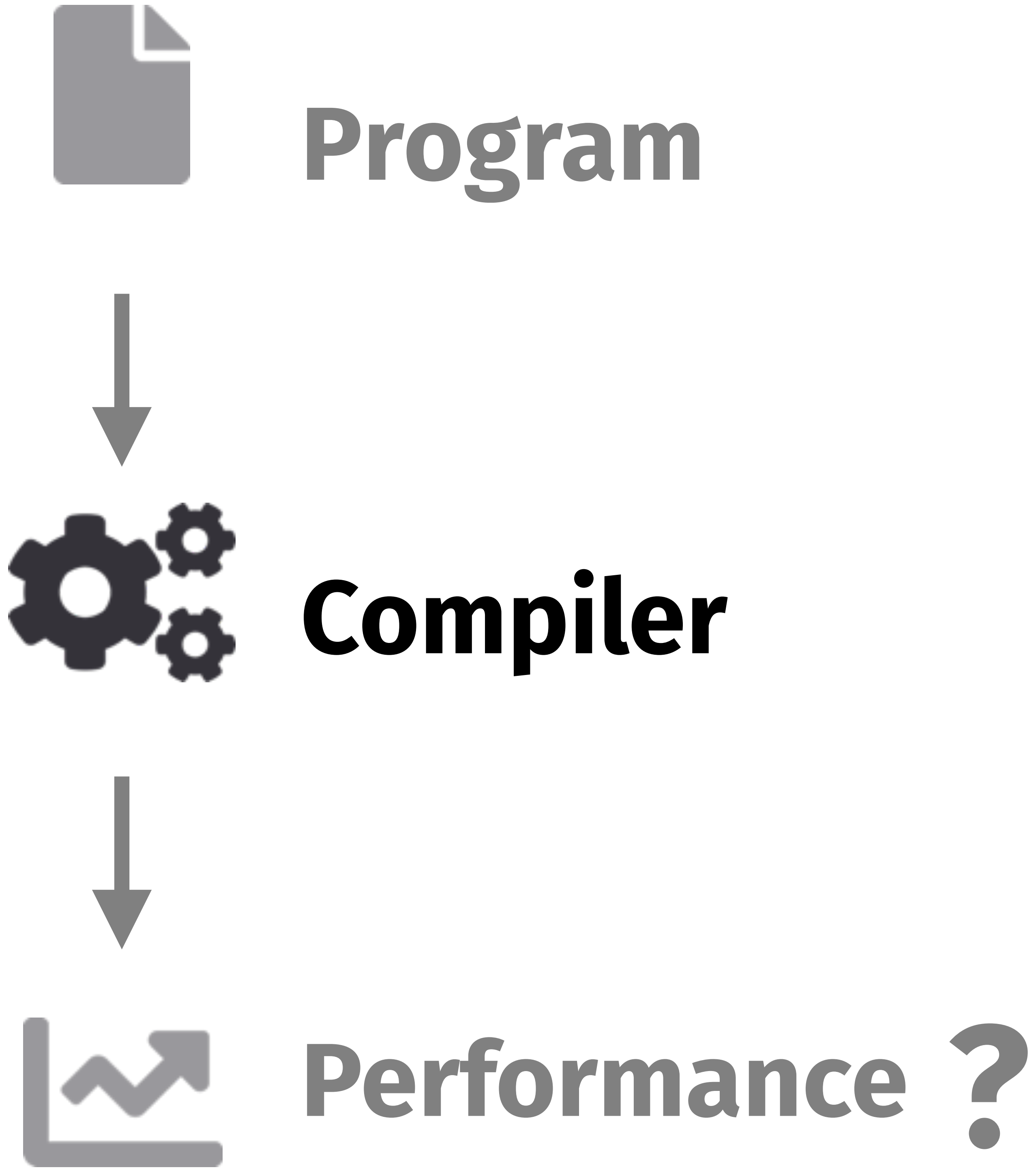
`-O` Equivalent to `-O2`.

`-O4` and higher

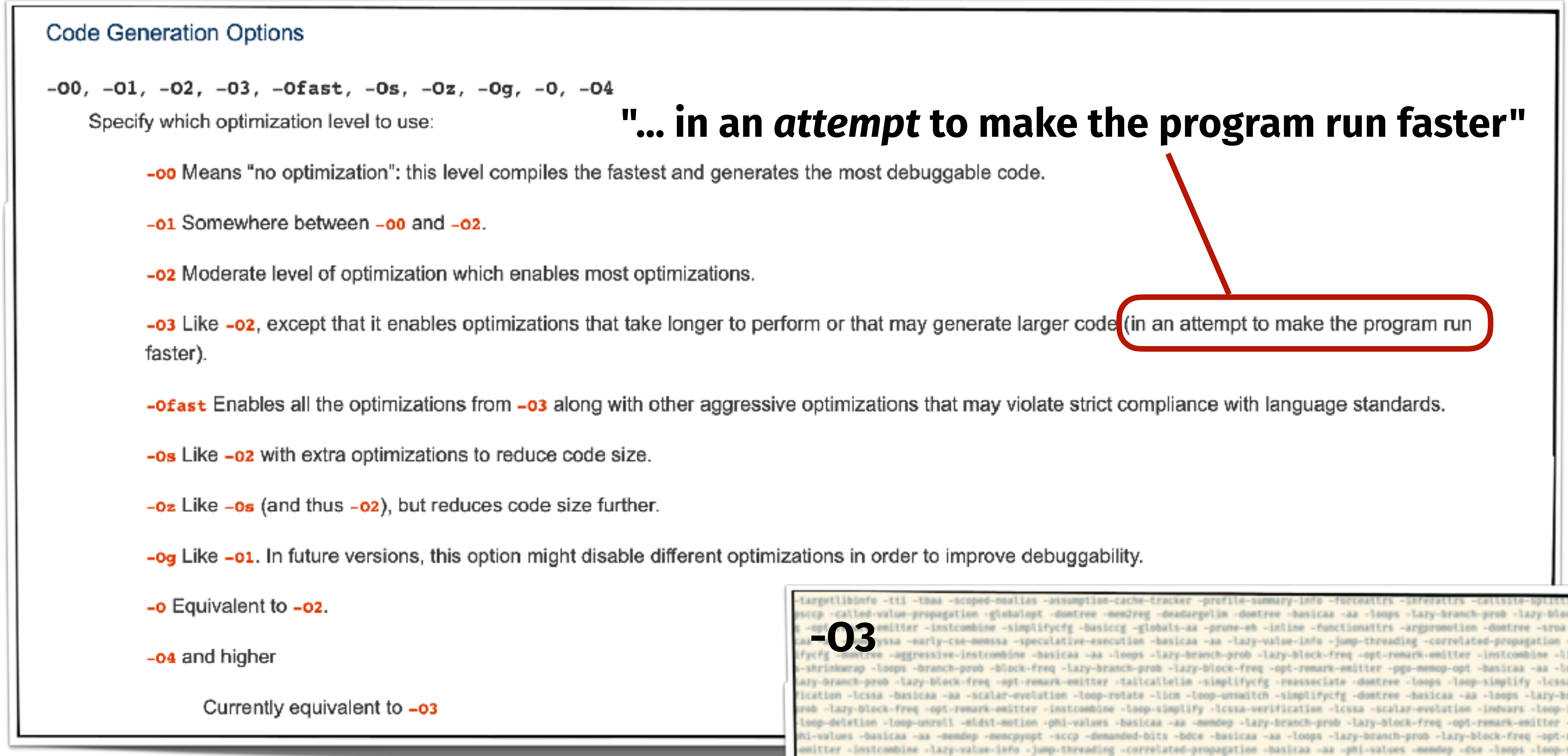
Currently equivalent to `-O3`

"... in an *attempt* to make the program run faster"

How do we optimise programs today?



From the LLVM manual:



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.

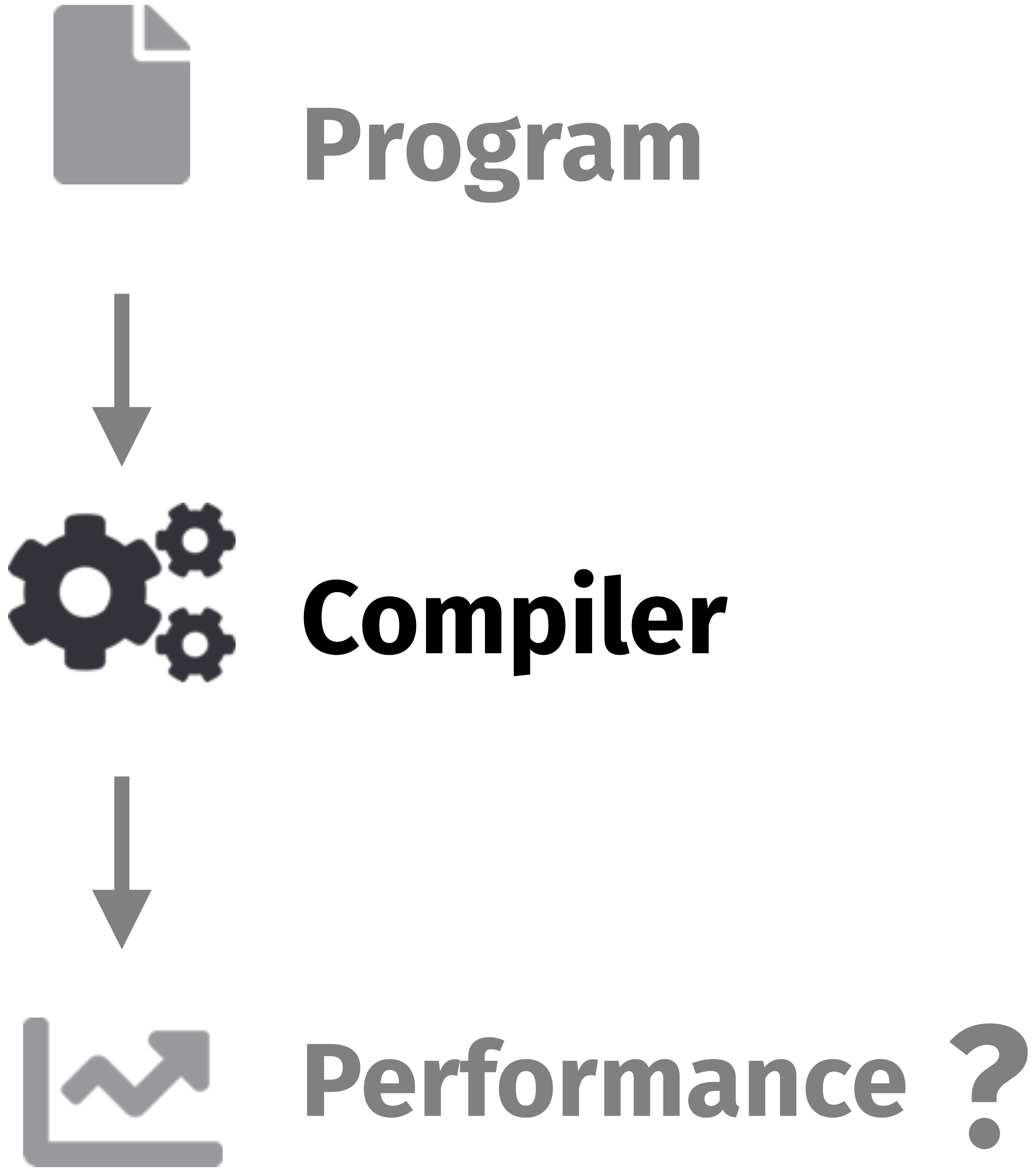
`-O` Equivalent to `-O2`.

`-O4` and higher

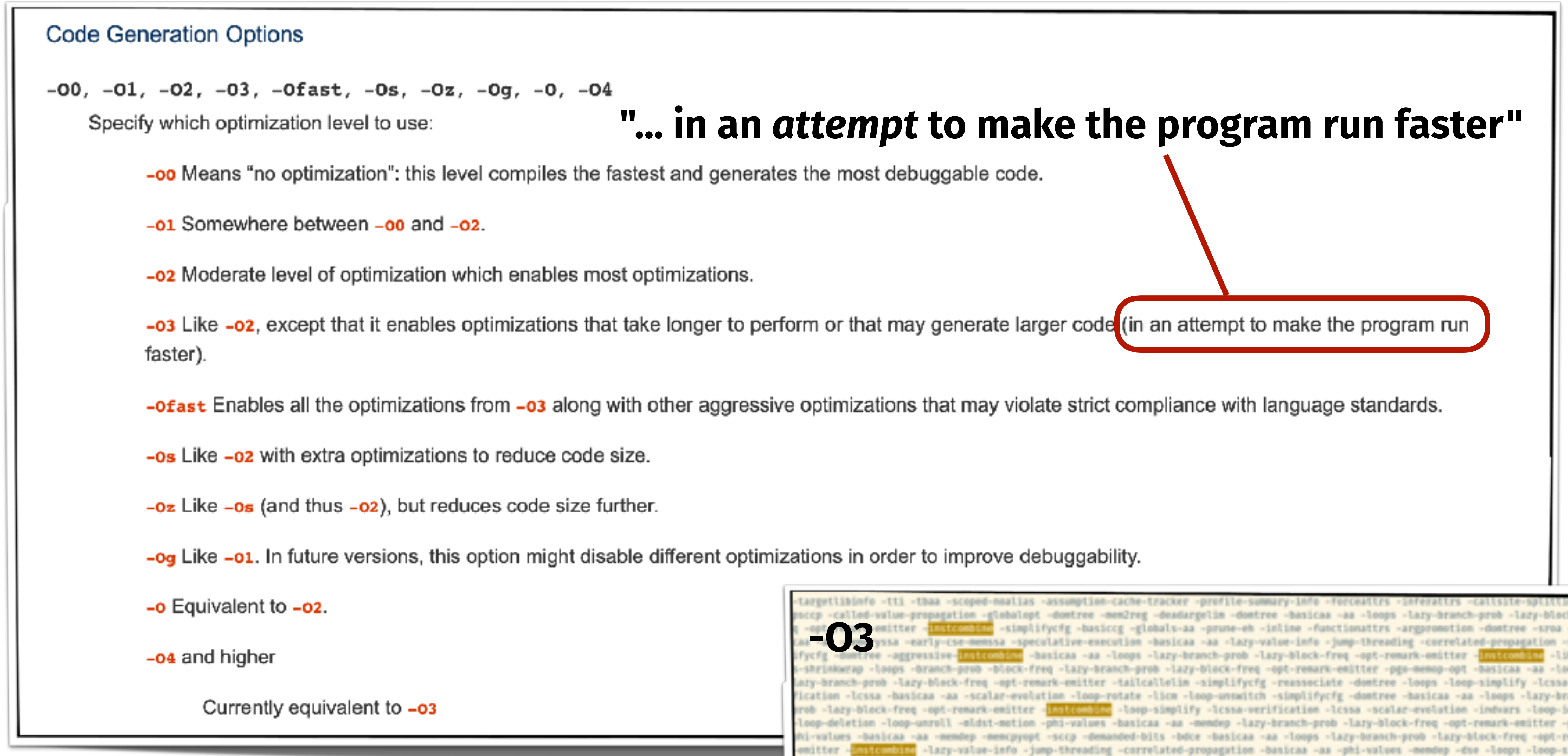
Currently equivalent to `-O3`

"... in an *attempt* to make the program run faster"

How do we optimise programs today?



From the LLVM manual:



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.

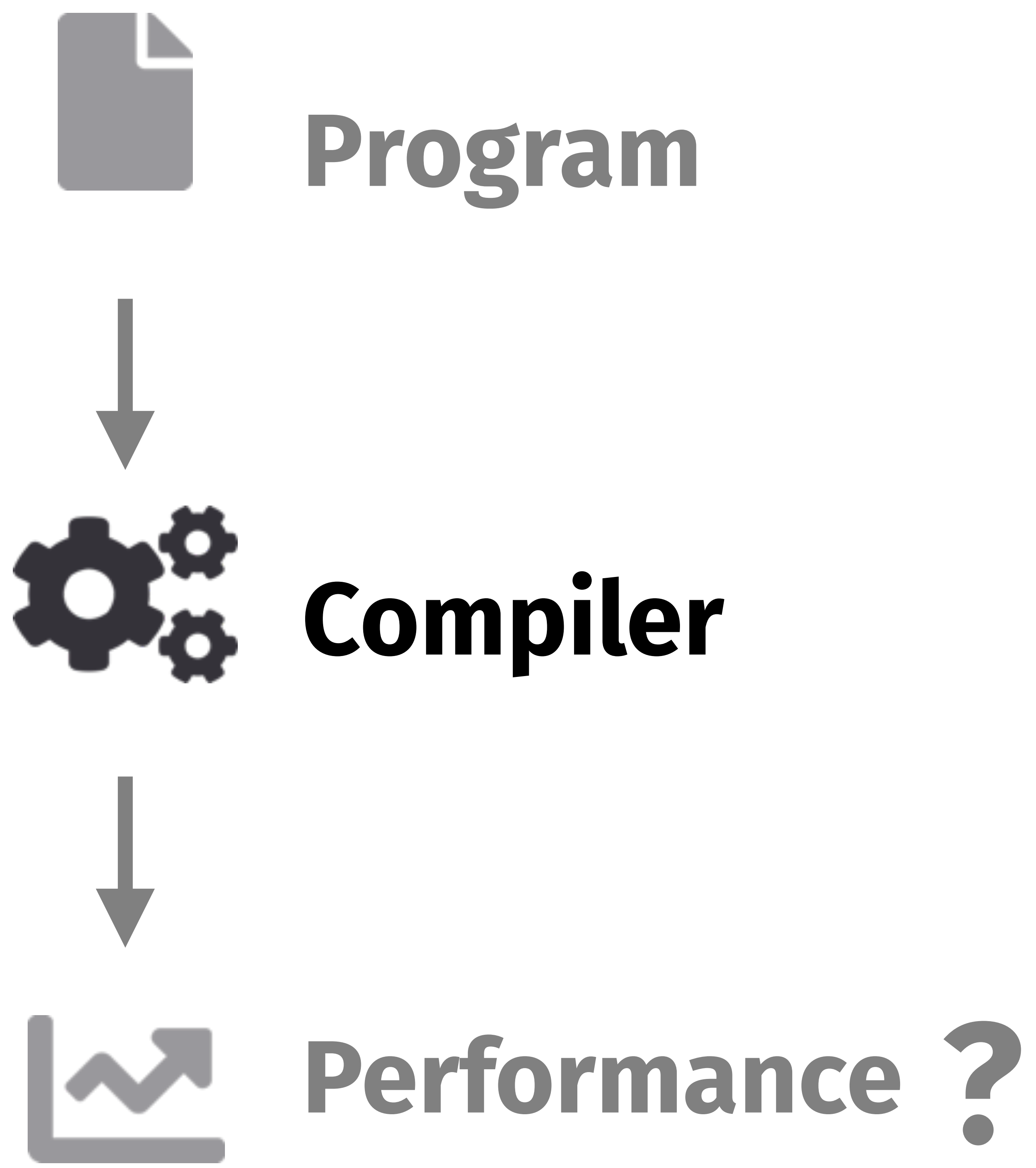
`-O` Equivalent to `-O2`.

`-O4` and higher

Currently equivalent to `-O3`

"... in an *attempt* to make the program run faster"

How do we optimise programs today?



From the LLVM manual:

Code Generation Options

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

Specify which optimization level to use:

"... in an *attempt* to make the program run faster"

- `-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`

Intel compiler:

`-opt-matmul`

Options -opt-matmul and /Qopt-matmul tell the compiler to **identify matrix multiplication loop nests (if any) and replace them with a matmul library call** for improved performance. The resulting executable **may get additional performance gain on Intel® microprocessors** than on non-Intel microprocessors.

How do we optimise programs today?

Program

Compiler

Performance ?

From the LLVM manual:

Code Generation Options

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

Specify which optimization level to use:

"... in an *attempt* to make the program run faster"

- `-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).

Impossible to understand what is going on in the compiler ⇒ Hard to control optimisations

th 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`

-O3

Intel compiler:

`-opt-matmul`

Options -opt-matmul and /Qopt-matmul tell the compiler to **identify matrix multiplication loop nests (if any) and replace them with a matmul library call** for improved performance. The resulting executable **may get additional performance gain on Intel® microprocessors** than on non-Intel microprocessors.

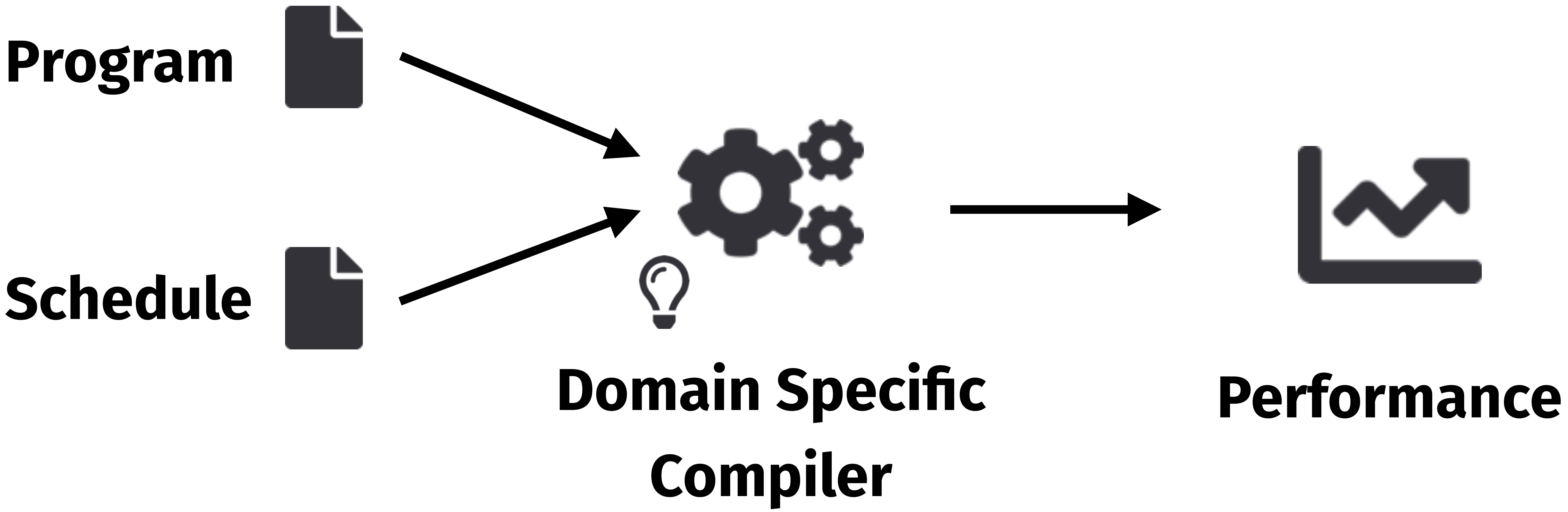
Separate Program from Optimisations

Halide

[PLDI'13] [SIGGRAPH'12, 16, 18]

Tiramisu Compiler

used by: tvm



Separation in Program and Schedule allows for portable performance

Halide - Program vs. Schedule

Program

Domain Specific Language embedded in C++

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

Schedule

C++ API for selecting optimisation options

```
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, TailStrategy::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, TailStrategy::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);

set_alignment_and_bounds(A, size);
set_alignment_and_bounds(B, size);
set_alignment_and_bounds(out, size);
```

Schedule much harder to write and reason about then functional program!

Halide - Program vs. Schedule

Program   **Schedule**

Domain

emb

```
Func prod(“
RDom r(0, s
prod(x, y)
out(x, y) =
```

```
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)
```

```
...
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);
...
```

Fixed set of optimisations ⇒ lack of extensibility

What happens if the order of these are swapped?

⇒ unclear semantics ⇒ unclear how to automatically generate schedules

```
set_alignment_and_bounds(A, size);
set_alignment_and_bounds(B, size);
set_alignment_and_bounds(out, size);
```

Halide - Program vs. Schedule

Unintuitive semantics: Why are these lines repeated

Program

Domain Specific Language

```
Func p
RDom r
prod(x
out(x,
```

Schedule

C++ API for selecting

```
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)
```

```
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, TailStrategy::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, TailStrategy::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);
```

```
     .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);

set_alignment_and_bounds(A, size);
set_alignment_and_bounds(B, size);
set_alignment_and_bounds(out, size);
```

Unintuitive semantics: "`Update`: *Get a handle on an update step for the purposes of scheduling it*"

Halide - Program vs. Schedule

Program

Domain Specific Language embedded in C++

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

Schedule

C++ API for selecting optimisation options

```
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, TailStrategy::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, TailStrategy::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);

set_alignment_and_bounds(A, size);
set_alignment_and_bounds(B, size);
set_alignment_and_bounds(out, size);
```

Schedules are second class citizens.

We should write schedules in a proper programming language!

ELEVATE
A programming language for program optimizations

ELEVATE is a functional language that allows to compose individual *program transformations* into larger *optimisation strategies*.

ELEVATE programs are composed of (possibly recursive) functions:

```
def transform(p: Program): RewriteResult[Program] = implementation
```

Program transformations are expressed as functions with a particular type:

```
Program → RewriteResult[Program]
```

Optimisation strategies are composed functions with the same type

A **`RewriteResult`** can either be **`Success`** or **`Failure`**

A successfully applied transformation contains the transformed program.
A unsuccessfully applied transformation is indicated as failure.

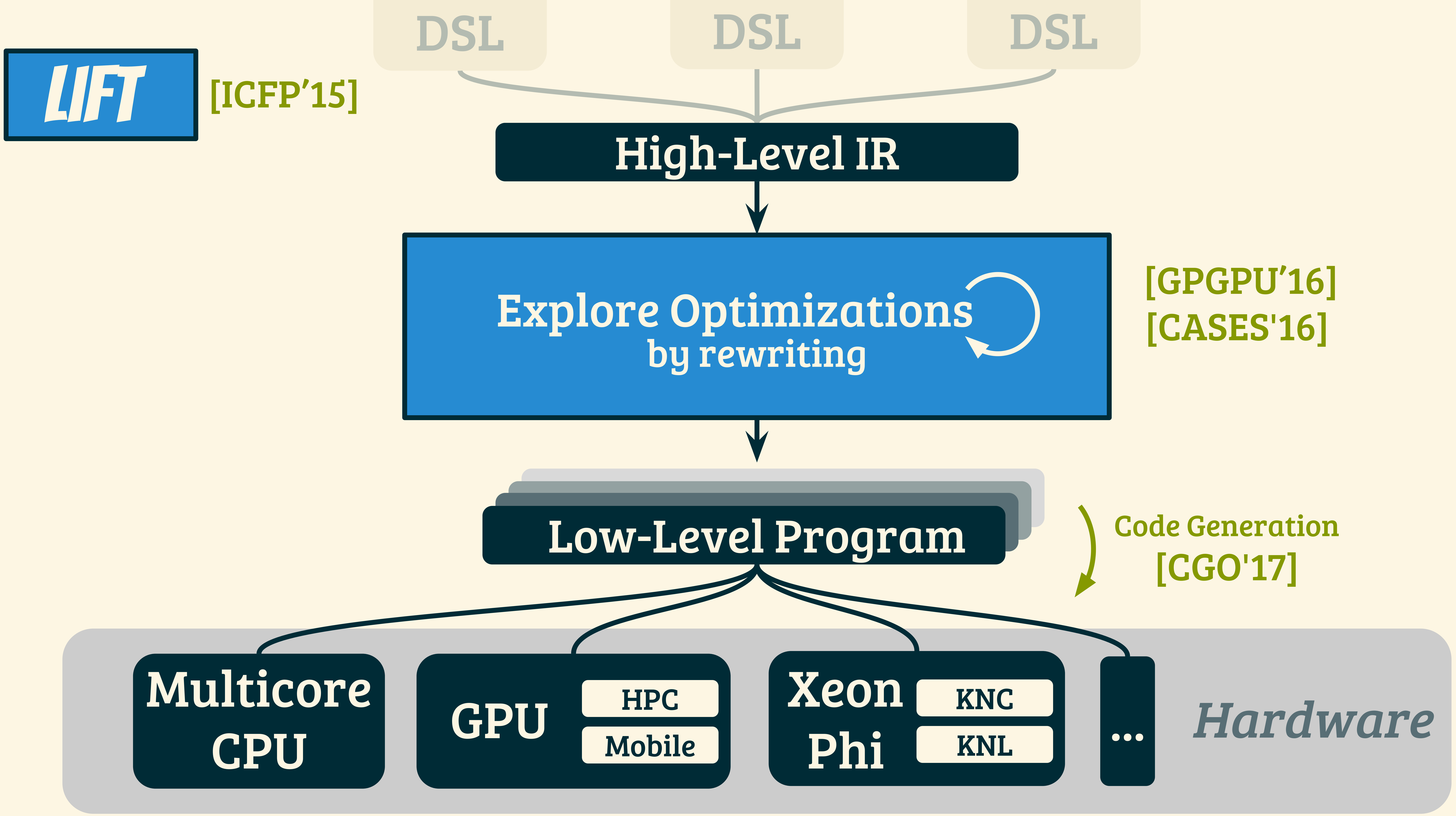
ELEVATE for optimising LIFT programs

ELEVATE can be used to optimise programs written in different languages

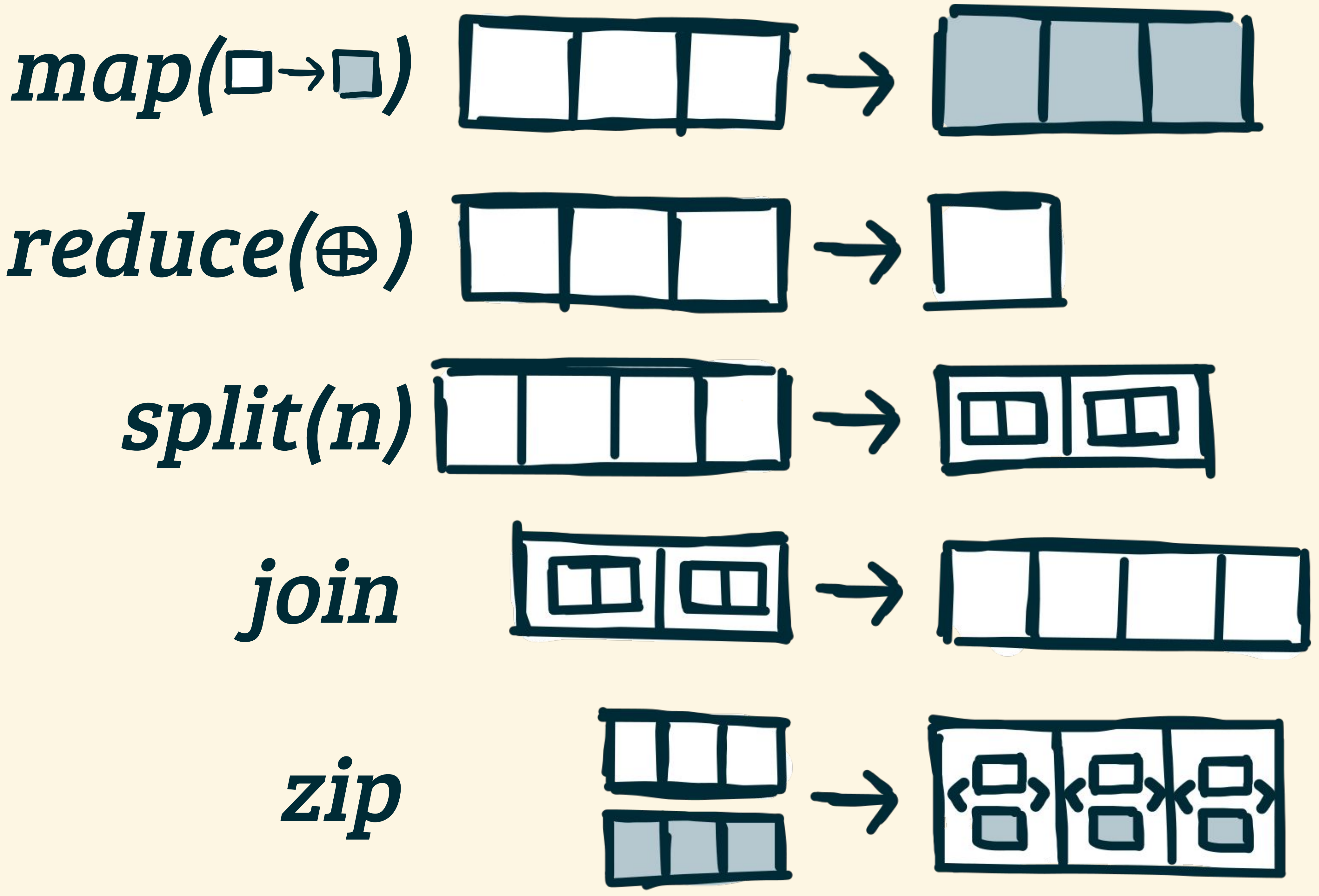
In this talk I focus on programs written in two functional languages:
- the data parallel LIFT programming language
- the **FSmooth** language used for automatic differentiation

We intend to use ELEVATE for additional high-level languages like TensorFlow

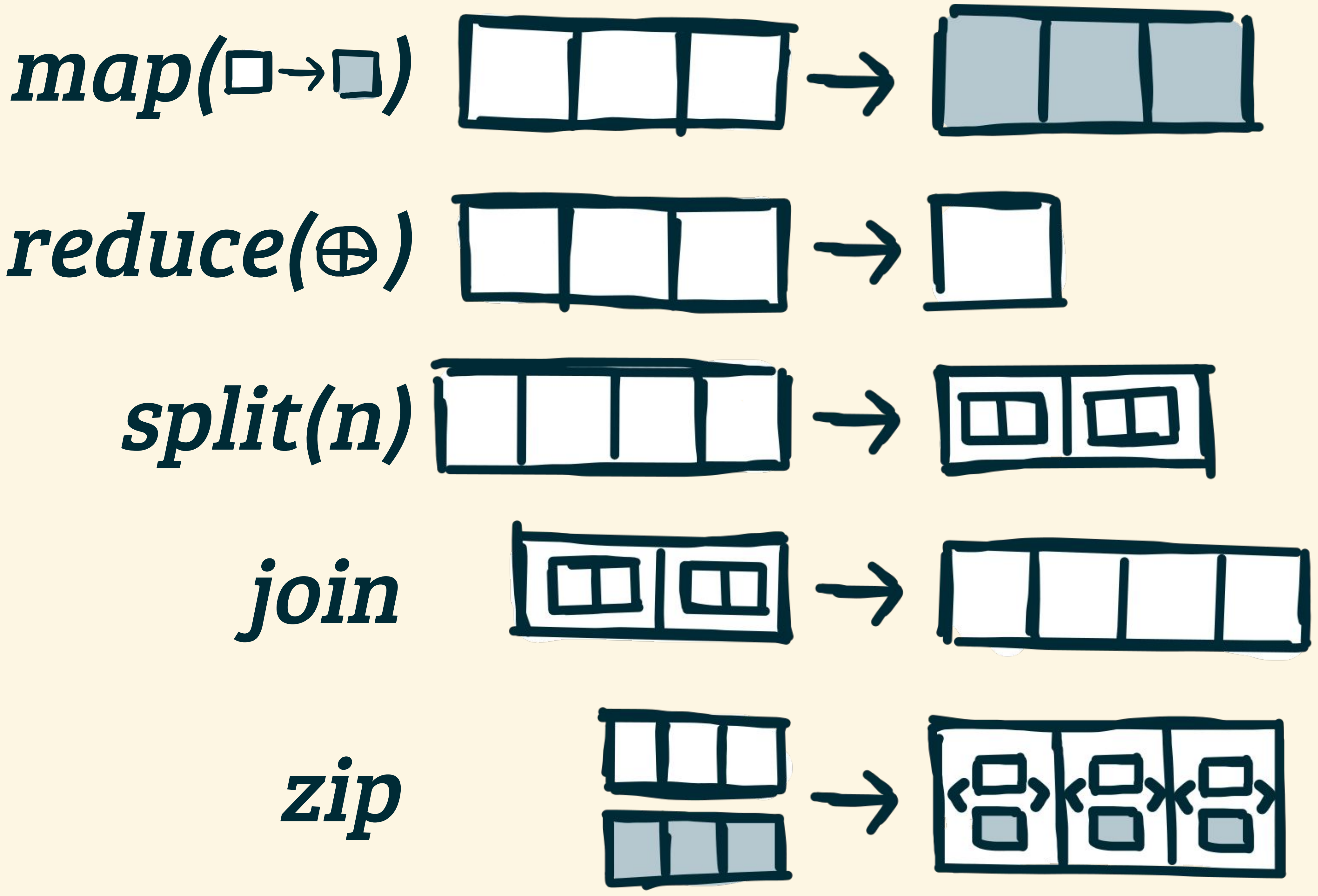
LIFT: More info at http://www.lift-project.org and papers at: [ICFP 2015] [CASES 2016] [CGO 2017 & 2018]
FSmmoth: [ICFP 2019]


DSL
DSL
DSL
LIFT
[ICFP'15]
High-Level IR
Explore Optimizations
by rewriting
[GPGPU'16]
[CASES'16]
Low-Level Program
Code Generation
[CGO'17]
Multicore CPU
GPU
HPC
Mobile
Xeon Phi
KNC
KNL
...
Hardware

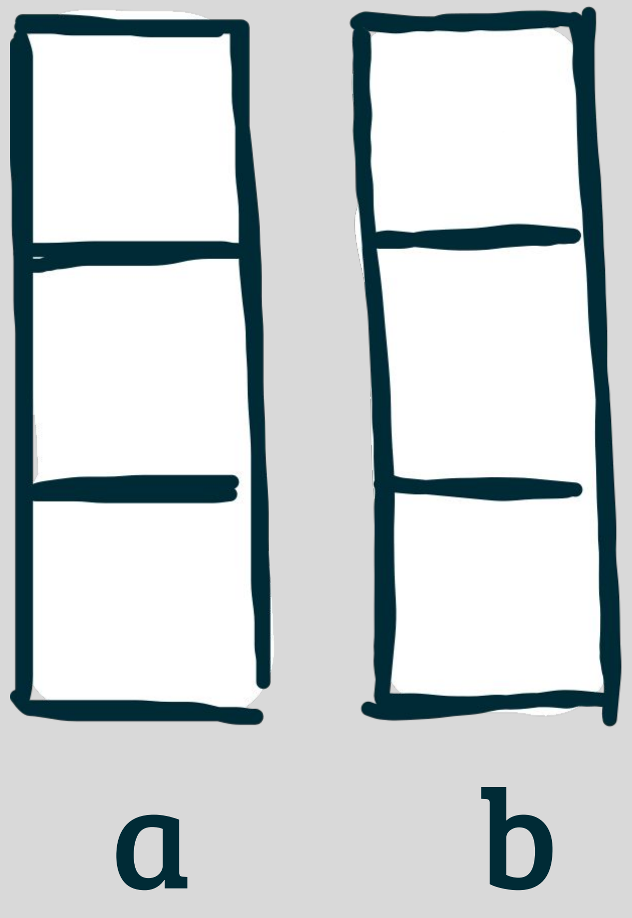
LIFT'S HIGH-LEVEL PRIMITIVES



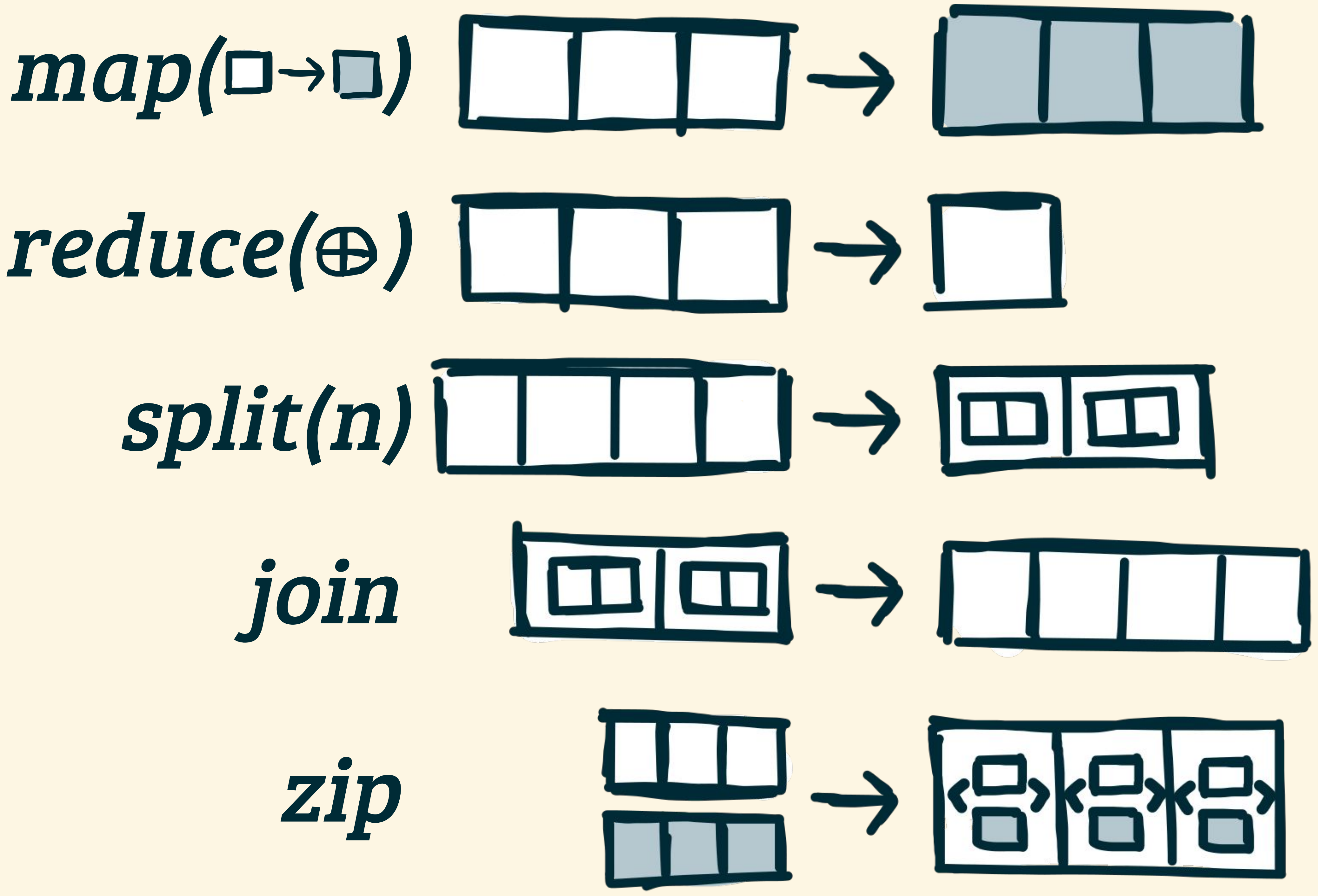
LIFT'S HIGH-LEVEL PRIMITIVES



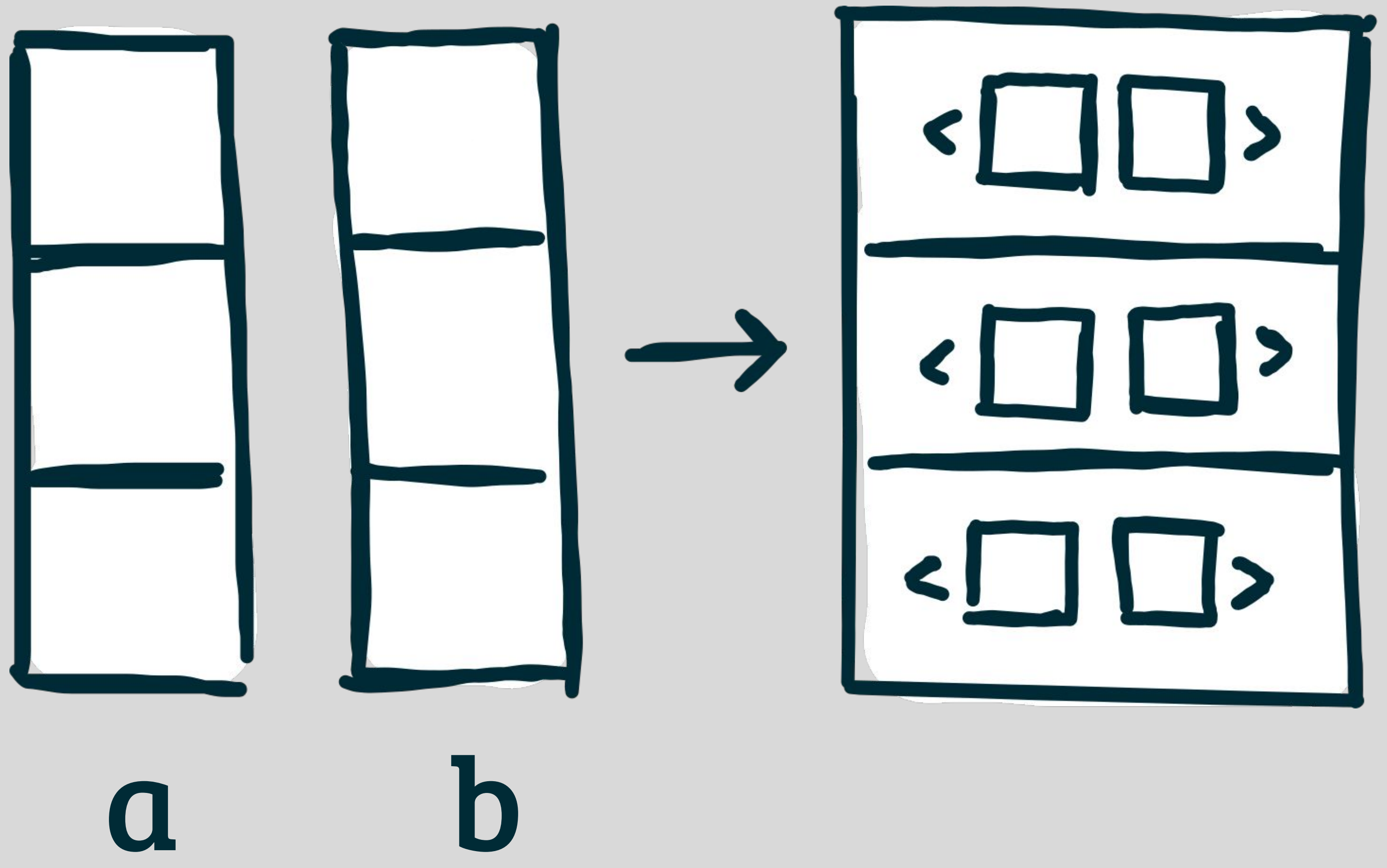
dotproduct.lift



LIFT'S HIGH-LEVEL PRIMITIVES

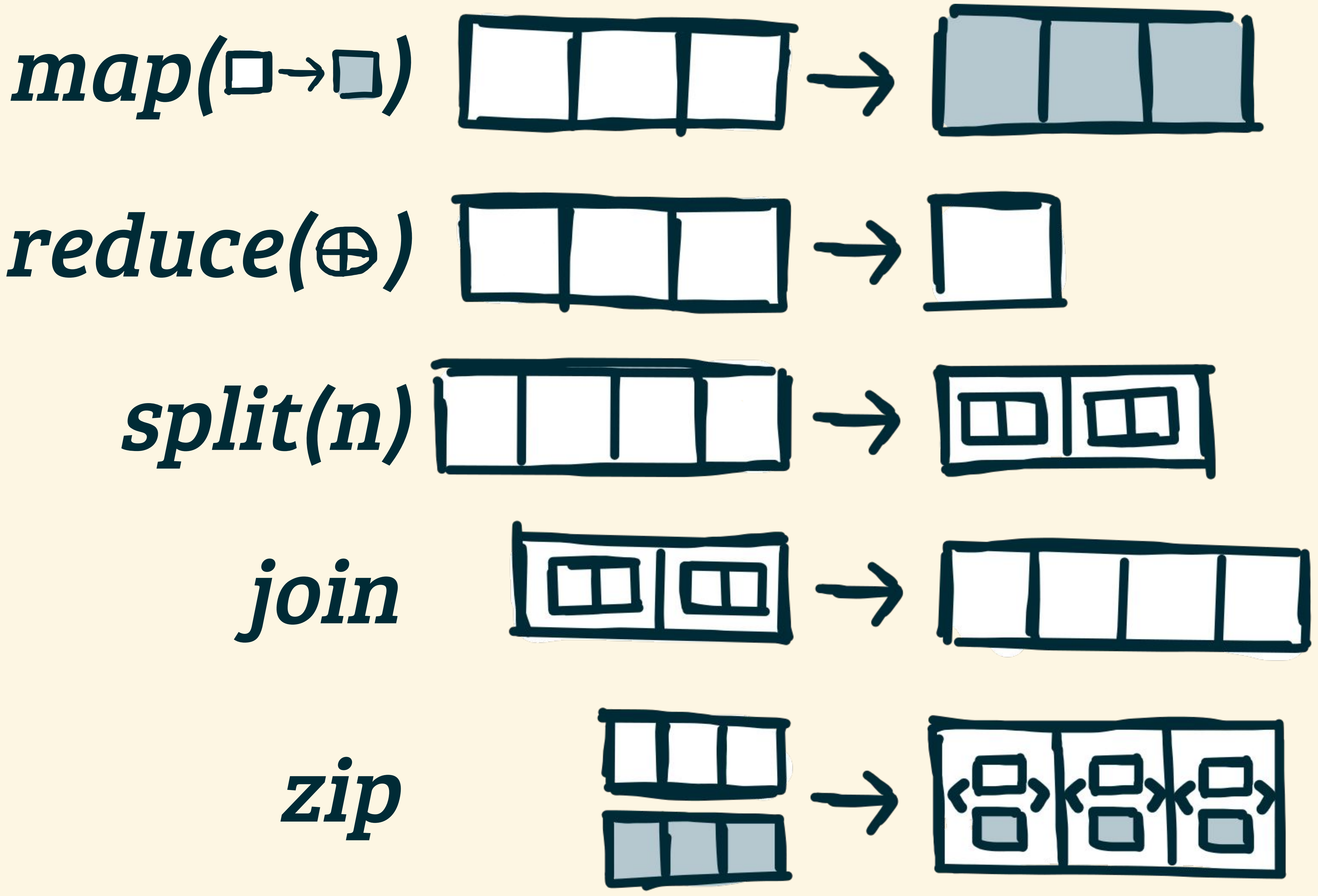


dotproduct.lift

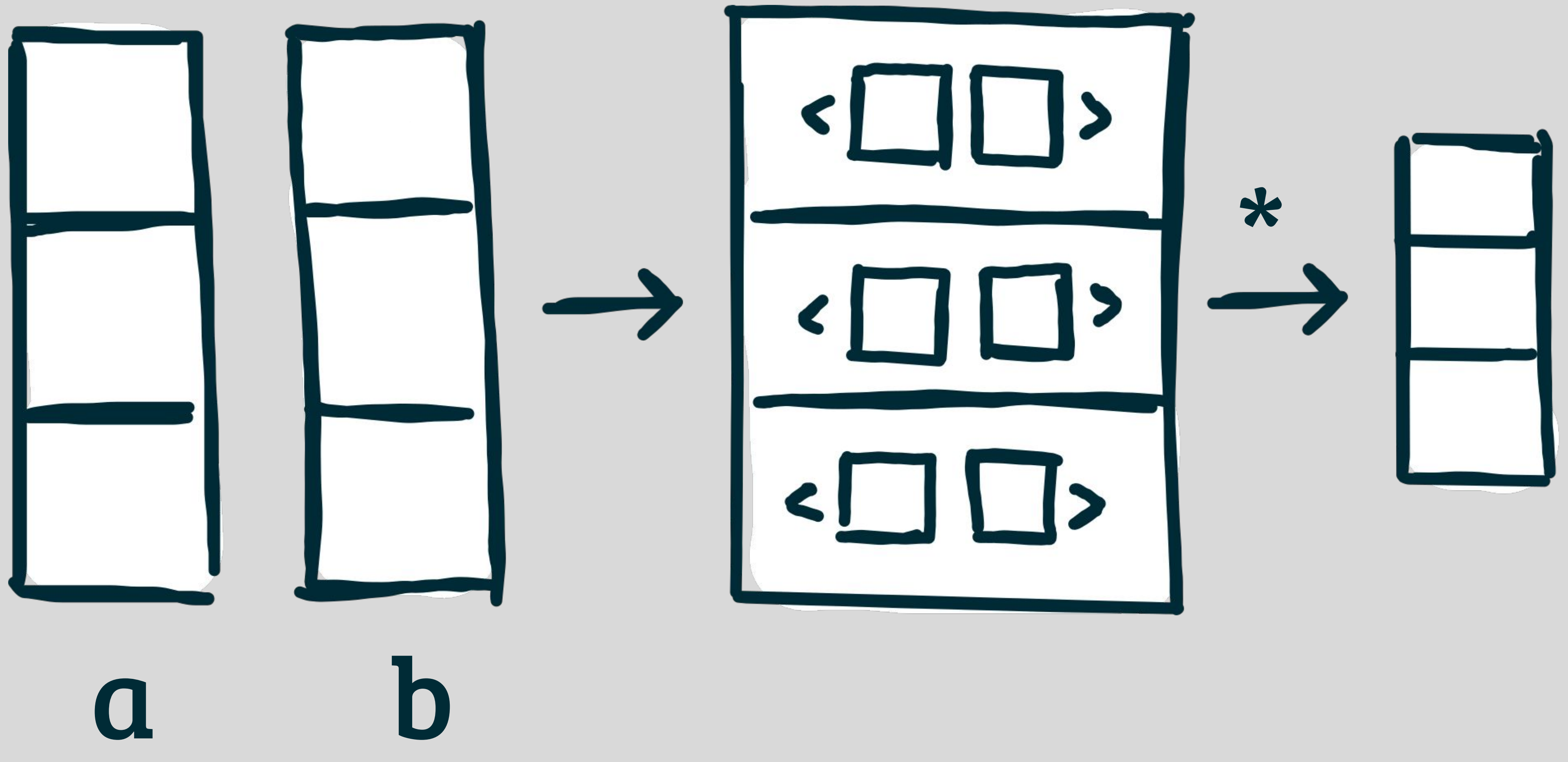


zip(a,b)

LIFT'S HIGH-LEVEL PRIMITIVES

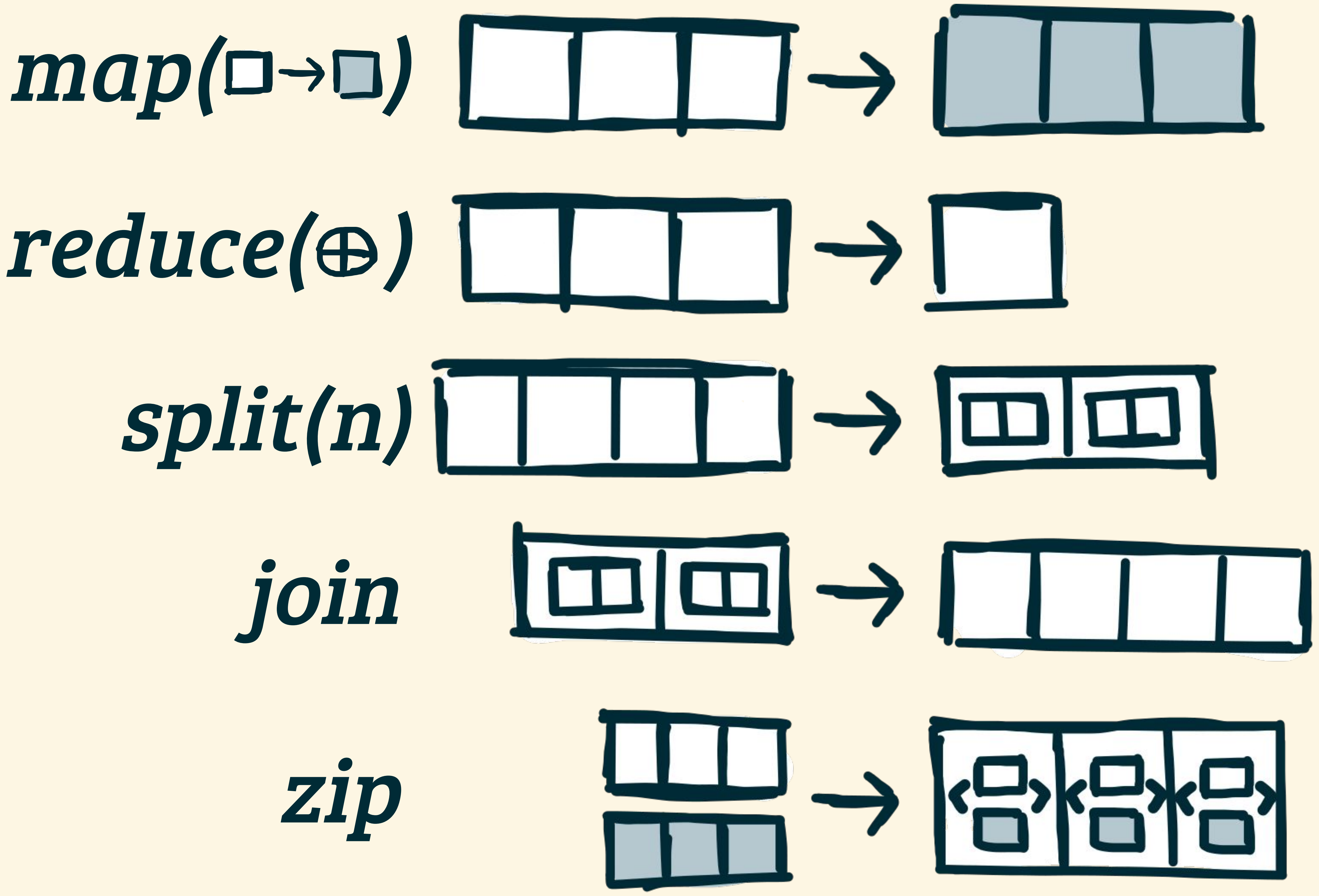


dotproduct.lift

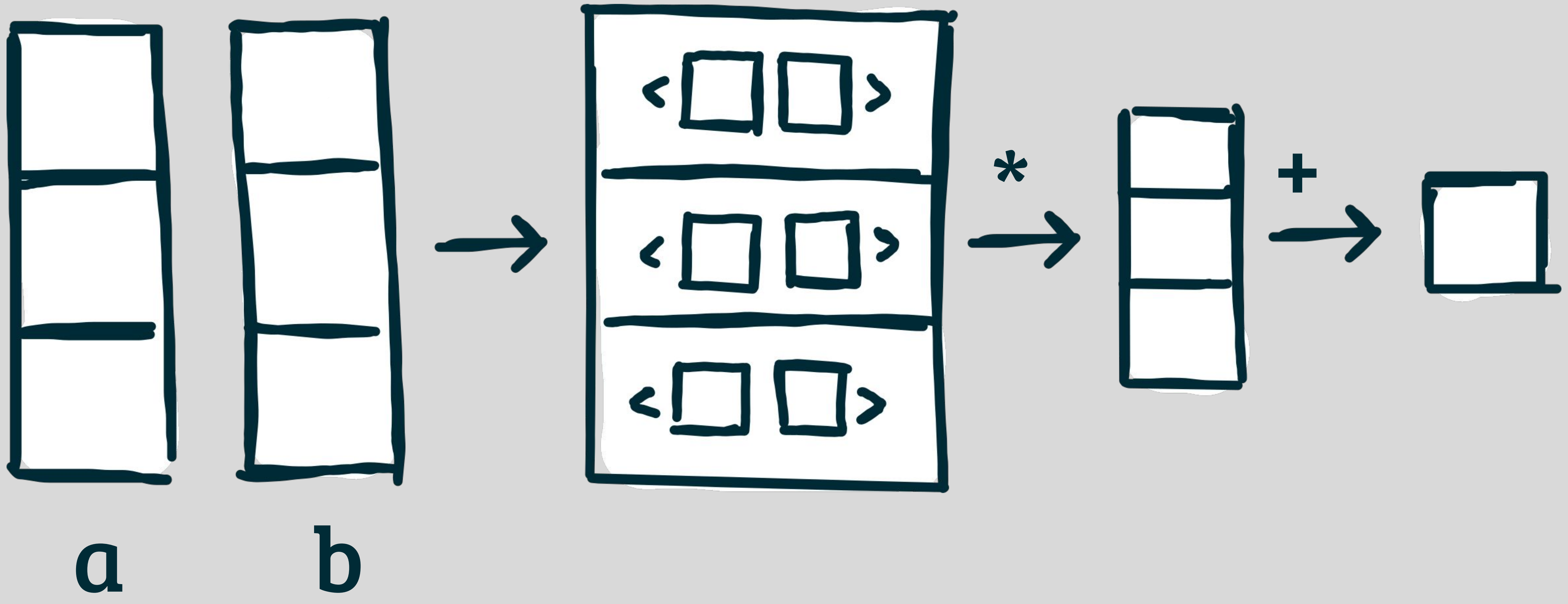



```
map(*, zip(a,b))
```

LIFT'S HIGH-LEVEL PRIMITIVES

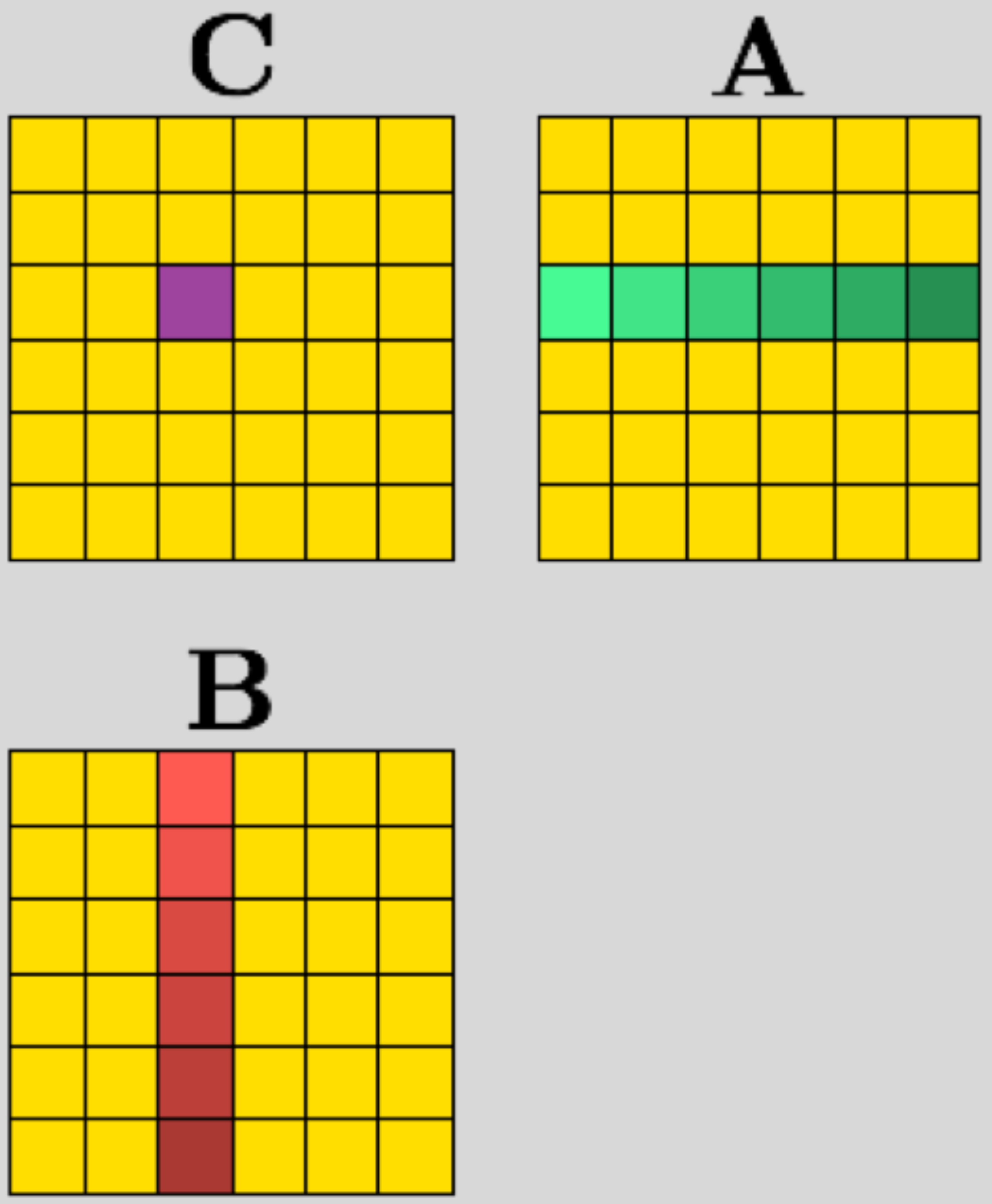


dotproduct.lift



reduce(+,0, *map*(*, *zip*(a,b)))

LIFT'S HIGH-LEVEL PRIMITIVES



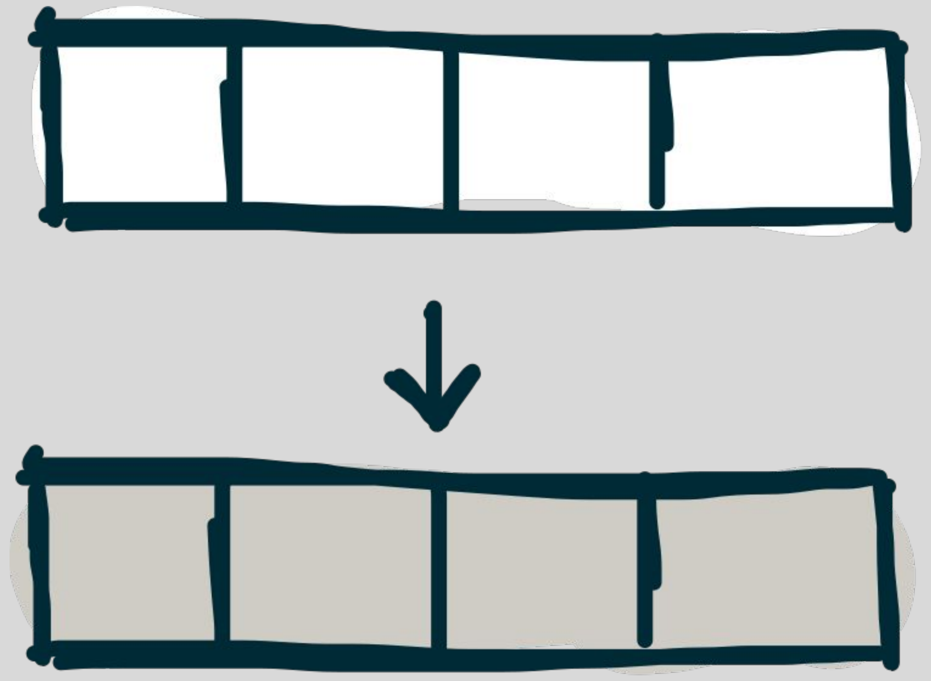
matrixMult.lift

```
map(λ rowA ↦
   map(λ colB ↦
      dotProduct(rowA, colB)
   , transpose(B))
 , A)
```

IMPLEMENTATION CHOICES AS REWRITE RULES

Divide & Conquer

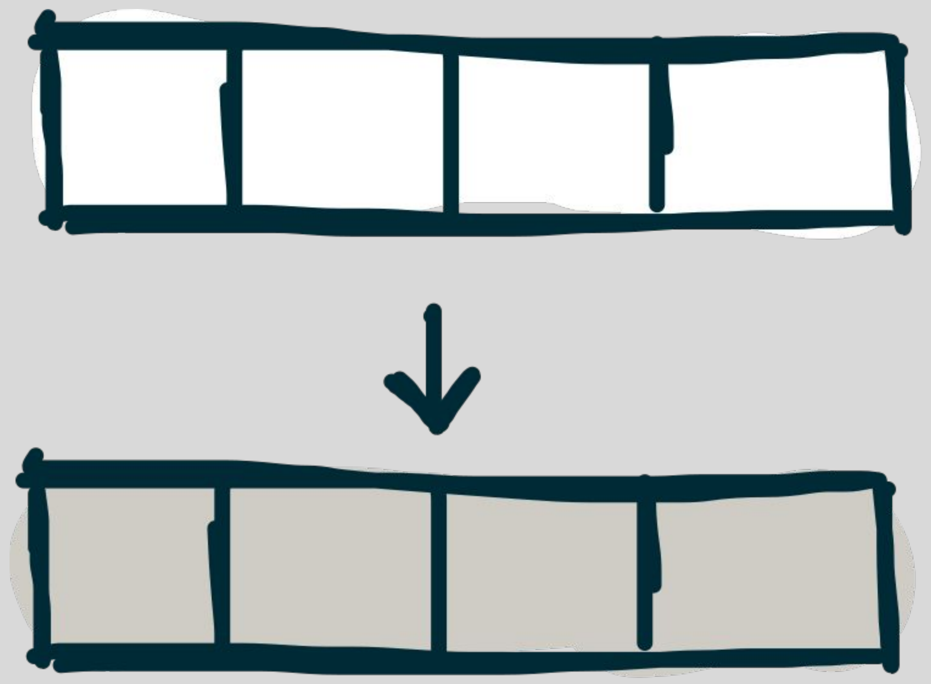
`map(f, A)`

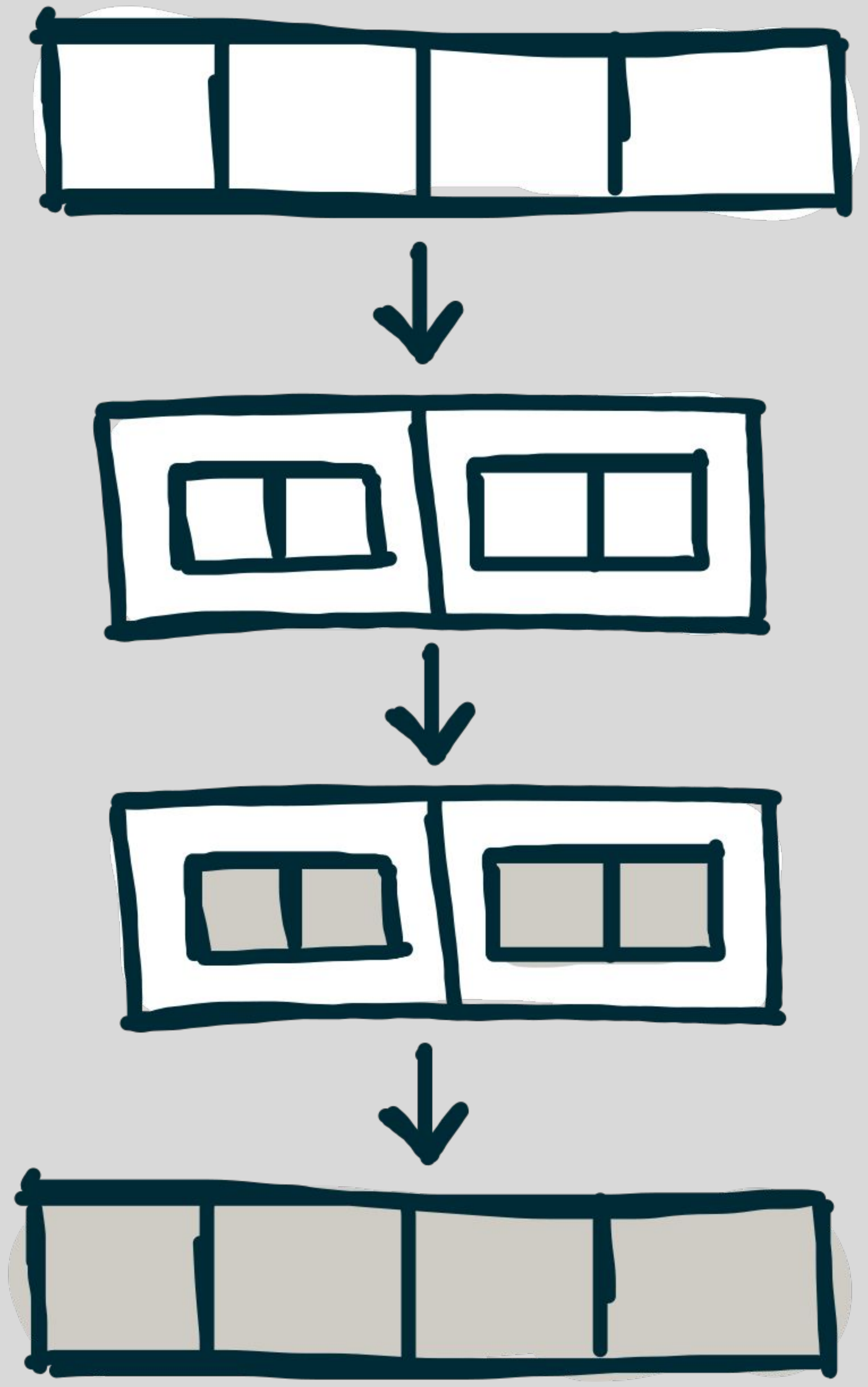


IMPLEMENTATION CHOICES AS REWRITE RULES

Divide & Conquer

$$\mathit{map}(f,\ A) \mapsto \mathit{join}(\mathit{map}(\mathit{map}(f),\ \mathit{split}(n,\ A)))$$





OPTIMIZATIONS AS MACRO RULES

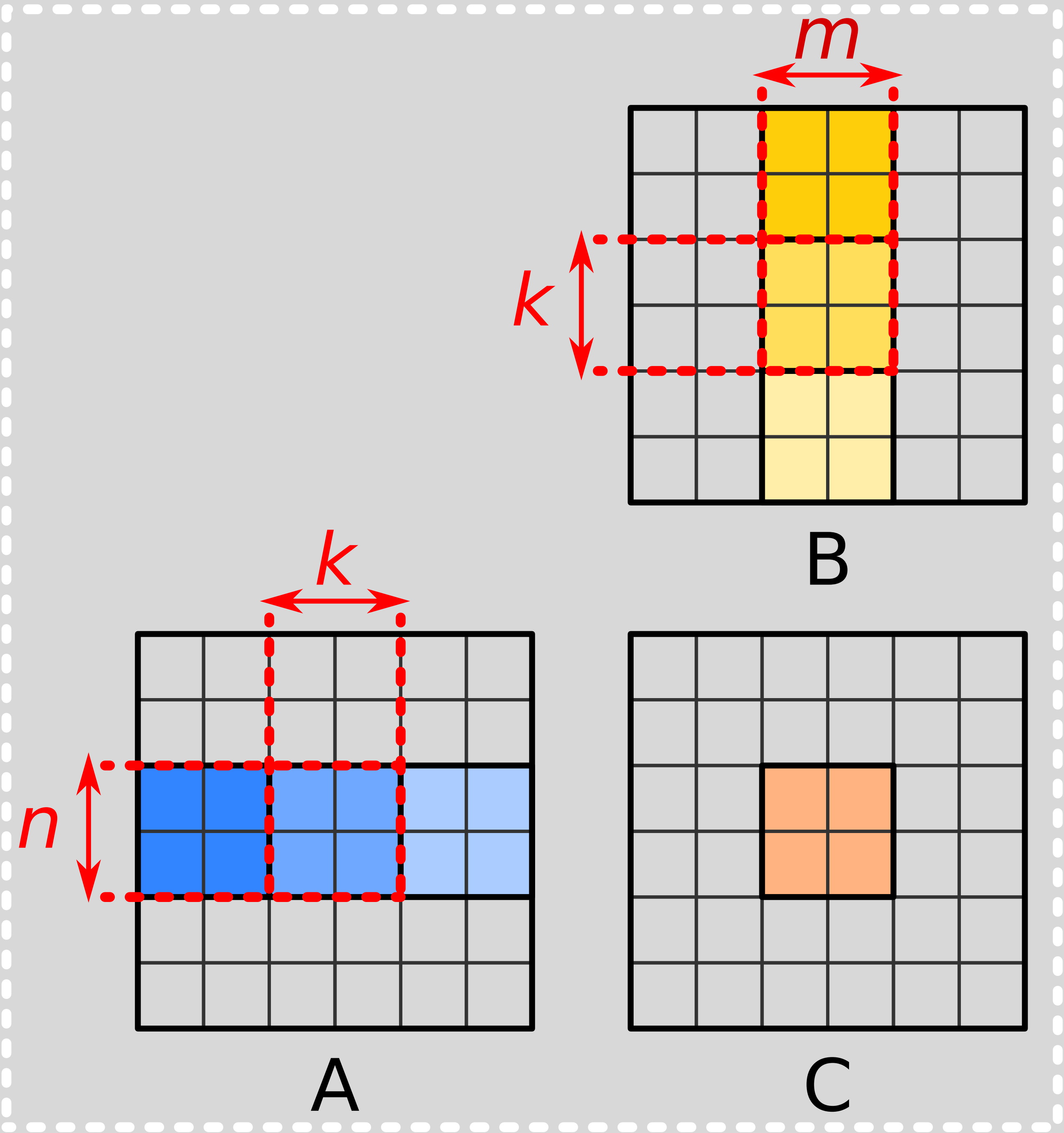
2D Tiling

Naïve matrix multiplication

```
map(λ arow .
  map(λ bcol .
    reduce(+, 0) ∘ map(×) ∘ zip(arow, bcol)
  , transpose(B))
, A)
```

↓ Apply tiling rules

```
untile ∘ map(λ rowOfTilesA .
  map(λ colOfTilesB .
    toGlobal(copy2D) ∘
    reduce(λ (tileAcc, (tileA, tileB)) .
      map(map(+)) ∘ zip(tileAcc) ∘
      map(λ as .
        map(λ bs .
          reduce(+, 0) ∘ map(×) ∘ zip(as, bs)
        , toLocal(copy2D(tileB)))
      , toLocal(copy2D(tileA)))
    ,0, zip(rowOfTilesA, colOfTilesB))
  ) ∘ tile(m, k, transpose(B))
) ∘ tile(n, k, A)
```



[GPGPU'16]

OPTIMIZATIONS AS MACRO RULES

2D Tiling

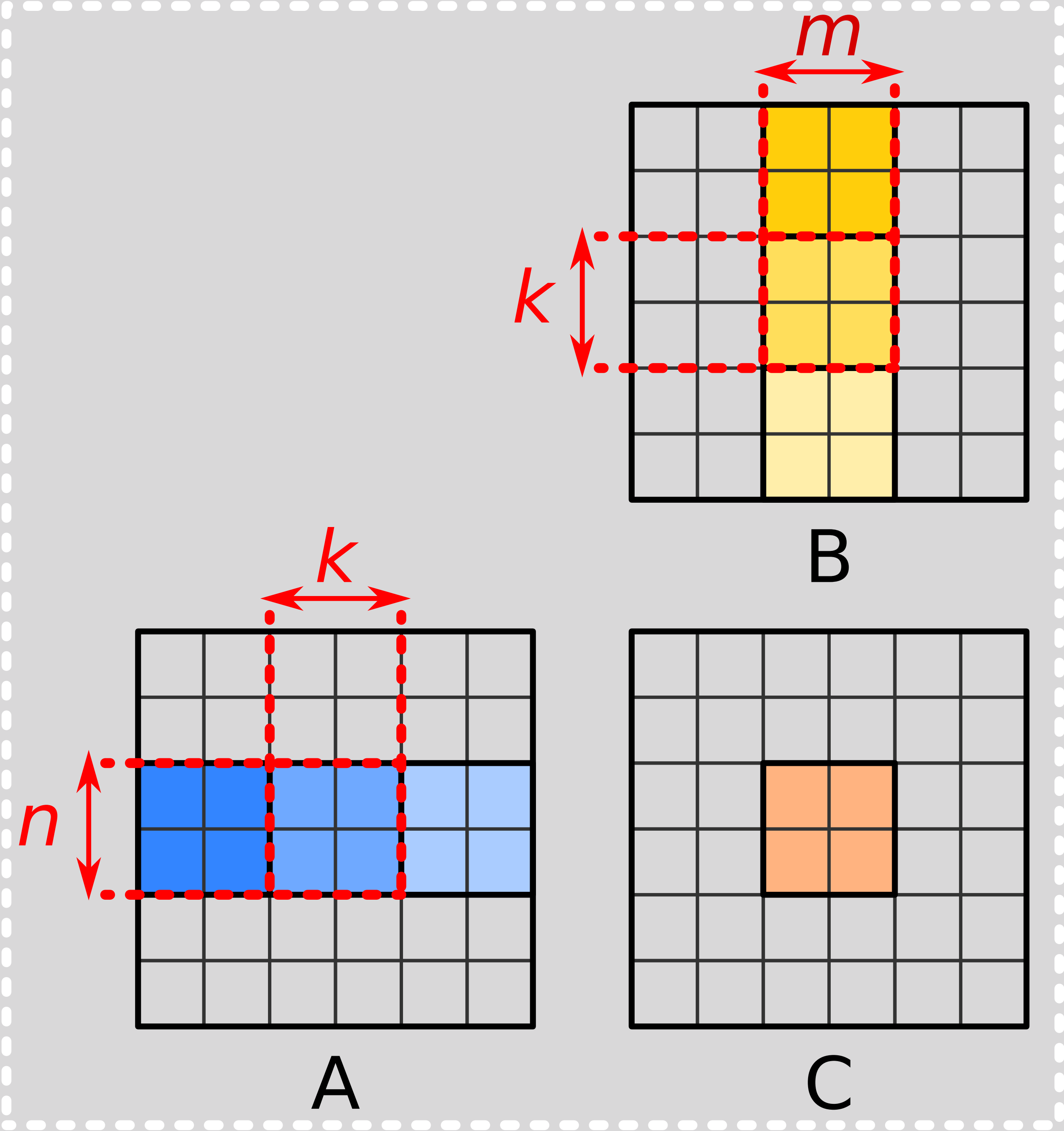
Naïve matrix multiplication

```
map(λ arow .
  map(λ bcol .
    reduce(+, 0) ∘ map(×) ∘ zip(arow, bcol)
  , transpose(B))
, A)
```

Many rewrite rules applied here

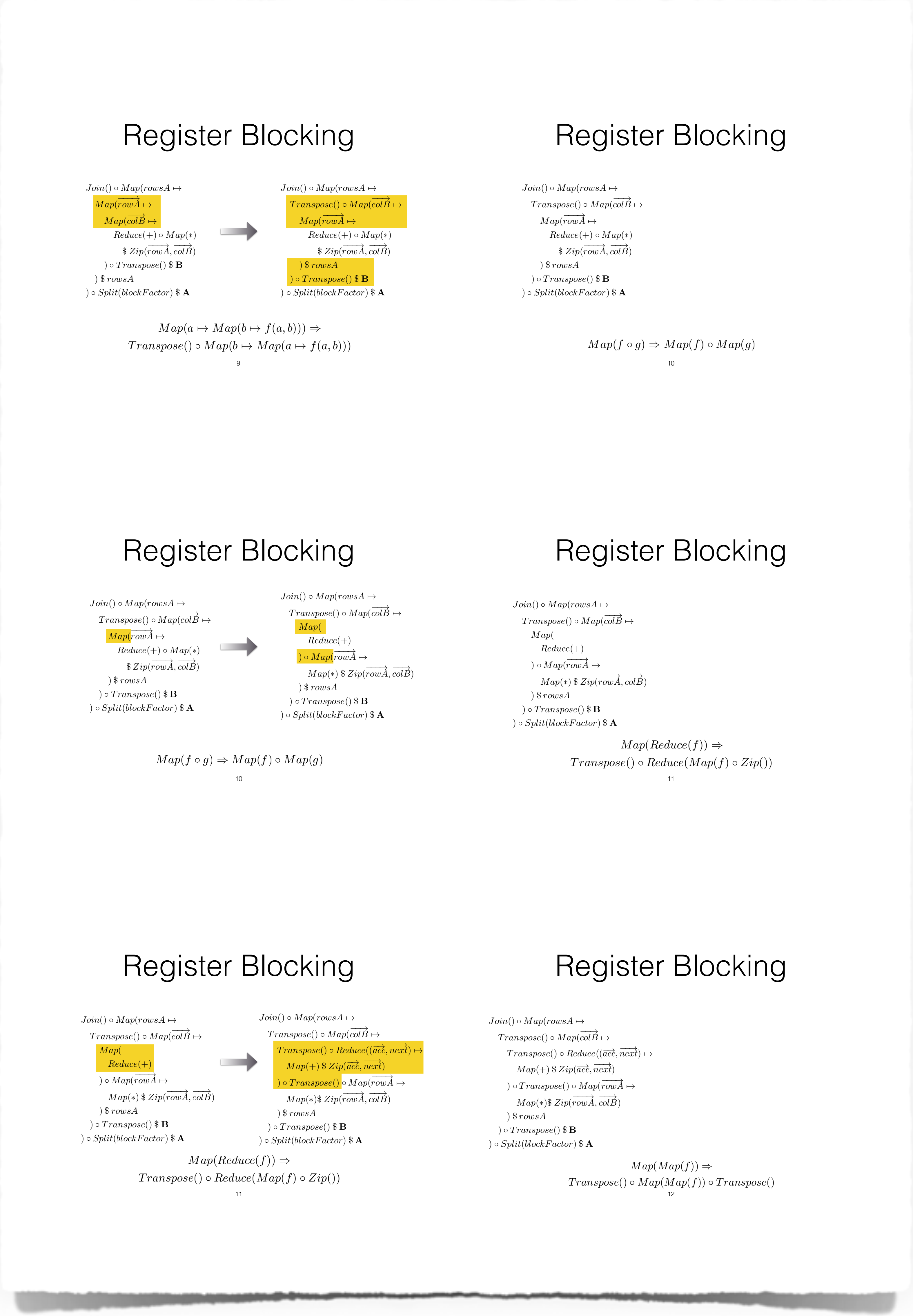
↓ Apply tiling rules

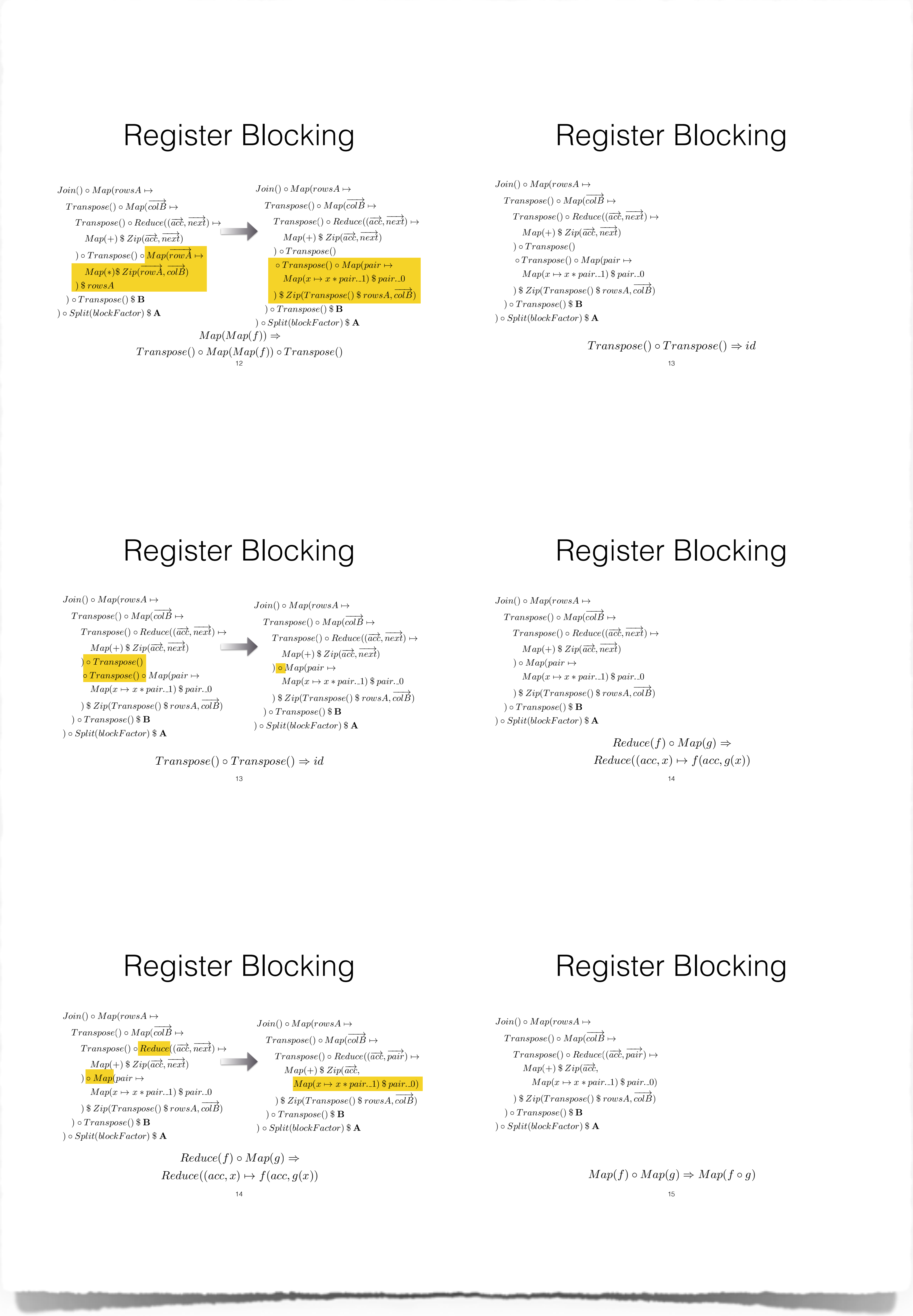
```
untile ∘ map(λ rowOfTilesA .
  map(λ colOfTilesB .
   toGlobal(copy2D) ∘
   reduce(λ (tileAcc, (tileA, tileB)) .
    map(map(+)) ∘ zip(tileAcc) ∘
    map(λ as .
     map(λ bs .
      reduce(+, 0) ∘ map(×) ∘ zip(as, bs)
     , toLocal(copy2D(tileB)))
    , toLocal(copy2D(tileA)))
   ,0, zip(rowOfTilesA, colOfTilesB))
  ) ∘ tile(m, k, transpose(B))
) ∘ tile(n, k, A)
```



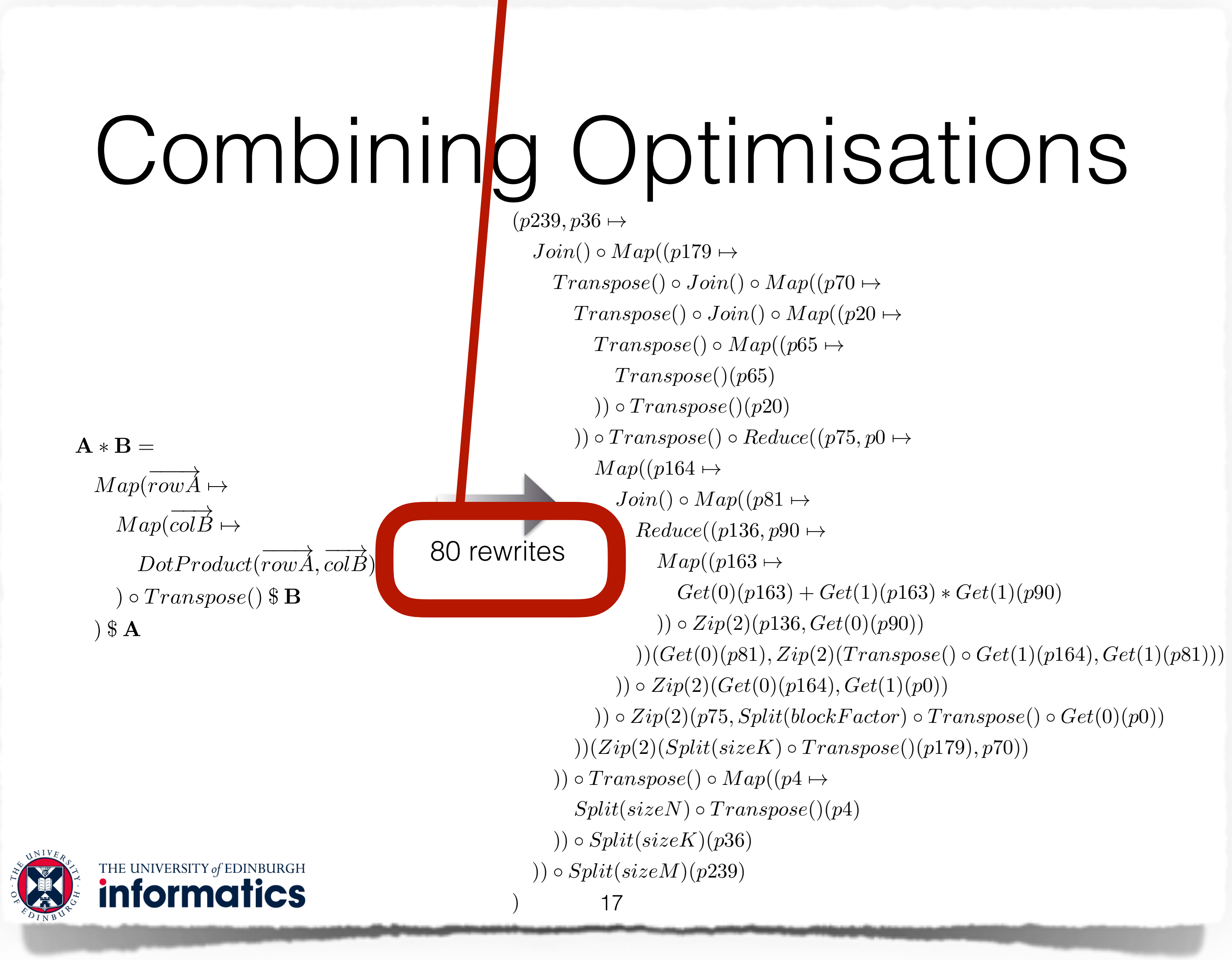
[GPGPU'16]

[GPGPU'16] Presentation Slides





80 rewrite steps!



ELEVATE for optimising LIFT programs

LIFT Program

Domain Specific Language embedded in Scala

```
val scale = fun(a ⇒ fun(xs ⇒
 xs ▷ map(fun(x ⇒ a * x )) ))
```

ELEVATE Program

Domain Specific Language embedded in Scala

LIFT rewrite rule in ELEVATE

LIFT **Program**

```
val scale = fun(a ⇒ fun(xs ⇒
 xs ▷ map(fun(x ⇒ a * x )) ))
```

ELEVATE **Program**

```
def splitJoin(n: Nat)(e: Lift): RewriteResult[Lift] = e match {
  case Apply(`map`, f) ⇒ Success(split(n) ▷ map(map(f)) ▷ join)
  case _ ⇒ Failure(splitJoin(n))
}
```

LIFT rewrite rule in ELEVATE

LIFT **Program**

```
val scale = fun(a ⇒ fun(xs ⇒
 xs ▷ map(fun(x ⇒ a * x )) ))
```

ELEVATE **Program**

```
def splitJoin(n: Nat)(e: Lift): RewriteResult[Lift] = e match {
  case Apply(`map`, f) ⇒ Success(split(n) ▷ map(map(f)) ▷ join)
  case _ ⇒ Failure(splitJoin(n))
}
```

LIFT

map(f, A) $\mapsto$ ***join***(***map***(***map***(f), ***split***(n, A)))

LIFT rewrite rule in ELEVATE

LIFT Program

```
val scale = fun(a ⇒ fun(xs ⇒
 xs ▷ map(fun(x ⇒ a * x )) ))
```

ELEVATE Program

```
def splitJoin(n: Nat)(e: Lift): RewriteResult[Lift] = e match {
  case Apply(`map`, f) ⇒ Success(split(n) ▷ map(map(f)) ▷ join)
  case _ ⇒ Failure(splitJoin(n))
}
```

Apply transformation:

```
splitJoin(n)(scale)
```

LIFT rewrite rule in ELEVATE

LIFT Program

```
val scale = fun(a ⇒ fun(xs ⇒
 xs ▷ map(fun(x ⇒ a * x )) ))
```

ELEVATE Program

```
def splitJoin(n: Nat)(e: Lift): RewriteResult[Lift] = e match {
  case Apply(`map`, f) ⇒ Success(split(n) ▷ map(map(f)) ▷ join)
  case _ ⇒ Failure(splitJoin(n))
}
```

Apply transformation:

```
splitJoin(n)(scale)
```

Failure!

LIFT rewrite rule in ELEVATE

LIFT Program

```
val scale = fun(a ⇒ fun(xs ⇒
 xs ▷ map(fun(x ⇒ a * x )) ))
```

ELEVATE Program

```
def splitJoin(n: Nat)(e: Lift): RewriteResult[Lift] = e match {
  case Apply(`map`, f) ⇒ Success(split(n) ▷ map(map(f)) ▷ join)
  case _ ⇒ Failure(splitJoin(n))
}
```

Apply transformation:

```
splitJoin(n)(scale)
```

Failure!

The transformation is applied at the wrong location

LIFT rewrite rule in E

LIFT Program

```
val scale = fun(a ⇒ fun(xs ⇒
 xs ▷ map(fun(x ⇒ a * x )) ))
```

ELEVATE Program

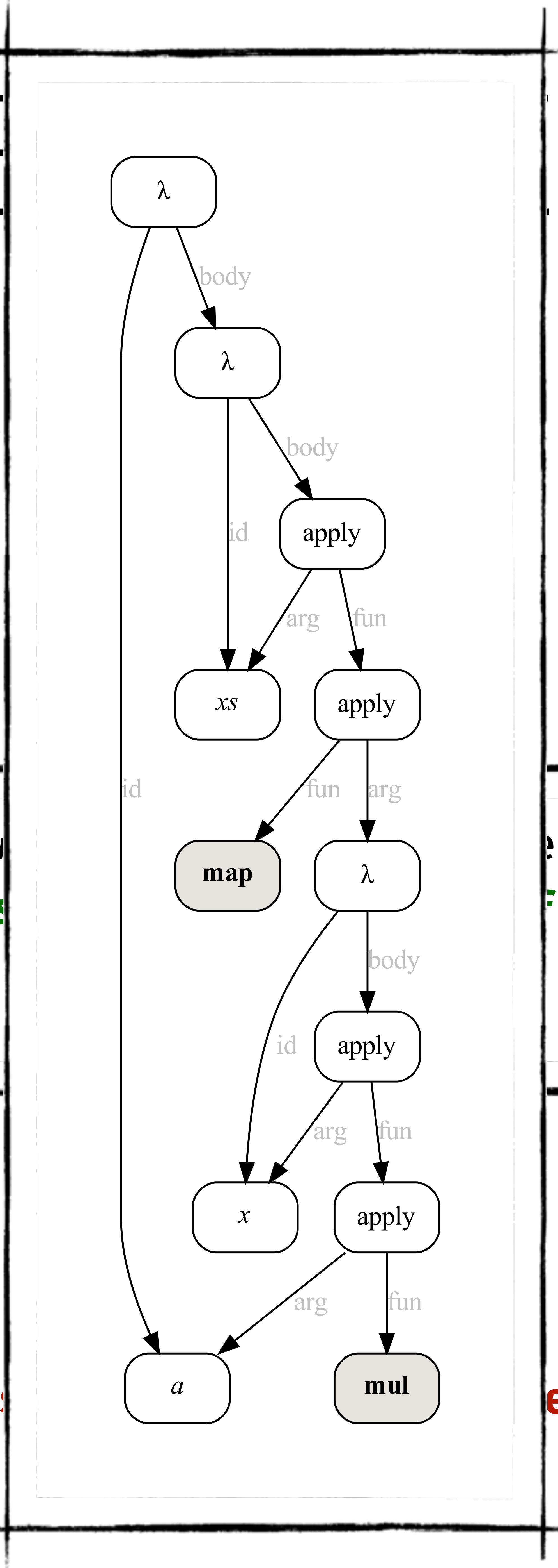
```
def splitJoin(n: Nat)(e: Lift): Rew                e match {
  case Apply(`map`, f) ⇒ Success(s                f)) ▷ join)
  case _ ⇒ Failure(splitJoin(n))
}
```

Apply transformation:

```
splitJoin(n)(scale)
```

Failure!

The trans   e wrong location



LIFT rewrite rule in E

LIFT Program

```
val scale = fun(a ⇒ fun(xs ⇒
 xs ▷ map(fun(x ⇒ a * x )) ))
```

ELEVATE Program

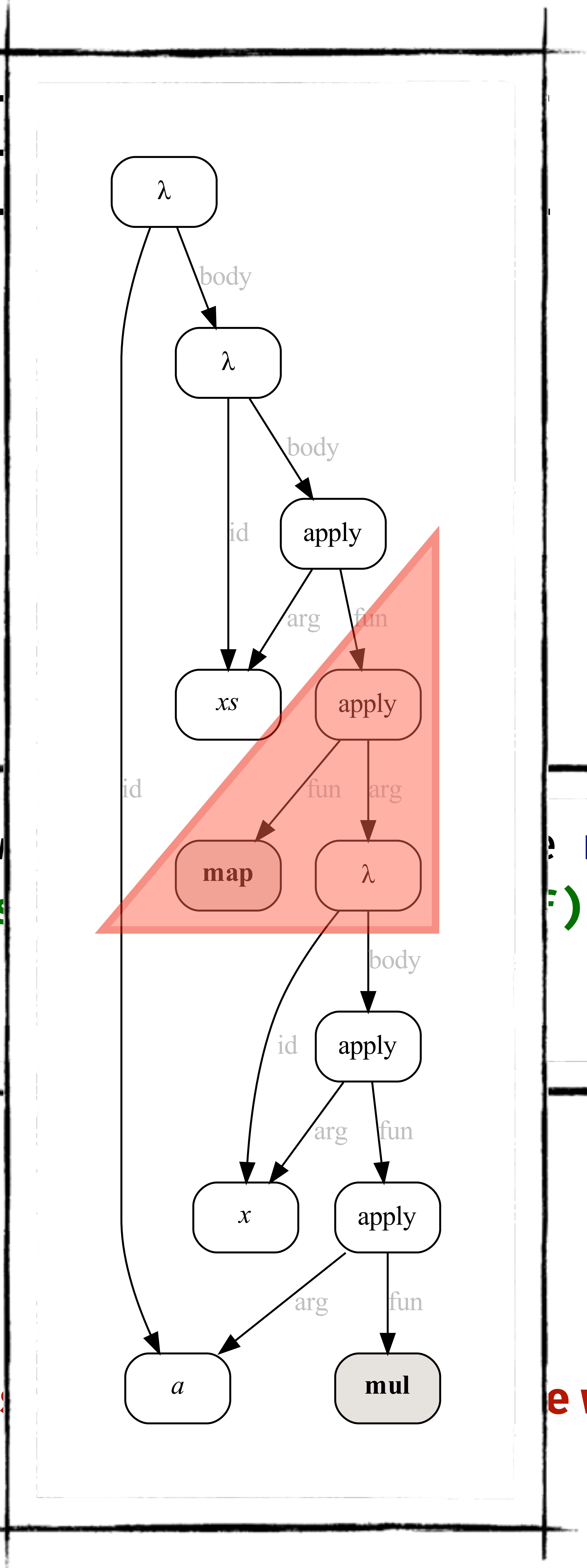
```
def splitJoin(n: Nat)(e: Lift): Rew                    e match {
  case Apply(`map`, f) ⇒ Success(s                    f)) ▷ join)
  case _ ⇒ Failure(splitJoin(n))
}
```

Apply transformation:

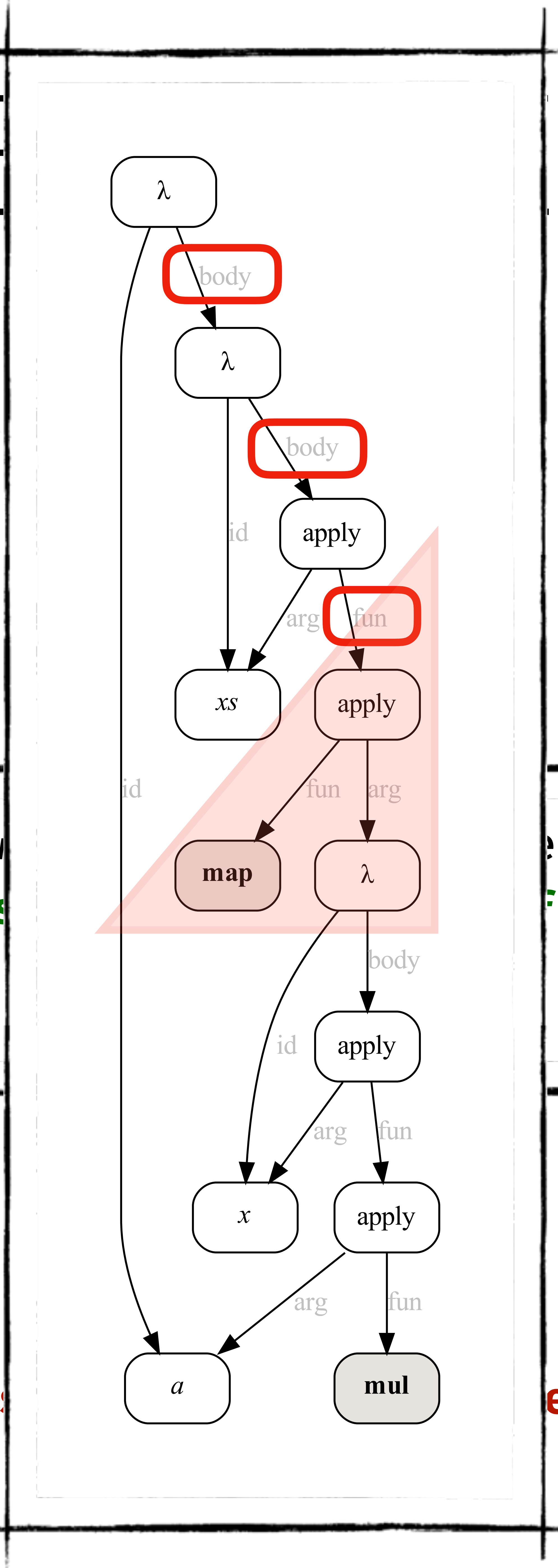
```
splitJoin(n)(scale)
```

Failure!

The trans   e wrong location



LIFT rewrite rule in E



LIFT Program

```
val scale = fun(a ⇒ fun(xs ⇒
 xs ▷ map(fun(x ⇒ a * x )) ))
```

ELEVATE Program

```
def splitJoin(n: Nat)(e: Lift): Rew                e match {
  case Apply(`map`, f) ⇒ Success(s                 f)) ▷ join)
  case _ ⇒ Failure(splitJoin(n))
}
```

Apply transformation:

```
splitJoin(n)(scale)
```

Failure!

The trans   e wrong location

Traversal LIFT programs

ELEVATE **Program**

```
def body(s: Lift ⇒ RewriteResult[Lift])
        (e: Lift): RewriteResult[Lift] = e match {
  case Lambda(f, x) ⇒ s(x).mapSuccess(y ⇒ Lambda(f, y) )
  case _ ⇒ Failure(s)
}

def function(s: Lift ⇒ RewriteResult[Lift])
            (e: Lift): RewriteResult[Lift] = e match {
  case Apply(f, e) ⇒ s(f).mapSuccess(g ⇒ Apply(g, e))
  case _ ⇒ Failure(s)
}
```

Traversal LIFT programs

ELEVATE **Program**

```
def body(s: Lift ⇒ RewriteResult[Lift])
        (e: Lift): RewriteResult[Lift] = e match {
  case Lambda(f, x) ⇒ s(x).mapSuccess(y ⇒ Lambda(f, y) )
  case _ ⇒ Failure(s)
}

def function(s: Lift ⇒ RewriteResult[Lift])
            (e: Lift): RewriteResult[Lift] = e match {
  case Apply(f, e) ⇒ s(f).mapSuccess(g ⇒ Apply(g, e))
  case _ ⇒ Failure(s)
}
```

Apply transformation:

```
body(body(function(splitJoin(n))))(
  fun(a ⇒ fun(xs ⇒
   xs ▷ map(fun(x ⇒ a * x )) ))
)
```

Traversal LIFT prog

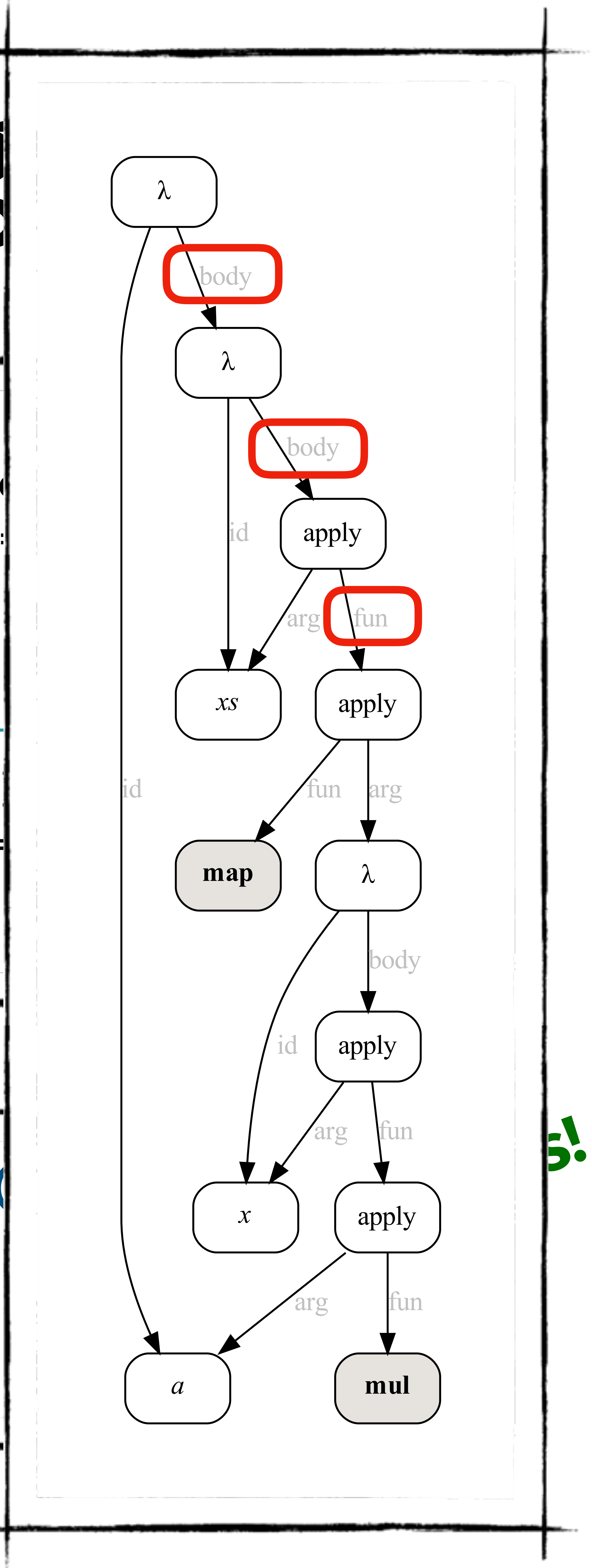
ELEVATE **Program**

```
def body(s: Lift ⇒ RewriteResult[Lift])
        (e: Lift): RewriteResult[Lift] = e
  case Lambda(f, x) ⇒ s(x).mapSuccess(y :
  case _ ⇒ Failure(s)
}

def function(s: Lift ⇒ RewriteResult[Lif
            (e: Lift): RewriteResult[Lift
  case Apply(f, e) ⇒ s(f).mapSuccess(g ⇒
  case _ ⇒ Failure(s)
}
```

Apply transformation:

```
body(body(function(splitJoin(n))))(
  fun(a ⇒ fun(xs ⇒
   xs ▷ map(fun(x ⇒ a * x )) ))
)
```



s!

Traversal LIFT programs

ELEVATE **Program**

```
def body(s: Lift ⇒ RewriteResult[Lift])
        (e: Lift): RewriteResult[Lift] = e match {
  case Lambda(f, x) ⇒ s(x).mapSuccess(y ⇒ Lambda(f, y) )
  case _ ⇒ Failure(s)
}

def function(s: Lift ⇒ RewriteResult[Lift])
            (e: Lift): RewriteResult[Lift] = e match {
  case Apply(f, e) ⇒ s(f).mapSuccess(g ⇒ Apply(g, e))
  case _ ⇒ Failure(s)
}
```

Apply transformation:

```
body(body(function(splitJoin(n))))(
  fun(a ⇒ fun(xs ⇒
   xs ▷ map(fun(x ⇒ a * x )) ))
)
```

Success!

Traversal LIFT programs

ELEVATE **Program**

Compose existing *strategies*

```
def body(s: Lift ⇒ RewriteResult[Lift])
        (e: Lift): RewriteResult[Lift] = e match {
  case Lambda(f, x) ⇒ s(x).mapSuccess(y ⇒ Lambda(f, y) )
  case _ ⇒ Failure(s)
}

def function(s: Lift ⇒ RewriteResult[Lift])
            (e: Lift): RewriteResult[Lift] = e match {
  case Apply(f, e) ⇒ s(f).mapSuccess(g ⇒ Apply(g, e))
  case _ ⇒ Failure(s)
}
```

Apply transformation:

```
body(body(function(splitJoin(n))))(
  fun(a ⇒ fun(xs ⇒
   xs ▷ map(fun(x ⇒ a * x )) ))
)
```

Success!

Traversal LIFT programs

ELEVATE **Program**

Compose existing *strategies*

```
def body(s: Lift ⇒ RewriteResult[Lift])
        (e: Lift): RewriteResult[Lift] = e match {
  case Lambda(f, x) ⇒ s(x).mapSuccess(y ⇒ Lambda(f, y) )
  case _ ⇒ Failure(s)
}

def function(s: Lift ⇒ RewriteResult[Lift])
            (e: Lift): RewriteResult[Lift] = e match {
  case Apply(f, e) ⇒ s(f).mapSuccess(g ⇒ Apply(g, e))
  case _ ⇒ Failure(s)
}
```

These are domain specific abstractions that makes sense for optimising LIFT **programs.**

These are not backed into ELEVATE

Apply transformation:

```
body(body(function(splitJoin(n))))(
  fun(a ⇒ fun(xs ⇒
   xs ▷ map(fun(x ⇒ a * x )) ))
)
```

Success!

Generic ELEVATE combinators

ELEVATE **defines generic combinators for programs written in an arbitrary language P**

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

def id[P](p: P) = Success(p)

def seq[P](f: Strategy[P], s: Strategy[P])
          (p: P): RewriteResult[P] = f(p).flatMapSuccess(s)

def leftChoice[P](f: Strategy[P], s: Strategy[P])
                 (p: P): RewriteResult[P] = f(p).flatMapFailure(_ ⇒ s(p))

def try[P](s: Strategy[P])
          (p: P): RewriteResult[P] = leftChoice[P](s, id)(p)

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

Generic ELEVATE combinators

ELEVATE **defines generic combinators for programs written in an arbitrary language P**

Syntactic sugar:

```
f `;` s

f <+ s
```

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

def id[P](p: P) = Success(p)

def seq[P](f: Strategy[P], s: Strategy[P])
          (p: P): RewriteResult[P] = f(p).flatMapSuccess(s)

def leftChoice[P](f: Strategy[P], s: Strategy[P])
                 (p: P): RewriteResult[P] = f(p).flatMapFailure(_ ⇒ s(p))

def try[P](s: Strategy[P])
          (p: P): RewriteResult[P] = leftChoice[P](s, id)(p)

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

Generic ELEVATE combinators

Syntactic sugar:

```
f `;` s

f <+ s
```

```
ty
de
de
de
de
de
..
```

[ICFP 1998]

Building Program Optimizers with Rewriting Strategies*

Eelco Visser[1], Zine-el-Abidine Benaissa[1], Andrew Tolmach[1,2]
Pacific Software Research Center

[1] Dept. of Comp. Science and Engineering, Oregon Graduate Institute, P.O. Box 91000, Portland, Oregon 97291-1000, USA
[2] Dept. of Computer Science, Portland State University, P.O. Box 751, Portland, Oregon 97207 USA
visser@acm.org, benaissa@cse.ogi.edu, apt@cs.pdx.edu

Abstract

We describe a language for defining term rewriting strategies, and its application to the production of program optimizers. Valid transformations on program terms can be described by a set of rewrite rules; rewriting strategies are used to describe when and how the various rules should be applied in order to obtain the desired optimization effects. Separating rules from strategies in this fashion makes it easier to reason about the behavior of the optimizer as a whole, compared to traditional monolithic optimizer implementa-

A program optimizer transforms the source code of a program into a program that has the same meaning, but is more efficient. On the level of specification and documentation, optimizers are often presented as a set of correctness-preserving *rewrite rules* that transform code fragments into equivalent more efficient code fragments (e.g., see Table 5). This is particularly attractive for functional language compilers (e.g., [3, 4, 24]) that operate via successive small transformations, and don't rely on analyses requiring significant auxiliary data structures. The paradigm provided by conventional rewrite engines is to compute the normal form of

Generic ELEVATE traversals

ELEVATE **defines generic traversals if three basic traversals are defined for P**

```
// applies strategy to all direct subexpressions
def all[P]: Strategy[P] ⇒ Strategy[P]

// applies strategy to one direct subexpression
def one[P]: Strategy[P] ⇒ Strategy[P]

// applies strategy to at least one direct subexpression
def some[P]: Strategy[P] ⇒ Strategy[P]

def oncetd[P](s: Strategy[P])
             (p: P): RewriteResult[P] = (s <+ one(oncetd(s)))(p)

def tryAll[P](s: Strategy[P])
             (p: P): RewriteResult[P] = (all(tryAll(try(s))) `;` try(s))(p)
...
```

Generic ELEVATE normalisation

ELEVATE **defines a normalisation strategy based on the generic traversals**

```
def normalize[P]: Strategy[P] ⇒ Strategy[P] = s ⇒ repeat(oncetd(s))
```

This applies a given strategy until this is not applicable anymore

Complex compiler optimisations in ELEVATE

With ELEVATE we easily express traditional compiler optimisations, like `tiling` or loop reordering:

```
def tileNDRec: Int ⇒ Int ⇒ Strategy[Lift] = dim ⇒ n ⇒ dim match {
  case x if x ⩽ 0 ⇒ id()
  case 1 ⇒ function(splitJoin(n))
  case 2 ⇒ fmap(function(splitJoin(n))) `;` function(splitJoin(n)) `;` shiftDim(2)
  case i ⇒ fmap(tileNDRec(dim-1)(n)) `;` tileNDRec(1)(n) `;` shiftDim(i)
}
```

```
def reorder: Seq[Int] ⇒ Strategy[Lift] = perm ⇒ {
  if(perm.length == 1) return id
  (perm.head match {
    case 1 ⇒ fmap(reorder(perm.tail.map(_-1)))
    case x ⇒
      val transposes = x-1
      shiftDimension(transposes) `;`
      moveTowardsArgument(transposes)(fmap(reorder(perm.tail.map(y ⇒ if(y > x) y-1 else y ))))
  }) `;` RNF `;` LCNF
}
```

Complex compiler optimisations in ELEVATE

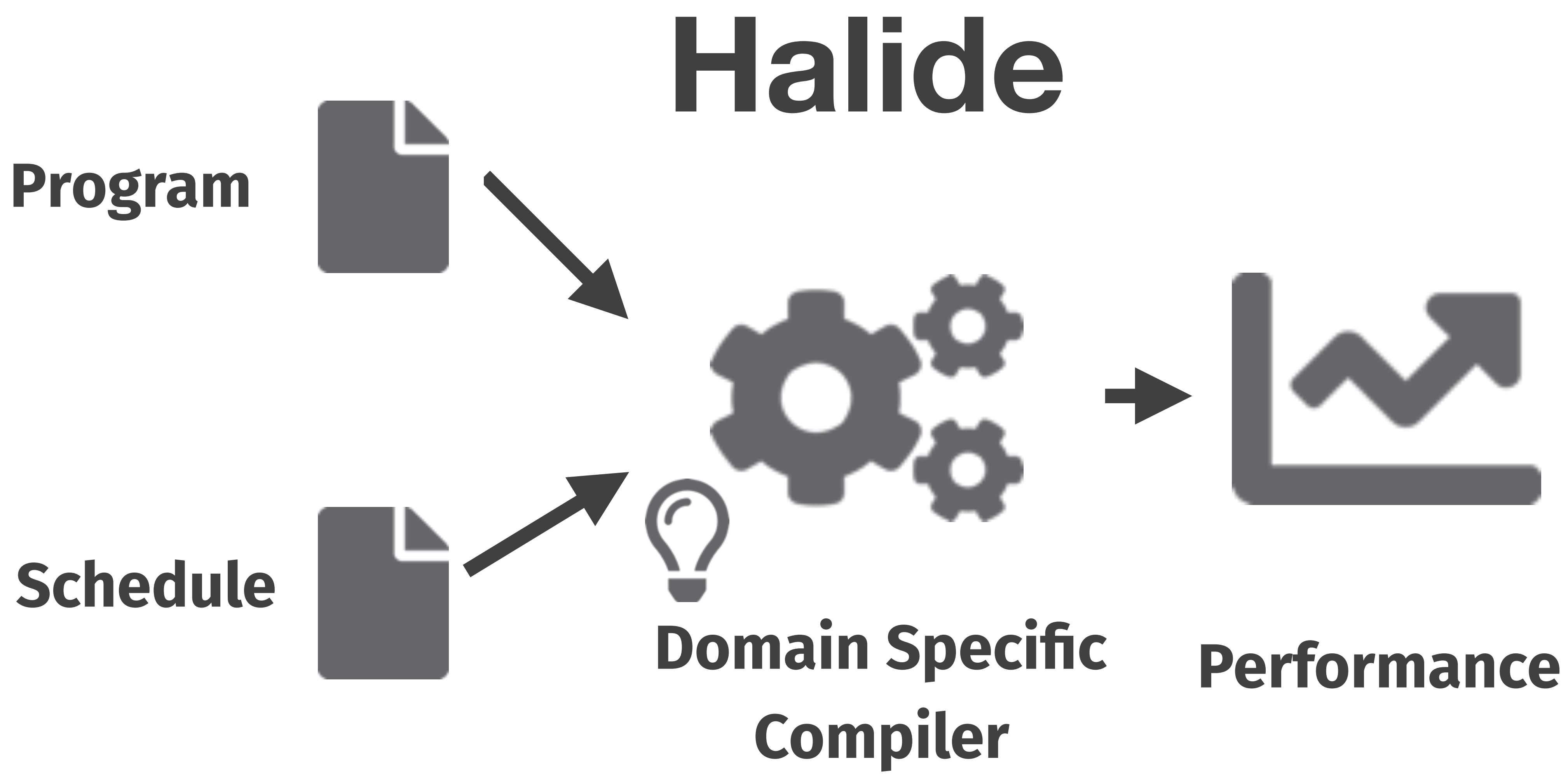
With ELEVATE we easily express traditional compiler optimisations, like `tiling` or loop reordering:

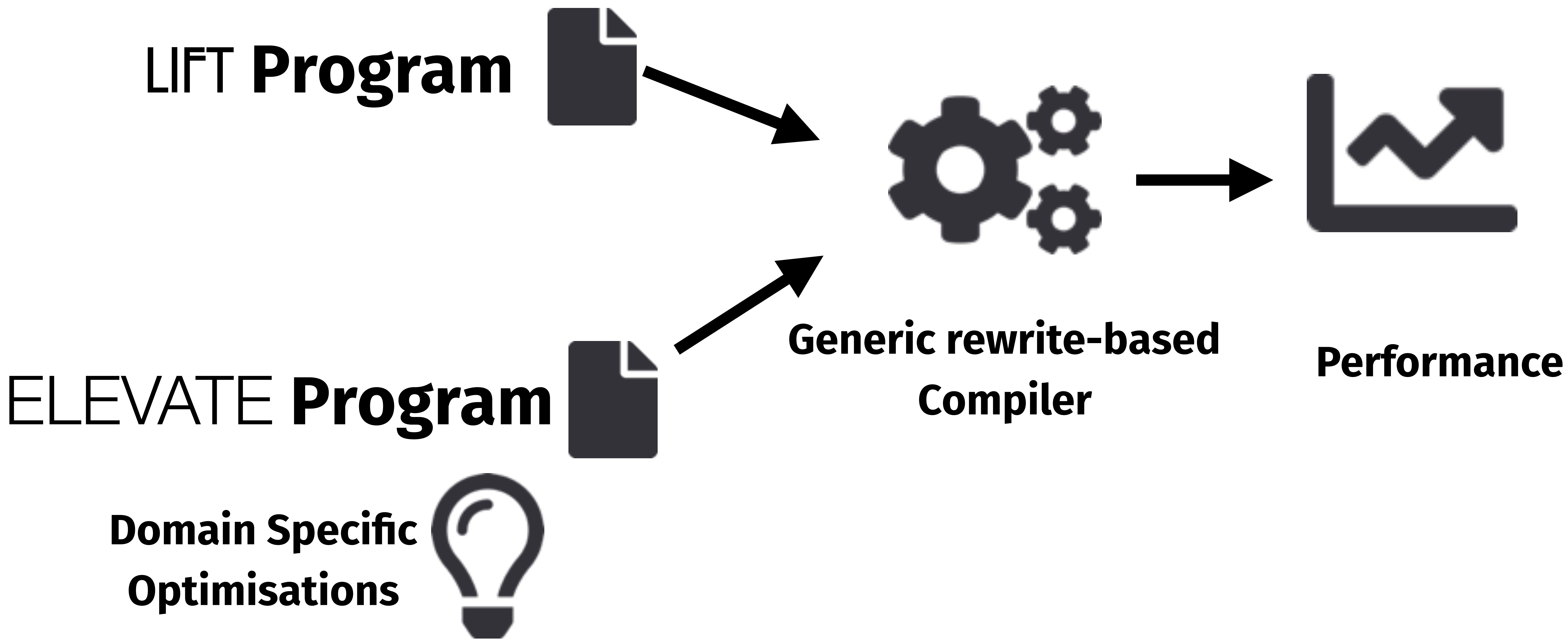
```
def tileNDRec: Int ⇒ Int ⇒ Strategy[Lift] = dim ⇒ n ⇒ dim match {
  case x if x ⩽ 0 ⇒ id()
  case 1 ⇒ function(splitJoin(n))
  case 2 ⇒ fmap(function(splitJoin(n))) `;` function(splitJoin(n)) `;` shiftDim(2)
```

```
   float[B][A]
1. float[bTile][B/bTile][A]                  // traverse to innermost dim and apply split join
2. float[bTile][B/bTile][aTile][A/aTile]     // apply splitJoin to next `map` going inner → outer
3. float[bTile][aTile][B/bTile][A/aTile]     // reorder tiles using map(transpose)
```

```
  if(perm.length == 1) return id
  (perm.head match {
    case 1 ⇒ fmap(reorder(perm.tail.map(_-1)))
    case x ⇒
      val transposes = x-1
      shiftDimension(transposes) `;`
      moveTowardsArgument(transposes)(fmap(reorder(perm.tail.map(y ⇒ if(y > x) y-1 else y ))))
  }) `;` RNF `;` LCNF
}
```

ELEVATE for optimising LIFT programs





Goal: Demonstrate same performance as Halide with a more extensible design

ELEVATE for optimising **FSmooth** programs

[ICFP 2019]

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 required for numerical algorithms such as Gauss-Newton and Levenberg-Marquardt can be efficiently computed.

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

5 EFFICIENT DIFFERENTIATION

…

One of the key challenges for applying these rewrite rules is the order in which these rules should be applied. We apply these rules based on **heuristics** and **cost models for the size of the code** (which is used by many optimising compilers, especially the ones for just-in-time scenarios). Furthermore, based on heuristics, we ensure that certain rules are applied only when some specific other rules are applicable. For example, the loop fission rule (Figure 8g) is usually applicable only when it can be combined with tuple projection partial evaluation rules (Figure 8f). **We leave the use of search strategies for automated rewriting** (e.g., using Monte-Carlo tree search [De Mesmay et al. 2009]) **as future work**.

…

ELEVATE for optimising **FSmooth** programs

(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
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.

```scala
def funToLet(e: FSmooth): RewriteResult[FSmooth] = e match {
  case Application(Abstraction(Seq(x), e0, _), Seq(e1), _) ⇒
    Success(Let(x, e1, e0))
  case _ ⇒ Failure(funToLet)
}

def additionZero(e: FSmooth): RewriteResult[FSmooth] = e match {
  case Application(`+`(_), Seq(e, ScalarValue(0)), _) ⇒
    Success(e)
  case Application(`+`(_), Seq(ScalarValue(0), e), _) ⇒
    Success(e)
  case _ ⇒ Failure(additionZero)
}

def trivialFold(e: FSmooth): RewriteResult[FSmooth] = e match {
  case Application(`ifold`(_), Seq(f, z, ScalarValue(0)), _) ⇒
    Success(z)
  case _ ⇒ Failure(trivialFold)
}

...
```

ELEVATE for optimising **FSmooth** programs

Example 5. It is known that for a matrix M, the following equality holds $(M^T)^T = M$. We show how we can derive the same equality in $\widetilde{\mathrm{dF}}$. In other words, we show that:

matrixTranspose (matrixTranspose M) = M

```
let MT =
  build (length M[0]) (fun i ->
    build (length M) (fun j ->
      M[j][i] ) ) in
build (length MT[0]) (fun i ->
  build (length MT) (fun j ->
    MT[j][i] ) )
```

Now, by applying the loop fusion rules (cf. Figure 8c) and performing further partial evaluation, the following expression is derived:

```
build (length M) (fun i ->
  build (length M[0]) (fun j ->
    M[i][j] ) )
```

Left choice combinator

```
normalize(
  buildGet <+
  lengthBuild <+
  letPartialEvaluation <+
  conditionalPartialEvalution <+
  conditionApplication <+
  letApplication <+
  funToLet <+
  letFission <+
  letInitDuplication
).apply(
  fun(M ⇒ matrixTranspose(matrixTranspose(M)))
)
```

ELEVATE
A programming language for program optimizations

This is work in progress.
No evaluation yet, and some open questions and challenges:

- How do we evaluate ELEVATE?

- How do we design a programming interface friendly to systems programmers?

- Can we use ELEVATE to help model stochastic searches in a design space?

- Can we automatically find good ELEVATE programs,
  e.g. using machine learning or program synthesis techniques?

LIFT IS OPEN SOURCE!

University of Glasgow

THE UNIVERSITY of EDINBURGH

WWU MÜNSTER

more info at:

lift-project.org

Paper

Artifacts

Source Code

Naums Mogers · Christof Schlaak · Bastian Hagedorn · Toomas Remmelg · Larisa Stoltzfus · Federico Pizzuti

Bastian Köpcke · Christophe Dubach · Andrej Ivanis · Michel Steuwer · Thomas Koehler · Lu Li

michel.steuwer.info

University of Glasgow

ELEVATE *a language to write composable program optimisations*

Michel Steuwer — *michel.steuwer@glasgow.ac.uk*

www.lift-project.org @LIFTlang

INSPIRING PEOPLE

#UofGWorldChangers

@UofGlasgow