ELEVATE

A Language for Describing Optimization Strategies

1 Wouldn't it be great...

...to ***look behind the curtains*** of optimizing compilers and actually understand how optimizations are applied

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

... to have a ***flexible*** way of specifying optimizations for your rewrite-based compiler

... to use a ***scalable*** approach that hides complexity behind high-level abstractions

Optimizing Programs like it's ~~1998~~ 2019

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

2 Core Concepts

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])
```

Examples

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

...and language-specific: *map(f) ∘ map(g) → map (f ∘ g)*

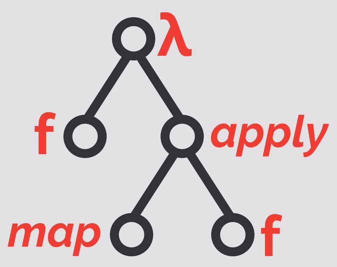
```
def mapFusion: Strategy[LIFT] = p ⇒ match         {
  case map(f) ∘ map(g)  ⇒ Success( map(f ∘ g )
  case _                ⇒ Failure( mapFusion )  }
```

Combinators

```
;seq      : Strategy[P] ⇒ Strategy[P] ⇒ Strategy[P]
          = fs ⇒ ss ⇒ p ⇒ fs(p).flatMapSuccess(ss)
<+choice  : Strategy[P] ⇒ Strategy[P] ⇒ Strategy[P]
          = fs ⇒ ss ⇒ p ⇒ fs(p).flatMapFailure(ss)
try       : Strategy[P] ⇒ Strategy[P]
          = s ⇒ p ⇒ (s <+ id)(p)
repeat    : Strategy[P] ⇒ Strategy[P]
          = s ⇒ p ⇒ try(s ; repeat(s))(p)
```

Traversals: Where to apply a strategy?

λf . map f →



Expression AST

Generic one-level traversals: `Strategy[P] → Strategy[P]`

all: ***one:*** ***some:***

....and language-specific: traversals

```
def body(s: Strategy[LIFT]): Strategy[LIFT] = p ⇒ match {
  case λx.b  ⇒ s(b).mapSuccess(nb ⇒ λx.nb)
  case _     ⇒ Failure( body(s) )                         }
```

Complete Traversals + Normalization

```
topdown : Strategy[P] ⇒ Strategy[P]
        = s ⇒ p ⇒ (s <+ one(topdown(s)))(p)
alltd   : Strategy[P] ⇒ Strategy[P]
        = s ⇒ p ⇒ (s ; all(alltd(s)))(p)
tryAll  : Strategy[P] ⇒ Strategy[P]
        = s ⇒ p ⇒ (all(tryAll(try(s))) ; try(s))(p)
norm    : Strategy[P] ⇒ Strategy[P]
        = s ⇒ p ⇒ repeat(topdown(s))(p)
```

3 Case Studies:

a *Automatic Differentiation*

Efficient Differentiable Programming in a Functional Array-Processing Language

ICFP'19

$\tilde{\mathrm{F}}$ achieves efficiency by rewriting differentiated code

"...the strategy for applying rewrite rules can become tricky."

lenRule = length (build e_0 e_1) ⇝ e_0

↓ ELEVATE

```
def lenRule:Strategy[F̃] = p ⇒ match {
  case length(build(e₀,e₁) ⇒ Success(e₀)
  case _                   ⇒ Failure(lenRule)}
```

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

`norm(lenRule <+ ... )((Mᵀ)ᵀ) == Success(M)`

Tracing shows 12 rule applications

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

b *Image Processing*

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

Separable Convolution: Sobel Filter

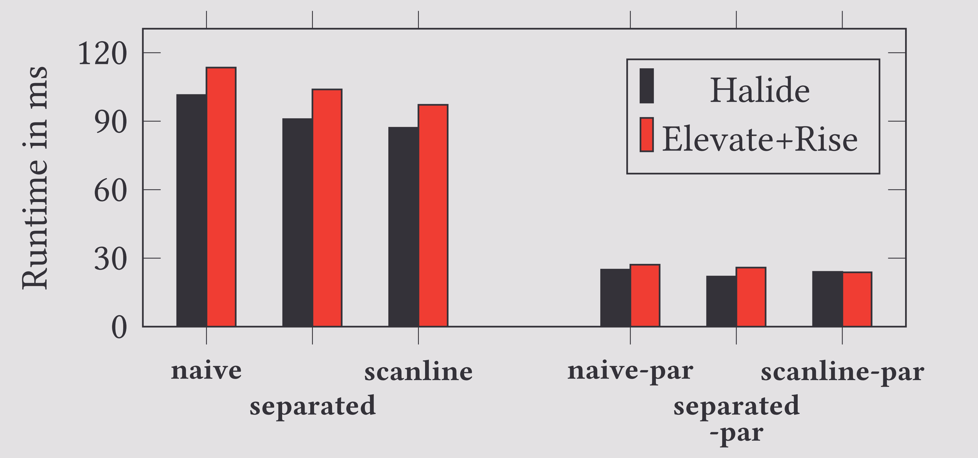
Not expressible as a schedule in Halide!
requires modification of the algorithm instead

```
λw::(3.3.float).λimg::(N.M.float).img ▷
pad2D(1) ▷ slide2D(3)(1) ▷ map2D(
  λnbh.dot(join(w))(join(nbh))
))))
```

2D Convolution in Rise (LIFT-like language)

```
(topdown(separateDot) ; lowerToC)(conv)
```

Separating Convolution using ELEVATE



Extensible: ELEVATE can be extended with custom domain-specific optimizations

c *Deep Learning*

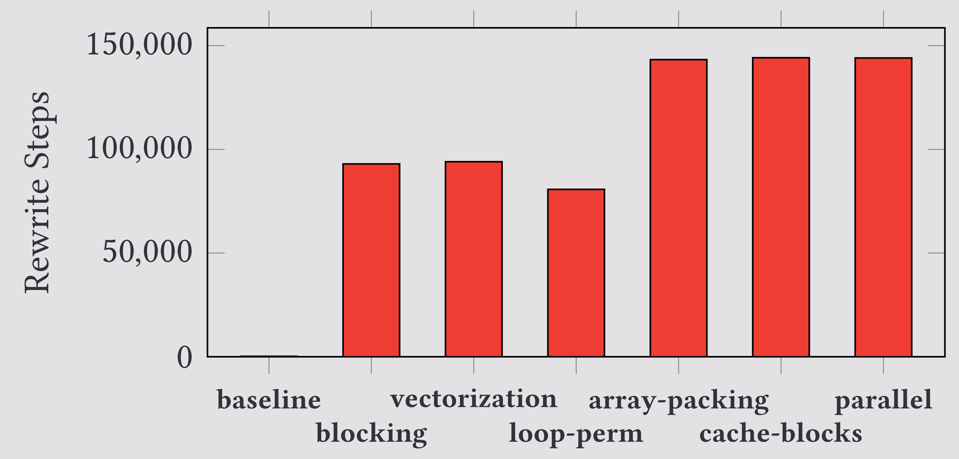
Implementing TVM's scheduling language:

```
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)
```

Blocking Schedule for Matrix Multiplication in TVM

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

Blocking Strategy for Matrix Multiplication in ELEVATE



Scalable: ELEVATE hides 100k's of rewrite steps behind high-level abstractions

Bastian Hagedorn University of Münster | **Johannes Lenfers** University of Münster | **Thomas Koehler** University of Glasgow | **Sergei Gorlatch** University of Münster | **Michel Steuwer** University of Glasgow

b.hagedorn@wwu.de | bastianhagedorn.github.io