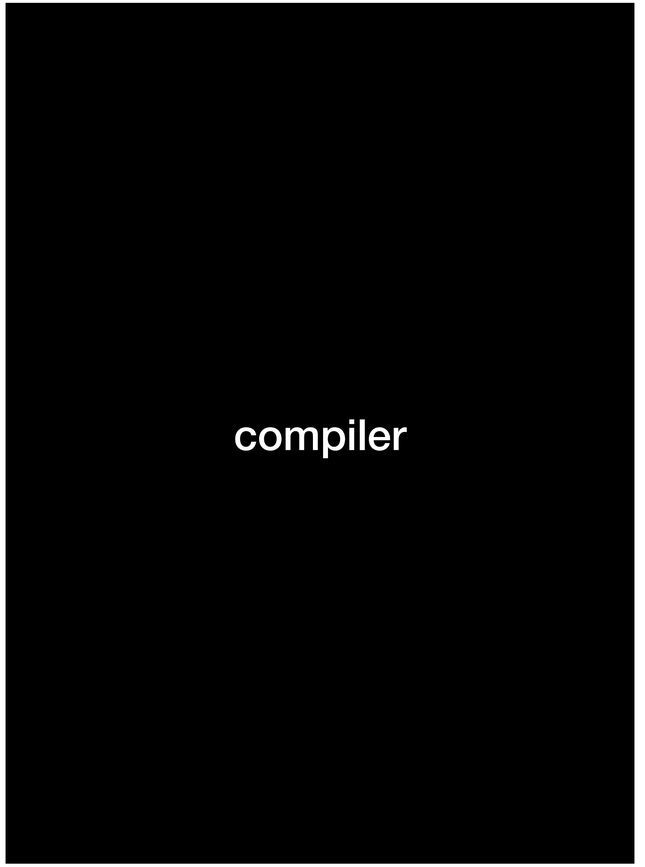
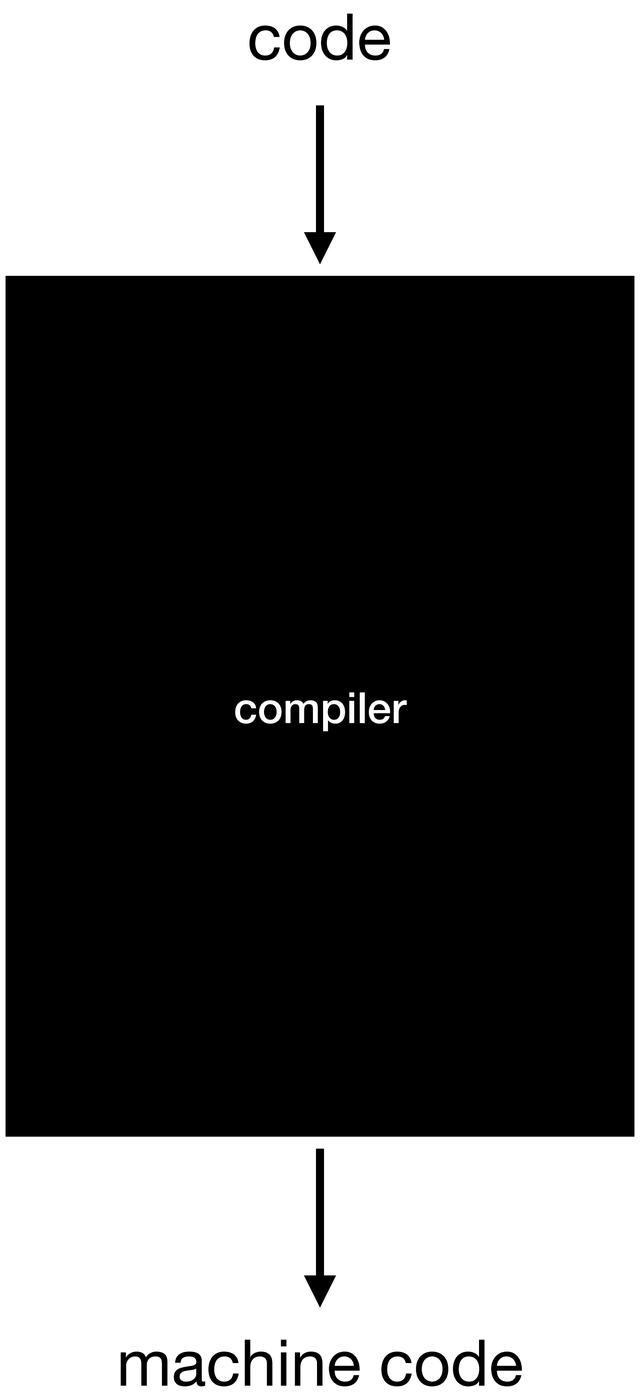
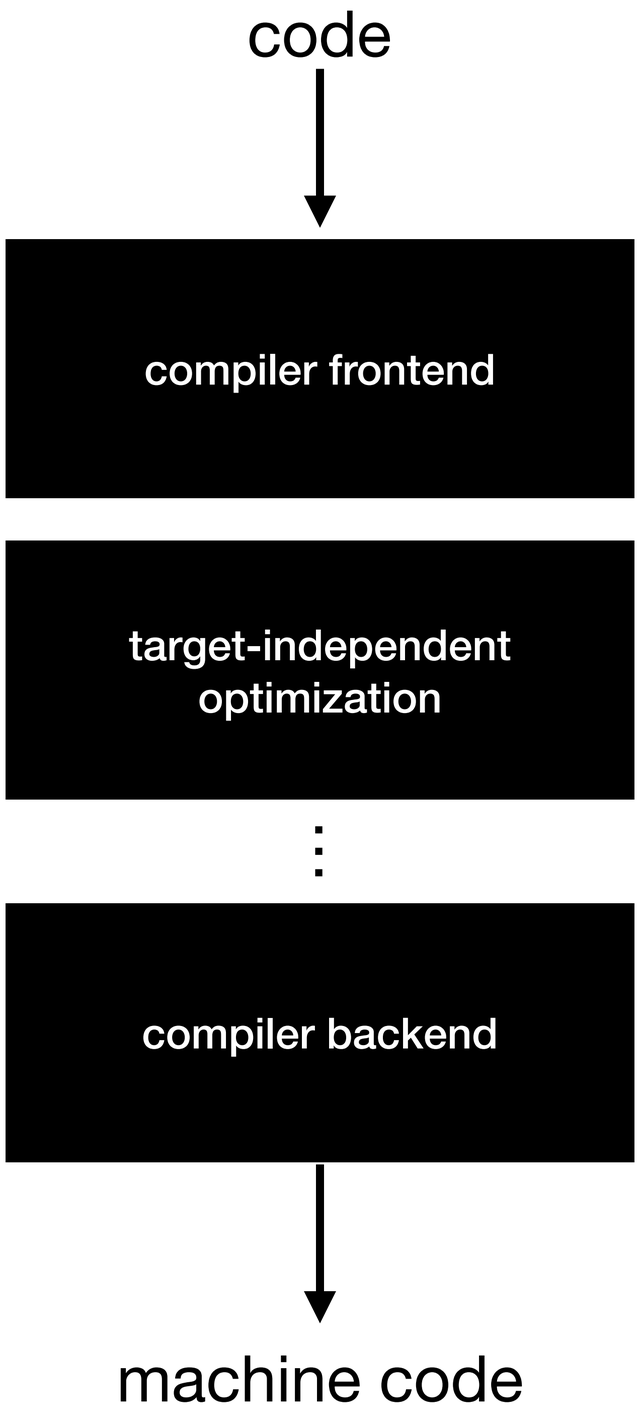
Generating Compiler Backends from Formal Models of Hardware

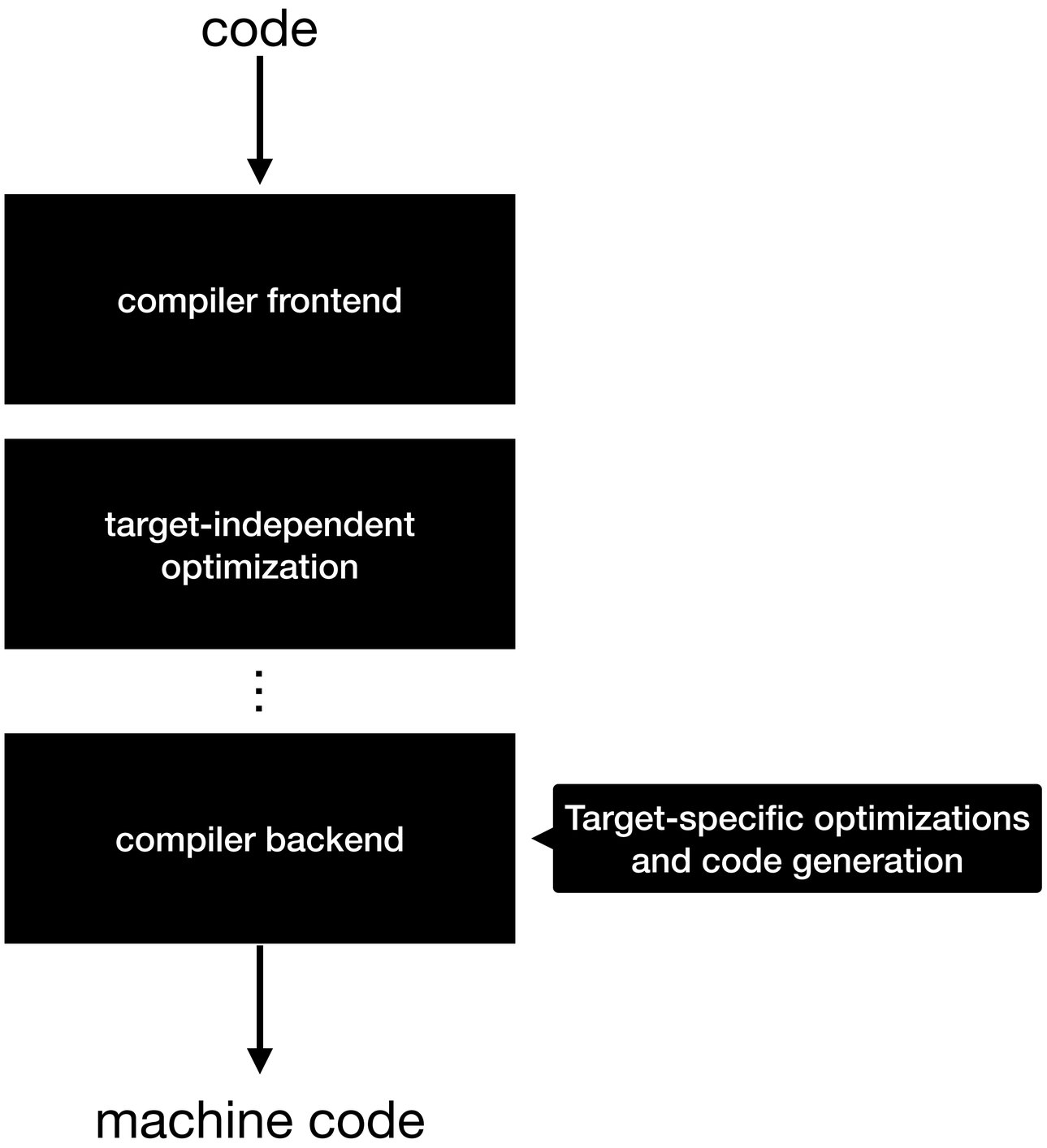
Gus Smith's Generals Exam

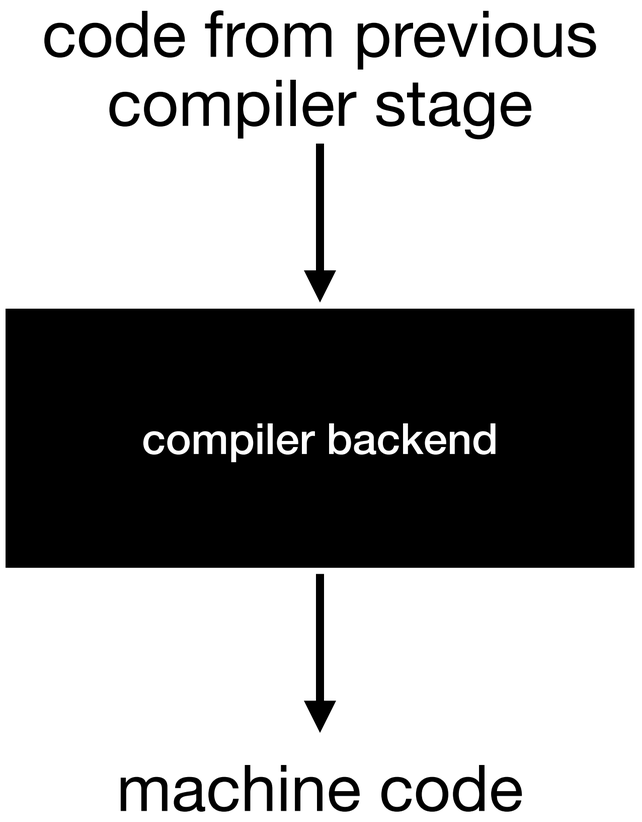
May 6th, 2022

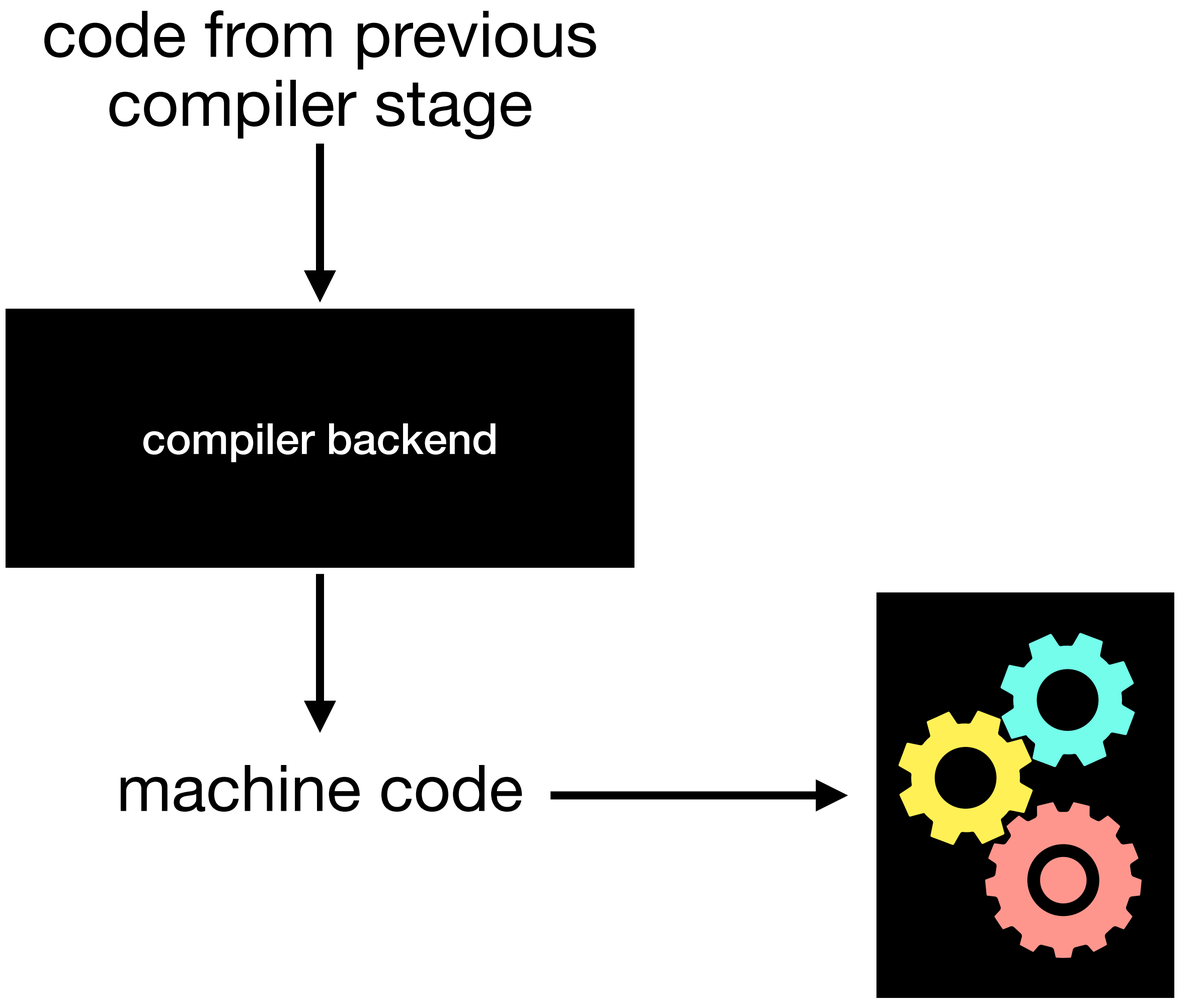

compiler

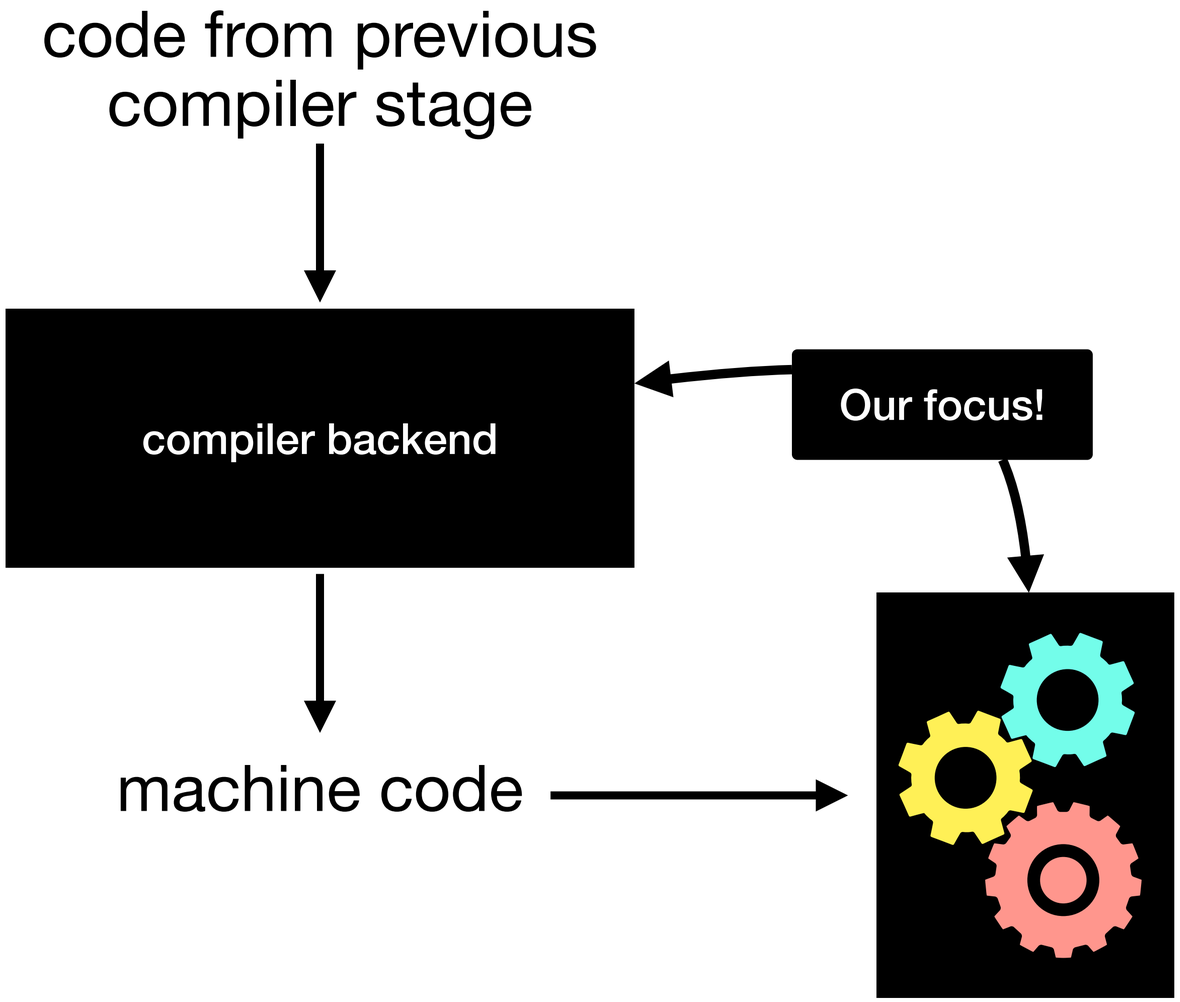

code
compiler
machine code

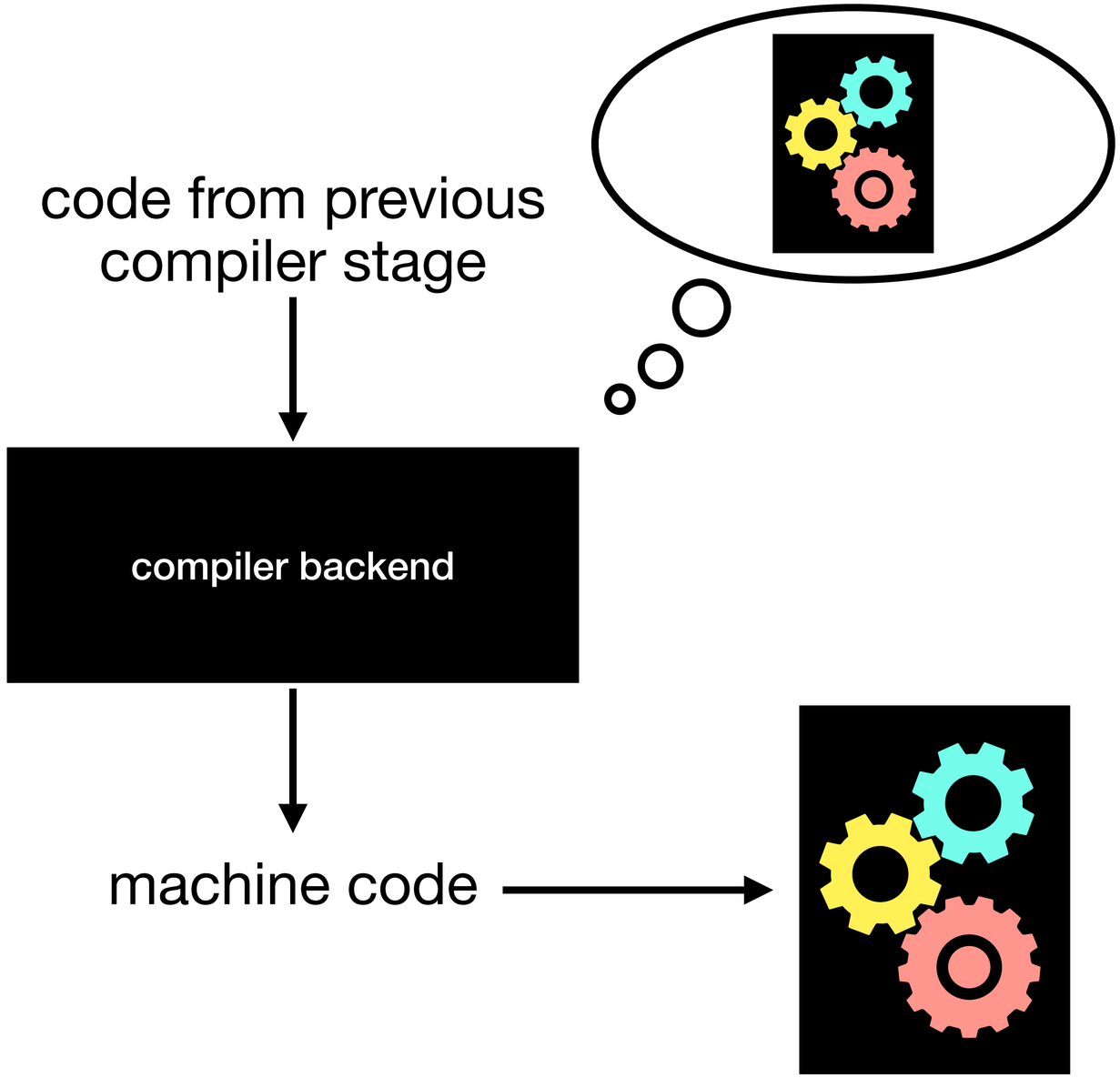

code
compiler frontend
target-independent
optimization
compiler backend
machine code

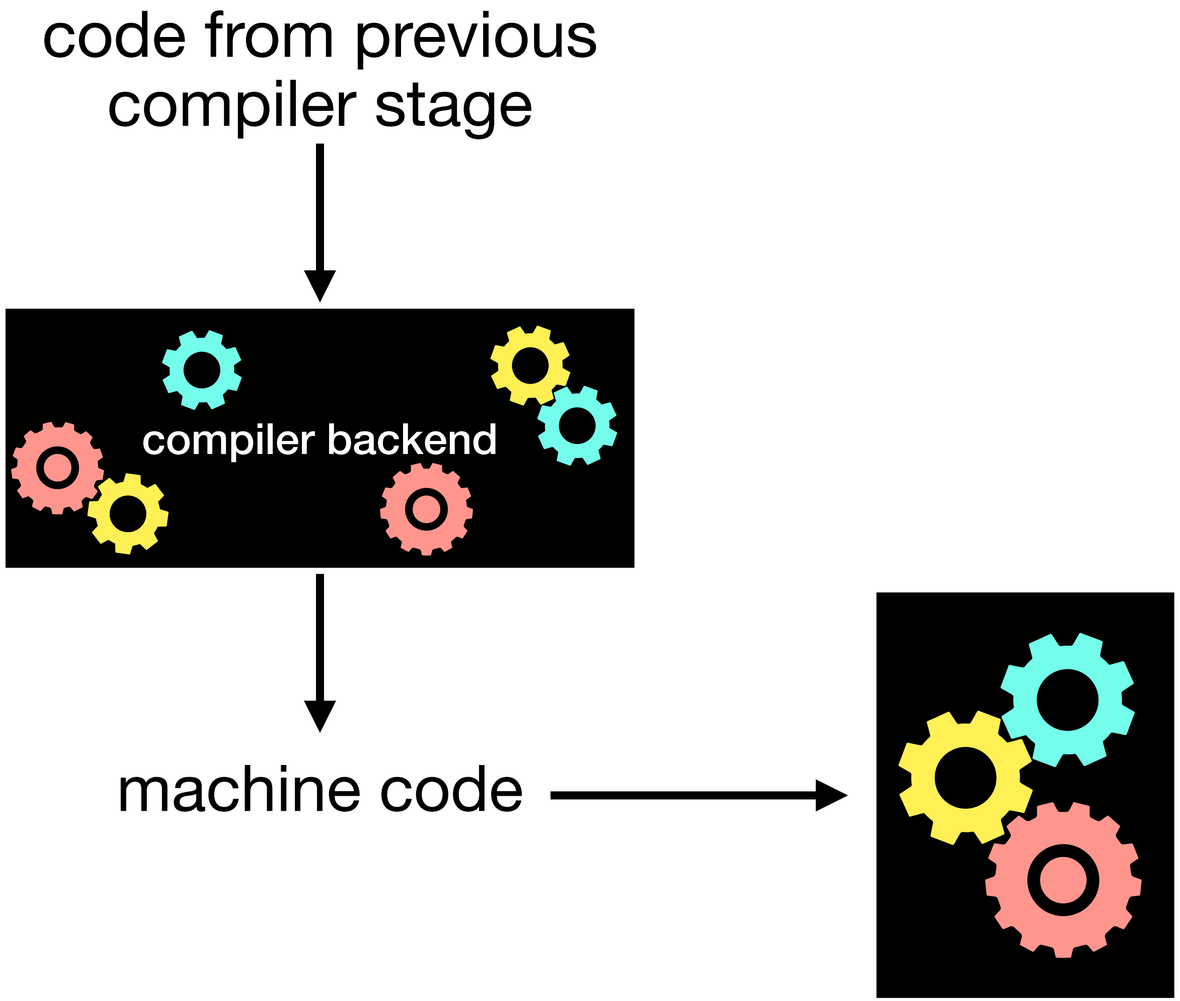

code
compiler frontend
target-independent optimization
compiler backend
Target-specific optimizations and code generation
machine code

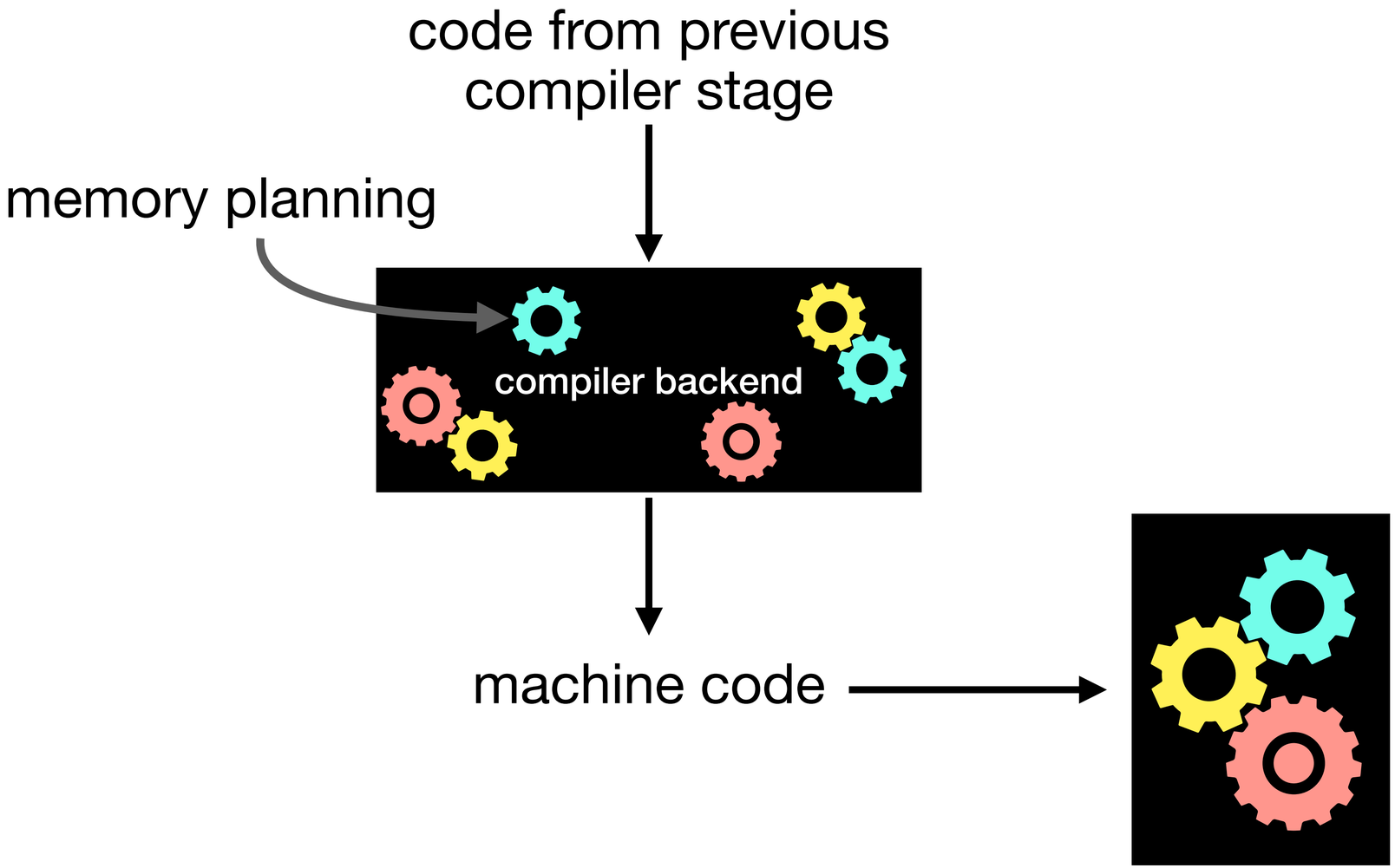

code from previous
compiler stage
compiler backend
machine code

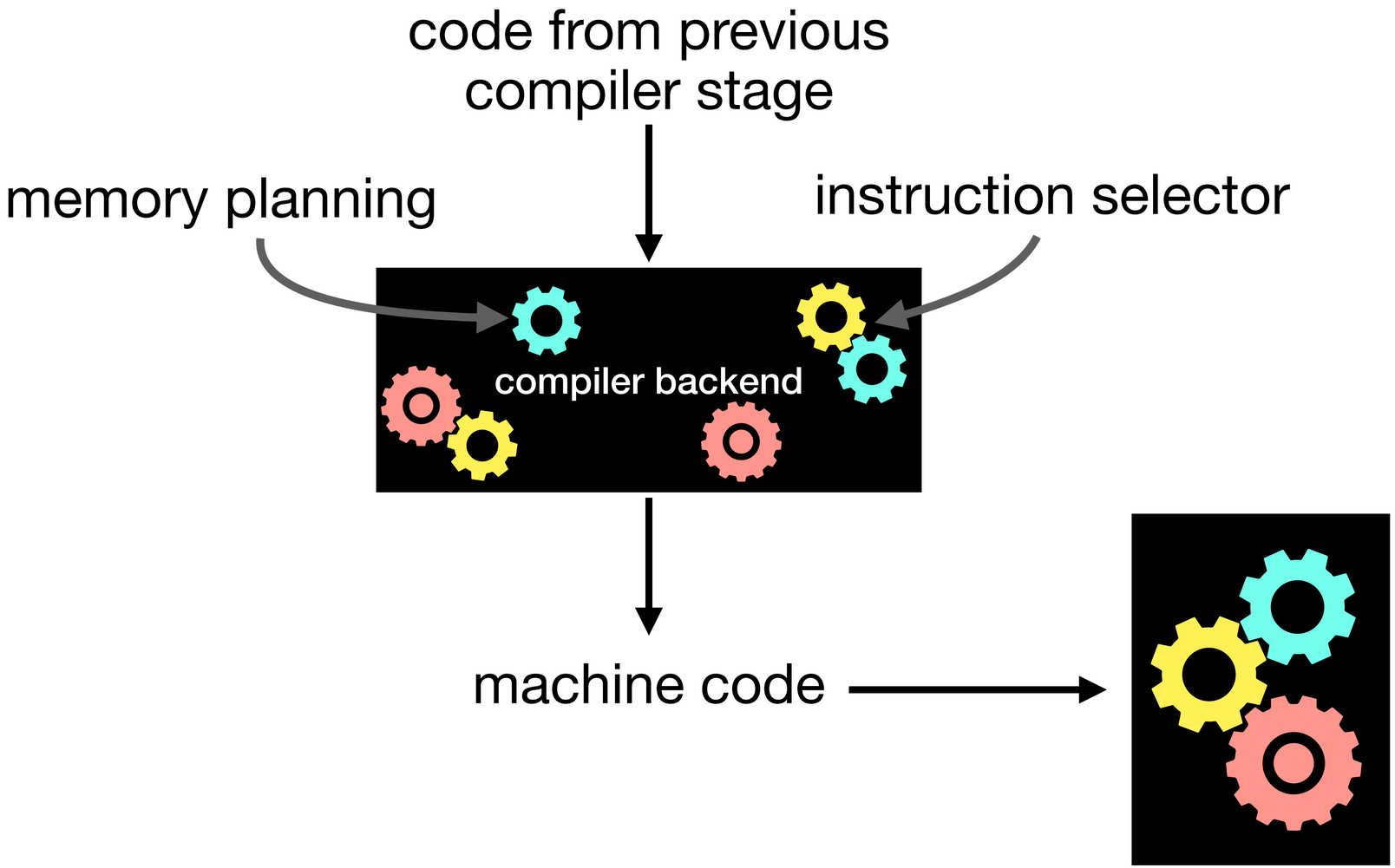

code from previous
compiler stage
compiler backend
machine code

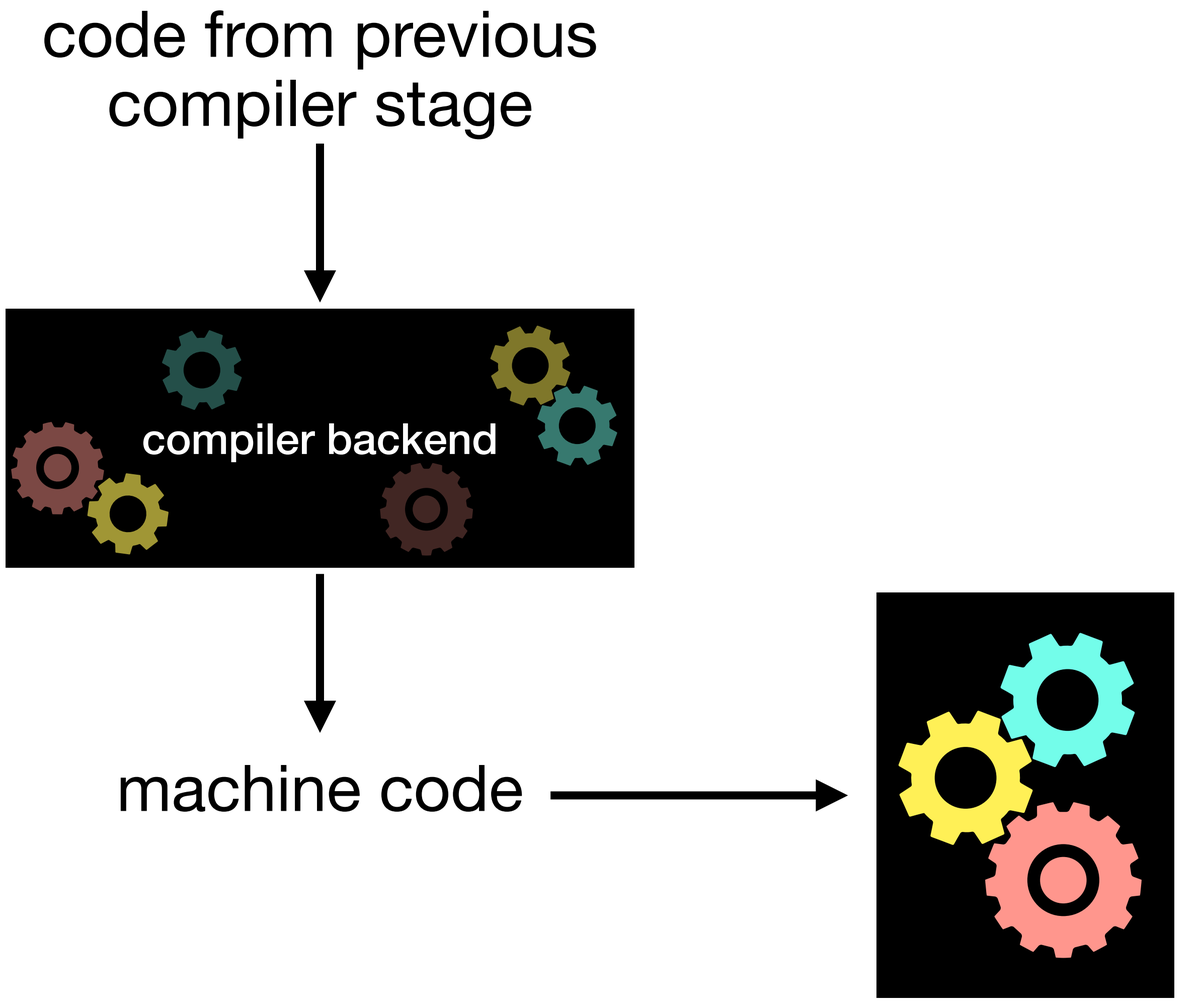

code from previous
compiler stage
compiler backend
Our focus!
machine code


code from previous
compiler stage
compiler backend
machine code


code from previous
compiler stage
compiler backend
machine code


code from previous
compiler stage
memory planning
compiler backend
machine code


code from previous
compiler stage
memory planning
instruction selector
compiler backend
machine code


code from previous
compiler stage
compiler backend
machine code

An Empirical Study of the Effect of Source-Level Loop Transformations on Compiler Stability

ZHANGXIAOWEN GONG, University of Illinois at Urbana-Champaign, USA
ZHI CHEN, University of California, Irvine, USA
JUSTIN SZADAY, University of Illinois at Urbana-Champaign, USA
DAVID WONG, Intel, USA
ZEHRA SURA, IBM, USA
NEFTALI WATKINSON, University of California, Irvine, USA
SAEED MALEKI, Microsoft, USA
DAVID PADUA, University of Illinois at Urbana-Champaign, USA
ALEXANDER VEIDENBAUM, University of California, Irvine, USA
ALEXANDRU NICOLAU, University of California, Irvine, USA
JOSEP TORRELLAS, University of Illinois at Urbana-Champaign, USA

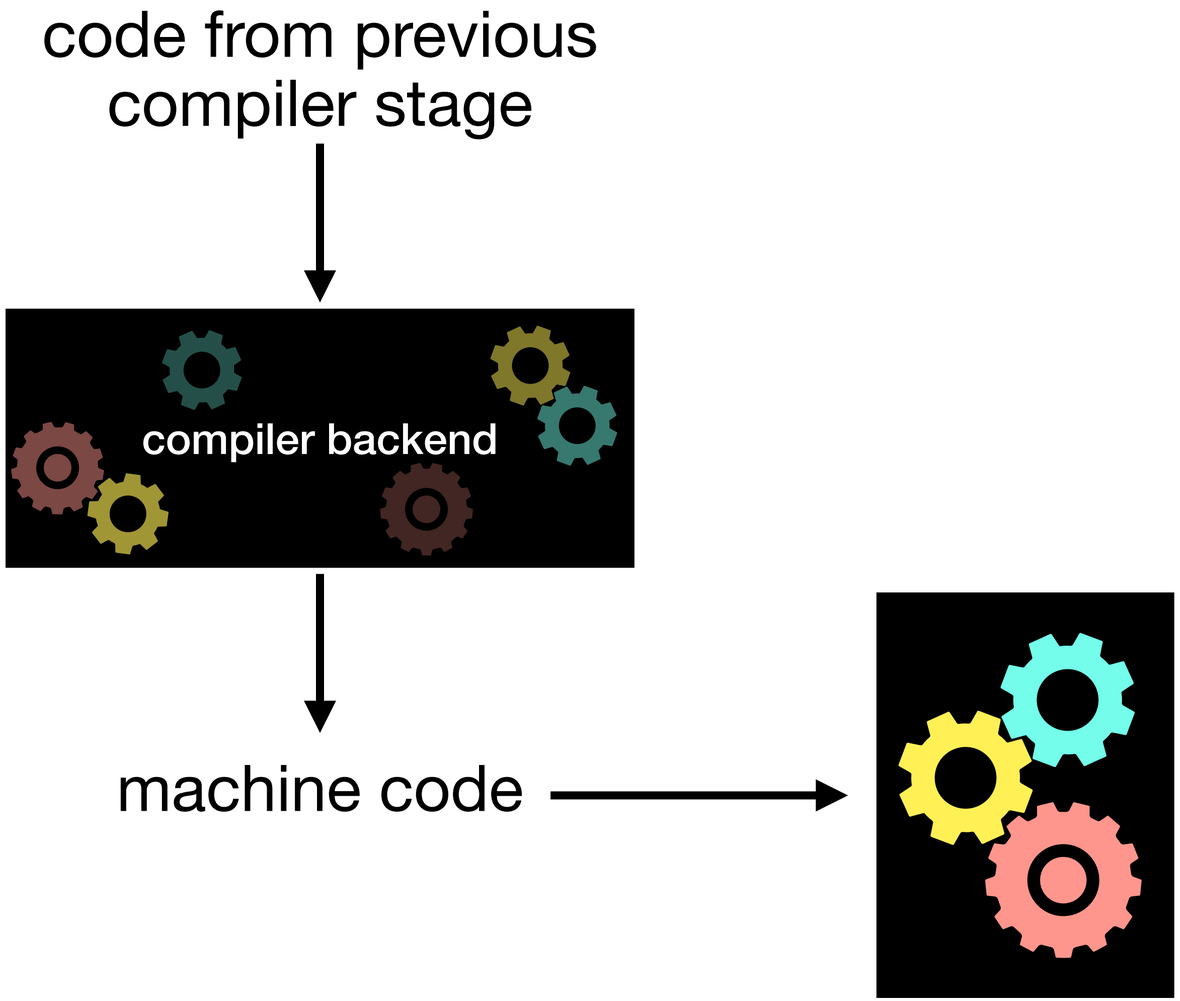
Modern compiler optimization is a complex process that offers no guarantees to deliver the fastest, most efficient target code. For this reason, compilers struggle to produce a stable performance from versions of code that carry out the same computation and only differ in the order of operations. This instability makes compilers much less effective program optimization tools and often forces programmers to carry out a brute

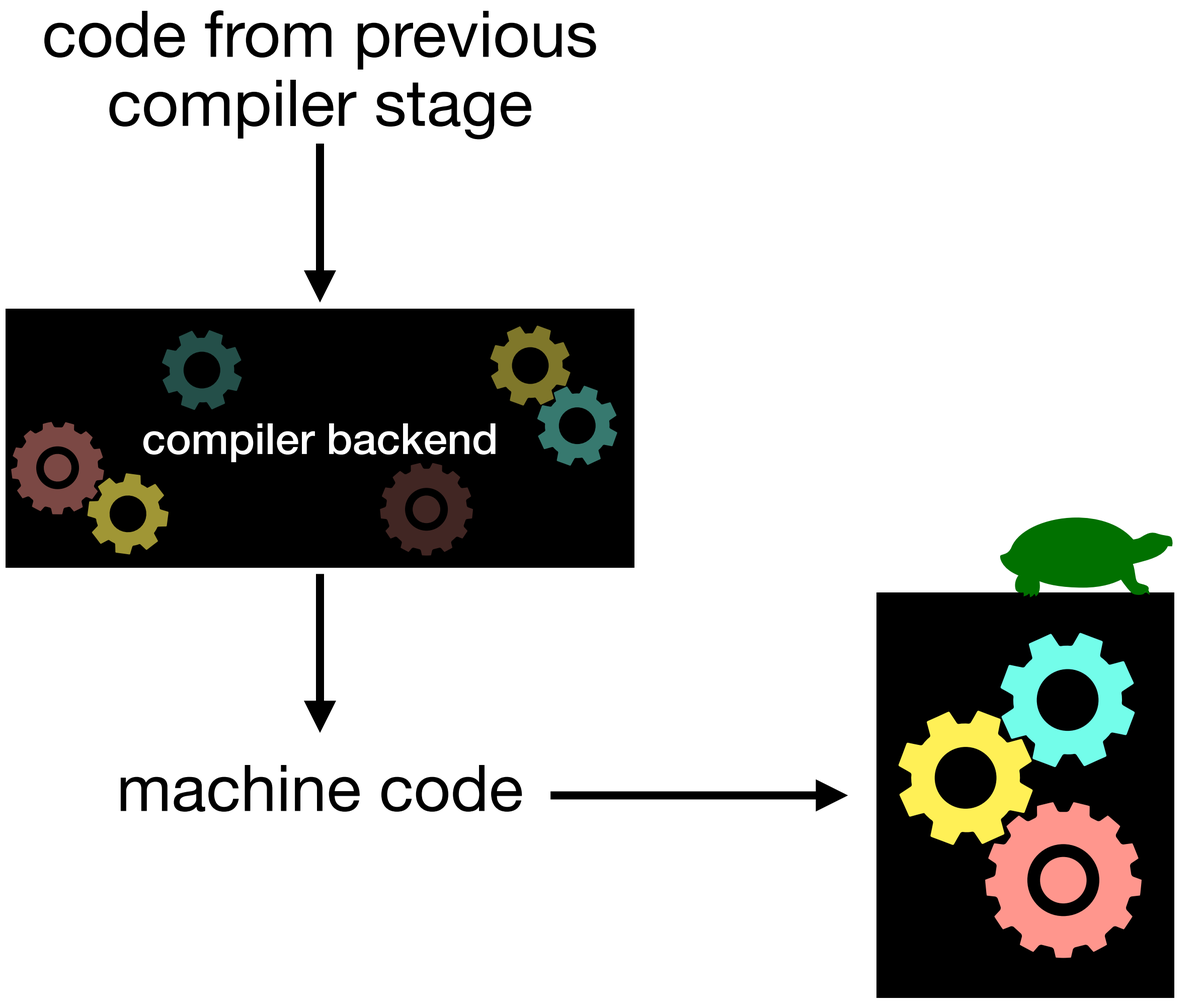
An Empirical Study of the Effect of Source-Level Loop Transformations on Compiler Stability

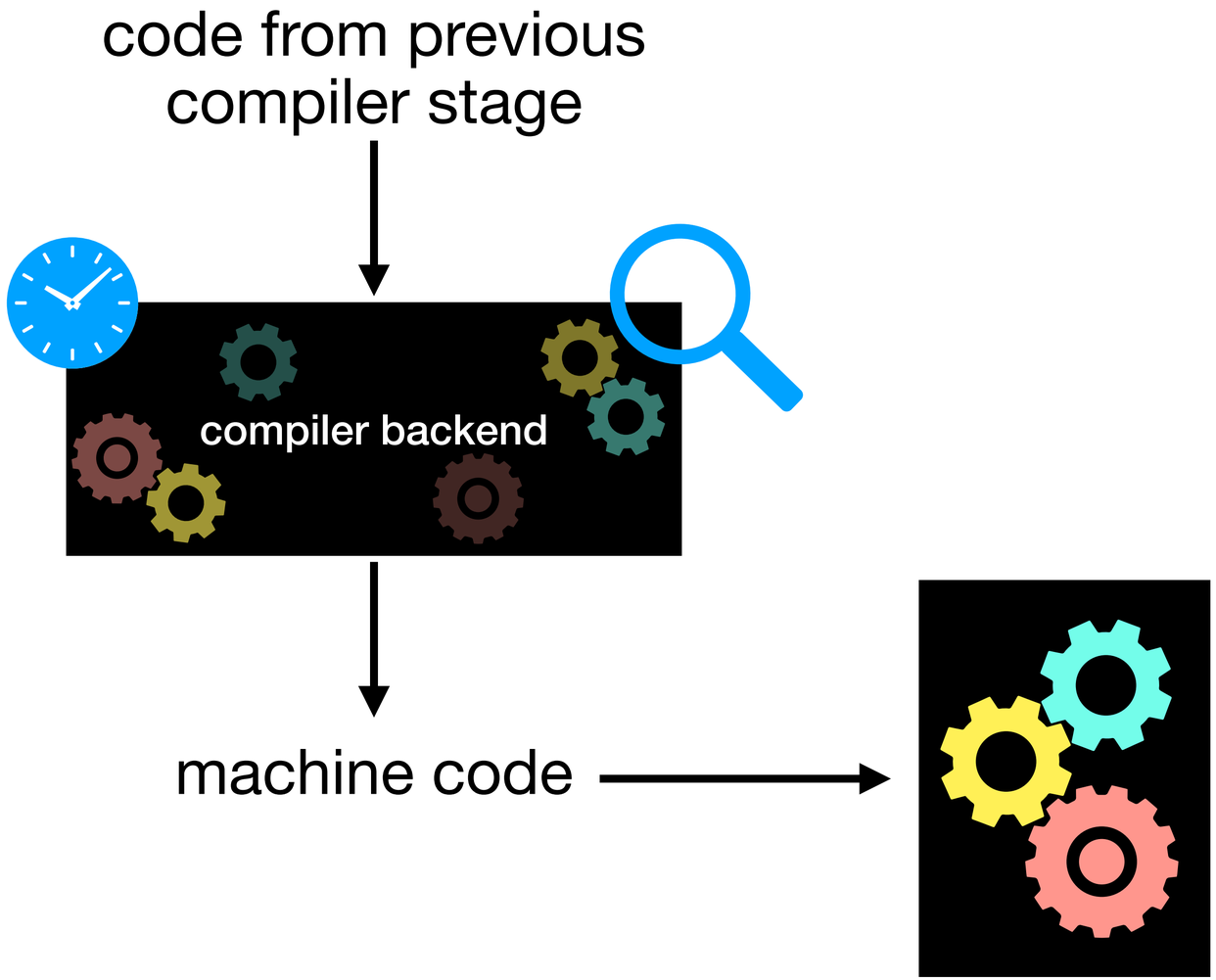
ZHANGXIAOWEN GONG, University of Illinois at Urbana-Champaign, USA
Z
J
DAVID WONG, Intel, USA
ZEHRA SURA, IBM, USA
NEFTALI WATKINSON, University of California, Irvine, USA
SAEED MALEKI, Microsoft, USA
DAVID PADUA, University of Illinois at Urbana-Champaign, USA
ALEXANDER VEIDENBAUM, University of California, Irvine, USA
ALEXANDRU NICOLAU, University of California, Irvine, USA
JOSEP TORRELLAS, University of Illinois at Urbana-Champaign, USA

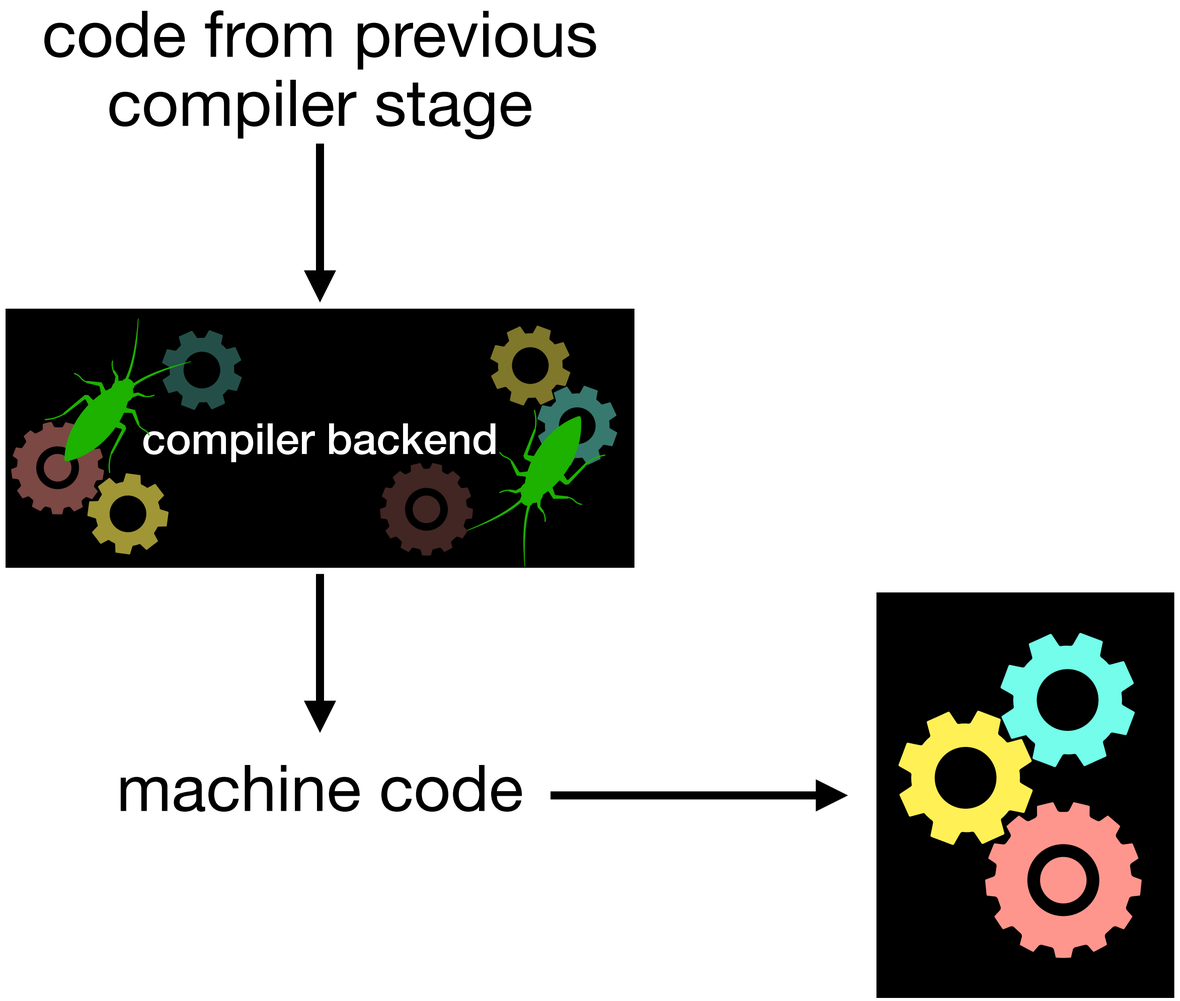
Inaccurate vectorization models are a primary source of low-quality compiler results!

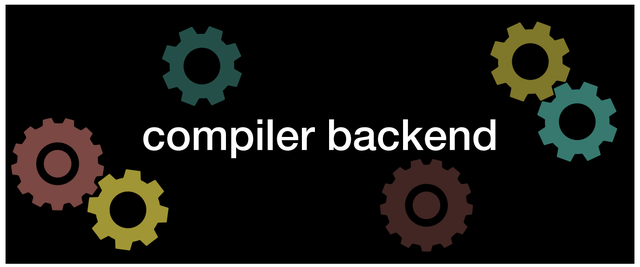
Modern compiler optimization is a complex process that offers no guarantees to deliver the fastest, most efficient target code. For this reason, compilers struggle to produce a stable performance from versions of code that carry out the same computation and only differ in the order of operations. This instability makes compilers much less effective program optimization tools and often forces programmers to carry out a brute


code from previous
compiler stage
compiler backend
machine code


code from previous
compiler stage
compiler backend
machine code


code from previous
compiler stage
compiler backend
machine code


code from previous
compiler stage
compiler backend
machine code



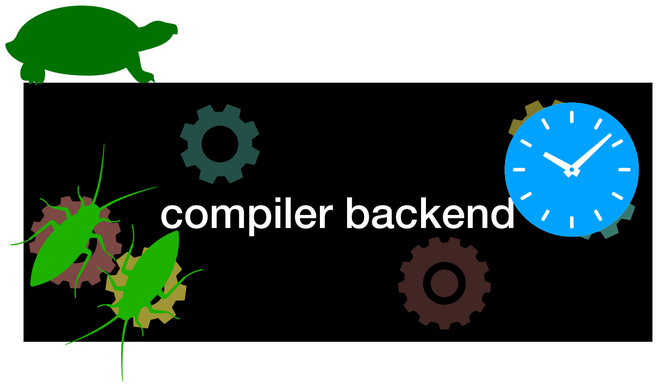
Implicit hardware models in compiler backends are potential sources of

Implicit hardware models in compiler backends are potential sources of imperfect optimization,

Implicit hardware models in compiler backends are potential sources of

imperfect optimization,

difficulties in development,



Implicit hardware models in compiler backends are potential sources of

imperfect optimization,

difficulties in development,

and hard-to-find bugs!

This leads directly to my thesis!

Automatically generating compiler backends from explicit, formal hardware models

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

optimizer *discovers* optimizations that are not explicitly programmed

Automatically Generating Instruction Selectors Using Declarative Machine Descriptions

João Dias
Tufts University
dias@cs.tufts.edu

Norman Ramsey
Tufts University
nr@cs.tufts.edu

Abstract

Despite years of work on retargetable compilers, creating a good, reliable back end for an optimizing compiler still entails a lot of hard work. Moreover, a critical component of the back end—the instruction selector—must be written by a person who is expert in both the compiler's intermediate code and the target machine's instruction set. By *generating* the instruction selector from declarative machine descriptions we have (a) made it unnecessary for one person to be both a compiler expert and a machine expert, and (b) made creating an optimizing back end easier than ever before.

Our achievement rests on two new results. First, finding a mapping from intermediate code to machine code is an undecidable problem. Second, using heuristic search, we can find mappings for machines of practical interest in at most a few minutes of CPU time.

Our most significant new idea is that heuristic search should be controlled by algebraic laws. Laws are used not only to show when a sequence of instructions implements part of an intermediate code, but also to limit the search: we drop a sequence of instructions not when it gets too long or when it computes too complicated a result, but when *too much reasoning* will be required to show that the result computed might be useful.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation; D.3.4 [*Processors*]: Retargetable compilers

General Terms Algorithms, Experimentation, Theory

technique, an instruction selector is generated automatically from *declarative machine descriptions*. (A declarative machine description contains no code and no information about any compiler's data structures; instead, it simply and formally describes properties of a target machine.)

Our contributions are as follows:

- We show that given a description of an arbitrary instruction set, generating an instruction selector is undecidable (Section 8). To find machine instructions that implement intermediate code, it is therefore necessary to search heuristically.
- We present a new heuristic-search algorithm, which starts with the expressions computed by the machine's instruction set and gradually adds to a pool of computable expressions until every intermediate-code expression is computable.

  A crucial invariant is that we consider *only* computations that we *know* can be implemented entirely by machine instructions. This invariant makes our algorithm significantly simpler than earlier search algorithms, which start with goal computations whose implementations by machine instructions are not known.
- To increase the pool of computable expressions, we rewrite existing computable expressions using algebraic laws. To match the left-hand side of an algebraic law, we have developed a new algorithm called *establishment*, which uses a novel combination of unification and machine code to make two expressions equal (Section 7, especially Figure 4).

Machine Descriptions to Build Tools for Embedded Systems

Norman Ramsey and Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22903
nr@cs.virginia.edu jwd@cs.virginia.edu

Abstract. Because of poor tools, developing embedded systems can be unnecessarily hard. Machine descriptions based on register-transfer lists (RTLs) have proven useful in building retargetable compilers, but not in building other retargetable tools. Simulators, assemblers, linkers, debuggers, and profilers are built by hand if at all—previous machine descriptions have lacked the detail and precision needed to generate them. This paper presents detailed and precise machine-description techniques that are based on a new formalization of RTLs. Unlike previous notations, these RTLs have a detailed, unambiguous, and machine-independent semantics, which makes them ideal for supporting automatic generation of retargetable tools. The paper also gives examples of λ-RTL, a notation that makes it possible for human beings to read and write RTLs without becoming overwhelmed by machine-dependent detail.

chine Descriptions for Machine-Level Tools

lopers for embedded systems often work without the benefit of the best vare tools. Embedded systems can have unusual architectural features, and processors can be introduced rapidly. Development is typically done on t processors, and cross-development can make it hard to get basic compil-

Automatically Generating Instruction Selectors Using Declarative Machine Descriptions

João Dias
Tufts University
dias@cs.tufts.edu

Norman Ramsey

Unlike Ramsey, we will focus not on CPUs, but on fixed-function accelerators and programmable hardware!

Abstract

Despite years of work on retargetable compilers, creating a good, reliable back end for an optimizing compiler still entails a lot of hard work. Moreover, a critical component of the back end—the instruction selector—must be written by a person who is expert in both the compiler's intermediate code and the target machine's instruction set. By *generating* the instruction selector from declarative machine descriptions we have (a) made it unnecessary for one person to be both a compiler expert and a machine expert, and (b) made creating an optimizing back end easier than ever before.

Our achievement rests on two new results. First, finding a mapping from intermediate code to machine code is an undecidable problem. Second, using heuristic search, we can find mappings for machines of practical interest in at most a few minutes of CPU time.

Our most significant new idea is that heuristic search should be controlled by algebraic laws. Laws are used not only to show when a sequence of instructions implements part of an intermediate code, but also to limit the search: we drop a sequence of instructions not when it gets too long or when it computes too complicated a result, but when *too much reasoning* will be required to show that the result computed might be useful.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation; D.3.4 [*Processors*]: Retargetable compilers

General Terms Algorithms, Experimentation, Theory

technique, an instruction selector is generated automatically from *declarative machine descriptions*. (A declarative machine description contains no code and no information about any compiler's data structures; instead, it simply and formally describes properties of a target machine.)

Our contributions are as follows:

- We show that given a description of an arbitrary instruction set, generating an instruction selector is undecidable (Section 8). To find machine instructions that implement intermediate code, it is therefore necessary to search heuristically.
- We present a new heuristic-search algorithm, which starts with the expressions computed by the machine's instruction set and gradually adds to a pool of computable expressions until every intermediate-code expression is computable.

  A crucial invariant is that we consider *only* computations that we *know* can be implemented entirely by machine instructions. This invariant makes our algorithm significantly simpler than earlier search algorithms, which start with goal computations whose implementations by machine instructions are not known.
- To increase the pool of computable expressions, we rewrite existing computable expressions using algebraic laws. To match the left-hand side of an algebraic law, we have developed a new algorithm called *establishment*, which uses a novel combination of unification and machine code to make two expressions equal (Section 7, especially Figure 4).

Machine Descriptions to Build Tools for Embedded Systems

Norman Ramsey and Jack W. Davidson

Department of Computer Science
University of Virginia
esville, VA 22903
u jwd@cs.virginia.edu

Abstract. Because of poor tools, developing embedded systems can be unnecessarily hard. Machine descriptions based on register-transfer lists (RTLs) have proven useful in building retargetable compilers, but not in building other retargetable tools. Simulators, assemblers, linkers, debuggers, and profilers are built by hand if at all—previous machine descriptions have lacked the detail and precision needed to generate them. This paper presents detailed and precise machine-description techniques that are based on a new formalization of RTLs. Unlike previous notations, these RTLs have a detailed, unambiguous, and machine-independent semantics, which makes them ideal for supporting automatic generation of retargetable tools. The paper also gives examples of λ-RTL, a notation that makes it possible for human beings to read and write RTLs without becoming overwhelmed by machine-dependent detail.

chine Descriptions for Machine-Level Tools

elopers for embedded systems often work without the benefit of the best vare tools. Embedded systems can have unusual architectural features, and processors can be introduced rapidly. Development is typically done on t processors, and cross-development can make it hard to get basic compil-

Structure of this talk

What I’ll show

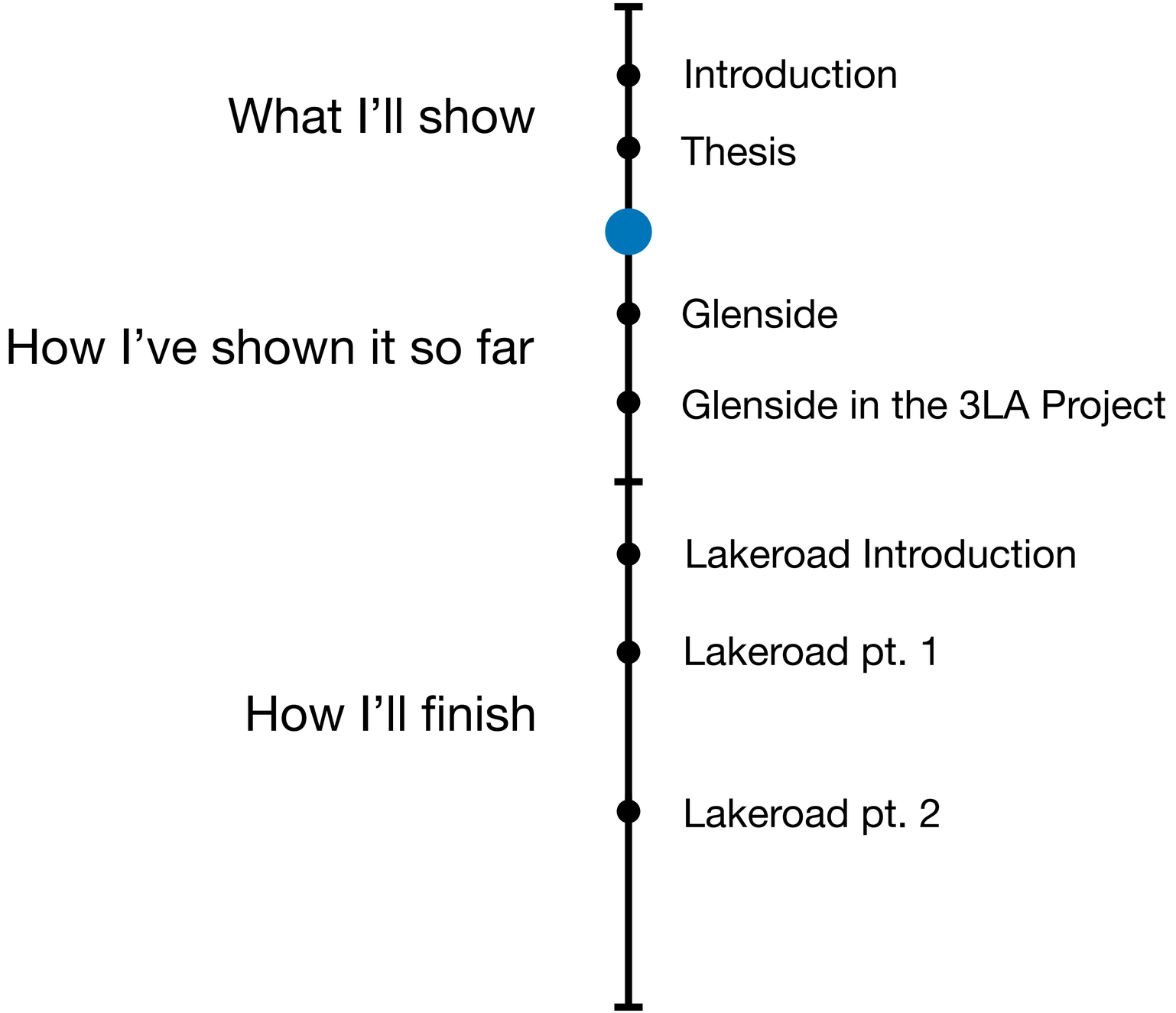
- Introduction
- Thesis

How I’ve shown it so far

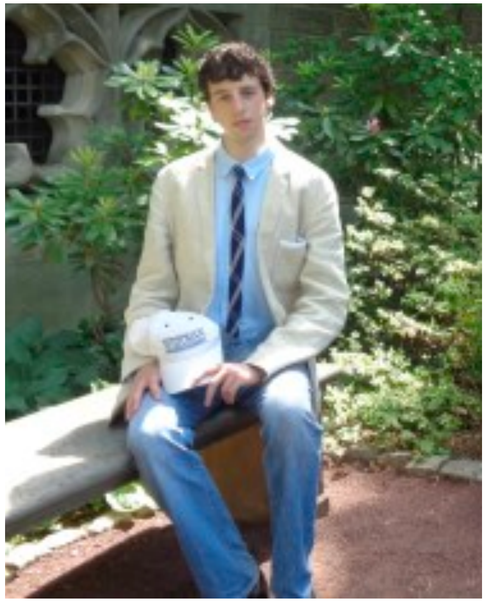
- Glenside
- Glenside in the 3LA Project

How I’ll finish

- Lakeroad Introduction
- Lakeroad pt. 1
- Lakeroad pt. 2


What I’ll show
Introduction
Thesis
How I’ve shown it so far
Glenside
Glenside in the 3LA Project
Lakeroad Introduction
Lakeroad pt. 1
How I’ll finish
Lakeroad pt. 2

Glenside




Gus Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock.
"Pure tensor program rewriting via access patterns (representation pearl).” MAPS 2021.

Glenside is a tensor IR* built for equality saturation.

* intermediate representation

Glenside is a tensor IR* built for equality saturation.

Glenside enables users to model hardware accelerators as program rewrites.

* intermediate representation

Glenside is a tensor IR* built for equality saturation.

Glenside enables users to model hardware accelerators as program rewrites.

These rewrites, in concert with Glenside's built-in rewrites, automatically discover ways to map machine learning workloads to accelerators.

* intermediate representation

Three design requirements for Glenside:

Three design requirements for Glenside:

1. The language must be **pure**—a necessary requirement for equality saturation.

Three design requirements for Glenside:

1. The language must be **pure**—a necessary requirement for equality saturation.

2. The language must be **low-level,** letting us reason about hardware.

Three design requirements for Glenside:

1. The language must be **pure**—a necessary requirement for equality saturation.

2. The language must be **low-level,** letting us reason about hardware.

3. The language must **not use binding,** making term rewriting much easier.

Let’s begin with an example: matrix multiplication!

We want to represent matrix multiplication in a way that

We want to represent matrix multiplication in a way that

1. is pure,

We want to represent matrix multiplication in a way that

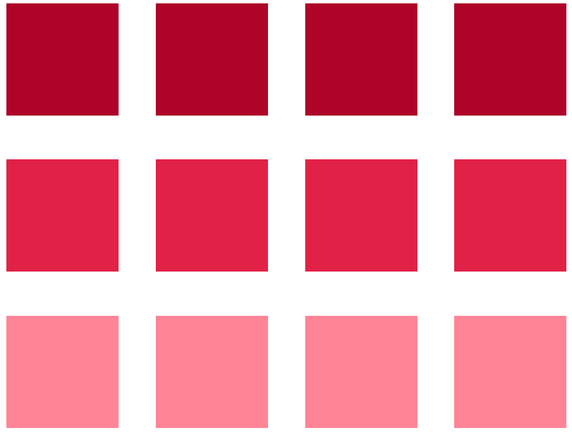
1. is pure,

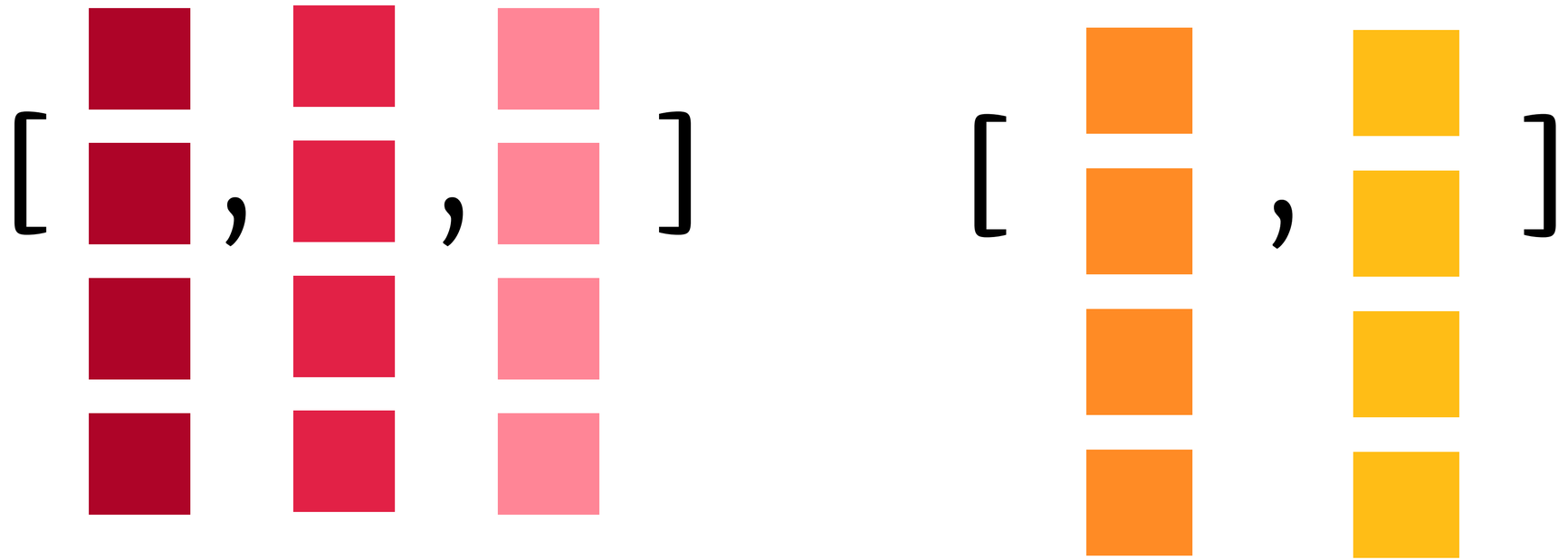
2. is low-level, and

We want to represent matrix multiplication in a way that

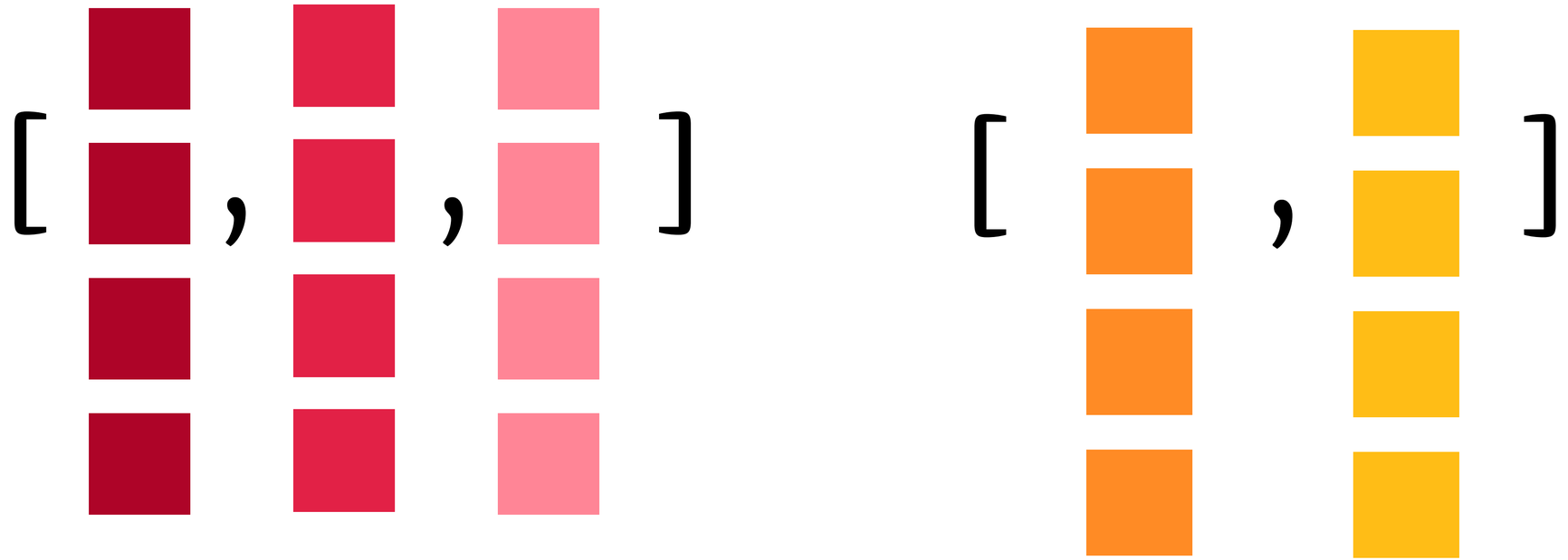
1. is pure,
2. is low-level, and
3. avoids binding.

Given matrices A and B, pair each row of A with each column of B, compute their dot products, and arrange the results back into a matrix.

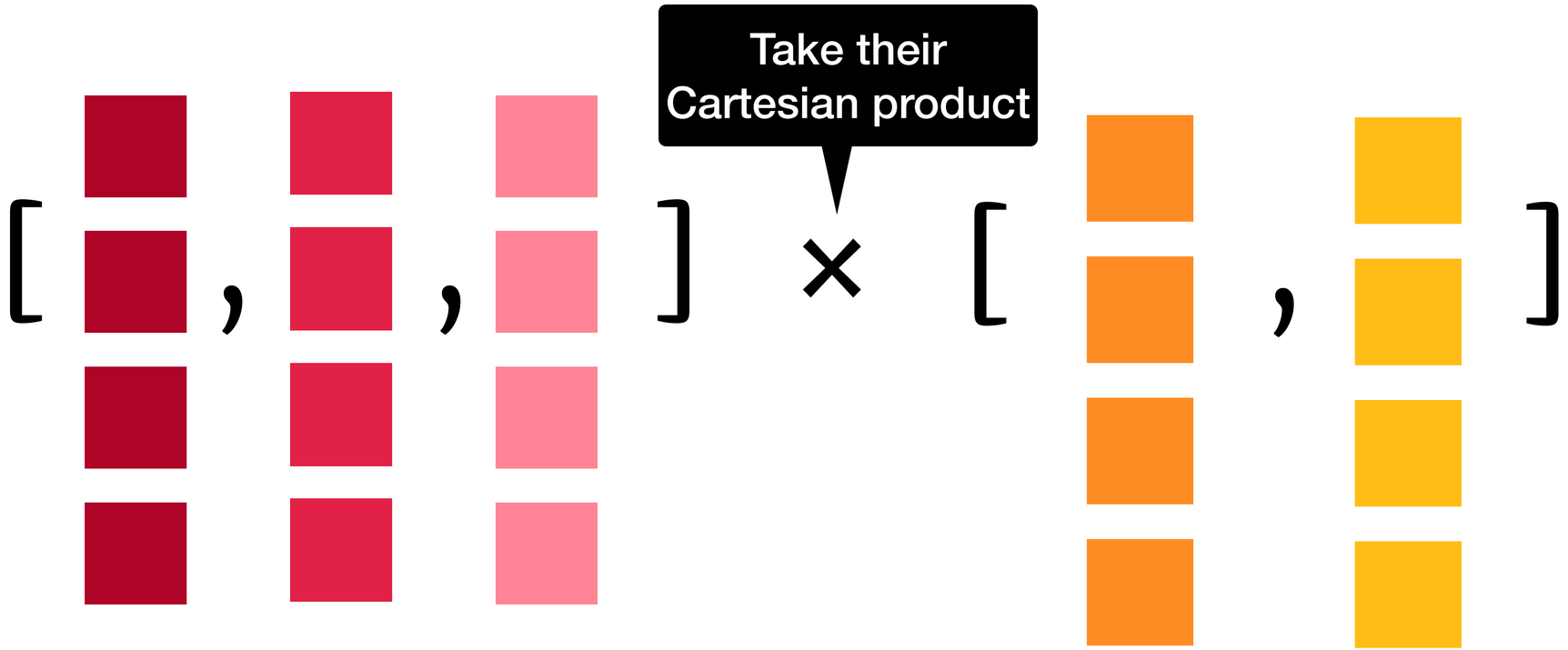


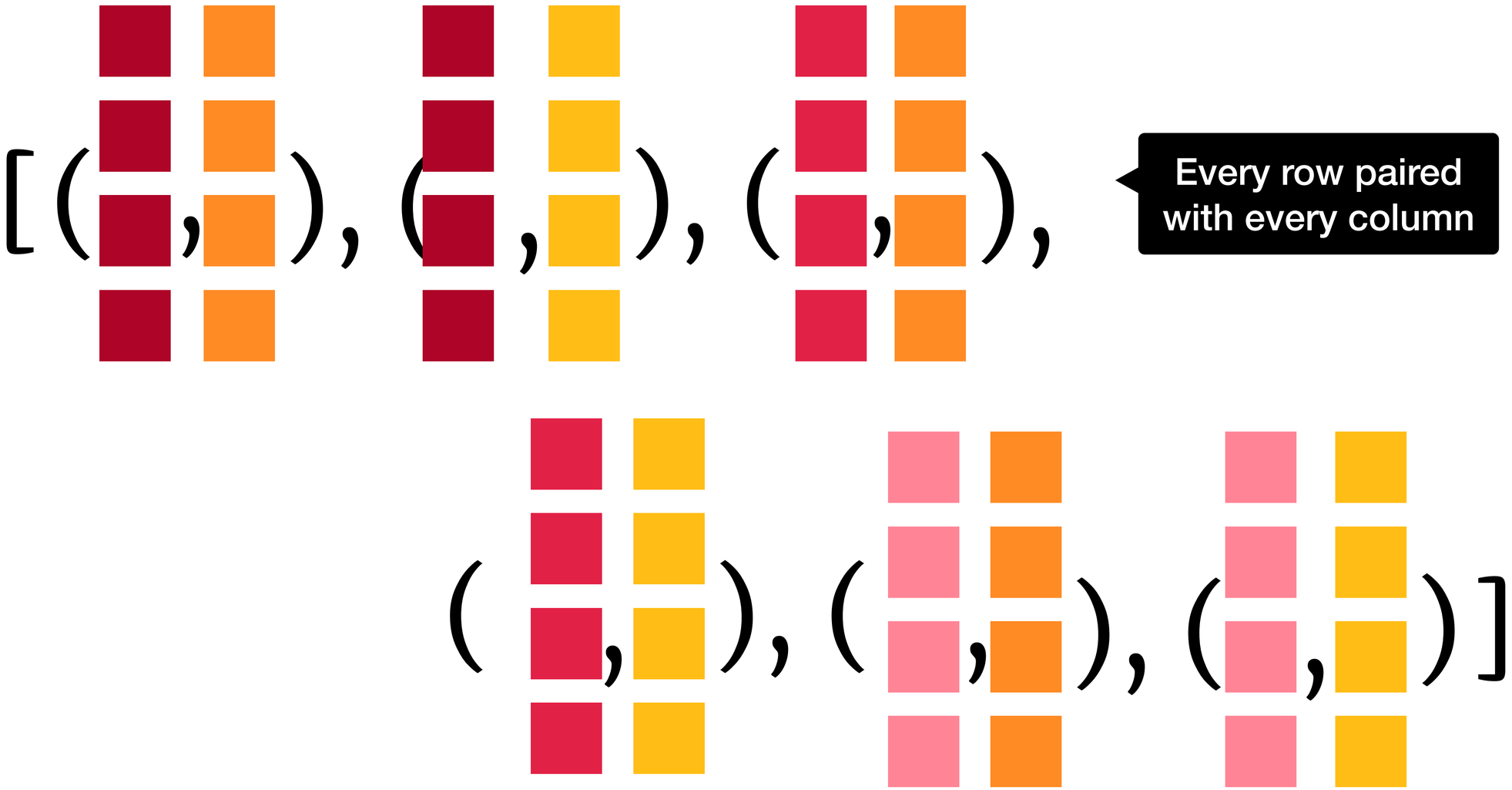


View matrices as lists of rows/columns



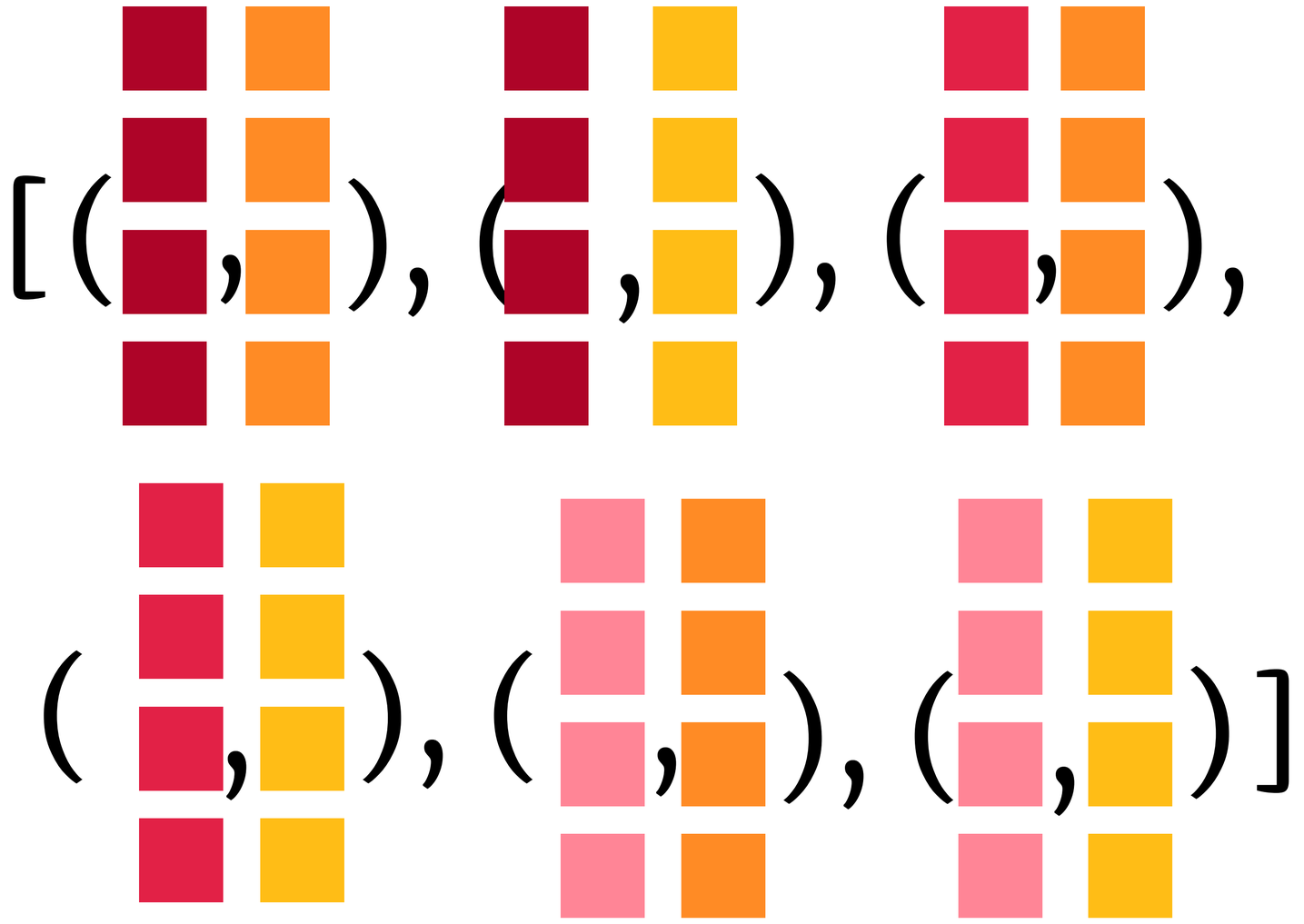
View matrices as lists of rows/columns

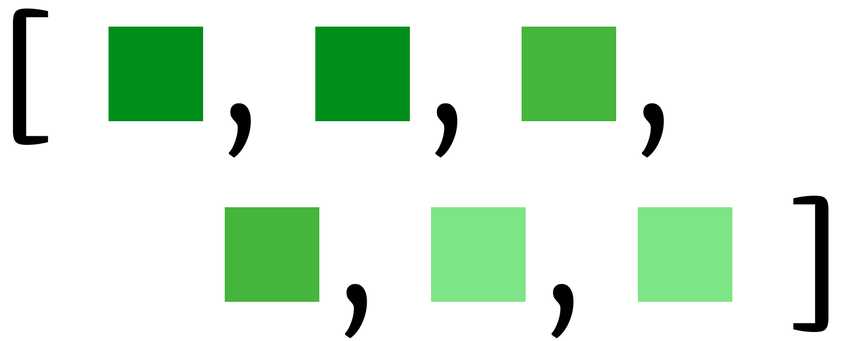

Take their
Cartesian product


Every row paired
with every column

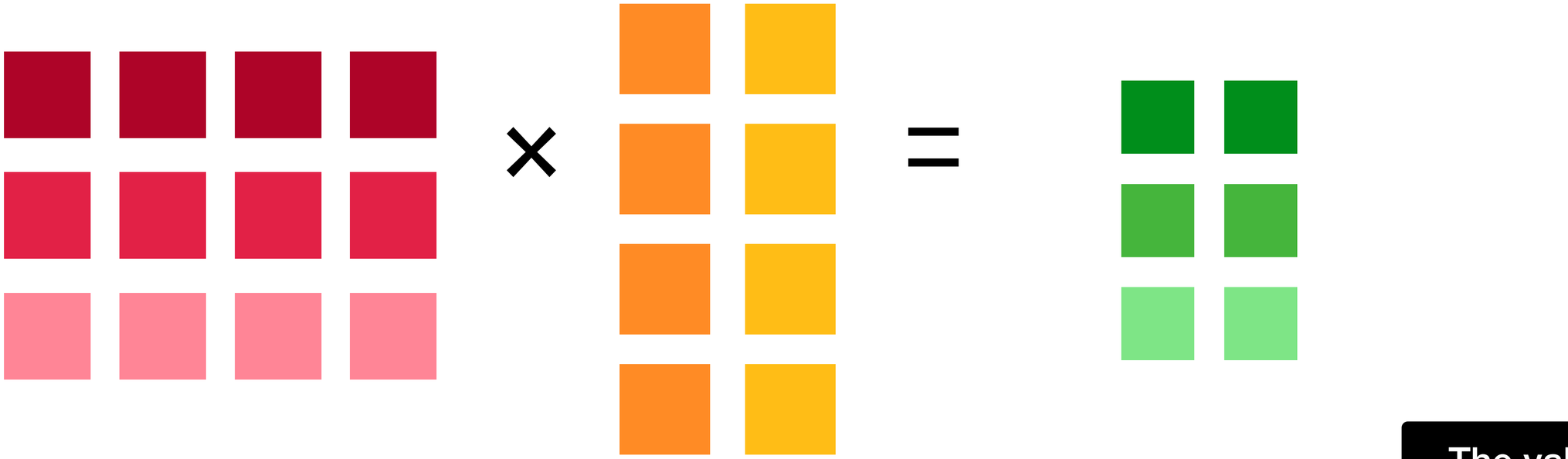
`map dotProd`

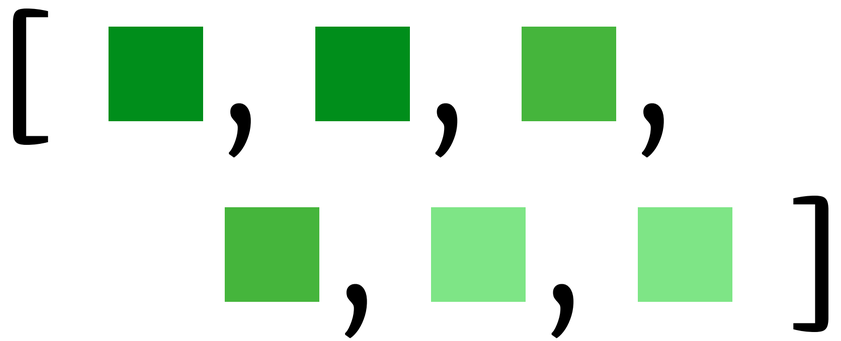
Map dot product operator over every row–column pair

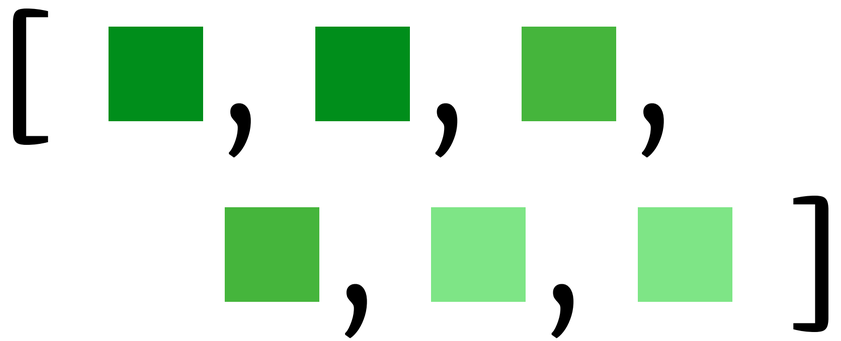




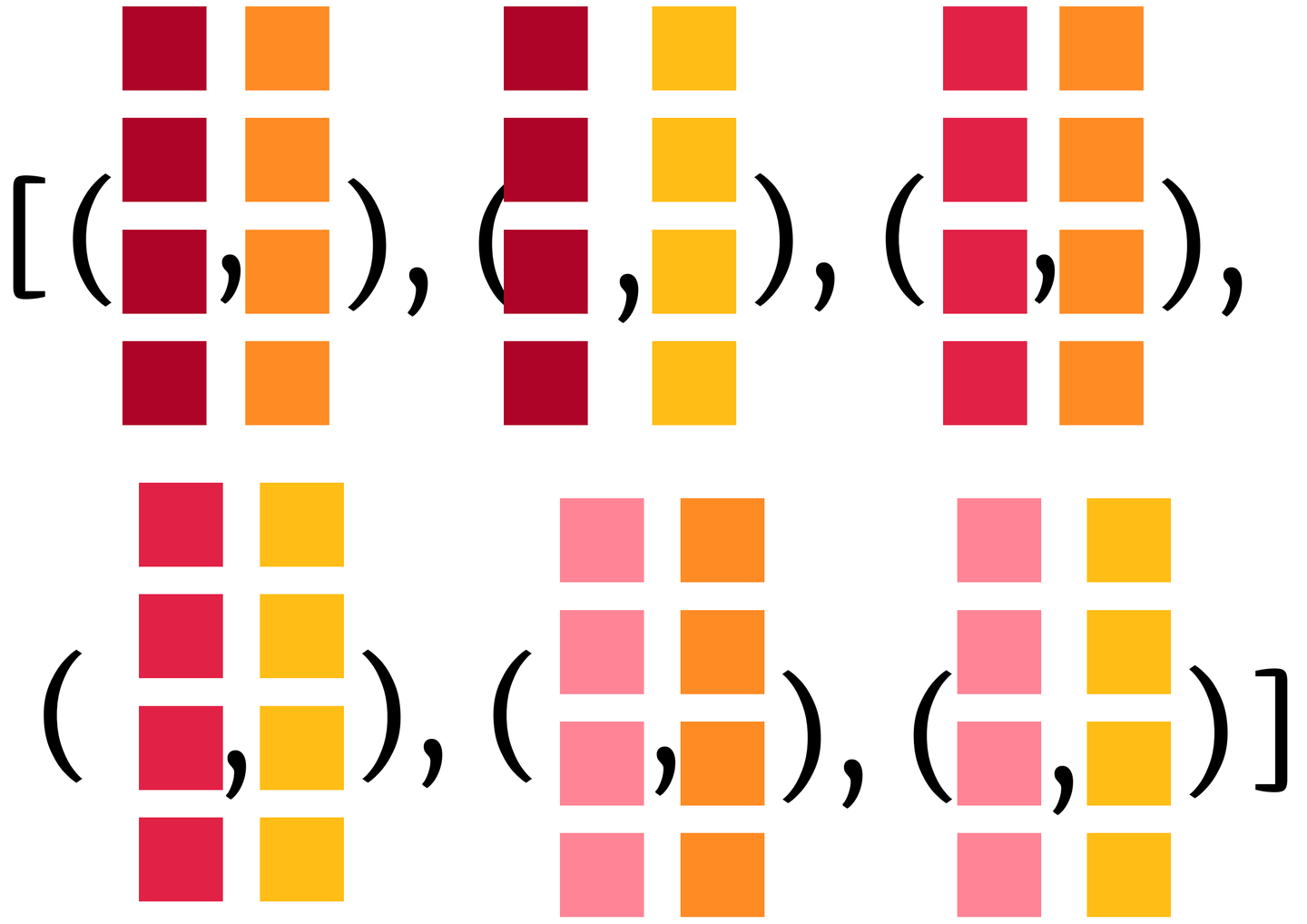
But there’s a problem!

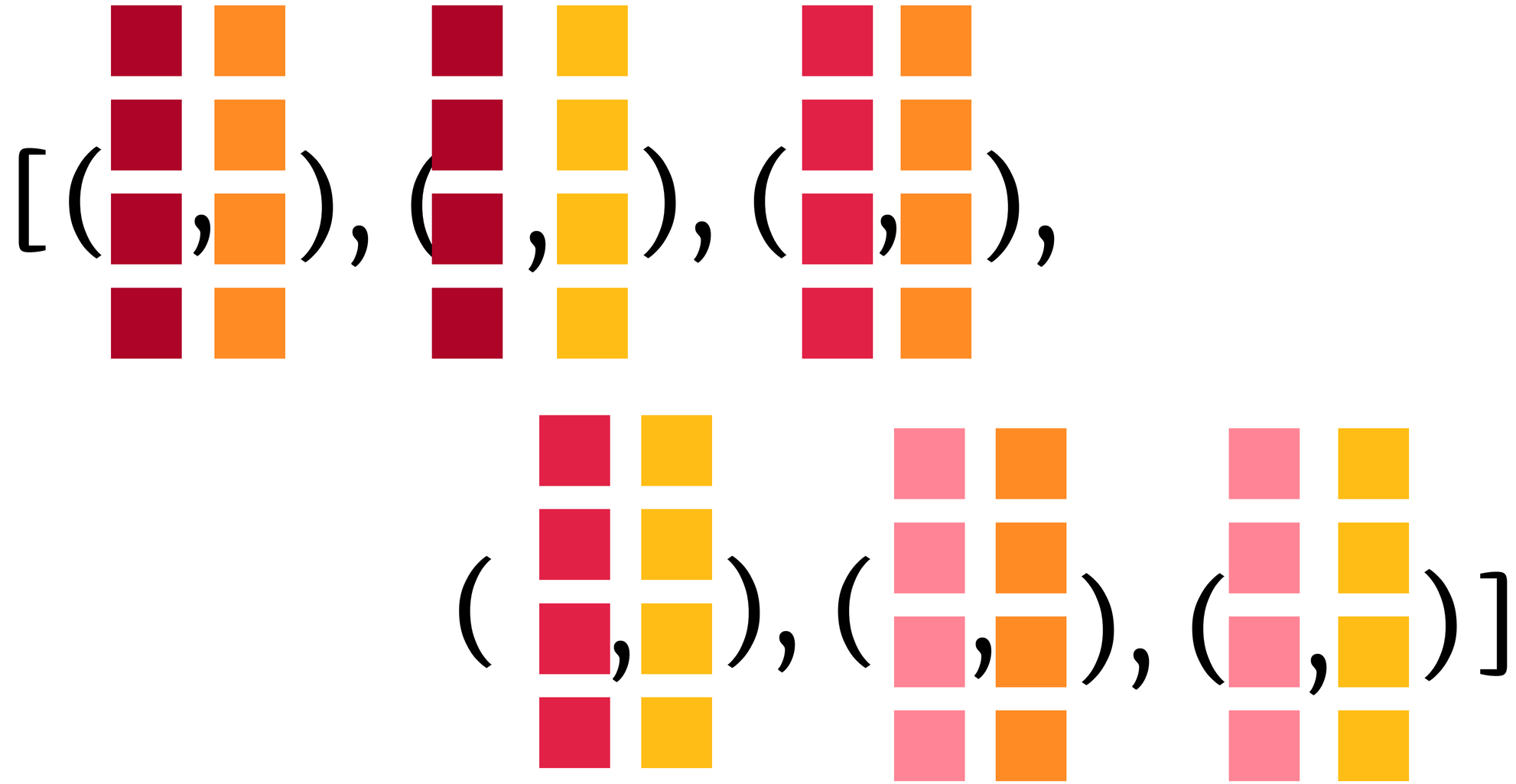

The values are correct, but the shape is missing!

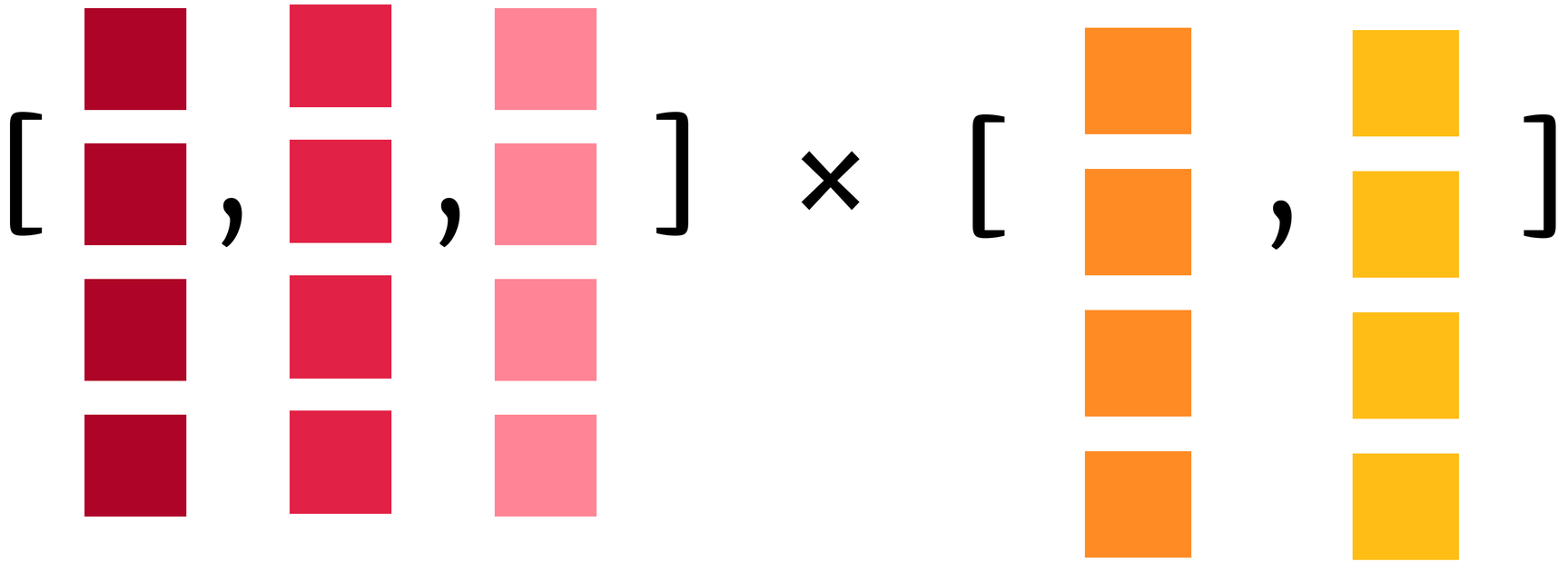


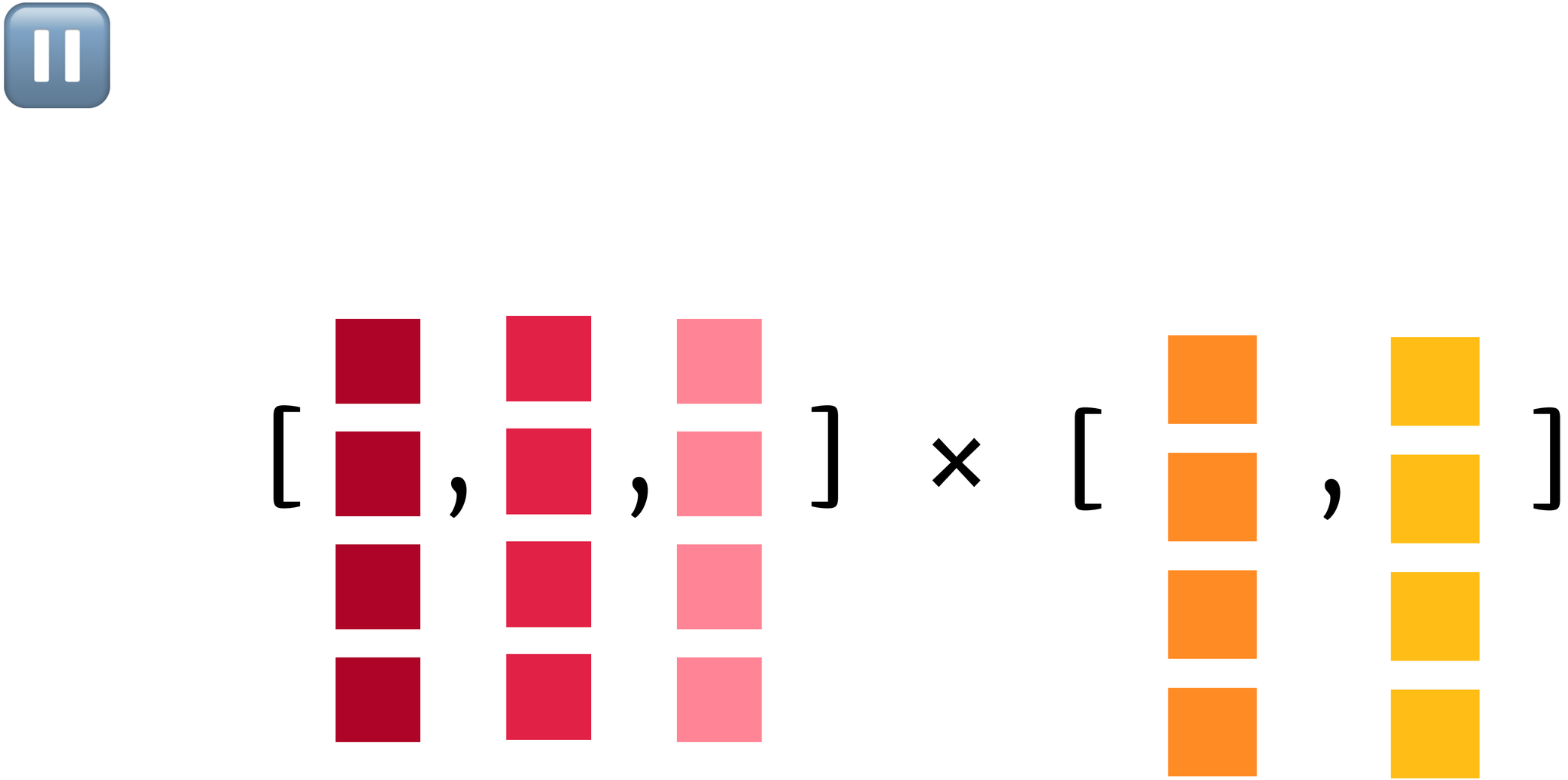


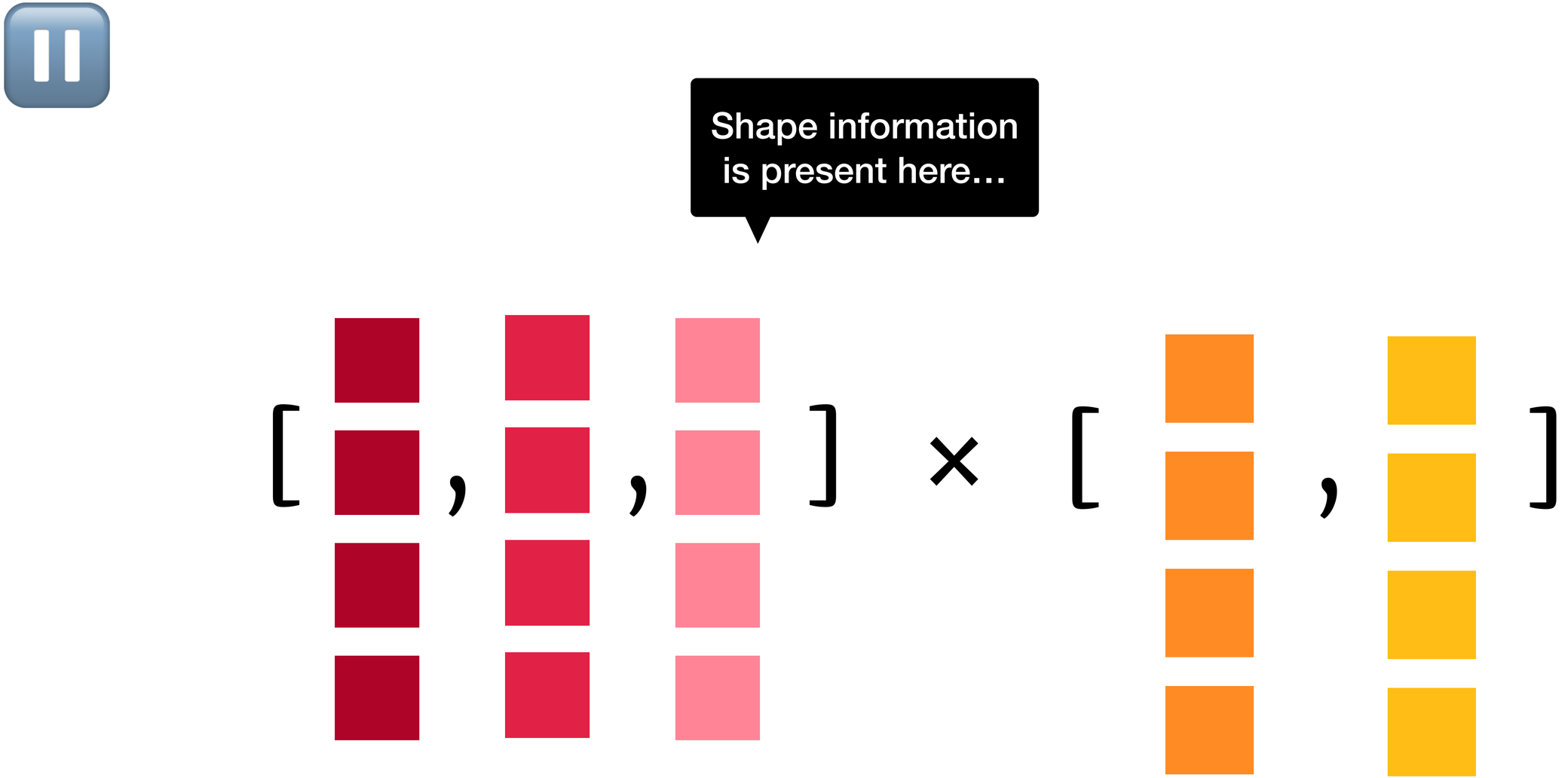
map dot-product

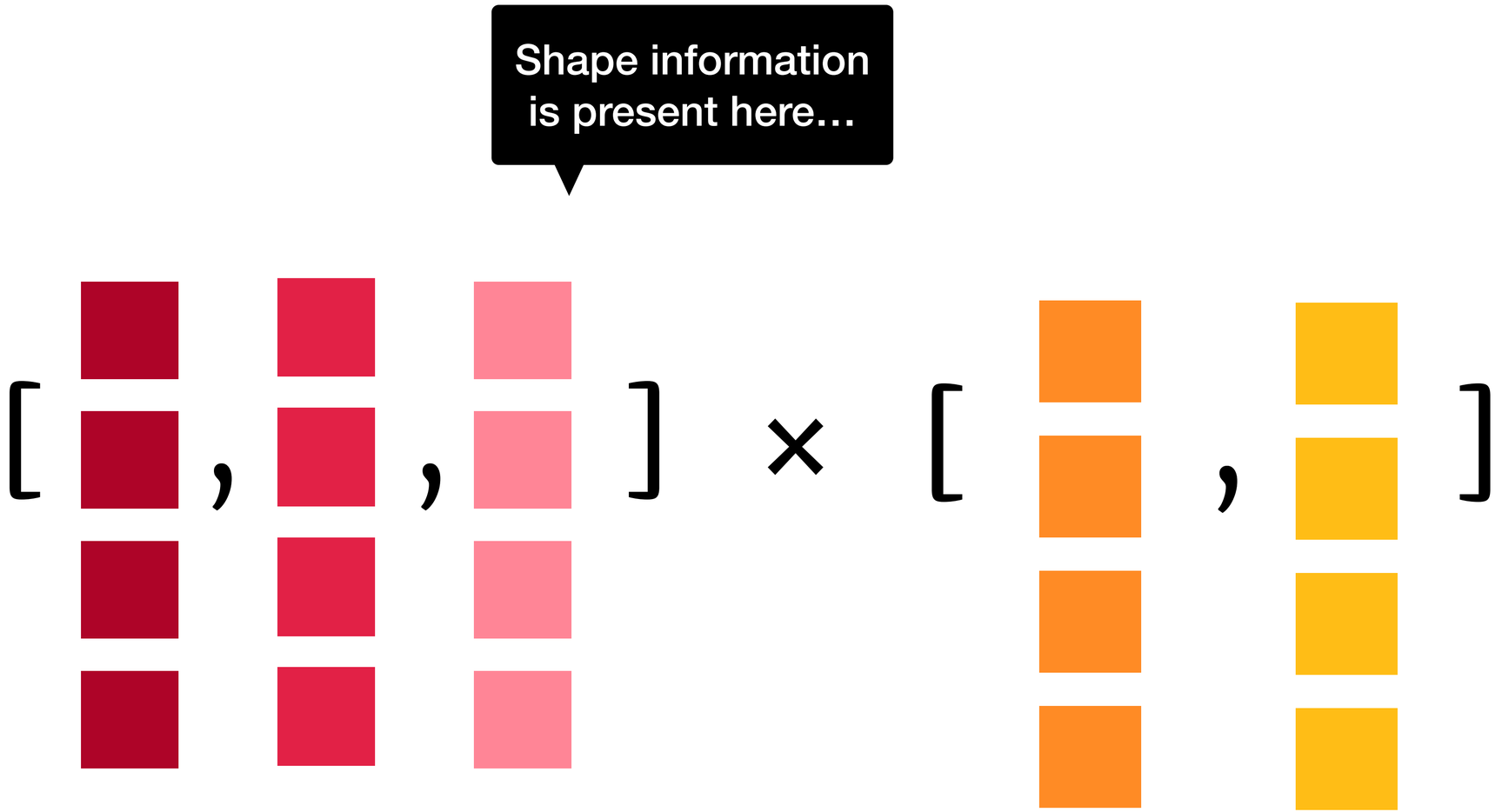


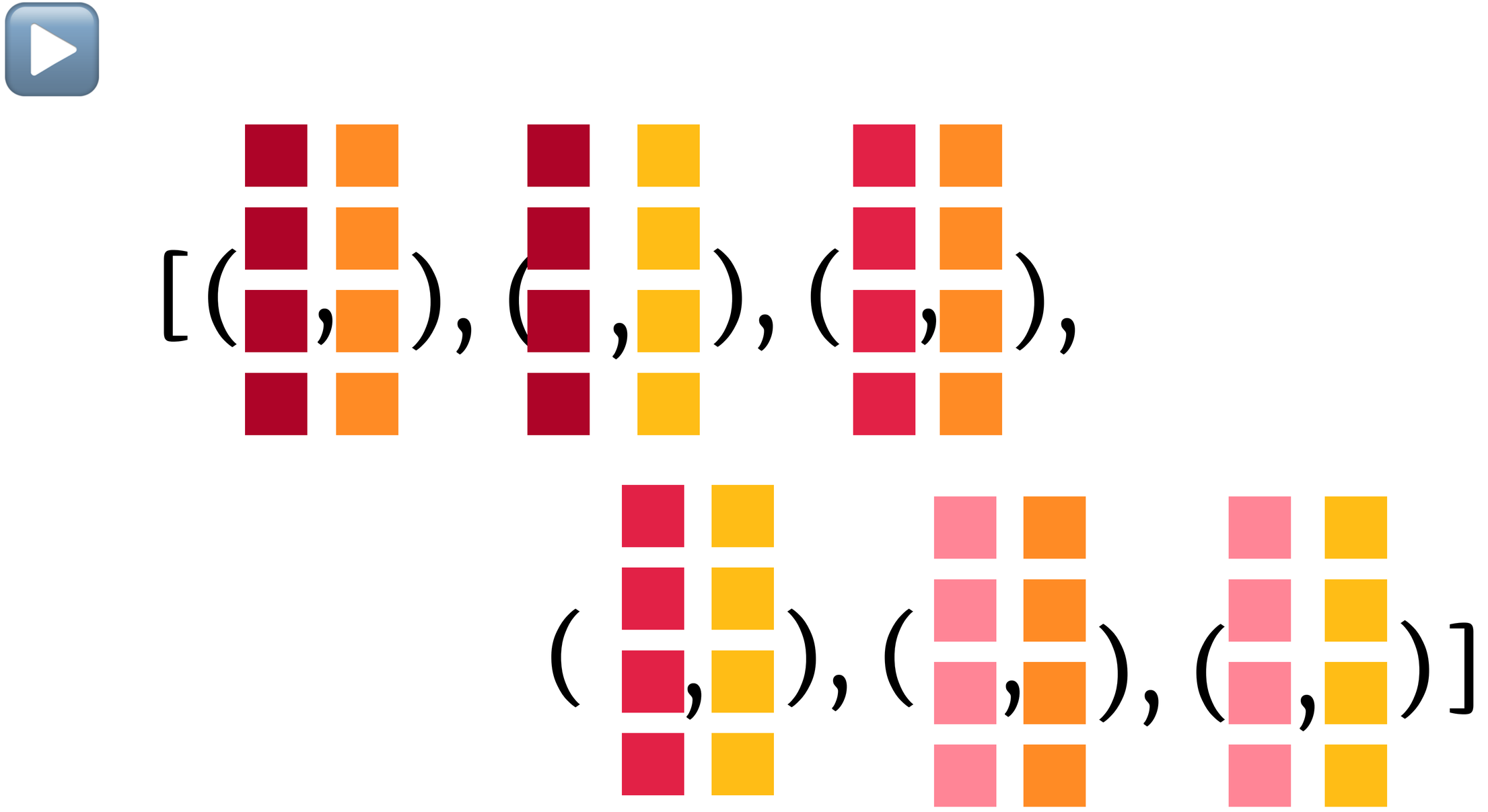


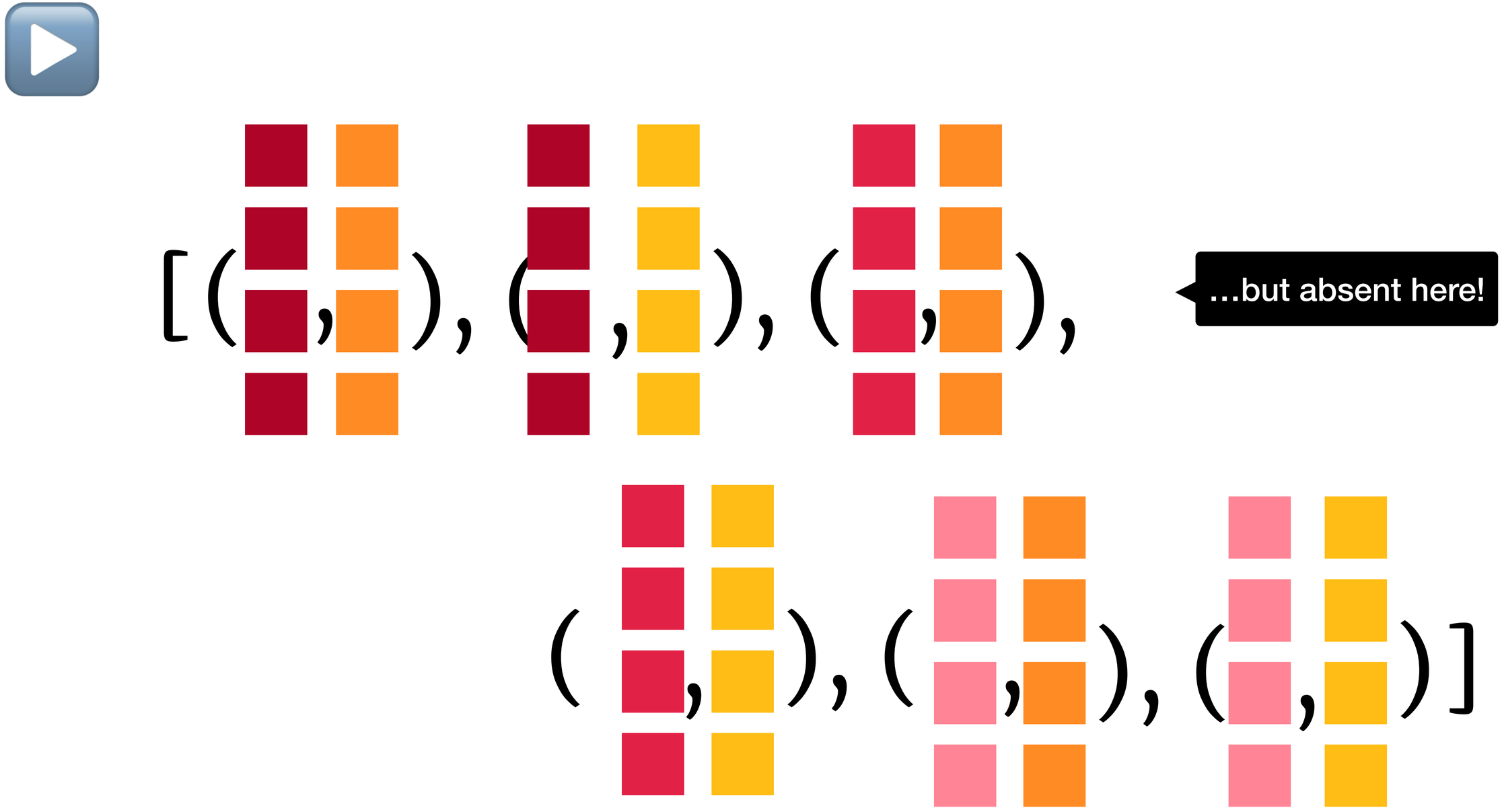




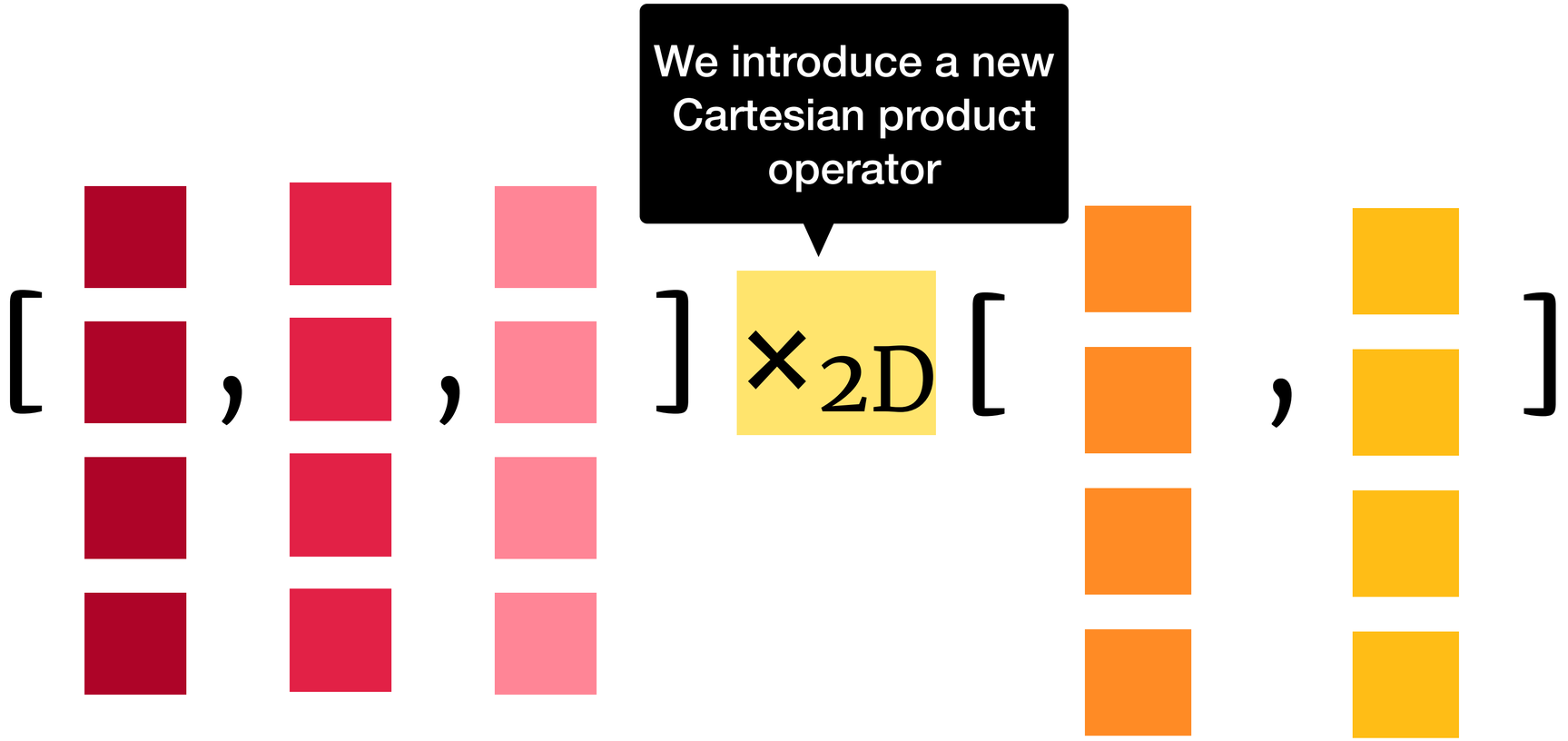

Shape information is present here…
[, ,] × [,]

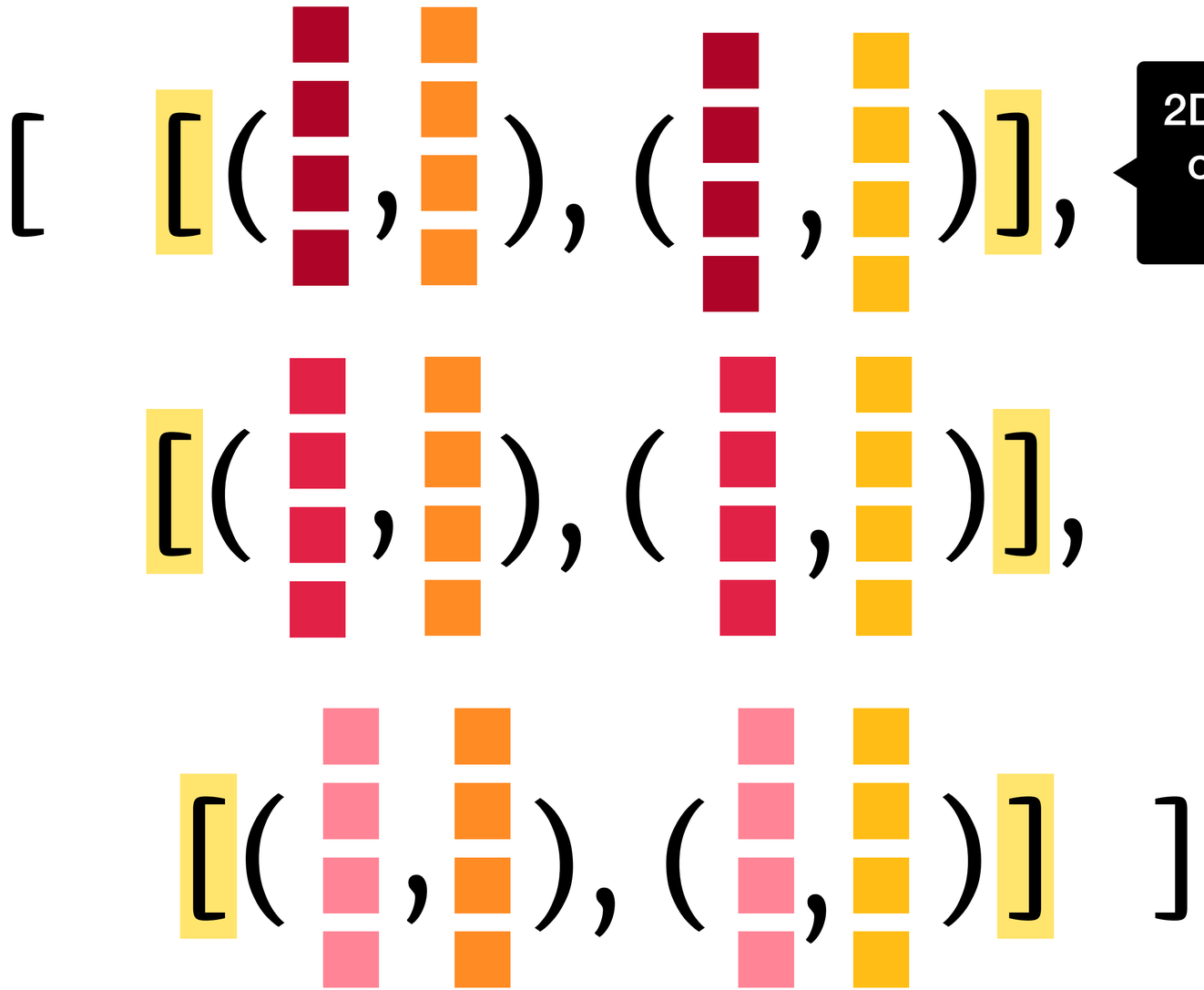

Shape information is present here…
[, ,] × [,]



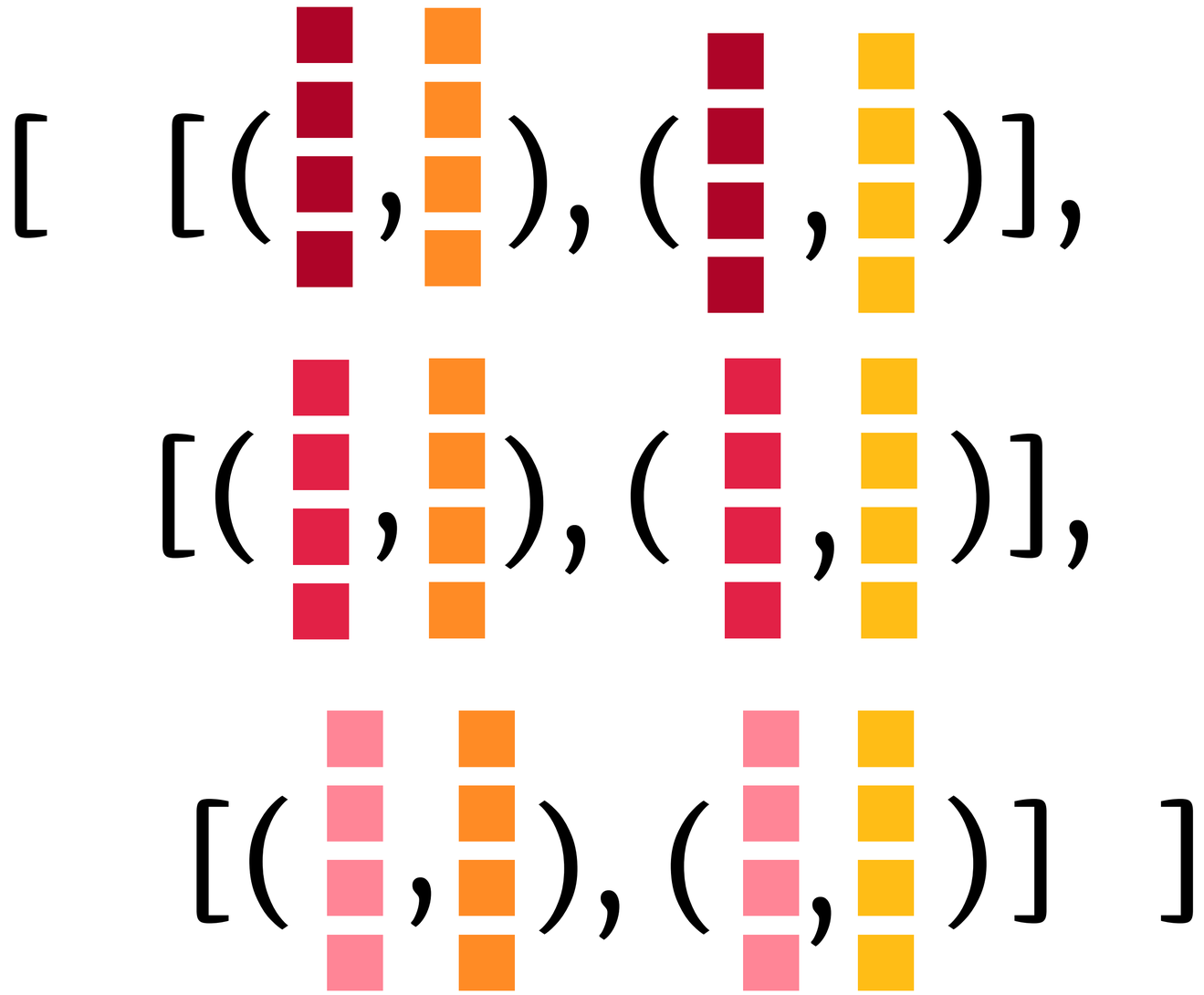

…but absent here!

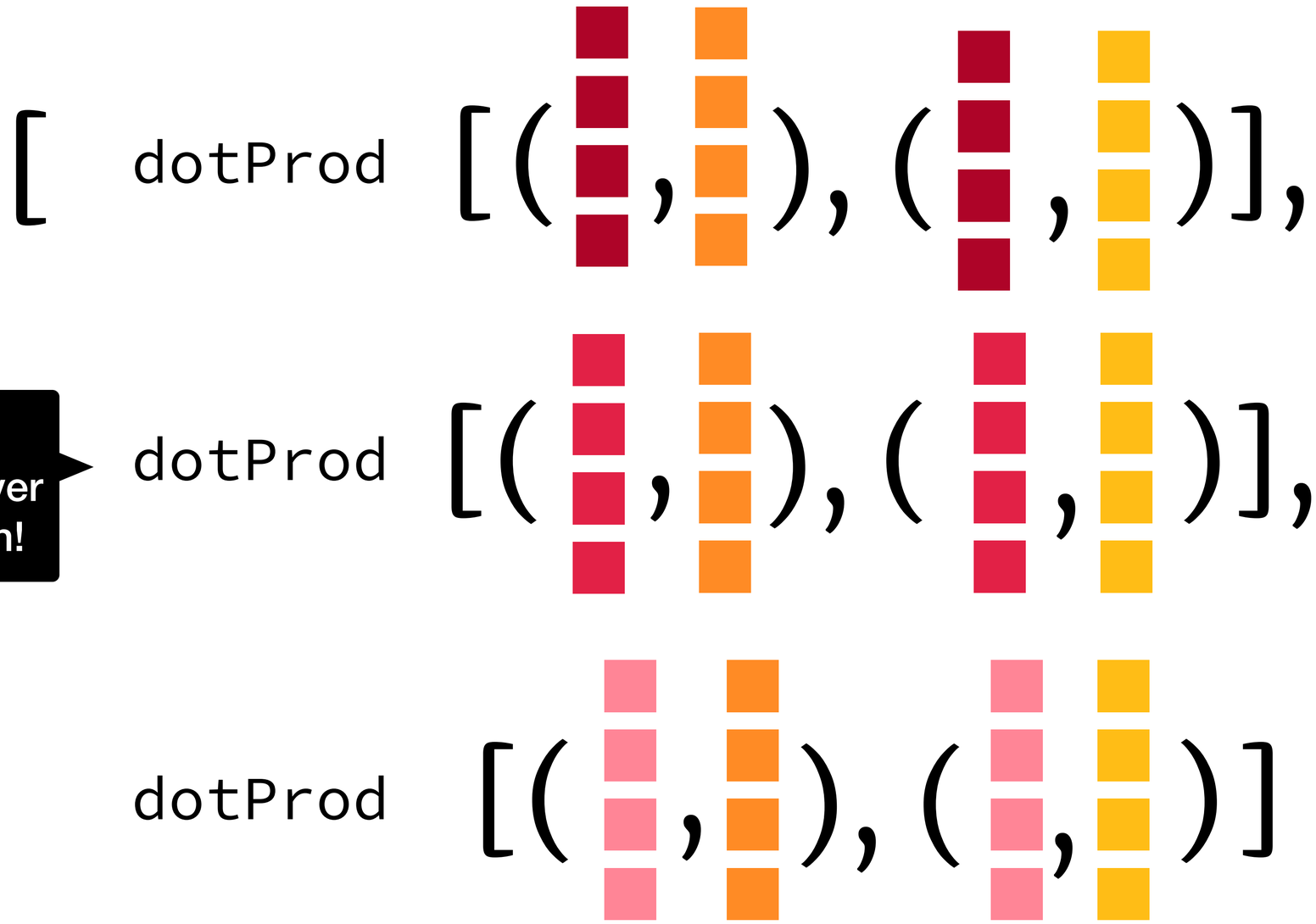
Cartesian product destroys our shape information!


We introduce a new Cartesian product operator
×2D


2D Cartesian product operator preserves shape info

map dotProd

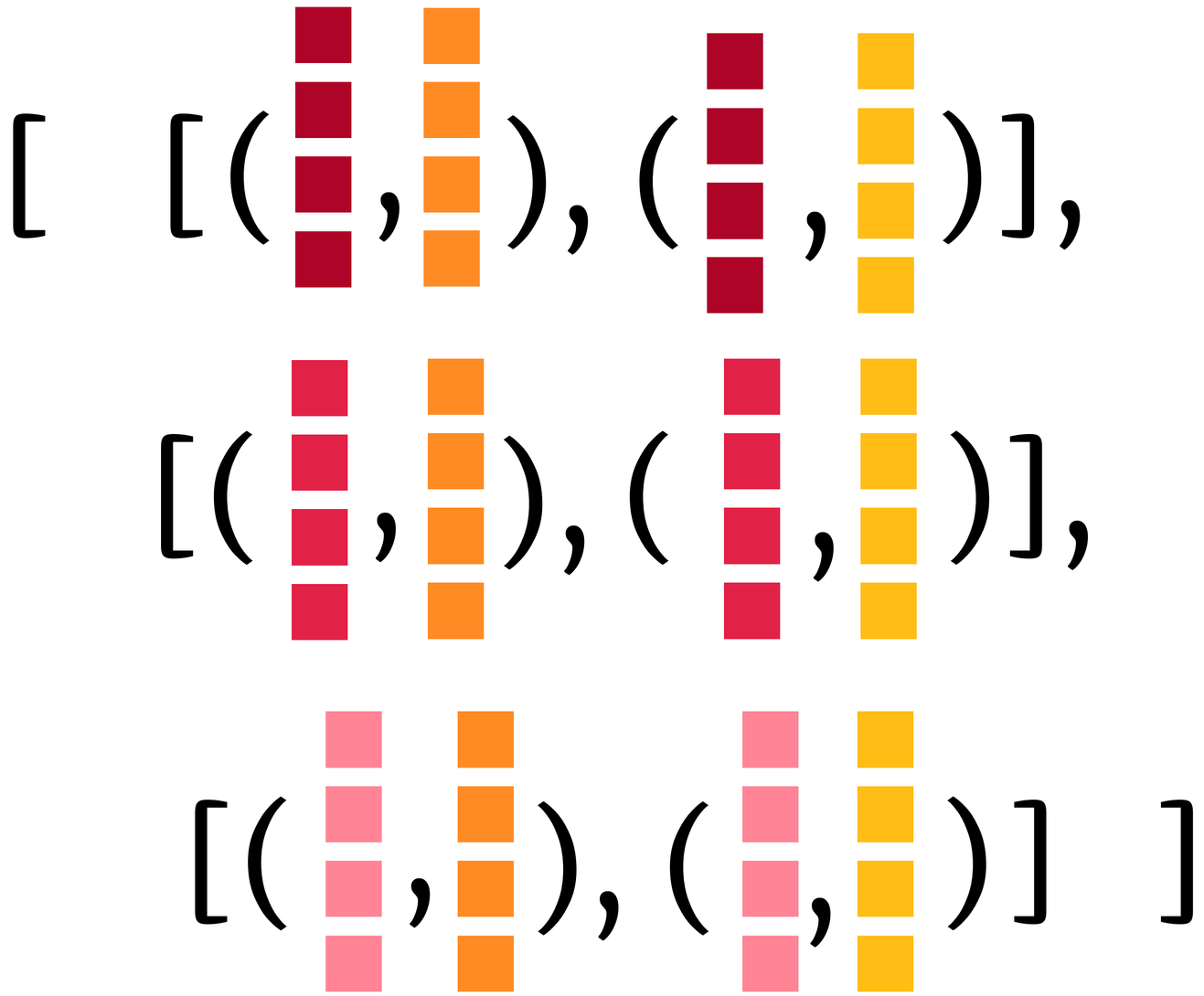


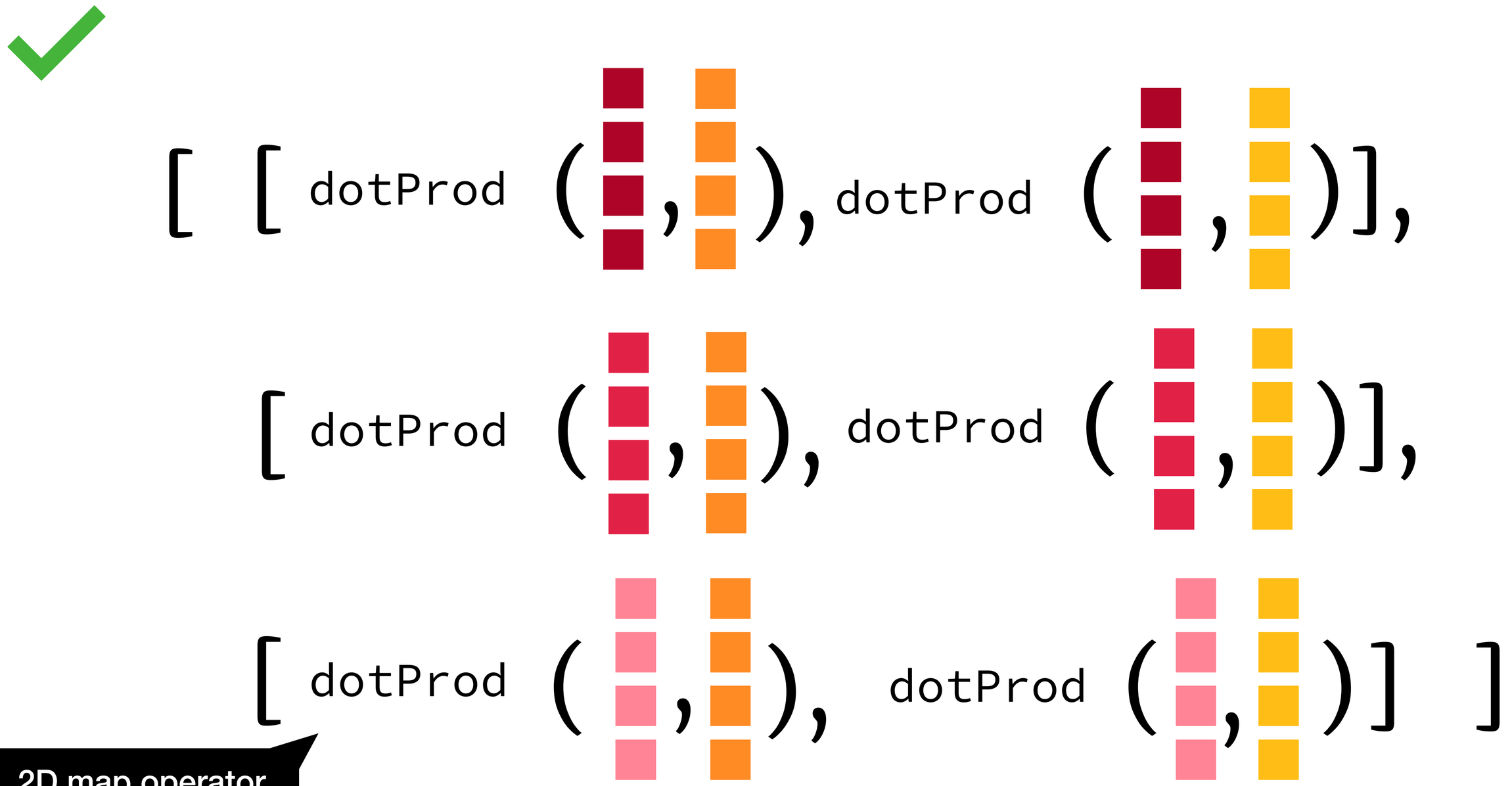


But now, map operator maps over wrong dimension!

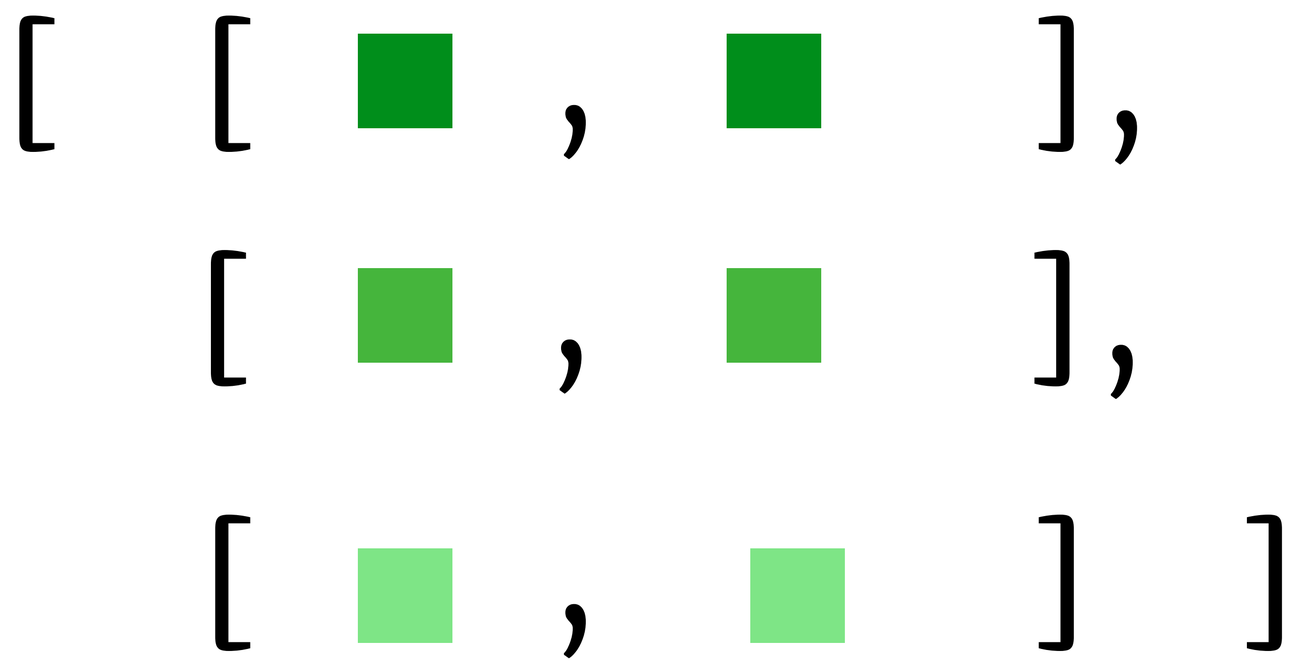
`map2D dotProd`

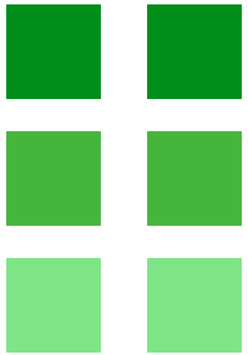
We also need a new map operator

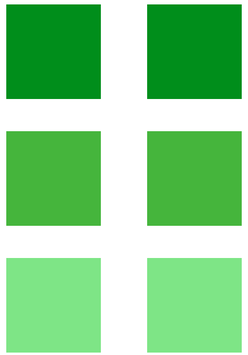




2D map operator maps over correct dimension







Shape information is preserved!

$\times_{2D}$ and `map2D` hard-code which dimensions are iterated over and which dimensions are computed on…

$\times_{2D}$ and `map2D` hard-code which dimensions are iterated over and which dimensions are computed on…

…but if tensor shapes change, we'll need entirely new operators!

$\times_{2D}$ and `map2D` hard-code which dimensions are **iterated over** and which dimensions are **computed on**…

…but if tensor shapes change, we'll need entirely new operators!

Can we encode this in the tensor itself?

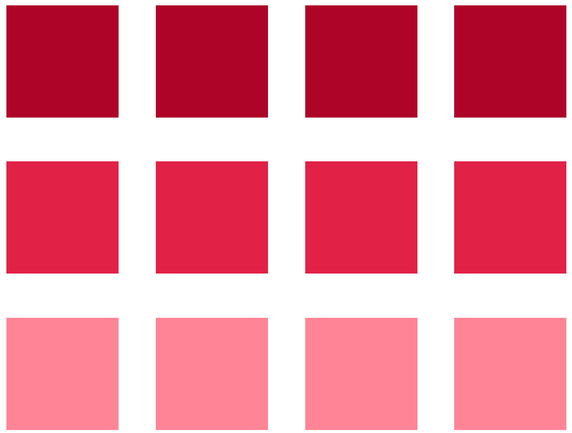
`×2D` and `map2D` hard-code which dimensions are iterated over and which dimensions are computed on…

…but if tensor shapes change, we'll need entirely new operators!

Can we encode this in the tensor itself?

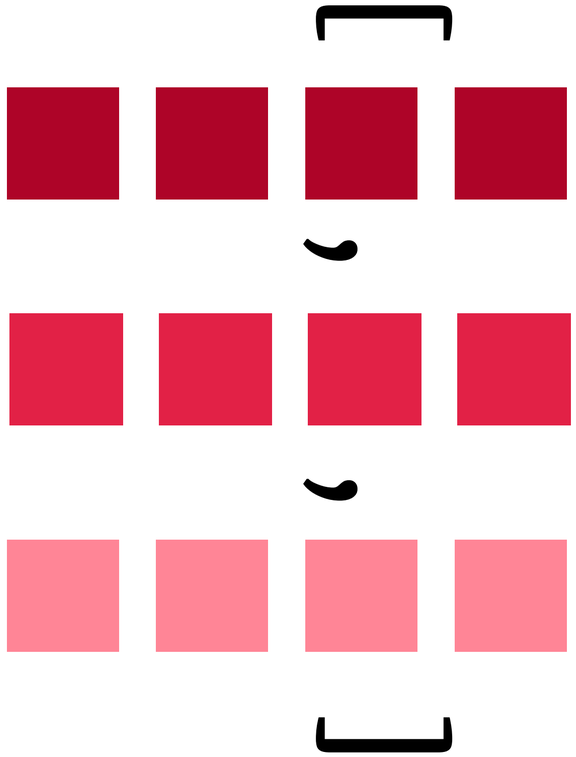
(Yes! This is what Glenside's access patterns do!)

A tensor looks like…



(3, 4)

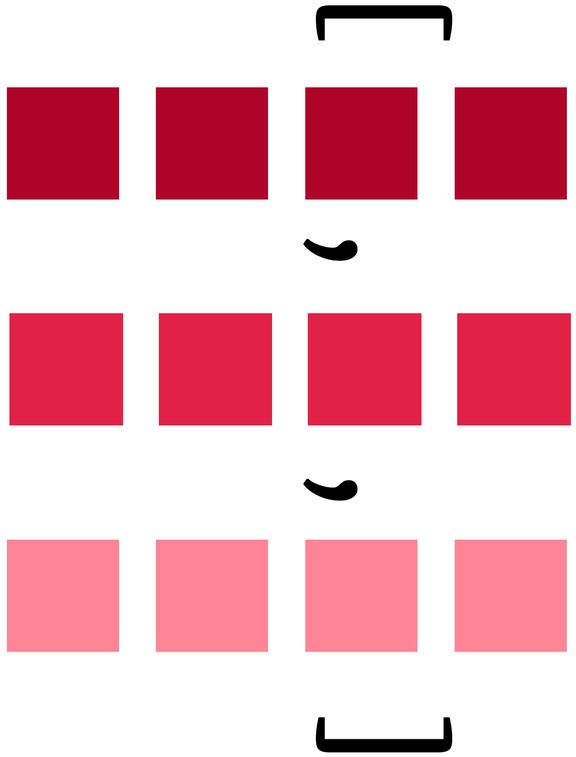
An access pattern looks like…



A 3-length vector of
4-length vectors

((3), (4))

An access pattern looks like…

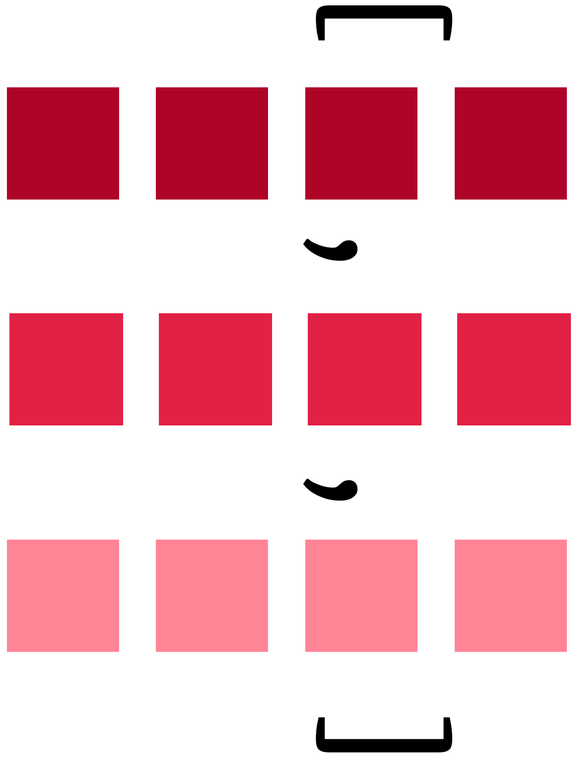


A 3-length vector of 4-length vectors

access dimensions (iterated over)

((3), (4))

An access pattern looks like…



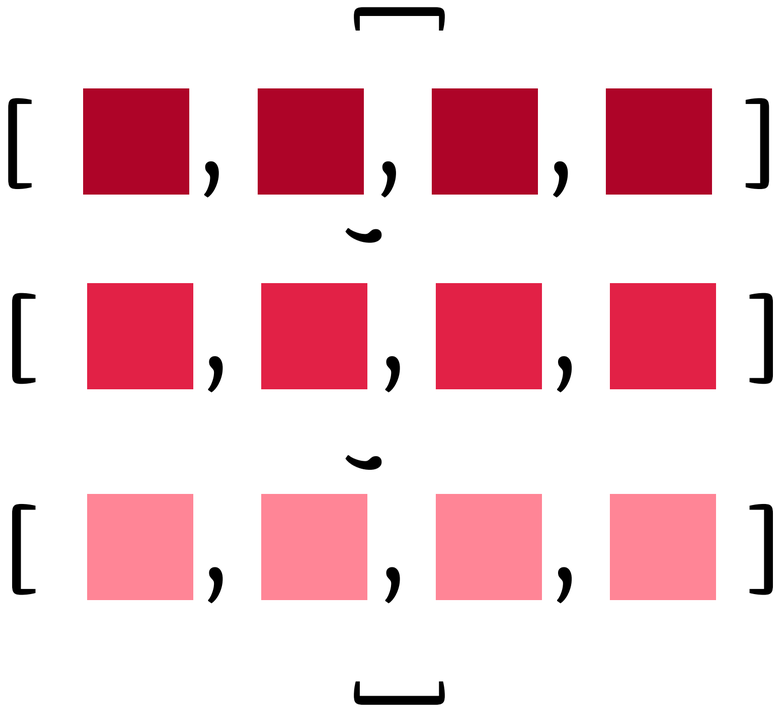
A 3-length vector of 4-length vectors

access dimensions
(iterated over)

((3), (4))

compute dimensions
(computed on)

An access pattern looks like…



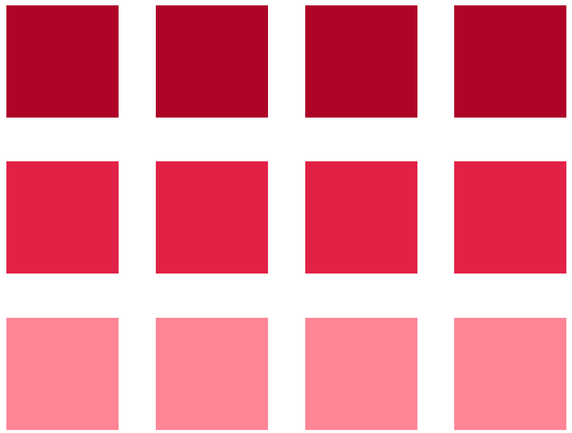
A (3, 4)-shaped tensor of scalars

access dimensions (iterated over)

((3, 4), ( ))

compute dimensions (computed on)

An access pattern looks like…



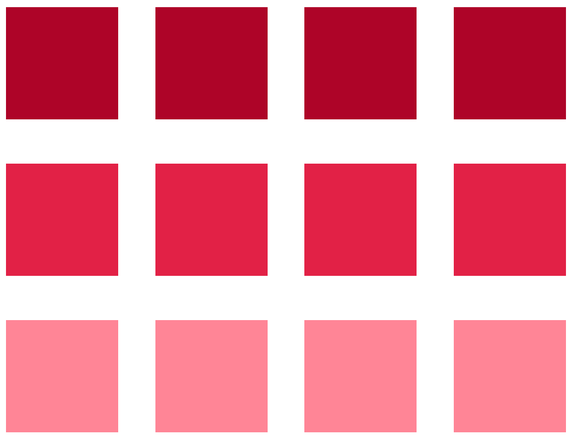
A scalar-shaped tensor of a single (3,4)-shaped tensor

access dimensions (iterated over)

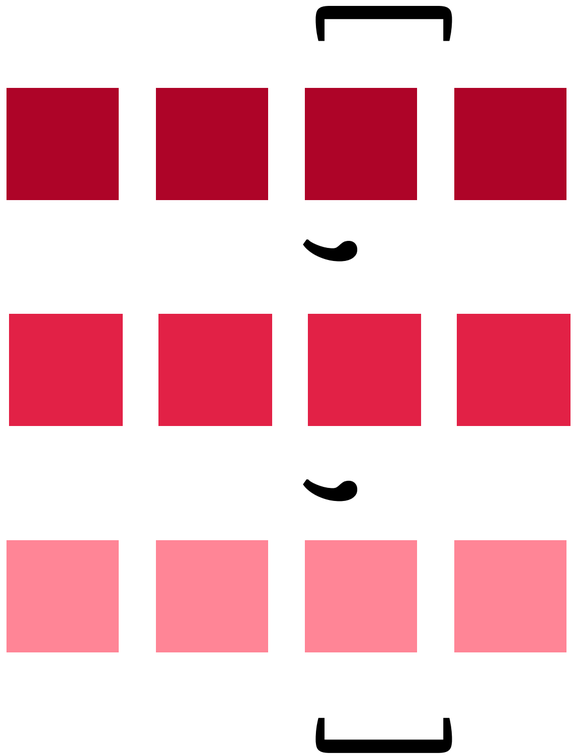
(( ), (3,4))

compute dimensions (computed on)

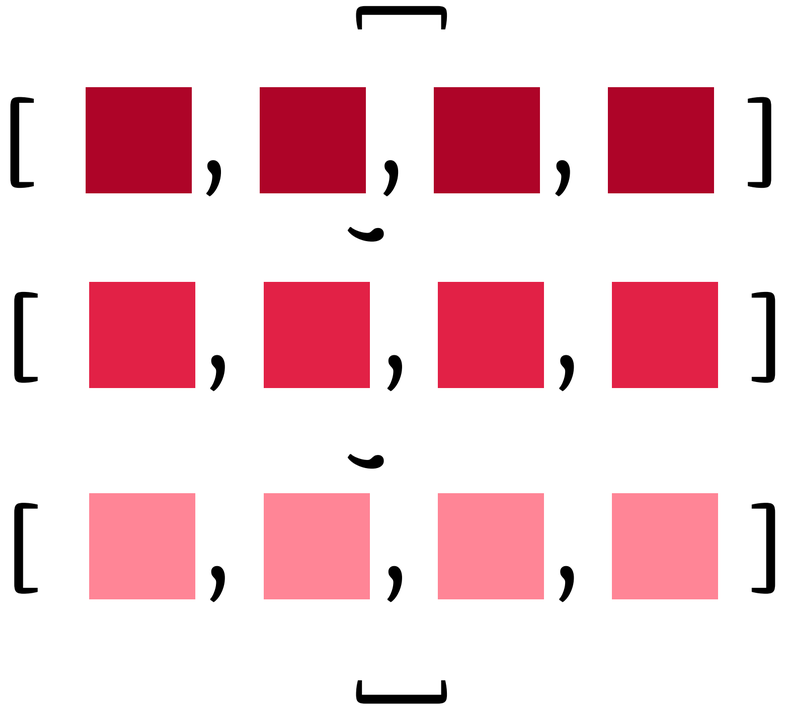
(( ), (3,4))



((3), (4))



((3,4), ( ))

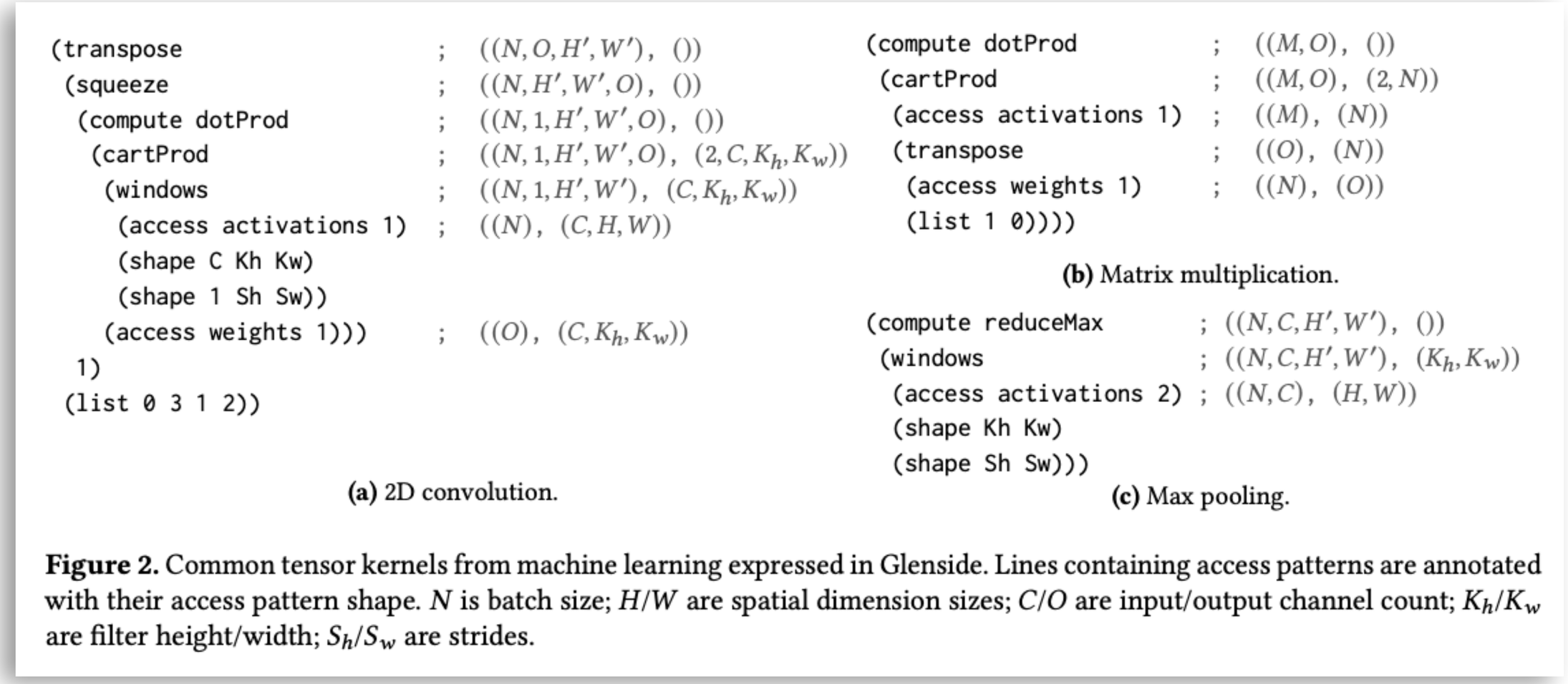


Same tensor, three possible views!

Transformer	Input(s)	Output Shape
`access`	$((a_0, \ldots), (\ldots, a_n))$ and non-negative integer i	$((a_0, \ldots, a_{i-1}), (a_i, \ldots, a_n))$
`cartProd`	$((a_0, \ldots, a_n), (c_0, \ldots, c_p))$ and $((b_0, \ldots, b_m), (c_0, \ldots, c_p))$	$((a_0, \ldots, a_n, b_0, \ldots, b_m), (2, c_0, \ldots, c_p))$
`windows`	$((a_0, \ldots, a_m), (b_0, \ldots, b_n))$,	$((a_0, \ldots, a_m, b'_0, \ldots, b'_n), (w_0, \ldots, w_n))$,
	wind	
`slice`	$((a_0,$	
	dime	
`squeeze`	$((a_0, \ldots), (\ldots, a_n))$,	$((a_0, \ldots), (\ldots, a_n))$
	dimension index d; we assume $a_d = 1$	with a_d removed
`flatten`	$((a_0, \ldots, a_m), (b_0, \ldots, b_n))$	$((a_0 \cdots a_m), (b_0 \cdots b_n))$
`reshape`	$((a_0, \ldots, a_m), (b_0, \ldots, b_n))$,	$((c_0, \ldots, c_p), (d_0, \ldots, d_q))$,
	access pattern shape literal $((c_0, \ldots, c_p), (d_0, \ldots, d_q))$	if $a_0 \cdots a_m = c_0 \cdots c_p$ and $b_0 \cdots b_n = d_0 \cdots d_q$

Table 1. Glenside's access pattern transformers.

We redefine common tensor and list operators with access pattern semantics, which gives us the **Glenside IR!**



```
(transpose                          ;  ((N, O, H′, W′), ())
 (squeeze                           ;  ((N, H′, W′, O), ())
  (compute dotProd                  ;  ((N, 1, H′, W′, O), ())
   (cartProd                        ;  ((N, 1, H′, W′, O), (2, C, Kh, Kw))
    (windows                        ;  ((N, 1, H′, W′), (C, Kh, Kw))
     (access activations 1)         ;  ((N), (C, H, W))
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))            ;  ((O), (C, Kh, Kw))
  1)
 (list 0 3 1 2))
```

(a) 2D convolution.

```
(compute dotProd                    ;  ((M, O), ())
 (cartProd                          ;  ((M, O), (2, N))
  (access activations 1)            ;  ((M), (N))
  (transpose                        ;  ((O), (N))
   (access weights 1)               ;  ((N), (O))
   (list 1 0))))
```

(b) Matrix multiplication.

```
(compute reduceMax                  ;  ((N, C, H′, W′), ())
 (windows                           ;  ((N, C, H′, W′), (Kh, Kw))
  (access activations 2)            ;  ((N, C), (H, W))
  (shape Kh Kw)
  (shape Sh Sw)))
```

(c) Max pooling.

Figure 2. Common tensor kernels from machine learning expressed in Glenside. Lines containing access patterns are annotated with their access pattern shape. N is batch size; H/W are spatial dimension sizes; C/O are input/output channel count; K_h/K_w are filter height/width; S_h/S_w are strides.

Glenside can represent common kernels in machine learning.

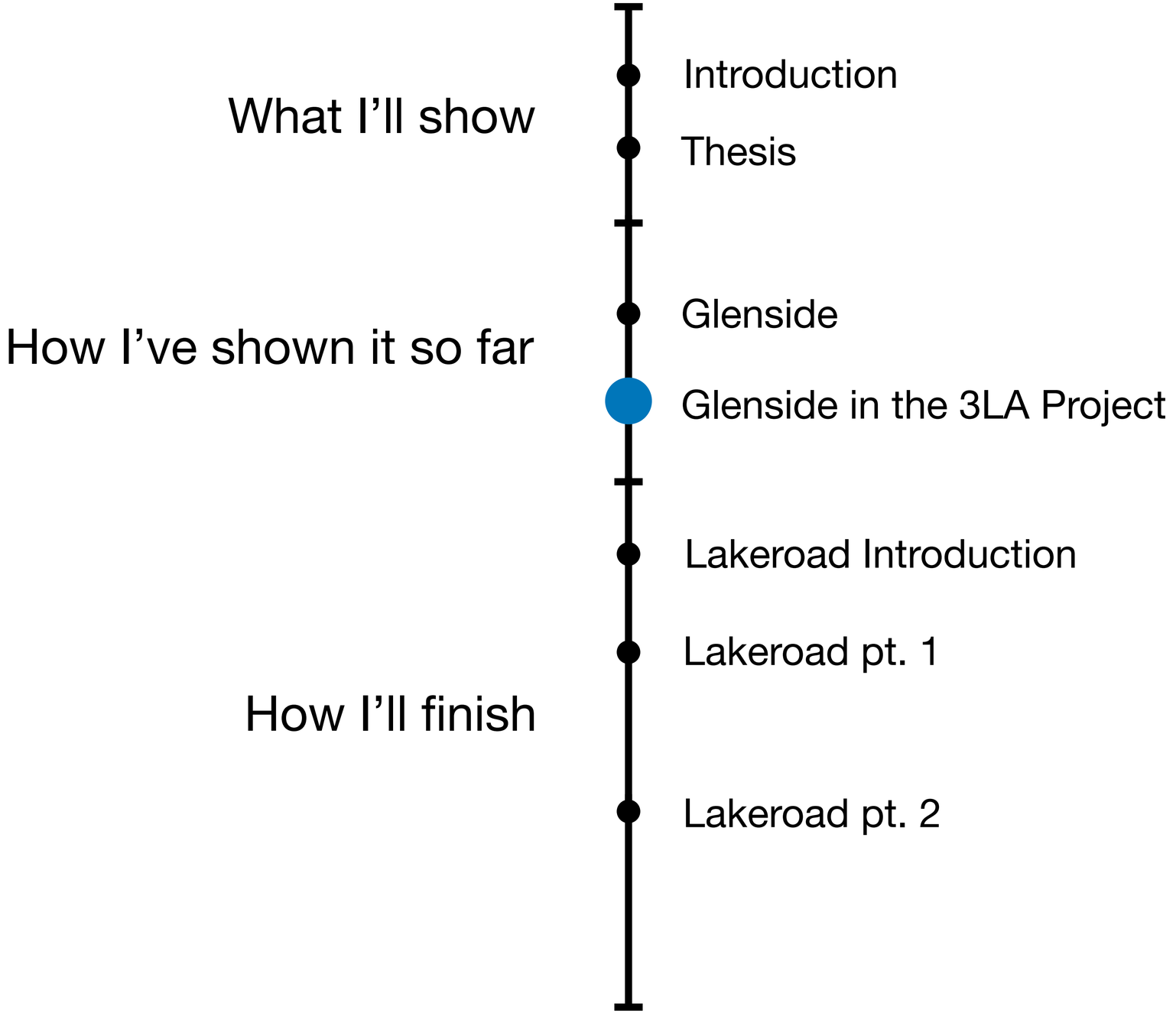
Smith, Gus Henry, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. "Pure tensor program rewriting via access patterns (representation pearl)." MAPS 2021.

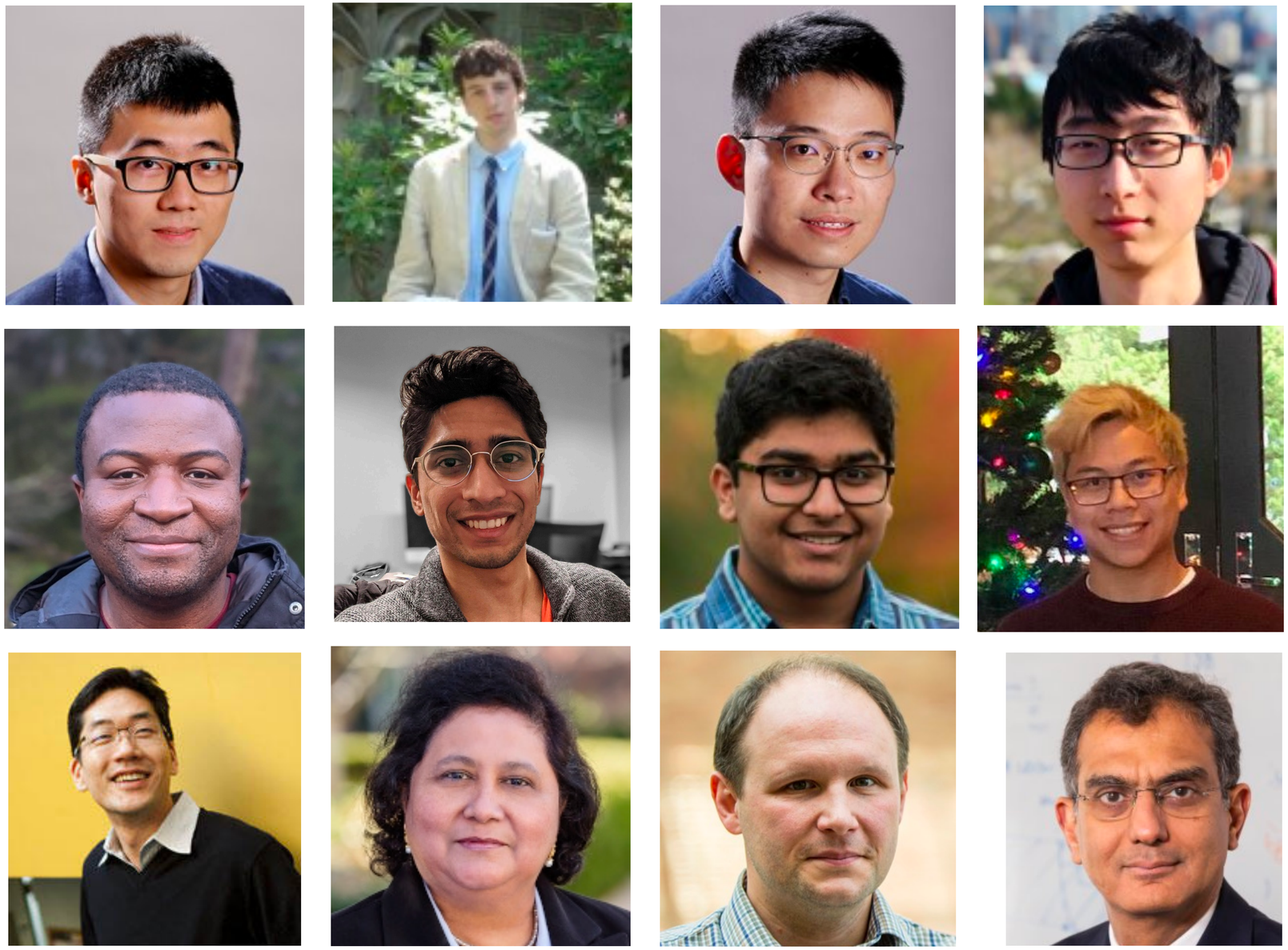
But how is Glenside useful?

But how is Glenside useful?

More importantly, how does it demonstrate my thesis?

Glenside in the 3LA project


What I’ll show
How I’ve shown it so far
How I’ll finish
Introduction
Thesis
Glenside
Glenside in the 3LA Project
Lakeroad Introduction
Lakeroad pt. 1
Lakeroad pt. 2



ada

Applications Driving Architectures

PAUL G. ALLEN SCHOOL

Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Thierry Tambe, **Gus Smith**, Akash Gaonkar, Vishal Canumalla, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, Sharad Malik
"Specialized Accelerators and Compiler Flows: Replacing Accelerator APIs with a Formal Software/Hardware Interface." *arXiv* 2022.

Simulating, verifying, and compiling workloads on custom accelerators is hard.

Simulating, verifying, and compiling workloads on custom accelerators is hard.

3LA is a toolkit which makes it easier, by compiling workloads to the ILA simulation and verification framework.

Simulating, verifying, and compiling workloads on custom accelerators is hard.

3LA is a toolkit which makes it easier, by compiling workloads to the ILA simulation and verification framework.

Glenside is a key component of 3LA, where it is used to discover mappings of workloads to accelerators.

Huang, B. Y., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., & Malik, S. (2019). Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SOC) verification. *ACM Transactions on Design Automation of Electronic Systems*, *24*(1), [10]. https://doi.org/10.1145/3282444

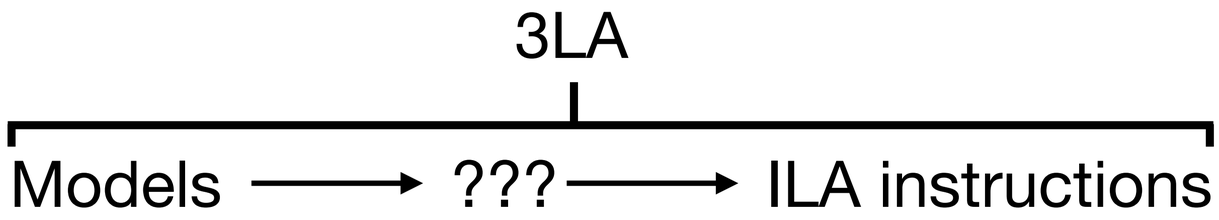
Allows hardware developers to specify ISA-like interface for their design

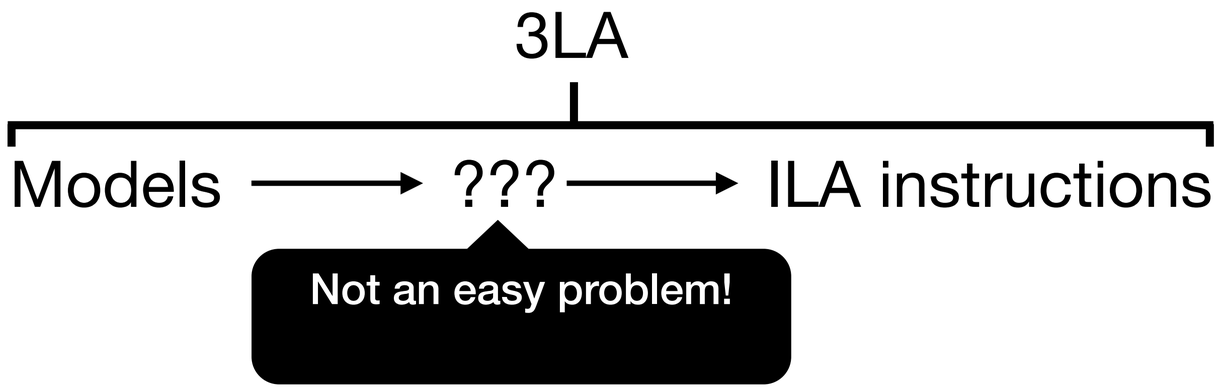
Huang, B. Y., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., & Malik, S. (2019). Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SOC) verification. *ACM Transactions on Design Automation of Electronic Systems*, *24*(1), [10]. https://doi.org/10.1145/3282444

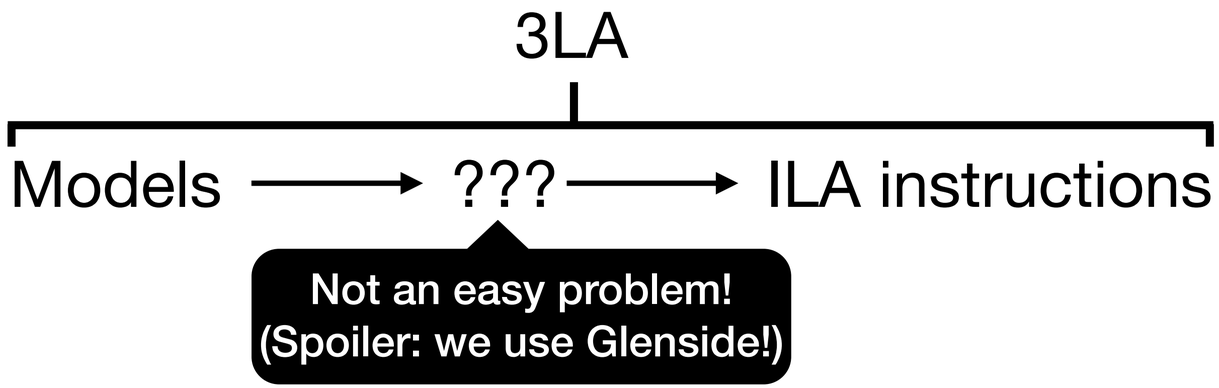
Allows hardware developers to specify ISA-like interface for their design

Portable, compiler-friendly, and provides verification and simulators out of the box!

Huang, B. Y., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., & Malik, S. (2019). Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SOC) verification. *ACM Transactions on Design Automation of Electronic Systems*, *24*(1), [10]. https://doi.org/10.1145/3282444


3LA
Models
???
ILA instructions


3LA
Models
???
ILA instructions
Not an easy problem!


3LA
Models
???
ILA instructions
Not an easy problem!
(Spoiler: we use Glenside!)

Can we use TVM's Bring Your Own Codegen?

Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. "Bring Your Own Codegen to Deep Learning Compiler." *arXiv* 2021.

Can we use TVM's Bring Your Own Codegen?

```
bias_add(dense(*, *), *))
```

Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. "Bring Your Own Codegen to Deep Learning Compiler." *arXiv* 2021.

Can we use **TVM's Bring Your Own Codegen?**

```
bias_add(dense(*, *), *))
```

Matches a linear layer: a dense followed by a bias addition.

Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. "Bring Your Own Codegen to Deep Learning Compiler." *arXiv* 2021.

	EfficientNet	MobileNet V2	ResMLP	ResNet-20	Transformer
VTA	0	1	38	2	66
FlexASR	0	0	0	2	0

Moreau, Thierry, et al. "VTA: an open hardware-software stack for deep learning." *arXiv preprint arXiv:1807.04188* (2018).

T. Tambe *et al.*, "9.8 A 25mm^2 SoC for IoT Devices with 18ms Noise-Robust Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET," *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, 2021, pp. 158-160, doi: 10.1109/ISSCC42613.2021.9366062.

```
%242 = dense(%240, %241, units=10);
add(%242, %linear_bias)
```

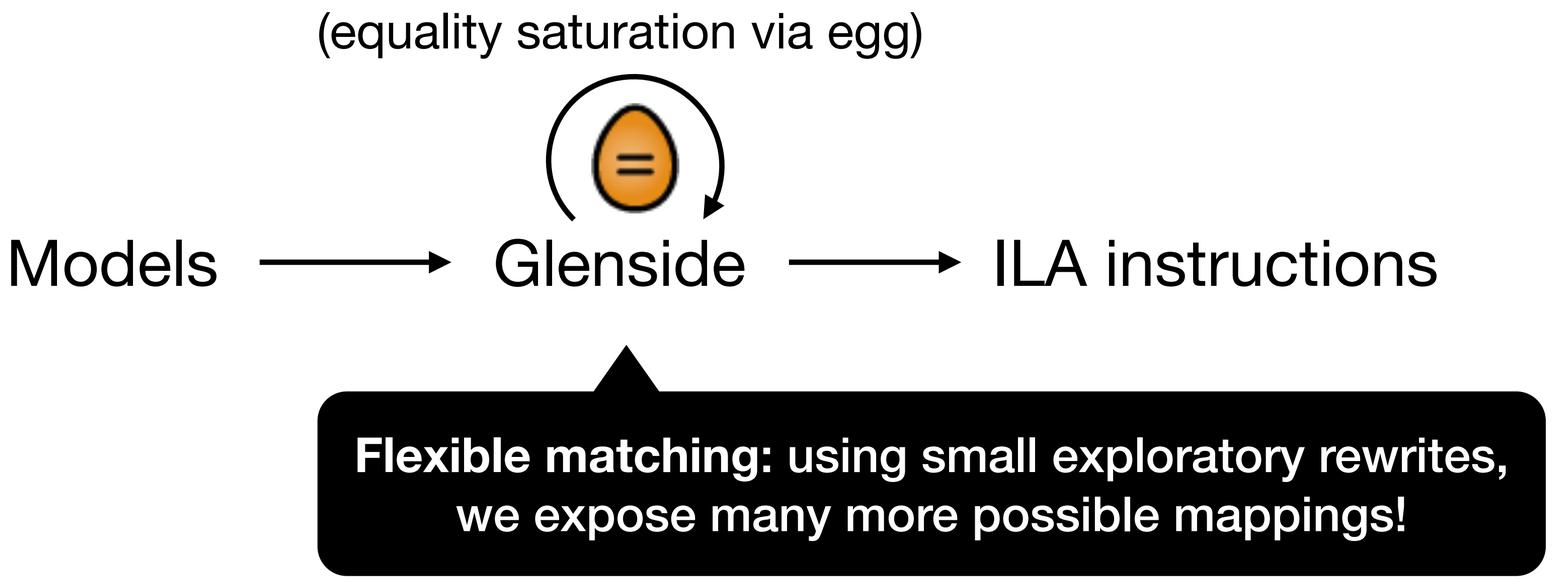
```
%242 = dense(%240, %241, units=10);
add(%242, %linear_bias)
```

Won’t match—this should be a `bias_add`!

```
%242 = dense(%240, %241, units=10);
add(%242, %linear_bias)
```

Won’t match—this should be a `bias_add`!
If only these rewrites were more flexible…

Let’s use Glenside and equality saturation!



Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. "egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

Gus Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. "Pure tensor program rewriting via access patterns (representation pearl)." MAPS 2021.

What is equality saturation?

Basic idea:

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality saturation: a new approach to optimization."
In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 264-276. 2009.

Basic idea:

instead of *destructively* rewriting a program with a predetermined list of program rewrites,

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality saturation: a new approach to optimization."
In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 264-276. 2009.

Basic idea:

instead of *destructively* rewriting a program with a predetermined list of program rewrites,

run *all* rewrites simultaneously and repeatedly,

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality saturation: a new approach to optimization."
In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 264-276. 2009.

Basic idea:

instead of *destructively* rewriting a program with a predetermined list of program rewrites,

run *all* rewrites simultaneously and repeatedly,

and keep *all* of the discovered versions of the program!

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality saturation: a new approach to optimization."
In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 264-276. 2009.

Enabled by the equality graph, or **egraph,** data structure!

Basic idea:

instead of *destructively* rewriting a program with a predetermined list of program rewrites,

run *all* rewrites simultaneously and repeatedly,

and keep *all* of the discovered versions of the program!

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality saturation: a new approach to optimization."
In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 264-276. 2009.

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1
```

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1
x / x ==> 1
```

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1

x / x ==> 1

x * 2 ==> x << 1
```

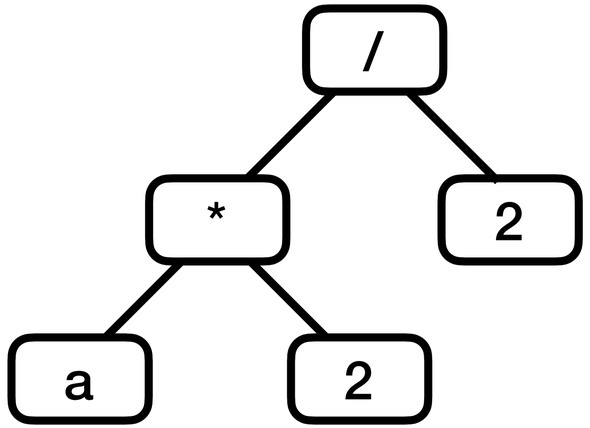
Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1
x / x ==> 1
x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
    x * 1 ==> 1
    x / x ==> 1
    x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```

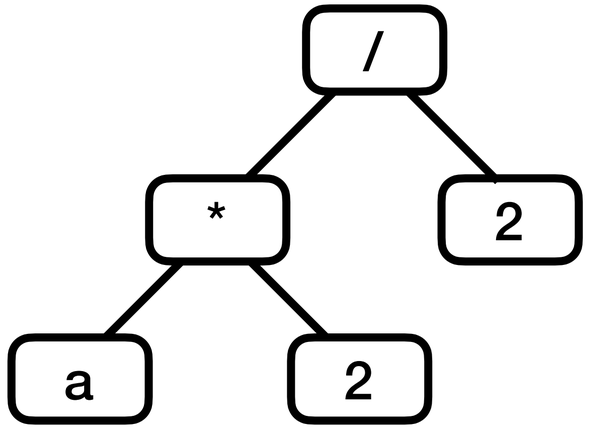
(a * 2)/2



Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
    x * 1 ==> 1
    x / x ==> 1
    x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```

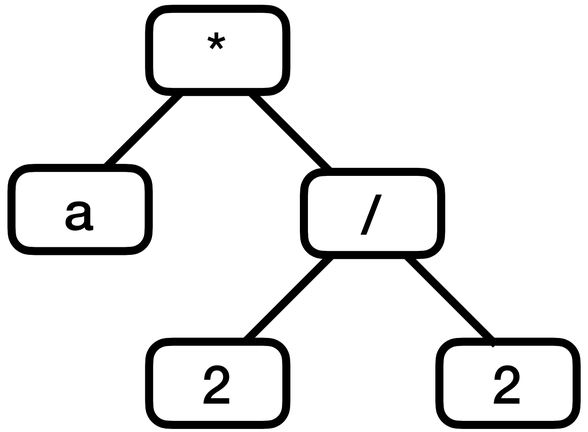
```
(a * 2)/2 == a
```



Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1
x / x ==> 1
x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```

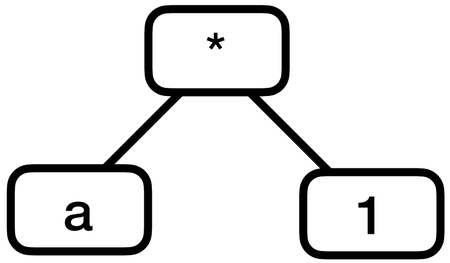
a * (2/2)



Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1
x / x ==> 1
x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```

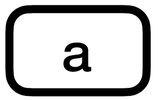
```
a * 1
```



Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1
x / x ==> 1
x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```

a

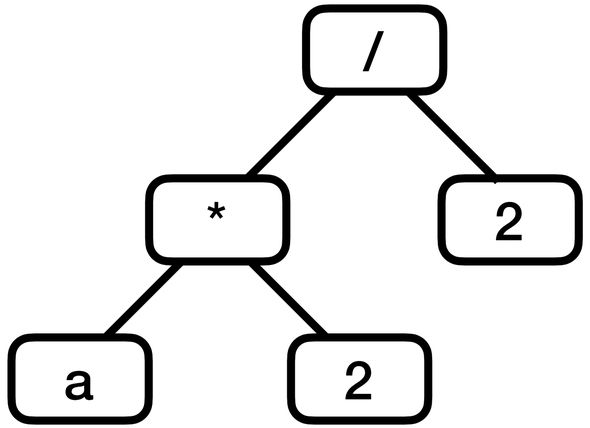


Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

But what if rewrites ran in a different order?

```
    x * 1 ==> 1
    x / x ==> 1
    x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```

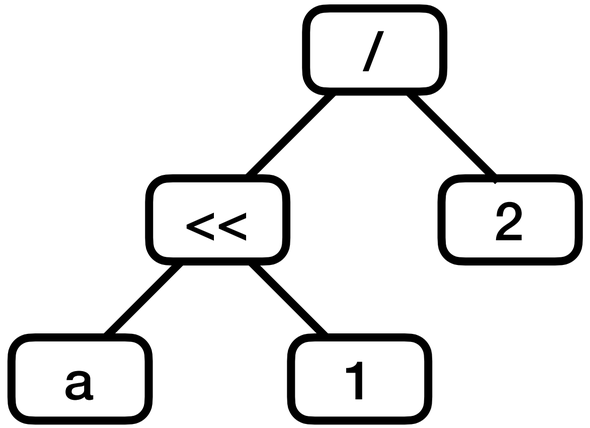
(a * 2)/2



Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
    x * 1 ==> 1
    x / x ==> 1
    x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```

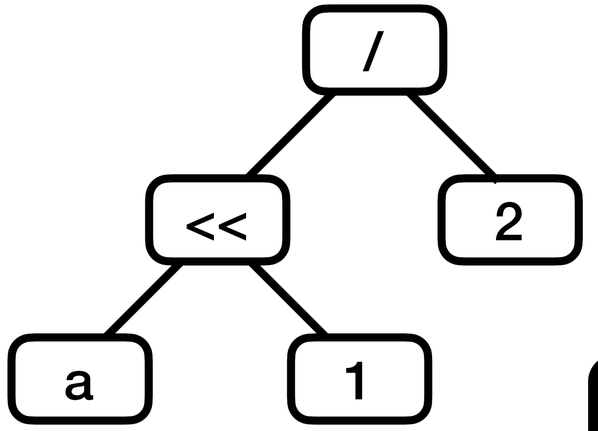
```
(a << 1)/2
```



Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1
x / x ==> 1
x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```

(a << 1)/2



We're stuck!

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

Ordering matters because rewrites are destructive.

Ordering matters because rewrites are destructive.

This is called the **phase ordering problem!**

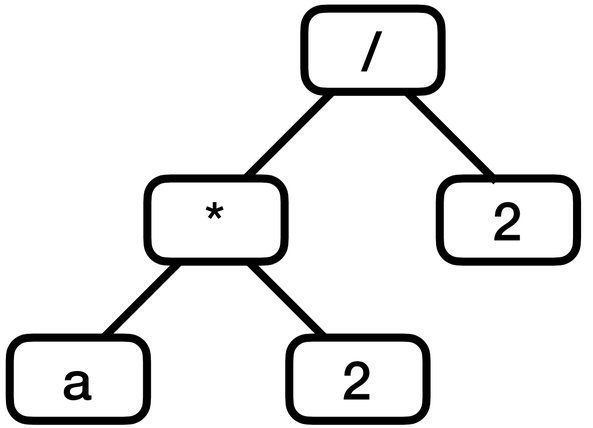
So why not keep around all discovered versions of the program?

So why not keep around all discovered versions of the program?

This is what egraphs do!

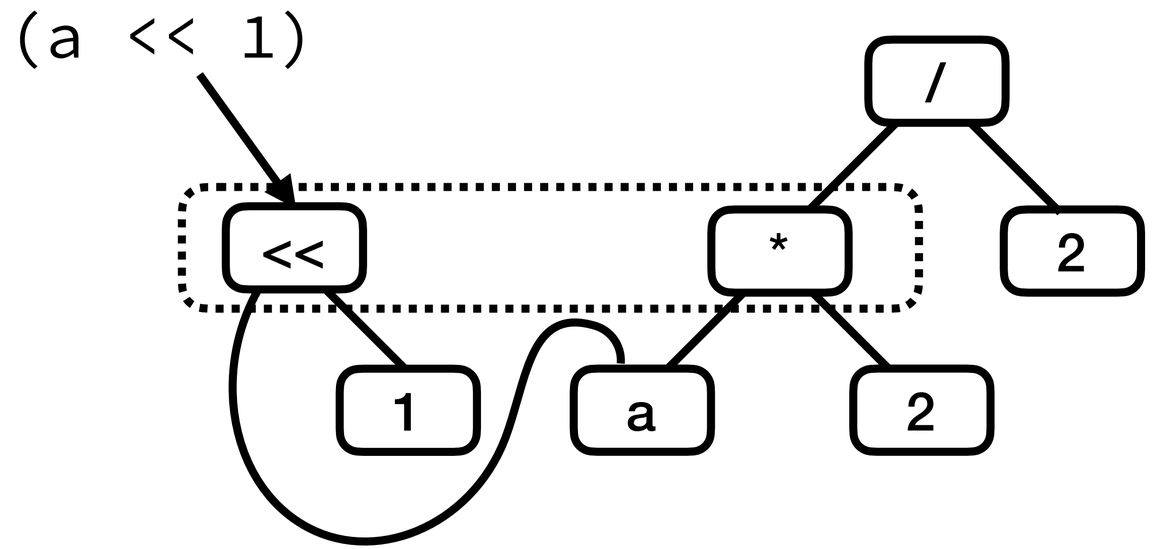
```
    x * 1 ==> 1
    x / x ==> 1
    x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```

(a * 2)/2



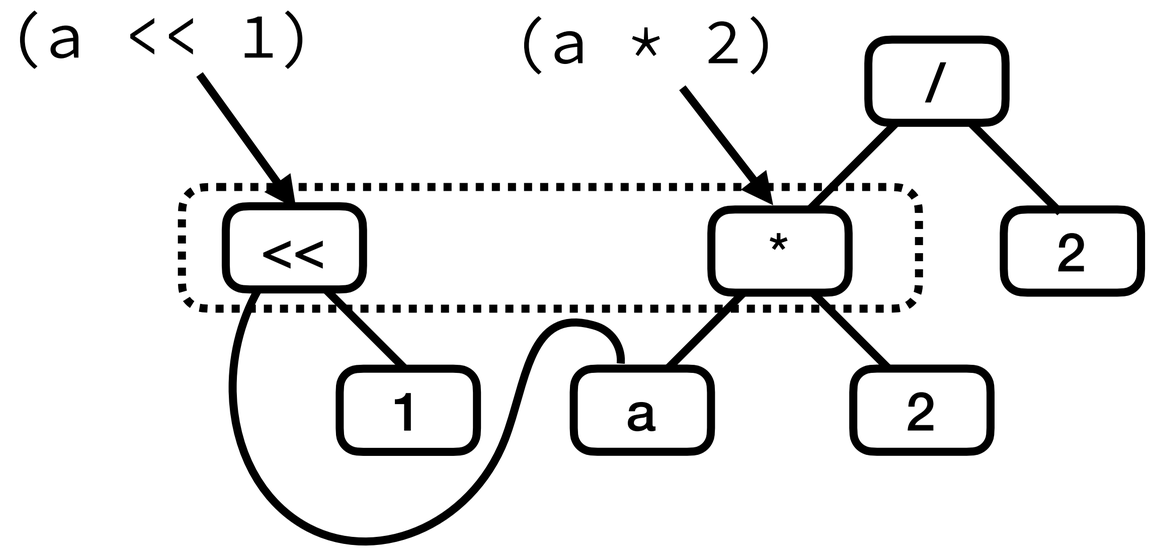
Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1
x / x ==> 1
x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```



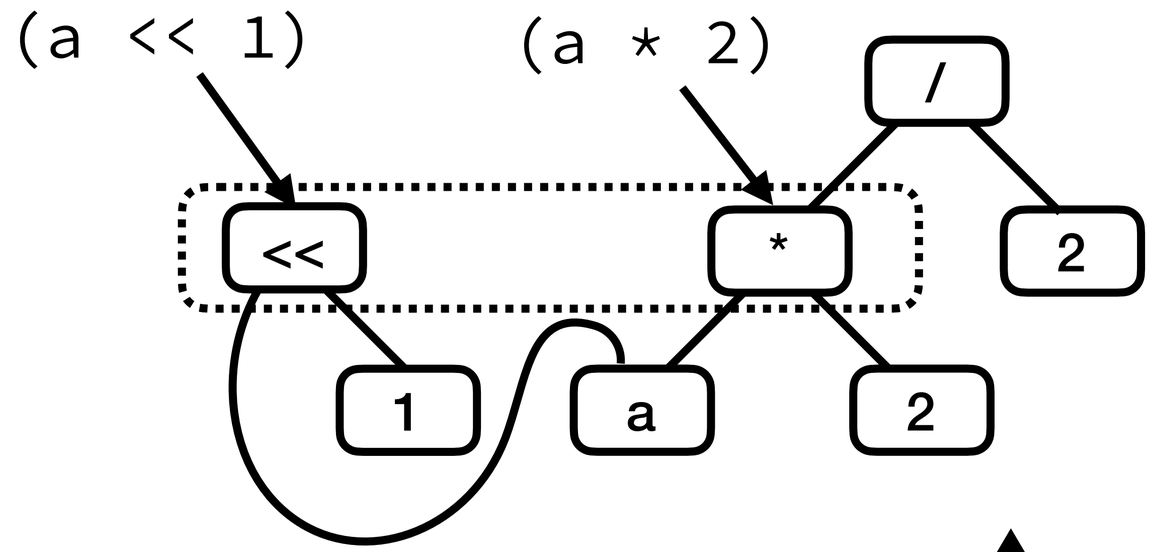
Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1
x / x ==> 1
x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```



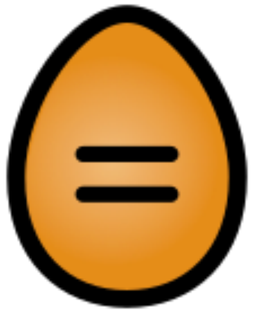
Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

```
x * 1 ==> 1
x / x ==> 1
x * 2 ==> x << 1
(x * y)/z ==> x * (y/z)
```



We can fire the rewrites in any order—
all discovered programs will be kept!

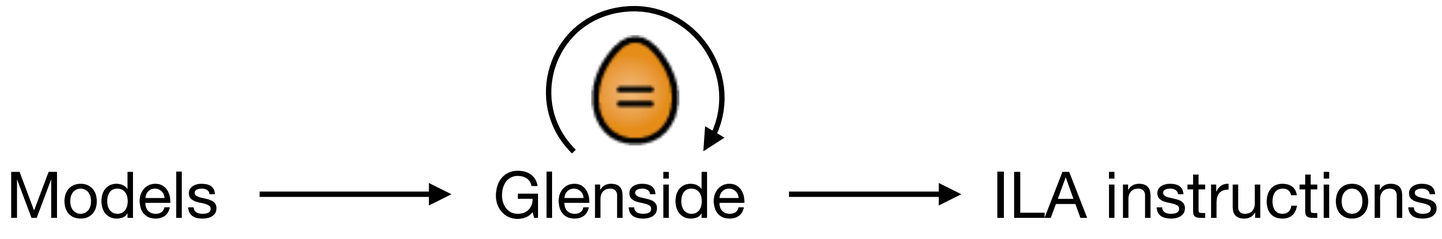
Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).



https://egraphs-good.github.io/

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.
"egg: Fast and extensible equality saturation." *Proceedings of the ACM on Programming Languages* 5, no. POPL (2021).

(equality saturation via egg)



Flexible matching: using small exploratory rewrites, we expose many more possible mappings!

We capture accelerator semantics as program rewrites…

```
(compute dot-product (access-cartesian-product ?x ?w))
  => (accelerator-call vta-dense ?x ?w)

(compute reduce-max (access-windows ?a (shape 2) (shape 2)))
  => (accelerator-call flex-maxpool ...)

(bias-add (dense ?x ?w) ?bias ?axis)
  => (accelerator-call flex-linear ?x ?w ?bias)
```

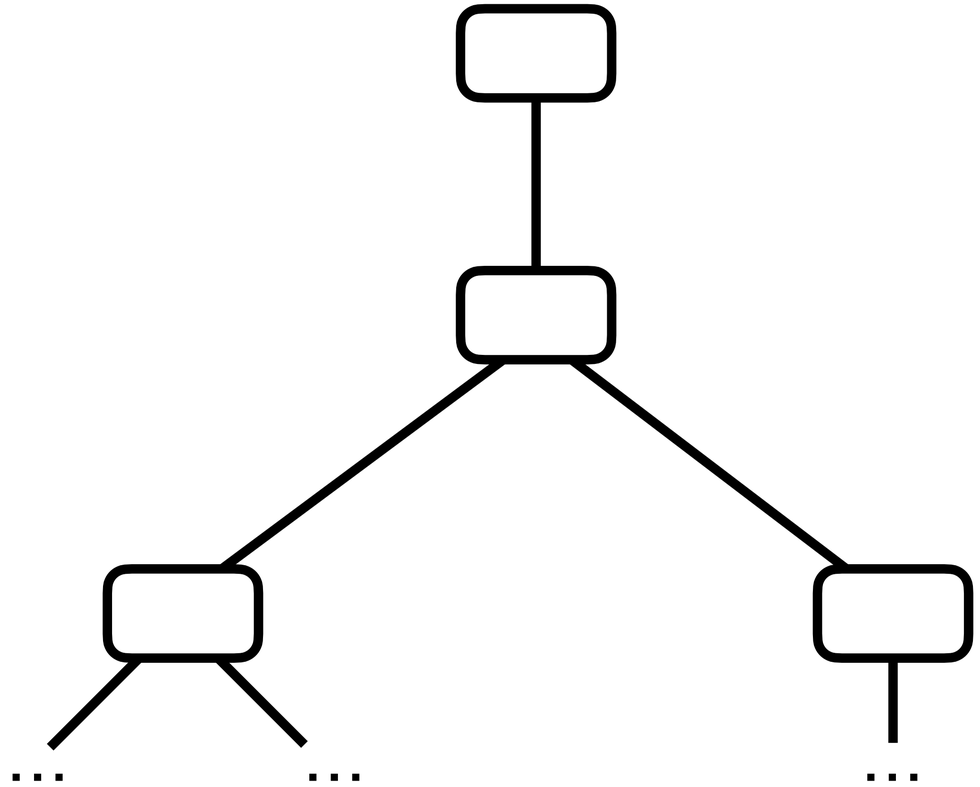
We capture accelerator semantics as program rewrites…

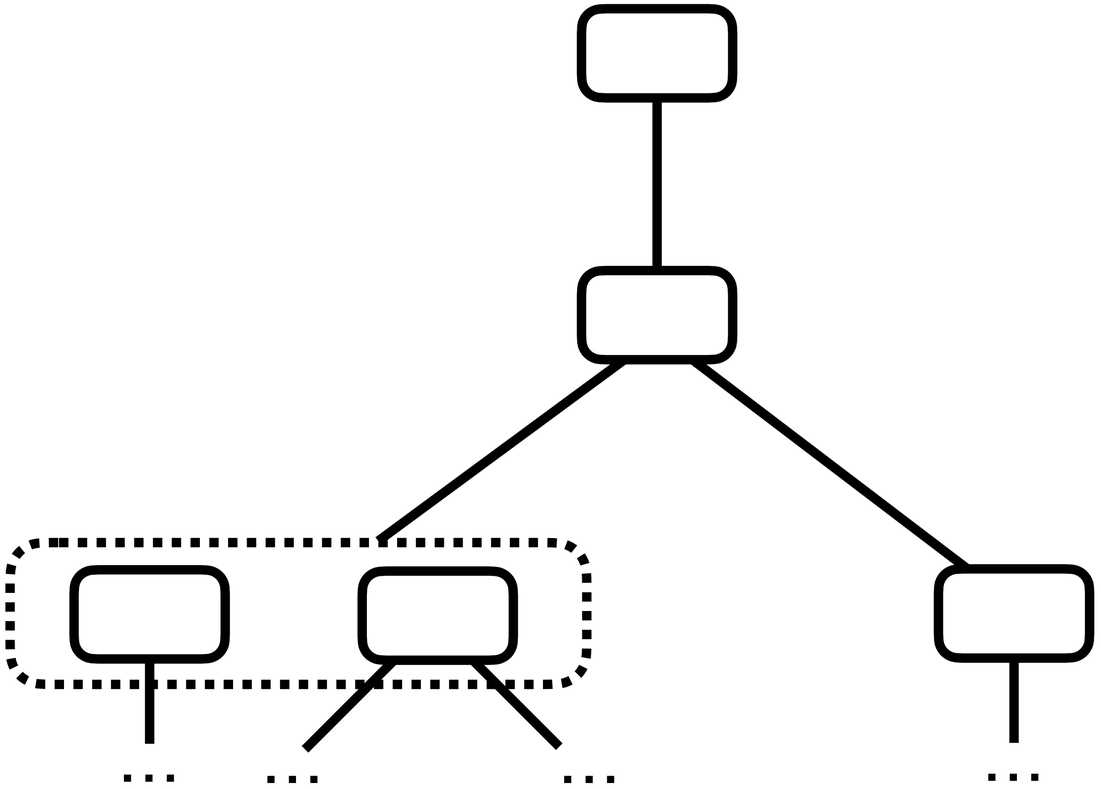
```
(compute dot-product (access-cartesian-product ?x ?w))
  => (accelerator-call vta-dense ?x ?w)

(compute reduce-max (access-windows ?a (shape 2) (shape 2)))
  => (accelerator-call flex-maxpool ...)

(bias-add (dense ?x ?w) ?bias ?axis)
  => (accelerator-call flex-linear ?x ?w ?bias)
```

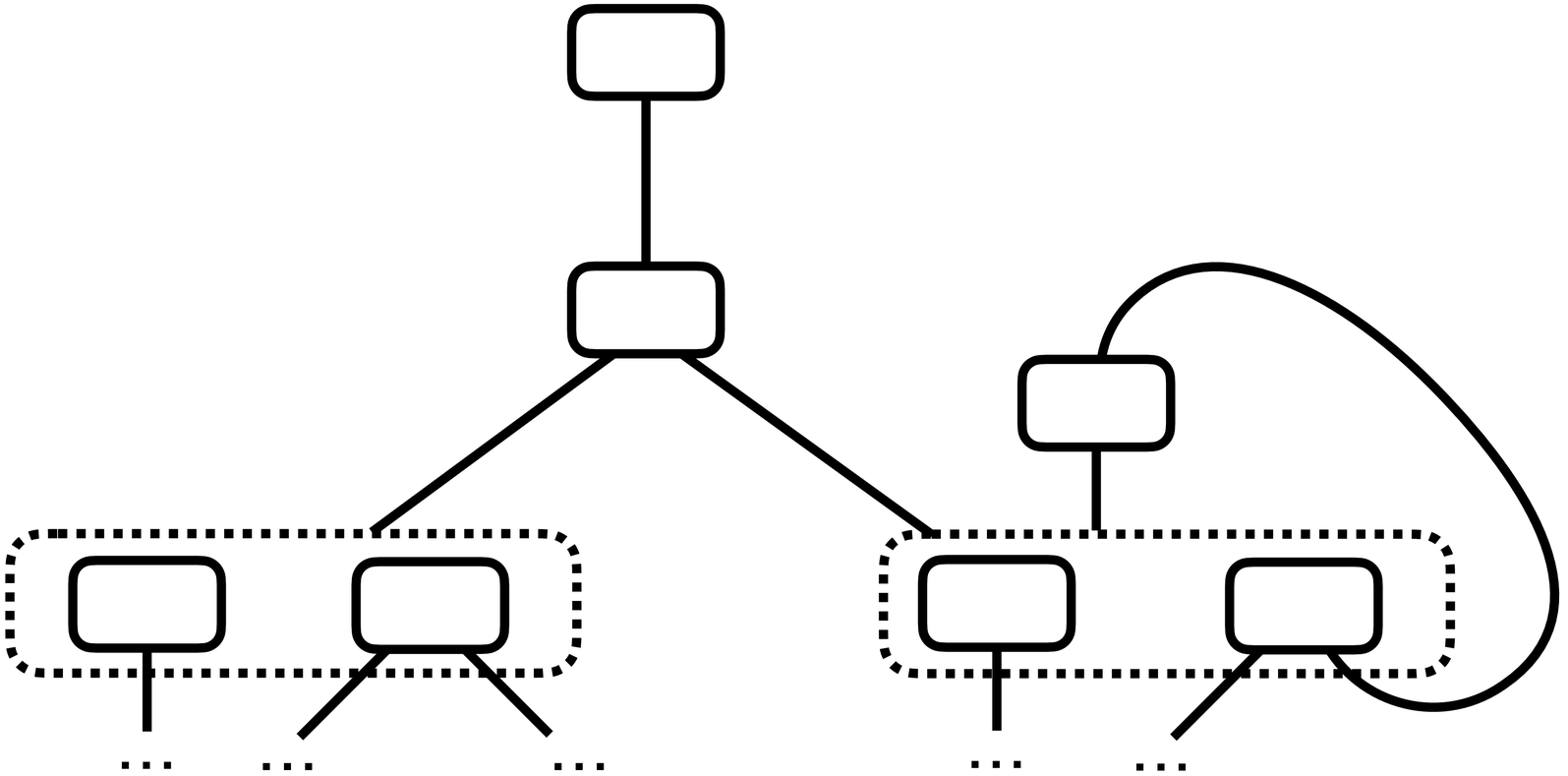
…and our exploratory rewrites are general-purpose rewrites over Glenside!

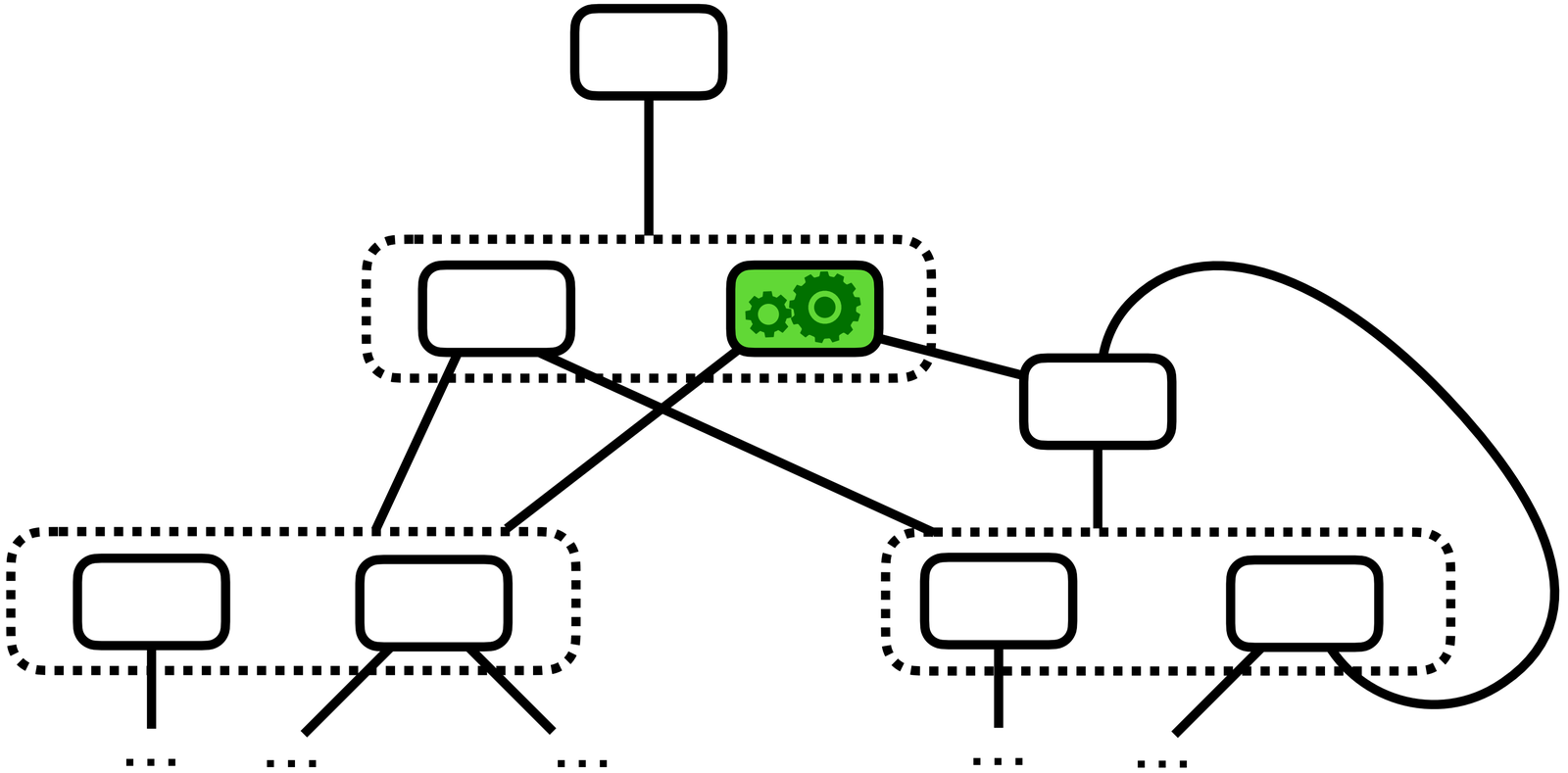
```
?x => (relay-operator-call bias-add ?x
        (relay-operator-call zeros (shape ...)) 1)
?a => (reshape (flatten ?a) ?shape)

(cartProd (reshape ?a0 ?shape0) (reshape ?a1 ?shape1))
  => (reshape (cartProd ?a0 ?a1) ?newShape)

(compute dotProd (reshape ?a ?shape))
  => (reshape (compute dotProd ?a) ?newShape)
```

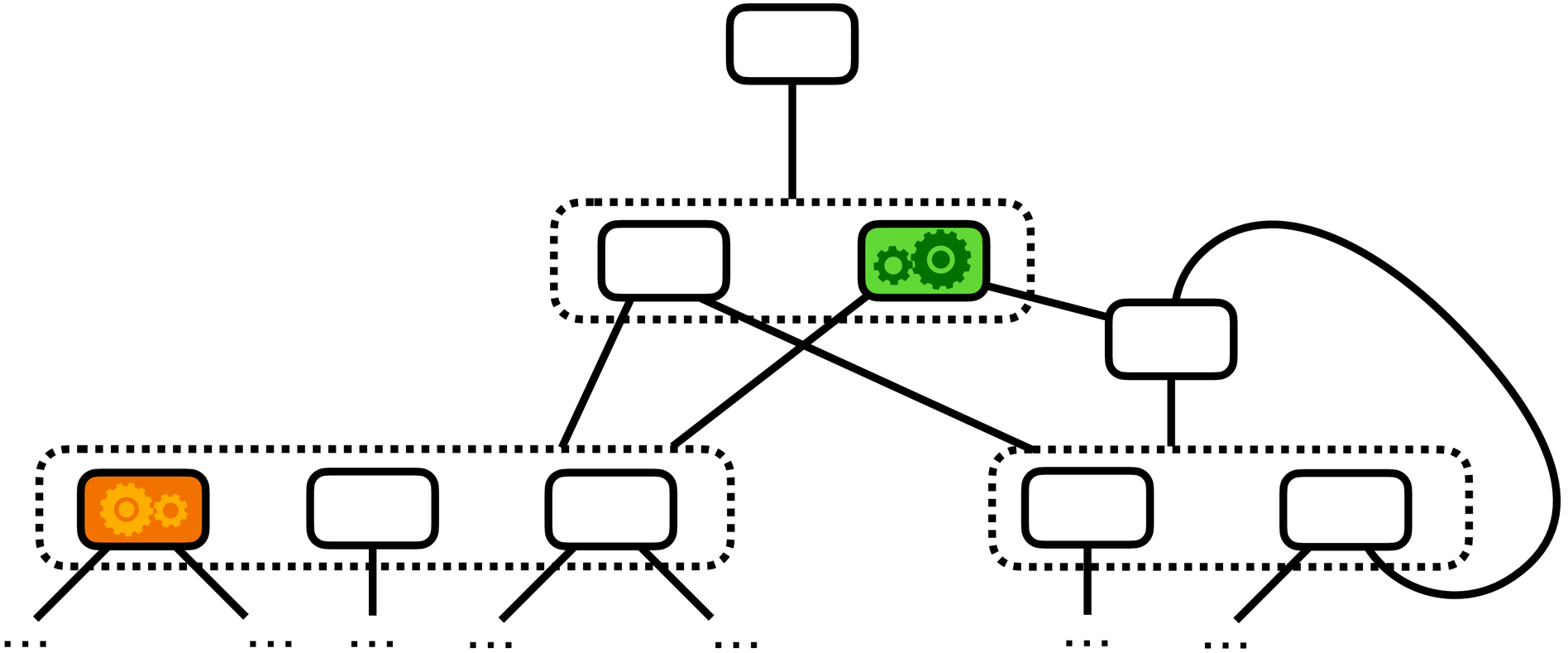
⋮











	EfficientNet	MobileNet V2	ResMLP	ResNet-20	Transformer
VTA	0 → 35	1 → 41	38	2 → 22	66
FlexASR	0 → 35	0 → 41	0 → 38	2 → 22	0 → 66

```
?a → (reshape (flatten ?a) ?shape)
```

```
(cartProd (reshape ?a0 ?shape0) (reshape ?a1 ?shape1))
  → (reshape (cartProd ?a0 ?a1) ?newShape)
```

```
(compute dotProd (reshape ?a ?shape))
  → (reshape (compute dotProd ?a) ?newShape)
```

These rewrites *rediscover* the im2col transformation, without explicitly encoding it!

What does it show about Glenside?

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations, im2col
- reduces development time, and
- enables verification.

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations, im2col
- reduces development time, and Writing rewrites is simpler than writing a mapper from scratch!
- enables verification.

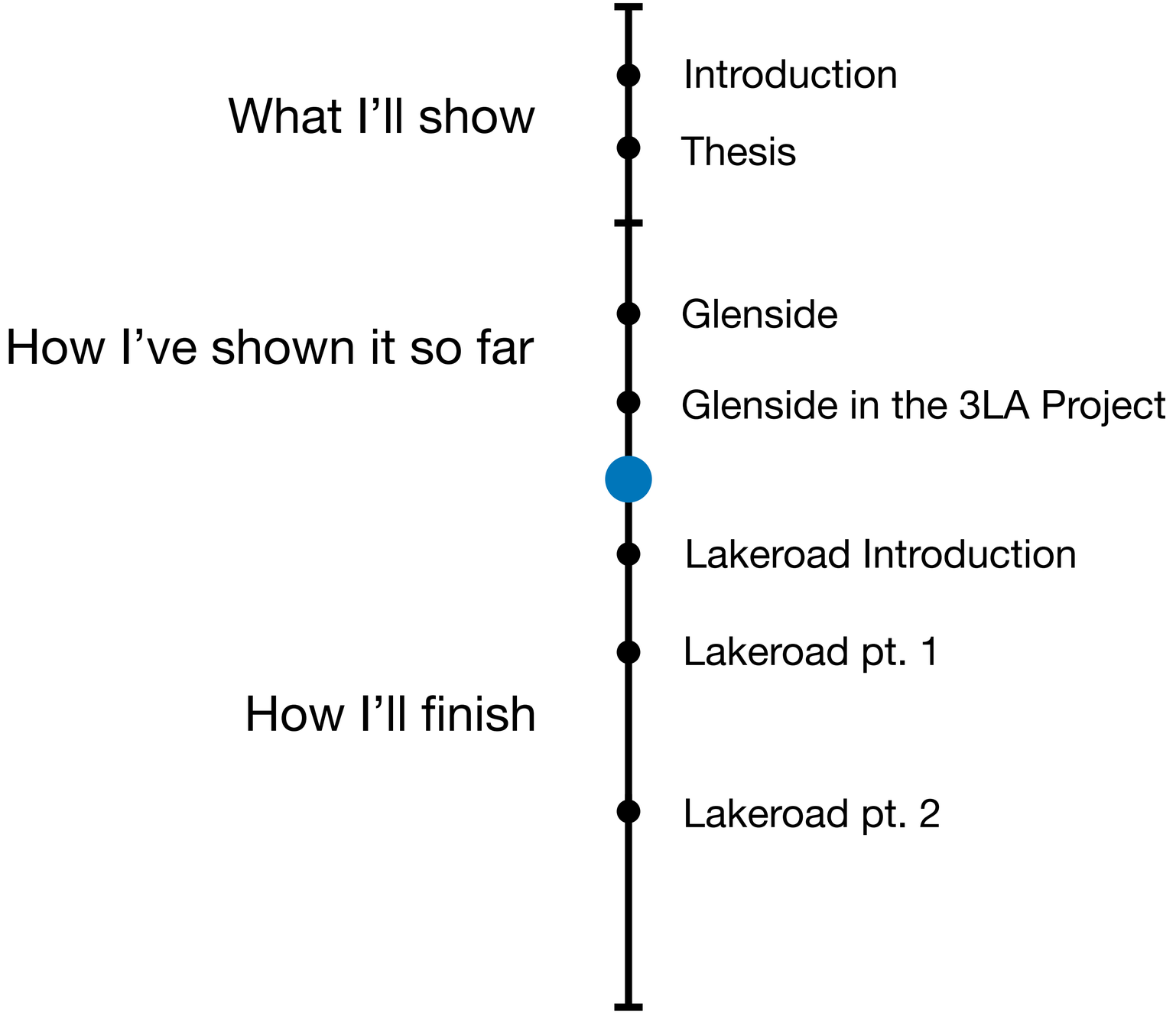
Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

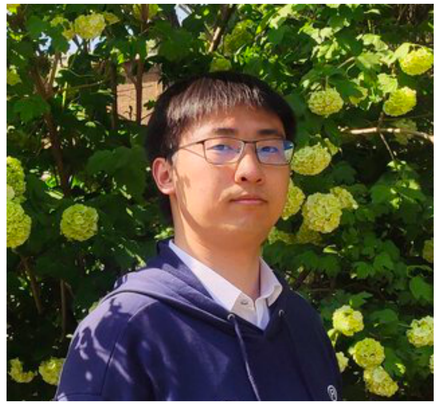
im2col

Writing rewrites is simpler than writing a mapper from scratch!

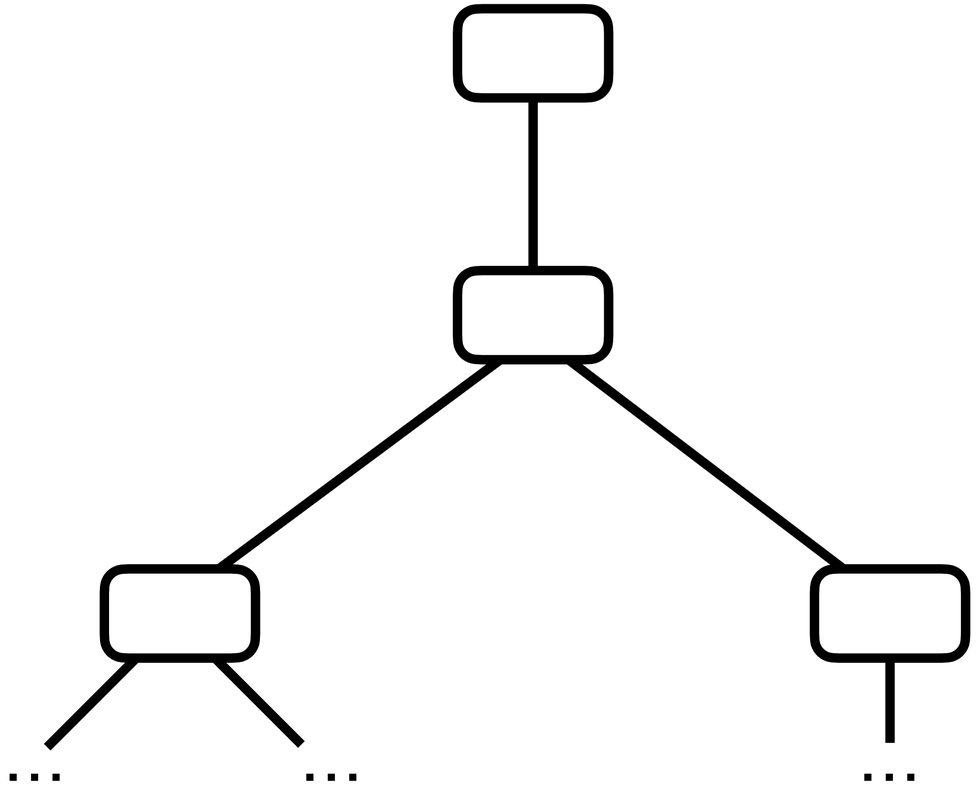
Via mapping to ILA!

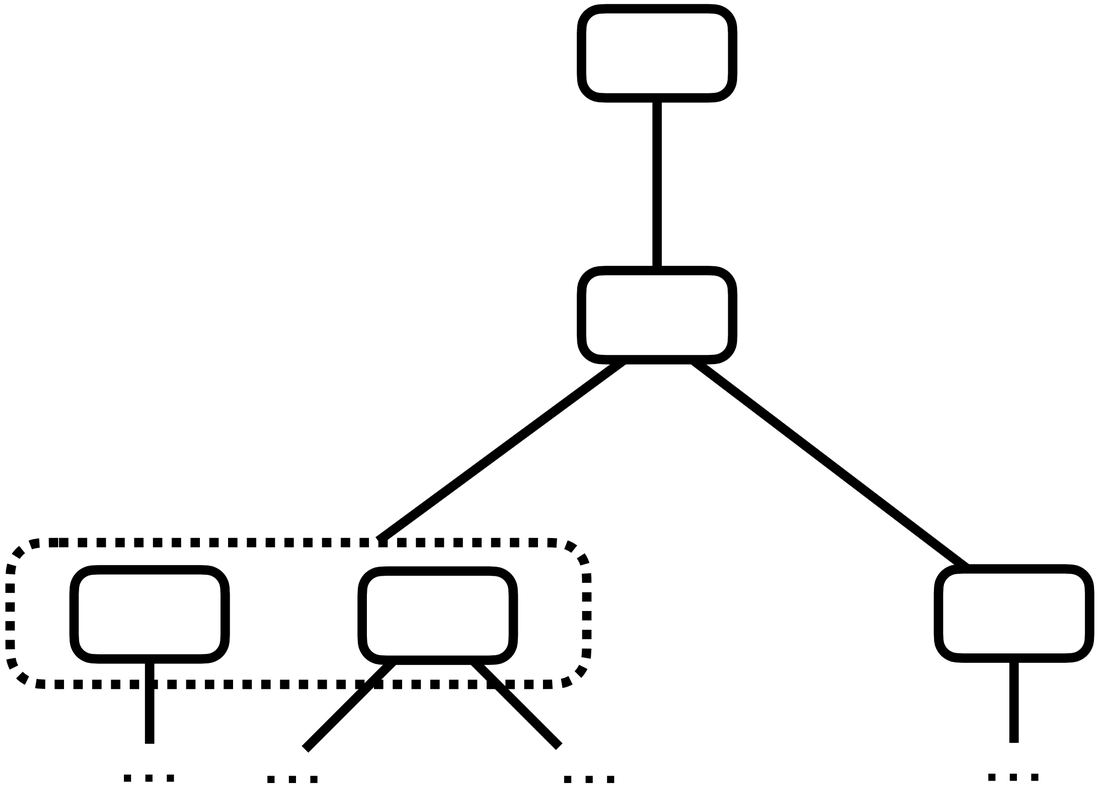

What I’ll show
Introduction
Thesis
How I’ve shown it so far
Glenside
Glenside in the 3LA Project
Lakeroad Introduction
How I’ll finish
Lakeroad pt. 1
Lakeroad pt. 2

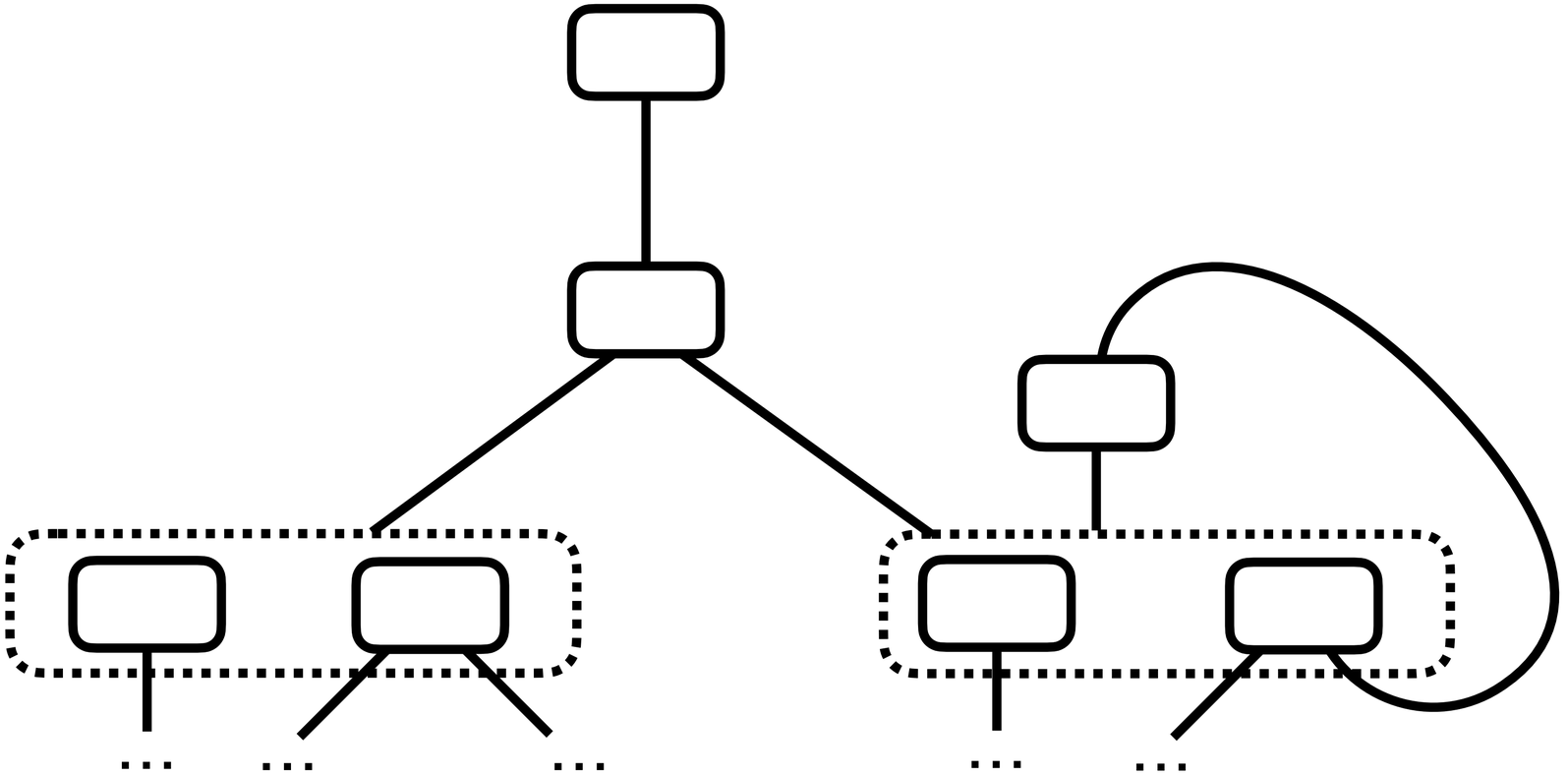
Lakeroad

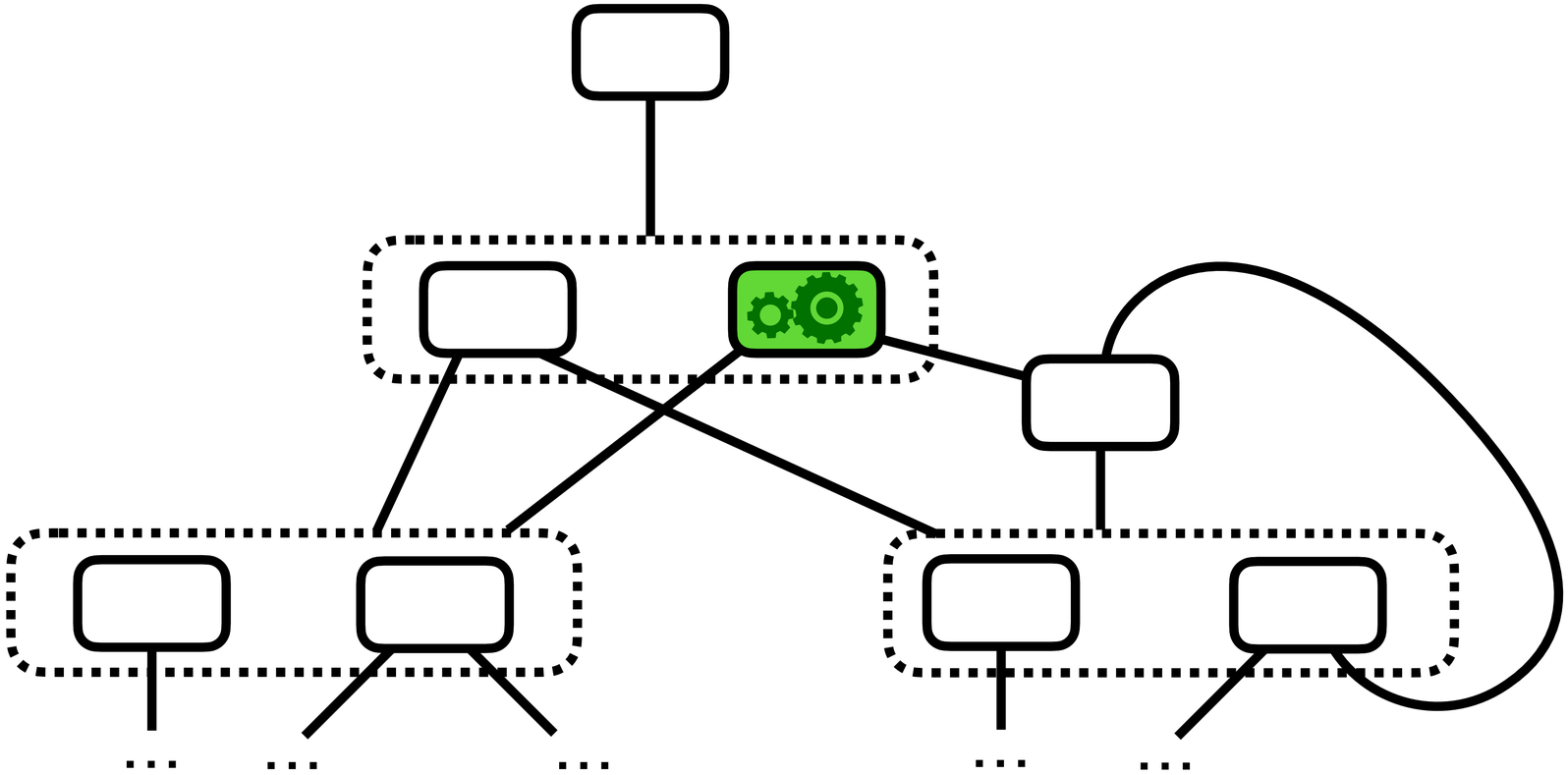


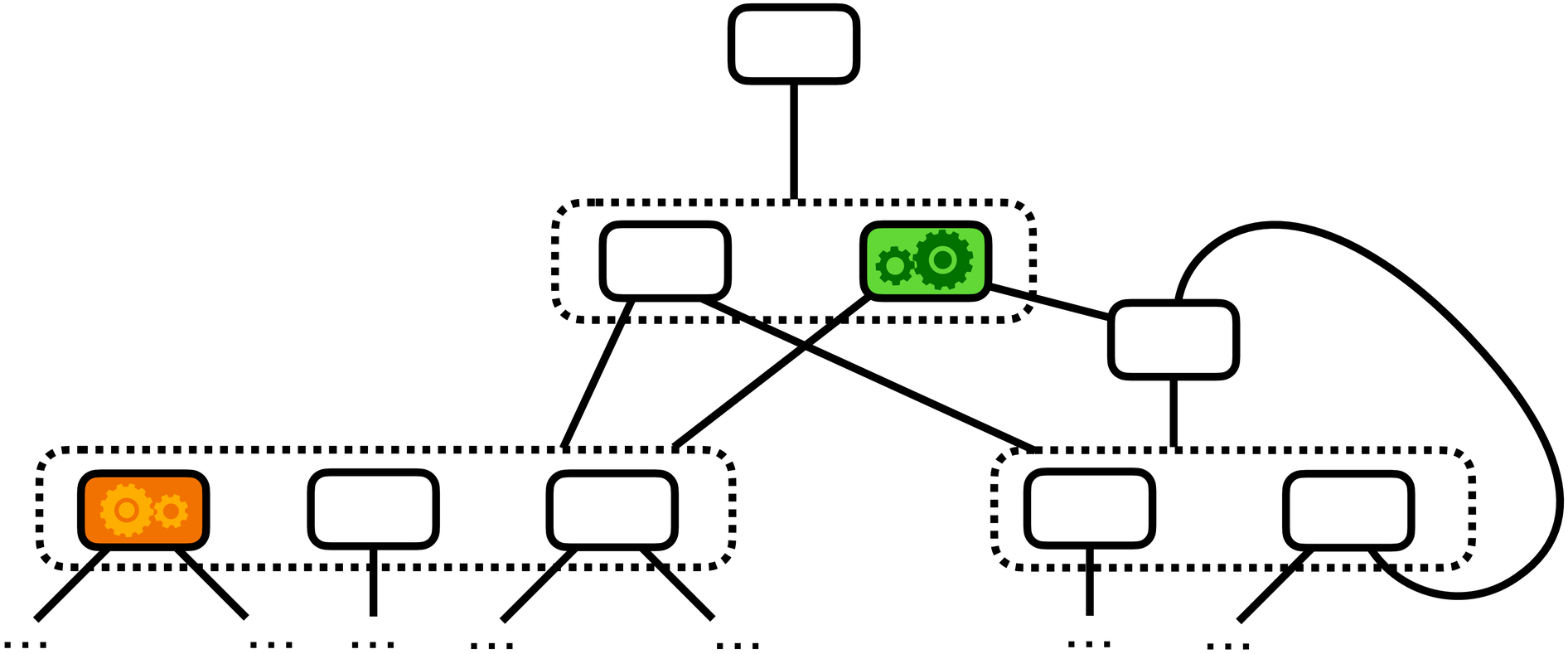


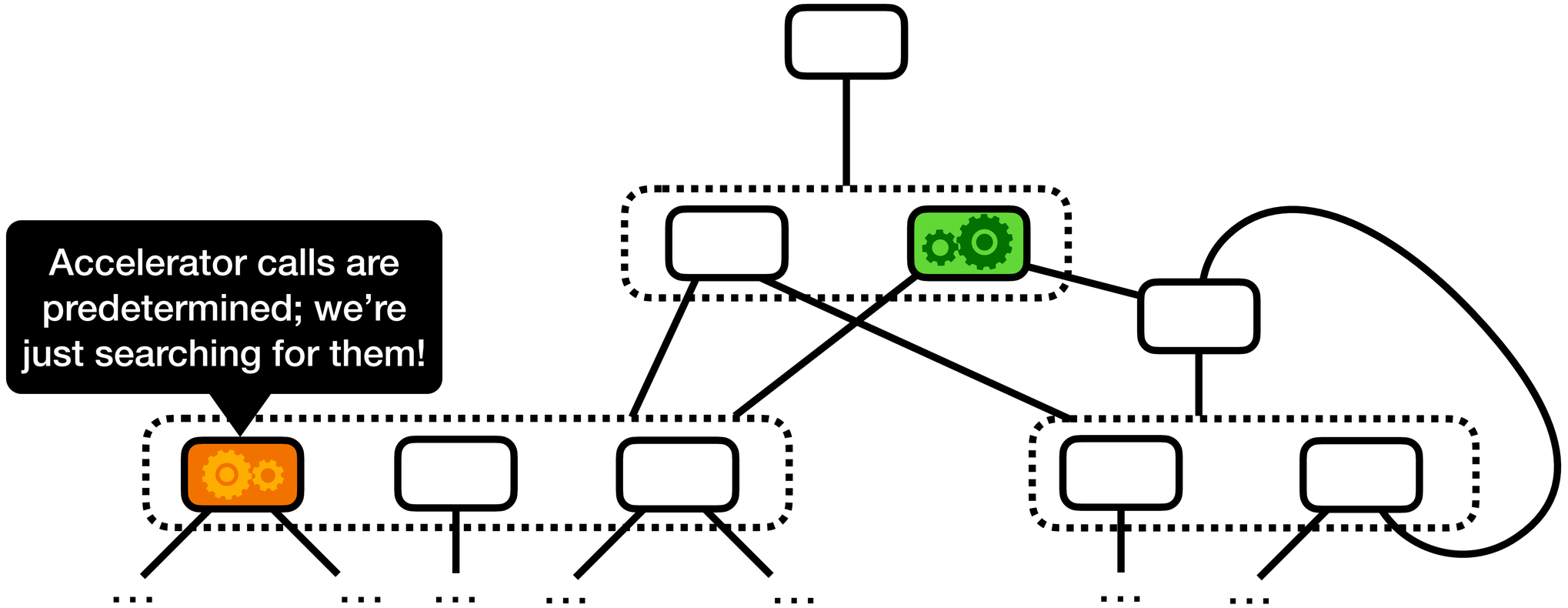


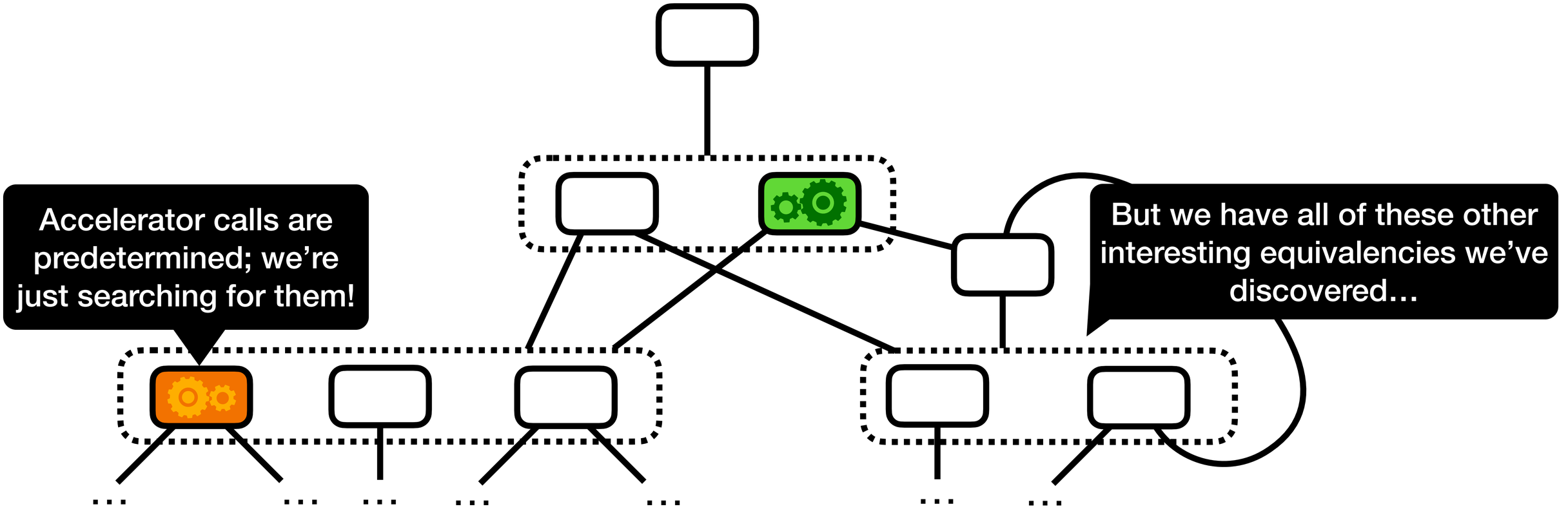


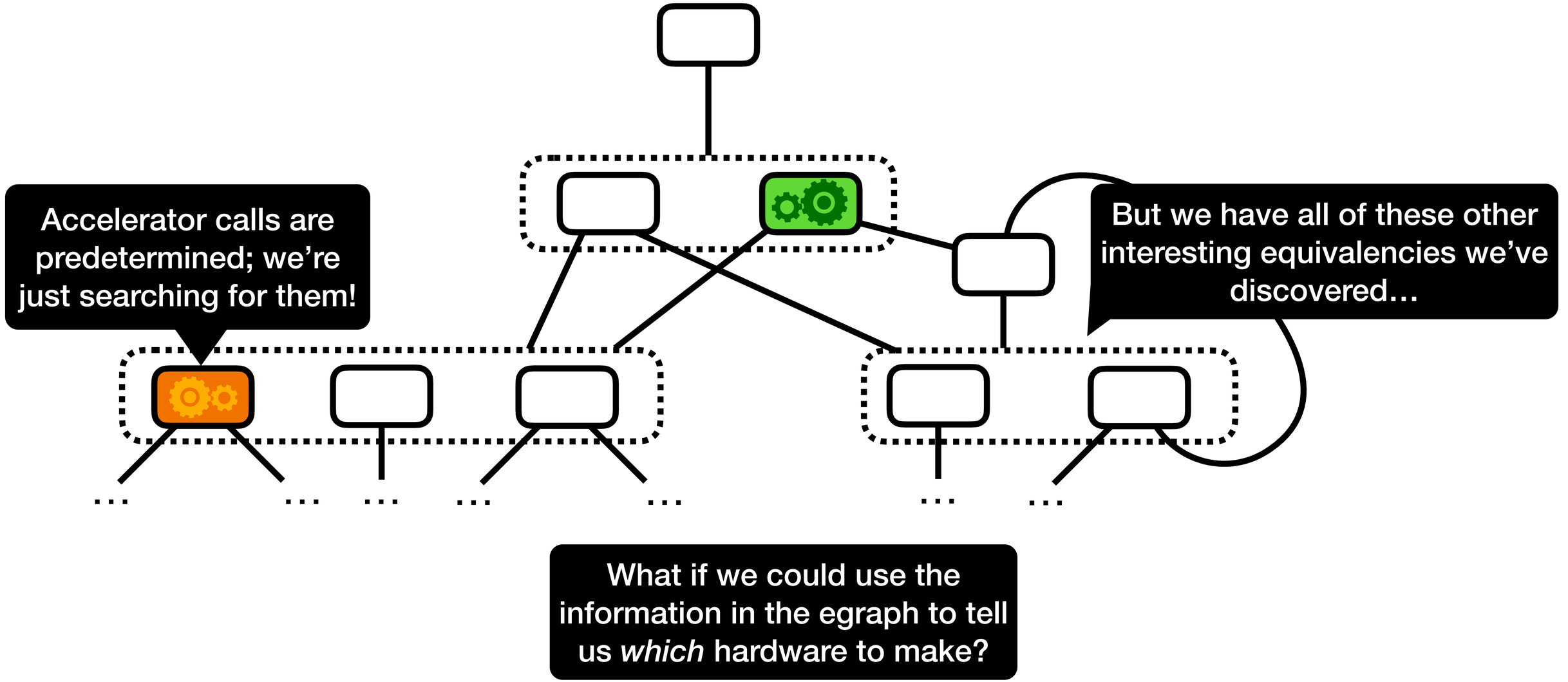


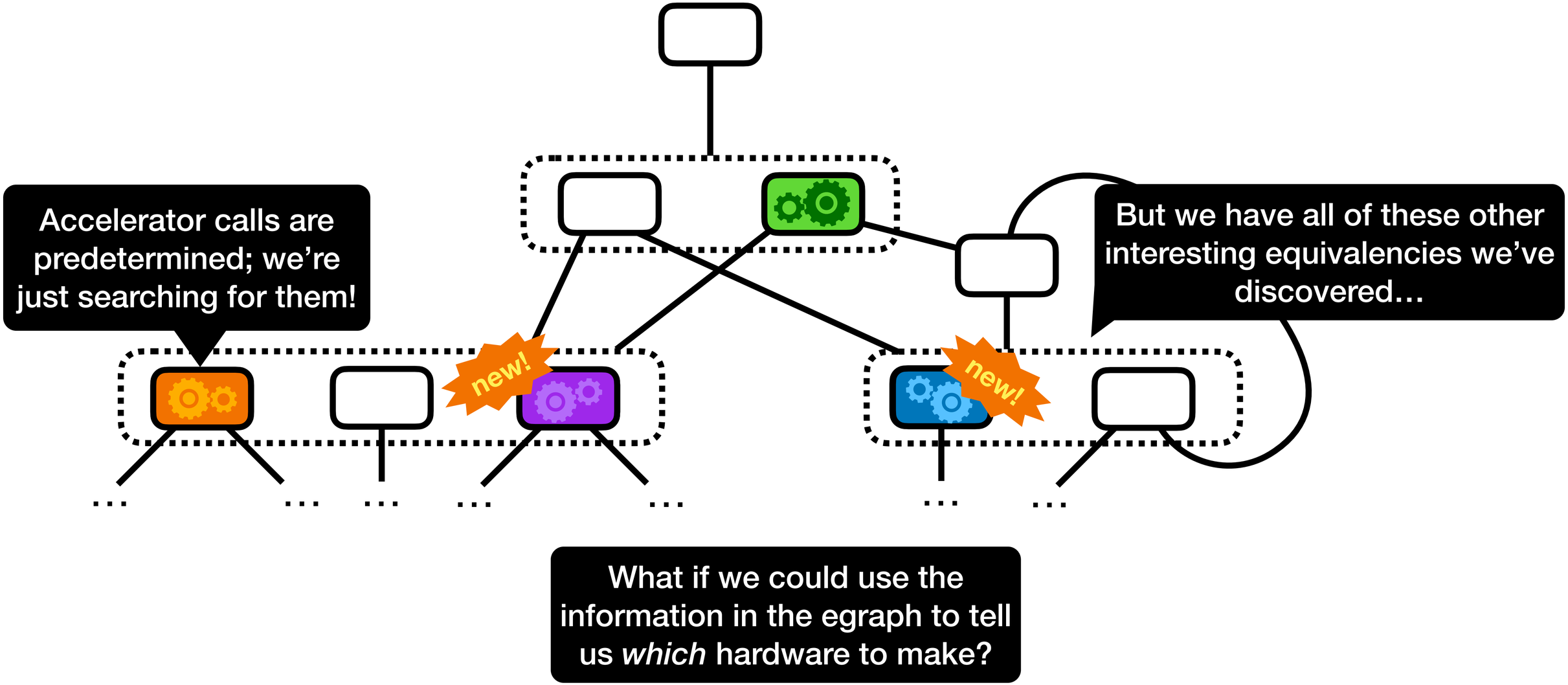





Accelerator calls are predetermined; we're just searching for them!


Accelerator calls are predetermined; we're just searching for them!
But we have all of these other interesting equivalencies we've discovered…


Accelerator calls are predetermined; we're just searching for them!
But we have all of these other interesting equivalencies we've discovered…
What if we could use the information in the egraph to tell us *which* hardware to make?


Accelerator calls are predetermined; we're just searching for them!
new!
new!
But we have all of these other interesting equivalencies we've discovered…
What if we could use the information in the egraph to tell us *which* hardware to make?

Lakeroad uses similar techniques to Glenside (i.e. equality saturation) to map computation to custom hardware—in this case, FPGAs.

Lakeroad uses similar techniques to Glenside (i.e. equality saturation) to map computation to custom hardware—in this case, FPGAs.

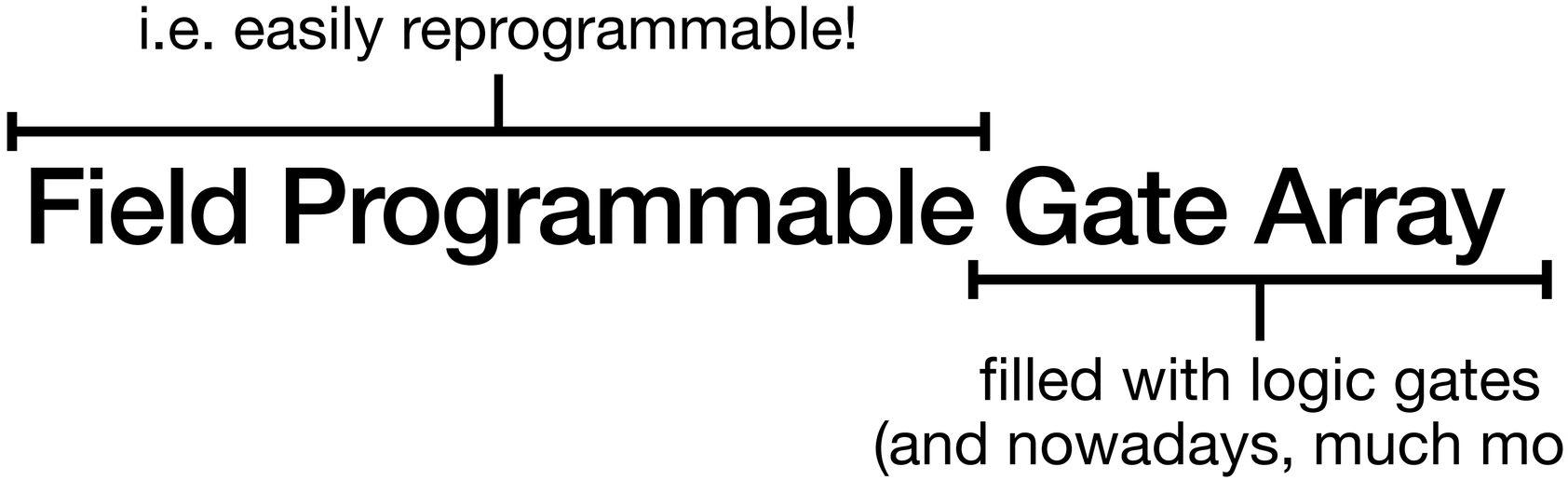
However, Lakeroad additionally uses what it discovers to propose entirely *new* hardware primitives!

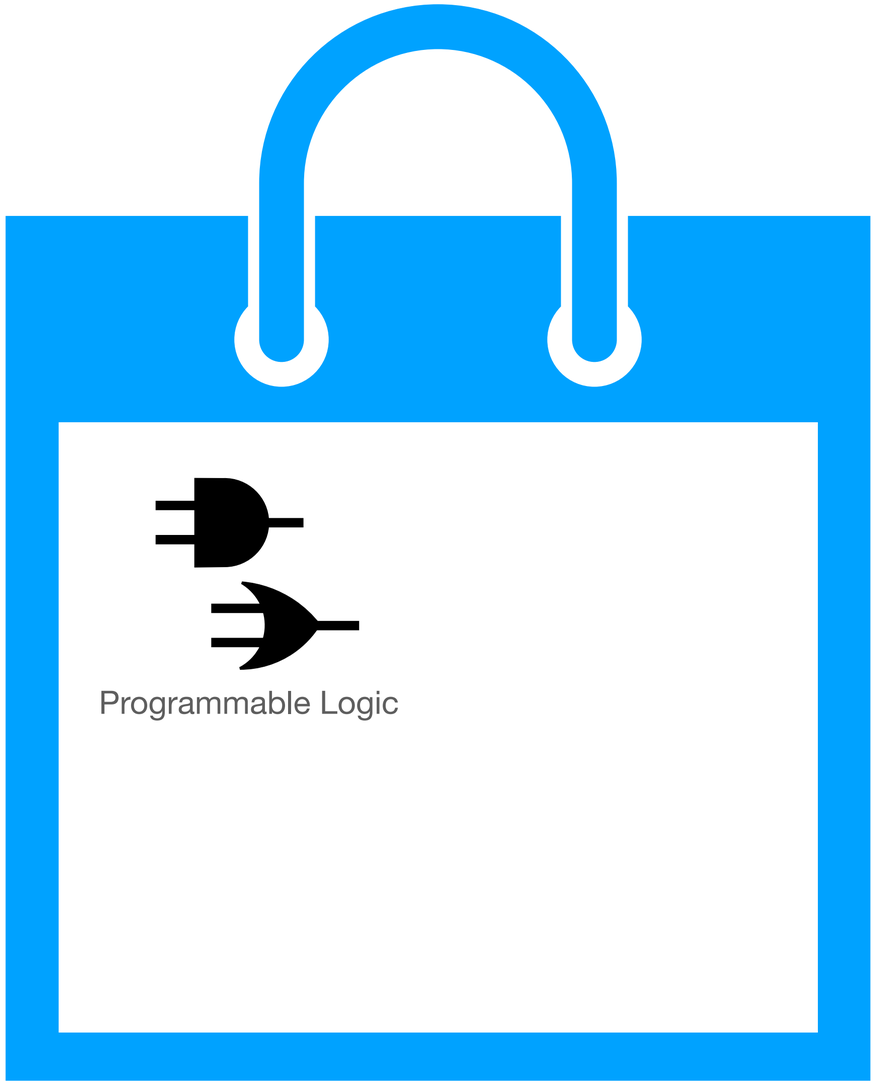
What are FPGAs?

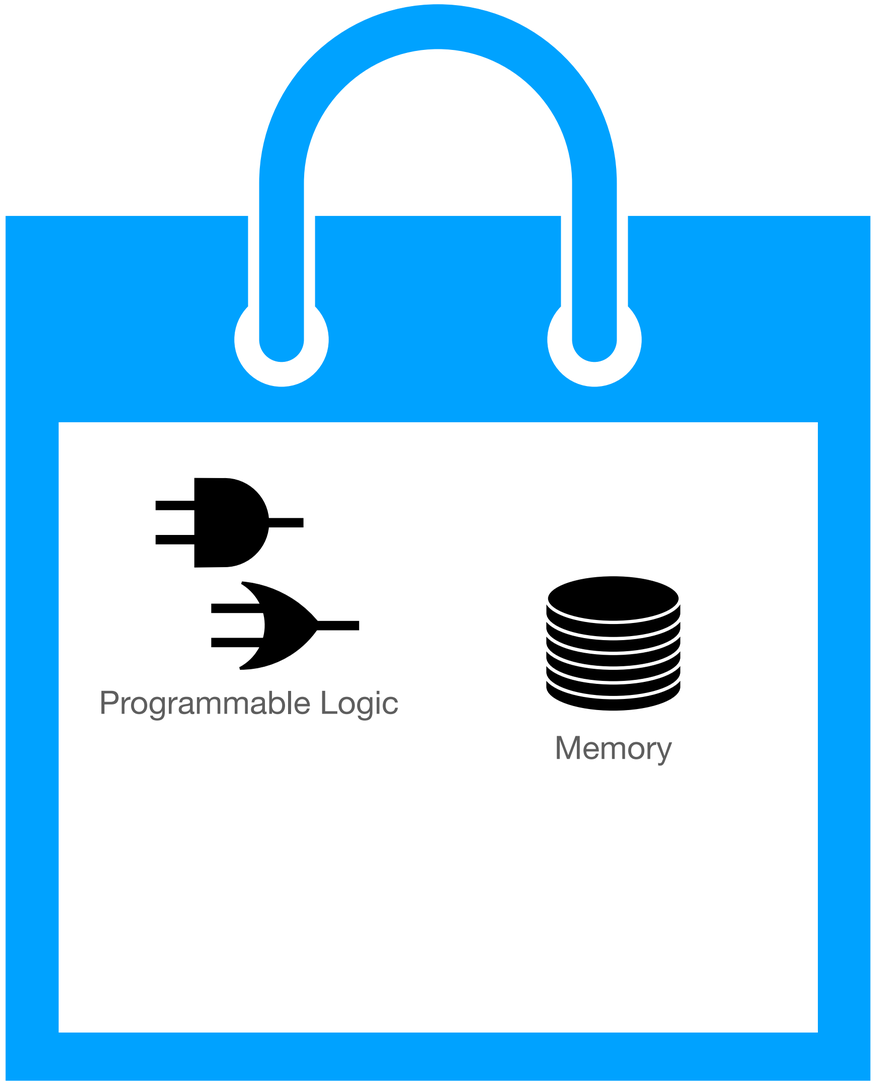
Field Programmable Gate Array

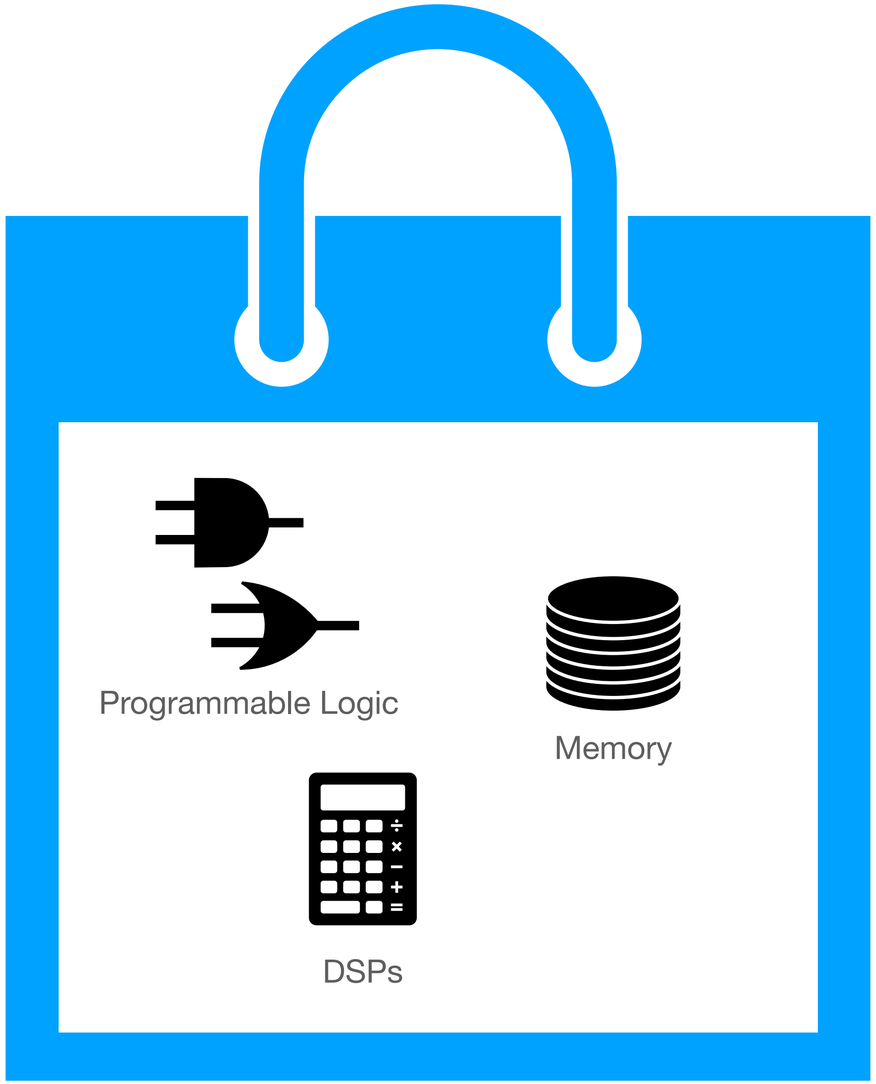
i.e. easily reprogrammable!

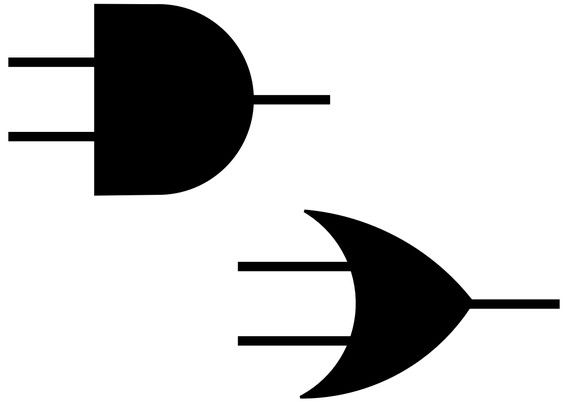
Field Programmable Gate Array

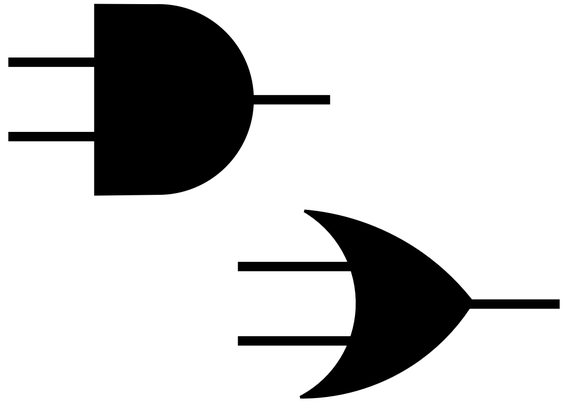

i.e. easily reprogrammable!
Field Programmable Gate Array
filled with logic gates
(and nowadays, much more!)

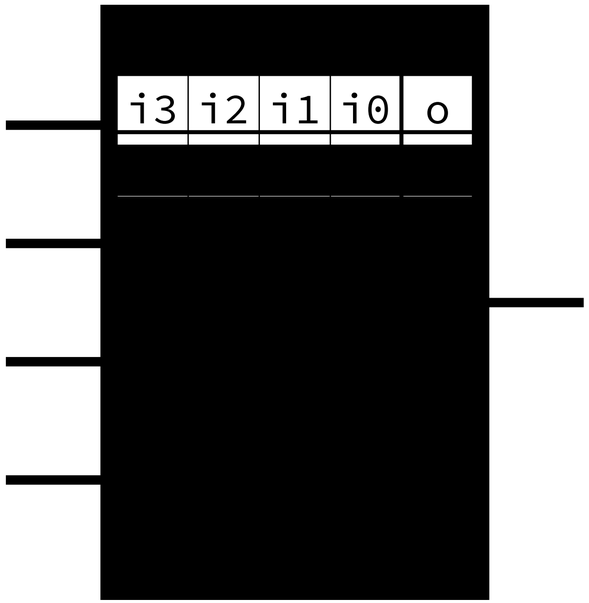

Programmable Logic

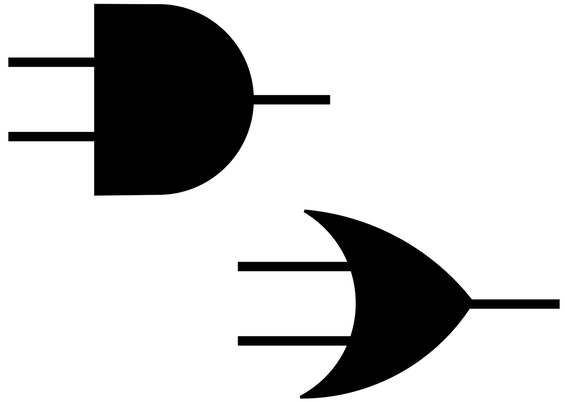

Programmable Logic
Memory

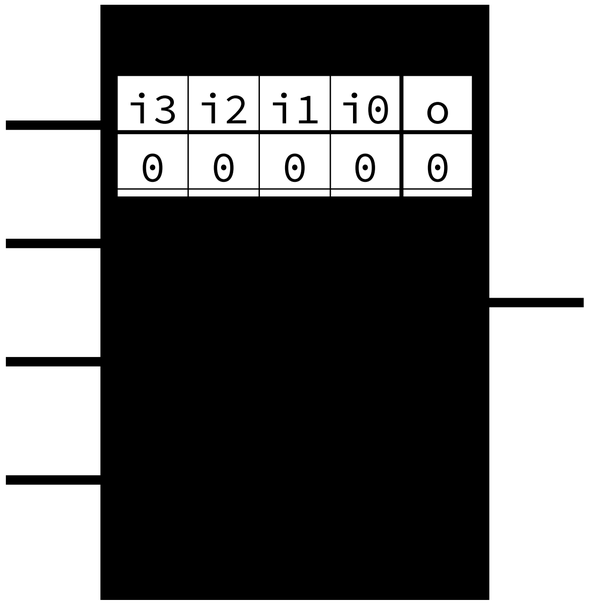

Programmable Logic
Memory
DSPs

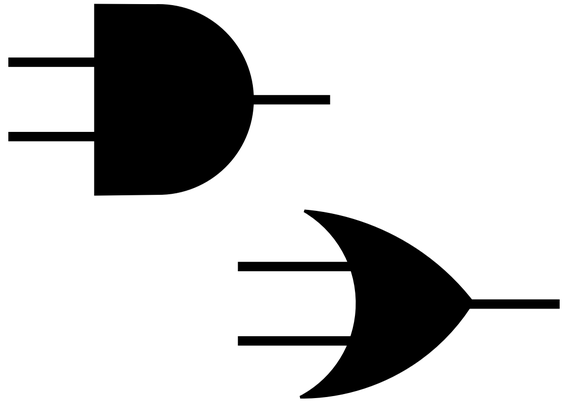


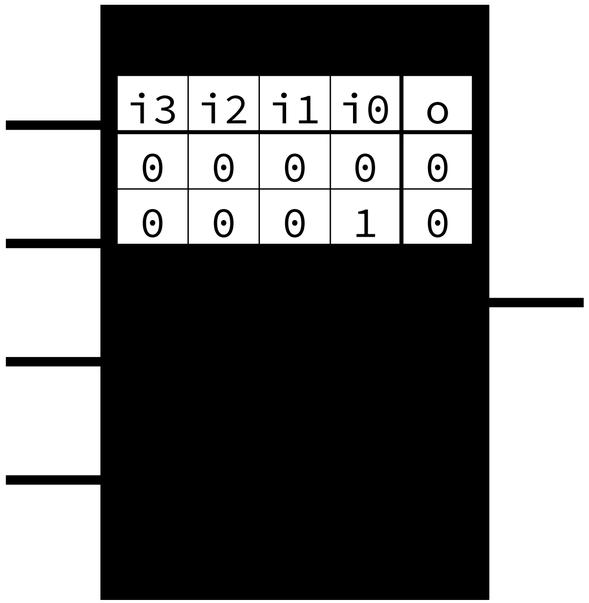



i3
i2
i1
i0
o

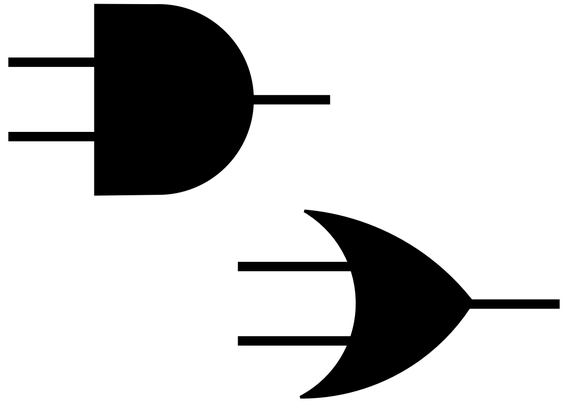


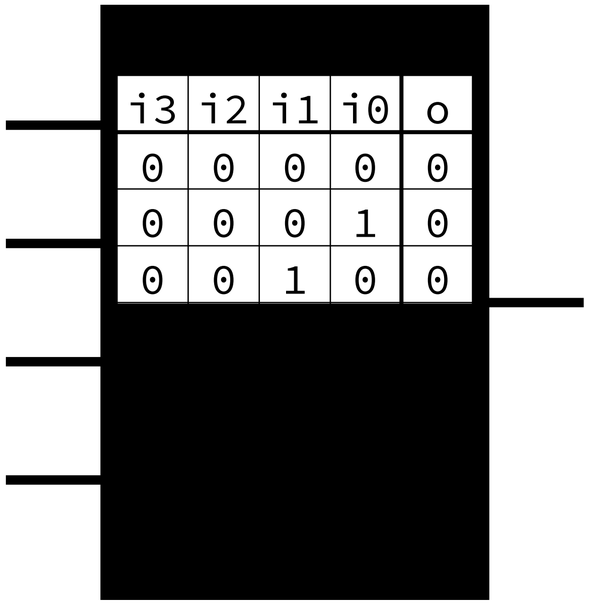

i3 i2 i1 i0 o
0 0 0 0 0



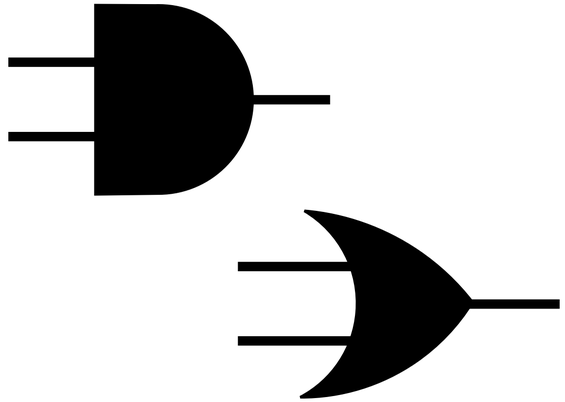


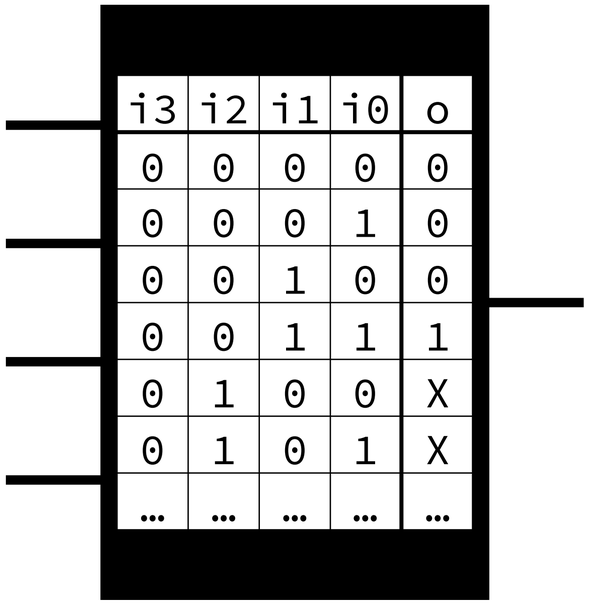
i3	i2	i1	i0	o
0	0	0	0	0
0	0	0	1	0



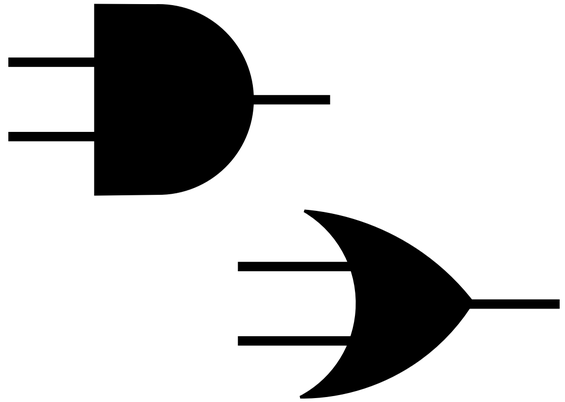


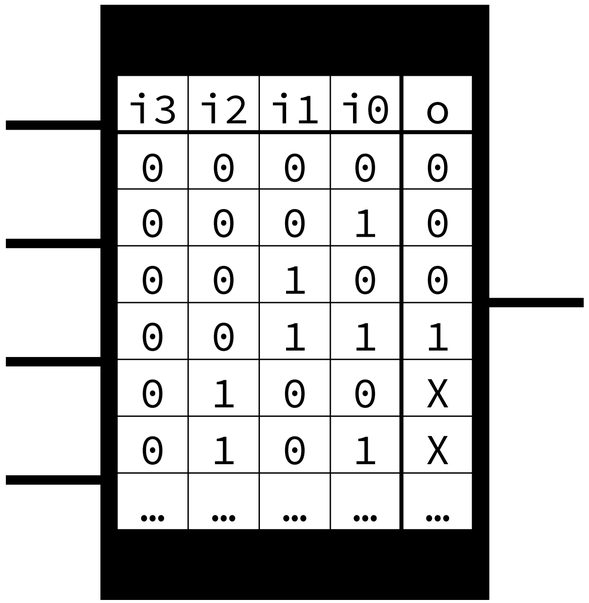
i3	i2	i1	i0	o
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0



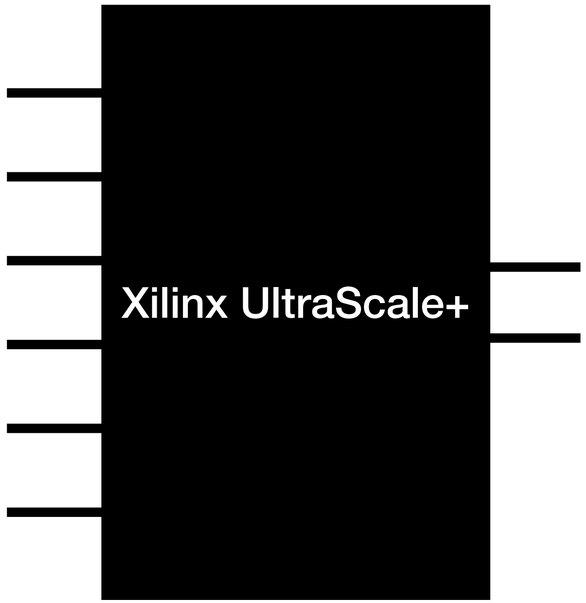


i3	i2	i1	i0	o
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	X
0	1	0	1	X
…	…	…	…	…





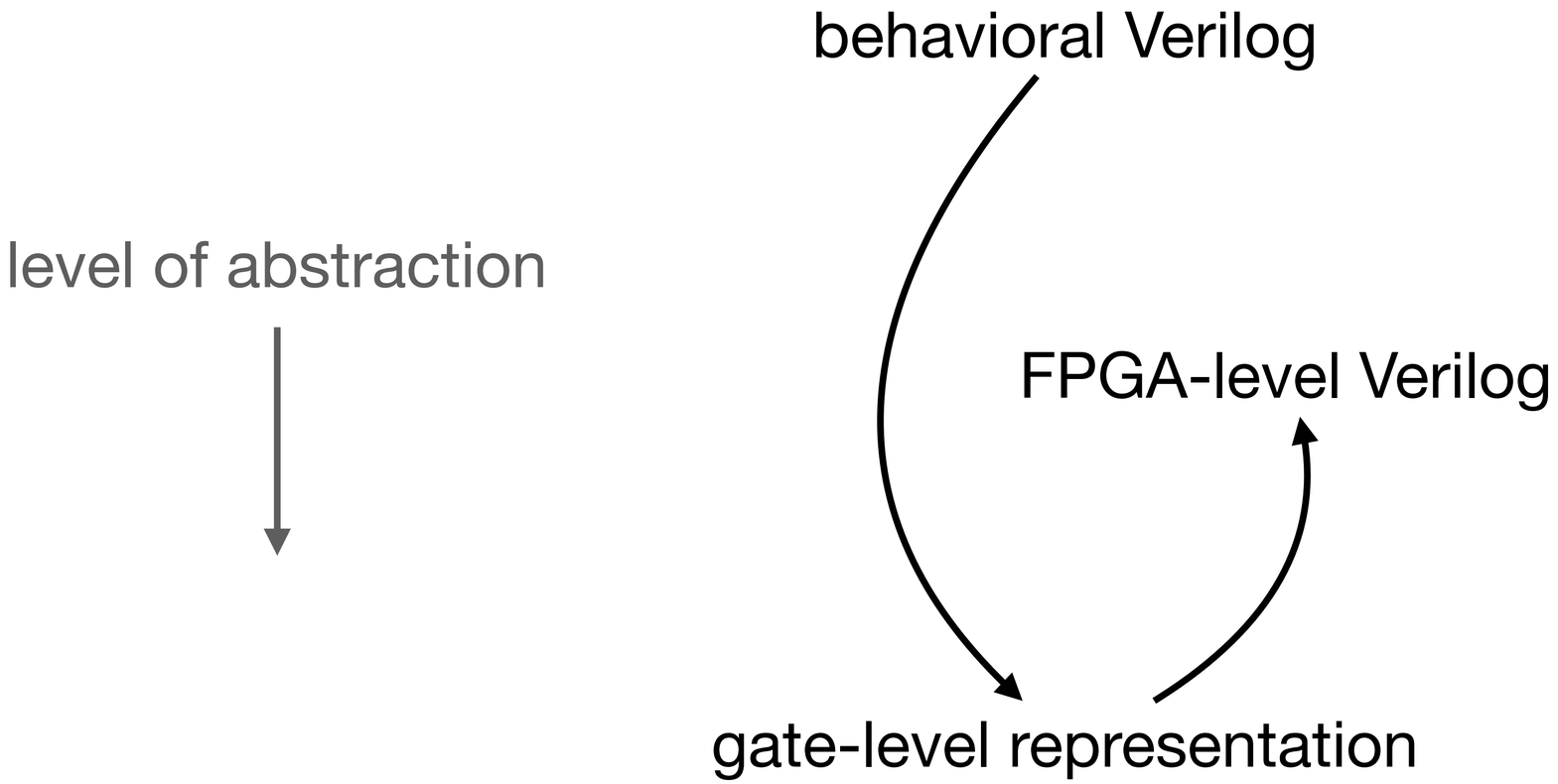
i3	i2	i1	i0	o
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	X
0	1	0	1	X
…	…	…	…	…



Current FPGA compilers are slow and unpredictable.

“UltraScale+ devices employ DSP blocks that are rated at 891MHz for the fastest speed grade. Nonetheless, large designs implemented on FPGAs typically achieve system frequencies lower than 400MHz.”

Lavin, Chris, and Alireza Kaviani. "RapidWright: Enabling custom crafted implementations for FPGAs." *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018.


level of abstraction
behavioral Verilog
FPGA-level Verilog
gate-level representation

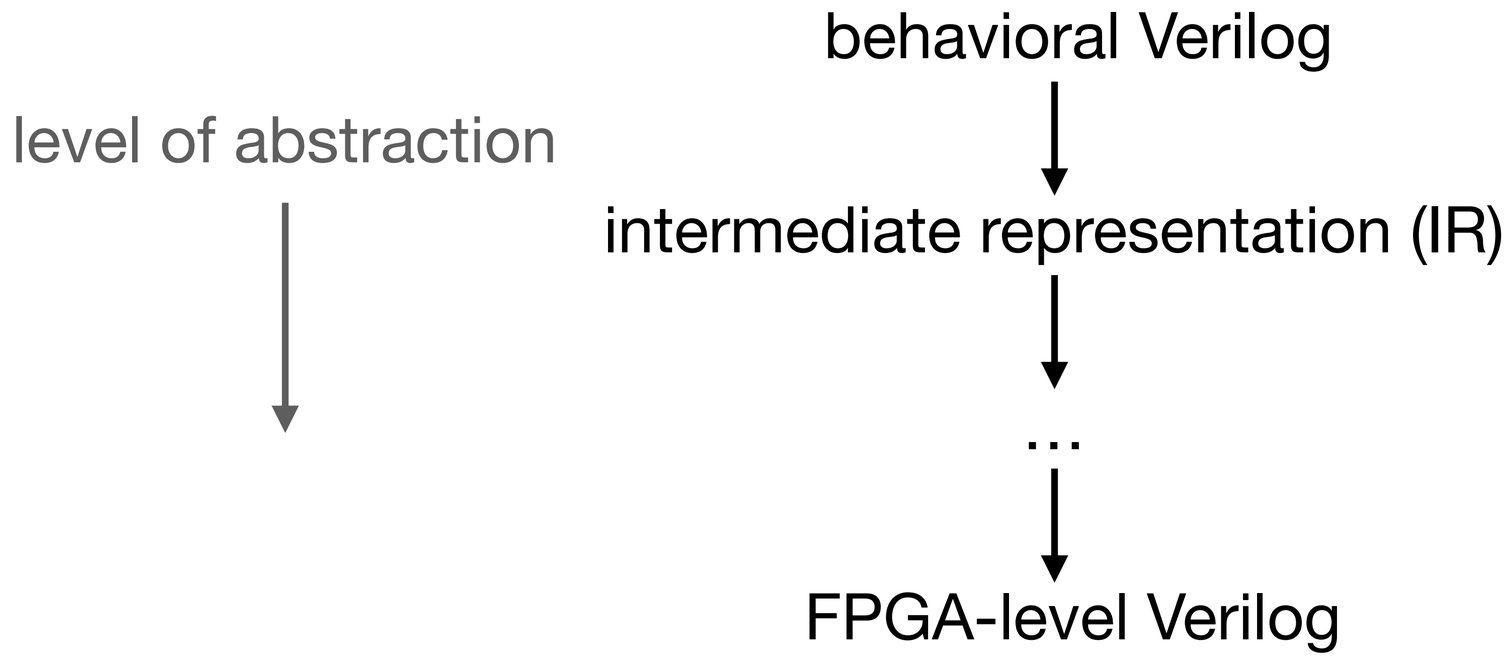
Recent works (Reticle!) have attempted a more direct, software-compiler-like approach.

behavioral Verilog

level of abstraction

Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. "Reticle: a virtual machine for programming modern FPGAs." PLDI 2021.
Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth et al.
"MLIR as Hardware Compiler Infrastructure." WOSET 2021.
Callahan, Timothy J., Philip Chong, Andre DeHon, and John Wawrzynek. “Fast module mapping and placement for datapaths in FPGAs." FPGA 1998.

Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. "Reticle: a virtual machine for programming modern FPGAs." PLDI 2021.
Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth et al.
"MLIR as Hardware Compiler Infrastructure." WOSET 2021.
Callahan, Timothy J., Philip Chong, Andre DeHon, and John Wawrzynek. “Fast module mapping and placement for datapaths in FPGAs." FPGA 1998.



Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. "Reticle: a virtual machine for programming modern FPGAs." PLDI 2021.
Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth et al.
"MLIR as Hardware Compiler Infrastructure." WOSET 2021.
Callahan, Timothy J., Philip Chong, Andre DeHon, and John Wawrzynek. “Fast module mapping and placement for datapaths in FPGAs." FPGA 1998.

To implement this, we need an FPGA “ISA”: the lowest-level IR which gets converted to FPGA-ready Verilog.


level of abstraction
behavioral Verilog
...
FPGA ISA
FPGA-level Verilog

New FPGA compiler toolchains specify their ISAs explicitly!

'comb' Dialect

Types and operations for comb dialect This dialect defines the `comb` dialect, which is intended to be a generic representation of combinational logic outside of a particular use-case.

100 lines (93 sloc) | 2.87 KB Raw Blame

```
extern "core.sv" {
  /// Primitives
  primitive std_const<"share"=1>[WIDTH, VALUE]() -> (out: WIDTH);
  primitive std_slice<"share"=1>[IN_WIDTH, OUT_WIDTH](in: IN_WIDTH) -> (out: OUT_WIDTH);
  primitive std_pad<"share"=1>[IN_WIDTH, OUT_WIDTH](in: IN_WIDTH) -> (out: OUT_WIDTH);

  /// Logical operators
  primitive std_not<"share"=1>[WIDTH](in: WIDTH) -> (out: WIDTH);
  primitive std_and<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: WIDTH);
  primitive std_or<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: WIDTH);
  primitive std_xor<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: WIDTH);

  /// Numerical Operators
  primitive std_add<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: WIDTH);
  primitive std_sub<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: WIDTH);
  primitive std_gt<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: 1);
  primitive std_lt<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: 1);
  primitive std_eq<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: 1);
  primitive std_neq<"share"=1>[WIDTH](left: WIDTH, right: WIDTH) -> (out: 1);
```

Reticle compiles designs to these ISA instructions, and then those instructions get converted to FPGA-specific Verilog.

But how do we choose the ISA?

But how do we choose the ISA?

And how do we implement it?

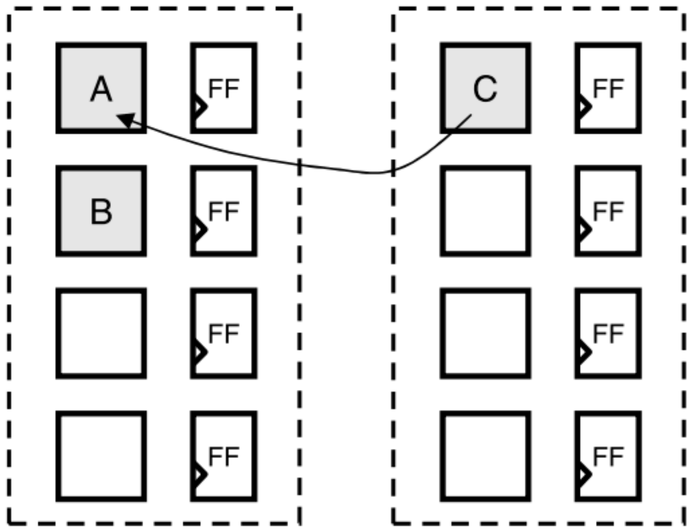
But how do we choose the ISA?

And how do we implement it?

Currently: by hand!

Choosing ISAs by hand may leave gaps in the ISA.

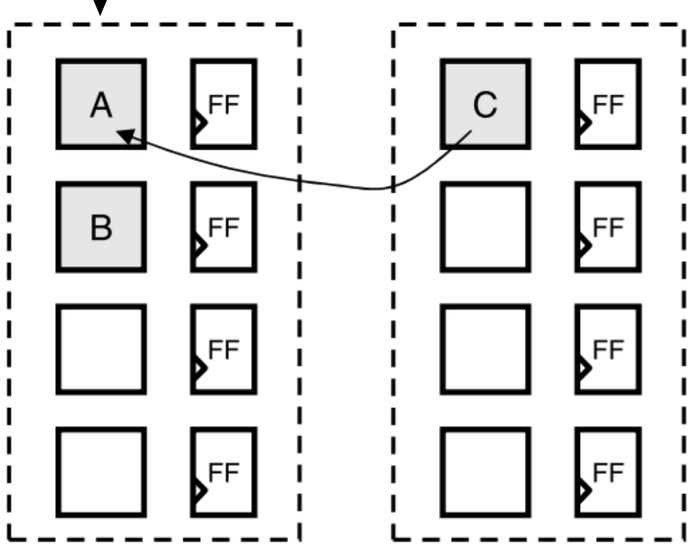
A key optimization for FPGAs: *packing* or *fusing* LUTs!



Taneem Ahmed, Paul D. Kundarewich, and Jason H. Anderson. "Packing techniques for virtex-5 FPGAs." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 2.3 (2009): 1-24.

A key optimization for FPGAs: *packing* or *fusing* LUTs!

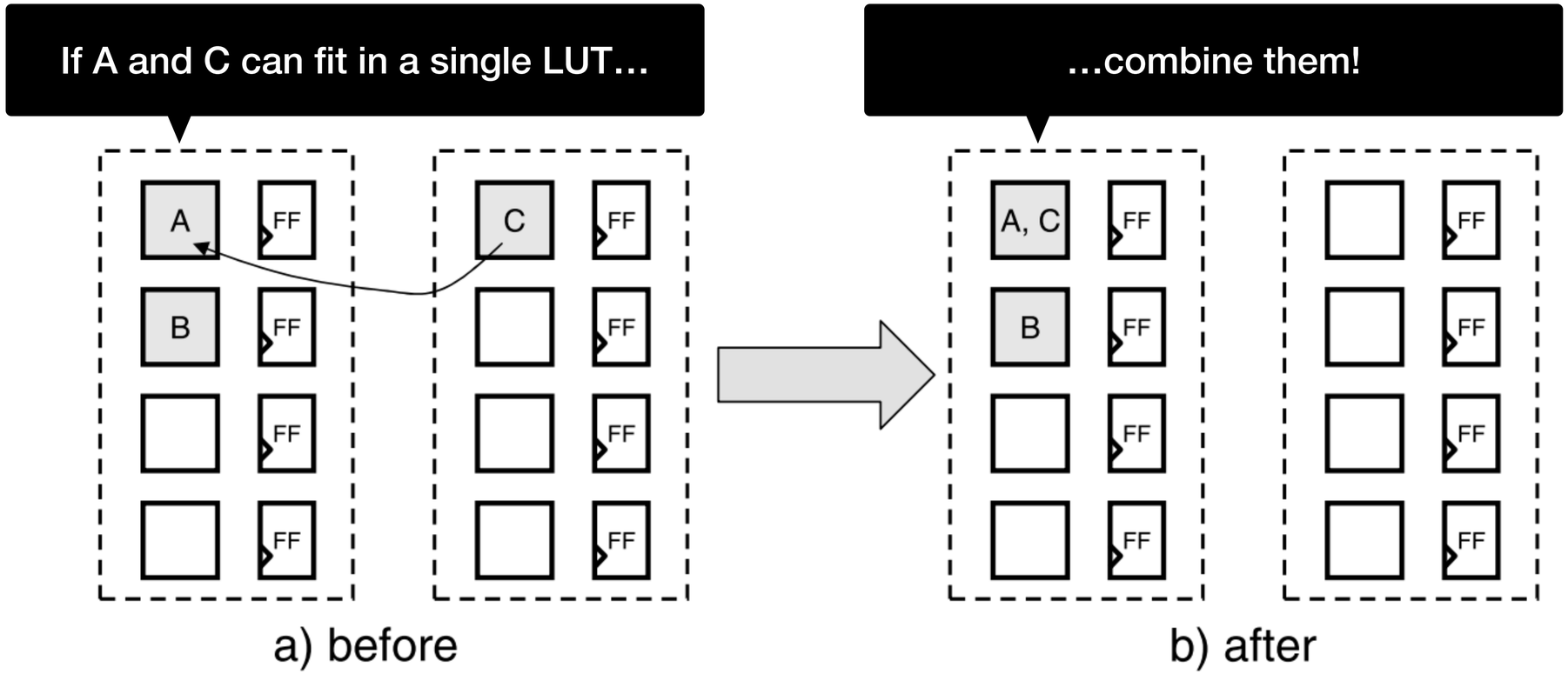
If A and C can fit in a single LUT…



a) before

Taneem Ahmed, Paul D. Kundarewich, and Jason H. Anderson. "Packing techniques for virtex-5 FPGAs." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 2.3 (2009): 1-24.

A key optimization for FPGAs: *packing* or *fusing* LUTs!

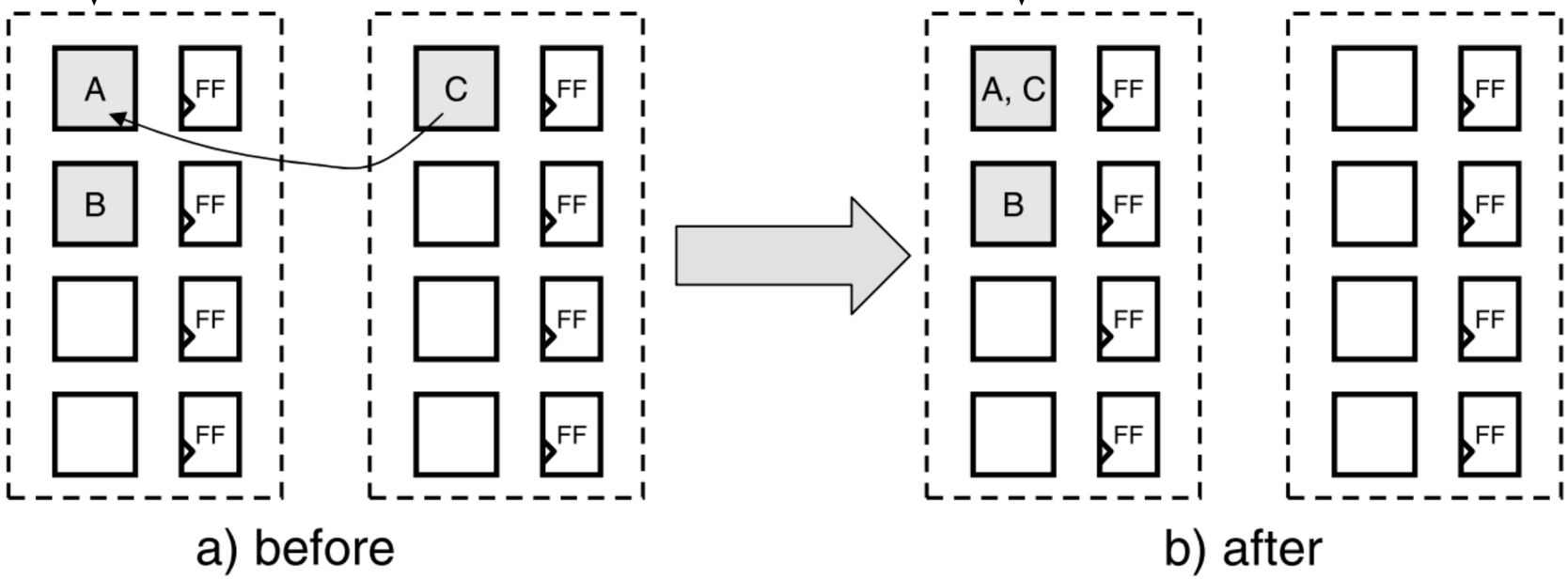


Taneem Ahmed, Paul D. Kundarewich, and Jason H. Anderson. "Packing techniques for virtex-5 FPGAs."
ACM Transactions on Reconfigurable Technology and Systems (TRETS) 2.3 (2009): 1-24.

A key optimization for FPGAs: *packing* or *fusing* LUTs!

If A and C can fit in a single LUT…

…combine them!



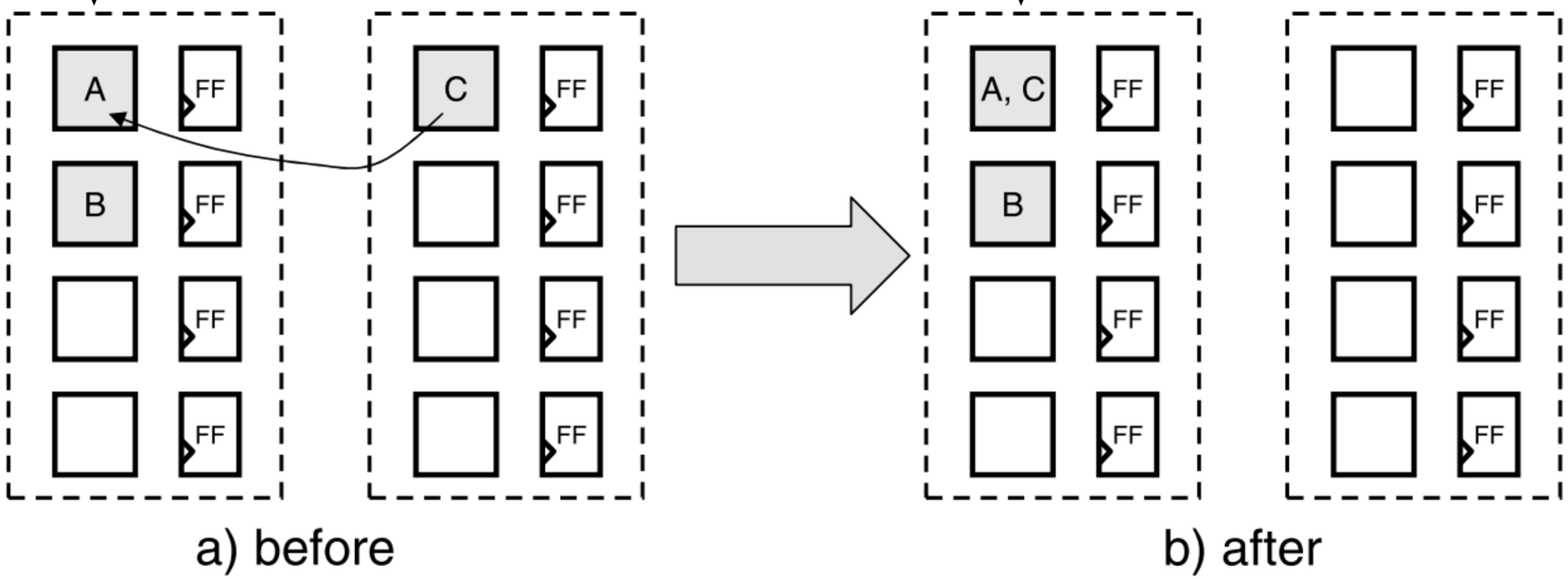
This requires us to have A,C in our ISA.

Taneem Ahmed, Paul D. Kundarewich, and Jason H. Anderson. "Packing techniques for virtex-5 FPGAs."
ACM Transactions on Reconfigurable Technology and Systems (TRETS) 2.3 (2009): 1-24.

A key optimization for FPGAs: *packing* or *fusing* LUTs!

If A and C can fit in a single LUT…

…combine them!



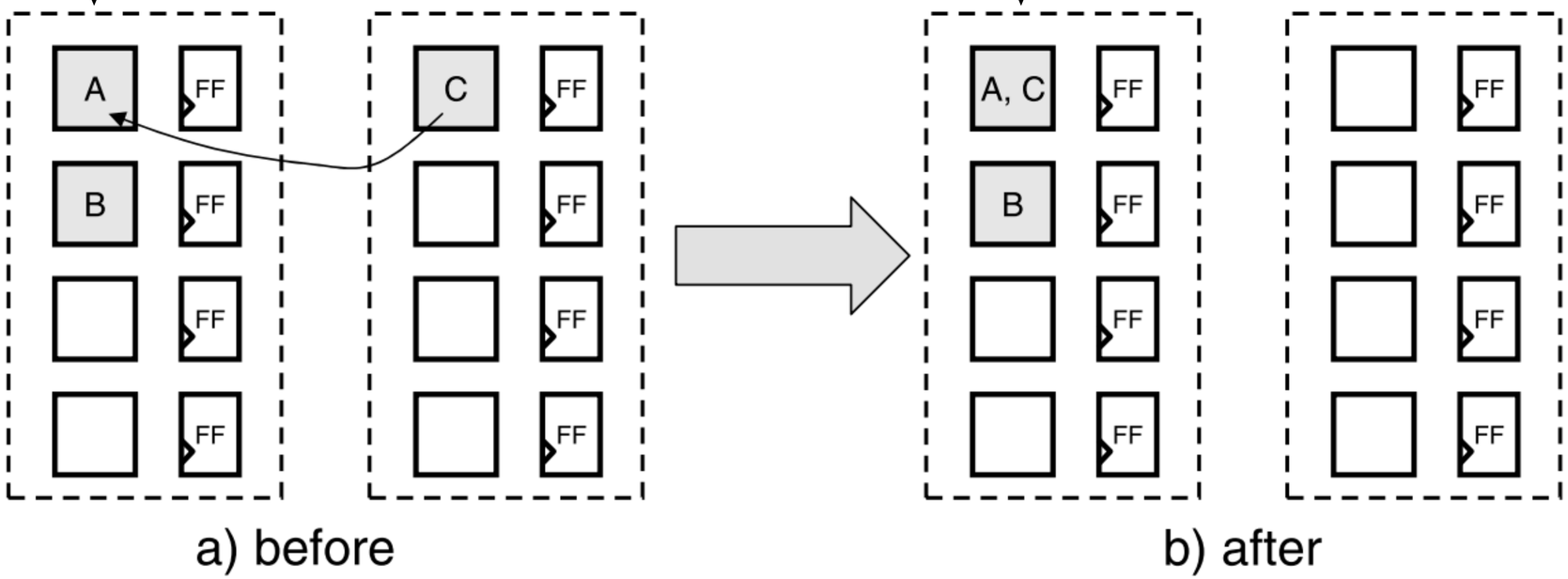
This requires us to have A,C in our ISA.
Do we also need A,B? Or B,C?

Taneem Ahmed, Paul D. Kundarewich, and Jason H. Anderson. "Packing techniques for virtex-5 FPGAs."
ACM Transactions on Reconfigurable Technology and Systems (TRETS) 2.3 (2009): 1-24.

A key optimization for FPGAs: *packing* or *fusing* LUTs!

If A and C can fit in a single LUT…

…combine them!



This requires us to have A,C in our ISA.
Do we also need A,B? Or B,C?
Think of all the possible combinations we will have to consider!

Taneem Ahmed, Paul D. Kundarewich, and Jason H. Anderson. "Packing techniques for virtex-5 FPGAs."
ACM Transactions on Reconfigurable Technology and Systems (TRETS) 2.3 (2009): 1-24.

Choosing ISA by hand will miss many fused instructions.

What about *implementing* ISAs?

Implementing ISAs by hand is infeasible for large ISAs—and a great source of bugs!

8-bit add ISA instruction:

```
pat add_i8(a:i8, b:i8) -> (y:i8) {
    y:i8 = add(a, b) @lut;
}
```

8-bit add ISA instruction:

```
pat add_i8(a:i8, b:i8) -> (y:i8) {
    y:i8 = add(a, b) @lut;
}
```

Xilinx 7-series implementation of 8-bit add:

```
imp add_i8[1, 2](a:i8, b:i8) -> (y:i8) {
    t0:bool = ext[0](a);
    t1:bool = ext[1](a);
    t2:bool = ext[2](a);
    t3:bool = ext[3](a);
    t4:bool = ext[4](a);
    t5:bool = ext[5](a);
    t6:bool = ext[6](a);
    t7:bool = ext[7](a);
    t8:bool = ext[0](b);
    t9:bool = ext[1](b);
    t10:bool = ext[2](b);
    t11:bool = ext[3](b);
    t12:bool = ext[4](b);
    t13:bool = ext[5](b);
    t14:bool = ext[6](b);
    t15:bool = ext[7](b);
    t16:bool = lut2[6](t0, t8) @a6(??, ??);
    t17:bool = lut2[6](t1, t9) @b6(??, ??);
    t18:bool = lut2[6](t2, t10) @c6(??, ??);
    t19:bool = lut2[6](t3, t11) @d6(??, ??);
    t20:bool = lut2[6](t4, t12) @e6(??, ??);
    t21:bool = lut2[6](t5, t13) @f6(??, ??);
    t22:bool = lut2[6](t6, t14) @g6(??, ??);
    t23:bool = lut2[6](t7, t15) @h6(??, ??);
    t24:i8 = cat(t16, t17, t18, t19, t20, t21, t22, t23);
    y:i8 = carryadd(a, t24) @c8(??, ??);
}
```

8-bit add ISA instruction:

```
pat add_i8(a:i8, b:i8) -> (y:i8) {
    y:i8 = add(a, b) @lut;
}
```

Xilinx 7-series implementation of 8-bit add:

```
imp add_i8[1, 2](a:i8, b:i8) -> (y:i8) {
    t0:bool = ext[0](a);
    t1:bool = ext[1](a);
    t2:bool = ext[2](a);
    t3:bool = ext[3](a);
    t4:bool = ext[4](a);
    t5:bool = ext[5](a);
    t6:bool = ext[6](a);
    t7:bool = ext[7](a);
    t8:bool = ext[0](b);
    t9:bool = ext[1](b);
    t10:bool = ext[2](b);
    t11:bool = ext[3](b);
    t12:bool = ext[4](b);
    t13:bool = ext[5](b);
    t14:bool = ext[6](b);
    t15:bool = ext[7](b);
    t16:bool = lut2[6](t0, t8) @a6(??, ??);
    t17:bool = lut2[6](t1, t9) @b6(??, ??);
    t18:bool = lut2[6](t2, t10) @c6(??, ??);
    t19:bool = lut2[6](t3, t11) @d6(??, ??);
    t20:bool = lut2[6](t4, t12) @e6(??, ??);
    t21:bool = lut2[6](t5, t13) @f6(??, ??);
    t22:bool = lut2[6](t6, t14) @g6(??, ??);
    t23:bool = lut2[6](t7, t15) @h6(??, ??);
    t24:i8 = cat(t16, t17, t18, t19, t20, t21, t22, t23);
    y:i8 = carryadd(a, t24) @c8(??, ??);
}
```

Luis wrote all of these implementations by hand for Reticle!

This is slow, especially if we have multiple backends and many fused instructions!

So, can we do it automatically?

We introduce Lakeroad, a tool for automatically *defining* and *implementing* ISAs for FPGAs.

ISA Exploration

ISA Implementation Synthesis

ISA Exploration

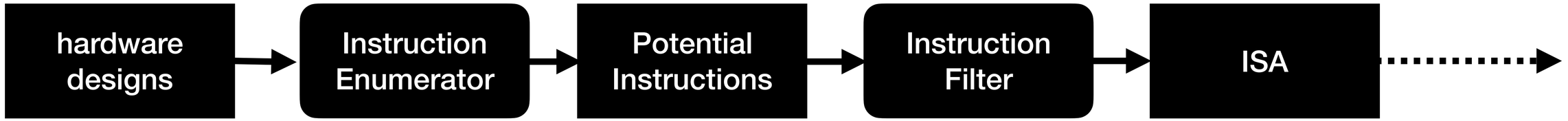
ISA Implementation Synthesis

Core idea of ISA exploration: define the ISA from instructions found in real designs.

ISA Exploration

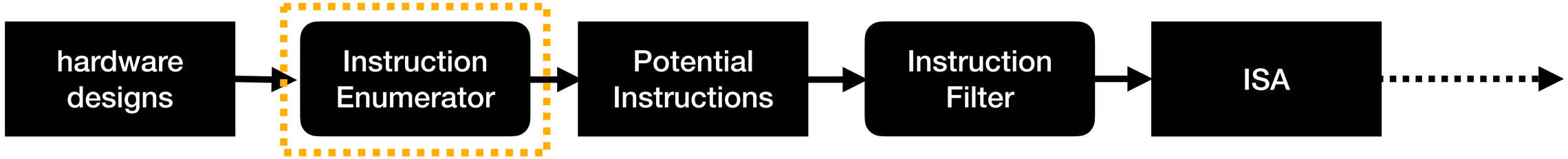
ISA Implementation Synthesis

ISA Exploration



ISA Implementation Synthesis

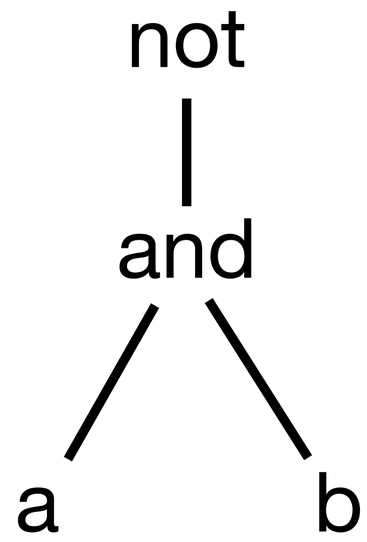
ISA Exploration



ISA Implementation Synthesis

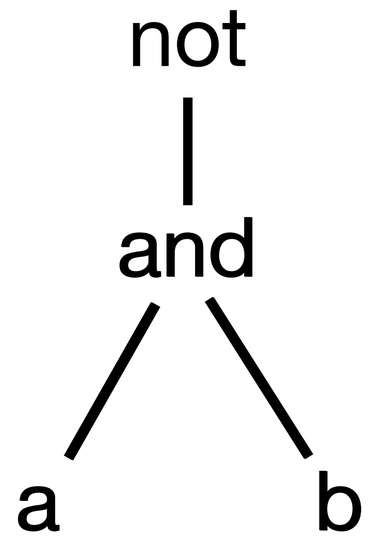
A simple example: enumerate the instructions present in not (a and b)

not (a and b)

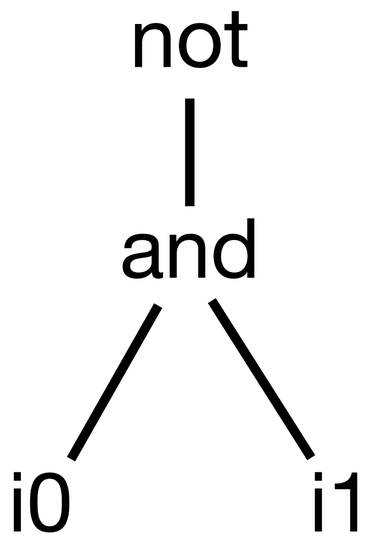


Naive approach: instructions are just the subexpressions!

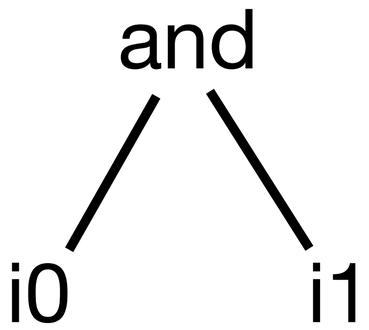
not (a and b)



not (i0 and i1)

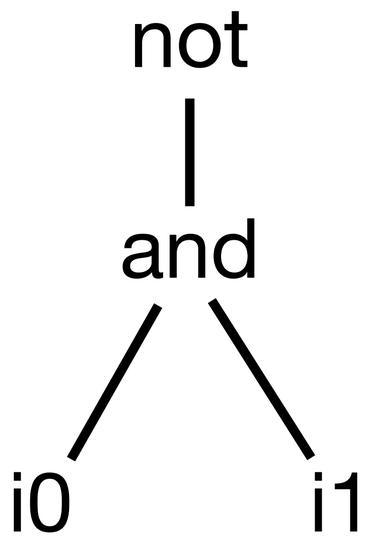


i0 and i1

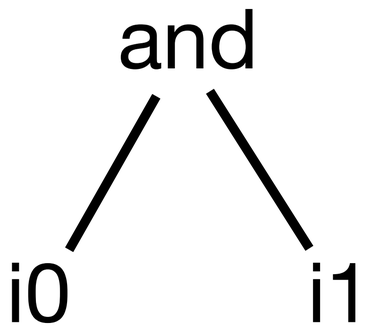


But we missed one!

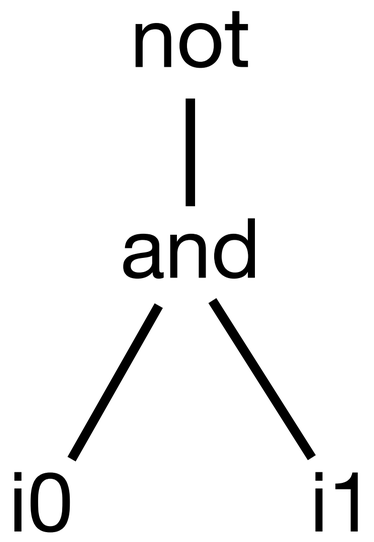
not (i0 and i1)



i0 and i1

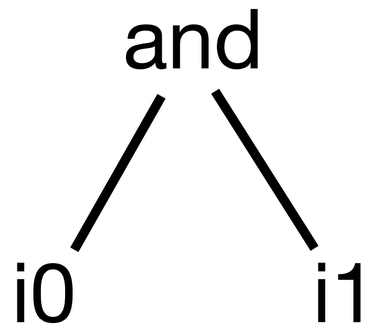


not (i0 and i1)



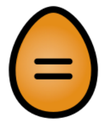
not i0

i0 and i1

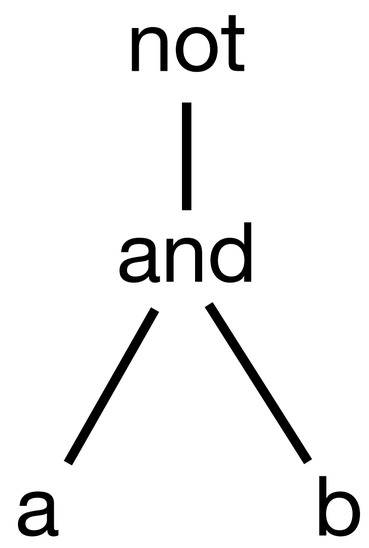


So how do we capture *all* instructions?

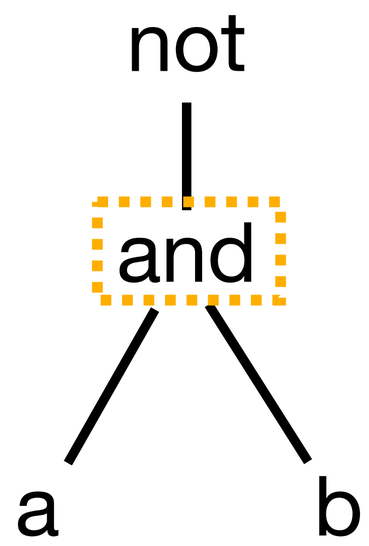
So how do we capture *all* instructions?

With rewrites! 

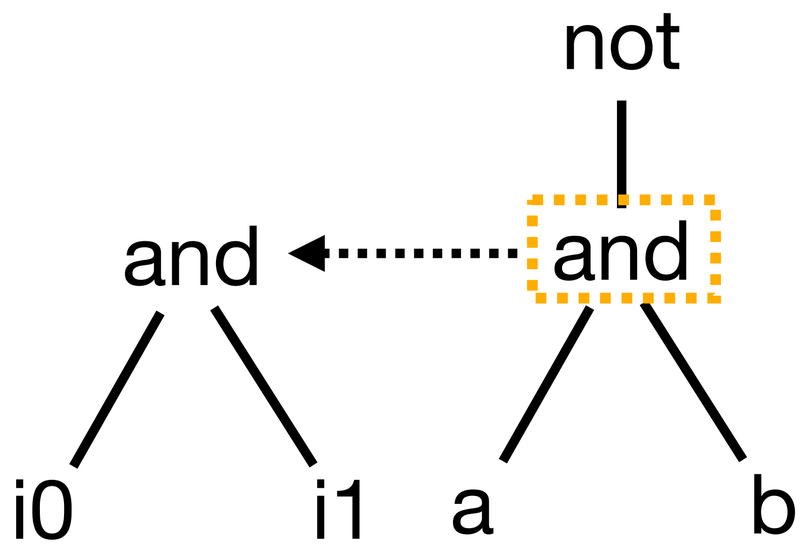
To convert a node into an instruction,
decide which of its children to convert to arguments.



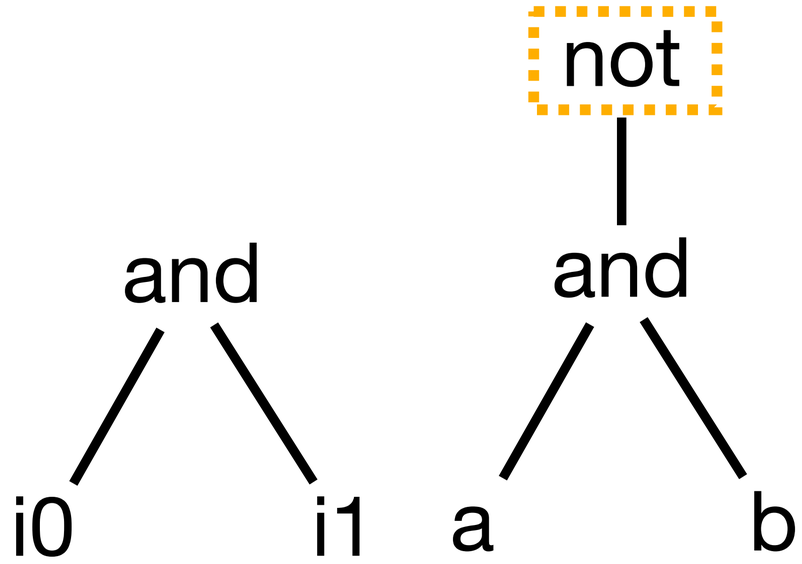
To convert a node into an instruction,
decide which of its children to convert to arguments.



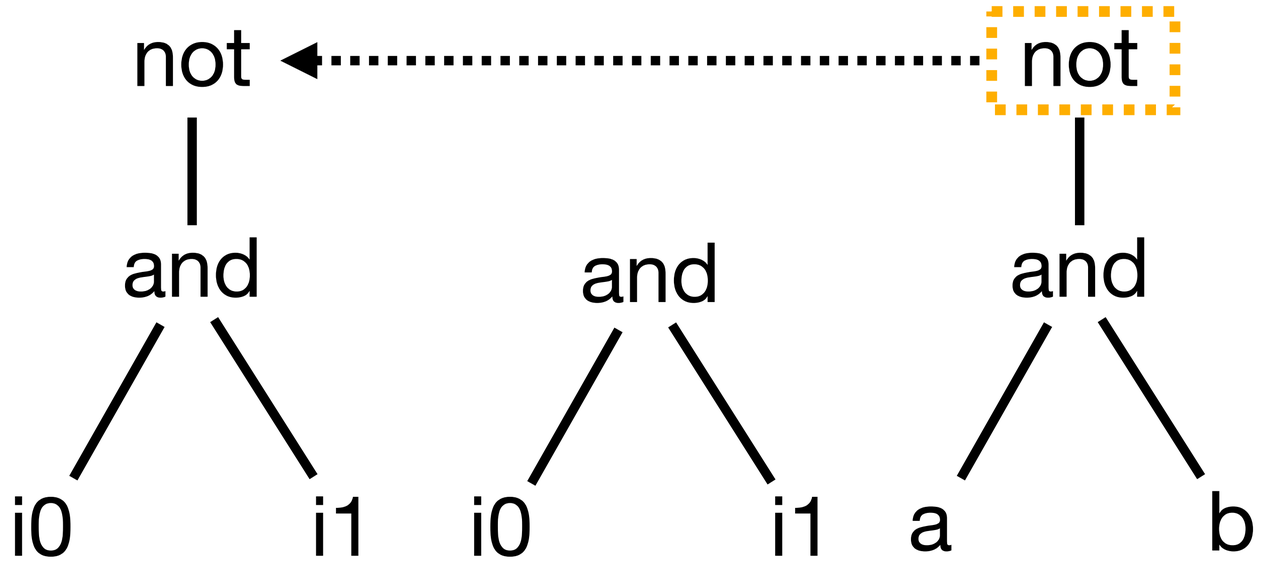
To convert a node into an instruction,
decide which of its children to convert to arguments.



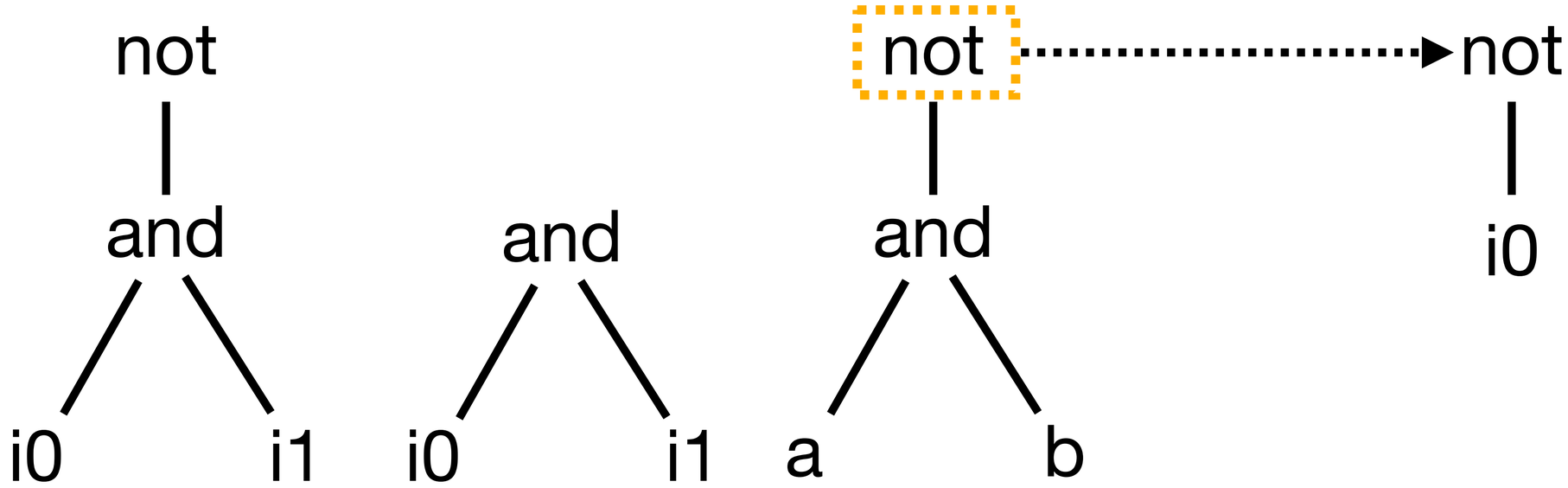
To convert a node into an instruction,
decide which of its children to convert to arguments.



To convert a node into an instruction,
decide which of its children to convert to arguments.

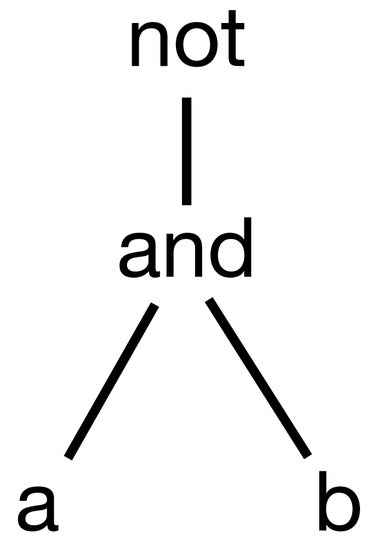


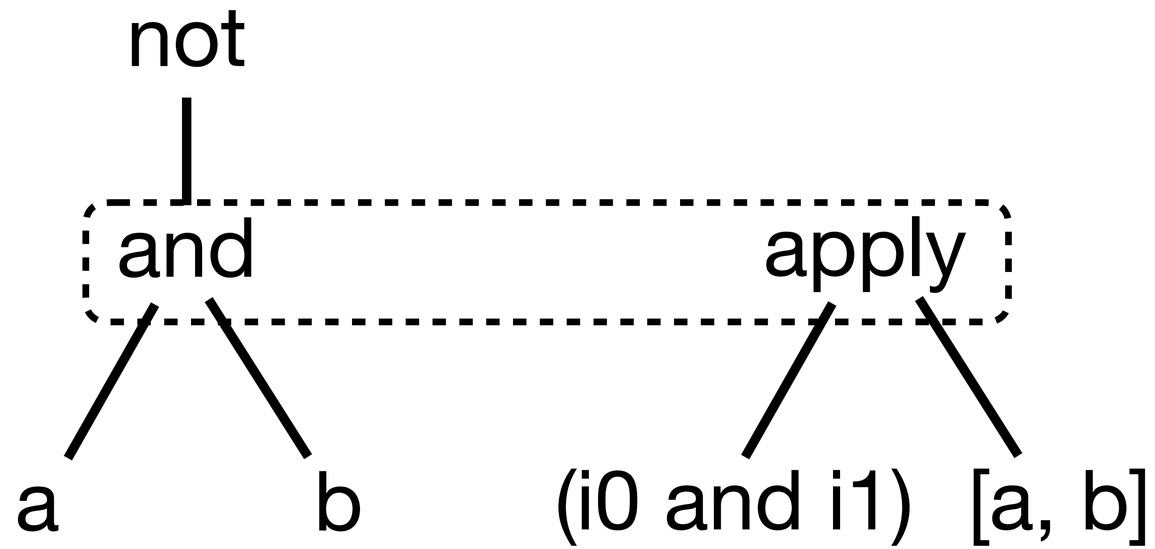
To convert a node into an instruction,
decide which of its children to convert to arguments.

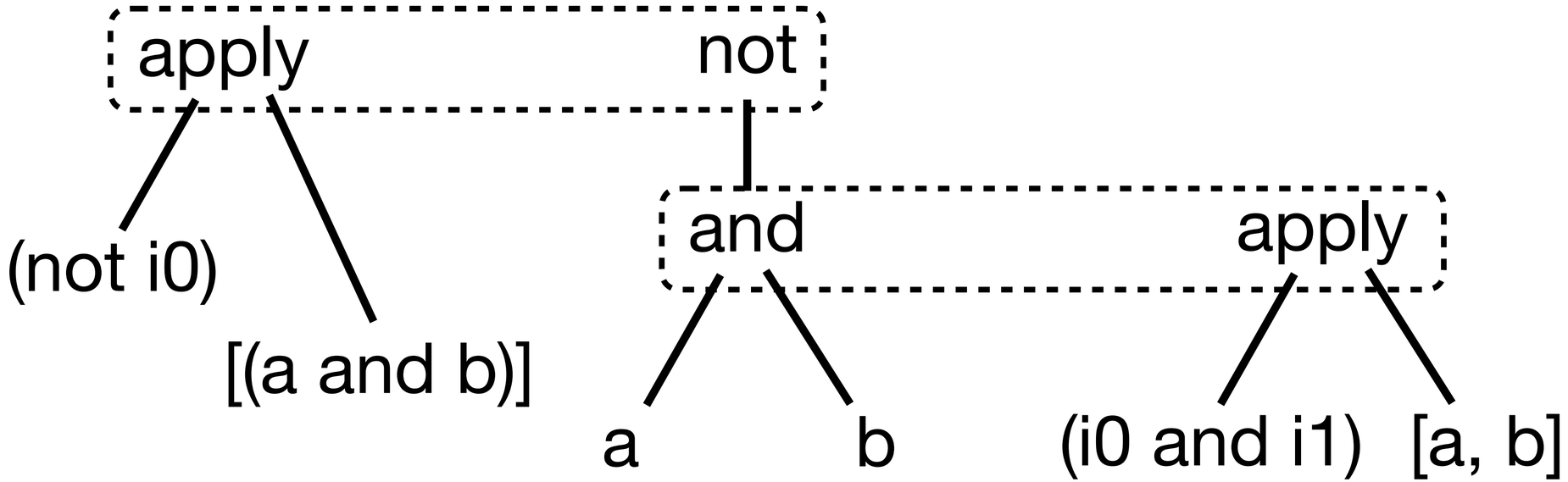


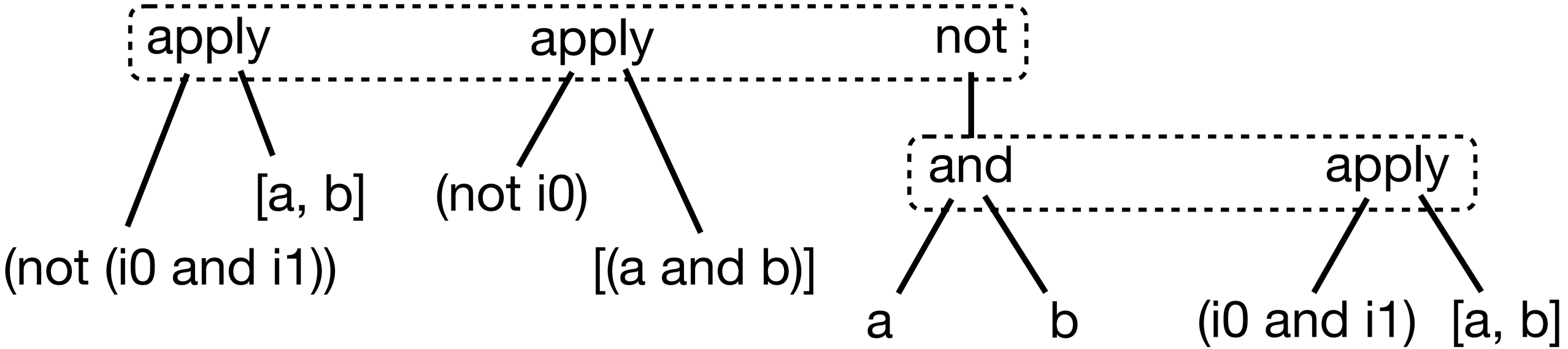
```
(binop ?op ?bw (apply (instr ?ast0 ?canonical-args0) ?args0) (apply (instr ?ast1 ?canonical-args1) ?args1))
 => (apply (instr (binop-ast ?op ?bw ?ast0 ?ast1) (canonicalize (concat ?args0 ?args1)))
           (concat ?args0 ?args1))

(binop ?op ?bw ?left (apply (instr ?ast1 ?canonical-args1) ?args1))
 => (apply (instr (binop-ast ?op ?bw (hole ?bw) ?ast1) (canonicalize (concat (list ?left) ?args1)))
           (concat (list

(binop ?op ?bw (apply (i
 => (apply (instr (binop-ast ?op ?bw ?ast0 (hole ?bw)) (canonicalize (concat ?args0 (list ?right))))
           (concat ?args0 (list ?right)))

(binop ?op ?bw ?a ?b)
 => (apply (instr (binop-ast ?op ?bw (hole ?bw) (hole ?bw)) (canonicalize (list ?a ?b)))
           (list ?a ?b))
```

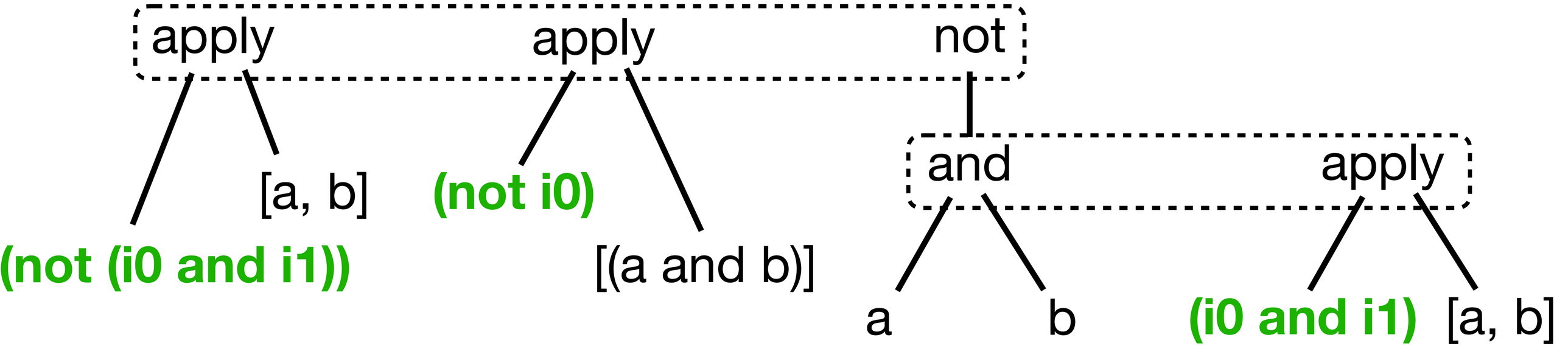
This can be encoded as a small set of rewrites in egg!


not
and
a
b


not
and
apply
a
b
(i0 and i1)
[a, b]


apply
not
(not i0)
[(a and b)]
and
apply
a
b
(i0 and i1)
[a, b]

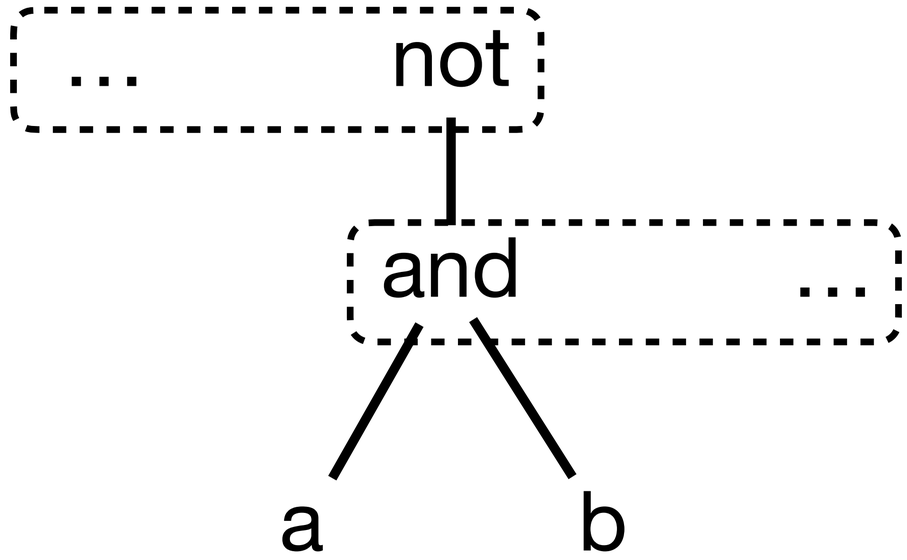

apply
apply
not
[a, b]
(not i0)
and
apply
(not (i0 and i1))
[(a and b)]
a
b
(i0 and i1)
[a, b]

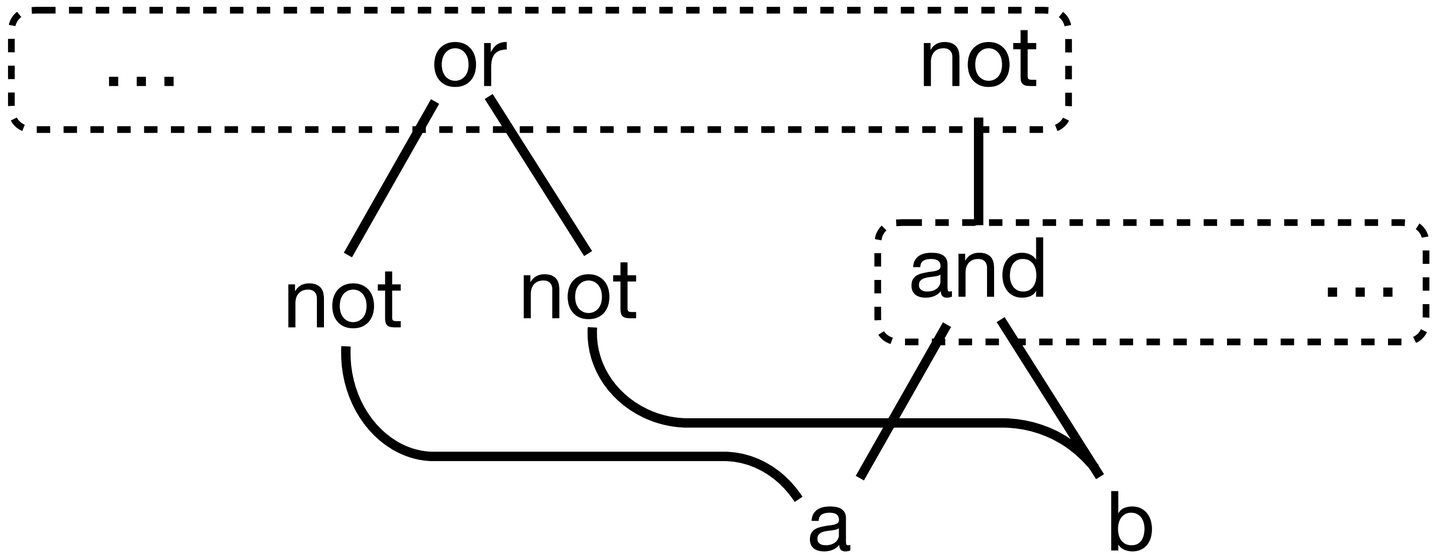

apply
apply
not
[a, b]
(not i0)
(not (i0 and i1))
[(a and b)]
and
apply
a
b
(i0 and i1)
[a, b]

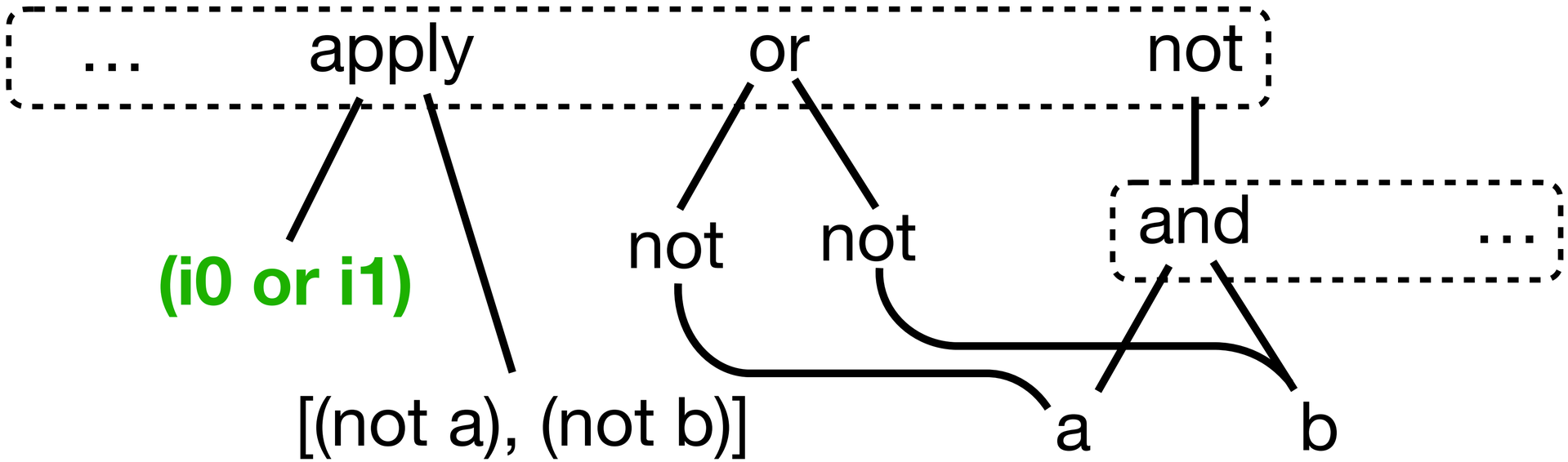
We can even apply other rewrites simultaneously!

De Morgan’s law

```
not (a and b) ==> (not a) or (not b)
```


...
not
and
...
a
b


...
or
not
not
not
and
...
a
b


…
apply
or
not
(i0 or i1)
not
not
and
…
[(not a), (not b)]
a
b

Our potential instructions!

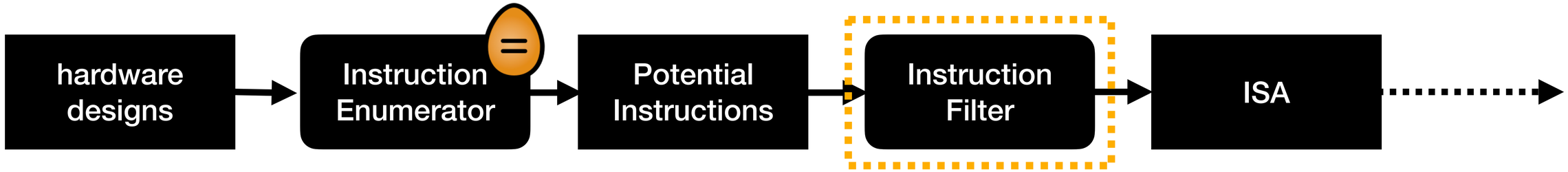
i0 and i1

not (i0 and i1)

not i0

i0 or i1

ISA Exploration



ISA Implementation Synthesis

i0 and i1

not (i0 and i1)

not i0

i0 or i1

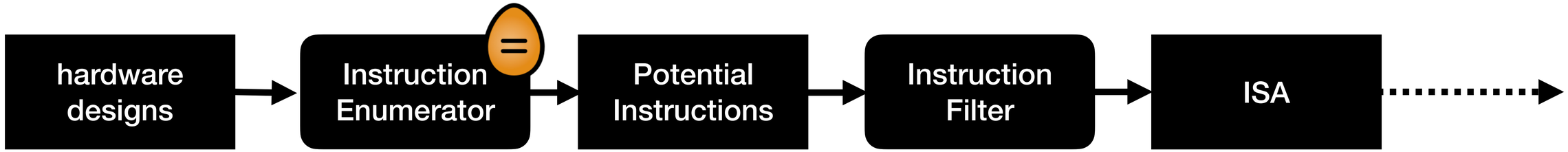
i0 and i1

~~not (i0 and i1)~~ Too specialized—filter it out!

not i0

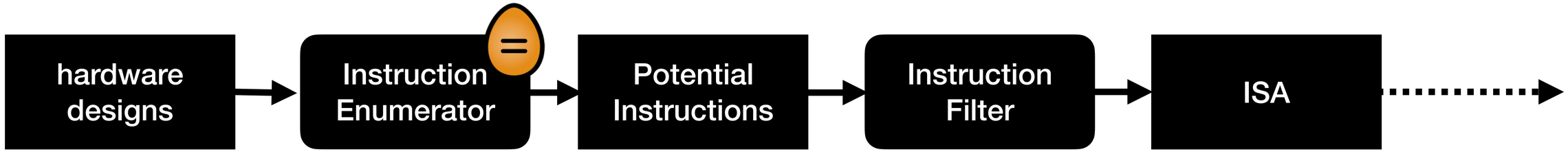
i0 or i1

ISA Exploration

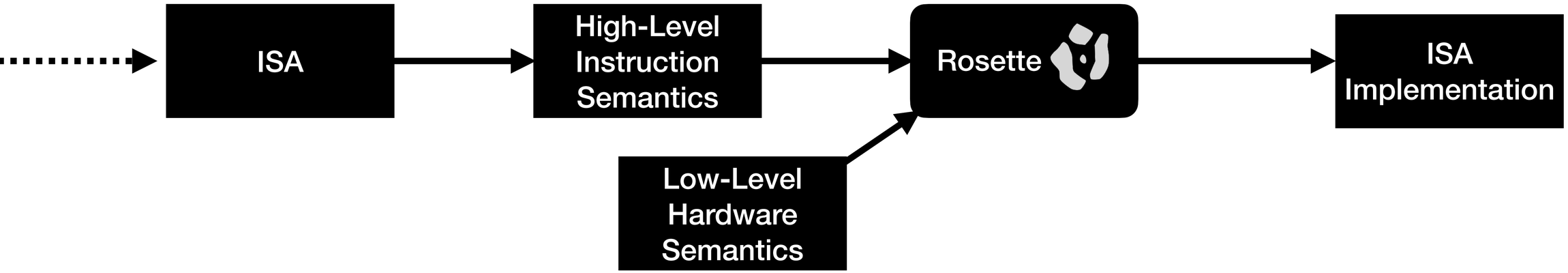


ISA Implementation Synthesis

ISA Exploration



ISA Implementation Synthesis



Rosette is a synthesis tool which allows us to ask:

Emina Torlak, and Rastislav Bodik. "Growing solver-aided languages with Rosette." In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software, pp. 135-152. 2013.

Rosette is a synthesis tool which allows us to ask:

For all inputs a and b,

Emina Torlak, and Rastislav Bodik. "Growing solver-aided languages with Rosette." In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software, pp. 135-152. 2013.

Rosette is a synthesis tool which allows us to ask:

For all inputs `a` and `b`,
find an implementation of `low-level-FPGA-impl` such that

Emina Torlak, and Rastislav Bodik. "Growing solver-aided languages with Rosette." In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software, pp. 135-152. 2013.

Rosette is a synthesis tool which allows us to ask:

For all inputs `a` and `b`,
find an implementation of `low-level-FPGA-impl` such that

```
low-level-FPGA-impl(a, b) == high-level-instr-impl(a, b)
```

Emina Torlak, and Rastislav Bodik. "Growing solver-aided languages with Rosette." In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software, pp. 135-152. 2013.

Rosette is a synthesis tool which allows us to ask:

For all inputs `a` and `b`,
find an implementation of `low-level-FPGA-impl` such that

```
low-level-FPGA-impl(a, b) == high-level-instr-impl(a, b)
```

To do so, we need to define the **high-level semantics** of the instruction,

Emina Torlak, and Rastislav Bodik. "Growing solver-aided languages with Rosette." In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software, pp. 135-152. 2013.

Rosette is a synthesis tool which allows us to ask:

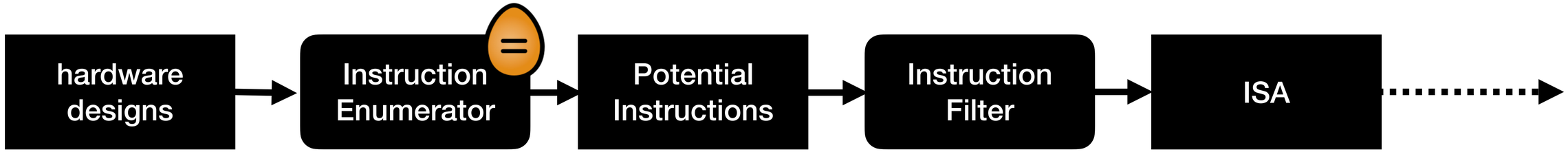
For all inputs `a` and `b`,
find an implementation of `low-level-FPGA-impl` such that

```
low-level-FPGA-impl(a, b) == high-level-instr-impl(a, b)
```

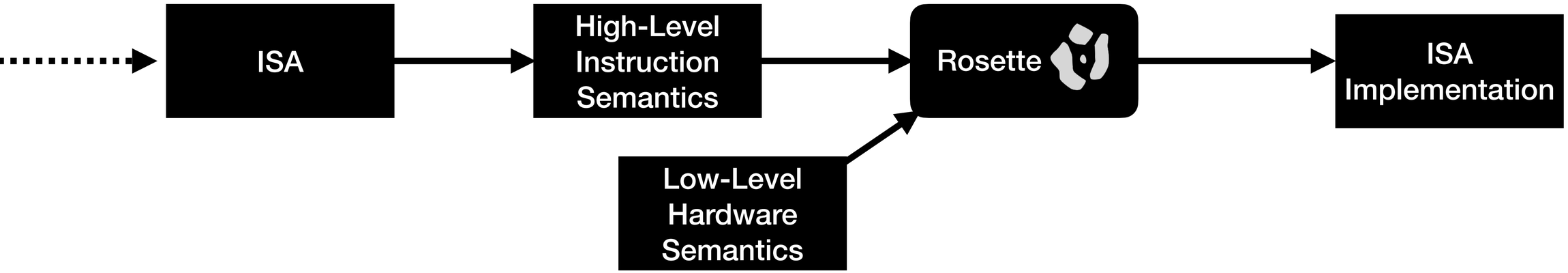
To do so, we need to define the **high-level semantics** of the instruction,
and the **low-level semantics** of the FPGA.

Emina Torlak, and Rastislav Bodik. "Growing solver-aided languages with Rosette." In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software, pp. 135-152. 2013.

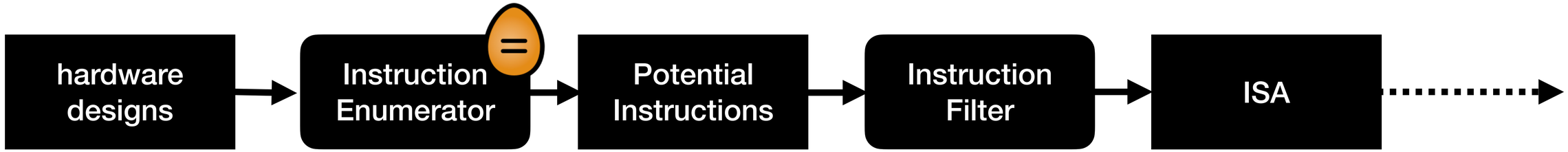
ISA Exploration



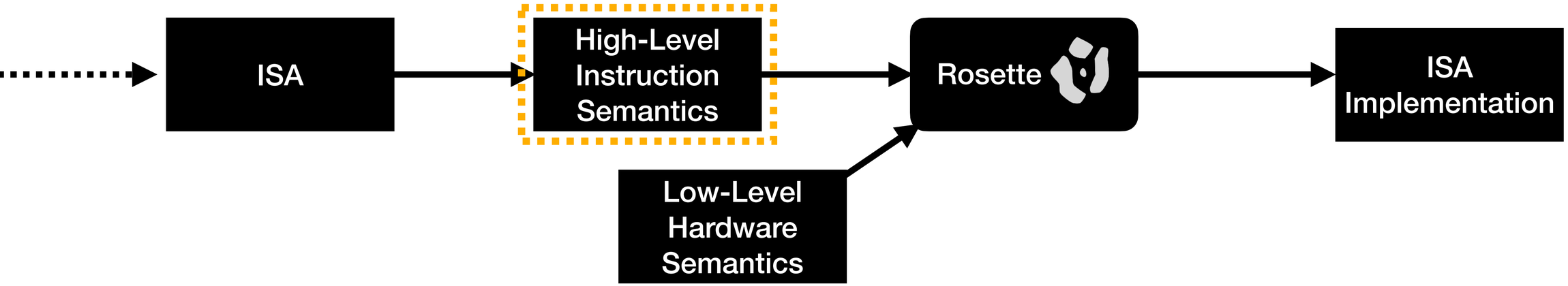
ISA Implementation Synthesis



ISA Exploration



ISA Implementation Synthesis



4.3.2 Bitwise Operators

(bvnot *x***)** → **(bitvector** n**)** procedure

x : **(bitvector** n**)**

Returns the bitwise negation of the given bitvector value.

Examples:

```
> (bvnot (bv -1 4))
(bv #x0 4)
> (bvnot (bv 0 4))
(bv #xf 4)
> (define-symbolic b boolean?)
> (bvnot (if b 0 (bv 0 4))) ; This typechecks only when b is false,
(bv #xf 4)
> (vc)                      ; so Rosette emits a corresponding assertion.
(vc #t (! b))
```

(bvand *x* ...+**)** → **(bitvector** n**)** procedure

x : **(bitvector** n**)**

(bvor *x* ...+**)** → **(bitvector** n**)**

x : **(bitvector** n**)**

(bvxor *x* ...+**)** → **(bitvector** n**)**

i0 and i1

not (i0 and i1)

not i0

i0 or i1

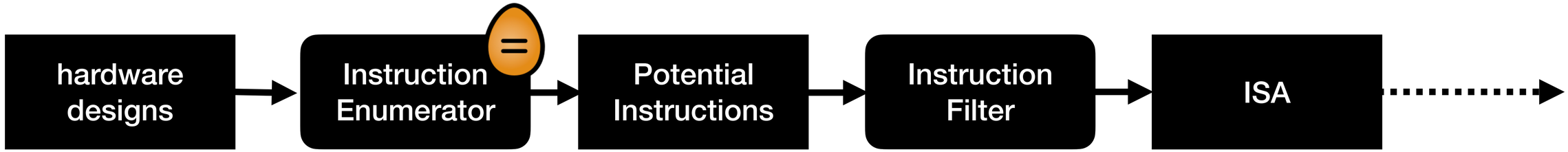
i0 and i1 ⟶ (bvand i0 i1)

not (i0 and i1) ⟶ (bvnot (bvand i0 i1))

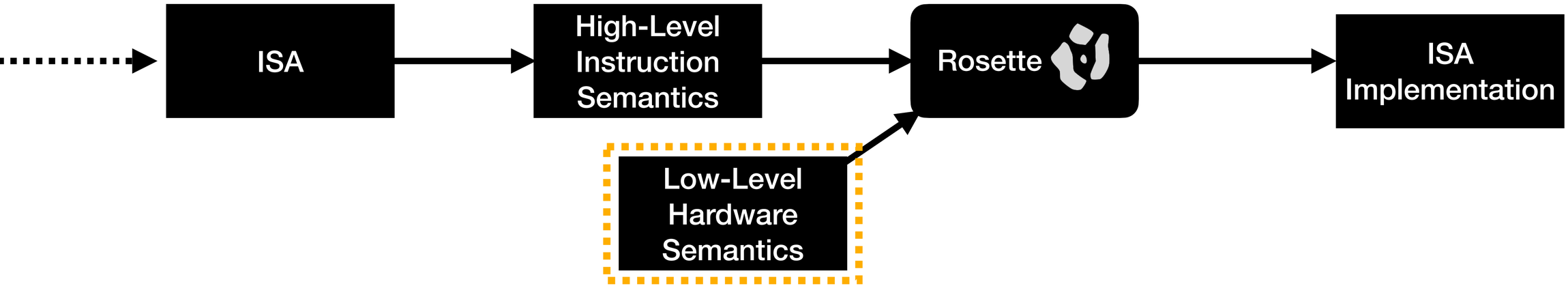
not i0 ⟶ (bvnot i0)

i0 or i1 ⟶ (bvor i0 i1)

ISA Exploration



ISA Implementation Synthesis



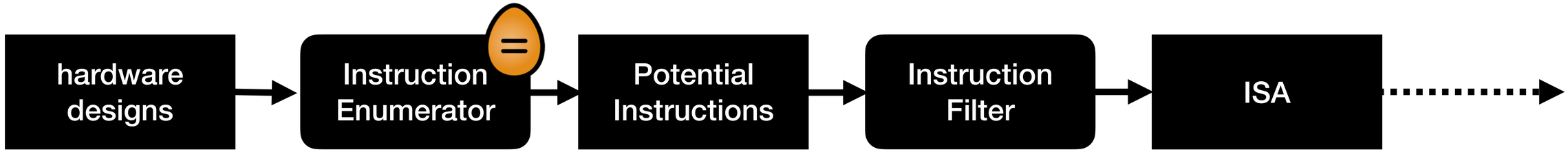
To capture architecture-level semantics of FPGAs, we simply build an *interpreter* for each FPGA component!

```
(define (lut memory inputs)
  (let* ([inputs (zero-extend inputs (bitvector (length (bitvector->bits memory))))])
    (extract 0 0 (bvlshr memory inputs))))
```

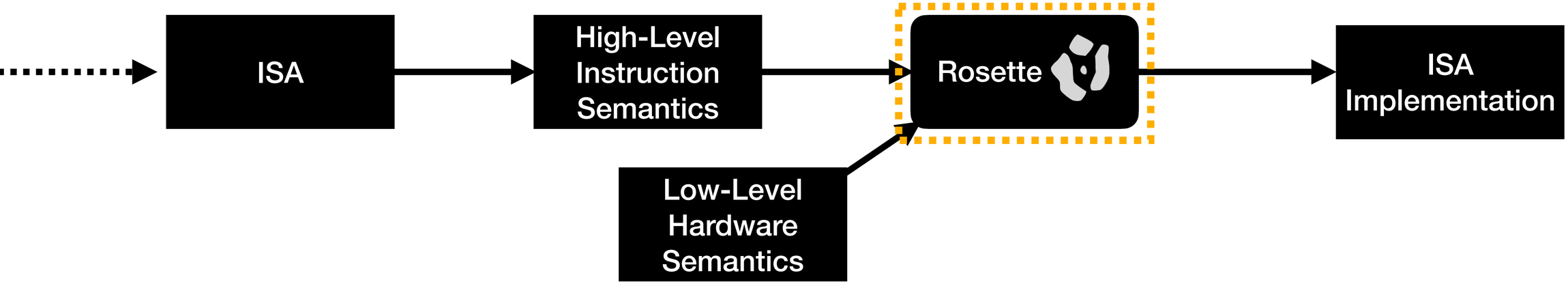
```
(define (lut memory inputs)
  (let* ([inputs (zero-extend inputs (bitvector (length (bitvector->bits memory))))])
    (extract 0 0 (bvlshr memory inputs))))
```

```
(define (ultrascale-plus-lut6-2 memory inputs)
  (let* ([lut5-0 (lut (extract 63 32 memory) (extract 4 0 inputs))]
         [lut5-1 (lut (extract 31 0 memory) (extract 4 0 inputs))]
         [O6 (if (bitvector->bool (bit 5 inputs)) lut5-0 lut5-1)]
         [O5 lut5-1])
    (list O5 O6)))
```

ISA Exploration



ISA Implementation Synthesis



i0 and i1

not (i0 and i1)

not i0

i0 or i1

For all inputs `a` and `b`,
find an implementation of `low-level-and-impl` such that

```
low-level-and-impl(a,b) == high-level-and-impl(a,b)
```

For all inputs `a` and `b`,
find an implementation of `low-level-and-impl` such that

```
low-level-and-impl(a,b) == (bvand a b)
```

For all inputs `a` and `b`,
find a setting of `memory` such that

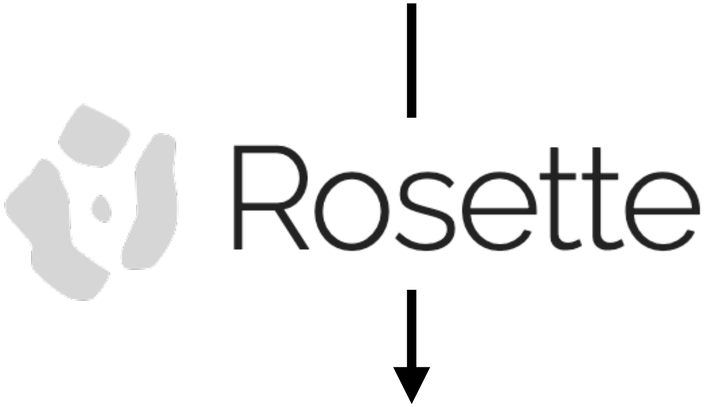
```
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)
```

For all inputs `a` and `b`,
find a setting of `memory` such that

```
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)
```

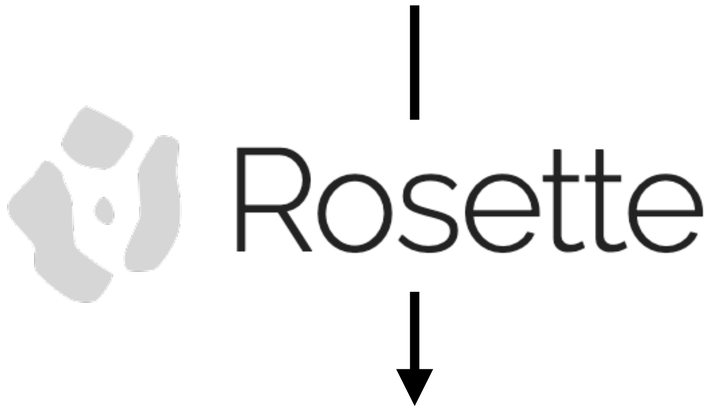
For all inputs `a` and `b`,
find a setting of `memory` such that

```
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)
```



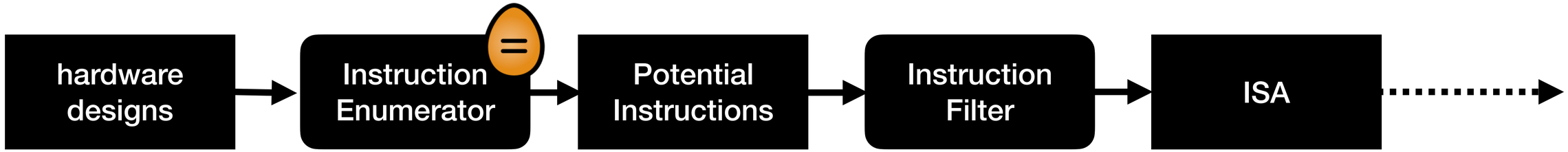
For all inputs a and b,
find a setting of memory such that

```
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)
```

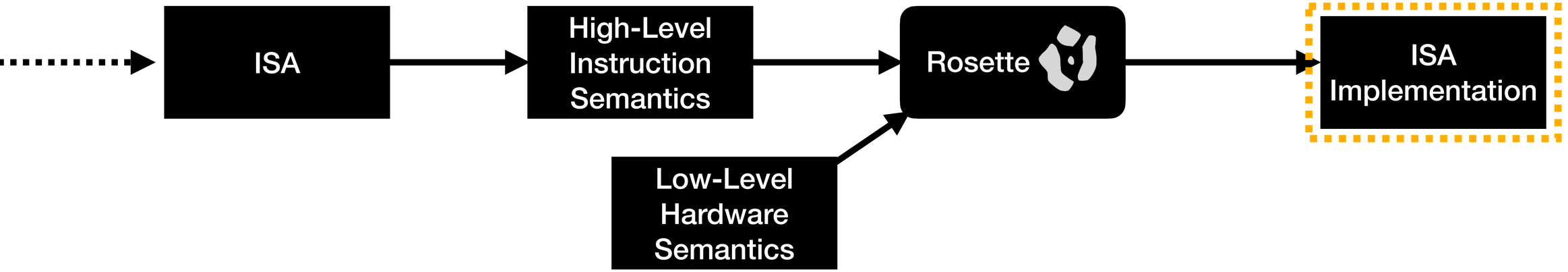



```
memory:
 (bv #x0000000000000008 64)
```

ISA Exploration

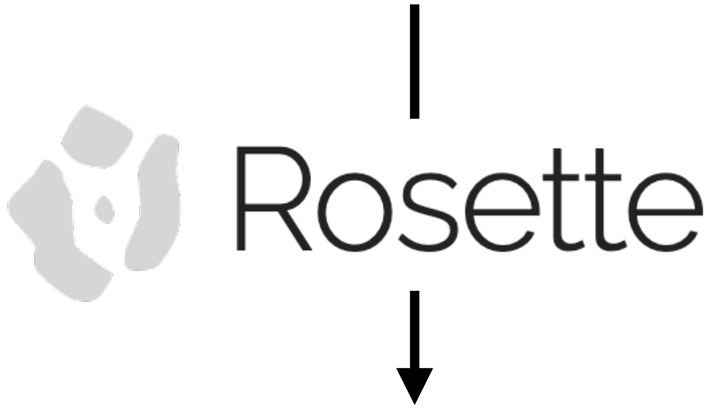


ISA Implementation Synthesis



For all inputs `a` and `b`,
find a setting of `memory` such that

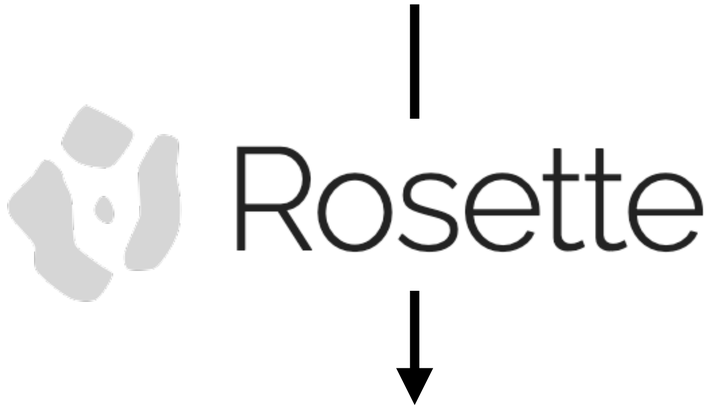
```
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)
```




```
memory:
 (bv #x0000000000000008 64)
```

For all inputs a and b,
find a setting of memory such that

```
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)
```

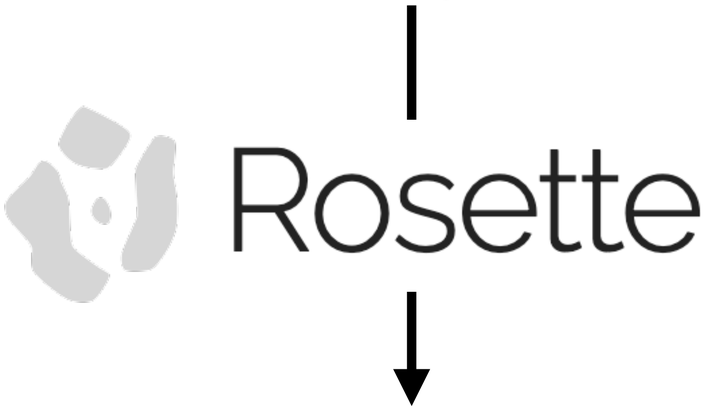



```
memory:
 (bv #x0000000000000008 64)
```

```
module and(a, b, out);
  LUT2 #(
    .INIT(4'h8)
  ) _0_ (.I0(a), .I1(b), .O(out));
endmodule
```

For all inputs a and b,
find a setting of memory such that

```
(ultrascale-plus-lut6-2 memory (list a b)) == (bvand a b)
```



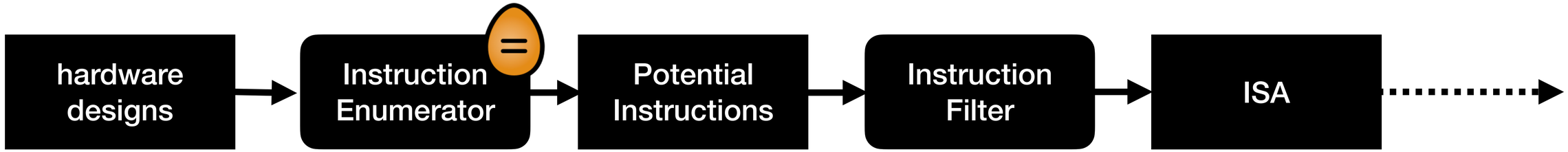

```
memory:
 (bv #x0000000000000008 64)
```

```
module and(a, b, out);
  LUT2 #(
    .INIT(4'h8)
  ) _0_ (.I0(a), .I1(b), .O(out));
endmodule
```

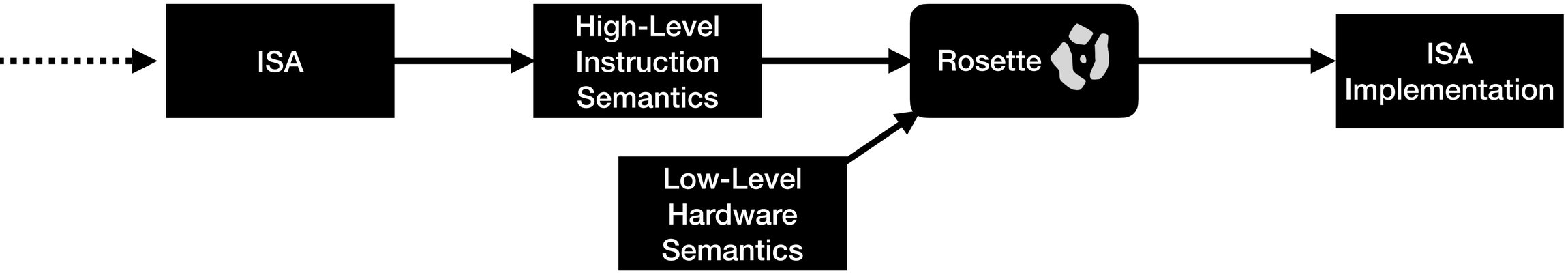
To support new FPGA architectures…

To support new FPGA architectures…
…just provide an interpreter!

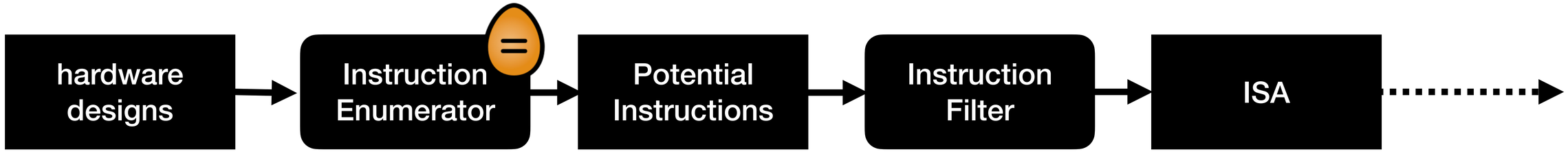
ISA Exploration



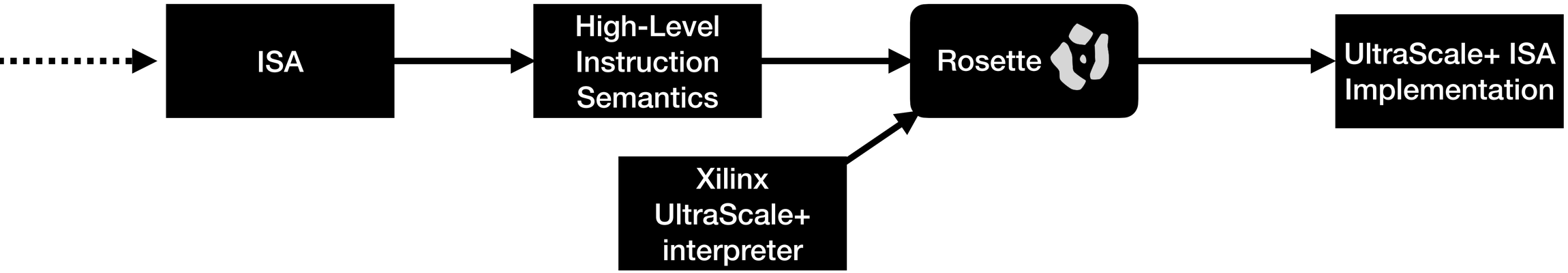
ISA Implementation Synthesis



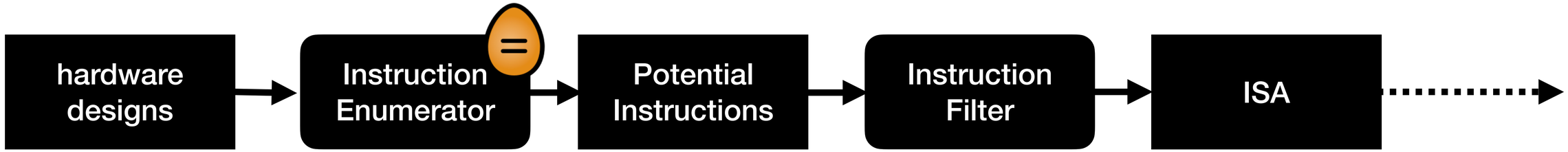
ISA Exploration



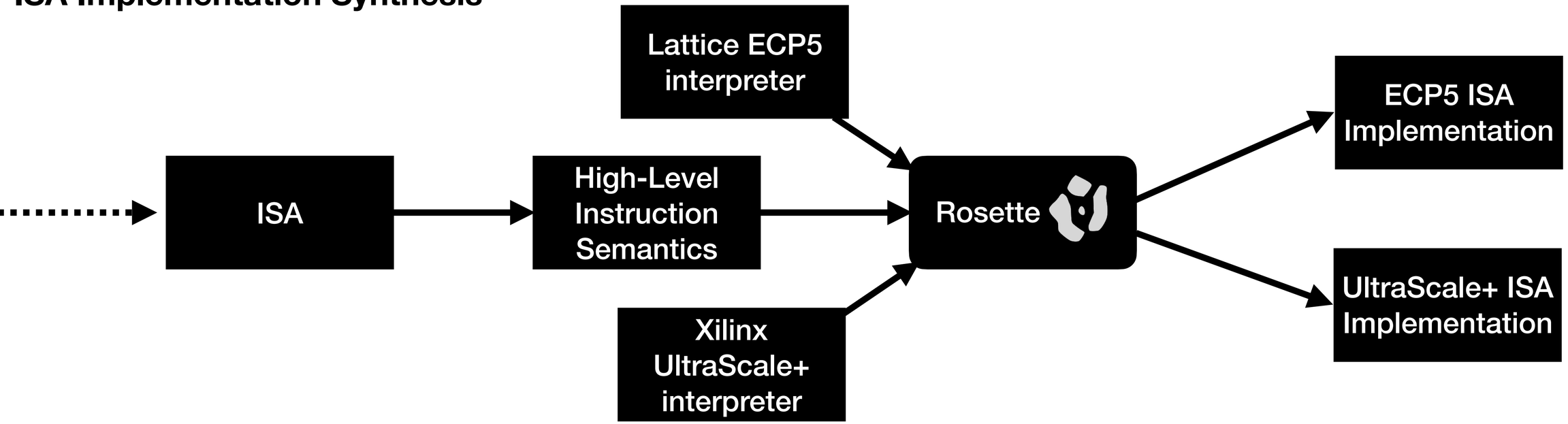
ISA Implementation Synthesis



ISA Exploration



ISA Implementation Synthesis



Finally, to compile a new design, we just need to:

Finally, to compile a new design, we just need to:

1. Insert it into the egraph

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites
3. Extract an implementation composed of instructions in our ISA

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites
3. Extract an implementation composed of instructions in our ISA
4. Output Verilog

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites
3. Extract an implementation composed of instructions in our ISA
4. Output Verilog

If the design isn’t covered with the current ISA, we can:

Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites
3. Extract an implementation composed of instructions in our ISA
4. Output Verilog

If the design isn’t covered with the current ISA, we can:

- Run rewrites to find alternative implementations of the design

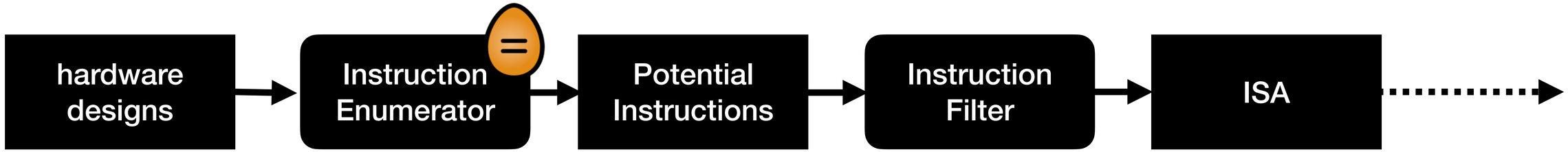
Finally, to compile a new design, we just need to:

1. Insert it into the egraph
2. Enumerate its instructions via rewrites
3. Extract an implementation composed of instructions in our ISA
4. Output Verilog

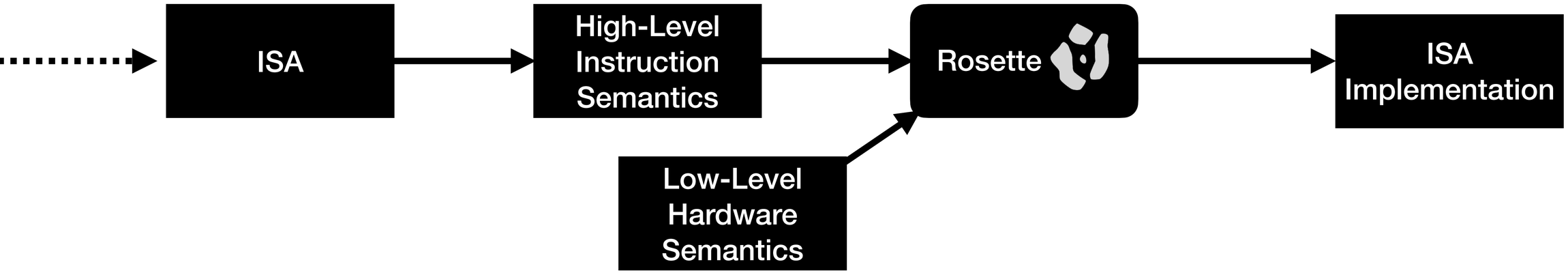
If the design isn’t covered with the current ISA, we can:

- Run rewrites to find alternative implementations of the design
- Find a minimal set of new instructions to add to the ISA

ISA Exploration



ISA Implementation Synthesis



Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

Complex optimizations emerge from small logic rewrites

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

Complex optimizations emerge from small logic rewrites

It's automatic!

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

Complex optimizations emerge from small logic rewrites

It's automatic!

ISA implementations found by Rosette are correct by construction

Proposed Evaluation

Proposed Evaluation

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

- Goal: Evaluate the quality of synthesized implementations; demonstrate ability to support new FPGAs.

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

- Goal: Evaluate the quality of synthesized implementations; demonstrate ability to support new FPGAs.
- We will synthesize three ISAs (Reticle, Calyx, MLIR Comb) for three FPGAs (UltraScale+, ECP5, SOFA)

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

- Goal: Evaluate the quality of synthesized implementations; demonstrate ability to support new FPGAs.
- We will synthesize three ISAs (Reticle, Calyx, MLIR Comb) for three FPGAs (UltraScale+, ECP5, SOFA)

Paper 2: ISA Exploration and Lakeroad End-to-End

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

- Goal: Evaluate the quality of synthesized implementations; demonstrate ability to support new FPGAs.
- We will synthesize three ISAs (Reticle, Calyx, MLIR Comb) for three FPGAs (UltraScale+, ECP5, SOFA)

Paper 2: ISA Exploration and Lakeroad End-to-End

- Goal: Demonstrate ability to enumerate a large space of interesting instructions; demonstrate fast compilation using the egraph.

Proposed Evaluation

Paper 1: ISA Implementation Synthesis

- Goal: Evaluate the quality of synthesized implementations; demonstrate ability to support new FPGAs.
- We will synthesize three ISAs (Reticle, Calyx, MLIR Comb) for three FPGAs (UltraScale+, ECP5, SOFA)

Paper 2: ISA Exploration and Lakeroad End-to-End

- Goal: Demonstrate ability to enumerate a large space of interesting instructions; demonstrate fast compilation using the egraph.
- We will run Lakeroad end-to-end on a large corpus of hardware benchmarks (from sources like MachSuite)

In Closing

Automatically generating compiler backends from explicit, formal hardware models

- gives rise to emergent optimizations,
- reduces development time, and
- enables verification.

So far, I have provided evidence for this thesis through Glenside and its application in 3LA.

So far, I have provided evidence for this thesis through Glenside and its application in 3LA.

I plan to demonstrate this thesis once more through Lakeroad.

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

October 2022: Submit Lakeroad part 1 to 3rd round of ASPLOS

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

October 2022: Submit Lakeroad part 1 to 3rd round of ASPLOS

Autumn Quarter 2022: Submit 3LA verification paper

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

October 2022: Submit Lakeroad part 1 to 3rd round of ASPLOS

Autumn Quarter 2022: Submit 3LA verification paper

Winter Quarter 2023: Fulfill final TA requirement

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

October 2022: Submit Lakeroad part 1 to 3rd round of ASPLOS

Autumn Quarter 2022: Submit 3LA verification paper

Winter Quarter 2023: Fulfill final TA requirement

Winter/Spring Quarter 2023: Deal with Lakeroad and 3LA resubmissions

June 2022: Submit Lakeroad part 2 to 2nd round of ASPLOS

June 2022: Resubmit 3LA paper to 2nd round of ASPLOS

October 2022: Submit Lakeroad part 1 to 3rd round of ASPLOS

Autumn Quarter 2022: Submit 3LA verification paper

Winter Quarter 2023: Fulfill final TA requirement

Winter/Spring Quarter 2023: Deal with Lakeroad and 3LA resubmissions

Spring Quarter 2023: Write thesis and defend

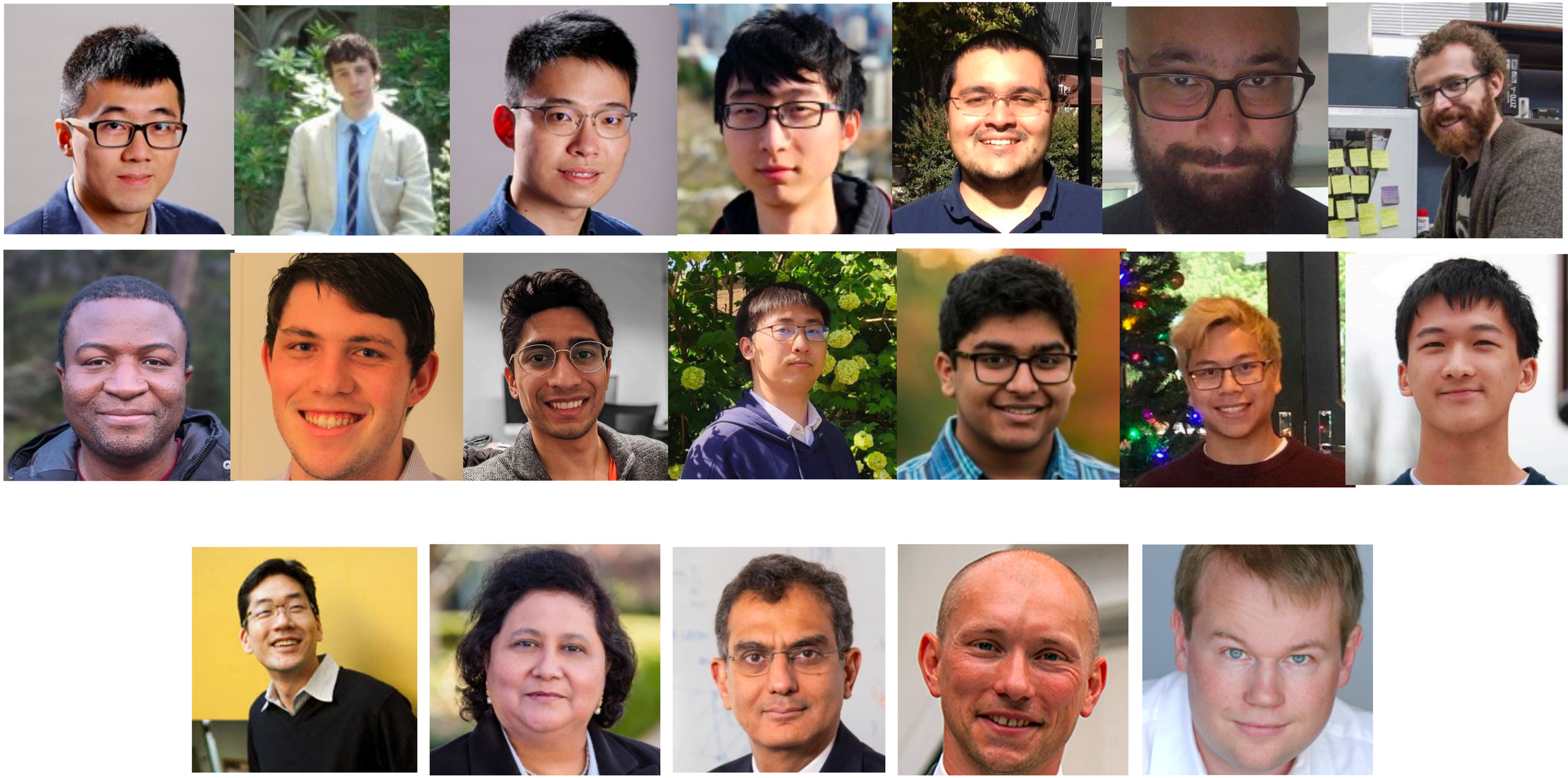
Acknowledgements

CRISP
Center for Research in Intelligent
Storage and Processing in Memory


sampl


PLSE


sampa



Thank you!

Extra Slides

Rest of Glenside Talk

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation

Given matrices A and B, pair each row of A with each column of B, compute their dot products, and arrange the results back into a matrix.

```
(cartProd
 (access A 1)

  (list 1 0))))
```
; ((3),(4))

```
(cartProd
 (access A 1)
 (list 1 0))))
```

Access A as a list of its rows

; ((3),(4))

```
(cartProd
 (access A 1)

  (list 1 0))))
```

; ((3), (4))

; ((4), (2))

```
(cartProd
 (access A 1)

  (list 1 0))))
```

```
; ((3), (4))
; ((2), (4))
; ((4), (2))
```

```
(cartProd
 (access A 1)

 (list 1 0))))
```

; ((3), (4))

; ((2), (4))

; ((4), (2))

Access B as a list of its rows, then transpose into a list of its columns

```
(cartProd
 (access A 1)

  (access B 1)
  (list 1 0))))
```

```
; ((3, 2), (2, 4))
; ((3), (4))
; ((2), (4))
; ((4), (2))
```

```
(cartProd
  (access A 1)
    (access B 1)
    (list 1 0))))
```

Create every row–column pair

```
; ((3, 2), (2, 4))
; ((3), (4))
; ((2), (4))
; ((4), (2))
```

```
(cartProd
  (access A 1)
  (transpose
    (access B 1)
    (list 1 0))))
```

```
; ((3, 2), ())
; ((3, 2), (2, 4))
; ((3), (4))
; ((2), (4))
; ((4), (2))
```

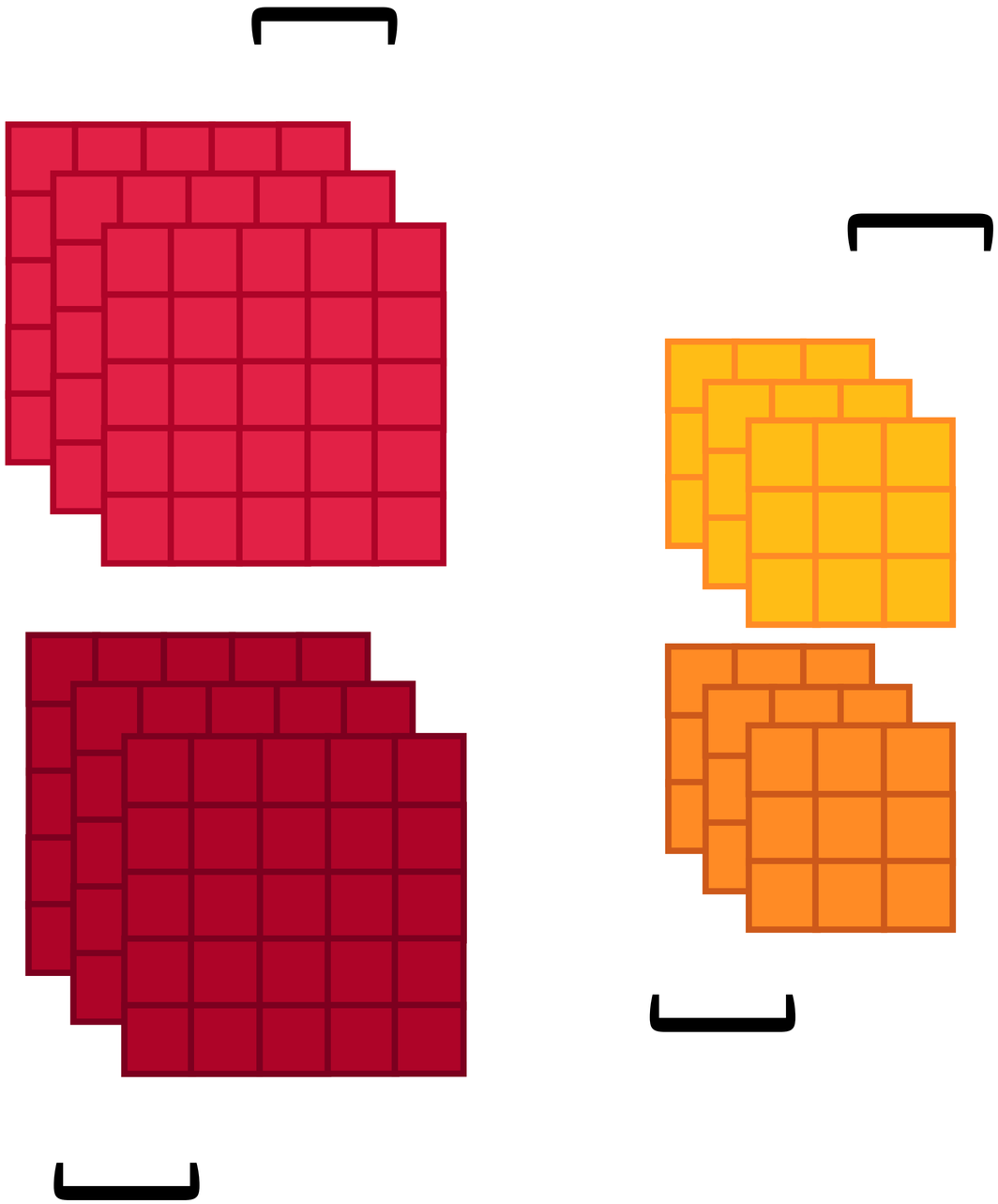
```
(cartProd
  (
  (transpose
   (access B 1)
   (list 1 0))))
```

Compute dot product of every row–column pair

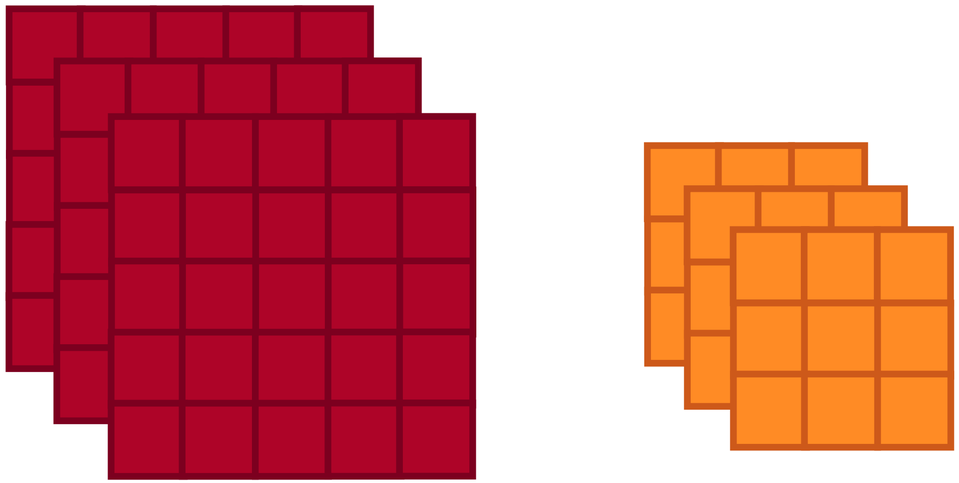
```
; ((3, 2), ())
; ((3, 2), (2, 4))
; ((3), (4))
; ((2), (4))
; ((4), (2))
```

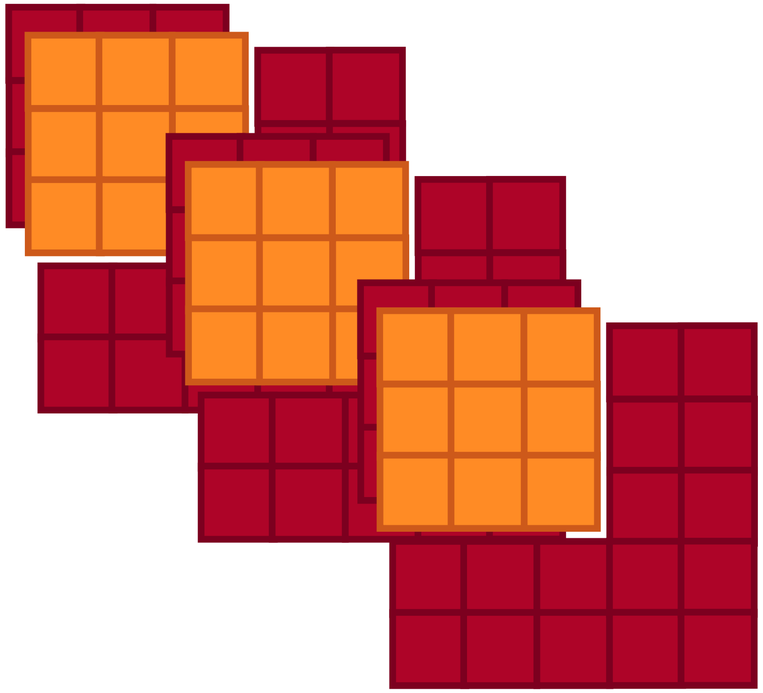
Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation

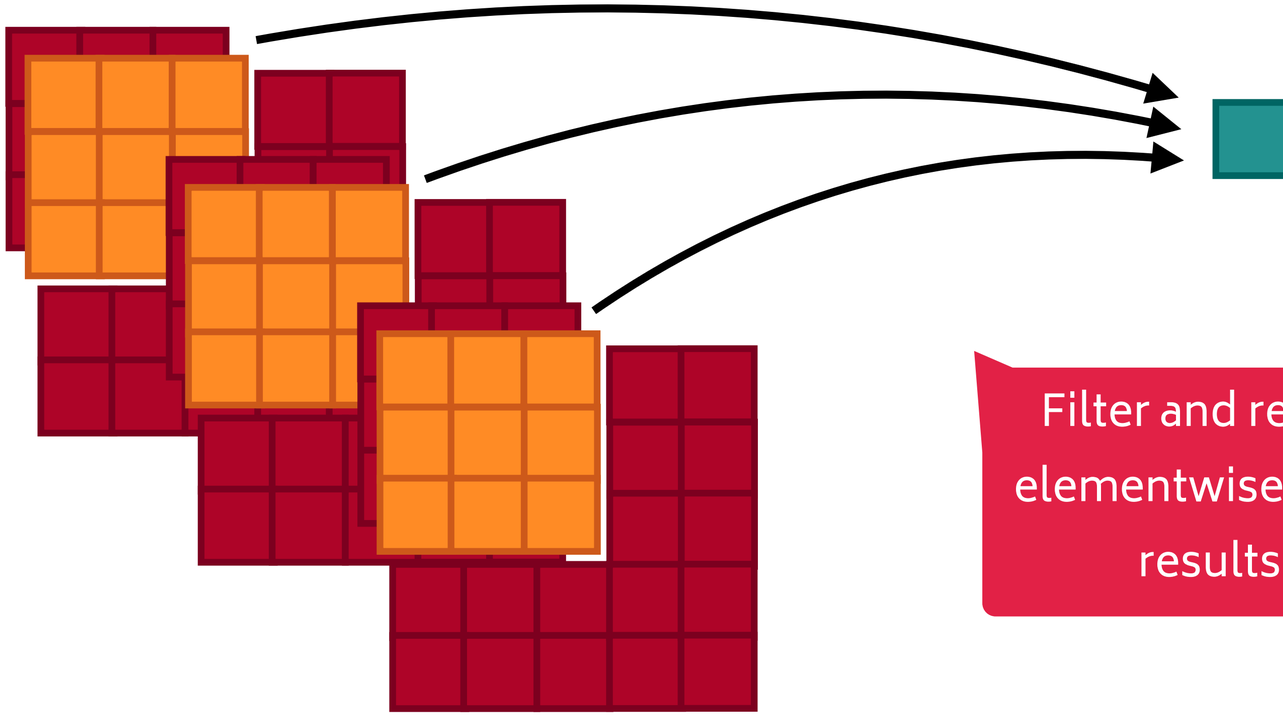


Inputs: a batch of image/activation tensors and a list of weight/filter tensors

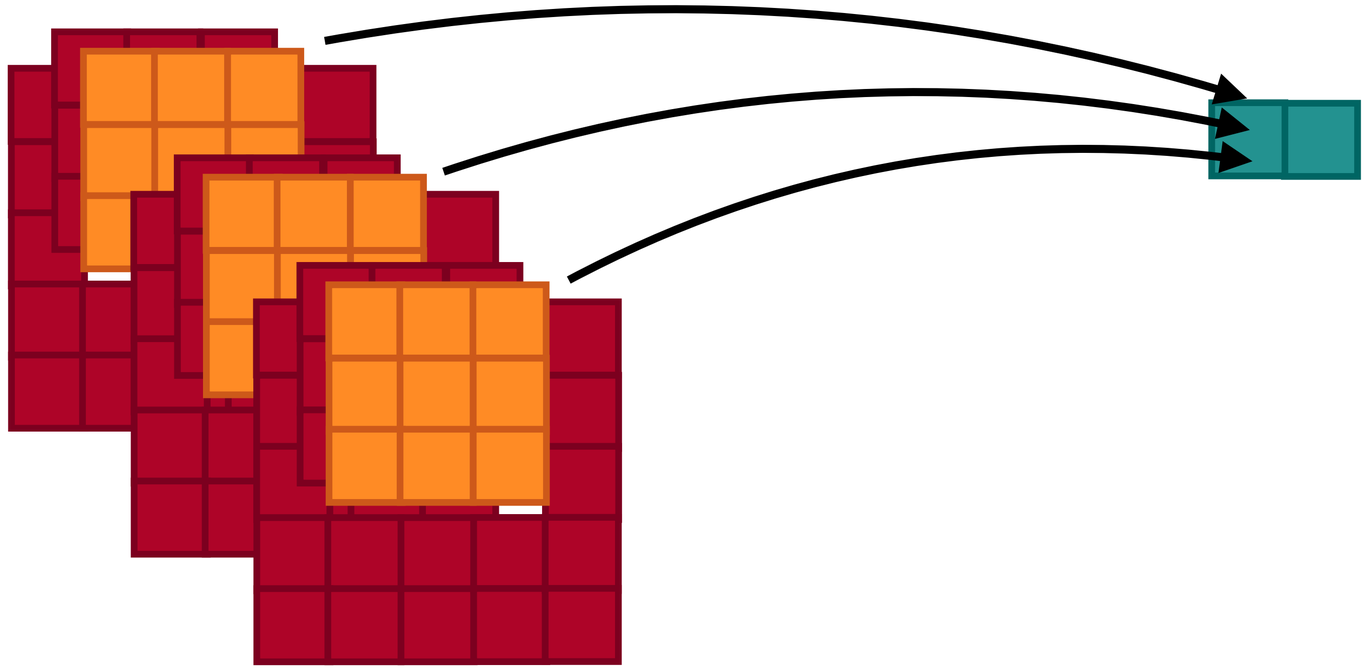


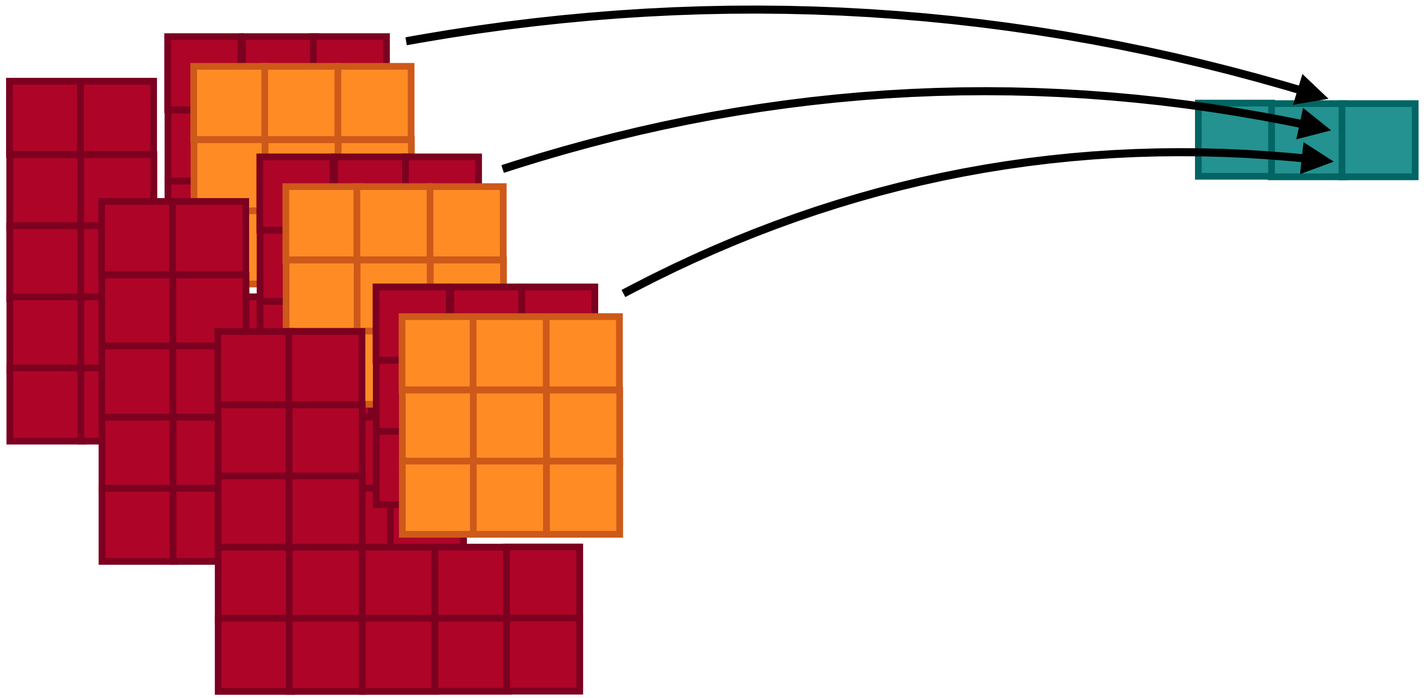


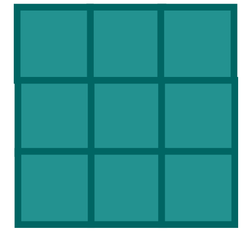
Filter and region of image are elementwise multiplied and the results are summed

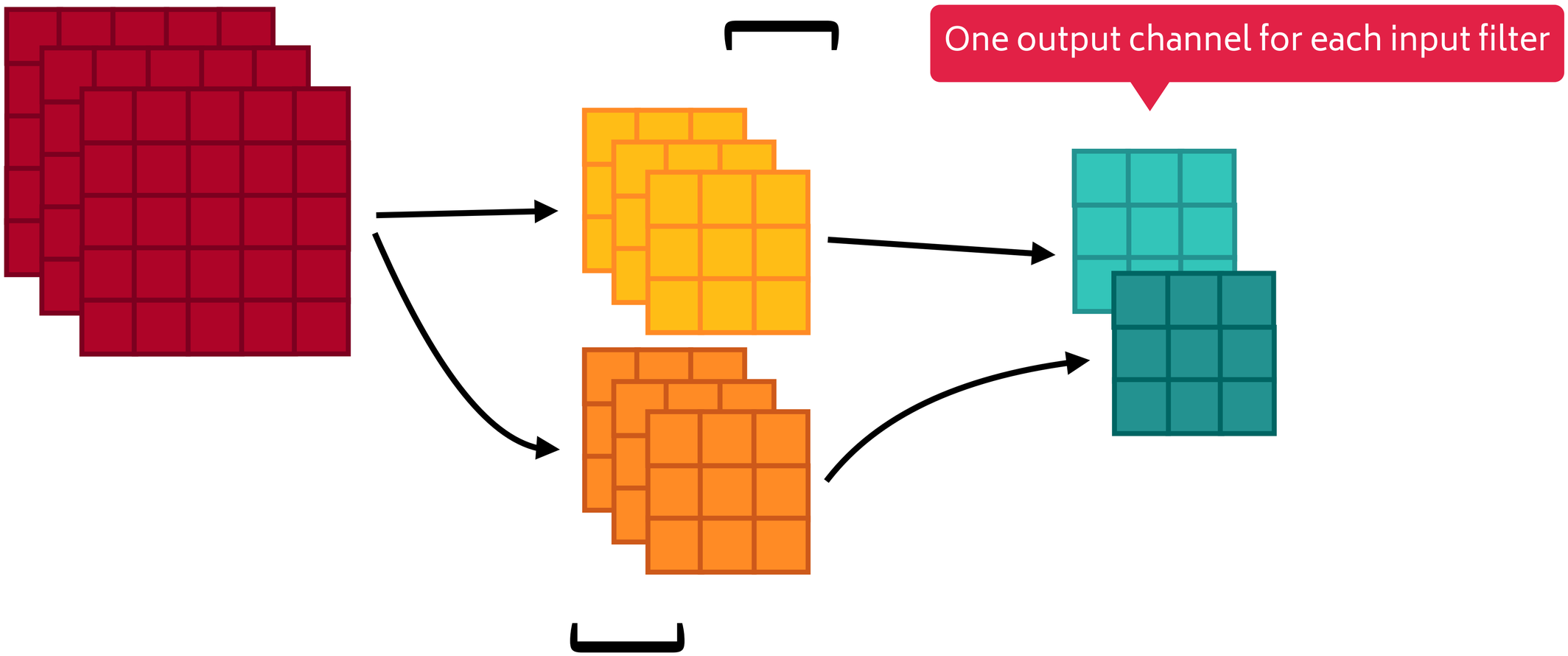


Filter and region of image are elementwise multiplied and the results are summed








One output channel for each input filter

(windows

(access activations 1)

Access weights as a list of 3D filters

; $((O), (C, K_h, K_w))$

```
(windows
 (access activations 1)
```

Access activations as a batch of 3D images

```
(list 0 3 1 2))
```

; $((N), (C, H, W))$

; $((O), (C, K_h, K_w))$

```
(windows
 (access activations 1)
 1)
(list 0 3 1 2))                    ; ((N), (C, H, W))
                                   ; ((O), (C, Kh, Kw))
```

Form windows over input images

```
   (windows
    (access activations 1)
 1)
(list 0 3 1 2))                                    ; ((N), (C, H, W))
                                                   ; ((O), (C, Kh, Kw))
```

These parameters control window shape and strides

```
(windows
 (access activations 1)
 1)
(list 0 3 1 2))
```

At each location in each new image, there is a (C, K_h, K_w)-shaped window

; $((N, 1, H', W'), (C, K_h, K_w))$

; $((N), (C, H, W))$

; $((O), (C, K_h, K_w))$

```
   (windows
    (access activations 1)
    (access weights 1)))
 1)
(list 0 3 1 2))
```

Pair windows with filters

; $((N, 1, H', W', O), (2, C, K_h, K_w))$

; $((N, 1, H', W'), (C, K_h, K_w))$

; $((N), (C, H, W))$

; $((O), (C, K_h, K_w))$

```
   (windows
    (access activations 1)
    (shape 1 sh sw))
   (access weights 1)))
 1)
(list 0 3 1 2))
```

Compute dot product of each window–filter pair

; ((N, 1, H', W', O), ())

; ((N, 1, H', W', O), (2, C, K_h, K_w))

; ((N, 1, H', W'), (C, K_h, K_w))

; ((N), (C, H, W))

; ((O), (C, K_h, K_w))

```
    (windows
     (access activations 1)
     (shape
     (shape 1 Sh Sw))
    (access weights 1)))
 1)
(list 0 3 1 2))
```

Remove and rearrange dimensions

; $((N, O, H', W'), ())$

; $((N, 1, H', W', O), ())$

; $((N, 1, H', W', O), (2, C, K_h, K_w))$

; $((N, 1, H', W'), (C, K_h, K_w))$

; $((N), (C, H, W))$

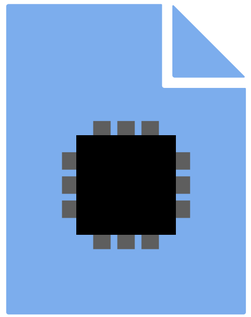
; $((O), (C, K_h, K_w))$

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation

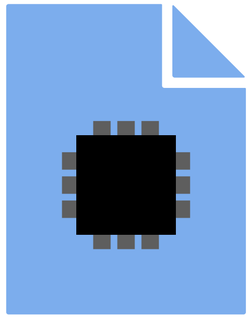
It reads an entire weight array of shape `rows` by `cols`.

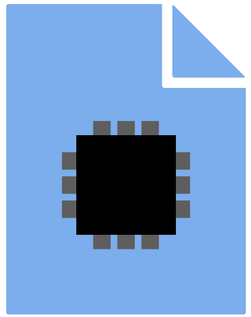
It then pushes `n` vectors of length `rows` through the array.



It computes the dot product of every vector with every column of the weights.

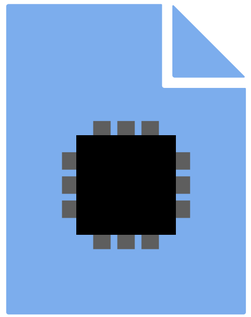
Finally, it writes out `n` vectors of length `cols`.





Can we represent hardware as a searchable pattern?

```
(compute dotProd
 (cartProd ?a0 ?a1))

  where ?a0 is of shape
   ((?n), (?rows))

  and ?a1 is of shape
   ((?cols), (?rows))
```



With Glenside, we can!

```
(compute dotProd
 (cartProd ?a0 ?a1))
  where ?a0 is of shape
   ((?n), (?rows))
  and ?a1 is of shape
   ((?cols), (?rows))
```

We can directly rewrite to hardware invocations!

```
(systolicArray ?rows ?cols ?a0 ?a1)
```

```
 ((?n), (?rows))

and ?a1 is of shape
 ((?cols), (?rows))
```

```
(systolicArray ?rows ?cols ?a0 ?a1)

(compute dotProd
 (cartProd
  (access A 1)
  (transpose
   (access B 1)
   (list 1 0))))
```

```
  ((?n), (?rows))

and ?a1 is of shape
  ((?cols), (?rows))
```

```
(systolicArray ?rows ?cols ?a0 ?a1)
```

```
(compute dotProd
 (cartProd
  (access A 1)
  (transpose
   (access B 1)
   (list 1 0))))
```

```
 ((?n), (?rows))

and ?a1 is of shape
 ((?cols), (?rows))
```

```
(systolicArray ?rows ?cols ?a0 ?a1)
```

```
(systolicArray
  4 2
  (access A 1)
  (transpose
   (access B 1)
   (list 1 0))))
```

Outline

- Motivating Example: Matrix Multiplication
- Access Pattern Definition
- Case Studies
  - Reimplementing Matrix Multiplication with Access Patterns
  - Implementing 2D Convolution with Access Patterns
  - Hardware Mapping as Program Rewriting
  - Flexible Hardware Mapping with Equality Saturation

```
    (windows
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))
 1)
(list 0 3 1 2))
```

```
(compute dotProd
 (cartProd
  (access A 1)
  (transpose
   (access B 1)
   (list 1 0))))
```

```
    (windows
     (access activ
     (shape C Kh K
     (shape 1 Sh Sw))
    (access weights 1)))
 1)
(list 0 3 1 2))
```

Convolution and matrix multiplication have similar structure!

```
(compute dotProd
 (cartProd
  (access A 1)
  (transpose
   (access B 1)
   (list 1 0))))
```

```
    (windows
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))
 1)
(list 0 3 1 2))
```

```
(compute dotProd
 (cartProd ?a0 ?a1))
```

Can we apply our hardware rewrite?

and ?a1 is of shape
((?cols), (?rows))

```
   (windows
    (access activations 1)
    (shape C Kh Kw)
    (shape 1 Sh Sw))
   (access weights 1)))
 1)
(list 0 3 1 2))
```

;((N, 1, H', W'), (C, Kh, Kw))

;((O), (C, Kh, Kw))

```
(compute dotProd
 (cartProd ?a0 ?a1))
  where ?a0 is of shape
   ((?n), (?rows))
  and ?a1 is of shape
   ((?cols), (?rows))
```

Our access pattern shapes do not pass the rewrite's conditions

```
   (windows
    (access activations 1)
    (shape C Kh Kw)
    (shape 1 Sh Sw))
   (access weights 1)))
 1)              ;((?n),(?rows))
(list 0 3 1 2))
```

```
(compute dotProd
 (cartProd ?a0 ?a1))
  where ?a0 is of shape
   ((?n), (?rows))
  and ?a1 is of shape
   ((?cols), (?rows))
```

```
;((?cols),(?rows))
```

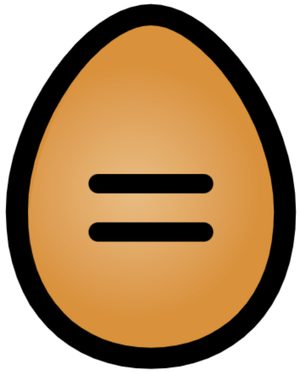
Can we flatten our access patterns?

```
?a → (reshape (flatten ?a) ?shape)
```

Flattens and immediately reshapes an access pattern

```
?a → (reshape (flatten ?a) ?shape)
```

Flattens and immediately reshapes an access pattern



```
    (windows
     (access activations 1)
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))
 1)                  ; ((N, 1, H', W'), (C, Kh, Kw))
(list 0 3 1 2))
```

; ((O), (C, Kh, Kw))

```
   (reshape (flatten (windows
    (access activations 1)
    (shape C Kh Kw)
    (shape 1 Sh Sw))) ?shape0)
   (reshape (flatten (access weights 1)) ?shape1)))
 1)                                    ; ((N, 1, H', W'), (C, Kh, Kw))
(list 0 3 1 2))
```

; ((O), (C, Kh, Kw))

```
   (reshape (flatten (windows
    (access activations 1)
    (shape C Kh Kw)
    (shape 1 Sh Sw))) ?shape0)
   (reshape (flatten (access we
 1)
(list 0 3 1 2))
```

But our access pattern shapes haven't changed!

; ((N, 1, H', W'), (C, Kh, Kw))

; ((O), (C, Kh, Kw))

```
(reshape (flatten (windows
 (access activations 1)
 (shape C Kh Kw)
 (shape 1 Sh Sw))) ?shape0)
(reshape (flatten (access weights 1)) ?shape1)))
```

```
1)
(list 0 3 1 2))
```

; ((N, 1, H', W'), (C, Kh, Kw))

We need to "bubble" the `reshapes` to the top

; ((O), (C, Kh, Kw))

These rewrites "bubble" `reshape` through `cartProd` and `compute dotProd`

```
(cartProd
 (reshape ?a0 ?shape0)
 (reshape ?a1 ?shape1)) → (reshape (cartProd ?a0 ?a1) ?newShape)
```

```
(compute dotProd
 (reshape ?a ?shape))   → (reshape (compute dotProd ?a) ?newShape)
```

```
   (reshape (flatten (windows
    (access activations 1)
    (shape C Kh Kw)
    (shape 1 Sh Sw))) ?shape0)
   (reshape (flatten (access weights 1)) ?shape1)))
 1)                                    ; ((N, 1, H', W'), (C, Kh, Kw))
(list 0 3 1 2))
```

; ((O), (C, Kh, Kw))

```
   (flatten (windows
    (access activations 1)
    (shape C Kh Kw)
    (shape 1 Sh Sw)))
   (flatten (acc
 1)
(list 0 3 1 2))
```

reshapes have been moved out, and the access patterns are flattened!

$; ((N \cdot 1 \cdot H' \cdot W'), (C \cdot Kh \cdot Kw))$

$; ((O), (C \cdot Kh \cdot Kw))$

```
   (flatten (windows
    (access activations 1)
    (shape C Kh Kw)
    (shape 1 Sh Sw)))
   (flatten (access weights 1)))) ?shape)
 1)
(list 0 3 1 2))
```

((?n), (?rows))

and ?a1 is of shape
((?cols), (?rows))

$;((N \cdot 1 \cdot H' \cdot W'), (C \cdot Kh \cdot Kw))$

Our rewrite can now map convolution to matrix multiplication hardware!

$;((O), (C \cdot Kh \cdot Kw))$

```
                    ?a → (reshape (flatten ?a) ?shape)
(cartProd
 (reshape ?a0 ?shape0)
 (reshape ?a1 ?shape1)) → (reshape (cartProd ?a0 ?a1) ?newShape)


(compute dotProd
 (reshape ?a ?shape))   → (reshape (compute dotProd ?a) ?newShape)
```

These rewrites rediscover the im2col transformation!

In conclusion,

In conclusion,

we have presented **access patterns** as a new tensor representation,

In conclusion,

we have presented **access patterns** as a new tensor representation,

we have used them to build the **pure, low-level, binder free IR Glenside,**

In conclusion,

we have presented **access patterns** as a new tensor representation,

we have used them to build the **pure, low-level, binder free IR Glenside,**

and have shown how they **enable hardware-level tensor program rewriting.**

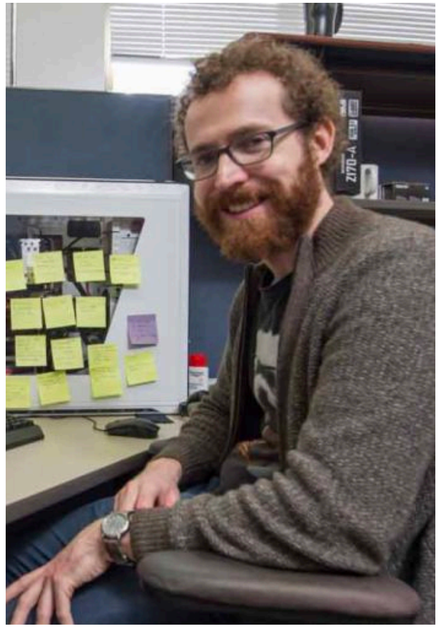
https://github.com/gussmith23/glenside

Glenside is an actively maintained Rust library!
Try it out and open issues if you have questions!

sampl

APLSE

DARPA



CRISP
Center for Research in Intelligent Storage and Processing in Memory

ada
Applications Driving Architectures

Thank you!