Verified Optimization and Mapping of Sparse Tensor Algebra

Scott Kovach Fred Kjolstad

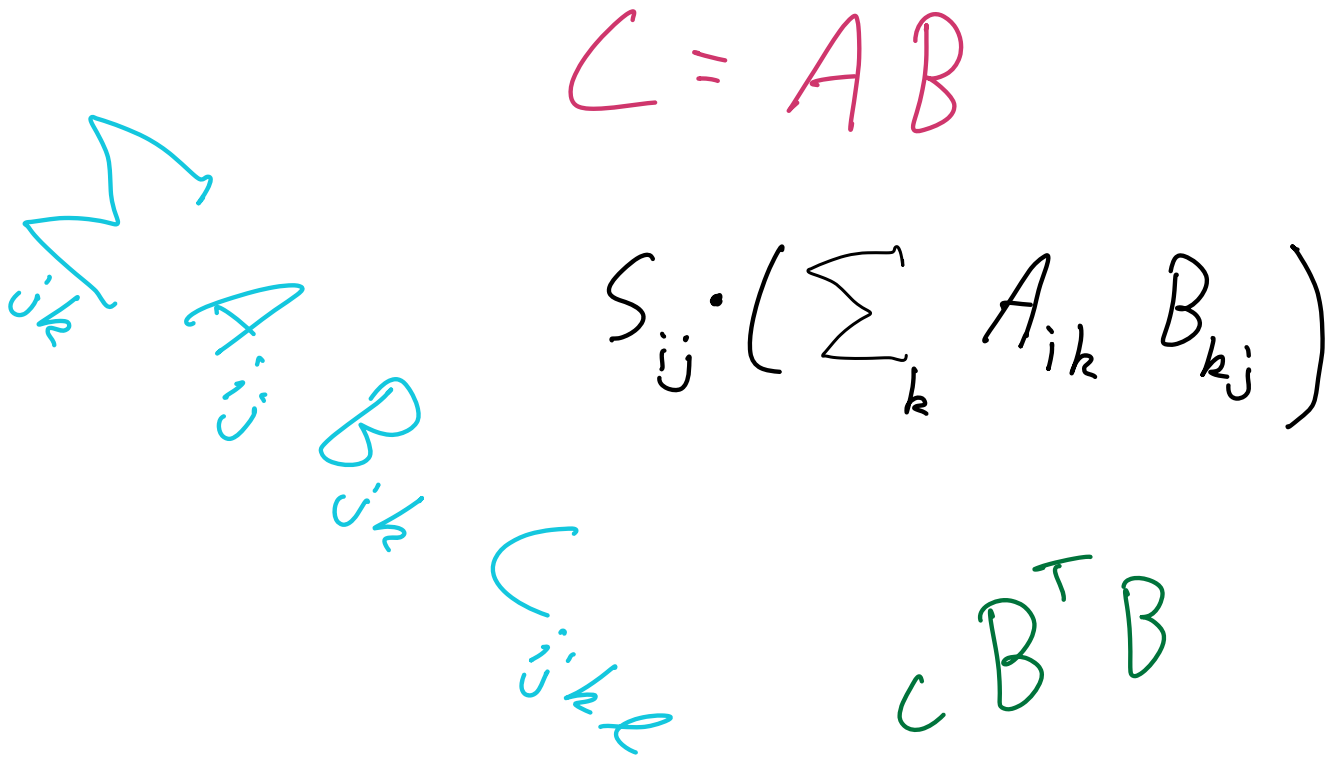
2022/4/6

For more info

http://web.stanford.edu/~dskovach/

Sparse Tensor Algebra

A powerful language for many users in scientific computing, data analytics, machine learning.



- A motivating example:

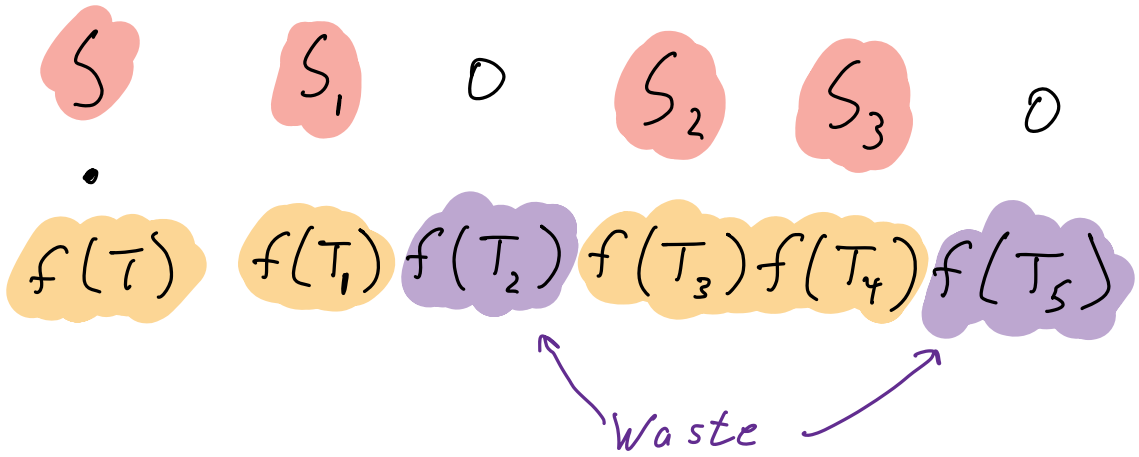
$$S \cdot f(T)$$

- S and T are sparse multi-dimensional arrays ("tensors")
- assume f is expensive, but its output elements can be computed individually (e.g. matrix multiplication)
- $\cdot$ = element-wise multiplication.

The key optimization: fusion

A motivating example for *fusion*:

$$S \cdot f(T)$$

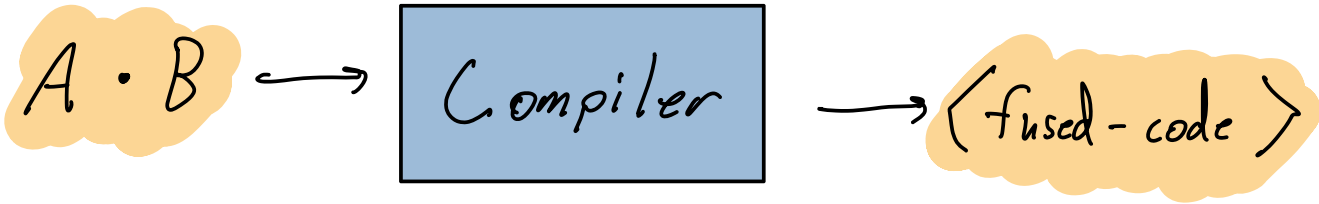


The key optimization: fusion

We want the structure of S to drive the computation of $f(T)$; that is, we want to *fuse* the sub-computations across the boundary of the product operation.

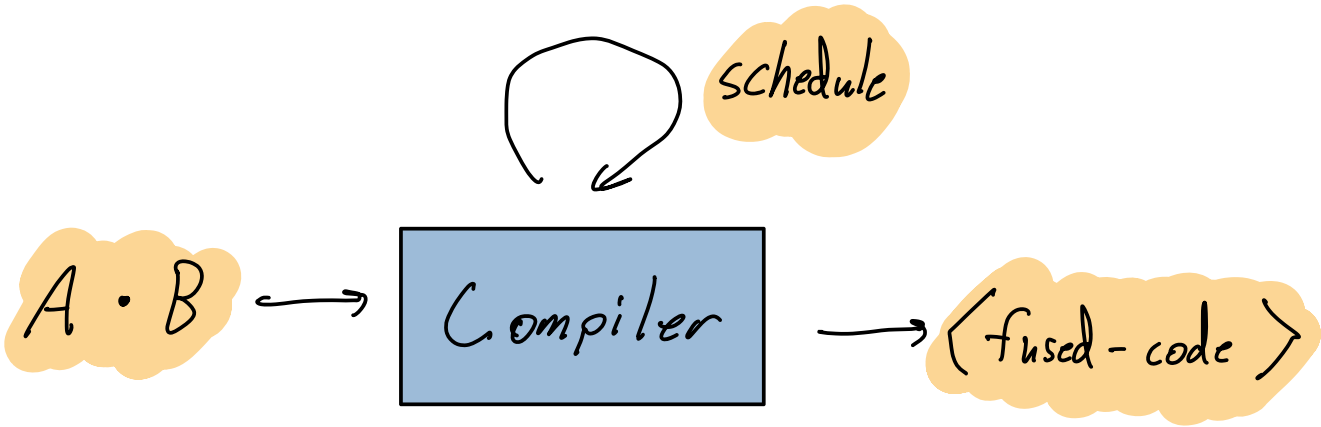
The key optimization: fusion

This can be automated [TACO]. Users write standard tensor expressions without worrying about fusion.

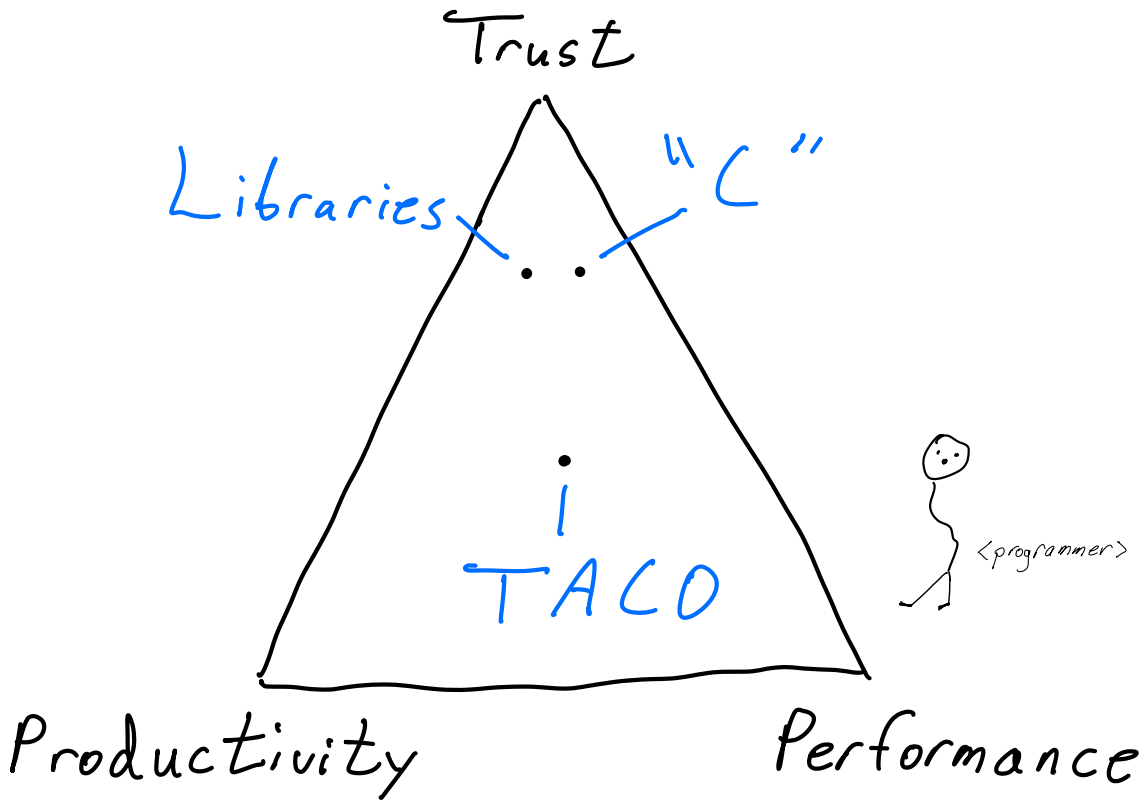


Scheduling Optimizations

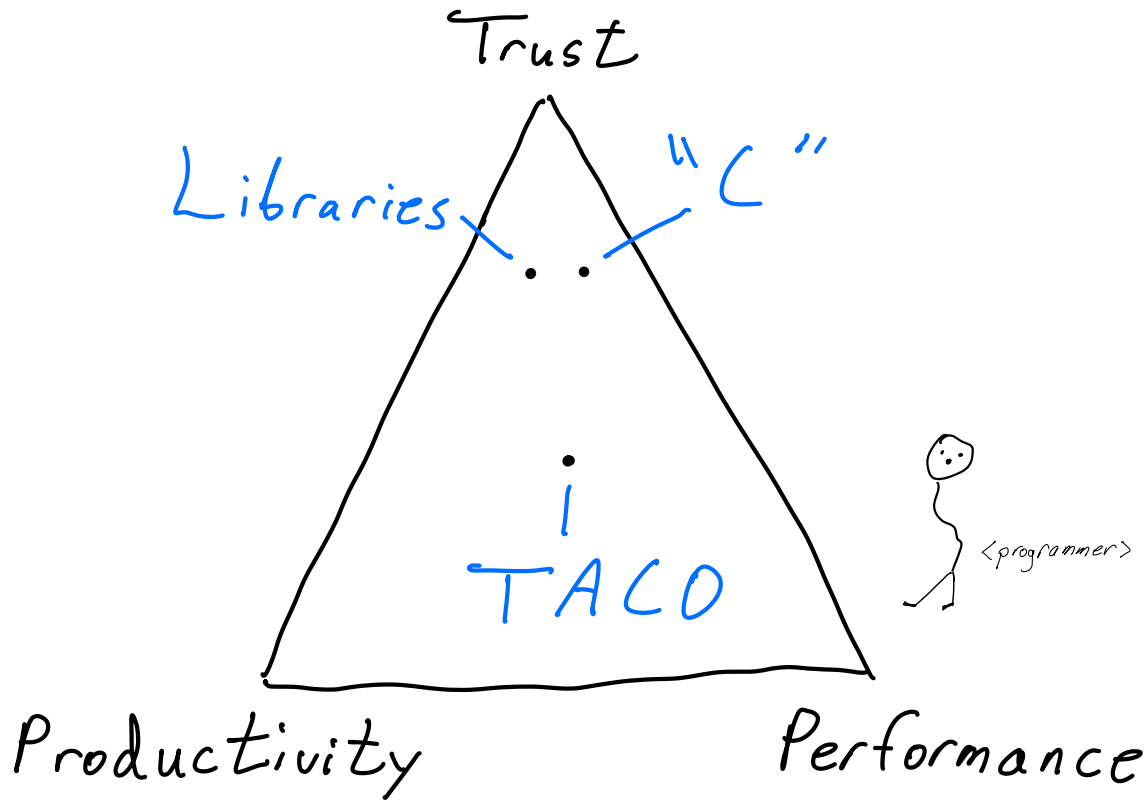
Separating algorithm from schedule further relaxes the productivity-performance tradeoff [Halide].



What does taco replace?



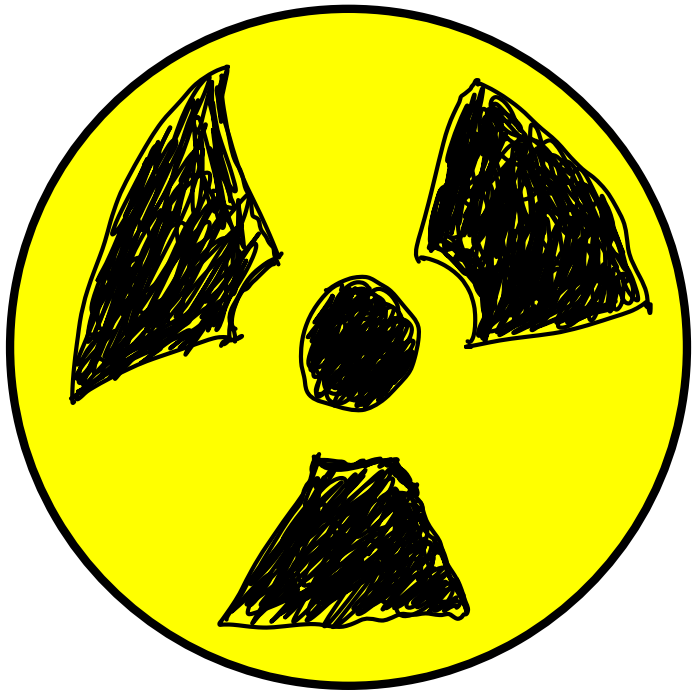
What does taco replace?



TACO automatically generates fused kernels, but...

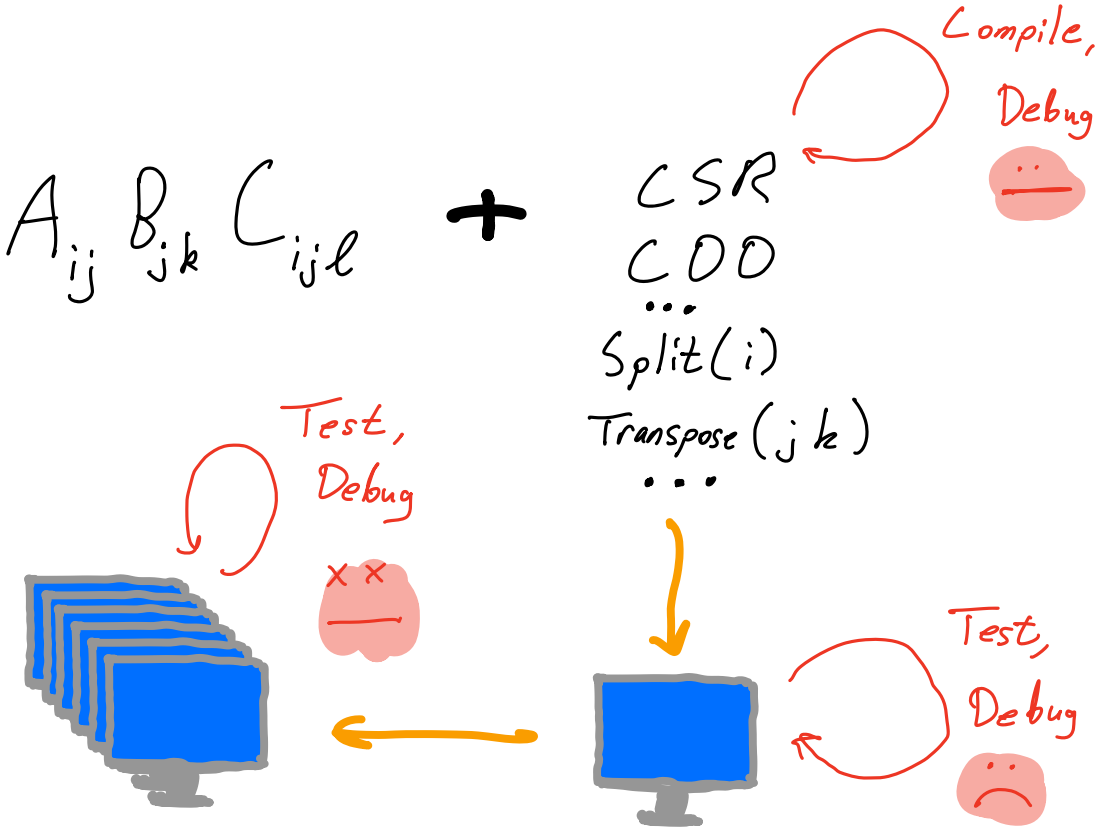
Compiler trust?

Users with especially stringent safety concerns (e.g. simulations or controllers for nuclear power, civil engineering, aerospace) may hesitate to take up an experimental research compiler.



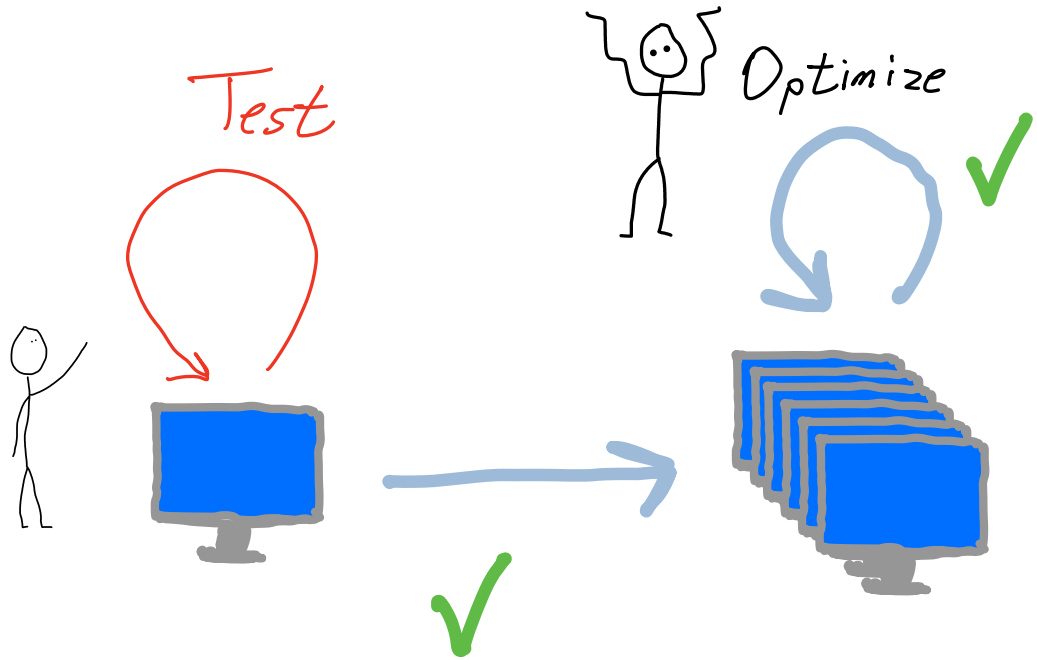
Compiler trust?

Users lose productivity if they must test generated code.



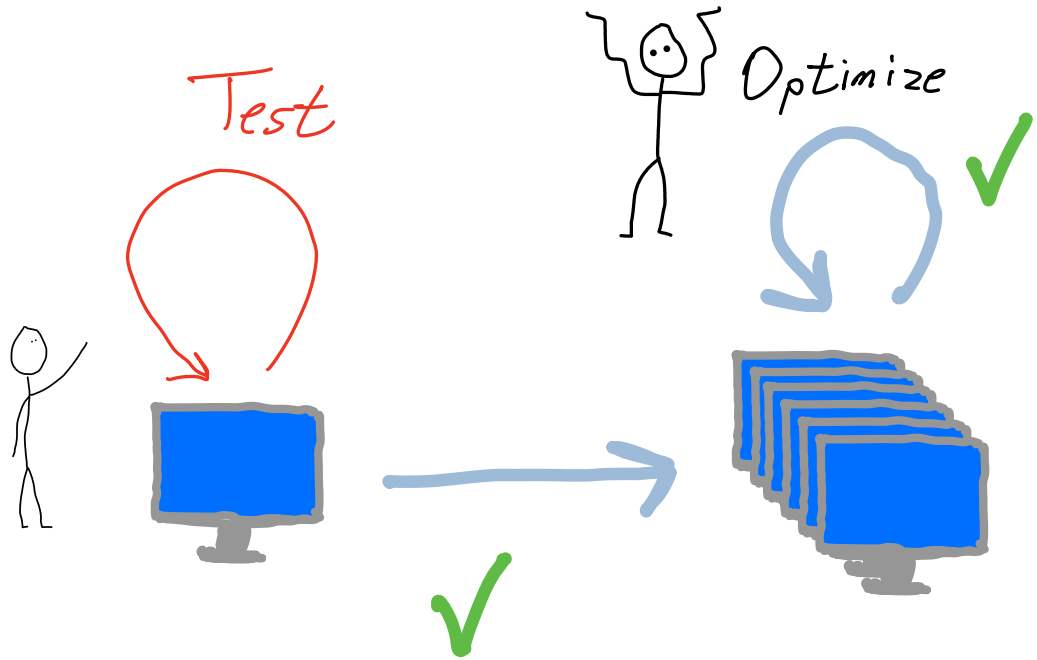
Goal:

- Higher productivity via extensible, trusted compilation



Goal:

- Higher productivity via extensible, trusted compilation



- Build machine-checked proof that a compiler preserves the program's meaning [a la CompCert].

How to even write a compiler for sparse tensor algebra?

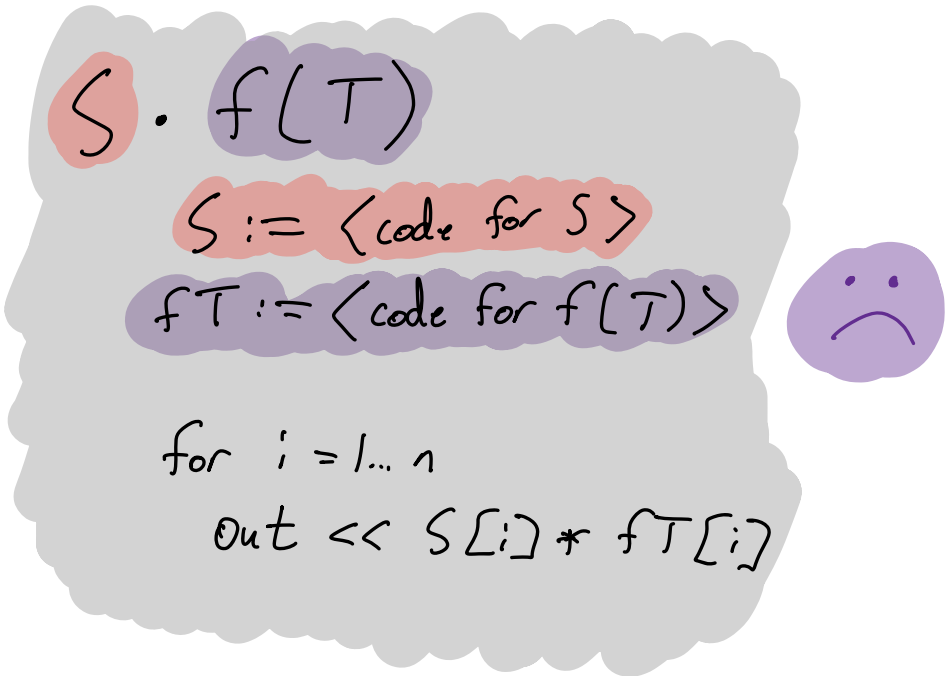
How do we solve the fusion problem automatically?

$$S \cdot f(T)$$

How to even write a compiler for sparse tensor algebra?

How do we solve the fusion problem automatically?

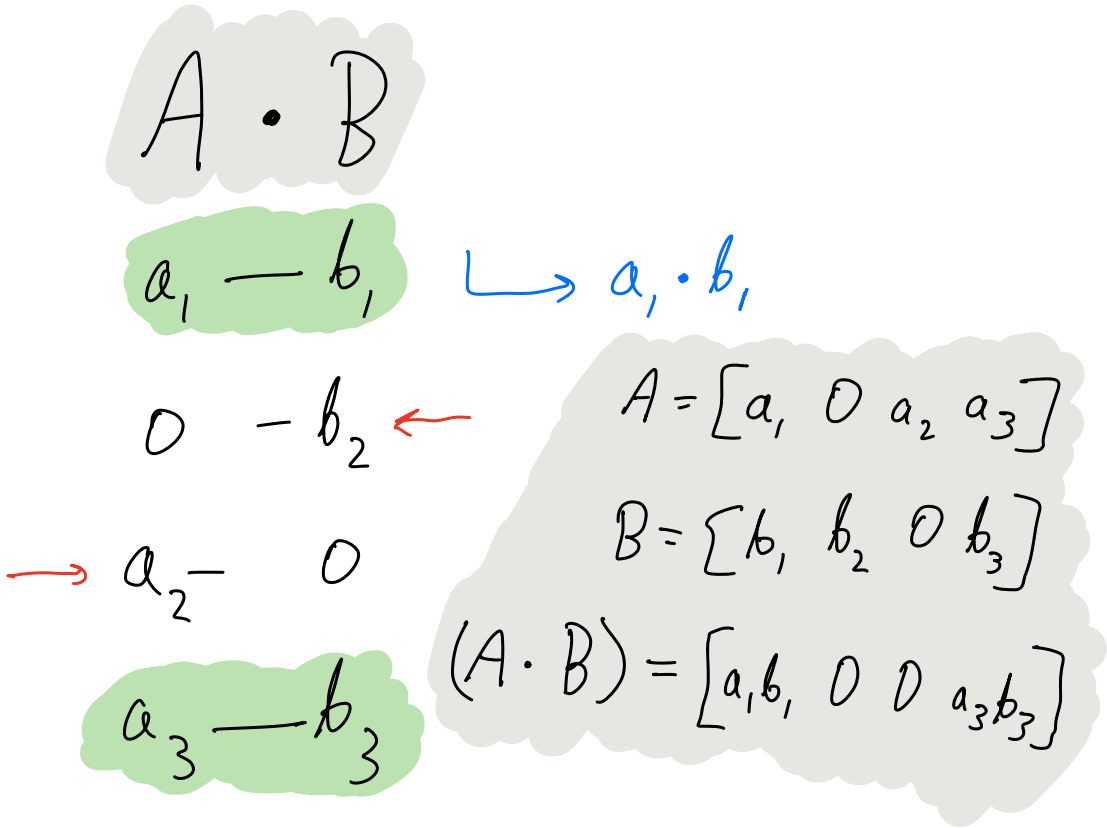
$$S \cdot f(T)$$



Key idea: *"sparse co-iteration"*

The sum or product of two sparse tensors (e.g. $A \cdot B$) can be computed *one coordinate at a time* as long as A and B can be computed one coordinate a time

Key idea: *"sparse co-iteration"*



Contribution: Indexed Streams

It appears that we have lost compositionality in the compiler (the code needed to compute a value may depend on its entire future).

Contribution: Indexed Streams

It appears that we have lost compositionality in the compiler (the code needed to compute a value may depend on its entire future).

Do we need a whole host of proofs relating various chunks of imperative code related by algebraic equivalences?

Contribution: Indexed Streams

It appears that we have lost compositionality in the compiler (the code needed to compute a value may depend on its entire future).

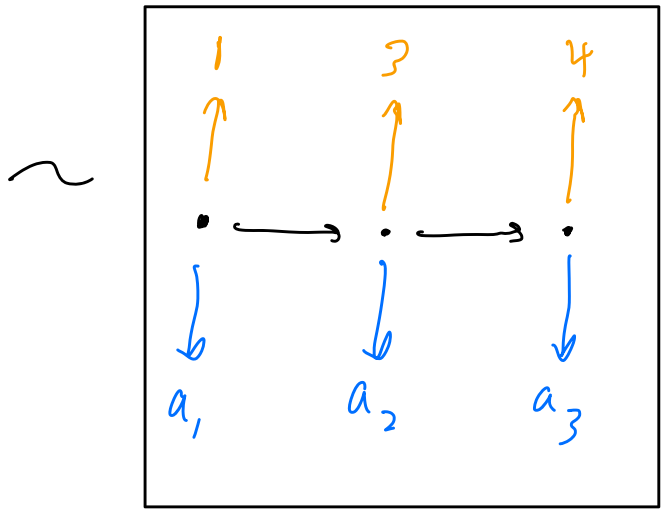
Do we need a whole host of proofs relating various chunks of imperative code related by algebraic equivalences?

No! We regain compositionality by implementing each operator as a function of *indexed streams*.

Indexed Streams

Instead of directly generating code to compute a given sub expression, we generate an *abstract machine* which in principle could generate that subexpression's non-zero entries one-by-one.

$$A = [a_1\ 0\ a_2\ a_3]$$



Technical Details

An indexed stream is a deterministic automaton (with state space Σ) that also produces at most one non-zero (coordinate,value) pair

$$q \mapsto (i, v) \in I \times V$$

per state $q \in \Sigma$, in coordinate order.

Technical Details

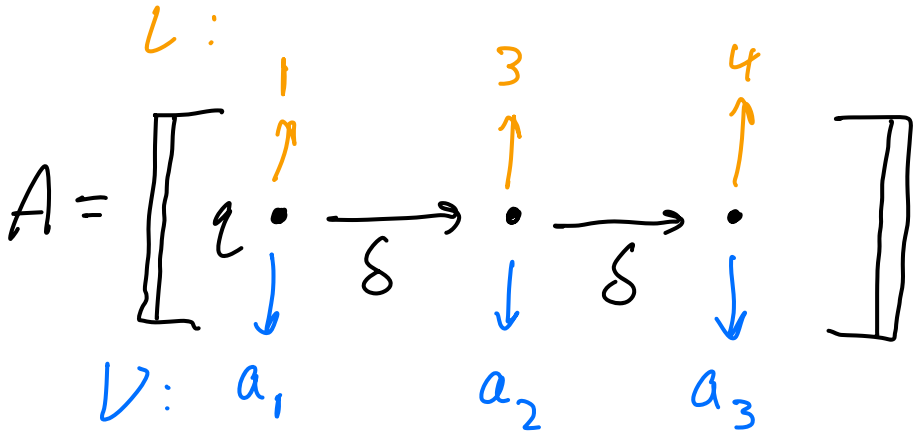
An indexed stream is a deterministic automaton (with state space Σ) that also produces at most one non-zero (coordinate,value) pair

$$q \mapsto (i, v) \in I \times V$$

per state $q \in \Sigma$, in coordinate order.

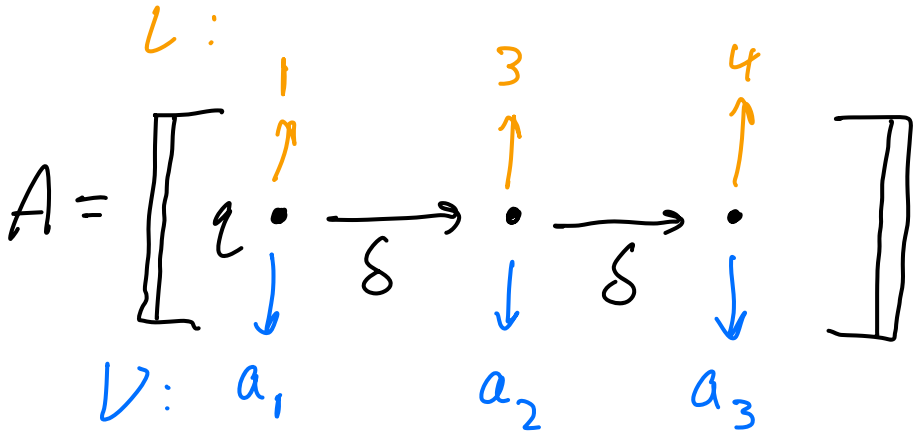
$$\begin{aligned} \text{initial state: } & q : \Sigma \\ \text{transition function: } & \delta : \Sigma \to \Sigma \\ \text{index function: } & \iota : \Sigma \to I \\ \text{value function: } & \nu : \Sigma \to V \cup \{\text{none}\} \end{aligned}$$

Stream Semantics



$$= [a_1, 0, a_2, a_3]$$

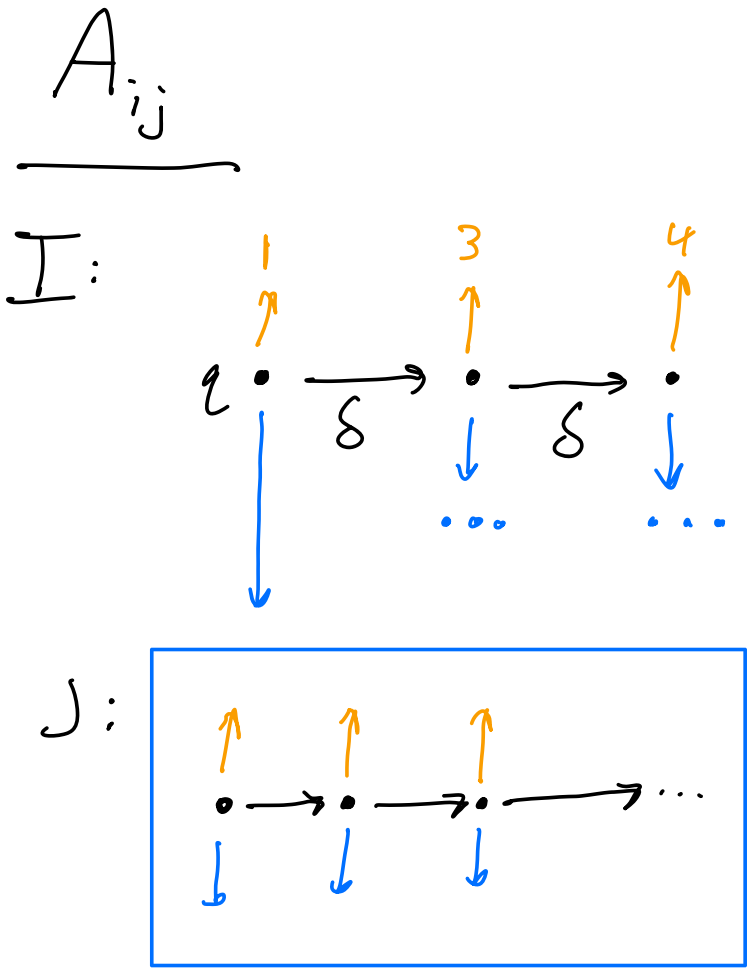
Stream Semantics



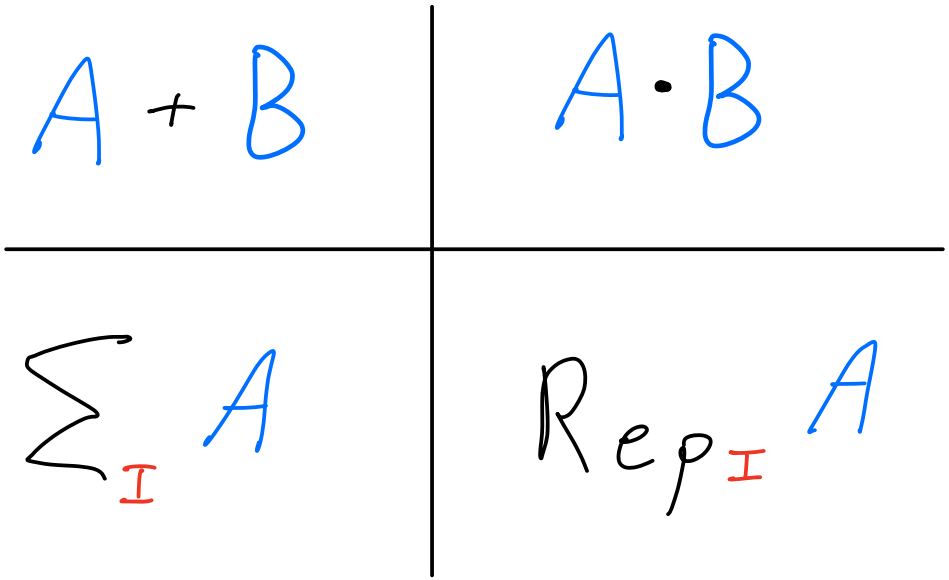
$$= [a_1, 0, a_2, a_3]$$

$$[\![q]\!] = \sum_{s=\delta^i(q)} (\iota(s) \mapsto \nu(s))$$

Tensors with Multiple Axes



Tensor Constructors

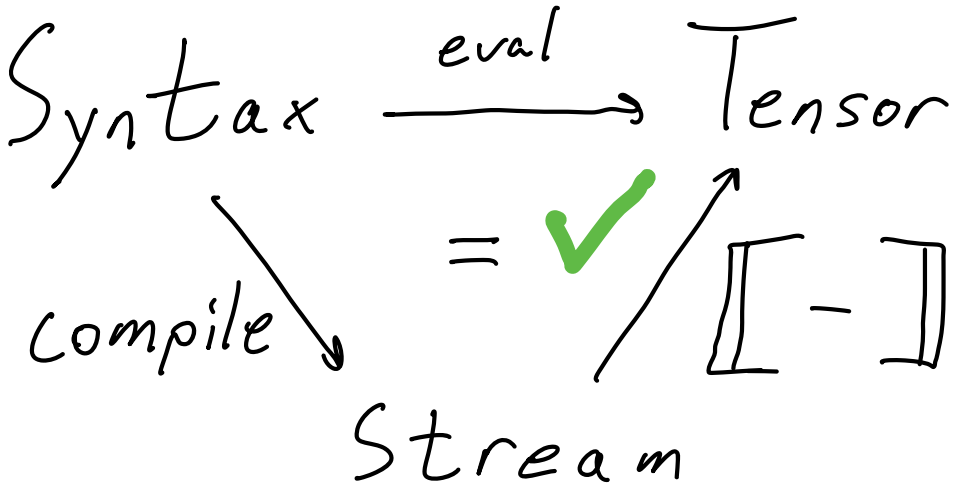


- Each one can be implemented with an operation on indexed streams.
- This approach enforces fusion automatically.

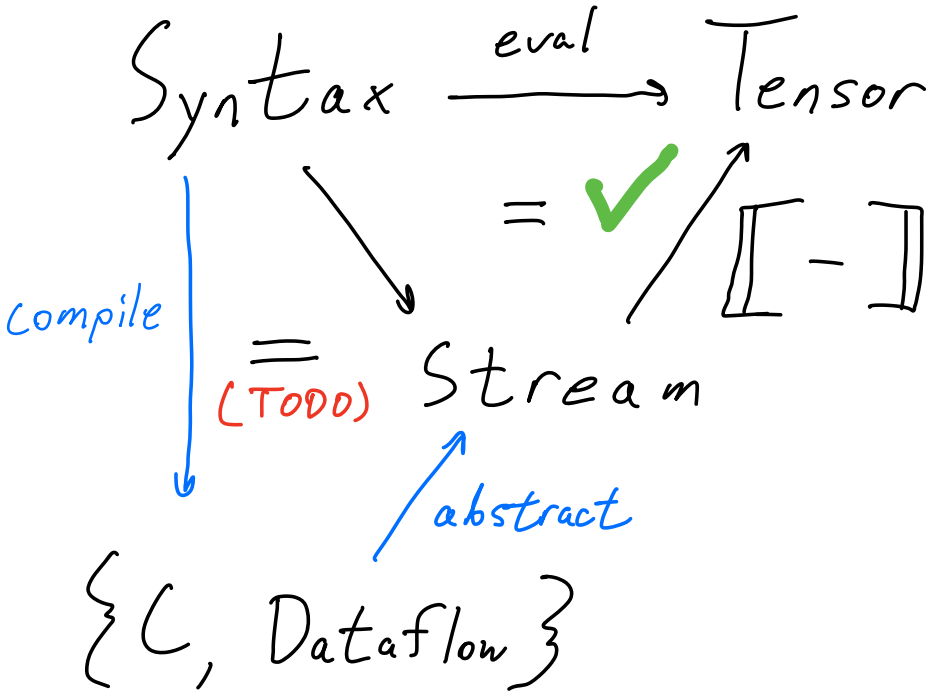
Tensor Constructors

We showed that each stream operation correctly implements the corresponding mathematical operation on the meanings of indexed streams

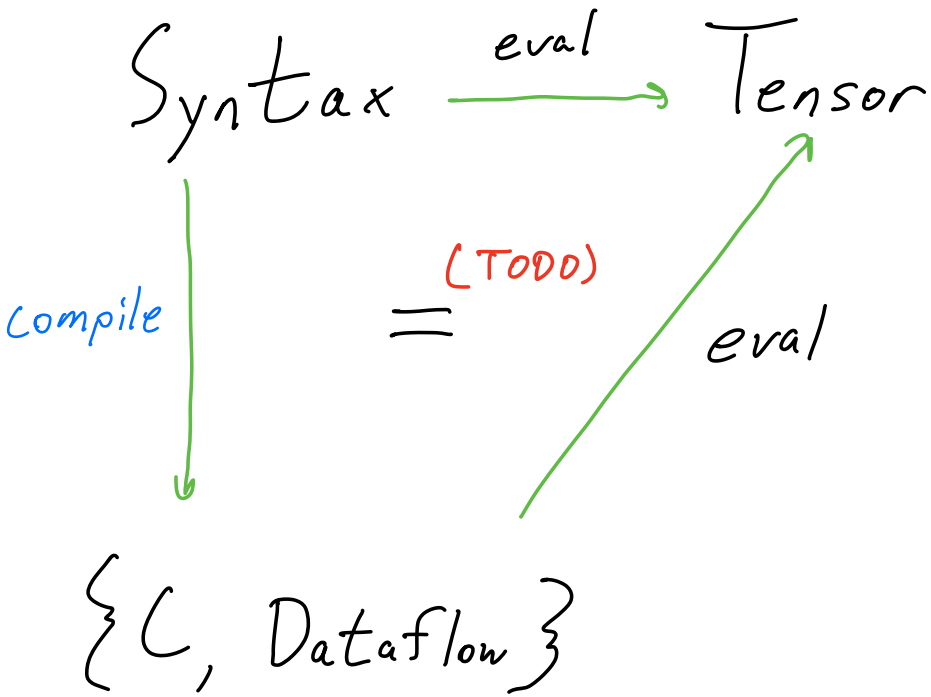
- e.g. $[\![A \cdot B]\!] = [\![A]\!] \cdot [\![B]\!]$



On-going work: Simplified Compilation

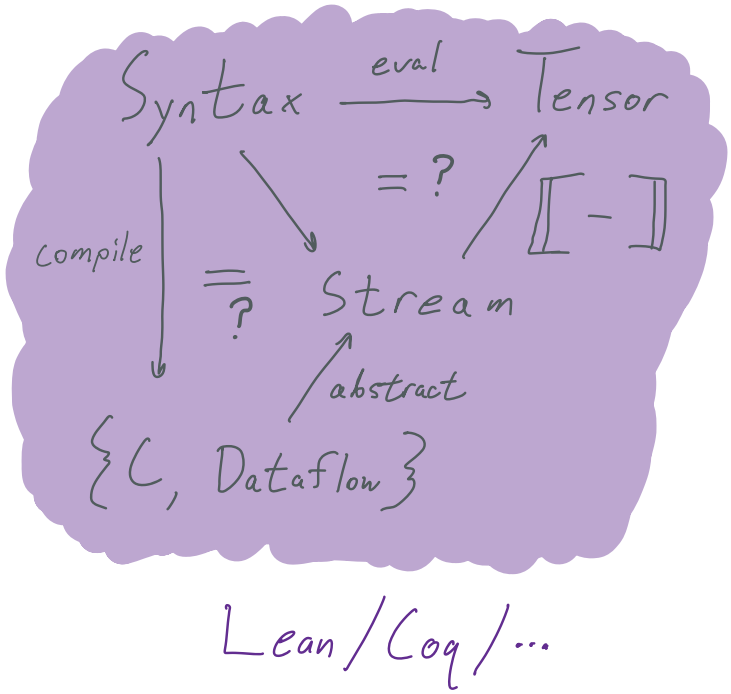


On-going work: Simplified Compilation



On-going work: Formalization in an interactive proof-assistant

Higher-order logic can express arbitrary mathematical properties, program semantics, and hardware semantics.

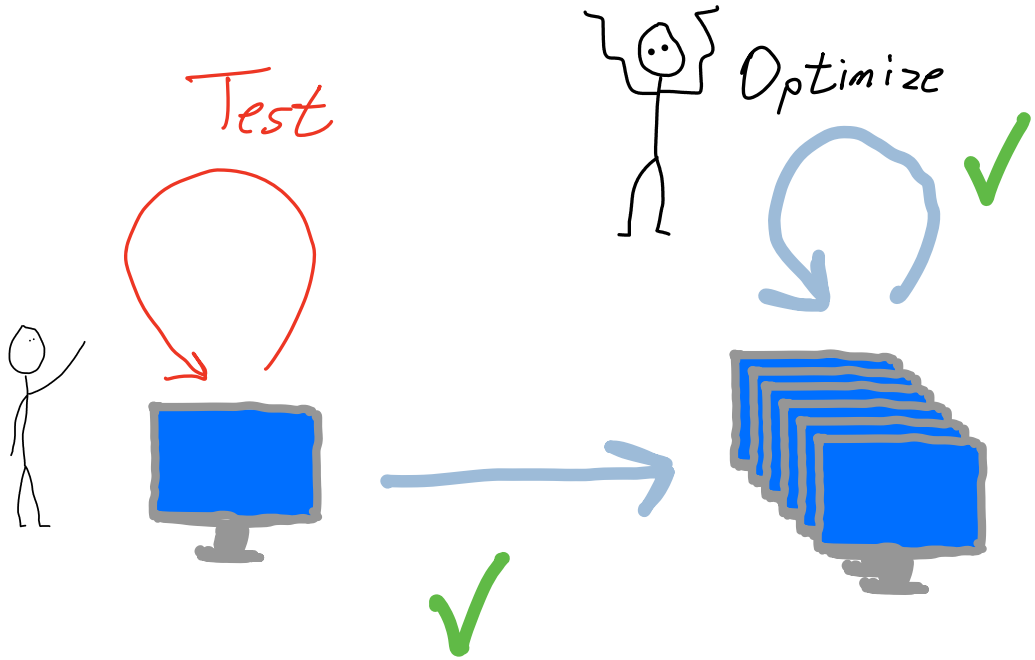


The designers of new hardware accelerators (and compilers to target them) stand to benefit in productivity!

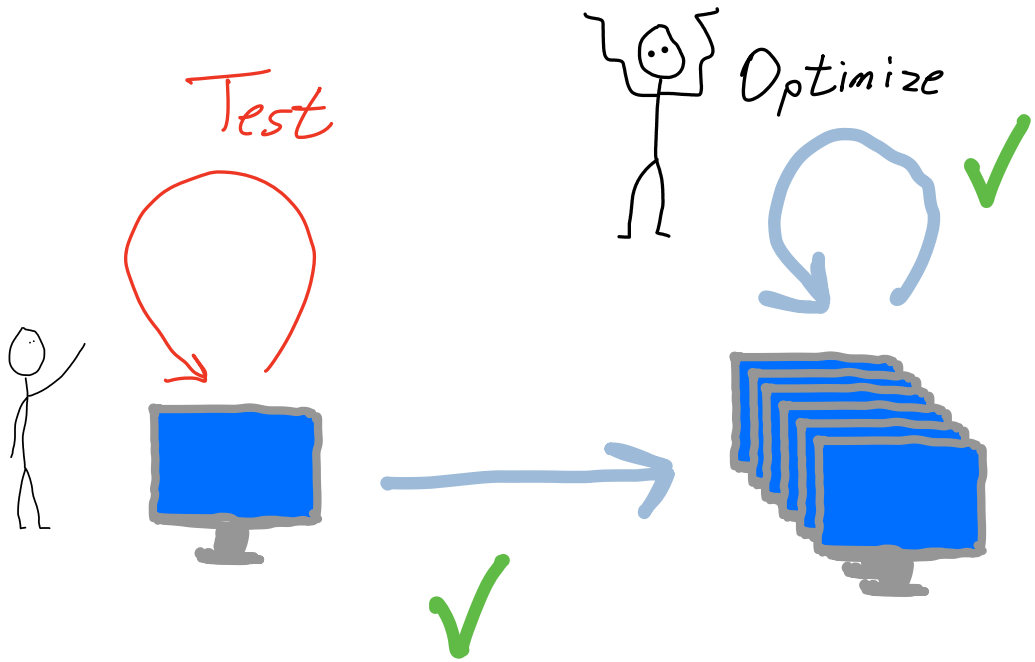
The designers of new hardware accelerators (and compilers to target them) stand to benefit in productivity!

- Theorem proving languages provide especially robust tools for embedding domain-specific languages.
- Higher-order logic makes it straightforward to introduce algebraic structures and write generic proofs.

Moreover, end-users of a verified compiler never need to worry about the changelog, as long as the correctness proof still stands.

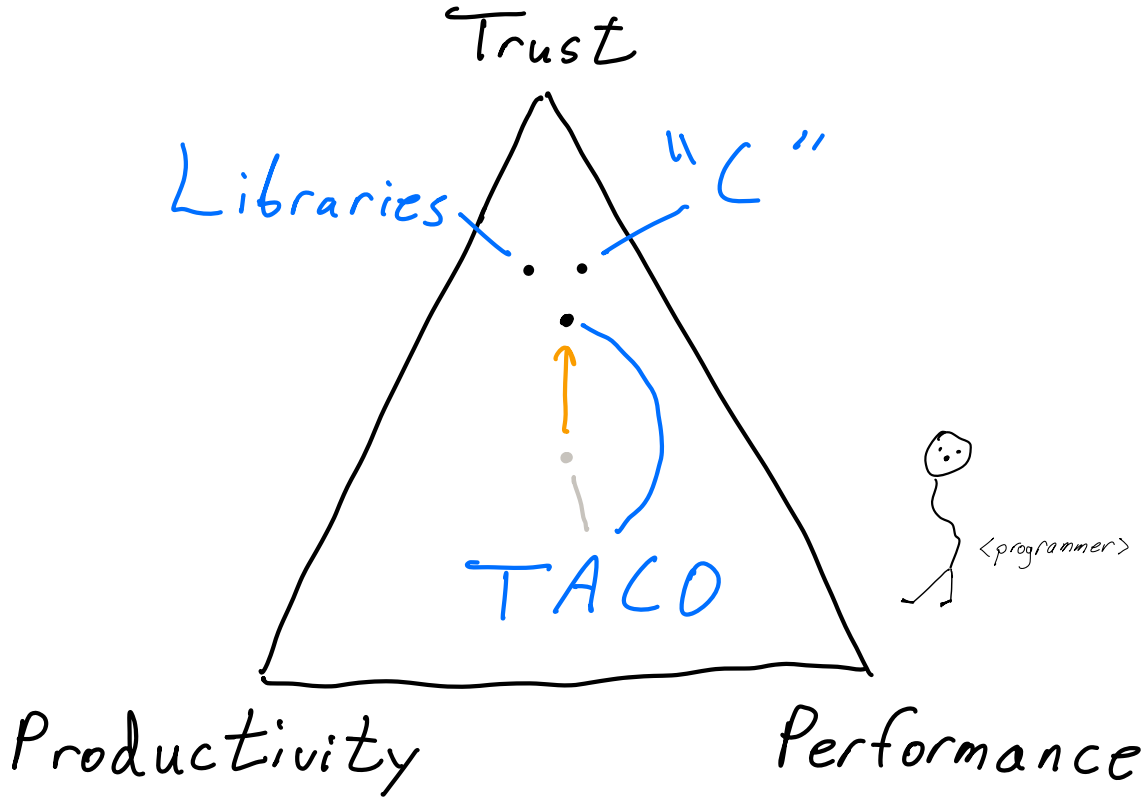


Moreover, end-users of a verified compiler never need to worry about the changelog, as long as the correctness proof still stands.



We see this as a step toward robust, shared libraries of knowledge and optimization procedures [cf FIAT].

Conclusion


Trust
Libraries
"C"
TACO
<programmer>
Productivity
Performance