



# Specifying and Verifying Sparse Matrix Codes

Gilad Arnold and Ras Bodík, UC Berkeley

Johannes Hölzl, TU München

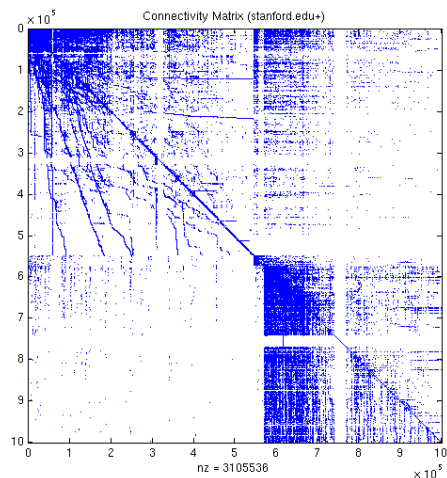
Ali Sinan Köksal, EPFL

Mooly Sagiv, Tel-Aviv University

# Why sparse matrices?

Heavy use in physics, civil/mechanical engineering simulations

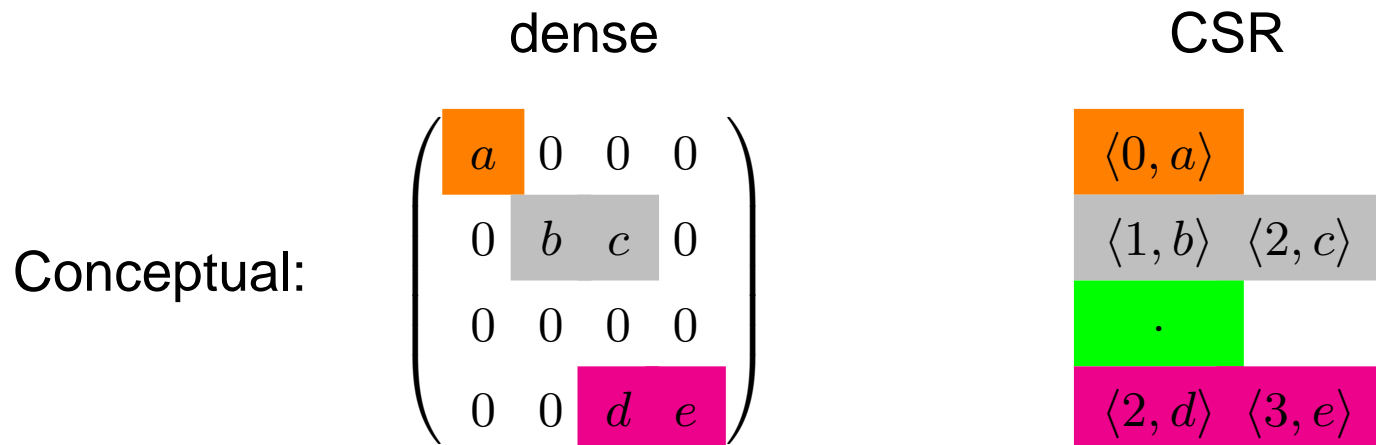
- more recently: various micro-/macro-scale network modeling
- increasingly pervasive in daily used applications



Large variety of formats being implemented

- emphasis on performance
- productivity + correctness becoming an issue—an opportunity?

# Sparse formats and their implementation



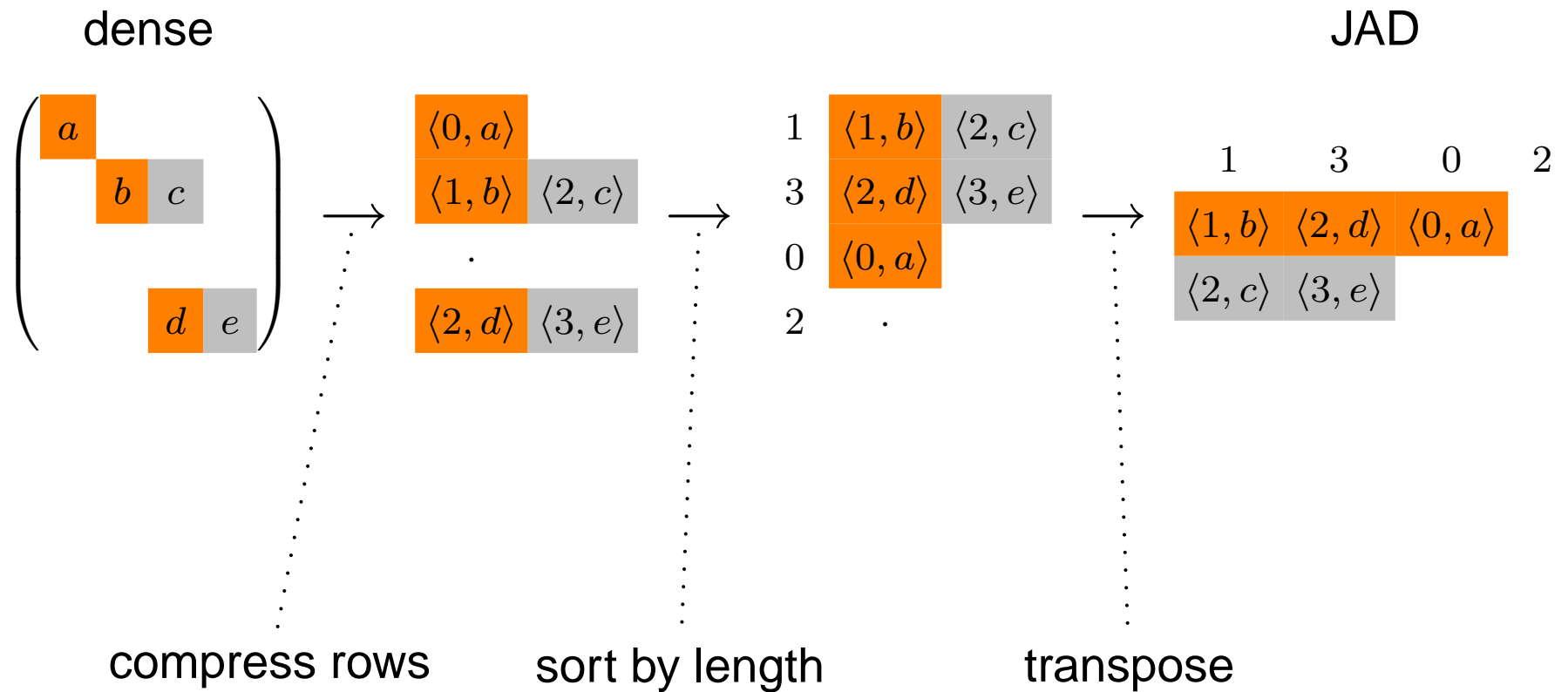
In practice:

```
/* CSR construction. */
for (i = 0, k = 0; i < m; i++) {
    R[i] = k;
    for (j = 0; j < n; j++)
        if (A[i][j] != 0)
            C[k] = j, V[k] = A[i][j], k++;
    R[i] = k;
}
```

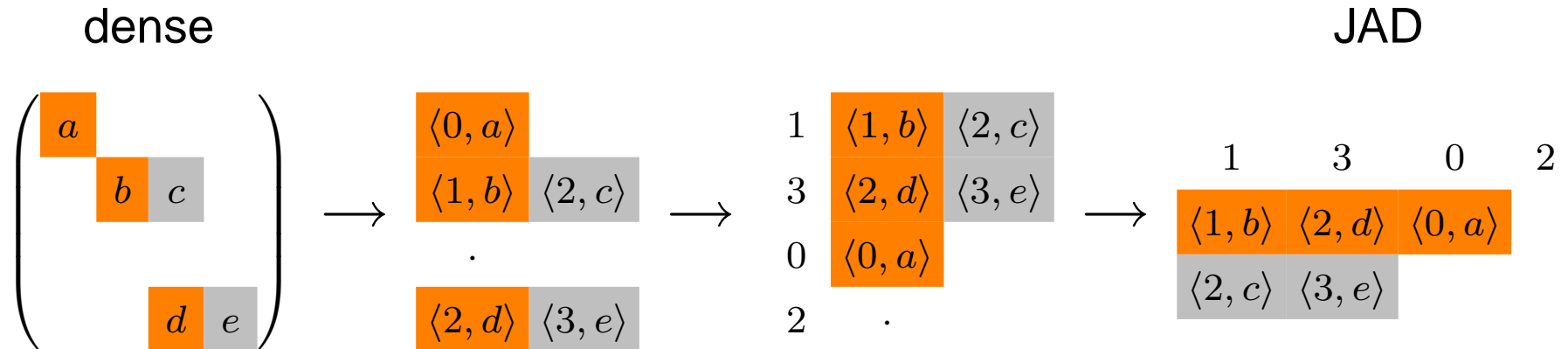
R [ 0 1 3 3 5 ]  
C [ 0 1 2 2 3 ]  
V [ a b c d e ]

Low-level, imperative implementation is challenging—why?

# Challenge: programming sparse formats



# Challenge: programming sparse formats



Currently: hand-crafted “flat” representation, obscured dataflow

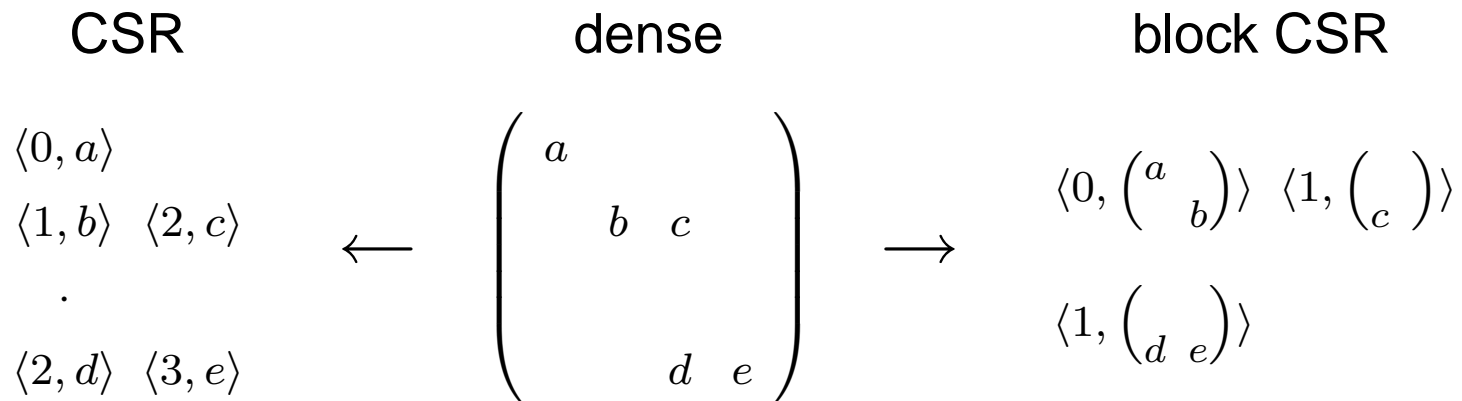
```

P [ 1 3 0 2 ]
D [ 0 3 5 ]
C [ 1 2 0 2 3 ]
V [ b d a c e ]

nzlenperm (P, A);
for (d = k = 0; d < n; d++) {
  for (i = 0, kk = k; i < m; i++) {
    for (j = nz = 0; j < n; j++)
      if (A[P[i]][j])
        if (++nz > d) break;
    if (j < n) C[k] = j, V[k] = A[P[i]][j], k++; }
  if (k == kk) break;
  D[d] = k; }

```

# Challenge: programming sparse formats (cont)



Currently: low-level code crafting allows little reuse

Instead we want:

- high-level datatypes: lists, pairs
  - operations on them: map, filter, block, transpose, ...
  - simple dataflow
- } composable  
} reusable

# Challenge: verifying sparse formats

```
/* CSR construction. */
csr_t csr(int *A) {
  for (i = 0, k = 0; i < m; i++) {
    R[i] = k;
    for (j = 0; j < n; j++)
      if (A[i][j] != 0)
        C[k] = j, V[k] = A[i][j], k++; }
  R[i] = k;
  /* return <R,C,V> */ }

```

```
/* CSR matrix-vector multiply. */
double *csr_mv(csr_t B, int *x) {
  /* R,C,V <- B */
  k = R[0];
  for (i = 0; i < m; i++) {
    y[i] = 0;
    for ( ; k < R[i + 1]; k++)
      y[i] += V[k] * x[C[k]]; } }

```

To prove:  $\forall A, x . \text{csr\_mv}(\text{csr}(A), x) = A \cdot x$

- failed on imperative programs: complex invariants, abstraction
- desired property profoundly hard

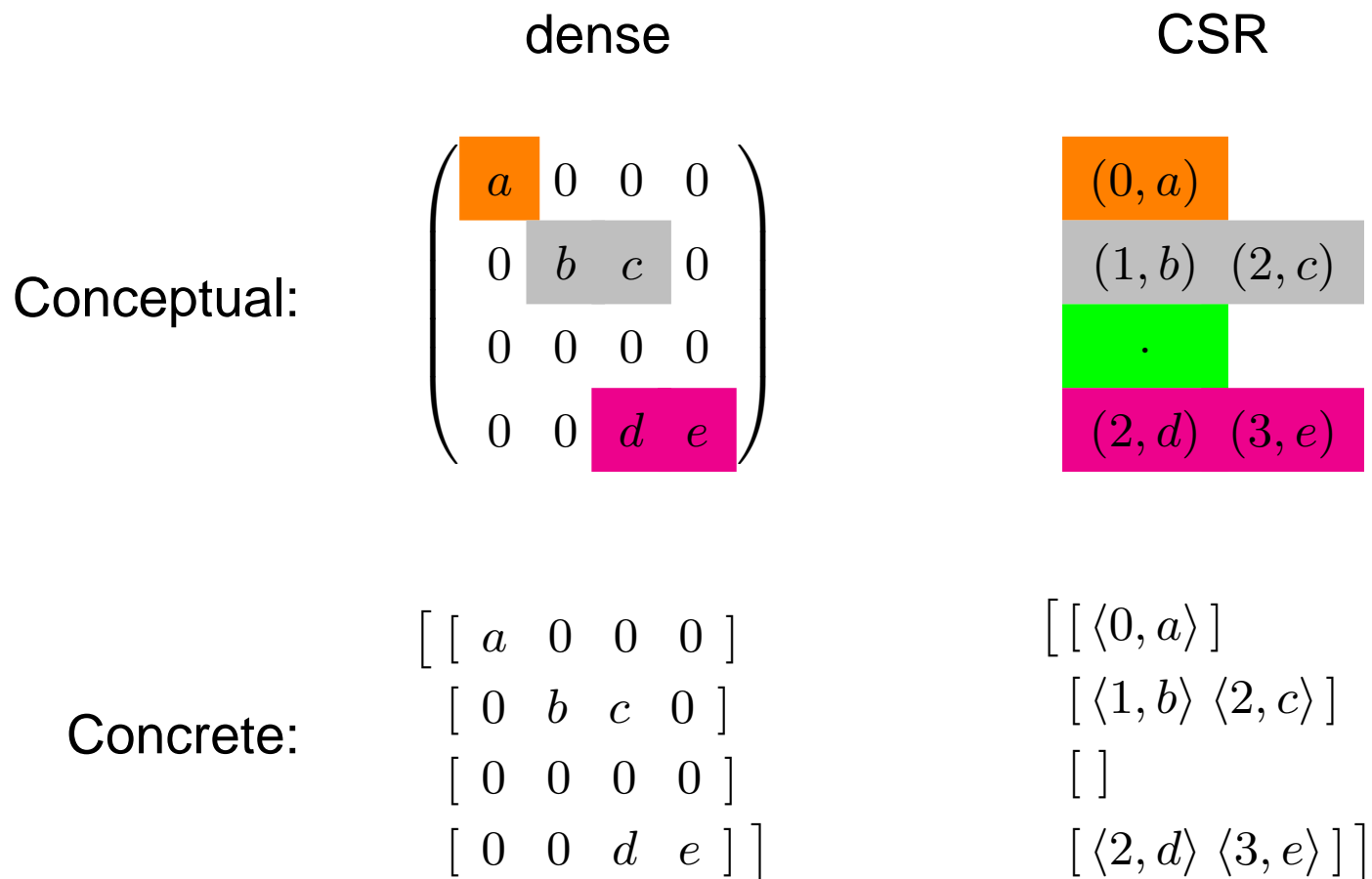
Instead we want:

- programs that are more restricted, structured
- powerful, automated prover

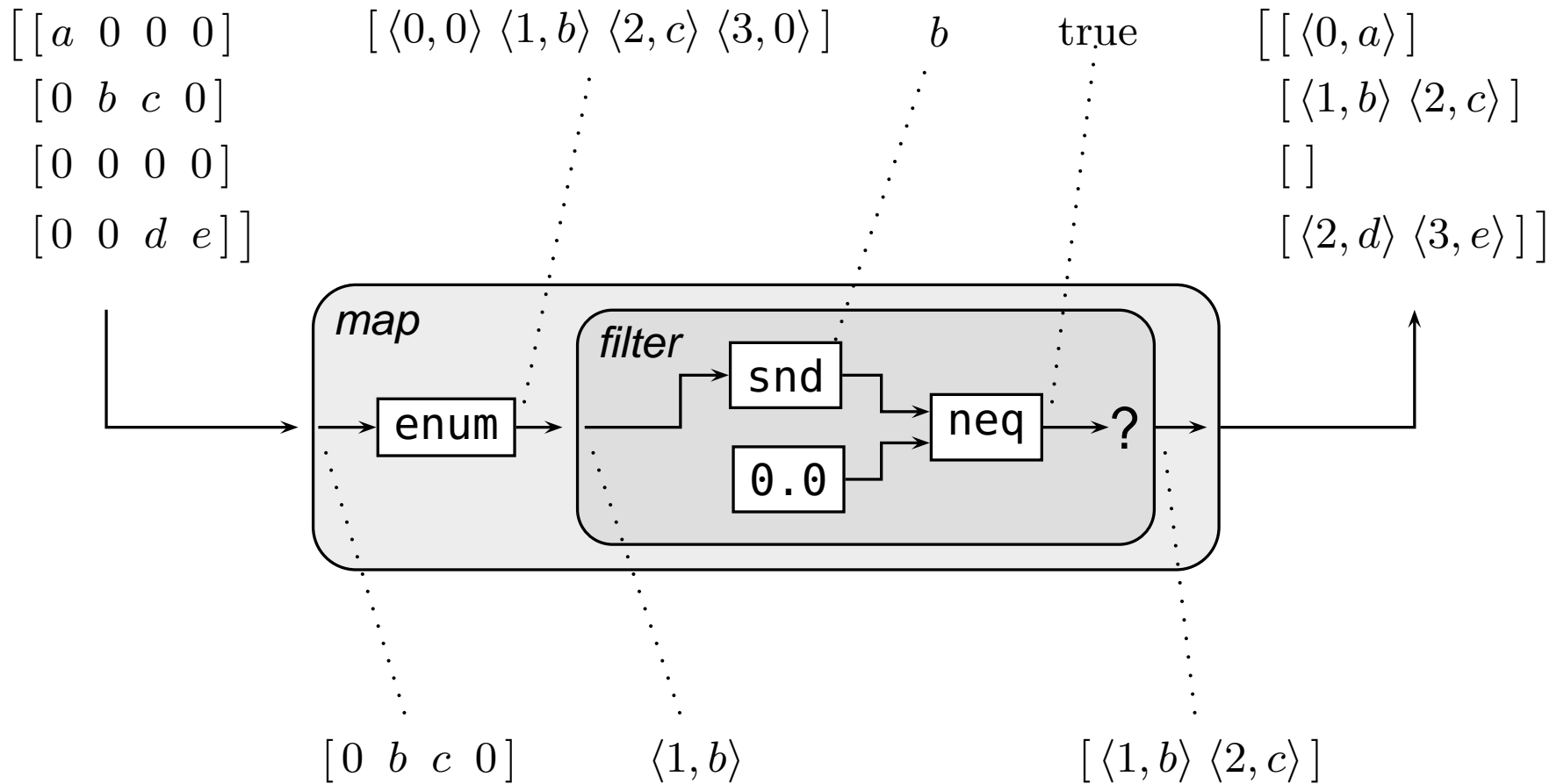
# This talk

- ① a “little language” (LL) for implementing sparse formats
  - reminiscent of FP, APL, NESL
  - demonstrating construction + SpMV of real-world formats
  
- ② full functional verification using Isabelle/HOL
  - first known result for complex sparse formats

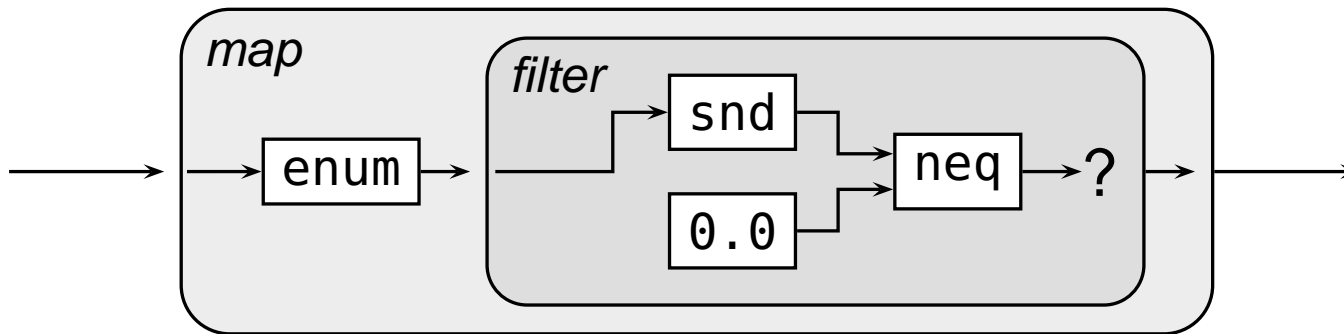
# Implementing sparse codes in LL



# CSR construction



# CSR construction

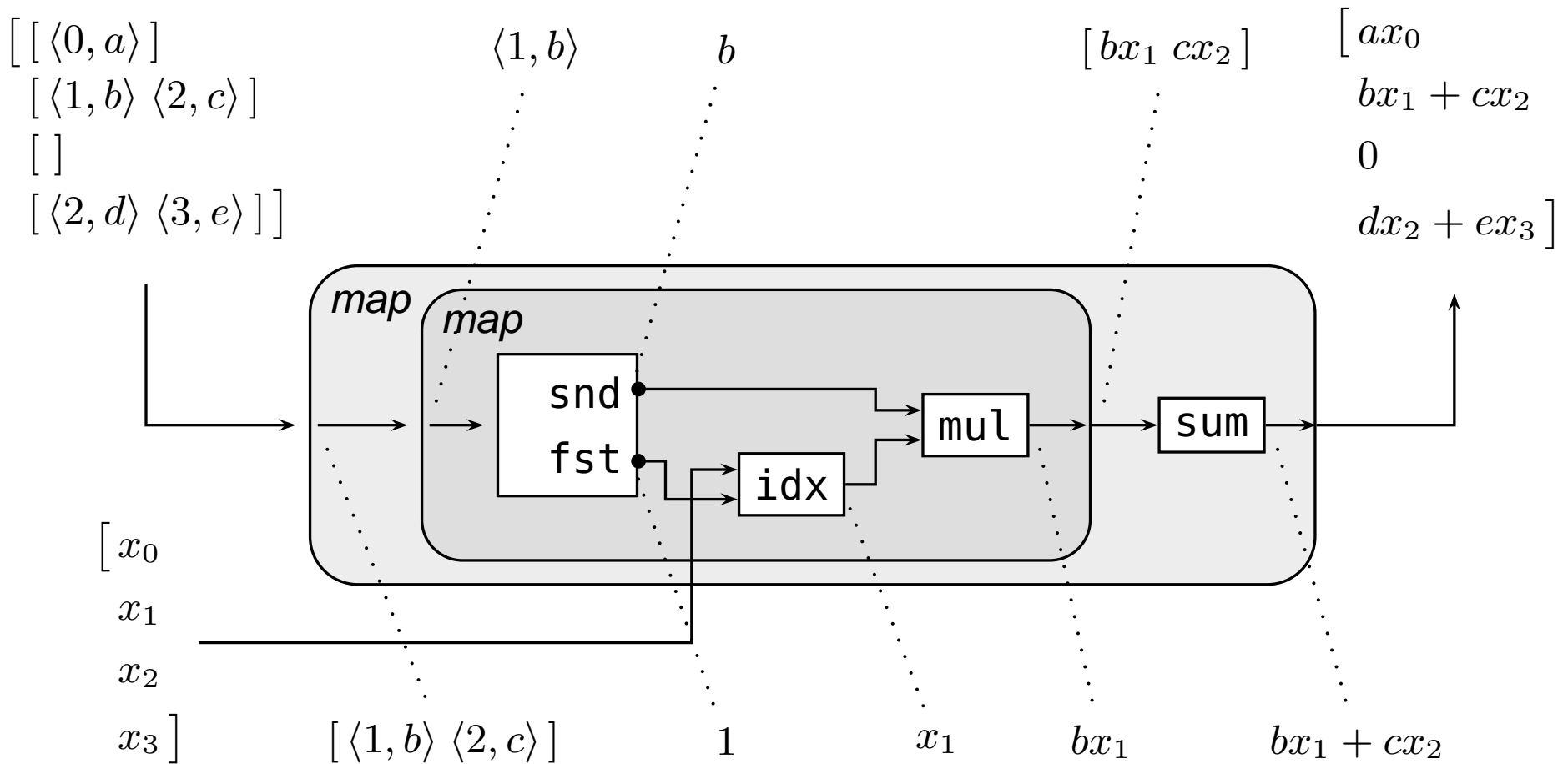


[enum -> [(snd, 0.0) -> neq ? ]]

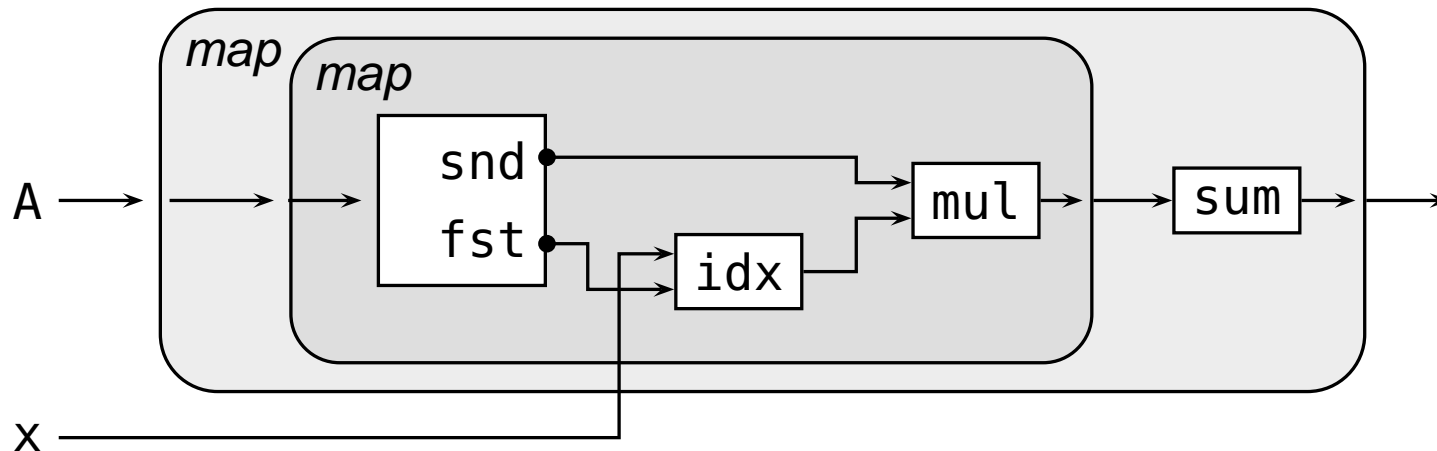
More convenient notation:

[enum -> [snd != 0.0 ? ]]

# CSR sparse matrix-vector multiply (SpMV)



# CSR sparse matrix-vector multiply (SpMV)



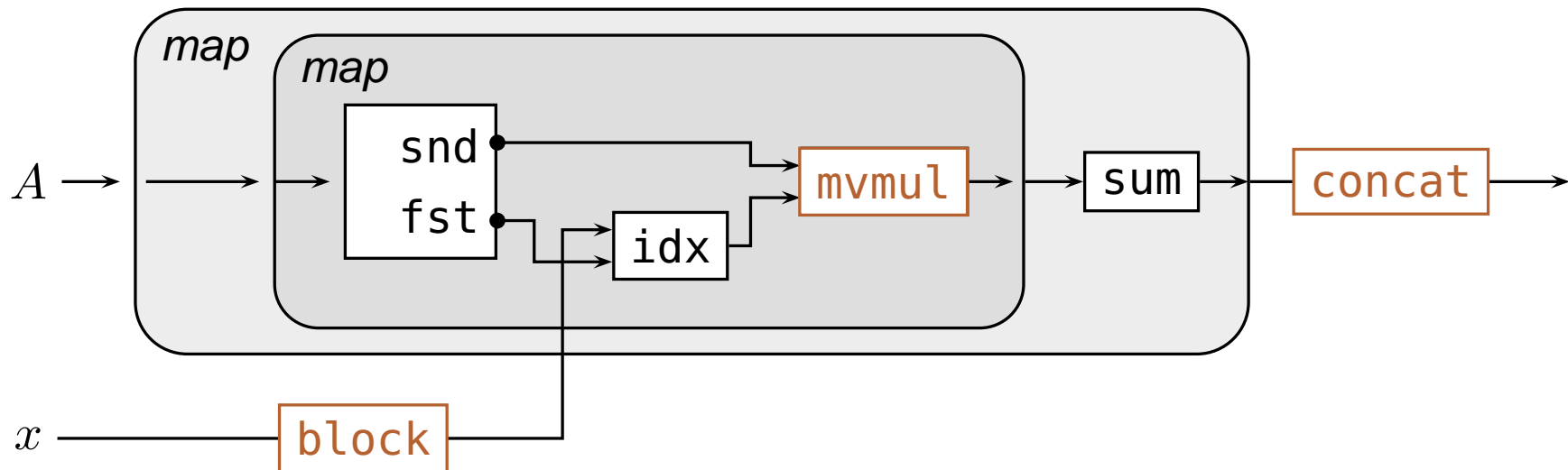
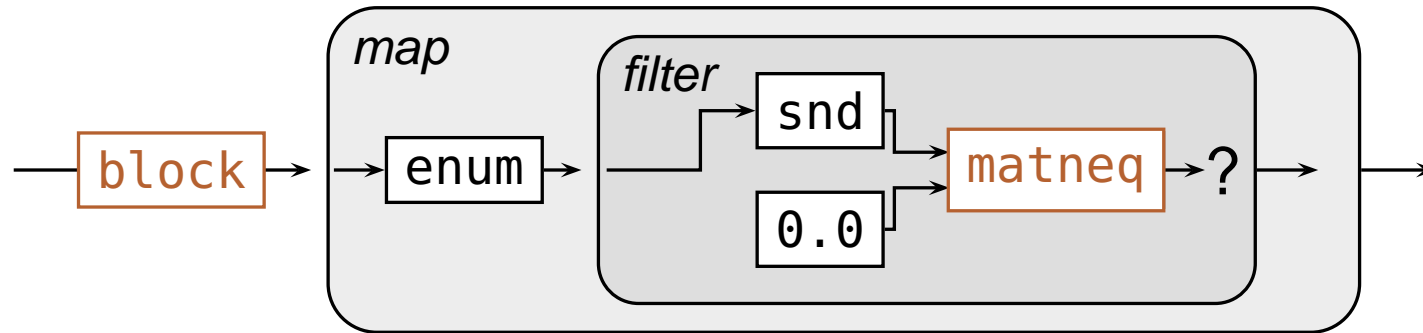
`A -> [[snd * x[fst]] -> sum]`

Python-style sugar:

```
[sum ([v * x[j] for j,v in Ai]) for Ai in A]
```

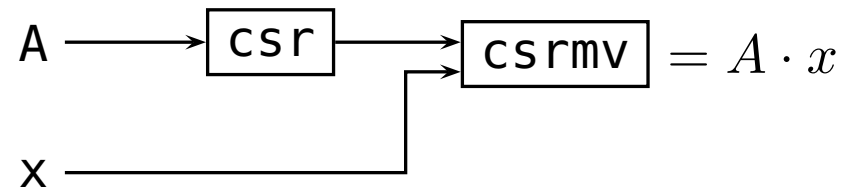
# Hierarchical compression: block CSR

→ easy to express? easily derived from ordinary CSR?



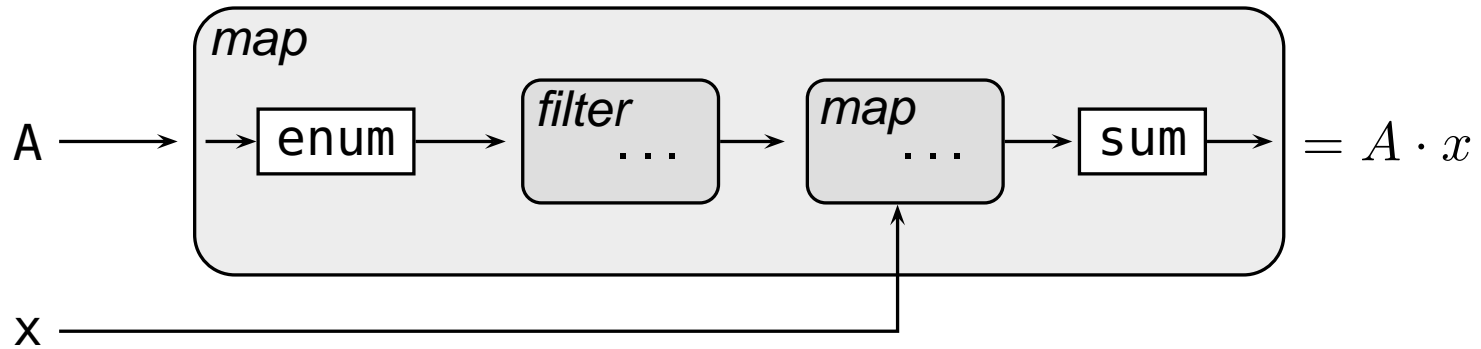
# Verifying sparse codes with Isabelle/HOL

- Isabelle: LCF-style theorem prover for higher-order logic
  - **shallow embedding**: LL translated to typed  $\lambda$ -calculus
- verification goal: for all  $A$  and  $x$





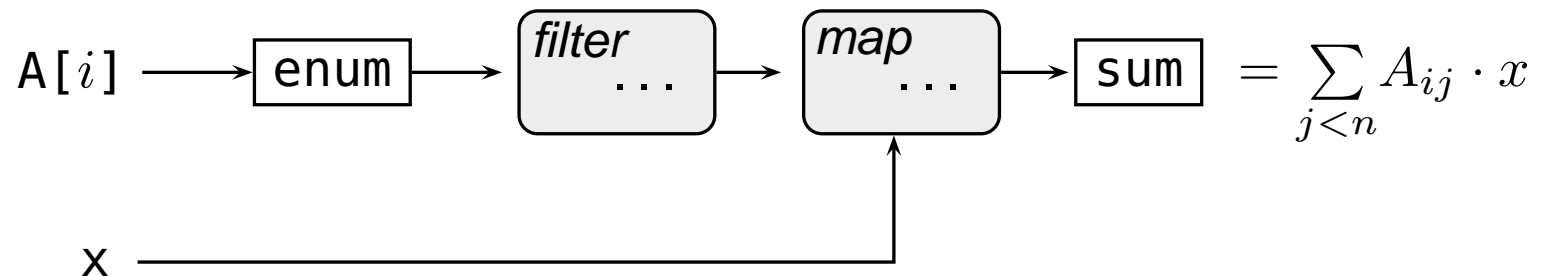
# Base technique #2: introduction



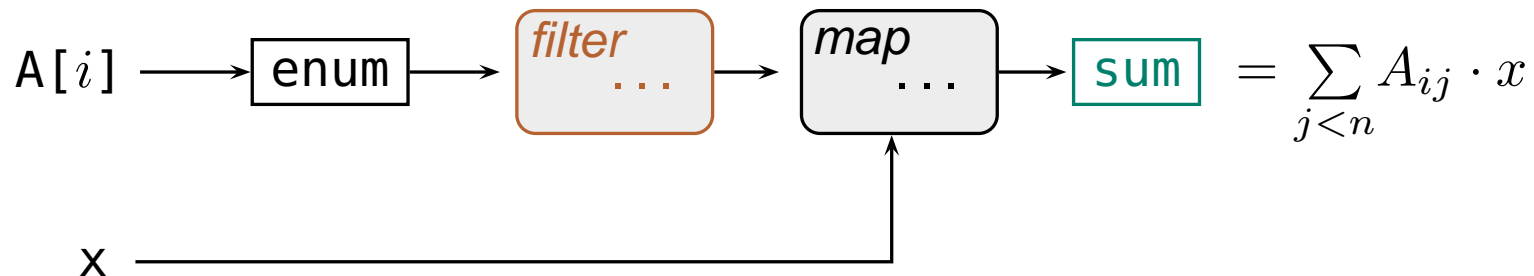
→ following rule peels off outermost map

$$\frac{\forall i < k . y[i] \rightarrow f = z_i}{y \rightarrow [f] = z}$$

→ new subgoal: for all  $i < m$



# Verification pitfalls

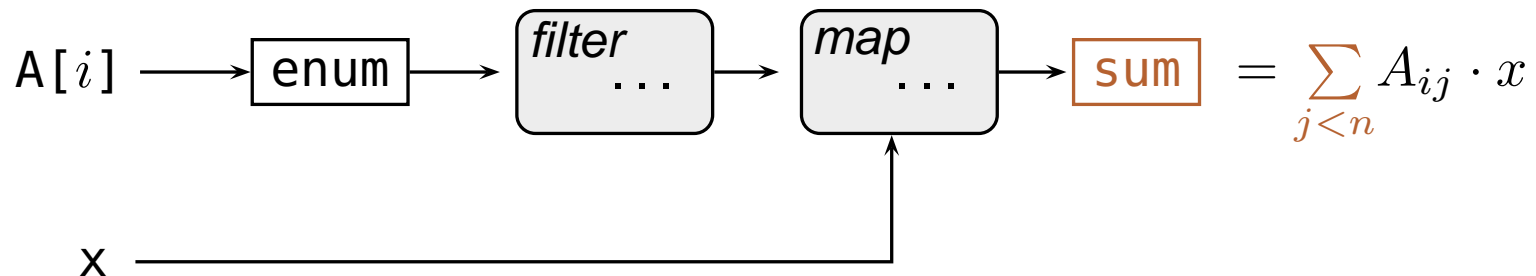


- observation: **filtered zeros** do not affect the result of **sum**
- naïve solution: add a simplification rule

$$\frac{\forall y . \neg p(y) \longrightarrow f(y) = 0}{[p \ ? \ ] \ -> [f] \ -> \text{sum} = [f] \ -> \text{sum}}$$

- filter-map-sum pattern useful for CSR but not other formats

## Verification pitfalls (cont)



- observation: remove **summation**, reason about list of values
  - arbitrarily permuted
  - nonzeros represented exactly once
  - may contain zeros } nasty introduction rule!
- explicit representation invariants in proof rules—a bad idea

# Vector/matrix representation predicates

- ① encapsulate representation invariants  $\implies$  “abstraction layer”

notation

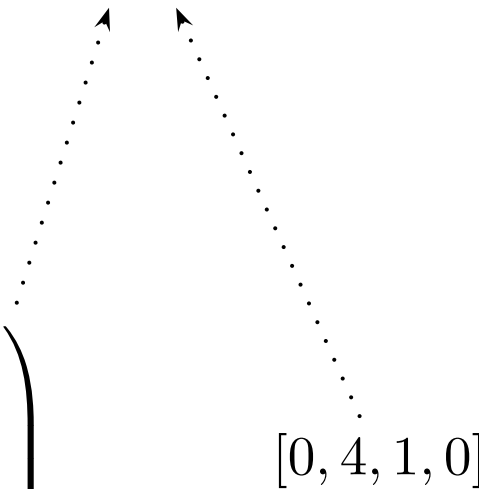
meaning: value  $x$  represents  $y$  as...

$ilist (op =) y x$

an indexed list

$$\begin{pmatrix} 0 \\ 4 \\ 1 \\ 0 \end{pmatrix}$$

$[0, 4, 1, 0]$



# Vector/matrix representation predicates

① encapsulate representation invariants  $\implies$  “abstraction layer”

notation	meaning: value $x$ represents $y$ as...
----------	---

---

$ilist (op =) y x$	an indexed list
--------------------	-----------------

$alist (op =) y x$	a sparse index-value (associative) list
--------------------	---

$$\begin{pmatrix} 0 \\ 4 \\ 1 \\ 0 \end{pmatrix}$$

$[\langle 2, 1 \rangle, \langle 1, 4 \rangle, \langle 0, 0 \rangle]$

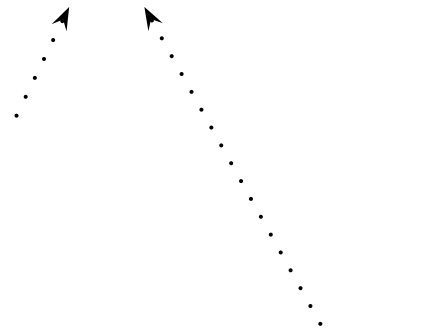
# Vector/matrix representation predicates

① encapsulate representation invariants  $\implies$  “abstraction layer”

notation	meaning: value $x$ represents $y$ as...
$ilist (op =) y x$	an indexed list
$alist (op =) y x$	a sparse index-value (associative) list
$vlist (op =) y x$	a sparse list of values

$$\begin{pmatrix} 0 \\ 4 \\ 1 \\ 0 \end{pmatrix}$$

[1, 0, 4]



# Vector/matrix representation predicates

① encapsulate representation invariants  $\implies$  “abstraction layer”

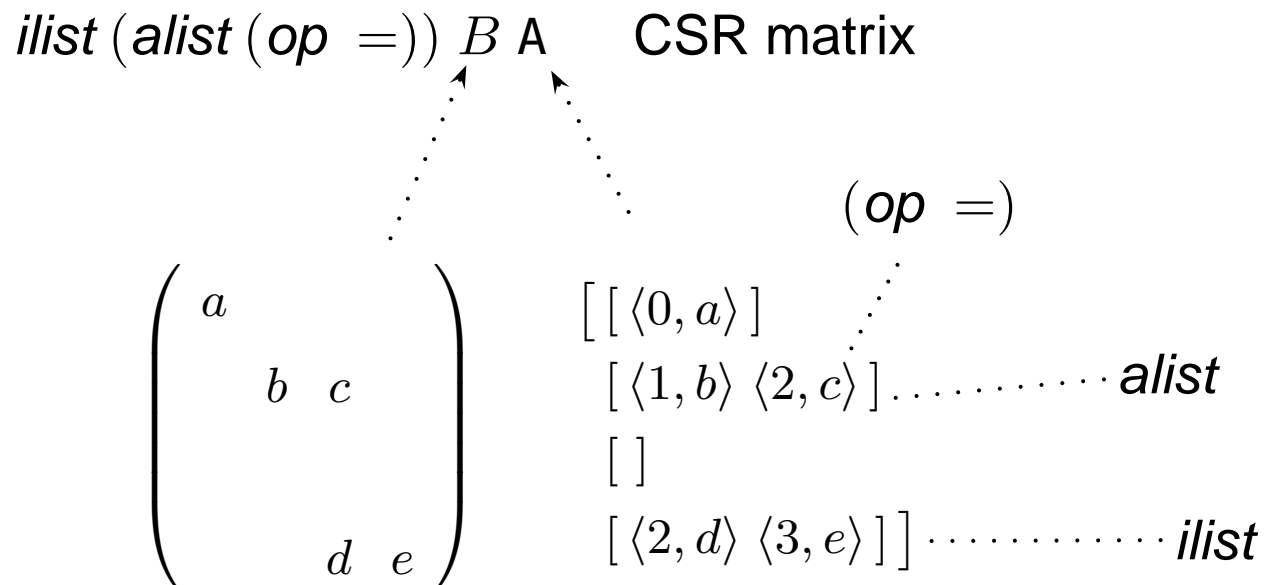
notation	meaning: value $x$ represents $y$ as...
$ilist$ $(op =)$ $y$ $x$	an indexed list
$alist$ $(op =)$ $y$ $x$	a sparse index-value (associative) list
$vlist$ $(op =)$ $y$ $x$	a sparse list of values

# Vector/matrix representation predicates

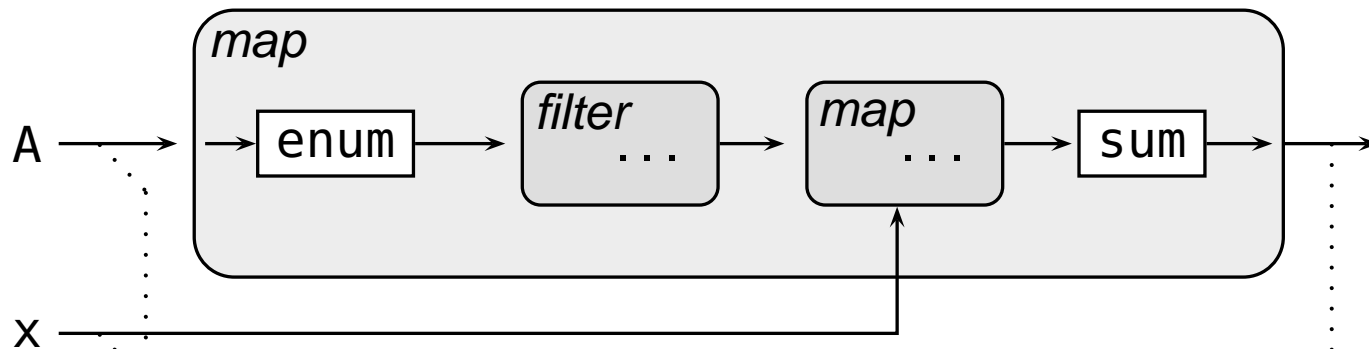
① **encapsulate representation invariants**  $\implies$  “abstraction layer”

notation	meaning: value $x$ represents $y$ as...
$ilist (op =) y x$	an indexed list
$alist (op =) y x$	a sparse index-value (associative) list
$vlist (op =) y x$	a sparse list of values

② **parametric, composable**  $\implies$  reusability



# Using predicates through proofs



assumptions:

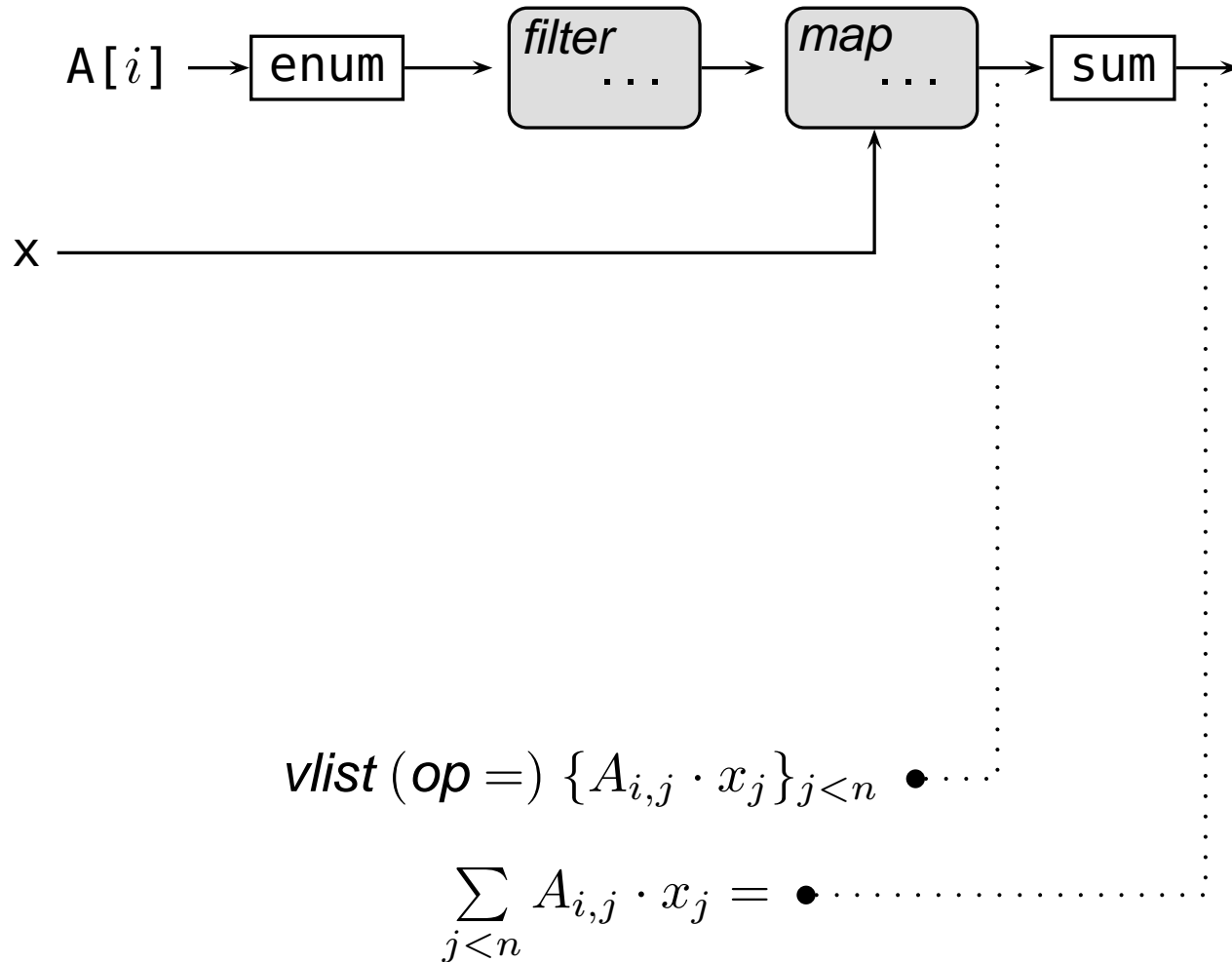
$\dots$   $ilist (ilist (op =)) A A$

$ilist (op =) x x$

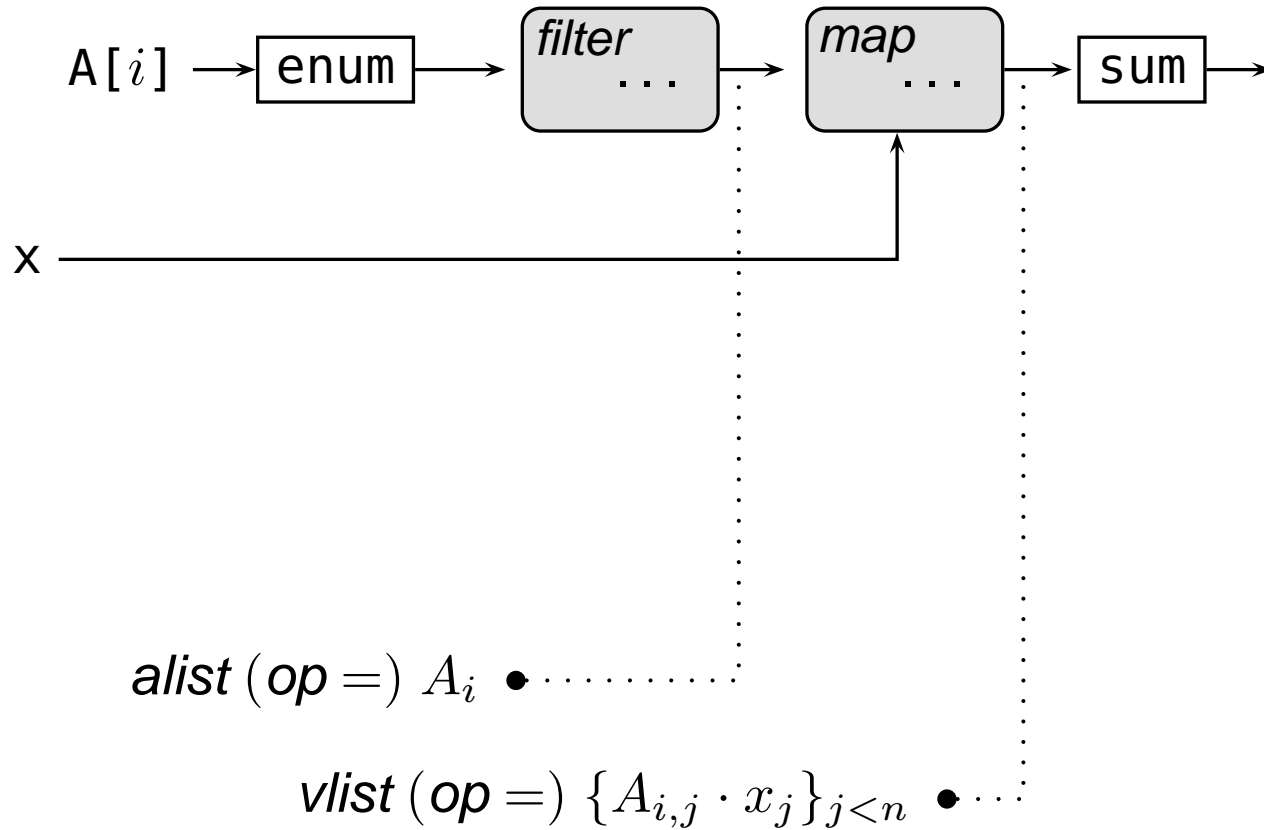
goal:

$ilist (op =) (A \cdot x) (A \rightarrow [ \dots ])$

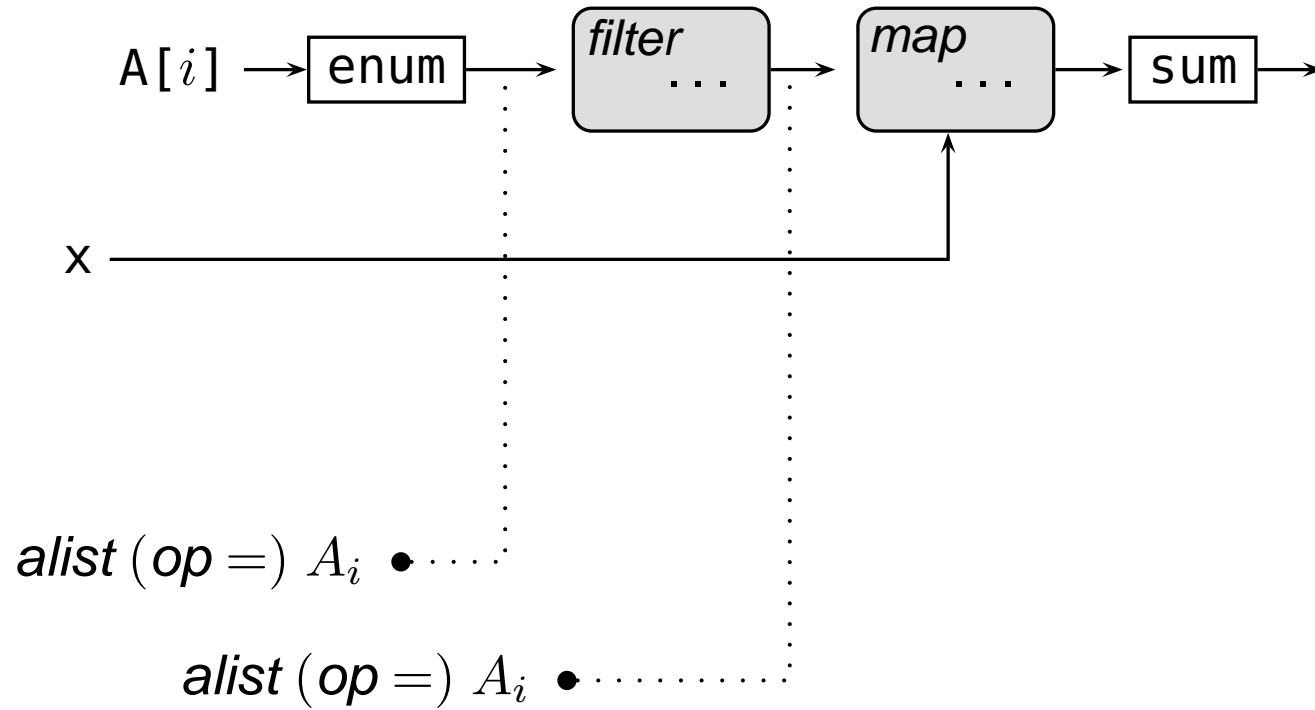
# Using predicates through proofs



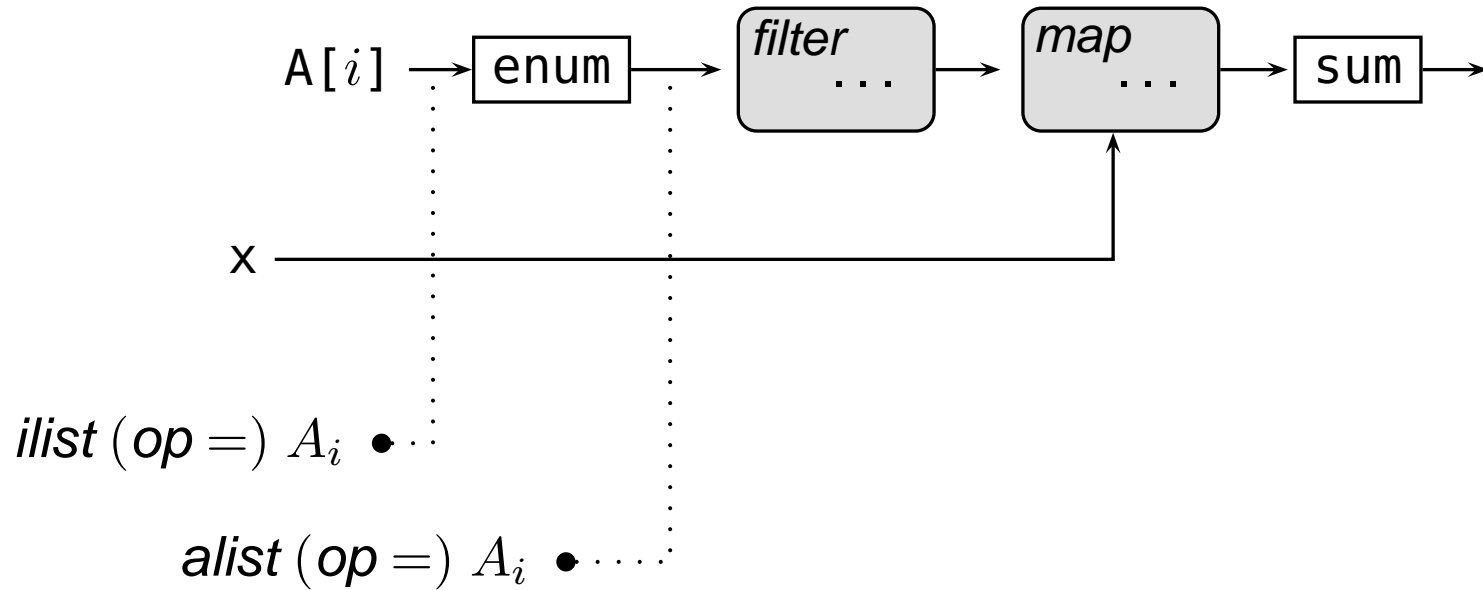
# Using predicates through proofs



# Using predicates through proofs



# Using predicates through proofs



# Evaluation

- implemented numerous real-world formats + SpMV in LL
  - row/column compression, coordinate, JAD
  - hierarchical compression: SCSR, RB-CSR, CB-CSR
  - succinct but not cryptic: 1–3 LOC
- proof theory developed with good level of reuse
  - 19% of proof rules used are format-specific
  - 66% used with 4 or more formats
- incremental extensions for hierarchical formats

# Conclusion

- small but adequate language + powerful verifier
  - extensions for supporting more operations:  $A^k x$ , LU
- compiler is work in progress
- roadmap
  - facilitate synthesis of sparse codes
  - application to other domains: ciphers, networking, ...

Thank you!

Questions?

arnold@cs.berkeley.edu

hoelzl@in.tum.de