Specifying and Verifying Sparse Matrix Codes *

Gilad Arnold
University of California, Berkeley
arnold@cs.berkeley.edu

Johannes Hölzl
Technische Universität München
hoelzl@in.tum.de

Ali Sinan Köksal
École Polytechnique Fédérale de Lausanne
alisinan.koksal@epfl.ch

Rastislav Bodík
University of California, Berkeley
bodik@cs.berkeley.edu

Mooly Sagiv
Tel Aviv University †
msagiv@acm.org

Abstract

Sparse matrix formats are typically implemented with low-level imperative programs. The optimized nature of these implementations hides the structural organization of the sparse format and complicates its verification. We define a variable-free functional language (LL) in which even advanced formats can be expressed naturally, as a pipeline-style composition of smaller construction steps. We translate LL programs to Isabelle/HOL and describe a proof system based on parametric predicates for tracking relationship between mathematical vectors and their concrete representations. This proof theory automatically verifies full functional correctness of many formats. We show that it is reusable and extensible to hierarchical sparse formats.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical Verification

General Terms Languages, Verification

1. Introduction

Sparse matrix formats compress large matrices with a small number of nonzero elements into a more compact representation. The goal is to both reduce memory footprint and increase efficiency of operations such as sparse matrix-vector multiplication (SpMV). More than fifty formats have been developed; the reason for this diversity is that a format may improve memory locality in a given memory hierarchy, expose parallelism that fits the hardware, and tailor the layout to the operations that will be performed on the matrix. The development of a sparse matrix format is nontrivial; formats exploit algebraic properties such as commutativity, associativity and zero; have to judiciously choose between linear and random access to array data to improve cache locality, memory bandwidth and use of vector instructions. Sparse codes are used heavily in scientific applications, simulations and data mining, as well as other domains. It is expected that more formats will be designed to support future (parallel) platforms. Our goal is to simplify their development.

Sparse matrix codes are typically implemented using imperative languages like C and Fortran. This gives programmers control over low-level details of the computation, allowing them to create optimized implementations. However, imperative implementations obfuscate the structure of the format because logically independent steps of sparse matrix construction are fused, resulting in code with loop nests that contain complex array indirections, in-place data mutation and other low-level optimizations. Not only is the code hard to read, it is also challenging to verify. In fact, we failed to verify the functional correctness of even simple formats using several state-of-the-art tools. The key reason was that describing the properties of the format construction expressed using such low-level implementations required complex invariants that were hard to formulate. Consequently, we sought to raise the level of abstraction in programming sparse matrix formats.

We describe a new approach to implementing and verifying sparse matrix codes. The main idea is to specify sparse codes as functional programs, where a computation is a sequence of high-level transformations on lists. We then use Isabelle/HOL to verify full functional correctness of programs.

We identify a *"little language"* (LL) for specifying a variety of sparse matrix formats. LL is a strongly typed, variable-free functional programming language in the spirit of FP [1]. It is also influenced by such languages as APL, J, NESL and Python, but favors simplicity and ease of programming over generality and terseness. LL provides several built-in functions and combinators for operations over vectors and matrices common in sparse formats. LL is restricted by design, lacking custom higher-order functions, recursive definitions, and a generic reduction operator. These limitations of LL, as well as its purely functional semantics, facilitate automatic verification of sparse codes.

The contributions of this paper can be summarized as follows.

- We design a variable-free functional language for sparse matrix codes. We show how interesting and complex sparse formats can be naturally and concisely expressed in LL (Section 3).

* Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.

† This work was done while visiting Stanford University supported in part by grants NSF CNS-050955 and NSF CCF-0430378 with additional support from DARPA.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

- We describe a powerful proof method for automatic verification of sparse matrix codes using Isabelle/HOL [11] (Section 4).
- We evaluate the reusability of proof rules in our theory and its extensibility to proving additional formats. We show that our language and verifier can accommodate complex formats including Jagged Diagonals (JAD) and Coordinate (COO), as well as hierarchical formats including Sparse CSR (SCSR), register- and cache-blocking schemes (Section 5). As far as we know, this is the first successful attempt in proving full functional correctness of operations on such formats.

We are currently writing a compiler which automatically generates efficient low-level code from LL programs.

2. Overview

This section outlines our solutions for implementing and verifying sparse matrix programs. We demonstrate our language using the JAD sparse format, and the proof system using the CSR sparse format. These formats are introduced properly in Section 3; in this section, we will make do with an informal overview of the formats and the examples shown in Fig. 1.

2.1 Sparse matrix codes in the LL language

Sparse matrix formats are usually constructed with a sequence of transformations. For example, a JAD sparse matrix is constructed in three steps, by (i) compressing each row in the dense matrix; (ii) sorting compressed rows by their length; and (iii) transposing the rows. Efficient imperative implementations usually fuse these distinct steps, which complicates code comprehension and maintenance. We define a small functional language that keeps these steps separate. The fusion, necessary for performance, will be performed by a data-parallel compiler (which is under development and outside the scope of this paper).

Let us compare the characteristics of imperative and functional implementations of JAD format construction. Consider first the C code that compresses a dense matrix `M` into the JAD format, represented by arrays `P`, `D`, `J`, and `V`. The low-level code reads and writes a single word at a time, relies heavily on array indirections (*i.e.*, array accesses whose index expressions are themselves array accesses), and explicitly spells out loop boundaries. The code does not distinguish the three construction steps provides little insight into the JAD format:

```
lenperm (M, P); /* obtain row permutation */
for (d = k = 0; d < n; d++) {
  kk = k;
  for (i = 0; i < n; i++) {
    for (j = nz = 0; j < m; j++)
      if (M[P[i]][j])
        if (++nz > d) break;
    if (j < m) {
      J[k] = j;
      V[k] = M[P[i]][j];
      k++;
  } }
  if (k == kk) break;
  D[d] = k; }
```

Contrast the C code with this LL program, which is a composition of three functions corresponding to the steps in JAD construction. The function composition operator is `->`.

```
def jad: csr -> lenperm -> (fst, snd -> trans)
```

LL is a functional language rooted in the variable-free style of FP/FL [1], which means that functions do not refer to their arguments by name; instead, they transform a single, unnamed input parameter. For example, if the input to a function is a pair, then a function extracts the first element using the built-in function `fst`. LL is strongly typed and datatypes include numbers, Boolean values, pairs and lists. Vectors are represented by lists, and matrices by lists of (row) vectors. Compressed matrix representations use a variety of nested data structures built of lists and pairs.

$$\begin{pmatrix} a & 0 & 0 & 0 \\ b & c & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & d & 0 & e \end{pmatrix}$$

(a) Dense matrix.

P: $[1\ 3\ 0\ 2]$
D: $[3\ 5]$
J: $[0\ 1\ 0\ 1\ 3]$
V: $[b\ d\ a\ c\ e]$

(b) JAD sparse format.

R: $[1\ 3\ 3\ 5]$
J: $[0\ 0\ 1\ 1\ 3]$
V: $[a\ b\ c\ d\ e]$

(c) CSR sparse format.

Figure 1. Two sparse matrix formats. Shown are "imperative" representations; their LL counterparts are in Figures 2 and 4.

The three steps of JAD construction in LL are visualized in Fig. 2, which shows the dense matrix, the resulting JAD matrix, as well as the intermediate values of JAD construction. Notice that the JAD representation in LL (the result in Fig. 2) is more abstract than the JAD format in C (Fig. 1(b)). Where LL formats rely on lists of lists, the C formats linearize the outer list and create explicit indexing structures to access the inner lists. LL thus frees the programmer from reasoning about these optimized data structure layouts, eliminating dependence on explicit array indirection.

The first step compresses rows by invoking the constructor for the sparse format CSR. In the second step, the function `lenperm` sorts the compressed rows by decreasing length:

```
def lenperm:
  [(len, (#, id))] -> sort -> rev -> [snd] -> unzip
```

Here, the syntax $[f]$ denotes a map that applies the function f over the elements of the input list: `len`, `#` and `id` return the length of the current element, the position index of that element in the list, and the element itself (identity), respectively. The third-step function `(fst, snd -> trans)` takes a pair and produces a pair in which first element is unchanged and the second element is transposed.

In summary, LL lifts an intricate imperative computation into a cleaner functional form, exposes high-level stages and the flow of data from one stage to another, and encourages the programmer to think about invariants over intermediate results. These benefits are not merely due to the use of functional programming. We believe that they are equally attributed to our careful selection of a very simple subset of functional language features, designed with the sparse matrix domain in mind. In particular, LL does not support lambda abstractions, which encourages expressing computations as pipelines of functions. LL also excludes definitions of recursive functions and a general fold operator, both of which are compensated for by a versatile set of built-ins (*e.g.*, `zip` and `sum`) and combinators for handling lists (*e.g.*, **`map`** and **`filter`**). These restrictions contribute to our ability to automatically verify LL programs because they sidestep the need to infer induction invariants, a hard task for automated tools. The LL language is introduced in detail in Section 3.

We have recently developed compiler for LL that relies on optimization techniques pioneered in NESL [3] and later generalized in Data Parallel Haskell [4]. Thanks to LL's simplicity, we were able to simplify the compilation and indentify more opportunities for optimization. Initial results indicate that code generated for real-world formats such as register-blocked CSR (see Section 5.2) runs as fast as a hand-optimized code and scales well to multiple cores.

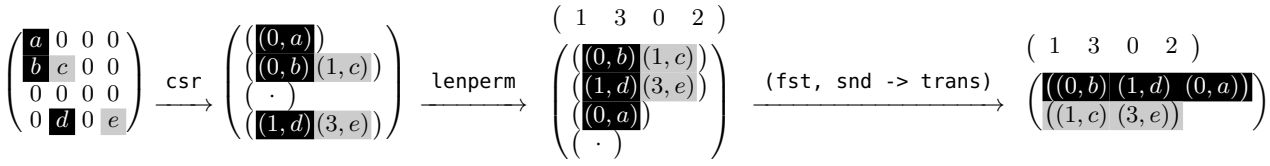

Figure 2. The three steps of JAD format construction. Shown are the dense matrix, the JAD matrix, and the two intermediate values.

2.2 Verifying sparse matrix codes

There are at least two arguments for full functional verification of sparse matrix codes. First, classical static typing is insufficient for static bug detection because these programs contain array indirection, whose memory safety would be typically guaranteed only with run-time safety checks. Dependent type systems may be able to prove memory safety but, in our experience, the necessary dependent-type predicates would need to capture invariants nearly as complex as those that we encountered during full functional verification. For example, to prove full functional correctness, one may need to show that a list is some permutation of a subset of values in another list; to prove memory safety, one may need to show that the values in a list are smaller than the length of another list. It thus seemed to us that with a little extra effort, we can use theorem proving to extend safety to full functional correctness.

The second reason for full functional verification is synthesis of sparse matrix programs, including the discovery of new formats. In inductive synthesis, which is conceptually a search over a space of plausible (*i.e.*, potentially semantically incorrect) implementations, a full functional verifier is a prerequisite for synthesis because it is an arbiter of correctness of the selected implementation. Synthesis, however, is outside the scope of this paper.

Before settling on the design presented in this paper, we set as our goal the full functional verification of *imperative* sparse code, in the style presented in Section 2.1. However, even the simple CSR format turned out to be rather overwhelming. We attempted to verify its correctness in multiple ways: (i) manually with Hoare-style logic, both with first-order predicates and inductive predicates; (ii) with ESC/Java [6]; (iii) with TVLA [13]; and (iv) using a SAT-based bounded model checker. The results were unsatisfactory either because it took weeks to develop the necessary invariants (i, ii), the abstraction was too complex for us to manage (iii), or because the checker scaled poorly (iv). Eventually, we concluded that we needed to verify sparse codes at a higher level of abstraction (and separately compile the verified code into efficient low-level code). Turning our attention to functional programs allowed us to replace explicit loops over arrays with maps and a few fixed reductions over lists, which in turn simplified the formulation and encapsulation of inductive invariants.

Let us the simple CSR format to give the rationale for the design of our proof system. Suppose that A and x are concrete language objects that, respectively, contain dense representations of a mathematical matrix B and a vector y. We want to prove that the product of the CSR-compressed A with x produces an object that is a valid (dense) representation of the vector $B \cdot y$. Note that the product is CSR-specific. Formally, our verification goal is

$$\texttt{csrmv}(\texttt{csr}(A), x) \overset{m}{\rhd} B \cdot y$$

The goal expresses the relationship between a mathematical object and its concrete counterpart with the *representation relation* $a \overset{k}{\rhd} b$, which states that the concrete object a represents the mathematical vector b: for all $i < k$, $a\texttt{[}i\texttt{]}$ equals b_i and the lengths of a and b are k. In the course of the proof, we may need to track relationships on various kinds of concrete objects; one of our contributions is to define suitable representation relations for the objects that arise in sparse matrix programs.

We use Isabelle/HOL as our underlying theorem prover. We embed LL functions in Isabelle using typed λ-calculus and Isabelle libraries. Our proofs deploy two techniques: (a) *term simplification*, which rewrites subterms in functions into simpler, equivalent ones; and (b) *introduction*, which substitutes a proof goal with a certain term for alternative goal(s) that do not contain the term, and whose validity implies the validity of the original goal. In our example, term simplification unfolds the definitions of `csrmv` and `csr` and applies standard rules for simplifying function application and composition, map and filter operations on lists, and extraction of elements from pairs. This results in the goal

$$\texttt{[enum -> [snd != 0 ?] ->[snd} * x\texttt{[fst]] -> sum]}(A) \overset{m}{\rhd} B \cdot y \tag{1}$$

The LL function on the left enumerates each row of A into a list of (column index, value) pairs, then filters out pairs whose second element is zero (`[snd != 0 ? ]`). For the remaining pairs, it multiplies the second (nonzero) component with the value of x at the index given by the first component (`[snd` $*$ x`[fst]]`). Finally, it sums the resulting products (`sum`). So far, simplification has done a good job.

To carry out the next step of the proof, we observe that the missing zeros do not affect the result of the computation, so we would like to simplify the left-hand-side by rewriting away the filter (`[snd != 0 ? ]`); this would effectively "desparsify" the left-hand side, moving it closer to the mathematical right-hand-side. Unfortunately, standard simplification available in prover libraries cannot perform the rewrite; we would need to add a rule tailored to this format. The hypothetical rule, shown below, would match p with `snd != 0` and f with `snd` $*$ x`[fst]`.

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

The rule would achieve the desired simplification but we refrain from adding such a rule because it would take a considerable effort to prove it. Additionally, the rule would of little use in cases where the LL operations appear in just a slightly syntactically different way.

We will instead rely on introduction which, by substituting the current goal with a set of goals, isolates independent pieces of reasoning. Introduction rules tend to be more general than simplification rules because they are concerned with a single construct from the current goal. Also, the validity of introduction rules is easier to establish.

Our first introduction rule substitutes in the goal (1) the *whole result vector* with a *single element* of that vector. In effect, this removes the outermost map from the LL function on the left-hand side. Semi-formally, the rule for map can be stated as follows:

$$\frac{\text{length of } A \text{ is } m \qquad \forall i < m \,.\, f(A\texttt{[}i\texttt{]}) = B_i}{\texttt{[}f\texttt{]}(A) \overset{m}{\rhd} B} \tag{2}$$

In goal (1), f matches the entire chain of `enum -> ... -> sum` and the new subgoals are

(i) length of A is m

(ii) $\forall i < m$.

`enum -> [snd != 0 ?] ->`

`[snd * x[fst]] -> sum` $(A[i]) = \sum_{j<n} B_{i,j} \cdot y_j$

We now need a second introduction step to remove the summation on both sides of the equality: instead of requiring equivalence between *sums of sequences* of numbers, we will require equivalence between the *values in the sequences* themselves. In order for such a rule to be general enough, we need to permit arbitrary permutations of the values in a sequence to prove programs that exploit associativity and commutativity of addition. A hypothetical rule may look as follows, where $[x_i|p(x_i)]_{i=a,\ldots,a+\delta}$ denotes a construction of an ordered list of elements out of $x_a, \ldots, x_{a+\delta}$ that satisfy p.

$$\frac{\exists n' \leq n, \text{ permutation } P\ .\quad f(A[i]) \overset{n'}{\triangleright} [B_{i,j} \mid B_{i,j} \neq 0]_{j=P_0,\ldots,P_{n-1}}}{\texttt{sum}(f(A[i])) = \sum_{j<n} B_{i,j}}$$

This rule is problematic for two reasons. First, it is more complex than what we may want to prove. For example, the premise constructs a filtered and permuted mathematical vector on the right-hand side (via list comprehension), rather than keeping the mathematical object untouched. This might hinder our ability to link our proof goal to the original input matrix in the assumptions of the theorem. Second, the rule is not as general as we would like because a concrete representation *may* contain zeros.

Our approach is to enrich the representation relation ($a \overset{k}{\triangleright} b$). This relation uses plain equality to relate single elements from the two vector objects, which limits its applicability to more subtle mappings. To express a relation where, say, each element in a concrete representation equals the corresponding vector element multiplied by some value, we parameterize the representation relation with an *inner relation* that describes how individual elements represent their mathematical counterparts. Individual elements need not be scalars; they could be, recursively, lists. Therefore, inner relations could be parameterized by further inner relations.

Our domain proof theory for sparse matrices is novel in two ways. First, we define common representation relations that occur in our domain. Our infrastructure is powerful because we (a) insist on relaxing invariants as much as possible (*e.g.*, zeros may still be present in a compressed representation); (b) encapsulate many quantifications and implications in the representation relations (*e.g.*, universal quantification on all indexes of a vector, existence of a permutation); (c) include necessary integrity constraints in the representation relations (*e.g.*, lengths must match). The representation relations we define include indexed list (*ilist*), where the element at position i represents the ith vector element; value list (*vlist*), in which all nonzero values are represented; and associative list (*alist*), which contains index-value pairs. These representation relations raise the level of abstraction and focus theory development on these prevalent data representation. The use of representation relations also prevents oversimplification of proof terms by concealing their internal conjuncts from Isabelle's simplifier.

The second novelty is parameterizing the inner predicate, which describes how the vector elements represent their mathematical counterparts. In the case of a vector of numbers, we use equality. For matrices, the inner relation relates a single row to its concrete indexed-list representation (*ilist*); technically, the inner relation predicate is a parameter to the (outer) representation predicate for the whole matrix. In addition to reducing the number of rules, parameterization helps with syntactic matching and substitution of inner comparators during introduction. For example, with a parameterized relation, an introduction rule for map similar to that in Eq. (2) can be written more generally and concisely: the conclusion of the rule contains an indexed-list representation relation where the concrete object is the term `[`f`]`(x) (*i.e.*, map with an arbitrary function f over x) and the inner representation relation is some arbitrary predicate P—our parameter. The premise of the rule is again an indexed-list representation relation where the concrete object is x and the inner representation relation is $\lambda i\ a\ b.\ P(i, a, f(b))$. Fortunately, Isabelle can match and substitute terms that contain parameters such as P (as well as f and x); these rules can thus be applied automatically.

The representation relations are described in Section 4. Section 5 evaluates whether they improve reuse of rules and thus simplify theory development; we argue that the principles used in our approach are crucial for proofs on *nested data representations*. It may be interesting to apply such parameterized representation relations also in other domains.

3. High-Level Sparse Matrix Programming

Sparse matrix codes can often be decomposed into sequences of high-level transformations. This section describes LL and its use for expressing such computations naturally and concisely.

3.1 Introduction to LL

The LL language constructs are presented in Fig. 3. The semantics of each construct is shown, either by translation to Isabelle/HOL λ-calculus and standard library for lists [11], or by de-sugaring to simpler LL constructs. The language includes (a) general functions such as identity, equality, constants, conditional branching, and a name binding form used for assigning names to components of an input value; (b) construction of pairs/tuples and extraction of values from pairs; (c) pipeline- and application-style composition, as well as a curried application operator; (d) standard arithmetic operators and comparators; (e) Boolean logic operators; and (f) list handling functions (*e.g.*, distribution of values onto lists, zipping, enumeration, concatenation) and combinators (map, filter, and a unified comprehension syntax).

3.2 Specification of sparse codes using LL

Compressed sparse rows (CSR). This format compresses each row by storing nonzero values together with their column indexes. The resulting sequence of compressed rows is not further compressed, so empty (all zero) rows are retained. This enables random access to the beginning of each row, but requires linear traversal to extract a particular element out of a row. CSR is widely used because it is relatively simple and entails good memory locality for row-wise computations such as SpMV.

Implementing CSR in C, shown below,[1] is not trivial. Traversal of the dense matrix (construction) or the compressed rows (SpMV) is done with nested loops. Single values are copied (construction) or extracted (SpMV) through array indirection. Compressed row boundaries need to be stored and observed. That said, the resulting SpMV code is rather efficient as the inner product of each row is incrementally accumulated, using very few instructions and avoiding unnecessary memory accesses. Applying CSR construction to the 4-by-4 matrix in Fig. 1(a) yields the data structure in Fig. 1(c).

[1] For brevity, we omit memory allocation and initialization and assume that matrix dimensions are known at compile-time.

`id`	$\lambda x.\ x$
`eq` (`=`), `neq` (`!=`)	$\lambda(x, y).\ x = y$, $\lambda(x, y).\ x \neq y$
n, `true`, `false`	$\lambda y.\ n$, $\lambda y.$ *true*, $\lambda y.$ *false*
f `?` g `|` h	$\lambda x.$ *if* $f\ x$ *then* $g\ x$ *else* $h\ x$
l_1`,...,`l_k `=` f`:` g †	$\big(\lambda(x_1, \ldots, x_k).\ g[l_i/\lambda y.\ x_i]\ (x_1, \ldots, x_k)\big) \circ f$
`(`f`)`	f
`(`f_1`,`f_2`,...,`f_k`)`	$\lambda x.\ (f_1\ x, f_2\ x, \ldots, f_k\ x)$
`fst`, `snd`	$\lambda(x, y).\ x$, $\lambda(x, y).\ y$
f `->` g	$g \circ f$
g`(`f_1`,...,`f_k`)`	`(`f_1`,...,`f_k`)` `->` g
g `‘` f	`(`f`, id)` `->` `g`
`add` (`+`), `sub` (`-`), `mul` (`*`), `div` (`/`), `mod` (`%`)	$\lambda(x, y).\ x + y$, $\lambda(x, y).\ x - y$,...
`leq` (`<=`), `lt` (`<`), `geq` (`>=`), `gt` (`>`)	$\lambda(x, y).\ x \leq y$, $\lambda(x, y).\ x < y$,...
`sum` (`/+`), `prod` (`/*`)	*foldl* $(op +)$ *0*, *foldl* $(op *)$ *1*
`and` (`&&`), `or` (`||`)	$\lambda(x, y).\ x \wedge y$, $\lambda(x, y).\ x \vee y$
`neg` (`!`)	$\lambda x.\ \neg x$
`conj` (`/&&`), `disj` (`/||`)	*foldl* $(op \wedge)$ *True*, *foldl* $(op \vee)$ *False*
`len`	*length*
`rev`	*rev*
`sub` (f`[`g`]`)	$\lambda(v, i).\ v\ !\ i$
`subseq` (f`[`g`]`)	$\lambda(v, s).$ *map* $(\lambda i.\ v\ !\ i)\ s$
`distl`, `distr`	$\lambda(x, v).$ *map* $(\lambda y.\ (x, y))\ v$, $\lambda(v, x).$ *map* $(\lambda y.\ (y, x))\ v$
`zip`, `unzip`	*unsplit zip*, $\lambda l.$ (*map fst*, *map snd*)
`enum`	$\lambda v.$ *zip* $[0\ ..<$ *length* $v]\ v$
`concat`	*concat*
`infl`	$\lambda(d, n, v).$ *foldr* $(\lambda(i, x)\ v.\ v[i := x])\ v$ (*replicate* $n\ d$)
`gather`	$\lambda xs.$ *map* $(\lambda k.\ (k,$ *map snd* (*filter* $(\lambda(k', v).\ k = k')\ xs)))$ (*remdups* (*map fst* xs))
`sort`	*sort_key fst*
`trans`	$\lambda v.\ [$*map* $(\lambda v.\ v\ !\ i)$ (*takeWhile* $(\lambda v.\ i <$ *length* $v)\ v)$. $i \leftarrow [0\ ..<$ *if* $v = []$ *then* 0 *else length* $(v\ !\ 0)]]$
`map` f	*map* f
`filter` f	*filter* f
`[`l_1`,...,`l_n `=` f`:` g `?` h`]` ‡	**`filter`** `(`l_1`,...,`l_n `=` f`:` g`);` **`map`** `(`l_1`,...,`l_n`:` h`)`

Figure 3. LL constructs and their translation to Isabelle/HOL. Here, f, g and h denote functions, n a number, and l a label. Alternative infix, prefix or mixfix notation is shown in parentheses. † f defaults to `id`. ‡ Value naming is optional, f and h default to `id` and g to `true`.

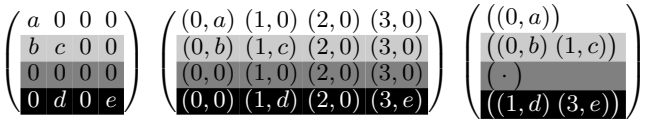

Figure 4. Conceptual phases in CSR construction.

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

```
/* CSR SpMV. */
for (i=k=0; i<m; i++)
 for (y[i]=0; k<R[i]; k++)
  y[i] += V[k] * x[J[k]];
```

Fig. 4 shows the high-level stages in CSR compression mentioned above. Given (a), each row is enumerated with column indexes, resulting in (b). Pairs containing a zero value are then filtered, yielding (c). A dataflow view of such a computation is shown in Fig. 5. Notice how similar it is to the following LL function.

```
def csr: [enum -> [snd != 0 ? ]]
```

Using name binding in comprehensions may improve clarity.

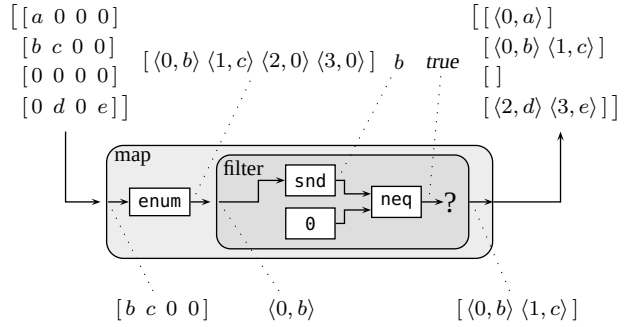

Figure 5. Dataflow view of high-level CSR construction.

```
[enum -> [j, v: v != 0 ? ]]
```

Alternatively, one can use an explicit enumeration operator inside comprehensions. The following variant appears more “integrated”, but in fact entails the exact same semantics.

```
[[v: v != 0 ? (#, v)]]
```

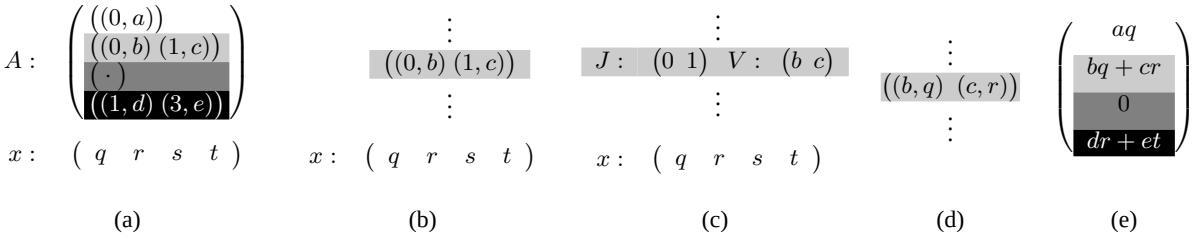

Figure 6. Conceptual phases in CSR SpMV.

A more verbose variant uses Python-style comprehension. This variant is de-sugared to the original definition.

```
def csr(A):
 [[(j, v) for j, v in enum(r) if v != 0]
  for r in A]
```

Fig. 6 shows the stages in CSR SpMV. Each compressed row is multiplied separately, as shown in (b). First, column indexes are separated from nonzero values as in (c). They are used to retrieve corresponding values from x, pairing them with their respective row values as in (d). Finally, values in pairs are multiplied and the products are summed, yielding the inner-product in (e). This maps to the following LL function.

```
def csrmv(A, x):
 A ->
 [J, V = unzip: (V, x[J]) -> zip -> [mul] -> sum]
```

Here, too, it is possible to write a more integrated variant that bundles multiplication with the extraction of single values. Although semantically equivalent, the resulting code is less amenable to vectorization due to the use of word-level operations.

```
A -> [[j, v: v * x[j]] -> sum]
```

Jagged diagonals (JAD). This format deploys a clever compression scheme that allows handling of sequences of nonzeros from multiple rows, taking advantage of vector instructions. The ith nonzero values from all rows are laid out consecutively in the compressed format, constituting a "jagged diagonal". Since nonzeros are distributed differently in each row, column indexes need to be stored as well. These steps can be thought of as per-row *compression* (as shown above for CSR), followed by *transposition* to invert the direction of compressed rows and ith-element columns.

However, packing ith elements in a predetermined order—*e.g.*, from the first to the last row—induces a problem: one needs to account for compressed rows that are shorter than other rows that succeed them.[2] This is addressed by adding a *sorting* step between row compression and transposition, in which rows are ordered by decreasing number of nonzeros. The sort permutation is stored with the resulting diagonals, so the correct order of rows can be restored. These conceptual steps in JAD compression are visualized in Fig. 2 and the LL implementation is shown in Section 2.1.

Fig. 7 shows the high-level steps in JAD SpMV. (b) is obtained by computing, for each diagonal, the cross-product of its induced vector of values with the elements of x corresponding to their column indexes. These are transposed to obtain the lists of products in each (nonzero) row as in (c). Products in corresponding rows are summed, obtaining (d). In (e), each inner product is paired with its row index, which are then "inflated" to obtain the dense result vector in (f). The following LL function implements these steps.

```
def jadmv((P, D), x):
 (P,
  D -> [unzip -> snd * x[fst]] -> trans -> [sum]) ->
 zip -> infl(0, m, id)
```

Other formats. Two additional standard formats are Coordinate (COO) and Compressed Sparse Columns (CSC) [10]. COO is a highly-portable compression in which nonzeros are stored together with their row and column indexes in a single, arbitrarily ordered sequence. Construction can be implemented in LL as follows.

```
def coo: [i = #: [v: v != 0 ? (i, #, v)]] -> concat
```

COO SpMV is less straightforward: one needs to account for the fact that nonzeros of a particular row might be scattered along the compressed list. It is necessary to *gather* those values prior to computing the inner-product. This is expressed as follows.

```
def coomv (A, x):
 A-> gather ->
 [(fst, snd -> [j, v: v * x[j]] -> sum)] ->
 infl(0, m, id)
```

A CSC representation is obtained by compressing the nonzero values in the column direction, instead of row direction as in CSR. In C, it is done by swapping the order of the loops iterating over the dense matrix, and storing the row index with the nonzero values. In LL, it amounts to prepending a transposition to CSR construction.

```
def csc: trans -> csr
```

Like COO, CSC SpMV calls for a gather operation prior to summing row cross-products.

```
def cscmv:
 zip -> [cj, xj: cj -> [i, v: (i, v * xj)]] ->
 concat -> gather ->
 [(fst, snd -> sum)] -> infl(0, m, id)
```

Here, too, the fact that data layout is not in line with the computation entailed by matrix-vector multiplication calls for additional steps to massage the result into a proper vector form.

In addition to the above formats, LL can naturally and succinctly describe hierarchical compression. This includes Sparse CSR (SCSR) and different block variants of all of the above. These will be described and studied in Section 5.

4. Verifying Sparse Codes using Isabelle/HOL

We make use of Isabelle's rich infrastructure in implementing a proof method for sparse matrix codes. This includes the *simplifier* and a powerful *tactical language*, which is used to combine existing proof methods in forming new ones. All parts of our proofs are checked from first principles by a small LCF-style kernel.

[2] Transposition inverts columns up to the first missing element, below which all other elements are omitted. In this respect it is "lossy" and the equality $A = A^{TT}$ only holds for matrices whose rows are sorted by length.

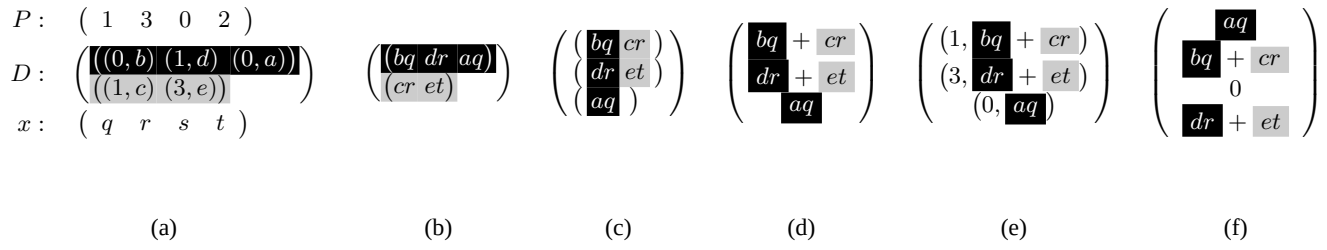

Figure 7. Conceptual phases in JAD SpMV.

4.1 Translating LL to Isabelle/HOL

Fig. 3 constitutes a *shallow embedding* [16] of LL in Isabelle/HOL, a standard technique when the goal is to verify correctness of programs in some language. In this approach, the functions and types of an object language (LL) are written directly in the language of the theorem prover (typed λ-calculus). Subsequent logical formulas relate to these translated programs as ordinary HOL objects, which allows to leverage existing support for proving properties of them. The CSR implementation in Section 3.2 translates to the following definitions, which will be used in our proofs.

$$\begin{aligned} \mathit{csr} &= (\mathit{filter}(\lambda(j,v).v \neq 0)) \circ \mathit{enum} \qquad \text{and} \\ \mathit{csrmv}\,(A,x) &= \mathit{map}\,(\mathit{listsum} \circ \mathit{map}\,(\lambda(x,y).\, x * y) \circ \\ &\qquad \mathit{unsplit}\ \mathit{zip} \circ \\ &\qquad (\lambda(J,V).\,(V, \mathit{map}\,(\lambda i.\, x\,!\,i)\ J)) \circ \\ &\qquad \mathit{map}\ \mathit{unzip} \end{aligned} \tag{3}$$

We now pose the verification theorem: when A index-represents the $m \times n$-matrix A' and x the n-vector x', the result of CSR SpMV applied to a CSR version of A and to x represents the m-vector that is equal to $A' \cdot x'$.

$$\begin{aligned} &\mathit{ilist}_M\ m\ n\ A'\ A \wedge \mathit{ilist}_v\ n\ x'\ x \\ &\longrightarrow \mathit{ilist}_v\ m\ (\lambda i.\ \Sigma j < n.\ A'\ i\ j * x'\ j)\ (\mathit{csrmv}\,(\mathit{csr}\,A,\ x)) \end{aligned} \tag{4}$$

The remainder of this section presents the formalism and explains the reasoning used in proving this goal.

4.2 Formalizing vector and matrix representations

We begin by formalizing vectors and matrices in HOL. Mathematical vectors and matrices are formalized as functions from indexes to values, namely nat $\to \alpha$ and nat $\to$ nat $\to \alpha$, respectively; note that the $\to$ type constructor is right-associative, hence a matrix is a vector of vectors. Dimensions are not encoded in the type itself, and values returned for indexes exceeding the dimensions can be arbitrary, which means that many functions can represent the same mathematical entity. Concrete representations of dense and sparse vectors/matrices are derived from the LL implementation and consist of lists and pairs. Commonly used representations include indexed lists, value lists and associative lists, all of which are explained below.

We introduce *representation relations* (defined as predicates in HOL) to link mathematical vectors and matrices with different concrete representations, for three reasons. First, in proving correctness of functions we map operations on concrete objects to their mathematical counterparts. This is easy to do for indexed list representations but gets unwieldy with others. We hide this complexity inside the definitions of the relations. Second, predicates can be used to enforce integrity constraints of the representation. For example, an associative list representation requires that index values are unique; or the lengths of a list of indexed list representations need to be fixed. Third, for some representations (*e.g.*, value list) there exists no injective mapping from concrete objects to abstract ones, forcing us to use relations rather than *representation functions*. Using relations across the board yields a more consistent and logically lightweight framework.

An *indexed list* representation of an n-vector x' by a list x is captured by the *ilist* predicate. Note that we refrain from fixing vector elements to a specific type (*e.g.*, integers) and instead use type parameters α and β to denote the types of inner elements of the mathematical and concrete vectors, respectively.

$$\begin{aligned} &\mathit{ilist} :: \text{nat} \to (\text{nat} \to \alpha \to \beta \to \text{bool}) \to \\ &\qquad (\text{nat} \to \alpha) \to [\beta] \to \text{bool} \\ &\mathit{ilist}\ n\ P\ x'\ x \iff \\ &\qquad (\mathit{length}\ x = n) \wedge (\forall i < n.\ P\ i\ (x'\ i)\ (x\,!\,i)) \end{aligned}$$

The parameter P is a relation that specifies the representation of each element in the vector. For ordinary vectors, it is equality of elements. However, P turns useful for matrix representation, as we can use arbitrary relations to determine the representation of inner vectors. We introduce abbreviations for the common cases of indexed list representations.

$$\begin{aligned} \mathit{ilist}_v\ n\ x'\ x &\iff \mathit{ilist}\ n\ (\lambda j.\ \mathit{op} =)\ x'\ x \\ \mathit{ilist}_M\ m\ n\ A'\ A &\iff \mathit{ilist}\ m\ (\lambda i.\ \mathit{ilist}_v\ n)\ A'\ A \end{aligned}$$

An *associative list* representation is central to sparse matrix codes as it is often used in vector compression. It is captured by the *alist* predicate.

$$\begin{aligned} &\mathit{alist} :: \text{nat} \to (\text{nat} \to \alpha \to \beta \to \text{bool}) \to \\ &\qquad (\alpha\ \text{set}) \to (\text{nat} \to \alpha) \to [\langle \text{nat}, \beta \rangle] \to \text{bool} \\ &\mathit{alist}\ n\ P\ D\ x'\ x \iff \\ &\qquad \mathit{distinct}\ (\mathit{map}\ \mathit{fst}\ x) \wedge \\ &\qquad (\forall (i,v) \in \mathit{set}\ x.\ P\ i\ (x'\ i)\ v \wedge i < n) \wedge \\ &\qquad (\forall i < n.\ x'\ i \notin D \longrightarrow \exists v.\ (i,v) \in \mathit{set}\ x) \end{aligned}$$

Here, *distinct* is a predicate stating the uniqueness of indexes (*i.e.*, keys) in x. Each element in an associative list must relate to the respective vector element, also requiring that index values are within the vector length. Finally, each element in the vector that is not a default value (specified by the set of values D) must appear in the representing list. Note that a *set* of default values accounts for cases where more than one such value exists, as in the case of nested vectors where each function mapping the valid dimensions to zero is a default value. Also note that *alist* does not enforce a particular order on elements in the compressed representation, nor does it insist that all default values are omitted.

Sometimes concrete objects contain only the values of the elements in a given vector, without mention of their indexes. This *value list* representation often occurs prior to computing a cross- or dot-product. It is captured by the *vlist* predicate, which states

that the list of values can be zipped with some list of indexes p to form a proper associative list representation. (The length restriction ensures that no elements are dropped from the tail of x.)

$$\begin{aligned}&\mathit{vlist} :: \text{nat} \to (\text{nat} \to \alpha \to \beta \to \text{bool}) \to \\ &\quad (\alpha \text{ set}) \to (\text{nat} \to \alpha) \to [\beta] \to \text{bool} \\ &\mathit{vlist}\ n\ P\ D\ x'\ x \iff \\ &\quad \exists p.\ \mathit{length}\ p = \mathit{length}\ x \wedge \\ &\qquad \mathit{alist}\ n\ P\ D\ x'\ (\mathit{zip}\ p\ x)\end{aligned}$$

Additional representations can be incorporated into our theory. For example, when a matrix is compressed into an associative list, a dual-index representation relation can be defined similarly to *alist*.

4.3 Proving correctness of sparse matrix computations

We prove Eq. (4) using term rewriting and introduction rules. Introduction rules are used whenever further rewriting cannot be applied. An introduction rule is applied by resolution: applying the rule $G\ x \wedge H\ y \longrightarrow F\ x\ y$ to the goal $F\ a\ b$ yields two new subgoals, $G\ a$ and $H\ b$.

The theorem in Eq. (4) makes the following two assumptions,

$$\mathit{ilist}_M\ m\ n\ A'\ A \tag{5}$$

$$\mathit{ilist}_v\ n\ x'\ x \tag{6}$$

which are added to the set of available introduction rules as $\mathit{true} \longrightarrow \ldots$. The conclusion of Eq. (4) is our initial proof goal,

$$\mathit{ilist}_v\ m\ (\lambda i.\ \Sigma j < n.\ A'\ i\ j * x'\ j)\ (\mathit{csrmv}\ (\mathit{csr}\ A)\ x) \tag{7}$$

Simplifying the goal. We begin by applying Isabelle's simplifier using Eq. (3) and standard rules for pairs, lists, arithmetic and Boolean operators. This removes most of the function abstractions, compositions and pair formations due to the translation from LL. Our new goal is analogous to Eq. (1) in Section 2.2.

$$\begin{aligned}&\mathit{ilist}_v\ m\ (\lambda i.\ \Sigma j < n.\ A'\ i\ j * x'\ j) \\ &\quad (\mathit{map}\ (\lambda r.\ \mathit{listsum}\ (\mathit{map}\ (\lambda v.\ \mathit{snd}\ v * x\ !\ \mathit{fst}\ v) \\ &\qquad\qquad (\mathit{filter}\ (\lambda v.\ \mathit{snd}\ v \neq 0)\ (\mathit{enum}\ r))))\ A)\end{aligned} \tag{8}$$

Solving the entire goal using rewriting alone calls for simplification rules that are too algorithm-specific. For example, the rule

$$\begin{aligned}&(\forall x \in \mathit{set}\ xs.\ \neg\ P\ x \longrightarrow f\ x = 0) \\ &\longrightarrow \mathit{listsum}\ (\mathit{map}\ f\ (\mathit{filter}\ P\ xs)) = \mathit{listsum}\ (\mathit{map}\ f\ xs)\end{aligned} \tag{9}$$

allows further simplification of Eq. (8), but fails for all formats that introduce more complex operations between *map* and *filter*.

Introduction rules on representation relations. Consider the equation in the conclusion of Eq. (9). We know that it holds when the two lists, xs and $\mathit{filter}\ P\ xs$, value-represent the same vector. By introducing rules, describing when it is allowed to apply *map*, *filter* and *enum* operations to value list representations, we prove that the result of *listsum* in Eq. (8) equals the mathematical dot-product.

Fig. 8 shows the introduction rules used in proving Eq. (4). Application of introduction rules is syntax directed, choosing rules whose conclusion matches the current goal. Given Eq. (8), the prover applies ILIST-MAP, which moves the map from the representing object into the inner representation relation, followed by ILIST-LISTSUM, which substitutes *listsum* with an equivalent notion of value-represented rows. This results in

$$\begin{aligned}&\mathit{ilist}\ m\ \big(\lambda i\ r'\ r.\ \mathit{vlist}\ n\ (\lambda j.\ \mathit{op} =)\ \{0\}\ r' \\ &\qquad (\mathit{map}\ (\lambda v.\ \mathit{snd}\ v * x\ !\ \mathit{fst}\ v) \\ &\qquad\quad (\mathit{filter}\ (\lambda v.\ \mathit{snd}\ v \neq 0)\ (\mathit{enum}\ r))))\big) \\ &\quad (\lambda i\ j.\ A'\ i\ j * x'\ j)\ A\end{aligned}$$

Further simplification is not possible at this point, nor can we modify the *vlist* relation inside *ilist*. Luckily, ILIST-VLIST matches our goal, lifting the inner *vlist* to the outermost level and permitting to further operate on the concrete parameters of *vlist*. Note that ILIST-VLIST has two assumptions, resulting in new subgoals

$$\mathit{ilist}\ m\ ?Q\ ?B'\ A \tag{10}$$

and

$$\begin{aligned}&\forall i < m.\ \mathit{vlist}\ n\ (\lambda j.\ \mathit{op} =)\ \{0\}\ (\lambda j.\ A'\ i\ j * x'\ j) \\ &\qquad \big(\mathit{map}\ (\lambda v.\ \mathit{snd}\ v * x\ !\ \mathit{fst}\ v) \\ &\qquad\quad (\mathit{filter}\ (\lambda v.\ \mathit{snd}\ v \neq 0)\ (\mathit{enum}\ (A\ !\ i))))\big)\end{aligned} \tag{11}$$

In Eq. (10), $?Q$ and $?B'$ are existentially quantified variables. They do not get instantiated when we apply ILIST-VLIST, and the subgoal Eq. (10) merely certifies that A has length n. Therefore, the prover is allowed to instantiate them arbitrarily and Eq. (10) is discharged by the assumption Eq. (5).

The rules VLIST-MAP, ALIST-FILTER and ALIST-ENUM can now be applied to Eq. (11). Note that applying them amounts to the effect of simplification using Eq. (9). However, they can be applied regardless of the way in which the three operations—*map*, *filter* and *enum*—are intertwined. Therefore, they are applicable in numerous cases where the context imposed by Eq. (9) is too restrictive.

The ALIST-FILTER rule forces us to prove that *filter* only removes default values, in the form of the following new subgoals,

$$\begin{aligned}&\forall i < m.\ \forall j < n.\ \forall v\ v'. \\ &\qquad \neg\ \mathit{snd}\ (j,\ v) \neq 0 \wedge v' = v * x\ !\ j \longrightarrow v' \in \{0\}\end{aligned} \tag{12}$$

$$\begin{aligned}&\forall i < m. \\ &\qquad \mathit{ilist}\ n\ (\lambda j\ v'\ v.\ v' = v * x\ !\ j)\ (\lambda j.\ A'\ i\ j * x'\ j)\ (A\ !\ i)\end{aligned}$$

Fortunately, subgoal Eq. (12) is completely discharged by the simplifier. The remaining goal is solved using the ILIST-MULT, ILIST-NTH, and $\text{ILIST}_v \to \text{ILIST}_M$, as well as the assumptions Eq. (5) and Eq. (6).

4.4 Automating the proof

The above proof outline already dictates a simple proof method. Isabelle's tactical language [15] provides us with ample methods and combinators that can be used to implement custom proof tactics. Our proof method is implemented as follows.

1. The *simplifier* attempts to rewrite the goal until no further rewrites are applicable, returning the new goal. If no rewrite rule could be applied, it returns an empty goal.
2. The *resolution* tactic attempts to apply each of the introduction rules and returns a new goal state for each of the matches. It is possible that more than one rule matches a given goal, *e.g.* ILIST-MAP and ILIST-NTH both match $\mathit{ilist}\ n\ (\lambda i\ v'\ v.\ v' = y\ !\ i)\ x'\ (\mathit{map}\ f\ x)$, resulting in a sequence of alternative goal states to be proved.

Invoking the proof method leads to a depth-first search on the combination of the two sub-methods. It maintains a sequence of goal states, initially containing only the main goal. After each

[3] The predicate P and the vector z' are arbitrary, they just help to state that x is a list of length m.

$$\frac{\mathit{ilist}\ n\ (\lambda i\ a\ b.\ P\ i\ a\ (f\ b))\ x'\ x}{\mathit{ilist}\ n\ P\ x'\ (\mathsf{map}\ f\ x)}\ \text{ILIST-MAP}$$

$$\frac{\mathit{ilist}\ m\ (\lambda i\ r'\ r.\ \mathit{vlist}\ n\ (\lambda j.\mathit{op} =)\ \{0\}\ r'\ (f\ r))\ A'\ A}{\mathit{ilist}\ m\ (\lambda i\ r'\ r.\ r' = \mathit{listsum}\ (f\ r))\ (\lambda\ i.\ \Sigma\ j < n.\ A'\ i\ j)\ A}\ \text{ILIST-LISTSUM}$$

$$\frac{\begin{array}{c}\mathit{ilist}\ m\ Q\ B'\ A\\ \forall i < m.\ \mathit{vlist}\ n\ (P\ i)\ (D\ i)\ (f\ (A'\ i)\ i)\ (g\ (A\ !\ i)\ i)\end{array}}{\mathit{ilist}\ m\ (\lambda i\ r'\ r.\ \mathit{vlist}\ n\ (P\ i)\ (D\ i)\ (f\ r'\ i)\ (g\ r\ i))\ A'\ A}\ \text{ILIST-VLIST}$$

$$\frac{\mathit{alist}\ m\ (\lambda i\ r'\ r.\ P\ i\ r'\ (f\ (i,\ r)))\ D\ x'\ x}{\mathit{vlist}\ m\ P\ D\ x'\ (\mathsf{map}\ f\ x)}\ \text{VLIST-MAP}$$

$$\frac{\begin{array}{c}\mathit{alist}\ n\ P\ D\ x'\ x\\ (\forall i < n.\ \forall v\ v'.\ \neg\ Q\ (i,\ v) \wedge P\ i\ v'\ v \longrightarrow v' \in D)\end{array}}{\mathit{alist}\ n\ P\ D\ x'\ (\mathit{filter}\ Q\ x)}\ \text{ALIST-FILTER}$$

$$\frac{\mathit{ilist}\ m\ P\ x'\ x}{\mathit{alist}\ m\ P\ D\ x'\ (\mathit{enum}\ x)}\ \text{ALIST-ENUM}$$

$$\frac{\begin{array}{c}\mathit{ilist}\ n\ (\lambda i\ v'\ v.\ v' = f\ i\ v)\ x'\ z\\ \mathit{ilist}\ n\ (\lambda i\ v'\ v.\ v' = g\ i\ v)\ y'\ z\end{array}}{\mathit{ilist}\ n\ (\lambda i\ v'\ v.\ v' = f\ i\ v * g\ i\ v)\ (\lambda i.\ x'\ i * y'\ i)\ z}\ \text{ILIST-MULT}$$

$$\frac{\mathit{ilist}_v\ m\ x'\ y \qquad \mathit{ilist}\ m\ P\ z'\ x}{\mathit{ilist}\ m\ (\lambda i\ v'\ v.\ v' = y\ !\ i)\ x'\ x}\ \text{ILIST-NTH}^3$$

$$\frac{\mathit{ilist}_M\ m\ n\ A'\ A \qquad i < m}{\mathit{ilist}_v\ n\ (A'\ i)\ (A\ !\ i)}\ \text{ILIST}_v \rightarrow \text{ILIST}_M$$

Figure 8. Introduction rules used in the proof of CSR SpMV.

$$\frac{k \neq 0}{\mathit{ilist}_v\ n\ (\lambda i.\ \mathit{block}\ k\ 1\ (\lambda i'\ j.\ A'\ (i * k + i')))\ A}{\mathit{ilist}_v\ (n * k)\ A'\ (\mathit{concat\_vectors}\ k\ A)}\ \text{ILIST-CONCAT\_VECTORS}$$

$$\frac{l \neq 0 \qquad \mathit{ilist}_v\ (m * l)\ x'\ x}{\begin{array}{c}\mathit{ilist}_v\ m\ (\lambda i.\ \mathit{block}\ l\ 1\ (\lambda i'\ j'.\ x\ (i * l + i')))\\ (\mathit{block\_vector}\ m\ l\ x)\end{array}}\ \text{ILIST-BLOCK\_VECTOR}$$

$$\frac{k \neq 0 \qquad l \neq 0 \qquad \mathit{ilist}_M\ (m * k)\ (n * l)\ A'\ A}{\begin{array}{c}\mathit{ilist}_M\ m\ n\ (\lambda i\ j.\ \mathit{block}\ k\ l\ (\lambda i'\ j'.A'(i * k + i')(j * l + j')))\\ (\mathit{block\_matrix}\ m\ n\ k\ l\ A)\end{array}}\ \text{ILIST-BLOCK\_MATRIX}$$

Figure 9. Introduction rules used for proving blocked format operations.

successful application of either sub-method, the result is prepended to the head of the sequence. A failure at any level causes the search to backtrack and continue with the next available goal state. When the top element of the goal state sequence is empty, the main goal has been discharged and the proof is complete.

5. Evaluation

In this section we evaluate programmability of sparse codes in LL and the extensibility of our verification method to new formats.

5.1 Verifying additional sparse formats

We examine to what extent our prover design allows us to verify additional formats without adding excessively many rules. Recall that our initial implementation of the prover for CSR SpMV (Section 4) insisted on minimizing reliance on format-specific rules, avoiding duplication of logic, and keeping representation relations general, for example by keeping the type of the value stored in the matrix parametric. In this section, we extend our prover to verify several formats that are strictly more complex than CSR.

Our experience indicates that our prover can overcome variations in both format construction and matrix-vector multiplication. The variations were both syntactic (*i.e.*, due to syntactic sugar) and structural (*i.e.*, inducing a different dataflow structure). This benefit is thanks to Isabelle's simplifier, which successfully canonicalizes these differences, requiring only minor tweaks to the prover's rule base. Therefore, we consider below only a single implementation for each format and argue that the single variant represents a larger class of similar implementations.

Jagged Diagonals (JAD). A prominent feature of JAD's proof goal is the double use of transpose, once during compression (`jad`) and once during multiplication (`jadmv`). This form can be simplified to the Isabelle *takeWhile* list operator on the premise that com-

pressed rows are sorted by length prior to being transposed. The form is matched by a rewrite rule for *transpose* (*transpose* xs). Adding introduction rules for *infl*, *takeWhile*, *rev* and *sort_key* was sufficient for our verifier to complete the proof.

The ability to prove full functional correctness of JAD SpMV documents the strength of our prover; no other verification framework that we know of can (i) handle the complex data transformations in JAD compression, and (ii) prove correctness of arithmetic operations on the resulting sparse representation (see Section 6).

Coordinate (COO). As mentioned in Section 3.2, the COO format is challenging because it associates matrix values with both row and column coordinates, and also because it requires concatenation and gather operations. It turns out that the COO pair coordinates do not call for a new representation relation. In fact, thanks to how the functions `coo` and `coomv` are composed, we need to handle the pair coordinates only between concatenation (in `coo`) and gather (in `coomv`). The simplifier moves these two functions together; therefore, we introduce a rule to relate the representation of the input and output of *gather* (*concat* xs), allowing the prover to automatically complete the proof.

ALIST-GATHER-CONCAT

$$\frac{\textit{vlist}\ n\ (\lambda i.\ \textit{vlist}\ m\ (\lambda j\ a\ b.\ a = \textit{snd}\ b\ \wedge i = \textit{fst}\ b)\ \{0\})\ \{x.\forall j < m.x\ j = 0\}\ M\ xs}{\textit{alist}\ n\ (\lambda i.\ \textit{vlist}\ m\ (\lambda i.\textit{op} =)\ \{0\})\ \{x.\forall j < m.x\ j = 0\}\ M\ (\textit{gather}\ (\textit{concat}\ xs))}$$

Compressed Sparse Columns (CSC). As CSC exhibits a peculiar use of concatenation and gather operations, it is handled similarly to COO. In contrast to COO, the input list to `concat` represents a transposed matrix, hence we use a rule similar to ALIST-GATHER-CONCAT, but with a transposed matrix M.

How many introduction rules did we need to prove our sparse formats? In total, 24 rules were needed, including both introduction and simplification rules. Introduction rules were typically used to (i) reason about some language construct such as **`map`**, `sum` and **`filter`**, in the context of a certain representation (*e.g.*, rules ILIST-MAP, ILIST-LISTSUM, ALIST-FILTER in Fig. 8); and (ii) formalize algebraic operations on vector and matrix representations, such as extracting an inner representation relation (ILIST-VLIST) and substituting a vector representation with a matrix representation ($\text{ILIST}_v \rightarrow \text{ILIST}_M$). Most operators were handled by a single introduction rule; a few (*e.g.*, *map*) required one rule per representation relation.

To quantify rule reuse in our prover, we summarize the reuse of the 24 rules that were needed for proving five sparse formats (see Fig. 10). On average, fewer than 19% of rules used by a particular format are specific to this format, while over 66% of these rules are used by at least three additional formats, a significant level of reuse. Even of the rules needed for more complex formats (CSC and JAD), only up to a third are format-specific. On the other hand, format-specific rules tend to be harder to prove, as indicated by the average number of lines of Isar code required to prove the rules. A detailed examination reveals that two rules for handling a *gather-concat* sequence (used in CSC and COO) account for over a hundred lines each. We believe that these rules can be refactored for better reuse of simpler lemmas and greater automation. Note that most of the effort in proving JAD was invested in stating and proving simplification of *transpose-transpose* composition. Fig. 10 does not account for these rules, as they are quite general and were implemented as an extension to Isabelle's theory of lists.

5.2 Case study: hierarchical compression formats

This subsection evaluates expressiveness of the LL language. This question is motivated by the absence from LL of some powerful constructs, such as first-class functions and folds. We show that LL can express advanced formats even without these constructs.

Sparse CSR (SCSR). The SCSR format extends CSR with another layer of compression. SCSR compresses the list of (compressed) rows, by filtering out empty rows (*i.e.*, those rows that have only zero-valued elements). The remaining rows are associated with their row index.

Implementing SCSR in LL amounts to obtaining the CSR format, which compresses individual rows, followed by compression of the resulting list of compressed rows. Again, LL manages to express format construction as a pipeline of stages.

```
def scsr: csr -> [len != 0 ? (#, id)]
```

The corresponding SpMV implementation needs to account for the row indexes. It must also inflate the resulting sparse vector into dense format:

```
def scsrmv(A, x):
 A ->
 [i, r: r -> unzip -> snd * x[fst] ->
  sum -> (i, id)] ->
 infl(0, m, id)
```

Alternatively, we can reuse SpMV for CSR:

```
A -> unzip -> (fst, csrmv(snd, x)) -> zip ->
infl(0, m, id)
```

SCSR demonstrates the ability of our prover to peel off the additional compression layer and prove correctness of the overall result, while requiring only two rules in addition to those needed by CSR (see Fig. 10).

Next, we investigate two optimizations for SpMV—register blocking and cache blocking—designed to improve temporal locality of the vector x at two different levels of the memory hierarchy. The locality is improved by reorganizing the computation to operate on smaller segments of the input matrix, which in turn allows the reuse of a segment of x.

Register blocking. This optimization is useful when the nonzero values in the matrix appear in clusters [14]. The idea is to place the cluster of nonzeros in a small dense matrix. To obtain register-blocked format, instead of compressing single-cell values (*i.e.*, numbers), a matrix is partitioned into uniformly sized rectangular blocks. These dense blocks form the base elements for compression: a block is filtered away if all its elements are zeros; if the block is nonzero, it is represented as a dense matrix. The size of these blocks is chosen so that the corresponding portion of the vector x can reside in registers during processing of a block.

Register blocking can be applied to all sparse formats described in Section 3.2. The 2×2 blocked representation of Fig. 1(a) can be seen in Fig. 11(a). Applying CSR compression to this blocked matrix results in the register-blocked CSR format in Fig. 11(b).

To construct the register-blocked CSR (RBCSR) in LL, we first "blockify" the dense matrix with the `block` function, which transforms a dense matrix A of size $mr \times nc$ to an $m \times n$ matrix of $r \times c$ dense blocks. Next, we pair these blocks with their column indices using `[enum]`, and filter out the all-zero blocks.

```
def rbcsr(A):
  block(r, c, A) ->
  [enum -> [snd -> [[neq ‘ 0] -> disj] -> disj ? ]]
```

In SpMV of an $r \times c$-RBCSR matrix A and a dense vector x, we first bind the names `B` and `l` to each dense block and its index, respectively, and perform dense matrix-vector multiplication (`densemv`) on each block and the corresponding c-sub-vector of x.

Reuse degree	# Rules	Avg. LOC	CSR	SCSR	COO	CSC	JAD	Total	%
1	11	28.3	1		2	5	3	11	18.3
2	3	6.3			3	3		6	10.0
3	1	6.0	1	1			1	3	5.0
4	5	6.4	3	5	4	3	5	20	33.3
5	4	2.3	4	4	4	4	4	20	33.3

Figure 10. Reusability analysis of our sparse-matrix code prover.

$$\begin{pmatrix} \begin{pmatrix} a & 0 \\ b & c \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \\ 0 & d \end{pmatrix} & \begin{pmatrix} 0 & 0 \\ 0 & e \end{pmatrix} \end{pmatrix} \quad \begin{pmatrix} \left(\left\langle 0, \begin{pmatrix} a & 0 \\ b & c \end{pmatrix}\right\rangle\right) & \\ \left(\left\langle 0, \begin{pmatrix} 0 & 0 \\ 0 & d \end{pmatrix}\right\rangle \left\langle 1, \begin{pmatrix} 0 & 0 \\ 0 & e \end{pmatrix}\right\rangle\right) & \end{pmatrix}$$

(a) (b)

Figure 11. Example dense and sparse 2×2 blocked matrix representations.

The latter is obtained by breaking x into a list of c-vectors using `block(c, x)` and selecting the appropriate sub-vector. The result vectors in a row block are summed, and the final result is obtained by concatenating the result sub-vectors from all row blocks.

```
def rbcsrmv(A, x):
  A ->
  [[l, B: (B, block(c, x)[l]) -> densemv] -> sum] ->
  concat
```

Our prover allowed us to easily extend proofs to blocked formats because our matrices are of parametric type; the prover can work with matrices of numbers as well as with matrices whose elements are matrices. Parameterization of matrices was expressed with Isabelle/HOL type classes, which are used to restrict types in introduction rules.

We use a theory of finite matrices [12]. Here, too, the size of a matrix is not encoded in the matrix type (denoted α matrix) but it is required that matrix dimensions are bounded. To represent matrices as abstract values, we introduce the *matrix* conversion function:

$$\mathit{matrix} :: \text{nat} \to \text{nat} \to (\text{nat} \to \text{nat} \to \alpha) \to \alpha \text{ matrix}$$

The first two parameters specify the row and column dimensions, respectively. The third parameter is the abstract value encoded into the matrix. Implementing functions on compressed matrices necessitates a few more conversion functions:

$$\mathit{block\_vector} :: \text{nat} \to \text{nat} \to [\alpha] \to [\alpha \text{ matrix}]$$

$$\mathit{block\_matrix} :: \text{nat} \to \text{nat} \to \text{nat} \to \text{nat} \to [[\alpha]] \to [[\alpha \text{ matrix}]]$$

$$\mathit{concat\_vectors} :: \text{nat} \to [\alpha \text{ matrix}] \to [\alpha]$$

The operation *block_matrix* $m\ n\ k\ l\ A$ transforms the object A, representing an $mk \times nl$-matrix, into an object representing an $m \times n$-matrix of $k \times l$-blocks; *block_vector* $m\ k\ x$ transforms the list x of length nk into a list of n k-vectors; *concat_vectors* $k\ x$ is the inverse operation, unpacking the k-vectors in x.

This code shows register-blocked CSR code in Isabelle.

$$\mathit{rbcsr}\ m\ n\ k\ l\ A = \mathit{map}\ (\mathit{filter}\ (\lambda(i,v).\ v \neq 0) \circ \mathit{enum})\ (\mathit{block\_matrix}\ m\ n\ k\ l\ A)$$

$$\mathit{rbcsrmv}\ m\ n\ k\ l\ (A,\ x) = \mathit{concat\_vectors}\ k\ (\mathit{map}\ (\mathit{listsum} \circ (\mathit{map}\ (\lambda v.\mathit{snd}\ v * \mathit{block\_vector}\ n\ l\ x\ !\ \mathit{fst}\ v)))\ A)$$

We require that block dimensions are greater than zero and properly divide the respective matrix and vector dimensions. The correctness theorem follows.

$$k \neq 0 \wedge l \neq 0 \wedge \mathit{ilist}_M\ (m*k)\ (n*l)\ A' \wedge \mathit{ilist}_v\ (n*l)\ x' \longrightarrow \mathit{ilist}_v\ (m*k)\ (\lambda i.\ \Sigma j < n*l.\ A'\ i\ j * x'\ j)\ (\mathit{rbcsrmv}\ m\ n\ k\ l\ (\mathit{rbcsr}\ m\ n\ k\ l\ A,\ x)) \tag{13}$$

After adding the introduction rules in Fig. 9 and a few rewrite rules for *matrix*, the prover automatically proves Eq. (13).

Cache blocking. The idea in cache blocking is to reduce cache misses for the source vector x when it is too large to entirely fit in the cache during SpMV. We consider *static cache blocking* [7]. The sparse matrix is partitioned into rectangular sub-matrices of size $r \times c$. While in register blocking these sub-matrices were kept dense, in the cache-blocked format they are compressed.

Our cache blocking scheme differs from the one in [7] in that we only allow cache blocks to start at column indices which are multiples of c; this restriction leads to suboptimal compression. We believe that this restriction can be relaxed by augmenting LL with a blocking function that creates optimally placed blocks.

Notice that the construction of a cache-blocked matrix is very similar to the construction for register blocking. The only difference is the additional compression applied to each block. The LL code for the CSR compressed cache-blocked matrix, whose blocks are stored in CSR format, is shown below.

```
def cbcsr(A):
  block(r, c, A) ->
  [enum -> [snd -> [[neq ‘ 0] -> disj] -> disj ? ] ->
   [l, B: (l, B -> csr)]]
```

The corresponding cache-blocked SpMV in LL:

```
def cbcsrmv(A, x):
  A ->
  [[l, B: (B, block(c, x)[l]) -> csrmv] -> sum] ->
  concat
```

In the cache-blocked SpMV, we again notice the similarity to the register-blocked SpMV. The two codes are identical except for the function used for multiplying a block by a vector. It is somewhat desirable to factor out these inner multiplications (`densemv` and `csrmv`) but this is not possible in LL. The reason is that LL does not support lambda abstraction, which would allow reuse of code common to register- and cache-blocked versions. We have refrained from enriching LL with first-order functions for now because this

allows for simpler verification and gives us broad verification coverage. We do not consider the absence of lambda abstraction a significant disadvantage because even optimized LL programs are small. In the future, we may decide to extend LL with a template mechanism that will be used to instantiate such hierarchical composition, allowing code reuse.

The verification of cache-blocked sparse formats was not yet implemented. We expect that the amount of work will not be substantial, based on our experience with other hierarchical formats.

6. Related Work

Specifying sparse matrix code Bernoulli [8, 9] is a system that synthesizes efficient low-level implementations of matrix operations given a description of the sparse format using relational algebra. This is impressive and permits rapid development of fast low level implementations. However, the functionality of the system was limited and it had limited impact. Instead, we are expressing formats using a functional programming language which can be mechanically verified. We believe that function-level programming provides the right level of abstraction for expressing the desired transformations. Moreover, LL can be embedded in existing call-by-value functional programming languages. Compiling LL into low-level code is a work in progress.

The synthesizer by Bik *et al.* [2] produces efficient implementations by replacing $A[i, j]$ in dense-matrix code with a representation function that maps to the corresponding sparse element; powerful compiler optimizations then yield efficient code.

Verifying sparse matrix code We are not aware of previous work on verifying full functional correctness of sparse matrix codes. We are not even aware of work that verified their memory safety without explicitly provided loop invariants. Our own attempts at verification included ESC/Java, TVLA and SAT-based bounded model checking, neither of which was satisfactory. Furthermore, neither of these tools was capable of proving higher-order properties like the ones we currently prove. This led us to raising the level of abstraction and deferring to purely functional programs where loops are replaced with comprehensions and specialized reduction operators.

Higher order verification Duan *et al.* [5] verified a set of block ciphers using the interactive theorem prover HOL-4. They proved that the decoding of an encoded text results in the original data. Their proofs are mostly done using inversion rules, namely rules of the form $f\ (f^{-1}\ x) = x$, and algebraic rules on bit-word identities. For the block ciphers used by AES and IDEA special rules where needed. The domain of block cipher verification does not seem to require more complicated rules than bit-word identities.

7. Conclusion

In this paper we showed how to raise the level of abstraction for sparse matrix programs from imperative code with loops to functional programs with comprehensions and limited reductions. We also developed an automated proof method for verifying a diverse range of sparse matrix formats and their SpMV operations. This was accomplished by introducing relations that map a sparse representation to the abstract (mathematical) one. Through a clever definition of these representation relations we were able to build a reusable set of simplification and introduction rules, which could be applied to a variety of computations.

We are currently working on the problem of compiling the functional code into an efficient C code. Deploying techniques from predecessor functional and data parallel languages, we already exhibit promising performance results with real-world sparse formats.

References

[1] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM (CACM)*, 21(8):613–641, 1978.

[2] A. J. C. Bik, P. Brinkhaus, P. M. W. Knijnenburg, and H. A. G. Wijshoff. The automatic generation of sparse primitives. *ACM Transactions on Mathematical Software*, 24(2):190–225, 1998.

[3] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM (CACM)*, 39(3):85–97, 1996.

[4] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, pages 10–18, New York, NY, USA, 2007. ACM.

[5] J. Duan, J. Hurd, G. Li, S. Owens, K. Slind, and J. Zhang. Functional correctness proofs of encryption algorithms. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 519–533, 2005.

[6] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Languages Design and Implementation*, pages 234–245, 2002.

[7] E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California, Berkeley, 2000.

[8] V. Kotlyar and K. Pingali. Sparse code generation for imperfectly nested loops with dependences. In *International Conference on Supercomputing (ICS)*, pages 188–195, 1997.

[9] V. Kotlyar, K. Pingali, and P. Stodghill. A relational approach to the compilation of sparse matrix programs. In *Euro-Par*, pages 318–327, 1997.

[10] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *International Conference on Supercomputing (ICS)*, pages 88–99, 2000.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[12] S. Obua. *Flyspeck II: The Basic Linear Programs*. PhD thesis, Technische Universität München, 2008.

[13] M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[14] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, 2004.

[15] M. Wenzel. *The Isabelle/Isar Implementation*. Technische Universität München. `http://isabelle.in.tum.de/doc/implementation.pdf`.

[16] M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *International Conference on Theorem Proving in Higher-Order Logics*, pages 305–320, 2004.