Sparse Matrices Correctness 2019 Paper

$$Value = \{Zero, Value_0, Value_1, \ldots, Value_n\}$$

Figure 1: The set of abstract values.

ACM Reference Format:
. 2019. Sparse Matrices Correctness 2019 Paper. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SPARSE MATRIX MODELS

In this section we introduce some basic modeling elements used throughout the study, and then we describe four models. First we describe an abstract model of a matrix that does not represent any specific data format, leaving out all implementation details and notions of sparsity. Then we describe three refinements of this model which introduce structure that supports a sparse representation: the DOK format, the ELL format, and the CSR format. For each matrix representation we model initialization, update, and matrix-vector multiplication. For each refinement, we provide representation invariants and abstraction functions to show that the refinement is sound.

By design, Alloy provides no means of working with floating point values. It includes integers, but in a limited scope, and so they are not useful when attempting to work with the complete integer set, as might be the case when considering values that could be stored in a sparse matrix. This is inconsequential, however, as we only aim to reason about the structural complexities of sparse matrix formats. Therefore, it is sufficient to create an abstract distinction between zero and non-zero values. Our models employ a Value signature, representing any numerical value, and a Zero signature, an extension of Value, that represents the value zero. Depending on the scope, this creates an abstract set of values, shown in Figure 1, that we can use to populate matrices in our models.

Throughout these models we make use of predicate overloading to define various predicates that can be applied universally. Overloading is enabled by Alloy's type system and type checker, which allow expressions to share a name as long as there is no ambiguity when resolving types. For example, the `rowInRange` and `colInRange` predicates are used to determine if a row or column index is within the bounds of some matrix. The predicate definitions for the three sparse matrix types considered in this study are shown in Figure **??**. The `rowInRange` predicate evaluates to true if the value `row` is greater than or equal to zero and less than the number of rows in, for example, the DOK matrix `d`, false otherwise. Similarly, the `colInRange` predicate evaluates to true if the value `col` is greather than or equal to zero and less than the number of columns in the DOK matrix `d`. Additional usage of overloading found in these models includes the representation invariant, which is always named `repInv`, and the abstraction function, which is always named `alpha`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Abstract Sparse Matrix

We begin with an abstract model of a two-dimensional mathematical matrix, sparse or otherwise, defined by the Matrix signature as shown in Figure **??**. There are three fields defined on the Matrix signature representing (1) the number of rows in the matrix, (2) the number of columns in the matrix, and (3) the values and their locations in the matrix. This model makes no assumptions about the contents or structure of the matrix, and will be the arbiter of correctness when determining if a refinement is sound.

The representation invariant is specified as a signature fact so that it is applied to every member of the Matrix signature. The representation invariant states that (1) the number of rows and columns in a matrix are each greater than or equal to zero, (2) all row, column indices fall within bounds, (3) the total number of values in the matrix is `rows` × `cols`, and (4) there is a value at every (i, j) index pair.

1.1.1 Matrix Initialization. The `init` predicate shown in Figure **??** is used to initialize an empty matrix. We consider the initialized state of a matrix to be one in which the number of rows and columns is defined and all values are zero.

1.1.2 Matrix Update. The `update` predicate shown in Figure **??** is used to perform an update. We consider a matrix update to be a transition in which a single value of a matrix is changed and the matrix does not change size. For all matrices, this includes four possible transitions: non-zero to non-zero, non-zero to zero, zero to non-zero, and zero to zero, or stutter.

1.2 ELL Format

ELL, coming soon.

1.3 CSR Format

The Compressed Sparse Row (CSR) format, shown in Figure **??**, uses three arrays to store a sparse matrix, one for values and two for integers. The value array contains non-zero values of the matrix ordered as they are traversed in a row-wise fashion. Column indices of each value are stored in

```
module matrix

open value
open util/integer

sig Matrix {
  rows, cols: Int,
  vals: Int → Int → lone Value
}

pred repInv [m: Matrix] {
  m.rows ≥ 0
  m.cols ≥ 0
  m.vals.univ = range[m.rows]→range[m.cols]
}

pred init [m: Matrix, nrows, ncols: Int] {
  nrows ≥ 0
  ncols ≥ 0
  m.rows = nrows
  m.cols = ncols
  m.vals = range[m.rows]→range[m.cols]→Zero
}

fun range [n: Int]: set Int {
  { i: Int | 0 ≤ i and i < n }
}

fun range [m, n: Int]: set Int {
  { i: Int | m ≤ i and i < n }
}
```

a separate array, while a third array stores the location in the values array that starts each row. We adopt the convention that the values array is named A, the column array is named JA, and the row index array is name IA.

The CSR format, making no assumptions about the sparsity structure of the matrix, is a general format capable of compressing any sparse matrix. The storage savings for this approach is significant, requiring $2nnz + n + 1$ storage locations[1] rather than $n \times m$. Memory locality is improved for row access over the DOK format, but the indirect addressing steps can have an impact on performance [13].

Need to describe the "get" predicate and why it is needed.

The abstraction function:

The representation invariant:

REFERENCES

[1] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, VictorEijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Web page," https://www.mcs.anl.gov/petsc, 2019. [Online]. Available: https://www.mcs.anl.gov/petsc

[2] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[3] A. Bik and H. Wijshoff, "Advanced compiler optimizations for sparse computations," *Journal of Parallel and Distributed Computing*, vol. 31, no. 1, pp. 14 – 24, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731585711410

[4] A. J. C. Bik and H. A. G. Wijshoff, "Automatic data structure selection and transformation for sparse matrix computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 2, pp. 109–126, Feb 1996.

[5] V. Kotlyar, K. Pingali, and P. Stodghill, "A relational approach to the compilation of sparse matrix programs," in *Euro-Par'97 Parallel Processing*, C. Lengauer, M. Griebl, and S. Gorlatch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 318–327.

[6] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[7] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1512–1542, 1994.

[8] J. Dingel and T. Filkorn, "Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving," in *Computer Aided Verification: 7th International Conference, CAV '95 Liège, Belgium, July 3–5, 1995 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 54–69. [Online]. Available: https://doi.org/10.1007/3-540-60045-0_40

[9] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *International Conference on Computer Aided Verification*. Springer, 1997, pp. 72–83.

[10] J. Woodcock, *Using Z : specification, refinement, and proof*. London New York: Prentice Hall, 1996.

[11] G. Arnold, J. Hlzl, A. Sinan Kksal, R. Bodik, and M. Sagiv, "Specifying and verifying sparse matrix codes," *ACM SIGPLAN Notices*, vol. 45, pp. 249–260, 09 2010.

[12] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[13] J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, Z. Bai, Ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000.

[14] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations - version 2," 1994.

[15] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy*: A general-purpose higher-order relational constraint solver," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 609–619.

[16] T. Dyer, "Article github repository," 2019. [Online]. Available: https://github.com/atdyer/alloy-lib

[1] where n is the number of rows, m is the number of columns, and nnz is the number of non-zero values