

# The GraphBLAS C API Specification <sup>†</sup>:

Version 2.0.1

[Scott: THIS IS A DRAFT VERION. Update acks and remove DRAFT before release.]

Benjamin Brock, Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira

Generated on 2022/11/14 at 08:31:24 EDT

<sup>†</sup>Based on *GraphBLAS Mathematics* by Jeremy Kepner

6 Copyright © 2017-2021 Carnegie Mellon University, The Regents of the University of California,  
7 through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from  
8 the U.S. Dept. of Energy), the Regents of the University of California (U.C. Davis and U.C.  
9 Berkeley), Intel Corporation, International Business Machines Corporation, and Massachusetts  
10 Institute of Technology Lincoln Laboratory.

11 Any opinions, findings and conclusions or recommendations expressed in this material are those of  
12 the author(s) and do not necessarily reflect the views of the United States Department of Defense,  
13 the United States Department of Energy, Carnegie Mellon University, the Regents of the University  
14 of California, Intel Corporation, or the IBM Corporation.

15 NO WARRANTY. THIS MATERIAL IS FURNISHED ON AN AS-IS BASIS. THE COPYRIGHT  
16 OWNERS AND/OR AUTHORS MAKE NO WARRANTIES OF ANY KIND, EITHER EX-  
17 PRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WAR-  
18 RANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RE-  
19 SULTS OBTAINED FROM USE OF THE MATERIAL. THE COPYRIGHT OWNERS AND/OR  
20 AUTHORS DO NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREE-  
21 DOM FROM PATENT, TRADE MARK, OR COPYRIGHT INFRINGEMENT.

22 Except as otherwise noted, this material is licensed under a Creative Commons Attribution 4.0  
23 license (<http://creativecommons.org/licenses/by/4.0/legalcode>), and examples are licensed under  
24 the BSD License (<https://opensource.org/licenses/BSD-3-Clause>).

# Contents

|    |  |           |
|----|--|-----------|
| 25 |  |           |
| 26 | List of Tables . . . . .   | 9         |
| 27 | List of Figures . . . . .  | 11        |
| 28 | Acknowledgments . . . . .  | 12        |
| 29 | <b>1 Introduction</b>  | <b>13</b> |
| 30 | <b>2 Basic concepts</b>  | <b>15</b> |
| 31 | 2.1 Glossary . . . . .   | 15        |
| 32 | 2.1.1 GraphBLAS API basic definitions . . . . .                            | 15        |
| 33 | 2.1.2 GraphBLAS objects and their structure . . . . .                      | 16        |
| 34 | 2.1.3 Algebraic structures used in the GraphBLAS . . . . .                 | 17        |
| 35 | 2.1.4 The execution of an application using the GraphBLAS C API . . . . .  | 18        |
| 36 | 2.1.5 GraphBLAS methods: behaviors and error conditions . . . . .          | 19        |
| 37 | 2.2 Notation . . . . .   | 21        |
| 38 | 2.3 Mathematical foundations . . . . .                                     | 22        |
| 39 | 2.4 GraphBLAS opaque objects . . . . .                                     | 23        |
| 40 | 2.5 Execution model . . . . .  | 24        |
| 41 | 2.5.1 Execution modes . . . . .  | 25        |
| 42 | 2.5.2 Multi-threaded execution . . . . .                                   | 26        |
| 43 | 2.6 Error model . . . . .  | 28        |
| 44 | <b>3 Objects</b>   | <b>31</b> |
| 45 | 3.1 Enumerations for <code>init()</code> and <code>wait()</code> . . . . . | 31        |
| 46 | 3.2 Indices, index arrays, and scalar arrays . . . . .                     | 31        |
| 47 | 3.3 Types (domains) . . . . .  | 32        |

|    |          |   |           |
|----|----------|---|-----------|
| 48 | 3.4      | Algebraic objects, operators and associated functions . . . . . | 33        |
| 49 | 3.4.1    | Operators . . . . .   | 34        |
| 50 | 3.4.2    | Monoids . . . . .   | 39        |
| 51 | 3.4.3    | Semirings . . . . .   | 39        |
| 52 | 3.5      | Collections . . . . .   | 43        |
| 53 | 3.5.1    | Scalars . . . . .   | 43        |
| 54 | 3.5.2    | Vectors . . . . .   | 43        |
| 55 | 3.5.3    | Matrices . . . . .  | 44        |
| 56 | 3.5.3.1  | External matrix formats . . . . .                               | 44        |
| 57 | 3.5.4    | Masks . . . . .   | 44        |
| 58 | 3.6      | Fields . . . . .  | 45        |
| 59 | 3.7      | Descriptors . . . . .   | 48        |
| 60 | 3.8      | GrB_Info return values . . . . .                                | 48        |
| 61 | <b>4</b> | <b>Methods</b>  | <b>53</b> |
| 62 | 4.1      | Context methods . . . . .                                       | 53        |
| 63 | 4.1.1    | init: Initialize a GraphBLAS context . . . . .                  | 53        |
| 64 | 4.1.2    | finalize: Finalize a GraphBLAS context . . . . .                | 54        |
| 65 | 4.1.3    | getVersion: Get the version number of the standard. . . . .     | 55        |
| 66 | 4.2      | Object methods . . . . .  | 55        |
| 67 | 4.2.1    | Query methods . . . . .   | 56        |
| 68 | 4.2.1.1  | get: Query the value of an object . . . . .                     | 56        |
| 69 | 4.2.1.2  | Descriptor_set: Set content of descriptor . . . . .             | 57        |
| 70 | 4.2.2    | Algebra methods . . . . .                                       | 58        |
| 71 | 4.2.2.1  | Type_new: Construct a new GraphBLAS (user-defined) type . . . . | 58        |
| 72 | 4.2.2.2  | UnaryOp_new: Construct a new GraphBLAS unary operator . . . .   | 59        |
| 73 | 4.2.2.3  | BinaryOp_new: Construct a new GraphBLAS binary operator . . .   | 61        |
| 74 | 4.2.2.4  | Monoid_new: Construct a new GraphBLAS monoid . . . . .          | 62        |
| 75 | 4.2.2.5  | Semiring_new: Construct a new GraphBLAS semiring . . . . .      | 63        |
| 76 | 4.2.2.6  | IndexUnaryOp_new: Construct a new GraphBLAS index unary op-     |           |
| 77 |          | erator [Scott: NEW CONTENT] . . . . .                           | 64        |

|     |          |  |    |
|-----|----------|--|----|
| 78  | 4.2.3    | Scalar methods . . . . .   | 66 |
| 79  | 4.2.3.1  | Scalar_new: Construct a new scalar . . . . .                         | 66 |
| 80  | 4.2.3.2  | Scalar_dup: Construct a copy of a GraphBLAS scalar . . . . .         | 67 |
| 81  | 4.2.3.3  | Scalar_clear: Clear/remove a stored value from a scalar . . . . .    | 68 |
| 82  | 4.2.3.4  | Scalar_nvals: Number of stored elements in a scalar . . . . .        | 69 |
| 83  | 4.2.3.5  | Scalar_setElement: Set the single element in a scalar . . . . .      | 70 |
| 84  | 4.2.3.6  | Scalar_extractElement: Extract a single element from a scalar. . . . | 71 |
| 85  | 4.2.4    | Vector methods . . . . .   | 73 |
| 86  | 4.2.4.1  | Vector_new: Construct new vector . . . . .                           | 73 |
| 87  | 4.2.4.2  | Vector_dup: Construct a copy of a GraphBLAS vector . . . . .         | 74 |
| 88  | 4.2.4.3  | Vector_resize: Resize a vector . . . . .                             | 75 |
| 89  | 4.2.4.4  | Vector_clear: Clear a vector . . . . .                               | 76 |
| 90  | 4.2.4.5  | Vector_size: Size of a vector . . . . .                              | 77 |
| 91  | 4.2.4.6  | Vector_nvals: Number of stored elements in a vector . . . . .        | 77 |
| 92  | 4.2.4.7  | Vector_build: Store elements from tuples into a vector . . . . .     | 78 |
| 93  | 4.2.4.8  | Vector_setElement: Set a single element in a vector . . . . .        | 80 |
| 94  | 4.2.4.9  | Vector_removeElement: Remove an element from a vector . . . . .      | 82 |
| 95  | 4.2.4.10 | Vector_extractElement: Extract a single element from a vector. . . . | 83 |
| 96  | 4.2.4.11 | Vector_extractTuples: Extract tuples from a vector . . . . .         | 85 |
| 97  | 4.2.5    | Matrix methods . . . . .   | 86 |
| 98  | 4.2.5.1  | Matrix_new: Construct new matrix . . . . .                           | 86 |
| 99  | 4.2.5.2  | Matrix_dup: Construct a copy of a GraphBLAS matrix . . . . .         | 88 |
| 100 | 4.2.5.3  | Matrix_diag: Construct a diagonal GraphBLAS matrix . . . . .         | 89 |
| 101 | 4.2.5.4  | Matrix_resize: Resize a matrix . . . . .                             | 90 |
| 102 | 4.2.5.5  | Matrix_clear: Clear a matrix . . . . .                               | 91 |
| 103 | 4.2.5.6  | Matrix_nrows: Number of rows in a matrix . . . . .                   | 92 |
| 104 | 4.2.5.7  | Matrix_ncols: Number of columns in a matrix . . . . .                | 92 |
| 105 | 4.2.5.8  | Matrix_nvals: Number of stored elements in a matrix . . . . .        | 93 |
| 106 | 4.2.5.9  | Matrix_build: Store elements from tuples into a matrix . . . . .     | 94 |
| 107 | 4.2.5.10 | Matrix_setElement: Set a single element in matrix . . . . .          | 96 |

|     |          |  |     |
|-----|----------|--|-----|
| 108 | 4.2.5.11 | Matrix_removeElement: Remove an element from a matrix . . . . .                        | 98  |
| 109 | 4.2.5.12 | Matrix_extractElement: Extract a single element from a matrix . . .                    | 99  |
| 110 | 4.2.5.13 | Matrix_extractTuples: Extract tuples from a matrix . . . . .                           | 101 |
| 111 | 4.2.5.14 | Matrix_exportHint: Provide a hint as to which storage format might                     |     |
| 112 |          | be most efficient for exporting a matrix . . . . .                                     | 103 |
| 113 | 4.2.5.15 | Matrix_exportSize: Return the array sizes necessary to export a                        |     |
| 114 |          | GraphBLAS matrix object . . . . .  | 104 |
| 115 | 4.2.5.16 | Matrix_export: Export a GraphBLAS matrix to a pre-defined format                       | 105 |
| 116 | 4.2.5.17 | Matrix_import: Import a matrix into a GraphBLAS object . . . . .                       | 107 |
| 117 | 4.2.5.18 | Matrix_serializeSize: Compute the serialize buffer size . . . . .                      | 109 |
| 118 | 4.2.5.19 | Matrix_serialize: Serialize a GraphBLAS matrix. . . . .                                | 110 |
| 119 | 4.2.5.20 | Matrix_deserialize: Deserialize a GraphBLAS matrix. . . . .                            | 111 |
| 120 | 4.2.6    | Descriptor methods . . . . .   | 112 |
| 121 | 4.2.6.1  | Descriptor_new: Create new descriptor . . . . .  | 112 |
| 122 | 4.2.6.2  | Descriptor_set: Set content of descriptor . . . . .                                    | 113 |
| 123 | 4.2.7    | free: Destroy an object and release its resources . . . . .                            | 114 |
| 124 | 4.2.8    | wait: Return once an object is either <i>complete</i> or <i>materialized</i> . . . . . | 116 |
| 125 | 4.2.9    | error: Retrieve an error string . . . . .  | 117 |
| 126 | 4.3      | GraphBLAS operations . . . . .   | 118 |
| 127 | 4.3.1    | mxm: Matrix-matrix multiply . . . . .  | 122 |
| 128 | 4.3.2    | vxm: Vector-matrix multiply . . . . .  | 127 |
| 129 | 4.3.3    | mxv: Matrix-vector multiply . . . . .  | 131 |
| 130 | 4.3.4    | eWiseMult: Element-wise multiplication . . . . .                                       | 135 |
| 131 | 4.3.4.1  | eWiseMult: Vector variant . . . . .  | 136 |
| 132 | 4.3.4.2  | eWiseMult: Matrix variant . . . . .  | 140 |
| 133 | 4.3.5    | eWiseAdd: Element-wise addition . . . . .  | 145 |
| 134 | 4.3.5.1  | eWiseAdd: Vector variant . . . . .   | 146 |
| 135 | 4.3.5.2  | eWiseAdd: Matrix variant . . . . .   | 150 |
| 136 | 4.3.6    | extract: Selecting sub-graphs . . . . .  | 156 |
| 137 | 4.3.6.1  | extract: Standard vector variant . . . . .   | 156 |
| 138 | 4.3.6.2  | extract: Standard matrix variant . . . . .   | 160 |

|     |          |  |            |
|-----|----------|--|------------|
| 139 | 4.3.6.3  | extract: Column (and row) variant . . . . .                                    | 165        |
| 140 | 4.3.7    | assign: Modifying sub-graphs . . . . .   | 170        |
| 141 | 4.3.7.1  | assign: Standard vector variant . . . . .                                      | 170        |
| 142 | 4.3.7.2  | assign: Standard matrix variant . . . . .                                      | 175        |
| 143 | 4.3.7.3  | assign: Column variant . . . . .   | 181        |
| 144 | 4.3.7.4  | assign: Row variant . . . . .  | 186        |
| 145 | 4.3.7.5  | assign: Constant vector variant[Scott: NEW CONTENT] . . . . .                  | 192        |
| 146 | 4.3.7.6  | assign: Constant matrix variant[Scott: NEW CONTENT] . . . . .                  | 197        |
| 147 | 4.3.8    | apply: Apply a function to the elements of an object . . . . .                 | 203        |
| 148 | 4.3.8.1  | apply: Vector variant . . . . .  | 203        |
| 149 | 4.3.8.2  | apply: Matrix variant . . . . .  | 208        |
| 150 | 4.3.8.3  | apply: Vector-BinaryOp variants[Scott: NEW CONTENT] . . . . .                  | 212        |
| 151 | 4.3.8.4  | apply: Matrix-BinaryOp variants[Scott: NEW CONTENT] . . . . .                  | 218        |
| 152 | 4.3.8.5  | apply: Vector index unary operator variant[Scott: NEW CONTENT] . . . . .       | 224        |
| 153 | 4.3.8.6  | apply: Matrix index unary operator variant[Scott: NEW CONTENT] . . . . .       | 229        |
| 154 | 4.3.9    | select: . . . . .  | 234        |
| 155 | 4.3.9.1  | select: Vector variant[Scott: NEW CONTENT] . . . . .                           | 234        |
| 156 | 4.3.9.2  | select: Matrix variant[Scott: NEW CONTENT] . . . . .                           | 239        |
| 157 | 4.3.10   | reduce: Perform a reduction across the elements of an object . . . . .         | 245        |
| 158 | 4.3.10.1 | reduce: Standard matrix to vector variant . . . . .                            | 245        |
| 159 | 4.3.10.2 | reduce: Vector-scalar variant[Scott: NEW CONTENT] . . . . .                    | 249        |
| 160 | 4.3.10.3 | reduce: Matrix-scalar variant[Scott: NEW CONTENT] . . . . .                    | 253        |
| 161 | 4.3.11   | transpose: Transpose rows and columns of a matrix . . . . .                    | 256        |
| 162 | 4.3.12   | kroncker: Kronecker product of two matrices . . . . .                          | 260        |
| 163 | <b>5</b> | <b>Nonpolymorphic interface[Scott: NEW CONTENT]</b>                            | <b>267</b> |
| 164 | <b>A</b> | <b>Revision history</b>  | <b>279</b> |
| 165 | <b>B</b> | <b>Non-opaque data format definitions</b>                                      | <b>285</b> |
| 166 | B.1      | GrB_Format: Specify the format for input/output of a GraphBLAS matrix. . . . . | 285        |
| 167 | B.1.1    | GrB_CSR_FORMAT . . . . .   | 285        |

|     |  |            |
|-----|--|------------|
| 168 | B.1.2 GrB_CSC_FORMAT . . . . .                                       | 286        |
| 169 | B.1.3 GrB_COO_FORMAT . . . . .                                       | 286        |
| 170 | <b>C Examples</b>  | <b>287</b> |
| 171 | C.1 Example: Level breadth-first search (BFS) in GraphBLAS . . . . . | 288        |
| 172 | C.2 Example: Level BFS in GraphBLAS using apply . . . . .            | 289        |
| 173 | C.3 Example: Parent BFS in GraphBLAS . . . . .                       | 290        |
| 174 | C.4 Example: Betweenness centrality (BC) in GraphBLAS . . . . .      | 291        |
| 175 | C.5 Example: Batched BC in GraphBLAS . . . . .                       | 293        |
| 176 | C.6 Example: Maximal independent set (MIS) in GraphBLAS . . . . .    | 295        |
| 177 | C.7 Example: Counting triangles in GraphBLAS . . . . .               | 297        |



# List of Tables

178

|     |      |   |     |
|-----|------|---|-----|
| 179 | 2.1  | Types of GraphBLAS opaque objects. . . . .  | 23  |
| 180 | 2.2  | Methods that forced completion prior to GraphBLAS v2.0. . . . .   | 28  |
| 181 | 3.1  | Enumeration literals and corresponding values input to various GraphBLAS methods. . . . .   | 32  |
| 182 | 3.2  | Predefined GrB_Type values. . . . .   | 33  |
| 183 | 3.3  | Operator input for relevant GraphBLAS operations. . . . .   | 34  |
| 184 | 3.4  | Properties and recipes for building GraphBLAS algebraic objects. . . . .  | 35  |
| 185 | 3.5  | Predefined unary and binary operators for GraphBLAS in C. . . . .   | 37  |
| 186 | 3.6  | Predefined index unary operators for GraphBLAS in C. . . . .  | 38  |
| 187 | 3.7  | Predefined monoids for GraphBLAS in C. . . . .  | 40  |
| 188 | 3.8  | Predefined “true” semirings for GraphBLAS in C. . . . .   | 41  |
| 189 | 3.9  | Other useful predefined semirings for GraphBLAS in C. . . . .   | 42  |
| 190 | 3.10 | GrB_Format enumeration literals and corresponding values for matrix import and export methods. . . . .  | 44  |
| 192 | 3.11 | Field values of type GrB_Field enumeration, corresponding types, and the objects which must implement that GrB_Field. Collection refers to GrB_Matrix, GrB_Vector, and GrB_Scalar, Algebraic refers to Operators, Monoids, and Semirings, while All refers to all GraphBLAS objects. Global fields are denoted by Global. All fields may be read, some may be written (denoted by W), and some are hints (denoted by H) which may be ignored by the implementation. . . . . | 47  |
| 198 | 3.12 | Descriptor types and literals for fields and values. . . . .  | 49  |
| 199 | 3.13 | Predefined GraphBLAS descriptors. . . . .   | 50  |
| 200 | 3.14 | Enumeration literals and corresponding values returned by GraphBLAS methods and operations. . . . .   | 51  |
| 202 | 4.1  | A mathematical notation for the fundamental GraphBLAS operations supported in this specification. . . . .   | 119 |
| 203 |      |   |     |

|     |      |   |     |
|-----|------|---|-----|
| 204 | 5.1  | Long-name, nonpolymorphic form of GraphBLAS methods. . . . .                | 267 |
| 205 | 5.2  | Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .    | 268 |
| 206 | 5.3  | Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .    | 269 |
| 207 | 5.4  | Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .    | 270 |
| 208 | 5.5  | Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .    | 271 |
| 209 | 5.6  | Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .    | 272 |
| 210 | 5.7  | Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .    | 273 |
| 211 | 5.8  | Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .    | 274 |
| 212 | 5.9  | Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW |     |
| 213 |      | CONTENT] . . . . .  | 275 |
| 214 | 5.10 | Long-name, nonpolymorphic form of GraphBLAS methods (continued).[Scott: NEW |     |
| 215 |      | CONTENT] . . . . .  | 276 |
| 216 | 5.11 | Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .    | 277 |

# 217 List of Figures

|     |   |     |
|-----|---|-----|
| 218 | 3.1 Hierarchy of algebraic object classes in GraphBLAS. . . . . | 43  |
| 219 | 4.1 Flowchart for the GraphBLAS operations. . . . .             | 120 |
| 220 | B.1 Data layout for CSR format. . . . .                         | 285 |
| 221 | B.2 Data layout for CSC format. . . . .                         | 286 |
| 222 | B.3 Data layout for COO format. . . . .                         | 286 |

## Acknowledgments

This document represents the work of the people who have served on the C API Subcommittee of the GraphBLAS Forum.

Those who served as C API Subcommittee members for GraphBLAS 2.0 are (in alphabetical order):

- Benjamin Brock (UC Berkeley)
- Aydin Buluç (Lawrence Berkeley National Laboratory)
- Timothy G. Mattson (Intel Corporation)
- Scott McMillan (Software Engineering Institute at Carnegie Mellon University)
- José Moreira (IBM Corporation)

Those who served as C API Subcommittee members for GraphBLAS 1.0 through 1.3 are (in alphabetical order):

- Aydin Buluç (Lawrence Berkeley National Laboratory)
- Timothy G. Mattson (Intel Corporation)
- Scott McMillan (Software Engineering Institute at Carnegie Mellon University)
- José Moreira (IBM Corporation)
- Carl Yang (UC Davis)

The GraphBLAS C API Specification is based upon work funded and supported in part by:

- NSF Graduate Research Fellowship under Grant No. DGE 1752814 and by the NSF under Award No. 1823034 with the University of California, Berkeley
- The Department of Energy Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231
- Intel Corporation
- Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute [DM-0003727, DM19-0929, DM21-0090]
- International Business Machines Corporation

The following people provided valuable input and feedback during the development of the specification (in alphabetical order): David Bader, Hollen Barmer, Bob Cook, Tim Davis, Jeremy Kepner, James Kitchen, Peter Kogge, Manoj Kumar, Roi Lipman, Andrew Mellinger, Maxim Naumov, Nancy M. Ott, Michel Pelletier, Gabor Szarnyas, Ping Tak Peter Tang, Erik Welch, Michael Wolf, Albert-Jan Yzelman.

# Chapter 1

## Introduction

The GraphBLAS standard defines a set of matrix and vector operations based on semiring algebraic structures. These operations can be used to express a wide range of graph algorithms. This document defines the C binding to the GraphBLAS standard. We refer to this as the *GraphBLAS C API* (Application Programming Interface).

The GraphBLAS C API is built on a collection of objects exposed to the C programmer as opaque data types. Functions that manipulate these objects are referred to as *methods*. These methods fully define the interface to GraphBLAS objects to create or destroy them, modify their contents, and copy the contents of opaque objects into non-opaque objects; the contents of which are under direct control of the programmer.

The GraphBLAS C API is designed to work with C99 (ISO/IEC 9899:199) extended with *static type-based* and *number of parameters-based* function polymorphism, and language extensions on par with the `_Generic` construct from C11 (ISO/IEC 9899:2011). Furthermore, the standard assumes programs using the GraphBLAS C API will execute on hardware that supports floating point arithmetic such as that defined by the IEEE 754 (IEEE 754-2008) standard.

The GraphBLAS C API assumes programs will run on a system that supports acquire-release memory orders. This is needed to support the memory models required for multithreaded execution as described in section 2.5.2.

Implementations of the GraphBLAS C API will target a wide range of platforms. We expect cases will arise where it will be prohibitive for a platform to support a particular type or a specific parameter for a method defined by the GraphBLAS C API. We want to encourage implementors to support the GraphBLAS C API even when such cases arise. Hence, an implementation may still call itself “conformant” as long as the following conditions hold.

- Every method and operation from chapter 4 is supported for the vast majority of cases.
- Any cases not supported must be documented as an implementation-defined feature of the GraphBLAS implementation. Unsupported cases must be caught as an API error (section 2.6) with the parameter `GrB_NOT_IMPLEMENTED` returned by the associated method call.
- It is permissible to omit the corresponding nonpolymorphic methods from chapter 5 when it

is not possible to express the signature of that method.

The number of allowed omitted cases is vague by design. We cannot anticipate the features of target platforms, on the market today or in the future, that might cause problems for the GraphBLAS specification. It is our expectation, however, that such omitted cases would be a minuscule fraction of the total combination of methods, types, and parameters defined by the GraphBLAS C API specification.

The remainder of this document is organized as follows:

- Chapter 2: Basic Concepts
- Chapter 3: Objects
- Chapter 4: Methods
- Chapter 5: Nonpolymorphic interface
- Appendix A: Revision history
- Appendix B: Non-opaque data format definitions
- Appendix C: Examples

## Chapter 2

# Basic concepts

The GraphBLAS C API is used to construct graph algorithms expressed “in the language of linear algebra.” Graphs are expressed as matrices, and the operations over these matrices are generalized through the use of a semiring algebraic structure.

In this chapter, we will define the basic concepts used to define the GraphBLAS C API. We provide the following elements:

- Glossary of terms and notation used in this document.
- Algebraic structures and associated arithmetic foundations of the API.
- Functions that appear in the GraphBLAS algebraic structures and how they are managed.
- Domains of elements in the GraphBLAS.
- Indices, index arrays, scalar arrays, and external matrix formats used to expose the contents of GraphBLAS objects.
- The GraphBLAS opaque objects.
- The execution and error models implied by the GraphBLAS C specification.
- Enumerations used by the API and their values.

## 2.1 Glossary

### 2.1.1 GraphBLAS API basic definitions

- *application*: A program that calls methods from the GraphBLAS C API to solve a problem.
- *GraphBLAS C API*: The application programming interface that fully defines the types, objects, literals, and other elements of the C binding to the GraphBLAS.

- *function*: Refers to a named group of statements in the C programming language. Methods, operators, and user-defined functions are typically implemented as C functions. When referring to the code programmers write, as opposed to the role of functions as an element of the GraphBLAS, they may be referred to as such.
- *method*: A function defined in the GraphBLAS C API that manipulates GraphBLAS objects or other opaque features of the implementation of the GraphBLAS API.
- *operator*: A function that performs an operation on the elements stored in GraphBLAS matrices and vectors.
- *GraphBLAS operation*: A mathematical operation defined in the GraphBLAS mathematical specification. These operations (not to be confused with *operators*) typically act on matrices and vectors with elements defined in terms of an algebraic semiring.

### 2.1.2 GraphBLAS objects and their structure

- *non-opaque datatype*: Any datatype that exposes its internal structure and can be manipulated directly by the user.
- *opaque datatype*: Any datatype that hides its internal structure and can be manipulated only through an API.
- *GraphBLAS object*: An instance of an *opaque datatype* defined by the *GraphBLAS C API* that is manipulated only through the GraphBLAS API. There are four kinds of GraphBLAS opaque objects: *domains* (i.e., types), *algebraic objects* (operators, monoids and semirings), *collections* (scalars, vectors, matrices and masks), and descriptors.
- *handle*: A variable that holds a reference to an instance of one of the GraphBLAS opaque objects. The value of this variable holds a reference to a GraphBLAS object but not the contents of the object itself. Hence, assigning a value to another variable copies the reference to the GraphBLAS object of one handle but not the contents of the object.
- *domain*: The set of valid values for the elements stored in a GraphBLAS *collection* or operated on by a GraphBLAS *operator*. Note that some GraphBLAS objects involve functions that map values from one or more input domains onto values in an output domain. These GraphBLAS objects would have multiple domains.
- *collection*: An opaque GraphBLAS object that holds a number of elements from a specified *domain*. Because these objects are based on an opaque datatype, an implementation of the GraphBLAS C API has the flexibility to optimize the data structures for a particular platform. GraphBLAS objects are often implemented as sparse data structures, meaning only the subset of the elements that have values are stored.
- *implied zero*: Any element that has a valid index (or indices) in a GraphBLAS vector or matrix but is not explicitly identified in the list of elements of that vector or matrix. From a mathematical perspective, an *implied zero* is treated as having the value of the zero element of the relevant monoid or semiring. However, GraphBLAS operations are purposefully defined



using set notation in such a way that it makes it unnecessary to reason about implied zeros. Therefore, this concept is not used in the definition of GraphBLAS methods and operators.

- *mask*: An internal GraphBLAS object used to control how values are stored in a method's output object. The mask exists only inside a method; hence, it is called an *internal opaque object*. A mask is formed from the elements of a collection object (vector or matrix) input as a mask parameter to a method. GraphBLAS allows two types of masks:
  1. In the default case, an element of the mask exists for each element that exists in the input collection object when the value of that element, when cast to a Boolean type, evaluates to `true`.
  2. In the *structure only* case, masks have structure but no values. The input collection describes a structure whereby an element of the mask exists for each element stored in the input collection regardless of its value.
- *complement*: The *complement* of a GraphBLAS mask,  $M$ , is another mask,  $M'$ , where the elements of  $M'$  are those elements from  $M$  that *do not* exist.

### 2.1.3 Algebraic structures used in the GraphBLAS

- *associative operator*: In an expression where a binary operator is used two or more times consecutively, that operator is *associative* if the result does not change regardless of the way operations are grouped (without changing their order). In other words, in a sequence of binary operations using the same associative operator, the legal placement of parenthesis does not change the value resulting from the sequence operations. Operators that are associative over infinitely precise numbers (e.g., real numbers) are not strictly associative when applied to numbers with finite precision (e.g., floating point numbers). Such non-associativity results, for example, from roundoff errors or from the fact some numbers can not be represented exactly as floating point numbers. In the GraphBLAS specification, as is common practice in computing, we refer to operators as *associative* when their mathematical definition over infinitely precise numbers is associative even when they are only approximately associative when applied to finite precision numbers.

No GraphBLAS method will imply a predefined grouping over any associative operators. Implementations of the GraphBLAS are encouraged to exploit associativity to optimize performance of any GraphBLAS method with this requirement. This holds even if the definition of the GraphBLAS method implies a fixed order for the associative operations.

- *commutative operator*: In an expression where a binary operator is used (usually two or more times consecutively), that operator is *commutative* if the result does not change regardless of the order the inputs are operated on.

No GraphBLAS method will imply a predefined ordering over any commutative operators. Implementations of the GraphBLAS are encouraged to exploit commutativity to optimize performance of any GraphBLAS method with this requirement. This holds even if the definition of the GraphBLAS method implies a fixed order for the commutative operations.

- *GraphBLAS operators*: Binary or unary operators that act on elements of GraphBLAS objects. *GraphBLAS operators* are used to express algebraic structures used in the GraphBLAS such as monoids and semirings. They are also used as arguments to several GraphBLAS methods. There are two types of *GraphBLAS operators*: (1) predefined operators found in Table 3.5 and (2) user-defined operators created using `GrB_UnaryOp_new()` or `GrB_BinaryOp_new()` (see Section 4.2.2).
- *monoid*: An algebraic structure consisting of one domain, an associative binary operator, and the identity of that operator. There are two types of GraphBLAS monoids: (1) predefined monoids found in Table 3.7 and (2) user-defined monoids created using `GrB_Monoid_new()` (see Section 4.2.2).
- *semiring*: An algebraic structure consisting of a set of allowed values (the *domain*), a commutative and associative binary operator called addition, a binary operator called multiplication (where multiplication distributes over addition), and identities over addition ( $0$ ) and multiplication ( $1$ ). The additive identity is an annihilator over multiplication.
- *GraphBLAS semiring*: is allowed to diverge from the mathematically rigorous definition of a *semiring* since certain combinations of domains, operators, and identity elements are useful in graph algorithms even when they do not strictly match the mathematical definition of a semiring. There are two types of *GraphBLAS semirings*: (1) predefined semirings found in Tables 3.8 and 3.9, and (2) user-defined semirings created using `GrB_Semiring_new()` (see Section 4.2.2).
- *index unary operator*: A variation of the unary operator that operates on elements of GraphBLAS vectors and matrices along with the index values representing their location in the objects. There are predefined index unary operators found in Table 3.6), and user-defined operators created using `GrB_IndexUnaryOp_new` (see Section 4.2.2).

#### 2.1.4 The execution of an application using the GraphBLAS C API

- *program order*: The order of the GraphBLAS method calls in a thread, as defined by the text of the program.
- *host programming environment*: The GraphBLAS specification defines an API. The functions from the API appear in a program. This program is written using a programming language and execution environment defined outside of the GraphBLAS. We refer to this programming environment as the “host programming environment”.
- *execution time*: time expended while executing instructions defined by a program. This term is specifically used in this specification in the context of computations carried out on behalf of a call to a GraphBLAS method.
- *sequence*: A GraphBLAS application uniquely defines a directed acyclic graph (DAG) of GraphBLAS method calls based on their program order. At any point in a program, the state of any GraphBLAS object is defined by a subgraph of that DAG. An ordered collection of GraphBLAS method calls in program order that defines that subgraph for a particular object is the *sequence* for that object.

- *complete*: A GraphBLAS object is complete when it can be used in a happens-before relationship with a method call that reads the variable on another thread. This concept is used when reasoning about memory orders in multithreaded programs. A GraphBLAS object defined on one thread that is complete can be safely used as an IN or INOUT argument in a method-call on a second thread assuming the method calls are correctly synchronized so the definition on the first thread *happens-before* it is used on the second thread. In blocking-mode, an object is complete after a GraphBLAS method call that writes to that object returns. In nonblocking-mode, an object is complete after a call to the `GrB_wait()` method with the `GrB_COMPLETE` parameter.
- *materialize*: A GraphBLAS object is materialized when it is (1) complete, (2) the computations defined by the sequence that define the object have finished (either fully or stopped at an error) and will not consume any additional computational resources, and (3) any errors associated with that sequence are available to be read according to the GraphBLAS error model. A GraphBLAS object that is never loaded into a non-opaque data structure may potentially never be materialized. This might happen, for example, if the operations associated with the object are fused or otherwise changed by the runtime system that supports the implementation of the GraphBLAS C API. An object can be materialized by a call to the `materialize` mode of the `GrB_wait()` method.
- *context*: An instance of the GraphBLAS C API implementation as seen by an application. An application can have only one context between the start and end of the application. A context begins with the first thread that calls `GrB_init()` and ends with the first thread to call `GrB_finalize()`. It is an error for `GrB_init()` or `GrB_finalize()` to be called more than one time within an application. The context is used to constrain the behavior of an instance of the GraphBLAS C API implementation and support various execution strategies. Currently, the only supported constraints on a context pertain to the mode of program execution.
- *program execution mode*: Defines how a GraphBLAS sequence executes, and is associated with the *context* of a GraphBLAS C API implementation. It is set by an application with its call to `GrB_init()` to one of two possible states. In *blocking mode*, GraphBLAS methods return after the computations complete and any output objects have been materialized. In *nonblocking mode*, a method may return once the arguments are tested as consistent with the method (i.e., there are no API errors), and potentially before any computation has taken place.

### 2.1.5 GraphBLAS methods: behaviors and error conditions

- *implementation-defined behavior*: Behavior that must be documented by the implementation and is allowed to vary among different compliant implementations.
- *undefined behavior*: Behavior that is not specified by the GraphBLAS C API. A conforming implementation is free to choose results delivered from a method whose behavior is undefined.
- *thread-safe*: Consider a function called from multiple threads with arguments that do not overlap in memory (i.e. the argument lists do not share memory). If the function is *thread-safe*

471 then it will behave the same when executed concurrently by multiple threads or sequentially  
472 on a single thread.

- 473 • *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as param-  
474 eters to a GraphBLAS method are dimension (or shape) compatible if they have the correct  
475 number of dimensions and sizes for each dimension to satisfy the rules of the mathematical def-  
476 inition of the operation associated with the method. If any *dimension compatibility* rule above  
477 is violated, execution of the GraphBLAS method ends and the GrB\_DIMENSION\_MISMATCH  
478 error is returned.
- 479 • *domain compatible*: Two domains for which values from one domain can be cast to values in  
480 the other domain as per the rules of the C language. In particular, domains from Table 3.2  
481 are all compatible with each other, and a domain from a user-defined type is only compatible  
482 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS  
483 method ends and the GrB\_DOMAIN\_MISMATCH error is returned.

## 2.2 Notation

| Notation   | Description   |
|--|---|
| $D_{out}, D_{in}, D_{in_1}, D_{in_2}$  | Refers to output and input domains of various GraphBLAS operators.  |
| $\mathbf{D}_{out}(*), \mathbf{D}_{in}(*),$<br>$\mathbf{D}_{in_1}(*), \mathbf{D}_{in_2}(*)$ | Evaluates to output and input domains of GraphBLAS operators (usually a unary or binary operator, or semiring).   |
| $\mathbf{D}(*)$  | Evaluates to the (only) domain of a GraphBLAS object (usually a monoid, vector, or matrix).   |
| $f$  | An arbitrary unary function, usually a component of a unary operator.   |
| $\mathbf{f}(F_u)$  | Evaluates to the unary function contained in the unary operator given as the argument.  |
| $\odot$  | An arbitrary binary function, usually a component of a binary operator.   |
| $\odot(*)$   | Evaluates to the binary function contained in the binary operator or monoid given as the argument.  |
| $\otimes$  | Multiplicative binary operator of a semiring.   |
| $\oplus$   | Additive binary operator of a semiring.   |
| $\otimes(S)$   | Evaluates to the multiplicative binary operator of the semiring given as the argument.  |
| $\oplus(S)$  | Evaluates to the additive binary operator of the semiring given as the argument.  |
| $\mathbf{0}(*)$  | The identity of a monoid, or the additive identity of a GraphBLAS semiring.   |
| $\mathbf{L}(*)$  | The contents (all stored values) of the vector or matrix GraphBLAS objects. For a vector, it is the set of (index, value) pairs, and for a matrix it is the set of (row, col, value) triples. |
| $\mathbf{v}(i)$ or $v_i$   | The $i^{th}$ element of the vector $\mathbf{v}$ .   |
| $\mathbf{size}(\mathbf{v})$  | The size of the vector $\mathbf{v}$ .   |
| $\mathbf{ind}(\mathbf{v})$   | The set of indices corresponding to the stored values of the vector $\mathbf{v}$ .  |
| $\mathbf{nrows}(\mathbf{A})$   | The number of rows in the $\mathbf{A}$ .  |
| $\mathbf{ncols}(\mathbf{A})$   | The number of columns in the $\mathbf{A}$ .   |
| $\mathbf{indrow}(\mathbf{A})$  | The set of row indices corresponding to rows in $\mathbf{A}$ that have stored values.   |
| $\mathbf{indcol}(\mathbf{A})$  | The set of column indices corresponding to columns in $\mathbf{A}$ that have stored values.   |
| $\mathbf{ind}(\mathbf{A})$   | The set of $(i, j)$ indices corresponding to the stored values of the matrix.   |
| $\mathbf{A}(i, j)$ or $A_{ij}$   | The element of $\mathbf{A}$ with row index $i$ and column index $j$ .   |
| $\mathbf{A}(:, j)$   | The $j^{th}$ column of matrix $\mathbf{A}$ .  |
| $\mathbf{A}(i, :)$   | The $i^{th}$ row of matrix $\mathbf{A}$ .   |
| $\mathbf{A}^T$   | The transpose of matrix $\mathbf{A}$ .  |
| $\neg \mathbf{M}$  | The complement of $\mathbf{M}$ .  |
| $\mathbf{s}(\mathbf{M})$   | The structure of $\mathbf{M}$ .   |
| $\tilde{\mathbf{t}}$   | A temporary object created by the GraphBLAS implementation.   |
| $< type >$   | A method argument type that is <code>void *</code> or one of the types from Table 3.2.  |
| <code>GrB_ALL</code>   | A method argument literal to indicate that all indices of an input array should be used.  |
| <code>GrB_Type</code>  | A method argument type that is either a user defined type or one of the types from Table 3.2.   |
| <code>GrB_Object</code>  | A method argument type referencing any of the GraphBLAS object types.   |
| <code>GrB_NULL</code>  | The GraphBLAS NULL.   |

## 2.3 Mathematical foundations

Graphs can be represented in terms of matrices. The values stored in these matrices correspond to attributes (often weights) of edges in the graph.<sup>1</sup> Likewise, information about vertices in a graph are stored in vectors. The set of valid values that can be stored in either matrices or vectors is referred to as their domain. Matrices are usually sparse because the lack of an edge between two vertices means that nothing is stored at the corresponding location in the matrix. Vectors may be sparse or dense, or they may start out sparse and become dense as algorithms traverse the graphs.

Operations defined by the GraphBLAS C API specification operate on these matrices and vectors to carry out graph algorithms. These GraphBLAS operations are defined in terms of GraphBLAS semiring algebraic structures. Modifying the underlying semiring changes the result of an operation to support a wide range of graph algorithms. Inside a given algorithm, it is often beneficial to change the GraphBLAS semiring that applies to an operation on a matrix. This has two implications for the C binding of the GraphBLAS API.

First, it means that we define a separate object for the semiring to pass into methods. Since in many cases the full semiring is not required, we also support passing monoids or even binary operators, which means the semiring is implied rather than explicitly stated.

Second, the ability to change semirings impacts the meaning of the *implied zero* in a sparse representation of a matrix or vector. This element in real arithmetic is zero, which is the identity of the *addition* operator and the annihilator of the *multiplication* operator. As the semiring changes, this implied zero changes to the identity of the *addition* operator and the annihilator (if present) of the *multiplication* operator for the new semiring. Nothing changes regarding what is stored in the sparse matrix or vector, but the implied zeros within them change with respect to a particular operation. In all cases, the nature of the implied zero does not matter since the GraphBLAS C API requires that implementations treat them as nonexistent elements of the matrix or vector.

As with matrices and vectors, GraphBLAS semirings have domains associated with their inputs and outputs. The semirings in the GraphBLAS C API are defined with two domains associated with the input operands and one domain associated with output. When used in the GraphBLAS C API these domains may not match the domains of the matrices and vectors supplied in the operations. In this case, only valid *domain compatible* casting is supported by the API.

The mathematical formalism for graph operations in the language of linear algebra often assumes that we can operate in the field of real numbers. However, the GraphBLAS C binding is designed for implementation on computers, which by necessity have a finite number of bits to represent numbers. Therefore, we require a conforming implementation to use floating point numbers such as those defined by the IEEE-754 standard (both single- and double-precision) wherever real numbers need to be represented. The practical implications of these finite precision numbers is that the result of a sequence of computations may vary from one execution to the next as the grouping of operands (because of associativity) within the operations changes. While techniques are known to reduce these effects, we do not require or even expect an implementation to use them as they may add

---

<sup>1</sup>More information on the mathematical foundations can be found in the following paper: J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. Owens, C. Yang, M. Zalewski, and T. Mattson. 2016, September. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)* (pp. 1-9). IEEE.

Table 2.1: Types of GraphBLAS opaque objects.

| GrB_Object types | Description  |
|------------------|--|
| GrB_Type         | Scalar type.   |
| GrB_UnaryOp      | Unary operator.  |
| GrB_IndexUnaryOp | Unary operator, that operates on a single value and its location index values.             |
| GrB_BinaryOp     | Binary operator.   |
| GrB_Monoid       | Monoid algebraic structure.  |
| GrB_Semiring     | A GraphBLAS semiring algebraic structure.  |
| GrB_Scalar       | One element; could be empty.   |
| GrB_Vector       | One-dimensional collection of elements; can be sparse.                                     |
| GrB_Matrix       | Two-dimensional collection of elements; typically sparse.                                  |
| GrB_Descriptor   | Descriptor object, used to modify behavior of methods (specifically GraphBLAS operations). |

considerable overhead. In most cases, these roundoff errors are not significant. When they are significant, the problem itself is ill-conditioned and needs to be reformulated.

## 2.4 GraphBLAS opaque objects

Objects defined in the GraphBLAS standard include types (the domains of elements), collections of elements (matrices, vectors, and scalars), operators on those elements (unary, index unary, and binary operators), algebraic structures (semirings and monoids), and descriptors. GraphBLAS objects are defined as opaque types; that is, they are managed, manipulated, and accessed solely through the GraphBLAS application programming interface. This gives an implementation of the GraphBLAS C specification flexibility to optimize objects for different scenarios or to meet the needs of different hardware platforms.

A GraphBLAS opaque object is accessed through its *handle*. A handle is a variable that references an instance of one of the types from Table 2.1. An implementation of the GraphBLAS specification has a great deal of flexibility in how these handles are implemented. All that is required is that the handle corresponds to a type defined in the C language that supports assignment and comparison for equality. The GraphBLAS specification defines a literal `GrB_INVALID_HANDLE` that is valid for each type. Using the logical equality operator from C, it must be possible to compare a handle to `GrB_INVALID_HANDLE` to verify that a handle is valid.

Every GraphBLAS object has a *lifetime*, which consists of the sequence of instructions executed in program order between the *creation* and the *destruction* of the object. The GraphBLAS C API predefines a number of these objects which are created when the GraphBLAS context is initialized by a call to `GrB_init` and are destroyed when the GraphBLAS context is terminated by a call to `GrB_finalize`.

An application using the GraphBLAS API can create additional objects by declaring variables of the appropriate type from Table 2.1 for the objects it will use. Before use, the object must be initialized

with a call to one of the object’s respective *constructor* methods. Each kind of object has at least one explicit constructor method of the form `GrB*_new` where ‘\*’ is replaced with the type of object (e.g., `GrB_Semiring_new`). Note that some objects, especially collections, have additional constructor methods such as duplication, import, or deserialization. Objects explicitly created by a call to a constructor should be destroyed by a call to `GrB_free`. The behavior of a program that calls `GrB_free` on a pre-defined object is undefined.

These constructor and destructor methods are the only methods that change the value of a handle. Hence, objects changed by these methods are passed into the method as pointers. In all other cases, handles are not changed by the method and are passed by value. For example, even when multiplying matrices, while the contents of the output product matrix changes, the handle for that matrix is unchanged.

Several GraphBLAS constructor methods take other objects as input arguments and use these objects to create a new object. For all these methods, the lifetime of the created object must end strictly before the lifetime of any dependent input objects. For example, a vector constructor `GrB_Vector_new` takes a `GrB_Type` object as input. That type object must not be destroyed until after the created vector is destroyed. Similarly, a `GrB_Semiring_new` method takes a monoid and a binary operator as inputs. Neither of these can be destroyed until after the created semiring is destroyed.

Note that some constructor methods like `GrB_Vector_dup` and `GrB_Matrix_dup` behave differently. In these cases, the input vector or matrix can be destroyed as soon as the call returns. However, the original type object used to create the input vector or matrix cannot be destroyed until after the vector or matrix created by `GrB_Vector_dup` or `GrB_Matrix_dup` is destroyed. This behavior must hold for any chain of duplicating constructors.

Programmers using GraphBLAS handles must be careful to distinguish between a handle and the object manipulated through a handle. For example, a program may declare two GraphBLAS objects of the same type, initialize one, and then assign it to the other variable. That assignment, however, only assigns the handle to the variable. It does not create a copy of that variable (to do that, one would need to use the appropriate duplication method). If later the object is freed by calling `GrB_free` with the first variable, the object is destroyed and the second variable is left referencing an object that no longer exists (a so-called “dangling handle”).

In addition to opaque objects manipulated through handles, the GraphBLAS C API defines an additional opaque object as an internal object; that is, the object is never exposed as a variable within an application. This opaque object is the mask used to control which computed values can be stored in the output operand of a *GraphBLAS operation*. Masks are described in Section 3.5.4.

## 2.5 Execution model

A program using the GraphBLAS C API is called a GraphBLAS application. The application constructs GraphBLAS objects, manipulates them to implement a graph algorithm, and then extracts values from the GraphBLAS objects to produce the results for that algorithm. Functions defined within the GraphBLAS C API that manipulate GraphBLAS objects are called *methods*. If the method corresponds to one of the operations defined in the GraphBLAS mathematical specifica-



tion, we refer to the method as an *operation*.

The GraphBLAS application specifies an ordered collection of GraphBLAS method calls defined by the order they appear in the text of the program (the *program order*). These define a directed acyclic graph (DAG) where nodes are GraphBLAS method calls and edges are dependencies between method calls.

Each method call in the DAG uniquely and unambiguously defines the output GraphBLAS objects as long as there are no execution errors that put objects in an invalid state (see Section 2.6). An ordered collection of method calls, a subgraph of the overall DAG for an application, defines the state of a GraphBLAS object at any point in a program. This ordered collection is the *sequence* for that object.

Since the GraphBLAS execution is defined in terms of a DAG and the GraphBLAS objects are opaque, the semantics of the GraphBLAS specification affords an implementation considerable flexibility to optimize performance. A GraphBLAS implementation can defer execution of nodes in the DAG, fuse nodes, or even replace whole subgraphs within the DAG to optimize performance. We discuss this topic further in section 2.5.1 when we describe *blocking* and *non-blocking* execution modes.

A correct GraphBLAS application must be *race-free*. This means that the DAG produced by an application and the results produced by execution of that DAG must be the same regardless of how the threads are scheduled for execution. It is the application programmer's responsibility to control memory orders and establish the required synchronized-with relationships to assure race-free execution of a multi-threaded GraphBLAS application. Writing race-free GraphBLAS applications is discussed further in Section 2.5.2.

### 2.5.1 Execution modes

The execution of the DAG defined by a GraphBLAS application depends on the *execution mode* of the GraphBLAS program. There are two modes: *blocking* and *nonblocking*.

- *blocking*: In blocking mode, each method finishes the GraphBLAS operation defined by the method and all output GraphBLAS objects are *materialized* before proceeding to the next statement. Even mechanisms that break the opaqueness of the GraphBLAS objects (e.g., performance monitors, debuggers, memory dumps) will observe that the operation has finished.
- *nonblocking*: In nonblocking mode, each method may return once the input arguments have been inspected and verified to define a well formed GraphBLAS operation. (That is, there are no API errors; see Section 2.6.) The GraphBLAS method may not have finished, but the output object is ready to be used by the next GraphBLAS method call. If needed, a call to `GrB_wait` with `GrB_COMPLETE` or `GrB_MATERIALIZE` can be used to force the sequence for a GraphBLAS object (obj) to finish its execution.

The *execution mode* is defined in the GraphBLAS C API when the context of the library invocation is defined. This occurs once before any GraphBLAS methods are called with a call to the

GrB\_init() function. This function takes a single argument of type GrB\_Mode with values shown in Table 3.1(a).

An application executing in nonblocking mode is not required to return immediately after input arguments have been verified. A conforming implementation of the GraphBLAS C API running in nonblocking mode may choose to execute *as if* in blocking mode. A sequence of operations in nonblocking mode where every GraphBLAS operation with output object `obj` is followed by a `GrB_wait(obj, GrB_MATERIALIZE)` call is equivalent to the same sequence in blocking mode with `GrB_wait(obj, GrB_MATERIALIZE)` calls removed.

Nonblocking mode allows for any execution strategy that satisfies the mathematical definition of the sequence. The methods can be placed into a queue and deferred. They can be chained together and fused (e.g., replacing a chained pair of matrix products with a matrix triple product). Lazy evaluation, greedy evaluation, and asynchronous execution are all valid as long as the final result agrees with the mathematical definition provided by the sequence of GraphBLAS method calls appearing in program order.

Blocking mode forces an implementation to carry out precisely the GraphBLAS operations defined by the methods and to complete each and every method call individually. It is valuable for debugging or in cases where an external tool such as a debugger needs to evaluate the state of memory during a sequence of operations.

In a sequence of operations free of execution errors, and with input objects that are well-conditioned, the results from blocking and nonblocking modes should be identical outside of effects due to roundoff errors associated with floating point arithmetic. Due to the great flexibility afforded to an implementation when using nonblocking mode, we expect execution of a sequence in nonblocking mode to potentially complete execution in less time.

It is important to note that, processing of nonopaque objects is never deferred in GraphBLAS. That is, methods that consume nonopaque objects (e.g., `GrB_Matrix_build()`, Section 4.2.5.9) and methods that produce nonopaque objects (e.g., `GrB_Matrix_extractTuples()`, Section 4.2.5.13) always finish consuming or producing those nonopaque objects before returning regardless of the execution mode.

Finally, after all GraphBLAS method calls have been made, the context is terminated with a call to `GrB_finalize()`. In the current version of the GraphBLAS C API, the context can be set only once in the execution of a program. That is, after `GrB_finalize()` is called, a subsequent call to `GrB_init()` is not allowed.

## 2.5.2 Multi-threaded execution

The GraphBLAS C API is designed to work with applications that utilize multiple threads executing within a shared address space. This specification does not define how threads are created, managed and synchronized. We expect the host programming environment to provide those services.

A conformant implementation of the GraphBLAS must be *thread safe*. A GraphBLAS library is thread safe when independent method calls (i.e., GraphBLAS objects are not shared between method calls) from multiple threads in a race-free program return the same results as would follow

from their sequential execution in some interleaved order. This is a common requirement in software libraries.

Thread safety applies to the behavior of multiple independent threads. In the more general case for multithreading, threads are not independent; they share variables and mix read and write operations to those variables across threads. A memory consistency model defines which values can be returned when reading an object shared between two or more threads. The GraphBLAS specification does not define its own memory consistency model. Instead the specification defines what must be done by a programmer calling GraphBLAS methods and by the implementor of a GraphBLAS library so an implementation of the GraphBLAS specification can work correctly with the memory consistency model for the host environment.

A memory consistency model is defined in terms of happens-before relations between methods in different threads. The defining case is a method that writes to an object on one thread that is read (i.e., used as an IN or INOUT argument) in a GraphBLAS method on a different thread. The following steps must occur between the different threads.

- A sequence of GraphBLAS methods results in the definition of the GraphBLAS object.
- The GraphBLAS object is put into a state of completion by a call to `GrB_wait()` with the `GrB_COMPLETE` parameter (see Table 3.1(b)). A GraphBLAS object is said to be *complete* when it can be safely used as an IN or INOUT argument in a GraphBLAS method call from a different thread.
- Completion happens before a synchronized-with relation that executes with *at least* a release memory order.
- A synchronized-with relation on the other thread executes with *at least* an acquire memory order.
- This synchronized-with relation happens-before the GraphBLAS method that reads the graph-BLAS object.

We use the phrase *at least* when talking about the memory orders to indicate that a stronger memory order such as *sequential consistency* can be used in place of the acquire-release order.

A program that violates these rules contains a data race. That is, its reads and writes are unordered across threads making the final value of a variable undefined. A program that contains a data race is invalid and the results of that program are undefined. We note that multi-threaded execution is compatible with both blocking and non-blocking modes of execution.

Completion is the central concept that allows GraphBLAS objects to be used in happens-before relations between threads. In earlier versions of GraphBLAS (1.X) completion was implied by any operation that produced non-opaque values from a GraphBLAS object. These operations are summarized in Table 2.2). In GraphBLAS 2.0, these methods no longer imply completion. This change was made since there are cases where the non-opaque value is needed but the object from which it is computed is not. We want implementations of the GraphBLAS to be able to exploit this case and not form the opaque object when that object is not needed.

Table 2.2: Methods that extract values from a GraphBLAS object that forcing completion of the operations contributing to that particular object in GraphBLAS 1.X. In GraphBLAS 2.0, these methods *do not* force completion.

| Method                                   | Section  |
|--|----------|
| GrB_Vector_nvals                         | 4.2.4.6  |
| GrB_Vector_extractElement                | 4.2.4.10 |
| GrB_Vector_extractTuples                 | 4.2.4.11 |
| GrB_Matrix_nvals                         | 4.2.5.8  |
| GrB_Matrix_extractElement                | 4.2.5.12 |
| GrB_Matrix_extractTuples                 | 4.2.5.13 |
| GrB_reduce (vector-scalar value variant) | 4.3.10.2 |
| GrB_reduce (matrix-scalar value variant) | 4.3.10.3 |

## 2.6 Error model

All GraphBLAS methods return a value of type `GrB_Info` (an enum) to provide information available to the system at the time the method returns. The returned value will be one of the defined values shown in Table 3.14. The return values fall into three groups: informational, API errors, and execution errors. While API and execution errors take on negative values, informational return values listed in Table 3.14(a) are non-negative and include `GrB_SUCCESS` (a value of 0) and `GrB_NO_VALUE`.

An API error (listed in Table 3.14(b)) means that a GraphBLAS method was called with parameters that violate the rules for that method. These errors are restricted to those that can be determined by inspecting the dimensions and domains of GraphBLAS objects, GraphBLAS operators, or the values of scalar parameters fixed at the time a method is called. API errors are deterministic and consistent across platforms and implementations. API errors are never deferred, even in nonblocking mode. That is, if a method is called in a manner that would generate an API error, it always returns with the appropriate API error value. If a GraphBLAS method returns with an API error, it is guaranteed that none of the arguments to the method (or any other program data) have been modified. The informational return value, `GrB_NO_VALUE`, is also deterministic and never deferred in nonblocking mode.

Execution errors (listed in Table 3.14(c)) indicate that something went wrong during the execution of a legal GraphBLAS method invocation. Their occurrence may depend on specifics of the execution environment and data values being manipulated. This does not mean that execution errors are the fault of the GraphBLAS implementation. For example, a memory leak could arise from an error in an application’s source code (a “program error”), but it may manifest itself in different points of a program’s execution (or not at all) depending on the platform, problem size, or what else is running at that time. Index out-of-bounds errors, for example, always indicate a program error.

If a GraphBLAS method returns with any execution error other than `GrB_PANIC`, it is guaranteed that the state of any argument used as input-only is unmodified. Output arguments may be left in an invalid state, and their use downstream in the program flow may cause additional errors. If a

731 GraphBLAS method returns with a `GrB_PANIC` execution error, no guarantees can be made about  
732 the state of any program data.

733 In nonblocking mode, execution errors can be deferred. A return value of `GrB_SUCCESS` only  
734 guarantees that there are no API errors in the method invocation. If an execution error value is  
735 returned by a method with output object `obj` in nonblocking mode, it indicates that an error was  
736 found during execution of any of the pending operations on `obj`, up to and including the `GrB_wait()`  
737 method (Section 4.2.8) call that completes those pending operations. When possible, that return  
738 value will provide information concerning the cause of the error.

739 As discussed in Section 4.2.8, a `GrB_wait(obj)` on a specific GraphBLAS object `obj` completes all  
740 pending operations on that object. No additional errors on the methods that precede the call to  
741 `GrB_wait` and have `obj` as an `OUT` or `INOUT` argument can be reported. From a GraphBLAS  
742 perspective, those methods are *complete*. Details on the guaranteed state of objects after a call to  
743 `GrB_wait` can be found in Section 4.2.8.

744 After a call to any GraphBLAS method that modifies an opaque object, the program can re-  
745 trieve additional error information (beyond the error code returned by the method) though a call  
746 to the function `GrB_error()`, passing the method's output object as described in Section 4.2.9.  
747 The function returns a pointer to a NULL-terminated string, and the contents of that string are  
748 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error  
749 string. `GrB_error()` is a thread-safe function, in the sense that multiple threads can call it simul-  
750 taneously and each will get its own error string back, referring to the object passed as an input  
751 argument.



## Chapter 3

# Objects

In this chapter, all of the enumerations, literals, data types, and predefined opaque objects defined in the GraphBLAS API are presented. Enumeration literals in GraphBLAS are assigned specific values to ensure compatibility between different runtime library implementations. The chapter starts by defining the enumerations that are used by the `init()` and `wait()` methods. Then a number of transparent (i.e., non-opaque) types that are used for interfacing with external data are defined. Sections that follow describe the various types of opaque objects in GraphBLAS: types (or *domains*), algebraic objects, collections and descriptors. Each of these sections also lists the predefined instances of each opaque type that are required by the API. This chapter concludes with a section on the definition for `GrB_Info` enumeration that is used as the return type of all methods.

### 3.1 Enumerations for `init()` and `wait()`

Table 3.1 lists the enumerations and the corresponding values used in the `GrB_init()` method to set the execution mode and in the `GrB_wait()` method for completing or materializing opaque objects.

### 3.2 Indices, index arrays, and scalar arrays

In order to interface with third-party software (i.e., software other than an implementation of the GraphBLAS), operations such as `GrB_Matrix_build` (Section 4.2.5.9) and `GrB_Matrix_extractTuples` (Section 4.2.5.13) must specify how the data should be laid out in non-opaque data structures. To this end we explicitly define the types for indices and the arrays used by these operations.

For indices a `typedef` is used to give a GraphBLAS name to a concrete type. We define it as follows:

```
typedef uint64_t GrB_Index;
```

The range of valid values for a variable of type `GrB_Index` is `[0, GrB_INDEX_MAX]` where the largest index value permissible is defined with a macro, `GrB_INDEX_MAX`. For example:

775 `#define GrB_INDEX_MAX ((GrB_Index) 0xffffffffffffffff);`

776 An implementation is required to define and document this value.

777 An index array is a pointer to a set of `GrB_Index` values that are stored in a contiguous block of  
 778 memory (i.e., `GrB_Index*`). Likewise, a scalar array is a pointer to a contiguous block of memory  
 779 storing a number of scalar values as specified by the user. Some GraphBLAS operations (e.g.,  
 780 `GrB_assign`) include an input parameter with the type of an index array. This input index array  
 781 selects a subset of elements from a GraphBLAS vector or matrix object to be used in the operation.  
 782 In these cases, the literal `GrB_ALL` can be used in place of the index array input parameter to  
 783 indicate that all indices of the associated GraphBLAS vector or matrix object should be used. An  
 784 implementation of the GraphBLAS C API has considerable freedom in terms of how `GrB_ALL`  
 785 is defined. Since `GrB_ALL` is used as an argument for an array parameter, it must use a type  
 786 consistent with a pointer. `GrB_ALL` must also have a non-null value to distinguish it from the  
 787 erroneous case of passing a `NULL` pointer as an array.

### 788 3.3 Types (domains)

789 In GraphBLAS, domains correspond to the valid values for types from the host language (in our  
 790 case, the C programming language). GraphBLAS defines a number of operators that take elements  
 791 from one or more domains and produce elements of a (possibly) different domain. GraphBLAS  
 792 also defines three kinds of collections: matrices, vectors and scalars. For any given collection, the  
 793 elements of the collection belong to a *domain*, which is the set of valid values for the elements. For  
 794 any variable or object  $V$  in GraphBLAS we denote as  $\mathbf{D}(V)$  the domain of  $V$ , that is, the set of  
 795 possible values that elements of  $V$  can take.

---

Table 3.1: Enumeration literals and corresponding values input to various GraphBLAS methods.

(a) `GrB_Mode` execution modes for the `GrB_init` method.

| Symbol                       | Value | Description                             |
|------------------------------|-------|---|
| <code>GrB_NONBLOCKING</code> | 0     | Specifies the nonblocking mode context. |
| <code>GrB_BLOCKING</code>    | 1     | Specifies the blocking mode context.    |

(b) `GrB_WaitMode` wait modes for the `GrB_wait` method.

| Symbol                       | Value | Description   |
|------------------------------|-------|---|
| <code>GrB_COMPLETE</code>    | 0     | The object is in a state where it can be used in a happens-before relation so that multithreaded programs can be properly synchronized. |
| <code>GrB_MATERIALIZE</code> | 1     | The object is <i>complete</i> , and in addition, all computation of the object is finished and any error information is available.      |

---



Table 3.2: Predefined `GrB_Type` values, and the corresponding GraphBLAS domain suffixes, C type (for scalar parameters), and domains for GraphBLAS. The domain suffixes are used in place of  $I$ ,  $F$ , and  $T$  in Tables 3.5, 3.6, 3.7, 3.8, and 3.9).

| GrB_Type   | Suffix | C type   | Domain                              |
|------------|--------|----------|-------------------------------------|
| GrB_BOOL   | BOOL   | bool     | {false, true}                       |
| GrB_INT8   | INT8   | int8_t   | $\mathbb{Z} \cap [-2^7, 2^7)$       |
| GrB_UINT8  | UINT8  | uint8_t  | $\mathbb{Z} \cap [0, 2^8)$          |
| GrB_INT16  | INT16  | int16_t  | $\mathbb{Z} \cap [-2^{15}, 2^{15})$ |
| GrB_UINT16 | UINT16 | uint16_t | $\mathbb{Z} \cap [0, 2^{16})$       |
| GrB_INT32  | INT32  | int32_t  | $\mathbb{Z} \cap [-2^{31}, 2^{31})$ |
| GrB_UINT32 | UINT32 | uint32_t | $\mathbb{Z} \cap [0, 2^{32})$       |
| GrB_INT64  | INT64  | int64_t  | $\mathbb{Z} \cap [-2^{63}, 2^{63})$ |
| GrB_UINT64 | UINT64 | uint64_t | $\mathbb{Z} \cap [0, 2^{64})$       |
| GrB_FP32   | FP32   | float    | IEEE 754 binary32                   |
| GrB_FP64   | FP64   | double   | IEEE 754 binary64                   |

The domains for elements that can be stored in collections and operated on through GraphBLAS methods are defined by GraphBLAS objects called `GrB_Type`. The predefined types and corresponding domains used in the GraphBLAS C API are shown in Table 3.2. The Boolean type (`bool`) is defined in `stdbool.h`, the integral types (`int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`) are defined in `stdint.h`, and the floating-point types (`float`, `double`) are native to the language and platform and in most cases defined by the IEEE-754 standard.

### 3.4 Algebraic objects, operators and associated functions

GraphBLAS operators operate on elements stored in GraphBLAS collections. A *binary operator* is a function that maps two input values to one output value. A *unary operator* is a function that maps one input value to one output value. Binary operators are defined over two input domains and produce an output from a (possibly different) third domain. Unary operators are specified over one input domain and produce an output from a (possibly different) second domain.

In addition to the operators that operate on stored values, GraphBLAS also supports *index unary operators* that maps a stored value and the indices of its position in the matrix or vector to an output value. That output value can be used in the index unary operator variants of `apply` (§ 4.3.8) to compute a new stored value, or be used in the `select` operation (§ 4.3.9) to determine if the stored input value should be kept or annihilated.

Some GraphBLAS operations require a monoid or semiring. A monoid contains an associative binary operator where the input and output domains are the same. The monoid also includes an identity value of the operator. The semiring consists of a binary operator – referred to as the “times” operator – with up to three different domains (two inputs and one output) and a monoid

Table 3.3: Operator input for relevant GraphBLAS operations. The semiring add and times are shown if applicable.

| Operation                        | Operator input  |
|----------------------------------|---|
| mxm, mxv, vxm                    | semiring  |
| eWiseAdd                         | binary operator<br>monoid<br>semiring (add)                           |
| eWiseMult                        | binary operator<br>monoid<br>semiring (times)                         |
| reduce (to vector or GrB_Scalar) | binary operator<br>monoid   |
| reduce (to scalar value)         | monoid  |
| apply                            | unary operator<br>binary operator with scalar<br>index unary operator |
| select                           | index unary operator  |
| kronecker                        | binary operator<br>monoid<br>semiring                                 |
| dup argument (build methods)     | binary operator   |
| accum argument (various methods) | binary operator   |

– referred to as the “plus” operator – that is also commutative. Furthermore, the domain of the monoid must be the same as the output domain of the “times” operator.

The GraphBLAS *algebraic objects* operators, monoids, and semirings are presented in this section. These objects can be used as input arguments to various GraphBLAS operations, as shown in Table 3.3. The specific rules for each algebraic object are explained in the respective sections of those objects. A summary of the properties and recipes for building these GraphBLAS algebraic objects is presented in Table 3.4.

A number of predefined operators are specified by the GraphBLAS C API. They are presented in tables in their respective subsections below. Each of these operators is defined to operate on specific GraphBLAS types and therefore, this type is built into the name of the object as a suffix. These suffixes and the corresponding predefined GrB\_Type objects that are listed in Table 3.2.

### 3.4.1 Operators

A GraphBLAS *unary operator*  $F_u = \langle D_{out}, D_{in}, f \rangle$  is defined by two domains,  $D_{out}$  and  $D_{in}$ , and an operation  $f : D_{in} \rightarrow D_{out}$ . For a given GraphBLAS unary operator  $F_u = \langle D_{out}, D_{in}, f \rangle$ , we define  $\mathbf{D}_{out}(F_u) = D_{out}$ ,  $\mathbf{D}_{in}(F_u) = D_{in}$ , and  $\mathbf{f}(F_u) = f$ .

A GraphBLAS *binary operator*  $F_b = \langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$  is defined by three domains,  $D_{out}$ ,  $D_{in_1}$ ,

---

Table 3.4: Properties and recipes for building GraphBLAS algebraic objects: unary operator, binary operator, monoid, and semiring (composed of operations *add* and *times*).

(a) Properties of algebraic objects.

| Object          | Must be commutative | Must be associative | Identity must exist | Number of domains |
|-----------------|---------------------|---------------------|---------------------|-------------------|
| Unary operator  | n/a                 | n/a                 | n/a                 | 2                 |
| Binary operator | no                  | no                  | no                  | 3                 |
| Monoid          | no                  | yes                 | yes                 | 1                 |
| Reduction add   | yes                 | yes                 | yes (see Note 1)    | 1                 |
| Semiring add    | yes                 | yes                 | yes                 | 1                 |
| Semiring times  | no                  | no                  | no                  | 3 (see Note 2)    |

(b) Recipes for algebraic objects.

| Object          | Recipe                                    | Number of domains |
|-----------------|---|-------------------|
| Unary operator  | Function pointer                          | 2                 |
| Binary operator | Function pointer                          | 3                 |
| Monoid          | Associative binary operator with identity | 1                 |
| Semiring        | Commutative monoid + binary operator      | 3                 |

Note 1: Some high-performance GraphBLAS implementations may require an identity to perform reductions to sparse objects like GraphBLAS vectors and scalars. According to the descriptions of the corresponding GraphBLAS operations, however, this identity is mathematically not necessary. There are API signatures to support both.

Note 2: The output domain of the semiring times must be same as the domain of the semiring’s add monoid. This ensures three domains for a semiring rather than four.

---

834  $D_{in_2}$ , and an operation  $\odot : D_{in_1} \times D_{in_2} \rightarrow D_{out}$ . For a given GraphBLAS binary operator  $F_b =$   
835  $\langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$ , we define  $\mathbf{D}_{out}(F_b) = D_{out}$ ,  $\mathbf{D}_{in_1}(F_b) = D_{in_1}$ ,  $\mathbf{D}_{in_2}(F_b) = D_{in_2}$ , and  $\odot(F_b) =$   
836  $\odot$ . Note that  $\odot$  could be used in place of either  $\oplus$  or  $\otimes$  in other methods and operations.

837 A GraphBLAS *index unary operator*  $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB\_Index}), D_{in_2}, f_i \rangle$  is defined by three  
838 domains,  $D_{out}$ ,  $D_{in_1}$ ,  $D_{in_2}$ , the domain of GraphBLAS indices, and an operation  $f_i : D_{in_1} \times I_{U64}^2 \times$   
839  $D_{in_2} \rightarrow D_{out}$  (where  $I_{U64}$  corresponds to the domain of a `GrB_Index`). For a given GraphBLAS  
840 index operator  $F_i$ , we define  $\mathbf{D}_{out}(F_i) = D_{out}$ ,  $\mathbf{D}_{in_1}(F_i) = D_{in_1}$ ,  $\mathbf{D}_{in_2}(F_i) = D_{in_2}$ , and  $\mathbf{f}(F_i) = f_i$ .

841 User-defined operators can be created with calls to `GrB_UnaryOp_new`, `GrB_BinaryOp_new`, and  
842 `GrB_IndexUnaryOp_new`, respectively. See Section 4.2.2 for information on these methods. The  
843 GraphBLAS C API predefines a number of these operators. These are listed in Tables 3.5 and 3.6.  
844 Note that most entries in these tables represent a “family” of predefined operators for a set of  
845 different types represented by the  $T$ ,  $I$ , or  $F$  in their names. For example, the multiplicative  
846 inverse (`GrB_MINV_F`) function is only defined for floating-point types ( $F = \text{FP32}$  or  $\text{FP64}$ ). The  
847 division (`GrB_DIV_T`) function is defined for all types, but only if  $y \neq 0$  for integral and floating  
848 point types and  $y \neq \text{false}$  for the Boolean type.

Table 3.5: Predefined unary and binary operators for GraphBLAS in C. The  $T$  can be any suffix from Table 3.2,  $I$  can be any integer suffix from Table 3.2, and  $F$  can be any floating-point suffix from Table 3.2.

| Operator type | GraphBLAS identifier | Domains  | Description                                      |
|---------------|----------------------|--|--|
| GrB_UnaryOp   | GrB_IDENTITY_ $T$    | $T \rightarrow T$  | $f(x) = x$ , identity                            |
| GrB_UnaryOp   | GrB_ABS_ $T$         | $T \rightarrow T$  | $f(x) =  x $ , absolute value                    |
| GrB_UnaryOp   | GrB_AINV_ $T$        | $T \rightarrow T$  | $f(x) = -x$ , additive inverse                   |
| GrB_UnaryOp   | GrB_MINV_ $F$        | $F \rightarrow F$  | $f(x) = \frac{1}{x}$ , multiplicative inverse    |
| GrB_UnaryOp   | GrB_LNOT             | $\text{bool} \rightarrow \text{bool}$                    | $f(x) = \neg x$ , logical inverse                |
| GrB_UnaryOp   | GrB_BNOT_ $I$        | $I \rightarrow I$  | $f(x) = \sim x$ , bitwise complement             |
| GrB_BinaryOp  | GrB_LOR              | $\text{bool} \times \text{bool} \rightarrow \text{bool}$ | $f(x, y) = x \vee y$ , logical OR                |
| GrB_BinaryOp  | GrB_LAND             | $\text{bool} \times \text{bool} \rightarrow \text{bool}$ | $f(x, y) = x \wedge y$ , logical AND             |
| GrB_BinaryOp  | GrB_LXOR             | $\text{bool} \times \text{bool} \rightarrow \text{bool}$ | $f(x, y) = x \oplus y$ , logical XOR             |
| GrB_BinaryOp  | GrB_LXNOR            | $\text{bool} \times \text{bool} \rightarrow \text{bool}$ | $f(x, y) = \overline{x \oplus y}$ , logical XNOR |
| GrB_BinaryOp  | GrB_BOR_ $I$         | $I \times I \rightarrow I$                               | $f(x, y) = x   y$ , bitwise OR                   |
| GrB_BinaryOp  | GrB_BAND_ $I$        | $I \times I \rightarrow I$                               | $f(x, y) = x \& y$ , bitwise AND                 |
| GrB_BinaryOp  | GrB_BXOR_ $I$        | $I \times I \rightarrow I$                               | $f(x, y) = x \wedge y$ , bitwise XOR             |
| GrB_BinaryOp  | GrB_BXNOR_ $I$       | $I \times I \rightarrow I$                               | $f(x, y) = \overline{x \wedge y}$ , bitwise XNOR |
| GrB_BinaryOp  | GrB_EQ_ $T$          | $T \times T \rightarrow \text{bool}$                     | $f(x, y) = (x == y)$ , equal                     |
| GrB_BinaryOp  | GrB_NE_ $T$          | $T \times T \rightarrow \text{bool}$                     | $f(x, y) = (x \neq y)$ , not equal               |
| GrB_BinaryOp  | GrB_GT_ $T$          | $T \times T \rightarrow \text{bool}$                     | $f(x, y) = (x > y)$ , greater than               |
| GrB_BinaryOp  | GrB_LT_ $T$          | $T \times T \rightarrow \text{bool}$                     | $f(x, y) = (x < y)$ , less than                  |
| GrB_BinaryOp  | GrB_GE_ $T$          | $T \times T \rightarrow \text{bool}$                     | $f(x, y) = (x \geq y)$ , greater than or equal   |
| GrB_BinaryOp  | GrB_LE_ $T$          | $T \times T \rightarrow \text{bool}$                     | $f(x, y) = (x \leq y)$ , less than or equal      |
| GrB_BinaryOp  | GrB_ONEB_ $T$        | $T \times T \rightarrow T$                               | $f(x, y) = 1$ , 1 (cast to $T$ )                 |
| GrB_BinaryOp  | GrB_FIRST_ $T$       | $T \times T \rightarrow T$                               | $f(x, y) = x$ , first argument                   |
| GrB_BinaryOp  | GrB_SECOND_ $T$      | $T \times T \rightarrow T$                               | $f(x, y) = y$ , second argument                  |
| GrB_BinaryOp  | GrB_MIN_ $T$         | $T \times T \rightarrow T$                               | $f(x, y) = (x < y) ? x : y$ , minimum            |
| GrB_BinaryOp  | GrB_MAX_ $T$         | $T \times T \rightarrow T$                               | $f(x, y) = (x > y) ? x : y$ , maximum            |
| GrB_BinaryOp  | GrB_PLUS_ $T$        | $T \times T \rightarrow T$                               | $f(x, y) = x + y$ , addition                     |
| GrB_BinaryOp  | GrB_MINUS_ $T$       | $T \times T \rightarrow T$                               | $f(x, y) = x - y$ , subtraction                  |
| GrB_BinaryOp  | GrB_TIMES_ $T$       | $T \times T \rightarrow T$                               | $f(x, y) = xy$ , multiplication                  |
| GrB_BinaryOp  | GrB_DIV_ $T$         | $T \times T \rightarrow T$                               | $f(x, y) = \frac{x}{y}$ , division               |

Table 3.6: Predefined index unary operators for GraphBLAS in C. The  $T$  can be any suffix from Table 3.2.  $I_{U64}$  refers to the unsigned 64-bit, GrB\_Index, integer type,  $I_{32}$  refers to the signed, 32-bit integer type, and  $I_{64}$  refers to signed, 64-bit integer type. The parameters,  $u_i$  or  $A_{ij}$ , are the stored values from the containers where the  $i$  and  $j$  parameters are set to the row and column indices corresponding to the location of the stored value. When operating on vectors,  $j$  will be passed with a zero value. Finally,  $s$  is an additional scalar value used in the operators. The expressions in the “Description” column are to be treated as mathematical specifications. That is, for the index arithmetic functions in the first two groups below, each one of  $i$ ,  $j$ , and  $s$  is interpreted as an integer number in the set  $\mathbb{Z}$ . Functions are evaluated using arithmetic in  $\mathbb{Z}$ , producing a result value that is also in  $\mathbb{Z}$ . The result value is converted to the output type according to the rules of the C language. In particular, if the value cannot be represented as a signed 32- or 64-bit integer type, the output is implementation defined. Any deviations from this ideal behavior, including limitations on the values of  $i$ ,  $j$ , and  $s$ , or possible overflow and underflow conditions, must be defined by the implementation.

| Operator type<br>Type | GraphBLAS<br>Name          | Domains (– is don’t care)<br>$A, u$ $i, j$ $s$ result |           |             |             | Description   |
|-----------------------|----------------------------|---|-----------|-------------|-------------|---|
| GrB_IndexUnaryOp      | GrB_ROWINDEX_ $I_{32/64}$  | –   | $I_{U64}$ | $I_{32/64}$ | $I_{32/64}$ | $f(A_{ij}, i, j, s) = (i + s)$ , replace with its row index (+ s)             |
|                       |                            | –   | $I_{U64}$ | $I_{32/64}$ | $I_{32/64}$ | $f(u_i, i, 0, s) = (i + s)$   |
| GrB_IndexUnaryOp      | GrB_COLINDEX_ $I_{32/64}$  | –   | $I_{U64}$ | $I_{32/64}$ | $I_{32/64}$ | $f(A_{ij}, i, j, s) = (j + s)$ replace with its column index (+ s)            |
| GrB_IndexUnaryOp      | GrB_DIAGINDEX_ $I_{32/64}$ | –   | $I_{U64}$ | $I_{32/64}$ | $I_{32/64}$ | $f(A_{ij}, i, j, s) = (j - i + s)$ replace with its diagonal index (+ s)      |
| GrB_IndexUnaryOp      | GrB_TRIL                   | –   | $I_{U64}$ | $I_{64}$    | bool        | $f(A_{ij}, i, j, s) = (j \leq i + s)$ triangle on or below diagonal s         |
| GrB_IndexUnaryOp      | GrB_TRIU                   | –   | $I_{U64}$ | $I_{64}$    | bool        | $f(A_{ij}, i, j, s) = (j \geq i + s)$ triangle on or above diagonal s         |
| GrB_IndexUnaryOp      | GrB_DIAG                   | –   | $I_{U64}$ | $I_{64}$    | bool        | $f(A_{ij}, i, j, s) = (j == i + s)$ diagonal s                                |
| GrB_IndexUnaryOp      | GrB_OFFDIAG                | –   | $I_{U64}$ | $I_{64}$    | bool        | $f(A_{ij}, i, j, s) = (j \neq i + s)$ all but diagonal s                      |
| GrB_IndexUnaryOp      | GrB_COLLE                  | –   | $I_{U64}$ | $I_{64}$    | bool        | $f(A_{ij}, i, j, s) = (j \leq s)$ columns less or equal to s                  |
| GrB_IndexUnaryOp      | GrB_COLGT                  | –   | $I_{U64}$ | $I_{64}$    | bool        | $f(A_{ij}, i, j, s) = (j > s)$ columns greater than s                         |
| GrB_IndexUnaryOp      | GrB_ROWLE                  | –   | $I_{U64}$ | $I_{64}$    | bool        | $f(A_{ij}, i, j, s) = (i \leq s)$ , rows less or equal to s                   |
|                       |                            | –   | $I_{U64}$ | $I_{64}$    | bool        | $f(u_i, i, 0, s) = (i \leq s)$  |
| GrB_IndexUnaryOp      | GrB_ROWGT                  | –   | $I_{U64}$ | $I_{64}$    | bool        | $f(A_{ij}, i, j, s) = (i > s)$ , rows greater than s                          |
|                       |                            | –   | $I_{U64}$ | $I_{64}$    | bool        | $f(u_i, i, 0, s) = (i > s)$   |
| GrB_IndexUnaryOp      | GrB_VALUEEQ_ $T$           | $T$   | –         | $T$         | bool        | $f(A_{ij}, i, j, s) = (A_{ij} == s)$ , elements equal to value s              |
|                       |                            | $T$   | –         | $T$         | bool        | $f(u_i, i, 0, s) = (u_i == s)$  |
| GrB_IndexUnaryOp      | GrB_VALUENE_ $T$           | $T$   | –         | $T$         | bool        | $f(A_{ij}, i, j, s) = (A_{ij} \neq s)$ , elements not equal to value s        |
|                       |                            | $T$   | –         | $T$         | bool        | $f(u_i, i, 0, s) = (u_i \neq s)$  |
| GrB_IndexUnaryOp      | GrB_VALUELT_ $T$           | $T$   | –         | $T$         | bool        | $f(A_{ij}, i, j, s) = (A_{ij} < s)$ , elements less than value s              |
|                       |                            | $T$   | –         | $T$         | bool        | $f(u_i, i, 0, s) = (u_i < s)$   |
| GrB_IndexUnaryOp      | GrB_VALUELE_ $T$           | $T$   | –         | $T$         | bool        | $f(A_{ij}, i, j, s) = (A_{ij} \leq s)$ , elements less or equal to value s    |
|                       |                            | $T$   | –         | $T$         | bool        | $f(u_i, i, 0, s) = (u_i \leq s)$  |
| GrB_IndexUnaryOp      | GrB_VALUEGT_ $T$           | $T$   | –         | $T$         | bool        | $f(A_{ij}, i, j, s) = (A_{ij} > s)$ , elements greater than value s           |
|                       |                            | $T$   | –         | $T$         | bool        | $f(u_i, i, 0, s) = (u_i > s)$   |
| GrB_IndexUnaryOp      | GrB_VALUEGE_ $T$           | $T$   | –         | $T$         | bool        | $f(A_{ij}, i, j, s) = (A_{ij} \geq s)$ , elements greater or equal to value s |
|                       |                            | $T$   | –         | $T$         | bool        | $f(u_i, i, 0, s) = (u_i \geq s)$  |

### 3.4.2 Monoids

A GraphBLAS *monoid*  $M = \langle D, \odot, 0 \rangle$  is defined by a single domain  $D$ , an *associative*<sup>1</sup> operation  $\odot : D \times D \rightarrow D$ , and an identity element  $0 \in D$ . For a given GraphBLAS monoid  $M = \langle D, \odot, 0 \rangle$  we define  $\mathbf{D}(M) = D$ ,  $\odot(M) = \odot$ , and  $\mathbf{0}(M) = 0$ . A GraphBLAS monoid is equivalent to the conventional *monoid* algebraic structure.

Let  $F = \langle D, D, D, \odot \rangle$  be an associative GraphBLAS binary operator with identity element  $0 \in D$ . Then  $M = \langle F, 0 \rangle = \langle D, \odot, 0 \rangle$  is a GraphBLAS monoid. If  $\odot$  is commutative, then  $M$  is said to be a *commutative monoid*. If a monoid  $M$  is created using an operator  $\odot$  that is not associative, the outcome of GraphBLAS operations using such a monoid is undefined.

User-defined monoids can be created with calls to `GrB_Monoid_new` (see Section 4.2.2). The GraphBLAS C API predefines a number of monoids that are listed in Table 3.7. Predefined monoids are named `GrB_op_MONOID_T`, where *op* is the name of the predefined GraphBLAS operator used as the associative binary operation of the monoid and *T* is the domain (type) of the monoid.

### 3.4.3 Semirings

A GraphBLAS *semiring*  $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$  is defined by three domains  $D_{out}$ ,  $D_{in_1}$ , and  $D_{in_2}$ ; an *associative*<sup>1</sup> and commutative additive operation  $\oplus : D_{out} \times D_{out} \rightarrow D_{out}$ ; a multiplicative operation  $\otimes : D_{in_1} \times D_{in_2} \rightarrow D_{out}$ ; and an identity element  $0 \in D_{out}$ . For a given GraphBLAS semiring  $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$  we define  $\mathbf{D}_{in_1}(S) = D_{in_1}$ ,  $\mathbf{D}_{in_2}(S) = D_{in_2}$ ,  $\mathbf{D}_{out}(S) = D_{out}$ ,  $\oplus(S) = \oplus$ ,  $\otimes(S) = \otimes$ , and  $\mathbf{0}(S) = 0$ .

Let  $F = \langle D_{out}, D_{in_1}, D_{in_2}, \otimes \rangle$  be an operator and let  $A = \langle D_{out}, \oplus, 0 \rangle$  be a commutative monoid, then  $S = \langle A, F \rangle = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0 \rangle$  is a semiring.

In a GraphBLAS semiring, the multiplicative operator does not have to distribute over the additive operator. This is unlike the conventional *semiring* algebraic structure.

Note: There must be one GraphBLAS monoid in every semiring which serves as the semiring's additive operator and specifies the same domain for its inputs and output parameters. If this monoid is not a commutative monoid, the outcome of GraphBLAS operations using the semiring is undefined.

A UML diagram of the conceptual hierarchy of object classes in GraphBLAS algebra (binary operators, monoids, and semirings) is shown in Figure 3.1.

User-defined semirings can be created with calls to `GrB_Semiring_new` (see Section 4.2.2). A list of predefined true semirings and convenience semirings can be found in Tables 3.8 and 3.9, respectively. Predefined semirings are named `GrB_add_mul_SEMIRING_T`, where *add* is the semiring additive operation, *mul* is the semiring multiplicative operation and *T* is the domain (type) of the semiring.

<sup>1</sup>It is expected that implementations of the GraphBLAS will utilize floating point arithmetic such as that defined in the IEEE-754 standard even though floating point arithmetic is not strictly associative.

Table 3.7: Predefined monoids for GraphBLAS in C. Maximum and minimum values for the various integral types are defined in `stdint.h`. Floating-point infinities are defined in `math.h`. The  $x$  in `UINT $x$`  or `INT $x$`  can be one of 8, 16, 32, or 64; whereas in `FP $x$` , it can be 32 or 64.

| GraphBLAS<br>identifier | Domains, $T$<br>( $T \times T \rightarrow T$ ) | Identity      | Description             |
|-------------------------|--|---------------|-------------------------|
| GrB_PLUS_MONOID_ $T$    | UINT $x$                                       | 0             | addition                |
|                         | INT $x$  | 0             |                         |
|                         | FP $x$   | 0             |                         |
| GrB_TIMES_MONOID_ $T$   | UINT $x$                                       | 1             | multiplication          |
|                         | INT $x$  | 1             |                         |
|                         | FP $x$   | 1             |                         |
| GrB_MIN_MONOID_ $T$     | UINT $x$                                       | UINT $x$ _MAX | minimum                 |
|                         | INT $x$  | INT $x$ _MAX  |                         |
|                         | FP $x$   | INFINITY      |                         |
| GrB_MAX_MONOID_ $T$     | UINT $x$                                       | 0             | maximum                 |
|                         | INT $x$  | INT $x$ _MIN  |                         |
|                         | FP $x$   | -INFINITY     |                         |
| GrB_LOR_MONOID_BOOL     | BOOL   | false         | logical OR              |
| GrB_LAND_MONOID_BOOL    | BOOL   | true          | logical AND             |
| GrB_LXOR_MONOID_BOOL    | BOOL   | false         | logical XOR (not equal) |
| GrB_LXNOR_MONOID_BOOL   | BOOL   | true          | logical XNOR (equal)    |



Table 3.8: Predefined true semirings for GraphBLAS in C where the additive identity is the multiplicative annihilator. The  $x$  can be one of 8, 16, 32, or 64 in `UINT $x$`  or `INT $x$` , and can be 32 or 64 in `FP $x$` .

| GraphBLAS identifier                     | Domains, $T$<br>( $T \times T \rightarrow T$ )   | + identity<br>$\times$ annihilator   | Description                  |
|--|--|--|------------------------------|
| <code>GrB_PLUS_TIMES_SEMIRING_T</code>   | <code>UINT<math>x</math></code><br><code>INT<math>x</math></code><br><code>FP<math>x</math></code> | 0<br>0<br>0  | arithmetic semiring          |
| <code>GrB_MIN_PLUS_SEMIRING_T</code>     | <code>UINT<math>x</math></code><br><code>INT<math>x</math></code><br><code>FP<math>x</math></code> | <code>UINT<math>x</math>_MAX</code><br><code>INT<math>x</math>_MAX</code><br><code>INFINITY</code> | min-plus semiring            |
| <code>GrB_MAX_PLUS_SEMIRING_T</code>     | <code>INT<math>x</math></code><br><code>FP<math>x</math></code>                                    | <code>INT<math>x</math>_MIN</code><br><code>-INFINITY</code>                                       | max-plus semiring            |
| <code>GrB_MIN_TIMES_SEMIRING_T</code>    | <code>UINT<math>x</math></code>  | <code>UINT<math>x</math>_MAX</code>  | min-times semiring           |
| <code>GrB_MIN_MAX_SEMIRING_T</code>      | <code>UINT<math>x</math></code><br><code>INT<math>x</math></code><br><code>FP<math>x</math></code> | <code>UINT<math>x</math>_MAX</code><br><code>INT<math>x</math>_MAX</code><br><code>INFINITY</code> | min-max semiring             |
| <code>GrB_MAX_MIN_SEMIRING_T</code>      | <code>UINT<math>x</math></code><br><code>INT<math>x</math></code><br><code>FP<math>x</math></code> | 0<br><code>INT<math>x</math>_MIN</code><br><code>-INFINITY</code>                                  | max-min semiring             |
| <code>GrB_MAX_TIMES_SEMIRING_T</code>    | <code>UINT<math>x</math></code>  | 0  | max-times semiring           |
| <code>GrB_PLUS_MIN_SEMIRING_T</code>     | <code>UINT<math>x</math></code>  | 0  | plus-min semiring            |
| <code>GrB_LOR_LAND_SEMIRING_BOOL</code>  | <code>BOOL</code>  | <code>false</code>   | Logical semiring             |
| <code>GrB_LAND_LOR_SEMIRING_BOOL</code>  | <code>BOOL</code>  | <code>true</code>  | "and-or" semiring            |
| <code>GrB_LXOR_LAND_SEMIRING_BOOL</code> | <code>BOOL</code>  | <code>false</code>   | same as <code>NE_LAND</code> |
| <code>GrB_LXNOR_LOR_SEMIRING_BOOL</code> | <code>BOOL</code>  | <code>true</code>  | same as <code>EQ_LOR</code>  |

Table 3.9: Other useful predefined semirings for GraphBLAS in C that don't have a multiplicative annihilator. The  $x$  can be one of 8, 16, 32, or 64 in  $\text{UINT}x$  or  $\text{INT}x$ , and can be 32 or 64 in  $\text{FP}x$ .

| GraphBLAS identifier                   | Domains, $T$<br>( $T \times T \rightarrow T$ ) | + identity          | Description                |
|--|--|---------------------|----------------------------|
| <code>GrB_MAX_PLUS_SEMIRING_T</code>   | $\text{UINT}x$                                 | 0                   | max-plus semiring          |
| <code>GrB_MIN_TIMES_SEMIRING_T</code>  | $\text{INT}x$                                  | $\text{INT}x\_MAX$  | min-times semiring         |
|  | $\text{FP}x$                                   | $INFINITY$          |                            |
| <code>GrB_MAX_TIMES_SEMIRING_T</code>  | $\text{INT}x$                                  | $\text{INT}x\_MIN$  | max-times semiring         |
|  | $\text{FP}x$                                   | $-INFINITY$         |                            |
| <code>GrB_PLUS_MIN_SEMIRING_T</code>   | $\text{INT}x$                                  | 0                   | plus-min semiring          |
|  | $\text{FP}x$                                   | 0                   |                            |
| <code>GrB_MIN_FIRST_SEMIRING_T</code>  | $\text{UINT}x$                                 | $\text{UINT}x\_MAX$ | min-select first semiring  |
|  | $\text{INT}x$                                  | $\text{INT}x\_MAX$  |                            |
|  | $\text{FP}x$                                   | $INFINITY$          |                            |
| <code>GrB_MIN_SECOND_SEMIRING_T</code> | $\text{UINT}x$                                 | $\text{UINT}x\_MAX$ | min-select second semiring |
|  | $\text{INT}x$                                  | $\text{INT}x\_MAX$  |                            |
|  | $\text{FP}x$                                   | $INFINITY$          |                            |
| <code>GrB_MAX_FIRST_SEMIRING_T</code>  | $\text{UINT}x$                                 | 0                   | max-select first semiring  |
|  | $\text{INT}x$                                  | $\text{INT}x\_MIN$  |                            |
|  | $\text{FP}x$                                   | $-INFINITY$         |                            |
| <code>GrB_MAX_SECOND_SEMIRING_T</code> | $\text{UINT}x$                                 | 0                   | max-select second semiring |
|  | $\text{INT}x$                                  | $\text{INT}x\_MIN$  |                            |
|  | $\text{FP}x$                                   | $-INFINITY$         |                            |

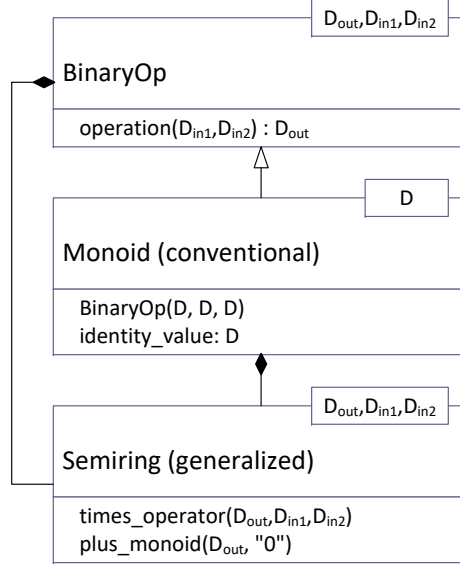


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

## 3.5 Collections

### 3.5.1 Scalars

A *GraphBLAS scalar*,  $s = \langle D, \{\sigma\} \rangle$ , is defined by a domain  $D$ , and a set of zero or one *scalar value*,  $\sigma$ , where  $\sigma \in D$ . We define  $\mathbf{size}(s) = 1$  (constant), and  $\mathbf{L}(s) = \{\sigma\}$ . The set  $\mathbf{L}(s)$  is called the *contents* of the GraphBLAS scalar  $s$ . We also define  $\mathbf{D}(s) = D$ . Finally,  $\mathbf{val}(s)$  is a reference to the scalar value,  $\sigma$ , if the GraphBLAS scalar is not empty, and is undefined otherwise.

### 3.5.2 Vectors

A vector  $\mathbf{v} = \langle D, N, \{(i, v_i)\} \rangle$  is defined by a domain  $D$ , a size  $N > 0$ , and a set of tuples  $(i, v_i)$  where  $0 \leq i < N$  and  $v_i \in D$ . A particular value of  $i$  can appear at most once in  $\mathbf{v}$ . We define  $\mathbf{size}(\mathbf{v}) = N$  and  $\mathbf{L}(\mathbf{v}) = \{(i, v_i)\}$ . The set  $\mathbf{L}(\mathbf{v})$  is called the *content* of vector  $\mathbf{v}$ . We also define the set  $\mathbf{ind}(\mathbf{v}) = \{i : (i, v_i) \in \mathbf{L}(\mathbf{v})\}$  (called the *structure* of  $\mathbf{v}$ ), and  $\mathbf{D}(\mathbf{v}) = D$ . For a vector  $\mathbf{v}$ ,  $\mathbf{v}(i)$  is a reference to  $v_i$  if  $(i, v_i) \in \mathbf{L}(\mathbf{v})$  and is undefined otherwise.

### 3.5.3 Matrices

A matrix  $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$  is defined by a domain  $D$ , its number of rows  $M > 0$ , its number of columns  $N > 0$ , and a set of tuples  $(i, j, A_{ij})$  where  $0 \leq i < M$ ,  $0 \leq j < N$ , and  $A_{ij} \in D$ . A particular pair of values  $i, j$  can appear at most once in  $\mathbf{A}$ . We define  $\mathbf{ncols}(\mathbf{A}) = N$ ,  $\mathbf{nrows}(\mathbf{A}) = M$ , and  $\mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\}$ . The set  $\mathbf{L}(\mathbf{A})$  is called the *content* of matrix  $\mathbf{A}$ . We also define the sets  $\mathbf{indrow}(\mathbf{A}) = \{i : \exists (i, j, A_{ij}) \in \mathbf{A}\}$  and  $\mathbf{indcol}(\mathbf{A}) = \{j : \exists (i, j, A_{ij}) \in \mathbf{A}\}$ . (These are the sets of nonempty rows and columns of  $\mathbf{A}$ , respectively.) The *structure* of matrix  $\mathbf{A}$  is the set  $\mathbf{ind}(\mathbf{A}) = \{(i, j) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}$ , and  $\mathbf{D}(\mathbf{A}) = D$ . For a matrix  $\mathbf{A}$ ,  $\mathbf{A}(i, j)$  is a reference to  $A_{ij}$  if  $(i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})$  and is undefined otherwise.

If  $\mathbf{A}$  is a matrix and  $0 \leq j < N$ , then  $\mathbf{A}(:, j) = \langle D, M, \{(i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\} \rangle$  is a vector called the  $j$ -th *column* of  $\mathbf{A}$ . Correspondingly, if  $\mathbf{A}$  is a matrix and  $0 \leq i < M$ , then  $\mathbf{A}(i, :) = \langle D, N, \{(j, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\} \rangle$  is a vector called the  $i$ -th *row* of  $\mathbf{A}$ .

Given a matrix  $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$ , its *transpose* is another matrix  $\mathbf{A}^T = \langle D, N, M, \{(j, i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\} \rangle$ .

#### 3.5.3.1 External matrix formats

The specification also supports the export and import of matrices to/from a number of commonly used formats, such as COO, CSR, and CSC formats. When importing or exporting a matrix to or from a GraphBLAS object using `GrB_Matrix_import` (§ 4.2.5.17) or `GrB_Matrix_export` (§ 4.2.5.16), it is necessary to specify the data format for the matrix data external to GraphBLAS, which is being imported from or exported to. This non-opaque data format is specified using an argument of enumeration type `GrB_Format` that is used to indicate one of a number of predefined formats. The predefined values of `GrB_Format` are specified in Table 3.10. A precise definition of the non-opaque data formats can be found in Appendix B.

Table 3.10: `GrB_Format` enumeration literals and corresponding values for matrix import and export methods.

| Symbol                      | Value | Description   |
|-----------------------------|-------|---|
| <code>GrB_CSR_FORMAT</code> | 0     | Specifies the compressed sparse row matrix format.    |
| <code>GrB_CSC_FORMAT</code> | 1     | Specifies the compressed sparse column matrix format. |
| <code>GrB_COO_FORMAT</code> | 2     | Specifies the sparse coordinate matrix format.        |

### 3.5.4 Masks

The GraphBLAS C API defines an opaque object called a *mask*. The mask is used to control how computed values are stored in the output from a method. The mask is an *internal* opaque object; that is, it is never exposed as a variable within an application.

The mask is formed from input objects to the method that uses the mask. For example, a GraphBLAS method may be called with a matrix as the mask parameter. The internal mask object is

constructed from the input matrix in one of two ways. In the default case, an element of the mask is created for each tuple that exists in the matrix for which the value of the tuple cast to Boolean evaluates to **true**. Alternatively, the user can specify *structure*-only behavior where an element of the mask is created for each tuple that exists in the matrix *regardless* of the value stored in the input matrix.

The internal mask object can be either a one- or a two-dimensional construct. One- and two-dimensional masks, described more formally below, are similar to vectors and matrices, respectively, except that they have structure (indices) but no values. When needed, a value is implied for the elements of a mask with an implied value of **true** for elements that exist and an implied value of **false** for elements that do not exist (i.e., the locations of the mask that do not have a stored value imply a value of **false**). Hence, even though a mask does not contain any values, it can be considered to imply values from a Boolean domain.

A one-dimensional mask  $\mathbf{m} = \langle N, \{i\} \rangle$  is defined by its number of elements  $N > 0$ , and a set  $\mathbf{ind}(\mathbf{m})$  of indices  $\{i\}$  where  $0 \leq i < N$ . A particular value of  $i$  can appear at most once in  $\mathbf{m}$ . We define  $\mathbf{size}(\mathbf{m}) = N$ . The set  $\mathbf{ind}(\mathbf{m})$  is called the *structure* of mask  $\mathbf{m}$ .

A two-dimensional mask  $\mathbf{M} = \langle M, N, \{(i, j)\} \rangle$  is defined by its number of rows  $M > 0$ , its number of columns  $N > 0$ , and a set  $\mathbf{ind}(\mathbf{M})$  of tuples  $(i, j)$  where  $0 \leq i < M, 0 \leq j < N$ . A particular pair of values  $i, j$  can appear at most once in  $\mathbf{M}$ . We define  $\mathbf{ncols}(\mathbf{M}) = N$ , and  $\mathbf{nrows}(\mathbf{M}) = M$ . We also define the sets  $\mathbf{indrow}(\mathbf{M}) = \{i : \exists (i, j) \in \mathbf{ind}(\mathbf{M})\}$  and  $\mathbf{indcol}(\mathbf{M}) = \{j : \exists (i, j) \in \mathbf{ind}(\mathbf{M})\}$ . These are the sets of nonempty rows and columns of  $\mathbf{M}$ , respectively. The set  $\mathbf{ind}(\mathbf{M})$  is called the *structure* of mask  $\mathbf{M}$ .

One common operation on masks is the *complement*. For a one-dimensional mask  $\mathbf{m}$  this is denoted as  $\neg \mathbf{m}$ . For a two-dimensional mask  $\mathbf{M}$ , this is denoted as  $\neg \mathbf{M}$ . The complement of a one-dimensional mask  $\mathbf{m}$  is defined as  $\mathbf{ind}(\neg \mathbf{m}) = \{i : 0 \leq i < N, i \notin \mathbf{ind}(\mathbf{m})\}$ . It is the set of all possible indices that do not appear in  $\mathbf{m}$ . The complement of a two-dimensional mask  $\mathbf{M}$  is defined as the set  $\mathbf{ind}(\neg \mathbf{M}) = \{(i, j) : 0 \leq i < M, 0 \leq j < N, (i, j) \notin \mathbf{ind}(\mathbf{M})\}$ . It is the set of all possible indices that do not appear in  $\mathbf{M}$ .

## 3.6 Fields

GraphBLAS objects and implementations contain internal fields which may provide information to users and allow setting runtime parameters and hints. All GraphBLAS objects are required to implement the **get** and **set** methods required to query and set these fields.

A GraphBLAS object may contain a number of (*field*, *value*) pairs, where the *value* type is determined by the *field*. Objects must implement a set of such pairs as determined by the specification, but may extend that set with implementation specific pairs.

The GraphBLAS implementation itself contains several (*field*, *value*) pairs, which provide defaults to object level fields, and implementation information such as the version number or implementation name.

A field must always be readable, but in many cases may not be writable. Such read-only fields might contain static, compile-time information such as `GrB_API_VER`, while others are determined

963 by other operations, such as `GrB_BLOCKING_MODE` which is determined by `GrB_Init`.  
964 Several fields are only *hints*. A GraphBLAS implementation is free to ignore a hint for any reason.

Table 3.11: Field values of type GrB\_Field enumeration, corresponding types, and the objects which must implement that GrB\_Field. Collection refers to GrB\_Matrix, GrB\_Vector, and GrB\_Scalar, Algebraic refers to Operators, Monoids, and Semirings, while All refers to all GraphBLAS objects. Global fields are denoted by Global. All fields may be read, some may be written (denoted by W), and some are hints (denoted by H) which may be ignored by the implementation.

(a) Types used with GraphBLAS descriptors.

| Field Name                   | W   H | Value | Implementing Objects     | Type  |
|------------------------------|-------|-------|--------------------------|---|
| GrB_OUTP                     | W   — | 0     | GrB_Descriptor           | GrB_Desc_Value  |
| GrB_MASK                     | W   — | 1     | GrB_Descriptor           | GrB_Desc_Value  |
| GrB_INP0                     | W   — | 2     | GrB_Descriptor           | GrB_Desc_Value  |
| GrB_INP1                     | W   — | 3     | GrB_Descriptor           | GrB_Desc_Value  |
| GrB_NAMESIZE                 | —   — | 10    | All                      | GrB_Index   |
| GrB_NAME                     | *     | 11    | All                      | Null terminated char* of size GrB_NAMESIZE<br>Minimum supported size of 512-bytes |
| GrB_LIBRARY_NAME             | —   — | 100   | Global                   | 256-byte null terminated char*  |
| GrB_LIBRARY_VER              | —   — | 101   | Global                   | Length 3 integer array  |
| GrB_API_VER                  | —   — | 102   | Global                   | Length 3 integer array  |
| GrB_BLOCKING_MODE            | —   — | 103   | Global                   | GrB_Mode  |
| GrB_NTHREADS                 | W     | 104   | Global, GrB_Descriptor   | GrB_Index   |
| GrB_STORAGE_ORIENTATION_HINT | W   H | 200   | Global, Collection       | GrB_ROWMAJOR, GrB_COLMAJOR  |
| GrB_STORAGE_FORMAT_HINT      | W   H | 201   | Collection               | GrB_Format  |
| GrB_ELTYPE??                 | —   — | 202   | Collection               | GrB_Type  |
| GrB_INPUT1TYPE??             | —   — | 300   | Algebraic                | GrB_Type  |
| GrB_INPUT2TYPE??             | —   — | 301   | Algebraic                | GrB_Type  |
| GrB_OUTPUTTYPE??             | —   — | 302   | Algebraic                | GrB_Type  |
| GrB_BINARYOP??               | —   — | 303   | GrB_Monoid, GrB_Semiring | GrB_BinaryOp  |
| GrB_MONOID??                 | —   — | 304   | GrB_Semiring             | GrB_Monoid  |

## 3.7 Descriptors

Descriptors are used to modify the behavior of a GraphBLAS method. When present in the signature of a method, they appear as the last argument in the method. Descriptors specify how the other input arguments corresponding to GraphBLAS collections – vectors, matrices, and masks – should be processed (modified) before the main operation of a method is performed. A complete list of what descriptors are capable of are presented in this section.

The descriptor is a lightweight object. It is composed of (*field*, *value*) pairs where the *field* selects one of the GraphBLAS objects from the argument list of a method and the *value* defines the indicated modification associated with that object. For example, a descriptor may specify that a particular input matrix needs to be transposed or that a mask needs to be complemented (defined in Section 3.5.4) before using it in the operation.

For the purpose of constructing descriptors, the arguments of a method that can be modified are identified by specific field names. The output parameter (typically the first parameter in a GraphBLAS method) is indicated by the field name, `GrB_OUTP`. The mask is indicated by the `GrB_MASK` field name. The input parameters corresponding to the input vectors and matrices are indicated by `GrB_INP0` and `GrB_INP1` in the order they appear in the signature of the GraphBLAS method. The descriptor is an opaque object and hence we do not define how objects of this type should be implemented. When referring to (*field*, *value*) pairs for a descriptor, however, we often use the informal notation `desc[GrB_Desc_Field].GrB_Desc_Value` without implying that a descriptor is to be implemented as an array of structures (in fact, field values can be used in conjunction with multiple values that are composable). We summarize all types, field names, and values used with descriptors in Table 3.12.

In the definitions of the GraphBLAS methods, we often refer to the *default behavior* of a method with respect to the action of a descriptor. If a descriptor is not provided or if the value associated with a particular field in a descriptor is not set, the default behavior of a GraphBLAS method is defined as follows:

- Input matrices are not transposed.
- The mask is used, as is, without complementing, and stored values are examined to determine whether they evaluate to `true` or `false`.
- Values of the output object that are not directly modified by the operation are preserved.

GraphBLAS specifies all of the valid combinations of (field, value) pairs as predefined descriptors. Their identifiers and the corresponding set of (field, value) pairs for that identifier are shown in Table 3.13.

## 3.8 GrB\_Info return values

All GraphBLAS methods return a `GrB_Info` enumeration value. The three types of return codes (informational, API error, and execution error) and their corresponding values are listed in Table 3.14.



---

Table 3.12: Descriptors are GraphBLAS objects passed as arguments to GraphBLAS operations to modify other GraphBLAS objects in the operation’s argument list. A descriptor, `desc`, has one or more (*field*, *value*) pairs indicated as `desc[GrB_Desc_Field].GrB_Desc_Value`. In this table, we define all types and literals used with descriptors.

(a) Types used with GraphBLAS descriptors.

| Type                        | Description                            |
|-----------------------------|--|
| <code>GrB_Descriptor</code> | Type of a GraphBLAS descriptor object. |
| <code>GrB_Desc_Field</code> | The descriptor field enumeration.      |
| <code>GrB_Desc_Value</code> | The descriptor value enumeration.      |

(b) Descriptor field names of type `GrB_Desc_Field` enumeration and corresponding values.

| Field Name            | Value | Description                                       |
|-----------------------|-------|---|
| <code>GrB_OUTP</code> | 0     | Field name for the output GraphBLAS object.       |
| <code>GrB_MASK</code> | 1     | Field name for the mask GraphBLAS object.         |
| <code>GrB_INP0</code> | 2     | Field name for the first input GraphBLAS object.  |
| <code>GrB_INP1</code> | 3     | Field name for the second input GraphBLAS object. |

(c) Descriptor field values of type `GrB_Desc_Value` enumeration and corresponding values.

| Value Name                 | Value | Description   |
|----------------------------|-------|---|
| (reserved)                 | 0     | Unused  |
| <code>GrB_REPLACE</code>   | 1     | Clear the output object before assigning computed values.   |
| <code>GrB_COMP</code>      | 2     | Use the complement of the associated object. When combined with <code>GrB_STRUCTURE</code> , the complement of the structure of the associated object is used without evaluating the values stored. |
| <code>GrB_TRAN</code>      | 3     | Use the transpose of the associated object.   |
| <code>GrB_STRUCTURE</code> | 4     | The write mask is constructed from the structure (pattern of stored values) of the associated object. The stored values are not examined.   |

---

Table 3.13: Predefined GraphBLAS descriptors. The list includes all possible descriptors, according to the current standard. Columns list the possible fields and entries list the value(s) associated with those fields for a given descriptor.

| Identifier       | GrB_OUTP    | GrB_MASK                | GrB_INP0 | GrB_INP1 |
|------------------|-------------|-------------------------|----------|----------|
| GrB_NULL         | –           | –                       | –        | –        |
| GrB_DESC_T1      | –           | –                       | –        | GrB_TRAN |
| GrB_DESC_T0      | –           | –                       | GrB_TRAN | –        |
| GrB_DESC_T0T1    | –           | –                       | GrB_TRAN | GrB_TRAN |
| GrB_DESC_C       | –           | GrB_COMP                | –        | –        |
| GrB_DESC_S       | –           | GrB_STRUCTURE           | –        | –        |
| GrB_DESC_CT1     | –           | GrB_COMP                | –        | GrB_TRAN |
| GrB_DESC_ST1     | –           | GrB_STRUCTURE           | –        | GrB_TRAN |
| GrB_DESC_CT0     | –           | GrB_COMP                | GrB_TRAN | –        |
| GrB_DESC_ST0     | –           | GrB_STRUCTURE           | GrB_TRAN | –        |
| GrB_DESC_CT0T1   | –           | GrB_COMP                | GrB_TRAN | GrB_TRAN |
| GrB_DESC_ST0T1   | –           | GrB_STRUCTURE           | GrB_TRAN | GrB_TRAN |
| GrB_DESC_SC      | –           | GrB_STRUCTURE, GrB_COMP | –        | –        |
| GrB_DESC_SCT1    | –           | GrB_STRUCTURE, GrB_COMP | –        | GrB_TRAN |
| GrB_DESC_SCT0    | –           | GrB_STRUCTURE, GrB_COMP | GrB_TRAN | –        |
| GrB_DESC_SCT0T1  | –           | GrB_STRUCTURE, GrB_COMP | GrB_TRAN | GrB_TRAN |
| GrB_DESC_R       | GrB_REPLACE | –                       | –        | –        |
| GrB_DESC_RT1     | GrB_REPLACE | –                       | –        | GrB_TRAN |
| GrB_DESC_RT0     | GrB_REPLACE | –                       | GrB_TRAN | –        |
| GrB_DESC_RT0T1   | GrB_REPLACE | –                       | GrB_TRAN | GrB_TRAN |
| GrB_DESC_RC      | GrB_REPLACE | GrB_COMP                | –        | –        |
| GrB_DESC_RS      | GrB_REPLACE | GrB_STRUCTURE           | –        | –        |
| GrB_DESC_RCT1    | GrB_REPLACE | GrB_COMP                | –        | GrB_TRAN |
| GrB_DESC_RST1    | GrB_REPLACE | GrB_STRUCTURE           | –        | GrB_TRAN |
| GrB_DESC_RCT0    | GrB_REPLACE | GrB_COMP                | GrB_TRAN | –        |
| GrB_DESC_RST0    | GrB_REPLACE | GrB_STRUCTURE           | GrB_TRAN | –        |
| GrB_DESC_RCT0T1  | GrB_REPLACE | GrB_COMP                | GrB_TRAN | GrB_TRAN |
| GrB_DESC_RST0T1  | GrB_REPLACE | GrB_STRUCTURE           | GrB_TRAN | GrB_TRAN |
| GrB_DESC_RSC     | GrB_REPLACE | GrB_STRUCTURE, GrB_COMP | –        | –        |
| GrB_DESC_RSCT1   | GrB_REPLACE | GrB_STRUCTURE, GrB_COMP | –        | GrB_TRAN |
| GrB_DESC_RSCT0   | GrB_REPLACE | GrB_STRUCTURE, GrB_COMP | GrB_TRAN | –        |
| GrB_DESC_RSCT0T1 | GrB_REPLACE | GrB_STRUCTURE, GrB_COMP | GrB_TRAN | GrB_TRAN |

Table 3.14: Enumeration literals and corresponding values returned by GraphBLAS methods and operations.

(a) Informational return values

| Symbol       | Value | Description  |
|--------------|-------|--|
| GrB_SUCCESS  | 0     | The method/operation completed successfully (blocking mode), or encountered no API errors (non-blocking mode). |
| GrB_NO_VALUE | 1     | A location in a matrix or vector is being accessed that has no stored value at the specified location.         |

(b) API errors

| Symbol                   | Value | Description  |
|--------------------------|-------|--|
| GrB_UNINITIALIZED_OBJECT | -1    | A GraphBLAS object is passed to a method before <code>new</code> was called on it.   |
| GrB_NULL_POINTER         | -2    | A NULL is passed for a pointer parameter.  |
| GrB_INVALID_VALUE        | -3    | Miscellaneous incorrect values.  |
| GrB_INVALID_INDEX        | -4    | Indices passed are larger than dimensions of the matrix or vector being accessed.  |
| GrB_DOMAIN_MISMATCH      | -5    | A mismatch between domains of collections and operations when user-defined domains are in use.   |
| GrB_DIMENSION_MISMATCH   | -6    | Operations on matrices and vectors with incompatible dimensions.   |
| GrB_OUTPUT_NOT_EMPTY     | -7    | An attempt was made to build a matrix or vector using an output object that already contains valid tuples (elements).                      |
| GrB_NOT_IMPLEMENTED      | -8    | An attempt was made to call a GraphBLAS method for a combination of input parameters that is not supported by a particular implementation. |

(c) Execution errors

| Symbol                  | Value | Description  |
|-------------------------|-------|--|
| GrB_PANIC               | -101  | Unknown internal error.  |
| GrB_OUT_OF_MEMORY       | -102  | Not enough memory for operations.  |
| GrB_INSUFFICIENT_SPACE  | -103  | The array provided is not large enough to hold output.   |
| GrB_INVALID_OBJECT      | -104  | One of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. |
| GrB_INDEX_OUT_OF_BOUNDS | -105  | Reference to a vector or matrix element that is outside the defined dimensions of the object.                      |
| GrB_EMPTY_OBJECT        | -106  | One of the opaque GraphBLAS objects does not have a stored value.  |



## Chapter 4

# Methods

This chapter defines the behavior of all the methods in the GraphBLAS C API. All methods can be declared for use in programs by including the `GraphBLAS.h` header file.

We would like to emphasize that no GraphBLAS method will imply a predefined order over any associative operators. Implementations of the GraphBLAS are encouraged to exploit associativity to optimize performance of any GraphBLAS method. This holds even if the definition of the GraphBLAS method implies a fixed order for the associative operations.

### 4.1 Context methods

The methods in this section set up and tear down the GraphBLAS context within which all GraphBLAS methods must be executed. The initialization of this context also includes the specification of which execution mode is to be used.

#### 4.1.1 `init`: Initialize a GraphBLAS context

Creates and initializes a GraphBLAS C API context.

#### C Syntax

```
GrB_Info GrB_init(GrB_Mode mode);
```

#### Parameters

`mode` Mode for the GraphBLAS context. Must be either `GrB_BLOCKING` or `GrB_NONBLOCKING`.

## 1020 **Return Values**

1021 `GrB_SUCCESS` operation completed successfully.

1022 `GrB_PANIC` unknown internal error.

1023 `GrB_INVALID_VALUE` invalid mode specified, or method called multiple times.

## 1024 **Description**

1025 The `init` method creates and initializes a GraphBLAS C API context. The argument to `GrB_init`  
1026 defines the mode for the context. The two available modes are:

- 1027 • `GrB_BLOCKING`: In this mode, each method in a sequence returns after its computations have  
1028 completed and output arguments are available to subsequent statements in an application.  
1029 When executing in `GrB_BLOCKING` mode, the methods execute in program order.
- 1030 • `GrB_NONBLOCKING`: In this mode, methods in a sequence may return after arguments in  
1031 the method have been tested for dimension and domain compatibility within the method  
1032 but potentially before their computations complete. Output arguments are available to sub-  
1033 sequent GraphBLAS methods in an application. When executing in `GrB_NONBLOCKING`  
1034 mode, the methods in a sequence may execute in any order that preserves the mathematical  
1035 result defined by the sequence.

1036 An application can only create one context per execution instance. An application may only call  
1037 `GrB_Init` once. Calling `GrB_Init` more than once results in undefined behavior.

### 1038 **4.1.2 finalize: Finalize a GraphBLAS context**

1039 Terminates and frees any internal resources created to support the GraphBLAS C API context.

## 1040 **C Syntax**

1041 `GrB_Info GrB_finalize();`

## 1042 **Return Values**

1043 `GrB_SUCCESS` operation completed successfully.

1044 `GrB_PANIC` unknown internal error.

## 1045 **Description**

1046 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS  
1047 C API context. `GrB_finalize` may only be called after a context has been initialized by calling  
1048 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-  
1049 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined  
1050 behavior.

### 1051 **4.1.3 getVersion: Get the version number of the standard.**

1052 Query the library for the version number of the standard that this library implements.

## 1053 **C Syntax**

```
1054         GrB_Info GrB_getVersion(unsigned int *version,  
1055                                unsigned int *subversion);
```

## 1056 **Parameters**

1057 version (OUT) On successful return will hold the value of the major version number.

1058 version (OUT) On successful return will hold the value of the subversion number.

## 1059 **Return Values**

1060 GrB\_SUCCESS operation completed successfully.

1061 GrB\_PANIC unknown internal error.

## 1062 **Description**

1063 The `getVersion` method is used to query the major and minor version number of the GraphBLAS  
1064 C API specification that the library implements at runtime. To support compile time queries the  
1065 following two macros shall also be defined by the library.

```
1066         #define GRB_VERSION      2  
1067         #define GRB_SUBVERSION  0
```

## 1068 **4.2 Object methods**

1069 This section describes methods that setup and operate on GraphBLAS opaque objects but are not  
1070 part of the the GraphBLAS math specification.

## 1071 4.2.1 Query methods

1072 The methods in this section query and, depending on the field, set internal fields of many Graph-  
1073 BLAS objects.

### 1074 4.2.1.1 get: Query the value of an object

#### 1075 C Syntax

```
1076     GrB_Info GrB_<OBJ>_get(GrB_<OBJ> o, GrB_Field field, ...);
1077
1078     GrB_Info GrB_Scalar_get(GrB_Scalar s, GrB_Field field, ...);
1079     GrB_Info GrB_Vector_get(GrB_Vector v, GrB_Field field, ...);
1080     GrB_Info GrB_Matrix_get(GrB_Matrix A, GrB_Field field, ...);
1081
1082     GrB_Info GrB_UnaryOp_get(GrB_UnaryOp op, GrB_Field field, ...);
1083     GrB_Info GrB_IndexUnaryOp_get(GrB_IndexUnaryOp op, GrB_Field field, ...);
1084     GrB_Info GrB_BinaryOp_get(GrB_BinaryOp op, GrB_Field field, ...);
1085     GrB_Info GrB_Monoid_get(GrB_Monoid op, GrB_Field field, ...);
1086     GrB_Info GrB_Semiring_get(GrB_Semiring op, GrB_Field field, ...);
1087
1088     GrB_Info GrB_Descriptor_get(GrB_Descriptor op, GrB_Field field, ...);
1089     GrB_Info GrB_Type_get(GrB_Type op, GrB_Field field, ...);
1090
1091     GrB_Info GrB_Global_get(GrB_Field field, ...);
```

#### 1092 Parameters

1093 OBJ is replaced in each signature by the object type being queried.

1094 OBJ (IN) An existing GraphBLAS object which is being queried.

1095 field (IN) The internal field being queried.

1096 ... (OUT) A pointer to a variable dependent on field to be filled with the value of the  
1097 internal field.

#### 1098 Return Value

1099 GrB\_SUCCESS The method completed successfully.

1100 GrB\_PANIC unknown internal error.

1101 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.



1102 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

1103 GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 1104 Description

1105 Queries a field of an existing GraphBLAS object.

### 1106 4.2.1.2 Descriptor\_set: Set content of descriptor

1107 Sets the content for a field for an existing descriptor.

## 1108 C Syntax

```
1109 GrB_Info GrB_Descriptor_set(GrB_Descriptor desc,  
1110                               GrB_Desc_Field field,  
1111                               GrB_Desc_Value val);
```

## 1112 Parameters

1113 desc (IN) An existing GraphBLAS descriptor to be modified.

1114 field (IN) The field being set.

1115 val (IN) New value for the field being set.

## 1116 Return Values

1117 GrB\_SUCCESS operation completed successfully.

1118 GrB\_PANIC unknown internal error.

1119 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1120 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

1121 GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 1122 Description

1123 For a given descriptor, the GrB\_Descriptor\_set method can be called for each field in the descriptor  
1124 to set the value associated with that field. Valid values for the field parameter include the following:

1125 GrB\_OUTP refers to the output parameter (result) of the operation.

1126       GrB\_MASK refers to the mask parameter of the operation.

1127       GrB\_INP0 refers to the first input parameters of the operation (matrices and vectors).

1128       GrB\_INP1 refers to the second input parameters of the operation (matrices and vectors).

1129   Valid values for the val parameter are:

1130       GrB\_STRUCTURE Use only the structure of the stored values of the corresponding mask  
1131                      (GrB\_MASK) parameter.

1132       GrB\_COMP Use the complement of the corresponding mask (GrB\_MASK) param-  
1133                      eter. When combined with GrB\_STRUCTURE, the complement of the  
1134                      structure of the mask is used without evaluating the values stored.

1135       GrB\_TRAN Use the transpose of the corresponding matrix parameter (valid for input  
1136                      matrix parameters only).

1137       GrB\_REPLACE When assigning the masked values to the output matrix or vector, clear  
1138                      the matrix first (or clear the non-masked entries). The default behavior  
1139                      is to leave non-masked locations unchanged. Valid for the GrB\_OUTP  
1140                      parameter only.

1141   Descriptor values can only be set, and once set, cannot be cleared. As, in the case of GrB\_MASK,  
1142   multiple values can be set and all will apply (for example, both GrB\_COMP and GrB\_STRUCTURE).  
1143   A value for a given field may be set multiple times but will have no additional effect. Fields that  
1144   have no values set result in their default behavior, as defined in Section 3.7.

## 1145   4.2.2   Algebra methods

### 1146   4.2.2.1   Type\_new: Construct a new GraphBLAS (user-defined) type

1147   Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,  
1148   monoids, semirings, vectors and matrices.

## 1149   C Syntax

```
1150       GrB_Info GrB_Type_new(GrB_Type   *utype,
1151                                      size_t       sizeof(ctype));
```

## 1152   Parameters

1153       utype (INOUT) On successful return, contains a handle to the newly created user-defined  
1154                      GraphBLAS type object.

1155       ctype (IN) A C type that defines the new GraphBLAS user-defined type.

## 1156 Return Values

1157                   GrB\_SUCCESS operation completed successfully.

1158                   GrB\_PANIC unknown internal error.

1159           GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1160           GrB\_NULL\_POINTER utype pointer is NULL.

## 1161 Description

1162 Given a C type `ctype`, the `Type_new` method returns in `utype` a handle to a new GraphBLAS type  
1163 that is equivalent to the C type. Variables of this `ctype` must be a struct, union, or fixed-size array.  
1164 In particular, given two variables, `src` and `dst`, of type `ctype`, the following operation must be a  
1165 valid way to copy the contents of `src` to `dst`:

1166                   `memcpy(&dst, &src, sizeof(ctype))`

1167 A new, user-defined type `utype` should be destroyed with a call to `GrB_free(utype)` when no longer  
1168 needed.

1169 It is not an error to call this method more than once on the same variable; however, the handle to  
1170 the previously created object will be overwritten.

### 1171 4.2.2.2 UnaryOp\_new: Construct a new GraphBLAS unary operator

1172 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types  
1173 (domains).

## 1174 C Syntax

```
1175           GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1176                                   void           (*unary_func)(void*, const void*),  
1177                                   GrB_Type       d_out,  
1178                                   GrB_Type       d_in);
```

## 1179 Parameters

1180       `unary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1181       unary operator object.

1182       `unary_func` (IN) a pointer to a user-defined function that takes one input parameter of `d_in`'s  
1183       type and returns a value of `d_out`'s type, both passed as `void` pointers. Specifically  
1184       the signature of the function is expected to be of the form:

```

1185         void func(void *out, const void *in);
1186

```

1187 **d\_out** (IN) The `GrB_Type` of the return value of the unary operator being created. Should  
1188 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1189 BLAS type.

1190 **d\_in** (IN) The `GrB_Type` of the input argument of the unary operator being created.  
1191 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1192 GraphBLAS type.

## 1193 Return Values

1194 `GrB_SUCCESS` operation completed successfully.

1195 `GrB_PANIC` unknown internal error.

1196 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1197 `GrB_UNINITIALIZED_OBJECT` any `GrB_Type` parameter (for user-defined types) has not been ini-  
1198 tialized by a call to `GrB_Type_new`.

1199 `GrB_NULL_POINTER` `unary_op` or `unary_func` pointers are NULL.

## 1200 Description

1201 The `UnaryOp_new` method creates a new GraphBLAS unary operator

1202  $f_u = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in}), \text{unary\_func} \rangle$

1203 and returns a handle to it in `unary_op`.

1204 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments  
1205 are aliased. In other words, for all invocations of the function:

```

1206     unary_func(out, in);

```

1207 the value of `out` must be the same as if the following code was executed:

```

1208     D(d_in) *tmp = malloc(sizeof(D(d_in)));
1209     memcpy(tmp, in, sizeof(D(d_in)));
1210     unary_func(out, tmp);
1211     free(tmp);

```

1212 It is not an error to call this method more than once on the same variable; however, the handle to  
1213 the previously created object will be overwritten.

### 1214 4.2.2.3 BinaryOp\_new: Construct a new GraphBLAS binary operator

1215 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types  
1216 (domains).

### 1217 C Syntax

```
1218     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
1219                             void          (*binary_func)(void*,  
1220                                                         const void*,  
1221                                                         const void*),  
1222                             GrB_Type      d_out,  
1223                             GrB_Type      d_in1,  
1224                             GrB_Type      d_in2);
```

### 1225 Parameters

1226 **binary\_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1227 binary operator object.

1228 **binary\_func** (IN) A pointer to a user-defined function that takes two input parameters of types  
1229 d\_in1 and d\_in2 and returns a value of type d\_out, all passed as void pointers.  
1230 Specifically the signature of the function is expected to be of the form:

```
1231         void func(void *out, const void *in1, const void *in2);
```

1233 **d\_out** (IN) The GrB\_Type of the return value of the binary operator being created. Should  
1234 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1235 BLAS type.

1236 **d\_in1** (IN) The GrB\_Type of the left hand argument of the binary operator being created.  
1237 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1238 GraphBLAS type.

1239 **d\_in2** (IN) The GrB\_Type of the right hand argument of the binary operator being cre-  
1240 ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-  
1241 defined GraphBLAS type.

### 1242 Return Values

1243 GrB\_SUCCESS operation completed successfully.

1244 GrB\_PANIC unknown internal error.

1245 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1246 GrB\_UNINITIALIZED\_OBJECT the GrB\_Type (for user-defined types) has not been initialized by a  
1247 call to GrB\_Type\_new.

1248 GrB\_NULL\_POINTER binary\_op or binary\_func pointer is NULL.

## 1249 Description

1250 The BinaryOp\_new method creates a new GraphBLAS binary operator

1251  $f_b = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in1}), \mathbf{D}(\mathbf{d\_in2}), \text{binary\_func} \rangle$

1252 and returns a handle to it in binary\_op.

1253 The implementation of binary\_func must be such that it works even if any of the d\_out, d\_in1, and  
1254 d\_in2 arguments are aliased to each other. In other words, for all invocations of the function:

1255 `binary_func(out, in1, in2);`

1256 the value of out must be the same as if the following code was executed:

```
1257     D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));  
1258     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));  
1259     memcpy(tmp1, in1, sizeof(D(d_in1)));  
1260     memcpy(tmp2, in2, sizeof(D(d_in2)));  
1261     binary_func(out, tmp1, tmp2);  
1262     free(tmp2);  
1263     free(tmp1);
```

1264 It is not an error to call this method more than once on the same variable; however, the handle to  
1265 the previously created object will be overwritten.

### 1266 4.2.2.4 Monoid\_new: Construct a new GraphBLAS monoid

1267 Creates a new monoid with specified binary operator and identity value.

## 1268 C Syntax

```
1269     GrB_Info GrB_Monoid_new(GrB_Monoid *monoid,  
1270                             GrB_BinaryOp binary_op,  
1271                             <type> identity);
```

## 1272 Parameters

1273        **monoid** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1274        monoid object.

1275        **binary\_op** (IN) An existing GraphBLAS associative binary operator whose input and output  
1276        types are the same.

1277        **identity** (IN) The value of the identity element of the monoid. Must be the same type as  
1278        the type used by the **binary\_op** operator.

## 1279 Return Values

1280        **GrB\_SUCCESS** operation completed successfully.

1281        **GrB\_PANIC** unknown internal error.

1282        **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1283 **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_BinaryOp** (for user-defined operators) has not been initial-  
1284        ized by a call to **GrB\_BinaryOp\_new**.

1285        **GrB\_NULL\_POINTER** monoid pointer is NULL.

1286        **GrB\_DOMAIN\_MISMATCH** all three argument types of the binary operator and the type of the  
1287        identity value are not the same.

## 1288 Description

1289 The **Monoid\_new** method creates a new monoid  $M = \langle \mathbf{D}(\text{binary\_op}), \text{binary\_op}, \text{identity} \rangle$  and re-  
1290 turns a handle to it in **monoid**.

1291 If **binary\_op** is not associative, the results of GraphBLAS operations that require associativity of  
1292 this monoid will be undefined.

1293 It is not an error to call this method more than once on the same variable; however, the handle to  
1294 the previously created object will be overwritten.

### 1295 4.2.2.5 Semiring\_new: Construct a new GraphBLAS semiring

1296 Creates a new semiring with specified domain, operators, and elements.

## 1297 C Syntax

```
1298        GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1299                                    GrB_Monoid     add_op,  
1300                                    GrB_BinaryOp   mul_op);
```

## 1301 Parameters

- 1302        **semiring** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1303        semiring.
- 1304        **add\_op** (IN) An existing GraphBLAS commutative monoid that specifies the addition op-  
1305        erator and its identity.
- 1306        **mul\_op** (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-  
1307        plication operator. In addition, **mul\_op**'s output domain,  $\mathbf{D}_{out}(\text{mul\_op})$ , must be  
1308        the same as the **add\_op**'s domain  $\mathbf{D}(\text{add\_op})$ .

## 1309 Return Values

- 1310        **GrB\_SUCCESS** operation completed successfully.
- 1311        **GrB\_PANIC** unknown internal error.
- 1312        **GrB\_OUT\_OF\_MEMORY** not enough memory available for this method to complete.
- 1313 **GrB\_UNINITIALIZED\_OBJECT** the **add\_op** (for user-define monoids) object has not been initialized  
1314        with a call to **GrB\_Monoid\_new** or the **mul\_op** (for user-defined  
1315        operators) object has not been not been initialized by a call to  
1316        **GrB\_BinaryOp\_new**.
- 1317        **GrB\_NULL\_POINTER** semiring pointer is NULL.
- 1318        **GrB\_DOMAIN\_MISMATCH** the output domain of **mul\_op** does not match the domain of the  
1319        **add\_op** monoid.

## 1320 Description

1321 The **Semiring\_new** method creates a new semiring:

$$1322 \quad S = \langle \mathbf{D}_{out}(\text{mul\_op}), \mathbf{D}_{in_1}(\text{mul\_op}), \mathbf{D}_{in_2}(\text{mul\_op}), \text{add\_op}, \text{mul\_op}, \mathbf{0}(\text{add\_op}) \rangle$$

1323 and returns a handle to it in **semiring**. Note that  $\mathbf{D}_{out}(\text{mul\_op})$  must be the same as  $\mathbf{D}(\text{add\_op})$ .

1324 If **add\_op** is not commutative, then GraphBLAS operations using this semiring will be undefined.

1325 It is not an error to call this method more than once on the same variable; however, the handle to  
1326 the previously created object will be overwritten.

### 1327 4.2.2.6 IndexUnaryOp\_new: Construct a new GraphBLAS index unary operator [Scott: 1328 NEW CONTENT]

1329 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its  
1330 types (domains).



## 1331 C Syntax

```
1332     GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp  *index_unary_op,  
1333                                   void (*index_unary_func)(void*,  
1334                                                             const void*,  
1335                                                             GrB_Index,  
1336                                                             GrB_Index,  
1337                                                             const void*),  
1338                                   GrB_Type          d_out,  
1339                                   GrB_Type          d_in1,  
1340                                   GrB_Type          d_in2);
```

## 1341 Parameters

1342 **index\_unary\_op** (INOUT) On successful return, contains a handle to the newly created Graph-  
1343 BLAS index unary operator object.

1344 **index\_unary\_func** (IN) A pointer to a user-defined function that takes input parameters of types  
1345 **d\_in1**, **GrB\_Index**, **GrB\_Index** and **d\_in2** and returns a value of type **d\_out**. Ex-  
1346 cept for the **GrB\_Index** parameters, all are passed as **void** pointers. Specifically  
1347 the signature of the function is expected to be of the form:

```
1348         void func(void      *out,  
1349                   const void *in1,  
1350                   GrB_Index  row_index,  
1351                   GrB_Index  col_index,  
1352                   const void *in2);  
1353
```

1354 **d\_out** (IN) The **GrB\_Type** of the return value of the index unary operator being created.  
1355 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1356 GraphBLAS type.

1357 **d\_in1** (IN) The **GrB\_Type** of the first input argument of the index unary operator being  
1358 created and corresponds to the stored values of the **GrB\_Vector** or **GrB\_Matrix**  
1359 being operated on. Should be one of the predefined GraphBLAS types in Ta-  
1360 ble 3.2, or a user-defined GraphBLAS type.

1361 **d\_in2** (IN) The **GrB\_Type** of the last input argument of the index unary operator be-  
1362 ing created and corresponds to a scalar provided by the GraphBLAS operation  
1363 that uses this operator. Should be one of the predefined GraphBLAS types in  
1364 Table 3.2, or a user-defined GraphBLAS type.

## 1365 Return Values

1366 **GrB\_SUCCESS** operation completed successfully.



## 1395 C Syntax

```
1396         GrB_Info GrB_Scalar_new(GrB_Scalar *s,  
1397                                 GrB_Type    d);
```

## 1398 Parameters

1399 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1400 scalar.

1401 **d** (IN) The type corresponding to the domain of the scalar being created. Can be  
1402 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1403 GraphBLAS type.

## 1404 Return Values

1405 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1406 blocking mode, this indicates that the API checks for the input  
1407 arguments passed successfully. Either way, output scalar **s** is ready  
1408 to be used in the next method of the sequence.

1409 **GrB\_PANIC** Unknown internal error.

1410 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1411 GraphBLAS objects (input or output) is in an invalid state caused  
1412 by a previous execution error. Call **GrB\_error()** to access any error  
1413 messages generated by the implementation.

1414 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1415 **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
1416 (needed for user-defined types).

1417 **GrB\_NULL\_POINTER** The **s** pointer is NULL.

## 1418 Description

1419 Creates a new GraphBLAS scalar **s** of domain **D(d)** and empty **L(s)**. The method returns a handle  
1420 to the new scalar in **s**.

1421 It is not an error to call this method more than once on the same variable; however, the handle to  
1422 the previously created object will be overwritten.

### 1423 4.2.3.2 Scalar\_dup: Construct a copy of a GraphBLAS scalar

1424 Creates a new scalar with the same domain and contents as another scalar.

## 1425 C Syntax

```
1426         GrB_Info GrB_Scalar_dup(GrB_Scalar      *t,  
1427                                 const GrB_Scalar  s);
```

## 1428 Parameters

1429 **t** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1430 scalar.

1431 **s** (IN) The GraphBLAS scalar to be duplicated.

## 1432 Return Values

1433 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1434 blocking mode, this indicates that the API checks for the input  
1435 arguments passed successfully. Either way, output scalar **t** is ready  
1436 to be used in the next method of the sequence.

1437 **GrB\_PANIC** Unknown internal error.

1438 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1439 GraphBLAS objects (input or output) is in an invalid state caused  
1440 by a previous execution error. Call **GrB\_error()** to access any error  
1441 messages generated by the implementation.

1442 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1443 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to  
1444 **Scalar\_new** or **Scalar\_dup**.

1445 **GrB\_NULL\_POINTER** The **t** pointer is NULL.

## 1446 Description

1447 Creates a new scalar  $t$  of domain  $\mathbf{D}(\mathbf{s})$  and contents  $\mathbf{L}(\mathbf{s})$ . The method returns a handle to the new  
1448 scalar in **t**.

1449 It is not an error to call this method more than once with the same output variable; however, the  
1450 handle to the previously created object will be overwritten.

### 1451 4.2.3.3 **Scalar\_clear**: Clear/remove a stored value from a scalar

1452 Removes the stored value from a scalar.

## 1453 C Syntax

1454           GrB\_Info GrB\_Scalar\_clear(GrB\_Scalar s);

## 1455 Parameters

1456           s (INOUT) An existing GraphBLAS scalar to clear.

## 1457 Return Values

1458           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1459                       blocking mode, this indicates that the API checks for the input  
1460                       arguments passed successfully. Either way, output scalar s is ready  
1461                       to be used in the next method of the sequence.

1462           GrB\_PANIC Unknown internal error.

1463           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1464                               GraphBLAS objects (input or output) is in an invalid state caused  
1465                               by a previous execution error. Call GrB\_error() to access any error  
1466                               messages generated by the implementation.

1467           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1468 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, s, has not been initialized by a call to  
1469                               Scalar\_new or Scalar\_dup.

## 1470 Description

1471 Removes the stored value from an existing scalar. After the call, L(s) is empty. The size of the  
1472 scalar does not change.

## 1473 4.2.3.4 Scalar\_nvals: Number of stored elements in a scalar

1474 Retrieve the number of stored elements in a scalar (either zero or one).

## 1475 C Syntax

1476           GrB\_Info GrB\_Scalar\_nvals(GrB\_Index           \*nvals,  
1477                                       const GrB\_Scalar s);

## 1478 Parameters

1479            **nvals** (OUT) On successful return, this is set to the number of stored elements in the  
1480            scalar (zero or one).

1481            **s** (IN) An existing GraphBLAS scalar being queried.

## 1482 Return Values

1483            **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1484            cessfully and the value of **nvals** has been set.

1485            **GrB\_PANIC** Unknown internal error.

1486            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1487            GraphBLAS objects (input or output) is in an invalid state caused  
1488            by a previous execution error. Call **GrB\_error()** to access any error  
1489            messages generated by the implementation.

1490            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1491            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to  
1492            **Scalar\_new** or **Scalar\_dup**.

1493            **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

## 1494 Description

1495    Return **nvals(s)** in **nvals**. This is the number of stored elements in scalar **s**, which is the size of  
1496    **L(s)**, and can only be either zero or one (see Section 3.5.1).

### 1497 4.2.3.5 Scalar\_setElement: Set the single element in a scalar

1498    Set the single element of a scalar to a given value.

## 1499 C Syntax

```
1500            GrB_Info GrB_Scalar_setElement(GrB_Scalar    s,  
1501                                                            <type>    val);
```

## 1502 Parameters

1503            **s** (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1504            **val** (IN) Scalar value to assign. The type must be compatible with the domain of **s**.

## 1505 Return Values

1506           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1507                       blocking mode, this indicates that the compatibility tests on in-  
1508                       dex/dimensions and domains for the input arguments passed suc-  
1509                       cessfully. Either way, the output scalar `s` is ready to be used in the  
1510                       next method of the sequence.

1511           GrB\_PANIC Unknown internal error.

1512           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1513                       GraphBLAS objects (input or output) is in an invalid state caused  
1514                       by a previous execution error. Call `GrB_error()` to access any error  
1515                       messages generated by the implementation.

1516           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1517 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, `s`, has not been initialized by a call to  
1518                       Scalar\_new or Scalar\_dup.

1519           GrB\_DOMAIN\_MISMATCH The domains of `s` and `val` are incompatible.

## 1520 Description

1521 First, `val` and output GraphBLAS scalar are tested for domain compatibility as follows: `D(val)` must  
1522 be compatible with `D(s)`. Two domains are compatible with each other if values from one domain  
1523 can be cast to values in the other domain as per the rules of the C language. In particular, domains  
1524 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-  
1525 patible with itself. If any compatibility rule above is violated, execution of `GrB_Scalar_setElement`  
1526 ends and the domain mismatch error listed above is returned.

1527 We are now ready to carry out the assignment `val`; that is:

$$1528 \qquad s(0) = \text{val}$$

1529 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1530 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1531 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1532 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be  
1533 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1534 4.2.3.6 Scalar\_extractElement: Extract a single element from a scalar.

1535 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

## 1536 C Syntax

```
1537         GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1538                                         const GrB_Scalar s);
```

## 1539 Parameters

1540 **val** (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain  
1541 of scalar **s**. On successful return, **val** holds the result of the operation, and any  
1542 previous value in **val** is overwritten.

1543 **s** (IN) The GraphBLAS scalar from which an element is extracted.

## 1544 Return Values

1545 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1546 cessfully. This indicates that the compatibility tests on dimensions  
1547 and domains for the input arguments passed successfully, and the  
1548 output scalar, **val**, has been computed and is ready to be used in  
1549 the next method of the sequence.

1550 **GrB\_PANIC** Unknown internal error.

1551 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1552 GraphBLAS objects (input or output) is in an invalid state caused  
1553 by a previous execution error. Call **GrB\_error()** to access any error  
1554 messages generated by the implementation.

1555 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1556 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS scalar, **s**, has not been initialized by a call to  
1557 **Scalar\_new** or **Scalar\_dup**.

1558 **GrB\_NULL\_POINTER** **val** pointer is NULL.

1559 **GrB\_DOMAIN\_MISMATCH** The domains of the scalar or scalar are incompatible.

1560 **GrB\_NO\_VALUE** There is no stored value in the scalar.

## 1561 Description

1562 First, **val** and input GraphBLAS scalar are tested for domain compatibility as follows: **D(val)**  
1563 must be compatible with **D(s)**. Two domains are compatible with each other if values from  
1564 one domain can be cast to values in the other domain as per the rules of the C language. In  
1565 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1566 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1567 **GrB\_Scalar\_extractElement** ends and the domain mismatch error listed above is returned.



1568 Then, if no value is currently stored in the GraphBLAS scalar, the method returns `GrB_NO_VALUE`  
1569 and `val` remains unchanged.

1570 Finally the extract into the output argument, `val` can be performed; that is:

1571 
$$\text{val} = \text{s}(0)$$

1572 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value  
1573 `GrB_SUCCESS`, the new contents of `val` are as defined above.

## 1574 4.2.4 Vector methods

### 1575 4.2.4.1 Vector\_new: Construct new vector

1576 Creates a new vector with specified domain and size.

#### 1577 C Syntax

```
1578     GrB_Info GrB_Vector_new(GrB_Vector *v,  
1579                             GrB_Type    d,  
1580                             GrB_Index   nsize);
```

#### 1581 Parameters

1582 `v` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1583 vector.

1584 `d` (IN) The type corresponding to the domain of the vector being created. Can be  
1585 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1586 GraphBLAS type.

1587 `nsize` (IN) The size of the vector being created.

#### 1588 Return Values

1589 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
1590 blocking mode, this indicates that the API checks for the input  
1591 arguments passed successfully. Either way, output vector `v` is ready  
1592 to be used in the next method of the sequence.

1593 `GrB_PANIC` Unknown internal error.

1594 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
1595 GraphBLAS objects (input or output) is in an invalid state caused  
1596 by a previous execution error. Call `GrB_error()` to access any error  
1597 messages generated by the implementation.

1598        GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1599 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
1600                                (needed for user-defined types).

1601        GrB\_NULL\_POINTER The v pointer is NULL.

1602        GrB\_INVALID\_VALUE nsize is zero or outside the range of the type GrB\_Index.

## 1603 Description

1604 Creates a new vector  $\mathbf{v}$  of domain  $\mathbf{D}(\mathbf{d})$ , size nsize, and empty  $\mathbf{L}(\mathbf{v})$ . The method returns a handle  
1605 to the new vector in v.

1606 It is not an error to call this method more than once on the same variable; however, the handle to  
1607 the previously created object will be overwritten.

### 1608 4.2.4.2 Vector\_dup: Construct a copy of a GraphBLAS vector

1609 Creates a new vector with the same domain, size, and contents as another vector.

## 1610 C Syntax

```
1611        GrB_Info GrB_Vector_dup(GrB_Vector        *w,  
1612                                const GrB_Vector    u);
```

## 1613 Parameters

1614        w (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1615                    vector.

1616        u (IN) The GraphBLAS vector to be duplicated.

## 1617 Return Values

1618        GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1619                    blocking mode, this indicates that the API checks for the input  
1620                    arguments passed successfully. Either way, output vector w is ready  
1621                    to be used in the next method of the sequence.

1622        GrB\_PANIC Unknown internal error.

1623        GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1624                    GraphBLAS objects (input or output) is in an invalid state caused  
1625                    by a previous execution error. Call GrB\_error() to access any error  
1626                    messages generated by the implementation.

1627       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1628 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector,  $u$ , has not been initialized by a call to  
1629       Vector\_new or Vector\_dup.

1630       GrB\_NULL\_POINTER The  $w$  pointer is NULL.

## 1631 Description

1632 Creates a new vector  $w$  of domain  $D(u)$ , size  $size(u)$ , and contents  $L(u)$ . The method returns a  
1633 handle to the new vector in  $w$ .

1634 It is not an error to call this method more than once on the same variable; however, the handle to  
1635 the previously created object will be overwritten.

### 1636 4.2.4.3 Vector\_resize: Resize a vector

1637 Changes the size of an existing vector.

## 1638 C Syntax

```
1639      GrB_Info GrB_Vector_resize(GrB_Vector  w,
1640                               GrB_Index   nsize);
```

## 1641 Parameters

1642        $w$  (INOUT) An existing Vector object that is being resized.

1643        $nsize$  (IN) The new size of the vector. It can be smaller or larger than the current size.

## 1644 Return Values

1645       GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1646       blocking mode, this indicates that the API checks for the input  
1647       arguments passed successfully. Either way, output vector  $w$  is ready  
1648       to be used in the next method of the sequence.

1649       GrB\_PANIC Unknown internal error.

1650       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1651       GraphBLAS objects (input or output) is in an invalid state caused  
1652       by a previous execution error. Call GrB\_error() to access any error  
1653       messages generated by the implementation.

1654       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1655           GrB\_NULL\_POINTER The  $w$  pointer is NULL.

1656           GrB\_INVALID\_VALUE  $nsz$  is zero or outside the range of the type GrB\_Index.

## 1657 Description

1658 Changes the size of  $w$  to  $nsz$ . The domain  $\mathbf{D}(w)$  of vector  $w$  remains the same. The contents  $\mathbf{L}(w)$   
1659 are modified as described below.

1660 Let  $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$  when the method is called. When the method returns,  $w = \langle \mathbf{D}(w), nsz, \mathbf{L}'(w) \rangle$   
1661 where  $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < nsz)\}$ . That is, all elements of  $w$  with index greater  
1662 than or equal to the new vector size ( $nsz$ ) are dropped.

### 1663 4.2.4.4 Vector\_clear: Clear a vector

1664 Removes all the elements (tuples) from a vector.

## 1665 C Syntax

1666           GrB\_Info GrB\_Vector\_clear(GrB\_Vector v);

## 1667 Parameters

1668            $v$  (INOUT) An existing GraphBLAS vector to clear.

## 1669 Return Values

1670           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1671 blocking mode, this indicates that the API checks for the input  
1672 arguments passed successfully. Either way, output vector  $v$  is ready  
1673 to be used in the next method of the sequence.

1674           GrB\_PANIC Unknown internal error.

1675           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1676 GraphBLAS objects (input or output) is in an invalid state caused  
1677 by a previous execution error. Call GrB\_error() to access any error  
1678 messages generated by the implementation.

1679           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1680           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector,  $v$ , has not been initialized by a call to  
1681 Vector\_new or Vector\_dup.

1682 **Description**

1683 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,  
1684  $L(v) = \emptyset$ . The size of the vector does not change.

1685 **4.2.4.5 Vector\_size: Size of a vector**

1686 Retrieve the size of a vector.

1687 **C Syntax**

```
1688         GrB_Info GrB_Vector_size(GrB_Index      *nsize,  
1689                                const GrB_Vector v);
```

1690 **Parameters**

1691 nsize (OUT) On successful return, is set to the size of the vector.

1692 v (IN) An existing GraphBLAS vector being queried.

1693 **Return Values**

1694 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1695 cessfully and the value of `nsize` has been set.

1696 GrB\_PANIC Unknown internal error.

1697 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1698 GraphBLAS objects (input or output) is in an invalid state caused  
1699 by a previous execution error. Call `GrB_error()` to access any error  
1700 messages generated by the implementation.

1701 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to  
1702 `Vector_new` or `Vector_dup`.

1703 GrB\_NULL\_POINTER `nsize` pointer is NULL.

1704 **Description**

1705 Return `size(v)` in `nsize`.

1706 **4.2.4.6 Vector\_nvals: Number of stored elements in a vector**

1707 Retrieve the number of stored elements (tuples) in a vector.

## 1708 C Syntax

```
1709         GrB_Info GrB_Vector_nvals(GrB_Index      *nvals,  
1710                                   const GrB_Vector v);
```

## 1711 Parameters

1712        **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)  
1713        in the vector.

1714        **v** (IN) An existing GraphBLAS vector being queried.

## 1715 Return Values

1716        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1717        cessfully and the value of **nvals** has been set.

1718        **GrB\_PANIC** Unknown internal error.

1719        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1720        GraphBLAS objects (input or output) is in an invalid state caused  
1721        by a previous execution error. Call **GrB\_error()** to access any error  
1722        messages generated by the implementation.

1723        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1724        **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to  
1725        **Vector\_new** or **Vector\_dup**.

1726        **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

## 1727 Description

1728        Return **nvals(v)** in **nvals**. This is the number of stored elements in vector **v**, which is the size of  
1729        **L(v)** (see Section 3.5.2).

### 1730 4.2.4.7 Vector\_build: Store elements from tuples into a vector

## 1731 C Syntax

```
1732         GrB_Info GrB_Vector_build(GrB_Vector      w,  
1733                                   const GrB_Index *indices,  
1734                                   const <type>    *values,  
1735                                   GrB_Index        n,  
1736                                   const GrB_BinaryOp dup);
```

## 1737 Parameters

- 1738        **w** (INOUT) An existing Vector object to store the result.
- 1739        **indices** (IN) Pointer to an array of indices.
- 1740        **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
1741        vector **w**.
- 1742        **n** (IN) The number of entries contained in each array (the same for **indices** and **values**).
- 1743        **dup** (IN) An associative and commutative binary operator to apply when duplicate  
1744        values for the same location are present in the input arrays. All three domains of  
1745        **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
1746        then duplicate locations will result in an error.

## 1747 Return Values

- 1748        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1749        blocking mode, this indicates that the API checks for the input  
1750        arguments passed successfully. Either way, output vector **w** is  
1751        ready to be used in the next method of the sequence.
- 1752        **GrB\_PANIC** Unknown internal error.
- 1753        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
1754        opaque GraphBLAS objects (input or output) is in an invalid  
1755        state caused by a previous execution error. Call **GrB\_error()** to  
1756        access any error messages generated by the implementation.
- 1757        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.
- 1758        **GrB\_UNINITIALIZED\_OBJECT** Either **w** has not been initialized by a call to **GrB\_Vector\_new**  
1759        or by **GrB\_Vector\_dup**, or **dup** has not been initialized by a call  
1760        to **GrB\_BinaryOp\_new**.
- 1761        **GrB\_NULL\_POINTER** **indices** or **values** pointer is **NULL**.
- 1762        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indices** is outside the allowed range for **w**.
- 1763        **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
1764        not all the same, or the domains of **values** and **w** are incompatible  
1765        with each other or  $D_{dup}$ .
- 1766        **GrB\_OUTPUT\_NOT\_EMPTY** Output vector **w** already contains valid tuples (elements). In  
1767        other words, **GrB\_Vector\_nvals(C)** returns a positive value.
- 1768        **GrB\_INVALID\_VALUE** **indices** contains a duplicate location and **dup** is **GrB\_NULL**.

## 1769 Description

1770 If `dup` is not `GrB_NULL`, an internal vector  $\tilde{\mathbf{w}} = \langle D_{dup}, \mathbf{size}(\mathbf{w}), \emptyset \rangle$  is created, which only differs  
 1771 from  $\mathbf{w}$  in its domain; otherwise,  $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$ .

1772 Each tuple  $\{\text{indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the output in the form of

$$1773 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1774 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
 1775 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{w}}$  as follows:

$$1776 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1777 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{w}}$  is copied into  $\mathbf{w}$  via typecasting its  
 1778 values to  $\mathbf{D}(\mathbf{w})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

1779 The nonopaque input arrays, `indices` and `values`, must be at least as large as `n`.

1780 It is an error to call this function on an output object with existing elements. In other words,  
 1781 `GrB_Vector_nvals(w)` should evaluate to zero prior to calling this function.

1782 After `GrB_Vector_build` returns, it is safe for a programmer to modify or delete the arrays `indices`  
 1783 or `values`.

### 1784 4.2.4.8 Vector\_setElement: Set a single element in a vector

1785 Set one element of a vector to a given value.

## 1786 C Syntax

```
1787 // scalar value
1788 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1789                               <type>         val,
1790                               GrB_Index        index);
1791
1792 // GraphBLAS scalar
1793 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1794                               const GrB_Scalar s,
1795                               GrB_Index        index);
```

## 1796 Parameters

1797 `w` (INOUT) An existing GraphBLAS vector for which an element is to be assigned.



1798            **val** or **s** (IN) Scalar assign. Its domain (type) must be compatible with the domain of **w**.  
1799            **index** (IN) The location of the element to be assigned.

## 1800 **Return Values**

1801            **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1802            blocking mode, this indicates that the compatibility tests on in-  
1803            dex/dimensions and domains for the input arguments passed suc-  
1804            cessfully. Either way, the output vector **w** is ready to be used in  
1805            the next method of the sequence.

1806            **GrB\_PANIC** Unknown internal error.

1807            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1808            GraphBLAS objects (input or output) is in an invalid state caused  
1809            by a previous execution error. Call **GrB\_error()** to access any error  
1810            messages generated by the implementation.

1811            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1812 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **w**, or GraphBLAS scalar, **s**, has not been  
1813            initialized by a call to a respective constructor.

1814            **GrB\_INVALID\_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1815            **GrB\_DOMAIN\_MISMATCH** The domains of the vector and the scalar are incompatible.

## 1816 **Description**

1817 First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
1818 must be compatible with **D(w)**. Two domains are compatible with each other if values from  
1819 one domain can be cast to values in the other domain as per the rules of the C language. In  
1820 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1821 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1822 **GrB\_Vector\_setElement** ends and the domain mismatch error listed above is returned.

1823 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1824 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1825 If this condition is violated, execution of **GrB\_Vector\_setElement** ends and the invalid index error  
1826 listed above is returned.

We are now ready to carry out the assignment; that is:

$$\mathbf{w}(\text{index}) = \begin{cases} \mathbf{L}(\mathbf{s}), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

1827 In the case of a transparent scalar or if  $\mathbf{L}(\mathbf{s})$  is not empty, then a value will be stored at the  
 1828 specified location in  $\mathbf{w}$ , overwriting any value that may have been stored there before. In the case  
 1829 of a GraphBLAS scalar, if  $\mathbf{L}(\mathbf{s})$  is empty, then any value stored at the specified location in  $\mathbf{w}$  will  
 1830 be removed.

1831 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 1832 of  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 1833 return value GrB\_SUCCESS and the new contents of vector  $\mathbf{w}$  is as defined above but may not be  
 1834 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1835 4.2.4.9 Vector\_removeElement: Remove an element from a vector

1836 Remove (annihilate) one stored element from a vector.

#### 1837 C Syntax

```
1838      GrB_Info GrB_Vector_removeElement(GrB_Vector  w,
1839                                     GrB_Index   index);
```

#### 1840 Parameters

1841  $\mathbf{w}$  (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1842  $\mathbf{index}$  (IN) The location of the element to be removed.

#### 1843 Return Values

1844 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 1845 blocking mode, this indicates that the compatibility tests on in-  
 1846 dex/dimensions and domains for the input arguments passed suc-  
 1847 cessfully. Either way, the output vector  $\mathbf{w}$  is ready to be used in  
 1848 the next method of the sequence.

1849 GrB\_PANIC Unknown internal error.

1850 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1851 GraphBLAS objects (input or output) is in an invalid state caused  
 1852 by a previous execution error. Call GrB\_error() to access any error  
 1853 messages generated by the implementation.

1854 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1855 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector,  $\mathbf{w}$ , has not been initialized by a call to  
 1856 Vector\_new or Vector\_dup.

1857 GrB\_INVALID\_INDEX  $\mathbf{index}$  specifies a location that is outside the dimensions of  $\mathbf{w}$ .

## 1858 Description

1859 First, the `index` parameter is checked for a valid value where the following condition must hold:

$$1860 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1861 If this condition is violated, execution of `GrB_Vector_removeElement` ends and the invalid index  
1862 error listed above is returned.

1863 We are now ready to carry out the removal of a value that may be stored at the location specified  
1864 by `index`. If a value does not exist at the specified location in  $\mathbf{w}$ , no error is reported and the  
1865 operation has no effect on the state of  $\mathbf{w}$ . In either case, the following will be true on return from  
1866 the method: `index`  $\notin$  `ind(w)`.

1867 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1868 of  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1869 return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may not be  
1870 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1871 4.2.4.10 Vector\_extractElement: Extract a single element from a vector.

1872 Extract one element of a vector into a scalar.

## 1873 C Syntax

```
1874 // scalar value
1875 GrB_Info GrB_Vector_extractElement(<type>          *val,
1876                                     const GrB_Vector u,
1877                                     GrB_Index        index);
1878
1879 // GraphBLAS scalar
1880 GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1881                                     const GrB_Vector u,
1882                                     GrB_Index        index);
```

## 1883 Parameters

1884 `val` or `s` (INOUT) An existing scalar of whose domain is compatible with the domain of vector  
1885 `u`. On successful return, this scalar holds the result of the extract. Any previous  
1886 value stored in `val` or `s` is overwritten.

1887 `u` (IN) The GraphBLAS vector from which an element is extracted.

1888 `index` (IN) The location in `u` to extract.

## 1889 Return Values

1890           GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1891                       cessfully. This indicates that the compatibility tests on dimensions  
1892                       and domains for the input arguments passed successfully, and the  
1893                       output scalar, **val** or **s**, has been computed and is ready to be used  
1894                       in the next method of the sequence.

1895           GrB\_NO\_VALUE When using the transparent scalar, **val**, this is returned when there  
1896                       is no stored value at specified location.

1897           GrB\_PANIC Unknown internal error.

1898           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1899                       GraphBLAS objects (input or output) is in an invalid state caused  
1900                       by a previous execution error. Call **GrB\_error()** to access any error  
1901                       messages generated by the implementation.

1902           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1903           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, **u**, or scalar, **s**, has not been initialized by  
1904                       a call to a corresponding constructor.

1905           GrB\_NULL\_POINTER **val** pointer is NULL.

1906           GrB\_INVALID\_INDEX **index** specifies a location that is outside the dimensions of **w**.

1907           GrB\_DOMAIN\_MISMATCH The domains of the vector and scalar are incompatible.

## 1908 Description

1909 First, the scalar and input vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
1910 must be compatible with **D(u)**. Two domains are compatible with each other if values from  
1911 one domain can be cast to values in the other domain as per the rules of the C language. In  
1912 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1913 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1914 **GrB\_Vector\_extractElement** ends and the domain mismatch error listed above is returned.

1915 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1916 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1917 If this condition is violated, execution of **GrB\_Vector\_extractElement** ends and the invalid index  
1918 error listed above is returned.

We are now ready to carry out the extract into the output scalar; that is:

$$\left. \begin{matrix} \mathbf{L}(\mathbf{s}) \\ \mathbf{val} \end{matrix} \right\} = \mathbf{u}(\text{index})$$

1919 If  $\text{index} \in \mathbf{ind}(u)$ , then the corresponding value from  $u$  is copied into  $s$  or  $val$  with casting as  
1920 necessary. If  $\text{index} \notin \mathbf{ind}(u)$ , then one of the follow occurs depending on output scalar type:

- 1921 • The GraphBLAS scalar,  $s$ , is cleared and `GrB_SUCCESS` is returned.
- 1922 • The non-opaque scalar,  $val$ , is unchanged, and `GrB_NO_VALUE` is returned.

1923 When using the non-opaque scalar variant ( $val$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
1924 mode, the new contents of  $val$  are as defined above if the method exits with return value `GrB_SUCCESS`  
1925 or `GrB_NO_VALUE`.

1926 When using the GraphBLAS scalar variant ( $s$ ) with a `GrB_SUCCESS` return value, the method  
1927 exits and the new contents of  $s$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
1928 In `GrB_NONBLOCKING` mode, the new contents of  $s$  is as defined above but may not be fully  
1929 computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1930 4.2.4.11 Vector\_extractTuples: Extract tuples from a vector

1931 Extract the contents of a GraphBLAS vector into non-opaque data structures.

#### 1932 C Syntax

```
1933      GrB_Info GrB_Vector_extractTuples(GrB_Index      *indices,  
1934                                     <type>          *values,  
1935                                     GrB_Index        *n,  
1936                                     const GrB_Vector  v);  
1937
```

1938 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored  
1939 values' indices.

1940 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
1941 the stored values whose type is compatible with  $\mathbf{D}(v)$ .

1942 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the  
1943 values and indices arrays can hold. Upon return, it will contain the number of  
1944 values written to the arrays.

1945 **v** (IN) An existing GraphBLAS vector.

#### 1946 Return Values

1947 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1948 cessfully. This indicates that the compatibility tests on the input  
1949 argument passed successfully, and the output arrays, **indices** and  
1950 **values**, have been computed.

1951                   GrB\_PANIC Unknown internal error.

1952           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1953                   GraphBLAS objects (input or output) is in an invalid state caused  
1954                   by a previous execution error. Call GrB\_error() to access any error  
1955                   messages generated by the implementation.

1956           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1957           GrB\_INSUFFICIENT\_SPACE Not enough space in `indices` and `values` (as indicated by the `n` pa-  
1958                   rameter) to hold all of the tuples that will be extracted.

1959 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, `v`, has not been initialized by a call to  
1960                   Vector\_new or Vector\_dup.

1961           GrB\_NULL\_POINTER `indices`, `values`, or `n` pointer is NULL.

1962           GrB\_DOMAIN\_MISMATCH The domains of the `v` vector or `values` array are incompatible with  
1963                   one another.

## 1964 Description

1965 This method will extract all the tuples from the GraphBLAS vector `v`. The values associated  
1966 with those tuples are placed in the `values` array and the indices are placed in the `indices` array.  
1967 Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least  
1968 GrB\_Vector\_nvals(`v`) elements before calling this function.

1969 Upon return of this function, `n` will be set to the number of values (and indices) copied. Also, the  
1970 entries of `indices` are unique, but not necessarily sorted. Each tuple  $(i, v_i)$  in `v` is unzipped and  
1971 copied into a distinct  $k$ th location in output vectors:

$$\{\text{indices}[k], \text{values}[k]\} \leftarrow (i, v_i),$$

1972 where  $0 \leq k < \text{GrB\_Vector\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if `indices`[ $k$ ]  
1973 and `values`[ $k$ ] exist upon return, so does `indices`[ $j$ ] and `values`[ $j$ ] for all  $j$  such that  $0 \leq j < k$ .

1974 Note that if the value in `n` on input is less than the number of values contained in the vector `v`,  
1975 then a GrB\_INSUFFICIENT\_SPACE error is returned because it is undefined which subset of values  
1976 would be extracted otherwise.

1977 In both GrB\_BLOCKING mode GrB\_NONBLOCKING mode if the method exits with return value  
1978 GrB\_SUCCESS, the new contents of the arrays `indices` and `values` are as defined above.

## 1979 4.2.5 Matrix methods

### 1980 4.2.5.1 Matrix\_new: Construct new matrix

1981 Creates a new matrix with specified domain and dimensions.

## 1982 C Syntax

```
1983         GrB_Info GrB_Matrix_new(GrB_Matrix *A,  
1984                                 GrB_Type      d,  
1985                                 GrB_Index    nrows,  
1986                                 GrB_Index    ncols);
```

## 1987 Parameters

1988 **A** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1989 matrix.

1990 **d** (IN) The type corresponding to the domain of the matrix being created. Can be  
1991 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1992 GraphBLAS type.

1993 **nrows** (IN) The number of rows of the matrix being created.

1994 **ncols** (IN) The number of columns of the matrix being created.

## 1995 Return Values

1996 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1997 blocking mode, this indicates that the API checks for the input ar-  
1998 guments passed successfully. Either way, output matrix **A** is ready  
1999 to be used in the next method of the sequence.

2000 **GrB\_PANIC** Unknown internal error.

2001 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2002 GraphBLAS objects (input or output) is in an invalid state caused  
2003 by a previous execution error. Call **GrB\_error()** to access any error  
2004 messages generated by the implementation.

2005 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2006 **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2007 (needed for user-defined types).

2008 **GrB\_NULL\_POINTER** The **A** pointer is **NULL**.

2009 **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

## 2010 Description

2011 Creates a new matrix **A** of domain **D**(**d**), size **nrows**  $\times$  **ncols**, and empty **L**(**A**). The method returns  
2012 a handle to the new matrix in **A**.

2013 It is not an error to call this method more than once on the same variable; however, the handle to  
2014 the previously created object will be overwritten.

#### 2015 **4.2.5.2 Matrix\_dup: Construct a copy of a GraphBLAS matrix**

2016 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

#### 2017 **C Syntax**

```
2018         GrB_Info GrB_Matrix_dup(GrB_Matrix      *C,  
2019                                const GrB_Matrix  A);
```

#### 2020 **Parameters**

2021 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2022 matrix.

2023 A (IN) The GraphBLAS matrix to be duplicated.

#### 2024 **Return Values**

2025 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2026 blocking mode, this indicates that the API checks for the input  
2027 arguments passed successfully. Either way, output matrix C is ready  
2028 to be used in the next method of the sequence.

2029 GrB\_PANIC Unknown internal error.

2030 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2031 GraphBLAS objects (input or output) is in an invalid state caused  
2032 by a previous execution error. Call GrB\_error() to access any error  
2033 messages generated by the implementation.

2034 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2035 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2036 any matrix constructor.

2037 GrB\_NULL\_POINTER The C pointer is NULL.

#### 2038 **Description**

2039 Creates a new matrix C of domain D(A), size nrows(A) × ncols(A), and contents L(A). It returns  
2040 a handle to it in C.



2041 It is not an error to call this method more than once on the same variable; however, the handle to  
2042 the previously created object will be overwritten.

#### 2043 4.2.5.3 Matrix\_diag: Construct a diagonal GraphBLAS matrix

2044 Creates a new matrix with the same domain and contents as a GrB\_Vector, and square dimensions  
2045 appropriate for placing the contents of the vector along the specified diagonal of the matrix.

#### 2046 C Syntax

```
2047         GrB_Info GrB_Matrix_diag(GrB_Matrix      *C,  
2048                                 const GrB_Vector  v,  
2049                                 int64_t           k);
```

#### 2050 Parameters

2051 C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2052 matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

2053 v (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the  
2054 matrix.

2055 k (IN) The diagonal to which the vector is assigned. k = 0 represents the main  
2056 diagonal, k > 0 is above the main diagonal, and k < 0 is below.

#### 2057 Return Values

2058 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2059 blocking mode, this indicates that the API checks for the input  
2060 arguments passed successfully. Either way, output matrix C is ready  
2061 to be used in the next method of the sequence.

2062 GrB\_PANIC Unknown internal error.

2063 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2064 GraphBLAS objects (input or output) is in an invalid state caused  
2065 by a previous execution error. Call GrB\_error() to access any error  
2066 messages generated by the implementation.

2067 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

2068 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, v, has not been initialized by a call to  
2069 Vector\_new or Vector\_dup.

2070 GrB\_NULL\_POINTER The C pointer is NULL.

## 2071 Description

2072 Creates a new matrix **C** of domain **D(v)**, size  $(\mathbf{size}(\mathbf{v}) + |k|) \times (\mathbf{size}(\mathbf{v}) + |k|)$ , and contents

$$\begin{aligned} 2073 \quad \mathbf{L}(\mathbf{C}) &= \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or} \\ 2074 \quad \mathbf{L}(\mathbf{C}) &= \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0. \end{aligned}$$

2075 It returns a handle to it in **C**. It is not an error to call this method more than once on the same  
2076 variable; however, the handle to the previously created object will be overwritten.

## 2077 4.2.5.4 Matrix\_resize: Resize a matrix

2078 Changes the dimensions of an existing matrix.

## 2079 C Syntax

```
2080      GrB_Info GrB_Matrix_resize(GrB_Matrix C,  
2081                               GrB_Index  nrows,  
2082                               GrB_Index  ncols);
```

## 2083 Parameters

2084 **C** (INOUT) An existing Matrix object that is being resized.

2085 **nrows** (IN) The new number of rows of the matrix. It can be smaller or larger than the  
2086 current number of rows.

2087 **ncols** (IN) The new number of columns of the matrix. It can be smaller or larger than  
2088 the current number of columns.

## 2089 Return Values

2090 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2091 blocking mode, this indicates that the API checks for the input  
2092 arguments passed successfully. Either way, output matrix **C** is ready  
2093 to be used in the next method of the sequence.

2094 **GrB\_PANIC** Unknown internal error.

2095 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2096 GraphBLAS objects (input or output) is in an invalid state caused  
2097 by a previous execution error. Call **GrB\_error()** to access any error  
2098 messages generated by the implementation.

2099 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2100           GrB\_NULL\_POINTER The C pointer is NULL.

2101           GrB\_INVALID\_VALUE nrows or ncols is zero or outside the range of the type GrB\_Index.

## 2102   **Description**

2103   Changes the number of rows and columns of C to nrows and ncols, respectively. The domain  $\mathbf{D}(\mathbf{C})$   
2104   of matrix C remains the same. The contents  $\mathbf{L}(\mathbf{C})$  are modified as described below.

2105   Let  $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), M, N, \mathbf{L}(\mathbf{C}) \rangle$  when the method is called. When the method returns C is modified  
2106   to  $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \text{nrows}, \text{ncols}, \mathbf{L}'(\mathbf{C}) \rangle$  where  $\mathbf{L}'(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j, C_{ij}) \in \mathbf{L}(\mathbf{C}) \wedge (i < \text{nrows}) \wedge (j < \text{ncols})\}$ . That is, all elements of C with row index greater than or equal to nrows or column index  
2107   greater than or equal to ncols are dropped.  
2108

### 2109   **4.2.5.5   Matrix\_clear: Clear a matrix**

2110   Removes all elements (tuples) from a matrix.

## 2111   **C Syntax**

2112           GrB\_Info GrB\_Matrix\_clear(GrB\_Matrix A);

## 2113   **Parameters**

2114           A (IN) An existing GraphBLAS matrix to clear.

## 2115   **Return Values**

2116           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2117                         blocking mode, this indicates that the API checks for the input ar-  
2118                         guments passed successfully. Either way, output matrix A is ready  
2119                         to be used in the next method of the sequence.

2120           GrB\_PANIC Unknown internal error.

2121           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2122                                 GraphBLAS objects (input or output) is in an invalid state caused  
2123                                 by a previous execution error. Call GrB\_error() to access any error  
2124                                 messages generated by the implementation.

2125           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2126           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2127   any matrix constructor.

2128 **Description**

2129 Removes all elements (tuples) from an existing matrix. After the call to `GrB_Matrix_clear(A)`,  
2130  $\mathbf{L}(\mathbf{A}) = \emptyset$ . The dimensions of the matrix do not change.

2131 **4.2.5.6 Matrix\_nrows: Number of rows in a matrix**

2132 Retrieve the number of rows in a matrix.

2133 **C Syntax**

```
2134         GrB_Info GrB_Matrix_nrows(GrB_Index      *nrows,  
2135                                   const GrB_Matrix A);
```

2136 **Parameters**

2137 nrows (OUT) On successful return, contains the number of rows in the matrix.

2138 A (IN) An existing GraphBLAS matrix being queried.

2139 **Return Values**

2140 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2141 cessfully and the value of `nrows` has been set.

2142 GrB\_PANIC Unknown internal error.

2143 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2144 GraphBLAS objects (input or output) is in an invalid state caused  
2145 by a previous execution error. Call `GrB_error()` to access any error  
2146 messages generated by the implementation.

2147 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2148 any matrix constructor.

2149 GrB\_NULL\_POINTER `nrows` pointer is NULL.

2150 **Description**

2151 Return `nrows(A)` in `nrows` (the number of rows).

2152 **4.2.5.7 Matrix\_ncols: Number of columns in a matrix**

2153 Retrieve the number of columns in a matrix.

## 2154 C Syntax

```
2155         GrB_Info GrB_Matrix_ncols(GrB_Index      *ncols,  
2156                                   const GrB_Matrix A);
```

## 2157 Parameters

2158 ncols (OUT) On successful return, contains the number of columns in the matrix.

2159 A (IN) An existing GraphBLAS matrix being queried.

## 2160 Return Values

2161 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2162 cessfully and the value of ncols has been set.

2163 GrB\_PANIC Unknown internal error.

2164 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2165 GraphBLAS objects (input or output) is in an invalid state caused  
2166 by a previous execution error. Call GrB\_error() to access any error  
2167 messages generated by the implementation.

2168 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2169 any matrix constructor.

2170 GrB\_NULL\_POINTER ncols pointer is NULL.

## 2171 Description

2172 Return **ncols(A)** in ncols (the number of columns).

## 2173 4.2.5.8 Matrix\_nvals: Number of stored elements in a matrix

2174 Retrieve the number of stored elements (tuples) in a matrix.

## 2175 C Syntax

```
2176         GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,  
2177                                   const GrB_Matrix A);
```

2178 **Parameters**

2179            **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in  
2180            the matrix.

2181            **A** (IN) An existing GraphBLAS matrix being queried.

2182 **Return Values**

2183            **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2184            cessfully and the value of **nvals** has been set.

2185            **GrB\_PANIC** Unknown internal error.

2186            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2187            GraphBLAS objects (input or output) is in an invalid state caused  
2188            by a previous execution error. Call **GrB\_error()** to access any error  
2189            messages generated by the implementation.

2190            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2191            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2192            any matrix constructor.

2193            **GrB\_NULL\_POINTER** The **nvals** pointer is **NULL**.

2194 **Description**

2195 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix **A**, which is the size of  
2196 **L(A)** (see Section 3.5.3).

2197 **4.2.5.9 Matrix\_build: Store elements from tuples into a matrix**

2198 **C Syntax**

```
GrB_Info GrB_Matrix_build(GrB_Matrix      C,  
                           const GrB_Index *row_indices,  
                           const GrB_Index *col_indices,  
                           const <type>   *values,  
                           GrB_Index      n,  
                           const GrB_BinaryOp dup);
```

2199 **Parameters**

2200            **C** (INOUT) An existing Matrix object to store the result.

2201 **row\_indices** (IN) Pointer to an array of row indices.

2202 **col\_indices** (IN) Pointer to an array of column indices.

2203 **values** (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
2204 matrix, **C**.

2205 **n** (IN) The number of entries contained in each array (the same for **row\_indices**,  
2206 **col\_indices**, and **values**).

2207 **dup** (IN) An associative and commutative binary operator to apply when duplicate  
2208 values for the same location are present in the input arrays. All three domains of  
2209 **dup** must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If **dup** is **GrB\_NULL**,  
2210 then duplicate locations will result in an error.

## 2211 Return Values

2212 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2213 blocking mode, this indicates that the API checks for the input  
2214 arguments passed successfully. Either way, output matrix **C** is  
2215 ready to be used in the next method of the sequence.

2216 **GrB\_PANIC** Unknown internal error.

2217 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2218 opaque GraphBLAS objects (input or output) is in an invalid  
2219 state caused by a previous execution error. Call **GrB\_error()** to  
2220 access any error messages generated by the implementation.

2221 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2222 **GrB\_UNINITIALIZED\_OBJECT** Either **C** has not been initialized by a call to any matrix construc-  
2223 tor, or **dup** has not been initialized by a call to **GrB\_BinaryOp\_new**.

2224 **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices** or **values** pointer is **NULL**.

2225 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** or **col\_indices** is outside the allowed range  
2226 for **C**.

2227 **GrB\_DOMAIN\_MISMATCH** Either the domains of the GraphBLAS binary operator **dup** are  
2228 not all the same, or the domains of **values** and **C** are incompatible  
2229 with each other or  $D_{dup}$ .

2230 **GrB\_OUTPUT\_NOT\_EMPTY** Output matrix **C** already contains valid tuples (elements). In  
2231 other words, **GrB\_Matrix\_nvals(C)** returns a positive value.

2232 **GrB\_INVALID\_VALUE** indices contains a duplicate location and **dup** is **GrB\_NULL**.

## 2233 Description

2234 If `dup` is not `GrB_NULL`, an internal matrix  $\tilde{\mathbf{C}} = \langle D_{dup}, \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$  is created, which  
 2235 only differs from  $\mathbf{C}$  in its domain; otherwise,  $\tilde{\mathbf{C}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \emptyset \rangle$ .

2236 Each tuple  $\{\text{row\_indices}[k], \text{col\_indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the  
 2237 output in the form of

$$2238 \quad \tilde{\mathbf{C}}(\text{row\_indices}[k], \text{col\_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{C})) \text{values}[k] & \text{otherwise.} \end{cases}$$

2239 If multiple values for the same location are present in the input arrays and `dup` is not `GrB_NULL`,  
 2240 `dup` is used to reduce the values before assignment into  $\tilde{\mathbf{C}}$  as follows:

$$2241 \quad \tilde{\mathbf{C}}_{ij} = \bigoplus_{k: \text{row\_indices}[k]=i \wedge \text{col\_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2242 where  $\oplus$  is the `dup` binary operator. Finally, the resulting  $\tilde{\mathbf{C}}$  is copied into  $\mathbf{C}$  via typecasting its  
 2243 values to  $\mathbf{D}(\mathbf{C})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

2244 The nonopaque input arrays `row_indices`, `col_indices`, and `values` must be at least as large as `n`.

2245 It is an error to call this function on an output object with existing elements. In other words,  
 2246 `GrB_Matrix_nvals(C)` should evaluate to zero prior to calling this function.

2247 After `GrB_Matrix_build` returns, it is safe for a programmer to modify or delete the arrays `row_indices`,  
 2248 `col_indices`, or `values`.

### 2249 4.2.5.10 Matrix\_setElement: Set a single element in matrix

2250 Set one element of a matrix to a given value.

## 2251 C Syntax

```
2252 // scalar value
2253 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2254                                <type>         val,
2255                                GrB_Index        row_index,
2256                                GrB_Index        col_index);
2257
2258 // GraphBLAS scalar
2259 GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2260                                const GrB_Scalar s,
2261                                GrB_Index        row_index,
2262                                GrB_Index        col_index);
```



## 2263 Parameters

2264           **C** (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.  
2265           **val** or **s** (IN) Scalar to assign. Its domain (type) must be compatible with the domain of  
2266           **C**.  
2267           **row\_index** (IN) Row index of element to be assigned  
2268           **col\_index** (IN) Column index of element to be assigned

## 2269 Return Values

2270           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2271           blocking mode, this indicates that the compatibility tests on in-  
2272           dex/dimensions and domains for the input arguments passed suc-  
2273           cessfully. Either way, the output matrix **C** is ready to be used in  
2274           the next method of the sequence.  
2275           **GrB\_PANIC** Unknown internal error.  
2276           **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2277           GraphBLAS objects (input or output) is in an invalid state caused  
2278           by a previous execution error. Call **GrB\_error()** to access any error  
2279           messages generated by the implementation.  
2280           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.  
2281 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been  
2282           initialized by a call to a respective constructor.  
2283           **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e., not less  
2284           than **nrows(C)** or **ncols(C)**, respectively).  
2285           **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and the scalar are incompatible.

## 2286 Description

2287 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or  
2288 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from  
2289 one domain can be cast to values in the other domain as per the rules of the C language. In  
2290 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
2291 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
2292 **GrB\_Matrix\_setElement** ends and the domain mismatch error listed above is returned.

2293 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2294 \quad & 0 \leq \text{row\_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col\_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2295 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid  
 2296 index error listed above is returned.

We are now ready to carry out the assignment; that is:

$$C(\text{row\_index}, \text{col\_index}) = \begin{cases} \mathbf{L}(s), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

2297 In the case of a transparent scalar or if  $\mathbf{L}(s)$  is not empty, then a value will be stored at the  
 2298 specified location in  $C$ , overwriting any value that may have been stored there before. In the case  
 2299 of a GraphBLAS scalar and if  $\mathbf{L}(s)$  is empty, then any value stored at the specified location in  $C$   
 2300 will be removed.

2301 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
 2302 of  $C$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
 2303 return value `GrB_SUCCESS` and the new content of vector  $C$  is as defined above but may not be  
 2304 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 2305 **4.2.5.11 Matrix\_removeElement: Remove an element from a matrix**

2306 Remove (annihilate) one stored element from a matrix.

#### 2307 **C Syntax**

```
2308      GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,
2309                                         GrB_Index   row_index,
2310                                         GrB_Index   col_index);
```

#### 2311 **Parameters**

2312  $C$  (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2313  $\text{row\_index}$  (IN) Row index of element to be removed

2314  $\text{col\_index}$  (IN) Column index of element to be removed

#### 2315 **Return Values**

2316 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 2317 blocking mode, this indicates that the compatibility tests on in-  
 2318 dex/dimensions and domains for the input arguments passed suc-  
 2319 cessfully. Either way, the output matrix  $C$  is ready to be used in  
 2320 the next method of the sequence.

2321 `GrB_PANIC` Unknown internal error.

2322        GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 2323        GraphBLAS objects (input or output) is in an invalid state caused  
 2324        by a previous execution error. Call GrB\_error() to access any error  
 2325        messages generated by the implementation.

2326        GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2327 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to  
 2328        any matrix constructor.

2329        GrB\_INVALID\_INDEX row\_index or col\_index is outside the allowable range (i.e., not less  
 2330        than nrows(C) or ncols(C), respectively).

## 2331 Description

2332 First, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2333 \quad & 0 \leq \text{row\_index} < \text{nrows}(\mathbf{C}), \\ & 0 \leq \text{col\_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2334 If either of these conditions is violated, execution of GrB\_Matrix\_removeElement ends and the  
 2335 invalid index error listed above is returned.

2336 We are now ready to carry out the removal of a value that may be stored at the location specified by  
 2337 (row\_index, col\_index). If a value does not exist at the specified location in C, no error is reported  
 2338 and the operation has no effect on the state of C. In either case, the following will be true on return  
 2339 from this method: (row\_index, col\_index)  $\notin$  ind(C)

2340 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 2341 of C is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 2342 return value GrB\_SUCCESS and the new content of vector C is as defined above but may not be  
 2343 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2344 4.2.5.12 Matrix\_extractElement: Extract a single element from a matrix

2345 Extract one element of a matrix into a scalar.

## 2346 C Syntax

```
2347 // scalar value
2348 GrB_Info GrB_Matrix_extractElement(<type>          *val,
2349                                   const GrB_Matrix A,
2350                                   GrB_Index         row_index,
2351                                   GrB_Index         col_index);
2352
2353 // GraphBLAS scalar
```

```

2354         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2355                                             const GrB_Matrix A,
2356                                             GrB_Index      row_index,
2357                                             GrB_Index      col_index);
2358

```

## 2359 Parameters

2360     **val or s** (INOUT) An existing scalar whose domain is compatible with the domain of matrix  
2361     **A**. On successful return, this scalar holds the result of the extract. Any previous  
2362     value stored in **val** or **s** is overwritten.

2363     **A** (IN) The GraphBLAS matrix from which an element is extracted.

2364     **row\_index** (IN) The row index of location in **A** to extract.

2365     **col\_index** (IN) The column index of location in **A** to extract.

## 2366 Return Values

2367     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2368     cessfully. This indicates that the compatibility tests on dimensions  
2369     and domains for the input arguments passed successfully, and the  
2370     output scalar, **val** or **s**, has been computed and is ready to be used  
2371     in the next method of the sequence.

2372     **GrB\_NO\_VALUE** When using the transparent scalar, **val**, this is returned when there  
2373     is no stored value at specified location.

2374     **GrB\_PANIC** Unknown internal error.

2375     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2376     GraphBLAS objects (input or output) is in an invalid state caused  
2377     by a previous execution error. Call **GrB\_error()** to access any error  
2378     messages generated by the implementation.

2379     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2380     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by  
2381     a call to a corresponding constructor.

2382     **GrB\_NULL\_POINTER** **val** pointer is **NULL**.

2383     **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e. less than  
2384     zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-  
2385     tively).

2386     **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and scalar are incompatible.

## 2387 Description

2388 First, the scalar and input matrix are tested for domain compatibility as follows:  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\mathbf{s})$   
 2389 must be compatible with  $\mathbf{D}(\mathbf{A})$ . Two domains are compatible with each other if values from  
 2390 one domain can be cast to values in the other domain as per the rules of the C language. In  
 2391 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 2392 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 2393 `GrB_Matrix_extractElement` ends and the domain mismatch error listed above is returned.

2394 Then, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2395 \quad & 0 \leq \text{row\_index} < \mathbf{nrows}(\mathbf{A}), \\ & 0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{A}) \end{aligned}$$

2396 If either condition is violated, execution of `GrB_Matrix_extractElement` ends and the invalid index  
 2397 error listed above is returned.

We are now ready to carry out the extract into the output scalar; that is,

$$\left. \begin{array}{l} \mathbf{L}(\mathbf{s}) \\ \text{val} \end{array} \right\} = \mathbf{A}(\text{row\_index}, \text{col\_index})$$

2398 If  $(\text{row\_index}, \text{col\_index}) \in \mathbf{ind}(\mathbf{A})$ , then the corresponding value from  $\mathbf{A}$  is copied into  $\mathbf{s}$  or  $\text{val}$   
 2399 with casting as necessary. If  $(\text{row\_index}, \text{col\_index}) \notin \mathbf{ind}(\mathbf{A})$ , then one of the follow occurs  
 2400 depending on output scalar type:

- 2401 • The GraphBLAS scalar,  $\mathbf{s}$ , is cleared and `GrB_SUCCESS` is returned.
- 2402 • The non-opaque scalar,  $\text{val}$ , is unchanged, and `GrB_NO_VALUE` is returned.

2403 When using the non-opaque scalar variant ( $\text{val}$ ) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
 2404 mode, the new contents of  $\text{val}$  are as defined above if the method exits with return value `GrB_SUCCESS`  
 2405 or `GrB_NO_VALUE`.

2406 When using the GraphBLAS scalar variant ( $\mathbf{s}$ ) with a `GrB_SUCCESS` return value, the method  
 2407 exits and the new contents of  $\mathbf{s}$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
 2408 In `GrB_NONBLOCKING` mode, the new contents of  $\mathbf{s}$  is as defined above but may not be fully  
 2409 computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2410 4.2.5.13 Matrix\_extractTuples: Extract tuples from a matrix

2411 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

## 2412 C Syntax

```
2413      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2414                                     GrB_Index      *col_indices,
```

```

2415                                     <type>          *values,
2416                                     GrB_Index         *n,
2417                                     const GrB_Matrix   A);

```

## 2418 Parameters

2419     **row\_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the  
2420         row indices.

2421     **col\_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the  
2422         column indices.

2423     **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
2424         the stored values whose type is compatible with  $\mathbf{D}(\mathbf{A})$ .

2425     **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,  
2426         **row\_indices**, and **col\_indices** arrays can hold. Upon return, it will contain the  
2427         number of values written to the arrays.

2428     **A** (IN) An existing GraphBLAS matrix.

## 2429 Return Values

2430     **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2431         cessfully. This indicates that the compatibility tests on the input  
2432         argument passed successfully, and the output arrays, **indices** and  
2433         **values**, have been computed.

2434     **GrB\_PANIC** Unknown internal error.

2435     **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
2436         GraphBLAS objects (input or output) is in an invalid state caused  
2437         by a previous execution error. Call **GrB\_error()** to access any error  
2438         messages generated by the implementation.

2439     **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2440     **GrB\_INSUFFICIENT\_SPACE** Not enough space in **row\_indices**, **col\_indices**, and **values** (as indi-  
2441         cated by the **n** parameter) to hold all of the tuples that will be  
2442         extracted.

2443     **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2444         any matrix constructor.

2445     **GrB\_NULL\_POINTER** **row\_indices**, **col\_indices**, **values** or **n** pointer is NULL.

2446     **GrB\_DOMAIN\_MISMATCH** The domains of the **A** matrix and **values** array are incompatible  
2447         with one another.

## 2448 Description

2449 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with  
2450 those tuples are placed in the **values** array, the column indices are placed in the **col\_indices** array,  
2451 and the row indices are placed in the **row\_indices** array. These output arrays are pre-allocated by  
2452 the user before calling this function such that each output array has enough space to hold at least  
2453 **GrB\_Matrix\_nvals(A)** elements.

2454 Upon return of this function, a pair of  $\{\text{row\_indices}[k], \text{col\_indices}[k]\}$  are unique for every valid  
2455  $k$ , but they are not required to be sorted in any particular order. Each tuple  $(i, j, A_{ij})$  in **A** is  
2456 unzipped and copied into a distinct  $k$ th location in output vectors:

$$\{\text{row\_indices}[k], \text{col\_indices}[k], \text{values}[k]\} \leftarrow (i, j, A_{ij}),$$

2457 where  $0 \leq k < \text{GrB\_Matrix\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if **row\_indices**[ $k$ ],  
2458 **col\_indices**[ $k$ ] and **values**[ $k$ ] exist upon return, so does **row\_indices**[ $j$ ], **col\_indices**[ $j$ ] and **values**[ $j$ ] for  
2459 all  $j$  such that  $0 \leq j < k$ .

2460 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,  
2461 then a **GrB\_INSUFFICIENT\_SPACE** error is returned since it is undefined which subset of values  
2462 would be extracted.

2463 In both **GrB\_BLOCKING** mode **GrB\_NONBLOCKING** mode if the method exits with return value  
2464 **GrB\_SUCCESS**, the new contents of the arrays **row\_indices**, **col\_indices** and **values** are as defined  
2465 above.

2466 **4.2.5.14 Matrix\_exportHint: Provide a hint as to which storage format might be most**  
2467 **efficient for exporting a matrix**

## 2468 C Syntax

```
GrB_Info GrB_Matrix_exportHint(GrB_Format      *hint,  
                               GrB_Matrix      A);
```

## 2469 Parameters

2470 **hint** (OUT) Pointer to a value of type **GrB\_Format**.

2471 **A** (IN) A GraphBLAS matrix object.

## 2472 Return Values

2473 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2474 cessfully and the value of **hint** has been set.

2475 **GrB\_PANIC** Unknown internal error.

2476           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 2477                                   opaque GraphBLAS objects (input or output) is in an invalid  
 2478                                   state caused by a previous execution error. Call GrB\_error() to  
 2479                                   access any error messages generated by the implementation.

2480           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2481           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
 2482                                   any matrix constructor.

2483           GrB\_NULL\_POINTER hint is NULL.

2484           GrB\_NO\_VALUE If the implementation does not have a preferred format, it may  
 2485                                   return the value GrB\_NO\_VALUE.

## 2486 Description

2487 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for  
 2488 exporting the matrix A. GraphBLAS implementations might return the current storage format of  
 2489 the matrix, or the format to which it could most efficiently be exported. However, implementations  
 2490 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is  
 2491 free to refuse to provide a format hint, returning GrB\_NO\_VALUE.

### 2492 4.2.5.15 Matrix\_exportSize: Return the array sizes necessary to export a GraphBLAS 2493 matrix object

## 2494 C Syntax

```

GrB_Info GrB_Matrix_exportSize(GrB_Index      *n_indptr,
                               GrB_Index      *n_indices,
                               GrB_Index      *n_values,
                               GrB_Format     format,
                               GrB_Matrix     A);

```

## 2495 Parameters

2496           n\_indptr (OUT) Pointer to a value of type GrB\_Index.

2497           n\_indices (OUT) Pointer to a value of type GrB\_Index.

2498           n\_values (OUT) Pointer to a value of type GrB\_Index.

2499           format (IN) a value indicating the format in which the matrix will be exported, as defined  
 2500                                   in Section 3.5.3.1.

2501           A (IN) A GraphBLAS matrix object.



## 2502 Return Values

2503                   GrB\_SUCCESS In blocking mode or non-blocking mode, the operation com-  
2504                   pleted successfully. This indicates that the API checks for the  
2505                   input arguments passed successfully, and the number of elements  
2506                   necessary for the export buffers have been written to `n_indptr`,  
2507                   `n_indices`, and `n_values`, respectively.

2508                   GrB\_PANIC Unknown internal error.

2509                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2510                   opaque GraphBLAS objects (input or output) is in an invalid  
2511                   state caused by a previous execution error. Call `GrB_error()` to  
2512                   access any error messages generated by the implementation.

2513                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2514                   GrB\_UNINITIALIZED\_OBJECT The GraphBLAS Matrix, `A`, has not been initialized by a call to  
2515                   any matrix constructor.

2516                   GrB\_NULL\_POINTER `n_indptr`, `n_indices`, or `n_values` is NULL.

## 2517 Description

2518 Given a matrix `A`, returns the required capacities of arrays `values`, `indptr`, and `indices` necessary to  
2519 export the matrix in the format specified by `format`. The output values `n_values`, `n_indptr`, and  
2520 `indices` will contain the corresponding sizes of the arrays (in number of elements) that must be  
2521 allocated to hold the exported matrix. The argument `format` can be chosen arbitrarily by the user  
2522 as one of the values defined in Section 3.5.3.1.

### 2523 4.2.5.16 Matrix\_export: Export a GraphBLAS matrix to a pre-defined format

## 2524 C Syntax

```
GrB_Info GrB_Matrix_export(GrB_Index      *indptr,  
                           GrB_Index      *indices,  
                           <type>         *values,  
                           GrB_Index      *n_indptr,  
                           GrB_Index      *n_indices,  
                           GrB_Index      *n_values,  
                           GrB_Format     format,  
                           GrB_Matrix     A);
```

## 2525 Parameters

2526        **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-  
2527        dices, depending on the value of **format**. It must be large enough to hold at  
2528        least **n\_indptr** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2529        **GrB\_Matrix\_exportSize()** method.

2530        **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements  
2531        in **values**, depending on the value of **format**. It must be large enough to hold at  
2532        least **n\_indices** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2533        **GrB\_Matrix\_exportSize()** method.

2534        **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-  
2535        ment must match the type of the values stored in **A**. It must be large enough  
2536        to hold at least **n\_values** elements of that type, where **n\_values** was returned from  
2537        **GrB\_Matrix\_exportSize**.

2538        **n\_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**  
2539        array can hold. Upon return, it will contain the number of elements written to the  
2540        array.

2541        **n\_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**  
2542        array can hold. Upon return, it will contain the number of elements written to the  
2543        array.

2544        **n\_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**  
2545        array can hold. Upon return, it will contain the number of elements written to the  
2546        array.

2547        **format** (IN) a value indicating the format in which the matrix will be exported, as defined  
2548        in Section 3.5.3.1.

2549        **A** (IN) A GraphBLAS matrix object.

## 2550 Return Values

2551        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2552        cessfully. This indicates that the compatibility tests on the input  
2553        argument passed successfully, and the output arrays, **indptr**, **in-**  
2554        **dices** and **values**, have been computed.

2555        **GrB\_PANIC** Unknown internal error.

2556        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2557        opaque GraphBLAS objects (input or output) is in an invalid  
2558        state caused by a previous execution error. Call **GrB\_error()** to  
2559        access any error messages generated by the implementation.

2560        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.



2585        **nrows** (IN) Integer value holding the number of rows in the matrix.

2586        **ncols** (IN) Integer value holding the number of columns in the matrix.

2587        **indptr** (IN) Pointer to an array of row or column offsets, or row indices, depending on the  
2588                value of **format**.

2589        **indices** (IN) Pointer to an array row or column indices of the elements in **values**, depending  
2590                on the value of **format**.

2591        **values** (IN) Pointer to an array of values. Type must match the type of **d**.

2592        **n\_indptr** (IN) Integer value holding the number of elements in the array pointed to by **indptr**.

2593        **n\_indices** (IN) Integer value holding the number of elements in the array pointed to by **indices**.

2594        **n\_values** (IN) Integer value holding the number of elements in the array pointed to by **values**.

2595        **format** (IN) a value indicating the format of the matrix being imported, as defined in  
2596                Section 3.5.3.1.

## 2597 **Return Values**

2598        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2599                blocking mode, this indicates that the API checks for the input  
2600                arguments passed successfully and the input arrays have been  
2601                consumed. Either way, output matrix **A** is ready to be used in  
2602                the next method of the sequence.

2603        **GrB\_PANIC** Unknown internal error.

2604        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2605        **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
2606                (needed for user-defined types).

2607        **GrB\_NULL\_POINTER** **A**, **indptr**, **indices** or **values** pointer is **NULL**.

2608        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indptr** or **indices** is outside the allowed range for indices  
2609                in **A** and or the size of **values**, **n\_values**, depending on the value  
2610                of **format**.

2611        **GrB\_INVALID\_VALUE** **nrows** or **ncols** is zero or outside the range of the type **GrB\_Index**.

2612        **GrB\_DOMAIN\_MISMATCH** The domain given in parameter **d** does not match the element  
2613                type of **values**.

## 2614 Description

2615 Creates a new matrix **A** of domain **D**(d) and dimension **nrows**  $\times$  **ncols**. The new GraphBLAS  
2616 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.  
2617 The method returns a handle to the new matrix in **A**. The structure of the data being imported is  
2618 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of  
2619 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2620 It is not an error to call this method more than once on the same output matrix; however, the  
2621 handle to the previously created object will be overwritten.

## 2622 4.2.5.18 Matrix\_serializeSize: Compute the serialize buffer size

2623 Compute the buffer size (in bytes) necessary to serialize a GrB\_Matrix using GrB\_Matrix\_serialize.

## 2624 C Syntax

```
GrB_Info GrB_Matrix_serializeSize(GrB_Index *size,  
                                  GrB_Matrix A);
```

## 2625 Parameters

2626 **size** (OUT) Pointer to GrB\_Index value where size in bytes of serialized object will be  
2627 written.

2628 **A** (IN) A GraphBLAS matrix object.

## 2629 Return Values

2630 **GrB\_SUCCESS** The operation completed successfully and the value pointed to  
2631 by **\*size** has been computed and is ready to use.

2632 **GrB\_PANIC** Unknown internal error.

2633 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2634 **GrB\_NULL\_POINTER** **size** is NULL.

## 2635 Description

2636 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object **A**.  
2637 Users may then allocate a buffer of **size** bytes to pass as a parameter to GrB\_Matrix\_serialize.

2638 **4.2.5.19 Matrix\_serialize: Serialize a GraphBLAS matrix.**

2639 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2640 **C Syntax**

```
GrB_Info GrB_Matrix_serialize(void      *serialized_data,  
                               GrB_Index *serialized_size,  
                               GrB_Matrix A);
```

2641 **Parameters**

2642 **serialized\_data** (INOUT) Pointer to the preallocated buffer where the serialized matrix will be  
2643 written.

2644 **serialized\_size** (INOUT) On input, the size in bytes of the buffer pointed to by **serialized\_data**.  
2645 On output, the number of bytes written to **serialized\_data**.

2646 **A** (IN) A GraphBLAS matrix object.

2647 **Return Values**

2648 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2649 cessfully. This indicates that the compatibility tests on the in-  
2650 put argument passed successfully, and the output buffer **serial-  
2651 ized\_data** and **serialized\_size**, have been computed and are ready  
2652 to use.

2653 **GrB\_PANIC** Unknown internal error.

2654 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2655 opaque GraphBLAS objects (input or output) is in an invalid  
2656 state caused by a previous execution error. Call **GrB\_error()** to  
2657 access any error messages generated by the implementation.

2658 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2659 **GrB\_NULL\_POINTER** **serialized\_data** or **serialize\_size** is NULL.

2660 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2661 any matrix constructor.

2662 **GrB\_INSUFFICIENT\_SPACE** The size of the buffer **serialized\_data** (provided as an input **seri-  
2663 alized\_size**) was not large enough.

## 2664 Description

2665 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,  
2666 the size of the buffer pointed to by `serialized_data`, provided as an input by `serialized_size`, must  
2667 be of at least the number of bytes returned from `GrB_Matrix_serializeSize`. The actual size of the  
2668 serialized matrix written to `serialized_data` is provided upon completion as an output written to  
2669 `serialized_size`.

2670 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created  
2671 with one library implementation is not necessarily valid for deserialization with another implemen-  
2672 tation.

### 2673 4.2.5.20 Matrix\_deserialize: Deserialize a GraphBLAS matrix.

2674 Construct a new GraphBLAS matrix from a serialized object.

## 2675 C Syntax

```
GrB_Info GrB_Matrix_deserialize(GrB_Matrix *A,  
                                GrB_Type   d,  
                                const void *serialized_data,  
                                GrB_Index   serialized_size);
```

## 2676 Parameters

2677 A (INOUT) On a successful return, contains a handle to the newly created Graph-  
2678 BLAS matrix.

2679 d (IN) the type of the matrix that was serialized in `serialized_data`.

2680 `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2681 `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

## 2682 Return Values

2683 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2684 blocking mode, this indicates that the API checks for the input  
2685 arguments passed successfully. Either way, output matrix A is  
2686 ready to be used in the next method of the sequence.

2687 GrB\_PANIC Unknown internal error.

2688 GrB\_INVALID\_OBJECT This is returned if `serialized_data` is invalid or corrupted.

2689 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2690 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
2691 (needed for user-defined types).

2692 GrB\_NULL\_POINTER serialized\_data or A is NULL.

2693 GrB\_DOMAIN\_MISMATCH The type given in d does not match the type of the matrix  
2694 serialized in serialized\_data.

## 2695 Description

2696 Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object  
2697 pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The  
2698 domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized  
2699 in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2700 Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix  
2701 serialized in one library implementation can be deserialized by another.

2702 It is not an error to call this method more than once on the same output matrix; however, the  
2703 handle to the previously created object will be overwritten.

## 2704 4.2.6 Descriptor methods

2705 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-  
2706 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

### 2707 4.2.6.1 Descriptor\_new: Create new descriptor

2708 Creates a new (empty or default) descriptor.

## 2709 C Syntax

2710 GrB\_Info GrB\_Descriptor\_new(GrB\_Descriptor \*desc);

## 2711 Parameters

2712 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2713 descriptor.

## 2714 Return Value

2715 GrB\_SUCCESS The method completed successfully.

2716 GrB\_PANIC unknown internal error.



2717        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2718        GrB\_NULL\_POINTER desc pointer is NULL.

## 2719 **Description**

2720        Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can  
2721        be populated by calls to Descriptor\_set.

2722        It is not an error to call this method more than once on the same variable; however, the handle to  
2723        the previously created object will be overwritten.

## 2724 **4.2.6.2 Descriptor\_set: Set content of descriptor**

2725        Sets the content for a field for an existing descriptor.

## 2726 **C Syntax**

```
2727        GrB_Info GrB_Descriptor_set(GrB_Descriptor        desc,  
2728                                    GrB_Desc_Field        field,  
2729                                    GrB_Desc_Value        val);
```

## 2730 **Parameters**

2731        desc (IN) An existing GraphBLAS descriptor to be modified.

2732        field (IN) The field being set.

2733        val (IN) New value for the field being set.

## 2734 **Return Values**

2735        GrB\_SUCCESS operation completed successfully.

2736        GrB\_PANIC unknown internal error.

2737        GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2738        GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

2739        GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 2740 Description

2741 For a given descriptor, the `GrB_Descriptor_set` method can be called for each field in the descriptor  
2742 to set the value associated with that field. Valid values for the `field` parameter include the following:

2743 `GrB_OUTP` refers to the output parameter (result) of the operation.

2744 `GrB_MASK` refers to the mask parameter of the operation.

2745 `GrB_INP0` refers to the first input parameters of the operation (matrices and vectors).

2746 `GrB_INP1` refers to the second input parameters of the operation (matrices and vectors).

2747 Valid values for the `val` parameter are:

2748 `GrB_STRUCTURE` Use only the structure of the stored values of the corresponding mask  
2749 (`GrB_MASK`) parameter.

2750 `GrB_COMP` Use the complement of the corresponding mask (`GrB_MASK`) param-  
2751 eter. When combined with `GrB_STRUCTURE`, the complement of the  
2752 structure of the mask is used without evaluating the values stored.

2753 `GrB_TRAN` Use the transpose of the corresponding matrix parameter (valid for input  
2754 matrix parameters only).

2755 `GrB_REPLACE` When assigning the masked values to the output matrix or vector, clear  
2756 the matrix first (or clear the non-masked entries). The default behavior  
2757 is to leave non-masked locations unchanged. Valid for the `GrB_OUTP`  
2758 parameter only.

2759 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of `GrB_MASK`,  
2760 multiple values can be set and all will apply (for example, both `GrB_COMP` and `GrB_STRUCTURE`).  
2761 A value for a given field may be set multiple times but will have no additional effect. Fields that  
2762 have no values set result in their default behavior, as defined in Section 3.7.

## 2763 4.2.7 free: Destroy an object and release its resources

2764 Destroys a previously created GraphBLAS object and releases any resources associated with the  
2765 object.

## 2766 C Syntax

2767 `GrB_Info GrB_free(<GrB_Object> *obj);`

## 2768 Parameters

2769       obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have  
2770       been created by an explicit call to a GraphBLAS constructor. It can be any of the  
2771       opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,  
2772       binary op, unary op, or type. On successful completion of GrB\_free, obj behaves  
2773       as an uninitialized object.

## 2774 Return Values

2775       GrB\_SUCCESS operation completed successfully

2776       GrB\_PANIC unknown internal error. If this return value is encountered when  
2777       in nonblocking mode, the error responsible for the panic condition  
2778       could be from any method involved in the computation of the input  
2779       object. The GrB\_error() method should be called for additional  
2780       information.

## 2781 Description

2782 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime  
2783 system. A call to GrB\_free frees those resources so they are available for use by other GraphBLAS  
2784 objects.

2785 The parameter passed into GrB\_free is a handle referencing a GraphBLAS opaque object of a data  
2786 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-  
2787 structor. The behavior of a program that calls GrB\_free on a pre-defined object is implementation  
2788 defined.

2789 After the GrB\_free method returns, the object referenced by the input handle is destroyed and the  
2790 handle has the value GrB\_INVALID\_HANDLE. The handle can be used in subsequent GraphBLAS  
2791 methods but only after the handle has been reinitialized with a call the the appropriate \_new or  
2792 \_dup method.

2793 Note that unlike other GraphBLAS methods, calling GrB\_free with an object with an invalid handle  
2794 is legal. The system may attempt to free resources that might be associated with that object, if  
2795 possible, and return normally.

2796 When using GrB\_free it is possible to create a dangling reference to an object. This would occur  
2797 when a handle is assigned to a second variable of the same opaque type. This creates two handles  
2798 that reference the same object. If GrB\_free is called with one of the variables, the object is destroyed  
2799 and the handle associated with the other variable no longer references a valid object. This is not an  
2800 error condition that the implementation of the GraphBLAS API can be expected to catch, hence  
2801 programmers must take care to prevent this situation from occurring.

2802 **4.2.8 wait: Return once an object is either *complete* or *materialized***

2803 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

2804 **C Syntax**

2805 `GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);`

2806 **Parameters**

2807 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an  
2808 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2809 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2810 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another  
2811 thread (completion) or all computing to produce `obj` by all GraphBLAS operations  
2812 in its sequence have finished (materialization).

2813 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the  
2814 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or  
2815 `GrB_MATERIALIZE`.

2816 **Return values**

2817 `GrB_SUCCESS` operation completed successfully.

2818 `GrB_INDEX_OUT_OF_BOUNDS` an index out-of-bounds execution error happened during com-  
2819 pletion of pending operations.

2820 `GrB_OUT_OF_MEMORY` and out-of-memory execution error happened during completion  
2821 of pending operations.

2822 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2823 or other constructor, method.

2824 `GrB_PANIC` unknown internal error.

2825 `GrB_INVALID_VALUE` method called with a `GrB_WaitMode` other than `GrB_COMPLETE`  
2826 `GrB_MATERIALIZE`.

2827 **Description**

2828 On successful return from `GrB_wait()`, the input object, `obj` is in one of two states depending on  
2829 the mode of `GrB_wait`:

- 2830 • *complete*: `obj` can be used in a happens-before relation, so in a properly synchronized program  
2831 it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another  
2832 thread. This result occurs when the mode parameter is set to `GrB_COMPLETE`.
- 2833 • *materialized*: `obj` is *complete*, but in addition, no further computing will be carried out on  
2834 behalf of `obj` and error information is available. This result occurs when the mode parameter  
2835 is set to `GrB_MATERIALIZE`.

2836 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,  
2837 `GrB_wait(obj,mode)` has no effect when called in blocking mode.

2838 In non-blocking mode, the status of any pending method calls, other than those associated with pro-  
2839 ducing the *complete* or *materialized* state of `obj`, are not impacted by the call to `GrB_wait(obj,mode)`.  
2840 Methods in the sequence for `obj`, however, most likely would be impacted by a call to `GrB_wait(obj,mode)`;  
2841 especially in the case of the *materialized* mode for which any computing on behalf of `obj` must be  
2842 finished prior to the return from `GrB_wait(obj,mode)`.

#### 2843 4.2.9 error: Retrieve an error string

2844 Retrieve an error-message about any errors encountered during the processing associated with an  
2845 object.

### 2846 C Syntax

```
2847      GrB_Info GrB_error(const char      **error,
2848                        const GrB_Object  obj);
```

#### 2849 Parameters

2850 `error` (OUT) A pointer to a null-terminated string. The contents of the string are im-  
2851 plementation defined.

2852 `obj` (IN) An existing GraphBLAS object. The object must have been created by an  
2853 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2854 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2855 or type.

#### 2856 Return value

2857 `GrB_SUCCESS` operation completed successfully.

2858 `GrB_UNINITIALIZED_OBJECT` object has not been initialized by a call to the respective `*_new`,  
2859 or other constructor, method.

2860 `GrB_PANIC` unknown internal error.

## Description

This method retrieves a message related to any errors that were encountered during the last GraphBLAS method that had the opaque GraphBLAS object, `obj`, as an OUT or INOUT parameter. The function returns a pointer to a null-terminated string and the contents of that string are implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error string. The string that is returned is owned by `obj` and will be valid until the next time `obj` is used as an OUT or INOUT parameter or the object is freed by a call to `GrB_free(obj)`. This is a thread-safe function. It can be safely called by multiple threads for the same object in a race-free program.

## 4.3 GraphBLAS operations

The GraphBLAS operations are defined in the GraphBLAS math specification and summarized in Table 4.1. In addition to methods that implement these fundamental GraphBLAS operations, we support a number of variants that have been found to be especially useful in algorithm development. A flowchart of the overall behavior of a GraphBLAS operation is shown in Figure 4.1.

### Domains and Casting

A GraphBLAS operation is only valid when the domains of the GraphBLAS objects are mathematically consistent. The C programming language defines implicit casts between built-in data types. For example, floats, doubles, and ints can be freely mixed according to the rules defined for implicit casts. It is the responsibility of the user to assure that these casts are appropriate for the algorithm in question. For example, a cast to int implies truncation of a floating point type. Depending on the operation, this truncation error could lead to erroneous results. Furthermore, casting a wider type onto a narrower type can lead to overflow errors. The GraphBLAS operations do not attempt to protect a user from these sorts of errors.

When user-defined types are involved, however, GraphBLAS requires strict equivalence between types and no casting is supported. If GraphBLAS detects these mismatches, it will return a domain mismatch error.

### Dimensions and Transposes

GraphBLAS operations also make assumptions about the numbers of dimensions and the sizes of vectors and matrices in an operation. An operation will test these sizes and report an error if they are not *shape compatible*. For example, when multiplying two matrices,  $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ , the number of rows of  $\mathbf{C}$  must equal the number of rows of  $\mathbf{A}$ , the number of columns of  $\mathbf{A}$  must match the number of rows of  $\mathbf{B}$ , and the number of columns of  $\mathbf{C}$  must match the number of columns of  $\mathbf{B}$ . This is the behavior expected given the mathematical definition of the operations.

For most of the GraphBLAS operations involving matrices, an optional descriptor can modify the matrix associated with an input GraphBLAS matrix object. For example, if an input matrix is an

Table 4.1: A mathematical notation for the fundamental GraphBLAS operations supported in this specification. Input matrices  $\mathbf{A}$  and  $\mathbf{B}$  may be optionally transposed (not shown). Use of an optional accumulate with existing values in the output object is indicated with  $\odot$ . Use of optional write masks and replace flags are indicated as  $\mathbf{C}\langle\mathbf{M}, r\rangle$  when applied to the output matrix,  $\mathbf{C}$ . The mask controls which values resulting from the operation on the right-hand side are written into the output object (complement and structure flags are not shown). The “replace” option, indicated by specifying the  $r$  flag, means that all values in the output object are removed prior to assignment. If “replace” is not specified, only the values/locations computed on the right-hand side and allowed by the mask will be written to the output (“merge” mode).

| Operation Name  | Mathematical Notation                         |   |   |
|-----------------|---|---|---|
| mxm             | $\mathbf{C}\langle\mathbf{M}, r\rangle$       | = | $\mathbf{C} \odot \mathbf{A} \oplus . \otimes \mathbf{B}$                               |
| mxv             | $\mathbf{w}\langle\mathbf{m}, r\rangle$       | = | $\mathbf{w} \odot \mathbf{A} \oplus . \otimes \mathbf{u}$                               |
| vxm             | $\mathbf{w}^T\langle\mathbf{m}^T, r\rangle$   | = | $\mathbf{w}^T \odot \mathbf{u}^T \oplus . \otimes \mathbf{A}$                           |
| eWiseMult       | $\mathbf{C}\langle\mathbf{M}, r\rangle$       | = | $\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$  |
|                 | $\mathbf{w}\langle\mathbf{m}, r\rangle$       | = | $\mathbf{w} \odot \mathbf{u} \otimes \mathbf{v}$  |
| eWiseAdd        | $\mathbf{C}\langle\mathbf{M}, r\rangle$       | = | $\mathbf{C} \odot \mathbf{A} \oplus \mathbf{B}$   |
|                 | $\mathbf{w}\langle\mathbf{m}, r\rangle$       | = | $\mathbf{w} \odot \mathbf{u} \oplus \mathbf{v}$   |
| extract         | $\mathbf{C}\langle\mathbf{M}, r\rangle$       | = | $\mathbf{C} \odot \mathbf{A}(i, j)$   |
|                 | $\mathbf{w}\langle\mathbf{m}, r\rangle$       | = | $\mathbf{w} \odot \mathbf{u}(i)$  |
| assign          | $\mathbf{C}\langle\mathbf{M}, r\rangle(i, j)$ | = | $\mathbf{C}(i, j) \odot \mathbf{A}$   |
|                 | $\mathbf{w}\langle\mathbf{m}, r\rangle(i)$    | = | $\mathbf{w}(i) \odot \mathbf{u}$  |
| reduce (row)    | $\mathbf{w}\langle\mathbf{m}, r\rangle$       | = | $\mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$  |
| reduce (scalar) | $s$   | = | $s \odot [\oplus_{i,j} \mathbf{A}(i, j)]$   |
|                 | $s$   | = | $s \odot [\oplus_i \mathbf{u}(i)]$  |
| apply           | $\mathbf{C}\langle\mathbf{M}, r\rangle$       | = | $\mathbf{C} \odot f_u(\mathbf{A})$  |
|                 | $\mathbf{w}\langle\mathbf{m}, r\rangle$       | = | $\mathbf{w} \odot f_u(\mathbf{u})$  |
| apply(indexop)  | $\mathbf{C}\langle\mathbf{M}, r\rangle$       | = | $\mathbf{C} \odot f_i(\mathbf{A}, \text{ind}(\mathbf{A}), s)$                           |
|                 | $\mathbf{w}\langle\mathbf{m}, r\rangle$       | = | $\mathbf{w} \odot f_i(\mathbf{u}, \text{ind}(\mathbf{u}), s)$                           |
| select          | $\mathbf{C}\langle\mathbf{M}, r\rangle$       | = | $\mathbf{C} \odot \mathbf{A}\langle f_i(\mathbf{A}, \text{ind}(\mathbf{A}), s) \rangle$ |
|                 | $\mathbf{w}\langle\mathbf{m}, r\rangle$       | = | $\mathbf{w} \odot \mathbf{u}\langle f_i(\mathbf{u}, \text{ind}(\mathbf{u}), s) \rangle$ |
| transpose       | $\mathbf{C}\langle\mathbf{M}, r\rangle$       | = | $\mathbf{C} \odot \mathbf{A}^T$   |
| kronecker       | $\mathbf{C}\langle\mathbf{M}, r\rangle$       | = | $\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$  |

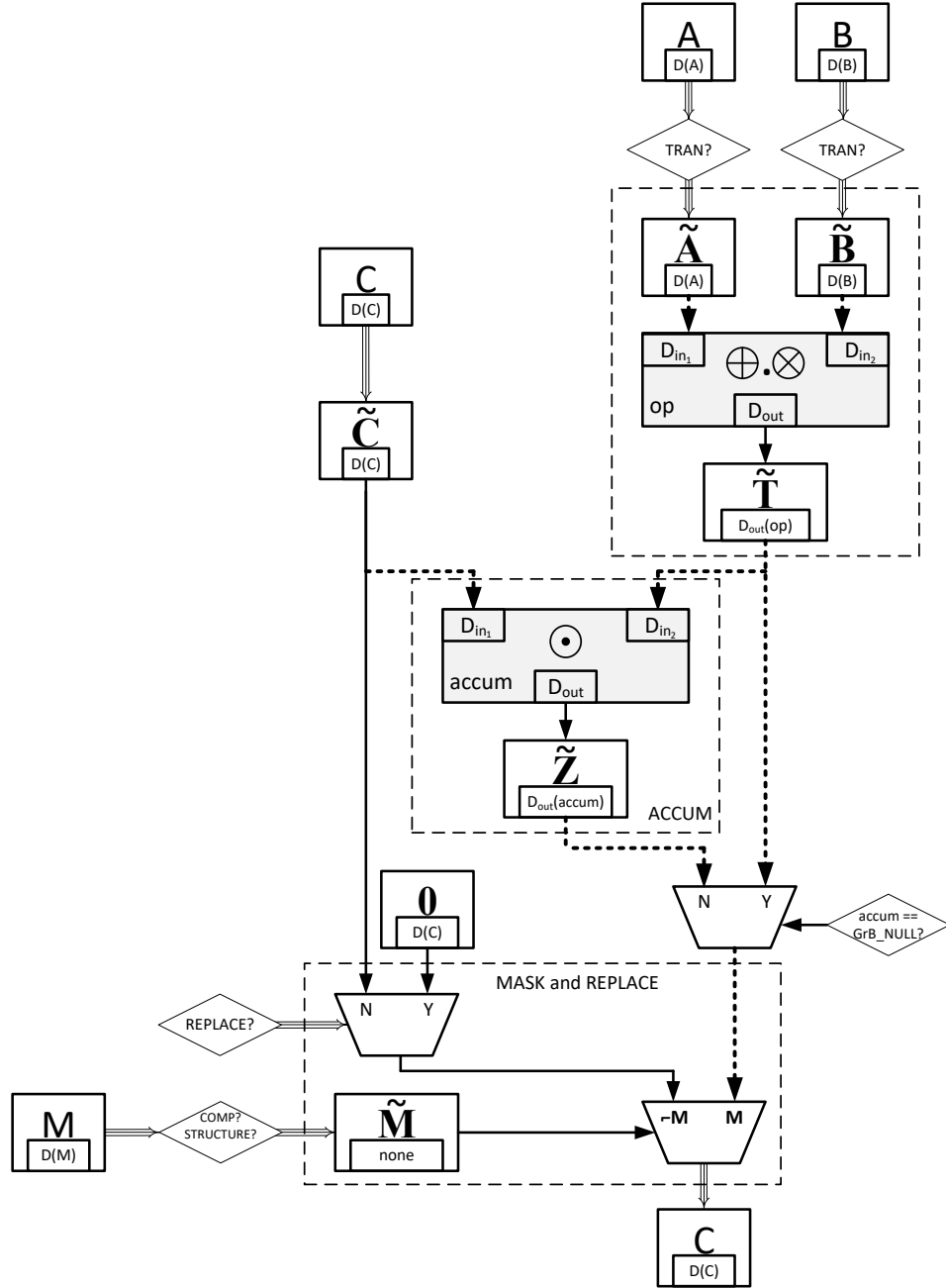


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows ( $\Rightarrow$ ) denote where “as if copy” takes place (including both collections and descriptor settings). The bold, dotted arrows indicate where casting may occur between different domains.



argument to a GraphBLAS operation and the associated descriptor indicates the transpose option, then the operation occurs as if on the transposed matrix. In this case, the relationships between the sizes in each dimension shift in the mathematically expected way.

## Masks: Structure-only, Complement, and Replace

When a GraphBLAS operation supports the use of an optional mask, that mask is specified through a GraphBLAS vector (for one-dimensional masks) or a GraphBLAS matrix (for two-dimensional masks). When a mask is used and the `GrB_STRUCTURE` descriptor value is not set, it is applied to the result from the operation wherever the stored values in the mask evaluate to true. If the `GrB_STRUCTURE` descriptor is set, the mask is applied to the result from the operation wherever the mask as a stored value (regardless of that value). Wherever the mask is applied, the result from the operation is either assigned to the provided output matrix/vector or, if a binary accumulation operation is provided, the result is accumulated into the corresponding elements of the provided output matrix/vector.

Given a GraphBLAS vector  $\mathbf{v} = \langle D, N, \{(i, v_i)\} \rangle$ , a one-dimensional mask is derived for use in the operation as follows:

$$\mathbf{m} = \begin{cases} \langle N, \{\mathbf{ind}(\mathbf{v})\} \rangle, & \text{if } \text{GrB\_STRUCTURE} \text{ is specified,} \\ \langle N, \{i : (\text{bool})v_i = \text{true}\} \rangle, & \text{otherwise} \end{cases}$$

where  $(\text{bool})v_i$  denotes casting the value  $v_i$  to a Boolean value (true or false). Likewise, given a GraphBLAS matrix  $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$ , a two-dimensional mask is derived for use in the operation as follows:

$$\mathbf{M} = \begin{cases} \langle M, N, \{\mathbf{ind}(\mathbf{A})\} \rangle, & \text{if } \text{GrB\_STRUCTURE} \text{ is specified,} \\ \langle M, N, \{(i, j) : (\text{bool})A_{ij} = \text{true}\} \rangle, & \text{otherwise} \end{cases}$$

where  $(\text{bool})A_{ij}$  denotes casting the value  $A_{ij}$  to a Boolean value. (true or false)

In both the one- and two-dimensional cases, the mask may also have a subsequent complement operation applied (*Section 3.5.4*) as specified in the descriptor, before a final mask is generated for use in the operation.

When the descriptor of an operation with a mask has specified that the `GrB_REPLACE` value is to be applied to the output (`GrB_OUTP`), then anywhere the mask is not true, the corresponding location in the output is cleared.

## Invalid and uninitialized objects

Upon entering a GraphBLAS operation, the first step is a check that all objects are valid and initialized. (Optional parameters can be set to `GrB_NULL`, which always counts as a valid object.) An invalid object is one that could not be computed due to a previous execution error. An uninitialized object is one that has not yet been created by a corresponding `new` or `dup` method. Appropriate error codes are returned if an object is not initialized (`GrB_UNINITIALIZED_OBJECT`) or invalid (`GrB_INVALID_OBJECT`).

2930 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-  
 2931 ommended to initialize all GraphBLAS objects to the predefined value `GrB_INVALID_HANDLE` at  
 2932 the point of their declaration, as shown in the following examples:

```
2933         GrB_Type          type = GrB_INVALID_HANDLE;
2934         GrB_Semiring      semiring = GrB_INVALID_HANDLE;
2935         GrB_Matrix        matrix = GrB_INVALID_HANDLE;
```

## 2936 Compliance

2937 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.  
 2938 That is, for each operation we give a recipe for producing its outcome. Any implementation that  
 2939 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error  
 2940 model (Section 2.6) is a conforming implementation.

### 2941 4.3.1 mxm: Matrix-matrix multiply

2942 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

## 2943 C Syntax

```
2944         GrB_Info GrB_mxm(GrB_Matrix          C,
2945                           const GrB_Matrix    Mask,
2946                           const GrB_BinaryOp   accum,
2947                           const GrB_Semiring   op,
2948                           const GrB_Matrix     A,
2949                           const GrB_Matrix     B,
2950                           const GrB_Descriptor desc);
```

## 2951 Parameters

2952 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 2953 that may be accumulated with the result of the matrix product. On output, the  
 2954 matrix holds the results of the operation.

2955 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 2956 stored into the output matrix C. The mask dimensions must match those of the  
 2957 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
 2958 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
 2959 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 2960 dimensions of C), `GrB_NULL` should be specified.

2961 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
 2962 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 2963 specified.

2964 **op** (IN) The semiring used in the matrix-matrix multiply.

2965 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 2966 multiplication.

2967 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 2968 multiplication.

2969 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 2970 should be specified. Non-default field/value pairs are listed as follows:  
 2971

| Param       | Field           | Value                | Description  |
|-------------|-----------------|----------------------|--|
| <b>C</b>    | <b>GrB_OUTP</b> | <b>GrB_REPLACE</b>   | Output matrix <b>C</b> is cleared (all elements removed) before the result is stored in it.  |
| <b>Mask</b> | <b>GrB_MASK</b> | <b>GrB_STRUCTURE</b> | The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined. |
| <b>Mask</b> | <b>GrB_MASK</b> | <b>GrB_COMP</b>      | Use the complement of <b>Mask</b> .  |
| <b>A</b>    | <b>GrB_INP0</b> | <b>GrB_TRAN</b>      | Use transpose of <b>A</b> for the operation.   |
| <b>B</b>    | <b>GrB_INP1</b> | <b>GrB_TRAN</b>      | Use transpose of <b>B</b> for the operation.   |

## 2973 Return Values

2974 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 2975 blocking mode, this indicates that the compatibility tests on di-  
 2976 mensions and domains for the input arguments passed successfully.  
 2977 Either way, output matrix **C** is ready to be used in the next method  
 2978 of the sequence.

2979 **GrB\_PANIC** Unknown internal error.

2980 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 2981 GraphBLAS objects (input or output) is in an invalid state caused  
 2982 by a previous execution error. Call **GrB\_error()** to access any error  
 2983 messages generated by the implementation.

2984 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

2985 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 2986 a call to **new** (or **Matrix\_dup** for matrix parameters).

2987 **GrB\_DIMENSION\_MISMATCH** Mask and/or matrix dimensions are incompatible.

2988 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 2989 corresponding domains of the semiring or accumulation operator,  
 2990 or the mask's domain is not compatible with `bool` (in the case where  
 2991 `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 2992 Description

2993 GrB\_mxm computes the matrix product  $C = A \oplus . \otimes B$  or, if an optional binary accumulation operator  
 2994  $(\odot)$  is provided,  $C = C \odot (A \oplus . \otimes B)$  (where matrices  $A$  and  $B$  can be optionally transposed).  
 2995 Logically, this operation occurs in three steps:

2996 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 2997 and dimensions are tested for compatibility.

2998 **Compute** The indicated computations are carried out.

2999 **Output** The result is written into the output matrix, possibly under control of a mask.

3000 Up to four argument matrices are used in the GrB\_mxm operation:

- 3001 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3002 2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)
- 3003 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3004 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3005 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for  
 3006 domain compatibility as follows:

- 3007 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3008 must be from one of the pre-defined types of Table 3.2.
- 3009 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3010 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3011 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3012 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3013 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 3014 of the accumulation operator.

3015 Two domains are compatible with each other if values from one domain can be cast to values in  
 3016 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3017 all compatible with each other. A domain from a user-defined type is only compatible with itself.

3018 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch  
 3019 error listed above is returned.

3020 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3021 ( $\leftarrow$  denotes copy):

- 3022 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3023 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3024 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3025  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3026 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3027 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3028  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3029 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3030  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 3031 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3032 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3033 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3034 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3035 must hold:

- 3036 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 3037 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 3038 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 3039 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .
- 3040 5.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .

3041 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the dimension mismatch  
 3042 error listed above is returned.

3043 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3044 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3045 We are now ready to carry out the matrix multiplication and any additional associated operations.  
 3046 We describe this in terms of two intermediate matrices:

- 3047 •  $\tilde{\mathbf{T}}$ : The matrix holding the product of matrices  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3048 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3049 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j)) \neq \emptyset\} \rangle$  is created. The value of each of its elements is computed by

$$3051 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{B}}(:, j))} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{B}}(k, j)),$$

3052 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring  $\mathbf{op}$ , respectively.

3053 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3054 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3055 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3056 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3057 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
3058 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 3059 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3060 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3061 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3062 \end{aligned}$$

3063 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3065 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
3066 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
3067 mask which acts as a “write mask”.

- 3068 • If  $\mathbf{desc}[\mathbf{GrB\_OUTP}].\mathbf{GrB\_REPLACE}$  is set, then any values in  $\mathbf{C}$  on input to this operation are  
3069 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3070 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3071 • If  $\mathbf{desc}[\mathbf{GrB\_OUTP}].\mathbf{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
3072 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
3073 mask are unchanged:

$$3074 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3075 In  $\mathbf{GrB\_BLOCKING}$  mode, the method exits with return value  $\mathbf{GrB\_SUCCESS}$  and the new content  
3076 of matrix  $\mathbf{C}$  is as defined above and fully computed. In  $\mathbf{GrB\_NONBLOCKING}$  mode, the method  
3077 exits with return value  $\mathbf{GrB\_SUCCESS}$  and the new content of matrix  $\mathbf{C}$  is as defined above but  
3078 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
3079 sequence.

### 3080 4.3.2 vxm: Vector-matrix multiply

3081 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

### 3082 C Syntax

```
3083         GrB_Info GrB_vxm(GrB_Vector          w,  
3084                           const GrB_Vector    mask,  
3085                           const GrB_BinaryOp    accum,  
3086                           const GrB_Semiring    op,  
3087                           const GrB_Vector    u,  
3088                           const GrB_Matrix     A,  
3089                           const GrB_Descriptor  desc);
```

### 3090 Parameters

3091 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3092 that may be accumulated with the result of the vector-matrix product. On output,  
3093 this vector holds the results of the operation.

3094 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3095 stored into the output vector **w**. The mask dimensions must match those of the  
3096 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3097 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3098 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3099 dimensions of **w**), **GrB\_NULL** should be specified.

3100 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3101 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3102 specified.

3103 **op** (IN) Semiring used in the vector-matrix multiply.

3104 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3105 multiplication.

3106 **A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
3107 multiplication.

3108 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
3109 should be specified. Non-default field/value pairs are listed as follows:  
3110

| Param | Field    | Value         | Description   |
|-------|----------|---------------|---|
| w     | GrB_OUTP | GrB_REPLACE   | Output vector w is cleared (all elements removed) before the result is stored in it.  |
| mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined. |
| mask  | GrB_MASK | GrB_COMP      | Use the complement of mask.   |
| A     | GrB_INP1 | GrB_TRAN      | Use transpose of A for the operation.   |

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

**GrB\_PANIC** Unknown internal error.

**GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call `GrB_error()` to access any error messages generated by the implementation.

**GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

**GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to `new` (or `dup` for matrix or vector parameters).

**GrB\_DIMENSION\_MISMATCH** Mask, vector, and/or matrix dimensions are incompatible.

**GrB\_DOMAIN\_MISMATCH** The domains of the various vectors/matrices are incompatible with the corresponding domains of the semiring or accumulation operator, or the mask's domain is not compatible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## Description

**GrB\_vxm** computes the vector-matrix product  $w^T = u^T \oplus . \otimes A$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w^T = w^T \odot (u^T \oplus . \otimes A)$  (where matrix A can be optionally transposed). Logically, this operation occurs in three steps:

**Setup** The internal vectors, matrices and mask used in the computation are formed and their domains/dimensions are tested for compatibility.

**Compute** The indicated computations are carried out.



3138 **Output** The result is written into the output vector, possibly under control of a mask.

3139 Up to four argument vectors or matrices are used in the `GrB_vxm` operation:

- 3140 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3141 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3142 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3143 4.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3144 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
 3145 tested for domain compatibility as follows:

- 3146 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3147 must be from one of the pre-defined types of Table 3.2.
- 3148 2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the semiring.
- 3149 3.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the semiring.
- 3150 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring.
- 3151 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3152 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$   
 3153 of the accumulation operator.

3154 Two domains are compatible with each other if values from one domain can be cast to values in  
 3155 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3156 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
 3157 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch  
 3158 error listed above is returned.

3159 From the argument vectors and matrices, the internal matrices and mask used in the computation  
 3160 are formed ( $\leftarrow$  denotes copy):

- 3161 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3162 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3163 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3164 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3165 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 3166 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 3167 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3168 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3169 4. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

3170 The internal matrices and masks are checked for shape compatibility. The following conditions  
3171 must hold:

3172 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$ .

3173 2.  $\text{size}(\tilde{\mathbf{w}}) = \text{ncols}(\tilde{\mathbf{A}})$ .

3174 3.  $\text{size}(\tilde{\mathbf{u}}) = \text{nrows}(\tilde{\mathbf{A}})$ .

3175 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch  
3176 error listed above is returned.

3177 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3178 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3179 We are now ready to carry out the vector-matrix multiplication and any additional associated  
3180 operations. We describe this in terms of two intermediate vectors:

- 3181 •  $\tilde{\mathbf{t}}$ : The vector holding the product of vector  $\tilde{\mathbf{u}}^T$  and matrix  $\tilde{\mathbf{A}}$ .
- 3182 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3183 The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{ncols}(\tilde{\mathbf{A}}), \{(j, t_j) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j)) \neq \emptyset\} \rangle$  is created.  
3184 The value of each of its elements is computed by

$$3185 \quad t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3186 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring `op`, respectively.

3187 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3188 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3189 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3190 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3191 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3192 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3193 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 3194 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3195 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3196 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 3197 \end{aligned}$$

3198 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3199 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3200 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3201 mask which acts as a “write mask”.

- 3202 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3203 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3204 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3205 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3206 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3207 mask are unchanged:

$$3208 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3209 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3210 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3211 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3212 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3213 sequence.

### 3214 4.3.3 mxv: Matrix-vector multiply

3215 Multiplies a matrix by a vector on a semiring. The result is a vector.

## 3216 C Syntax

```
3217      GrB_Info GrB_mxv(GrB_Vector      w,
3218                      const GrB_Vector mask,
3219                      const GrB_BinaryOp accum,
3220                      const GrB_Semiring op,
3221                      const GrB_Matrix A,
3222                      const GrB_Vector u,
3223                      const GrB_Descriptor desc);
```

## 3224 Parameters

3225 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
 3226 that may be accumulated with the result of the matrix-vector product. On output,  
 3227 this vector holds the results of the operation.

3228 **mask** (IN) An optional “write” mask that controls which results from this operation are  
 3229 stored into the output vector  $\mathbf{w}$ . The mask dimensions must match those of the  
 3230 vector  $\mathbf{w}$ . If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain

3231 of the `mask` vector must be of type `bool` or any of the predefined “built-in” types  
 3232 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
 3233 dimensions of `w`), `GrB_NULL` should be specified.

3234 `accum` (IN) An optional binary operator used for accumulating entries into existing `w`  
 3235 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
 3236 specified.

3237 `op` (IN) Semiring used in the vector-matrix multiply.

3238 `A` (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3239 multiplication.

3240 `u` (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3241 multiplication.

3242 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
 3243 should be specified. Non-default field/value pairs are listed as follows:  
 3244

| Param             | Field                 | Value                      | Description  |
|-------------------|-----------------------|----------------------------|--|
| <code>w</code>    | <code>GrB_OUTP</code> | <code>GrB_REPLACE</code>   | Output vector <code>w</code> is cleared (all elements removed) before the result is stored in it.  |
| <code>mask</code> | <code>GrB_MASK</code> | <code>GrB_STRUCTURE</code> | The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined. |
| <code>mask</code> | <code>GrB_MASK</code> | <code>GrB_COMP</code>      | Use the complement of <code>mask</code> .  |
| <code>A</code>    | <code>GrB_INP0</code> | <code>GrB_TRAN</code>      | Use transpose of <code>A</code> for the operation.   |

## 3246 Return Values

3247 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
 3248 blocking mode, this indicates that the compatibility tests on di-  
 3249 mensions and domains for the input arguments passed successfully.  
 3250 Either way, output vector `w` is ready to be used in the next method  
 3251 of the sequence.

3252 `GrB_PANIC` Unknown internal error.

3253 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
 3254 GraphBLAS objects (input or output) is in an invalid state caused  
 3255 by a previous execution error. Call `GrB_error()` to access any error  
 3256 messages generated by the implementation.

3257 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

3258 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by  
 3259 a call to `new` (or `dup` for matrix or vector parameters).

3260 GrB\_DIMENSION\_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3261 GrB\_DOMAIN\_MISMATCH The domains of the various vectors/matrices are incompatible with  
3262 the corresponding domains of the semiring or accumulation opera-  
3263 tor, or the mask's domain is not compatible with **bool** (in the case  
3264 where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 3265 Description

3266 GrB\_mvx computes the matrix-vector product  $w = A \oplus . \otimes u$ , or, if an optional binary accumulation  
3267 operator ( $\odot$ ) is provided,  $w = w \odot (A \oplus . \otimes u)$  (where matrix  $A$  can be optionally transposed).  
3268 Logically, this operation occurs in three steps:

3269 **Setup** The internal vectors, matrices and mask used in the computation are formed and their  
3270 domains/dimensions are tested for compatibility.

3271 **Compute** The indicated computations are carried out.

3272 **Output** The result is written into the output vector, possibly under control of a mask.

3273 Up to four argument vectors or matrices are used in the GrB\_mvx operation:

- 3274 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3275 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3276 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3277 4.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

3278 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
3279 tested for domain compatibility as follows:

- 3280 1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
3281 must be from one of the pre-defined types of Table 3.2.
- 3282 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3283 3.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3284 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3285 5. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3286 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
3287 of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of `GrB_m xv` ends and the domain mismatch error listed above is returned.

From the argument vectors and matrices, the internal matrices and mask used in the computation are formed ( $\leftarrow$  denotes copy):

1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - (b) If `mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
4. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

The internal matrices and masks are checked for shape compatibility. The following conditions must hold:

1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$ .
2.  $\text{size}(\tilde{\mathbf{w}}) = \text{nrows}(\tilde{\mathbf{A}})$ .
3.  $\text{size}(\tilde{\mathbf{u}}) = \text{ncols}(\tilde{\mathbf{A}})$ .

If any compatibility rule above is violated, execution of `GrB_m xv` ends and the dimension mismatch error listed above is returned.

From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

We are now ready to carry out the matrix-vector multiplication and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$ : The vector holding the product of matrix  $\tilde{\mathbf{A}}$  and vector  $\tilde{\mathbf{u}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{nrows}(\tilde{\mathbf{A}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}}) \neq \emptyset\} \rangle$  is created. The value of each of its elements is computed by

$$t_i = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{A}}(i, :)) \cap \text{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3320 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring **op**, respectively.

3321 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3322 • If **accum** = **GrB\_NULL**, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3323 • If **accum** is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3324 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3325 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3326 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3327 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3328 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3329 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3330 \end{aligned}$$

3332 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3333 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector **w**,  
 3334 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3335 mask which acts as a “write mask”.

- 3336 • If **desc[GrB\_OUTP].GrB\_REPLACE** is set, then any values in **w** on input to this operation are  
 3337 deleted and the content of the new output vector, **w**, is defined as,

$$3338 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3339 • If **desc[GrB\_OUTP].GrB\_REPLACE** is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3340 copied into the result vector, **w**, and elements of **w** that fall outside the set indicated by the  
 3341 mask are unchanged:

$$3342 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3343 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 3344 of vector **w** is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 3345 exits with return value **GrB\_SUCCESS** and the new content of vector **w** is as defined above but  
 3346 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3347 sequence.

#### 3348 4.3.4 eWiseMult: Element-wise multiplication

3349 **Note:** The difference between **eWiseAdd** and **eWiseMult** is not about the element-wise operation  
 3350 but how the index sets are treated. **eWiseAdd** returns an object whose indices are the “union” of  
 3351 the indices of the inputs whereas **eWiseMult** returns an object whose indices are the “intersection”  
 3352 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3353 the set of values from the resulting index set.

#### 3354 4.3.4.1 eWiseMult: Vector variant

3355 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-  
3356 ducing a third vector as result.

#### 3357 C Syntax

```
3358     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3359                           const GrB_Vector mask,  
3360                           const GrB_BinaryOp accum,  
3361                           const GrB_Semiring op,  
3362                           const GrB_Vector u,  
3363                           const GrB_Vector v,  
3364                           const GrB_Descriptor desc);  
3365  
3366     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3367                           const GrB_Vector mask,  
3368                           const GrB_BinaryOp accum,  
3369                           const GrB_Monoid op,  
3370                           const GrB_Vector u,  
3371                           const GrB_Vector v,  
3372                           const GrB_Descriptor desc);  
3373  
3374     GrB_Info GrB_eWiseMult(GrB_Vector      w,  
3375                           const GrB_Vector mask,  
3376                           const GrB_BinaryOp accum,  
3377                           const GrB_BinaryOp op,  
3378                           const GrB_Vector u,  
3379                           const GrB_Vector v,  
3380                           const GrB_Descriptor desc);
```

#### 3381 Parameters

3382 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3383 that may be accumulated with the result of the element-wise operation. On output,  
3384 this vector holds the results of the operation.

3385 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3386 stored into the output vector **w**. The mask dimensions must match those of the  
3387 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3388 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3389 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3390 dimensions of **w**), **GrB\_NULL** should be specified.

3391 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**



entries. If assignment rather than accumulation is desired, `GrB_NULL` should be specified.

**op** (IN) The semiring, monoid, or binary operator used in the element-wise “product” operation. Depending on which type is passed, the following defines the binary operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

Monoid:  $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$ ; the identity element is ignored.

Semiring:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$ ; the additive monoid is ignored.

**u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the operation.

**v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the operation.

**desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL` should be specified. Non-default field/value pairs are listed as follows:

| Param       | Field                 | Value                      | Description  |
|-------------|-----------------------|----------------------------|--|
| <b>w</b>    | <code>GrB_OUTP</code> | <code>GrB_REPLACE</code>   | Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.  |
| <b>mask</b> | <code>GrB_MASK</code> | <code>GrB_STRUCTURE</code> | The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined. |
| <b>mask</b> | <code>GrB_MASK</code> | <code>GrB_COMP</code>      | Use the complement of <b>mask</b> .  |

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector **w** is ready to be used in the next method of the sequence.

**GrB\_PANIC** Unknown internal error.

**GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call `GrB_error()` to access any error messages generated by the implementation.

**GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3422 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3423 a call to `new` (or `dup` for vector parameters).

3424 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3425 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
 3426 responding domains of the binary operator (`op`) or accumulation  
 3427 operator, or the mask's domain is not compatible with `bool` (in the  
 3428 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3429 Description

3430 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS vectors:  
 3431  $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$ .  
 3432 Logically, this operation occurs in three steps:

3433 **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3434 and dimensions are tested for compatibility.

3435 **Compute** The indicated computations are carried out.

3436 **Output** The result is written into the output vector, possibly under control of a mask.

3437 Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3438 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3439 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3440 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3441 4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3442 The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)  
 3443 are tested for domain compatibility as follows:

- 3444 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3445 must be from one of the pre-defined types of Table 3.2.
- 3446 2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3447 3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3448 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3449 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3450 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
 3451 the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch error listed above is returned.

From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$  denotes copy):

1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - (b) If `mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
4. Vector  $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

The internal vectors and mask are checked for dimension compatibility. The following conditions must hold:

1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$ .

If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension mismatch error listed above is returned.

From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

We are now ready to carry out the element-wise “product” and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$ : The vector holding the element-wise “product” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The value of each of its elements is computed by:

$$t_i = (\tilde{\mathbf{u}}(i) \otimes \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}}))$$

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

3483 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

3484 • If  $\text{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

3485 
$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3486 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3487 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

3488 
$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

3489

3490 
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3491

3492 
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

3493 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3494 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3495 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3496 mask which acts as a “write mask”.

3497 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3498 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

3499 
$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3500 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3501 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3502 mask are unchanged:

3503 
$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3504 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 3505 of vector  $\mathbf{w}$  is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 3506 exits with return value **GrB\_SUCCESS** and the new content of vector  $\mathbf{w}$  is as defined above but  
 3507 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3508 sequence.

#### 3509 4.3.4.2 eWiseMult: Matrix variant

3510 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-  
 3511 ducing a third matrix as result.

## 3512 C Syntax

```

3513     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3514                           const GrB_Matrix Mask,
3515                           const GrB_BinaryOp accum,
3516                           const GrB_Semiring op,
3517                           const GrB_Matrix A,
3518                           const GrB_Matrix B,
3519                           const GrB_Descriptor desc);
3520
3521     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3522                           const GrB_Matrix Mask,
3523                           const GrB_BinaryOp accum,
3524                           const GrB_Monoid op,
3525                           const GrB_Matrix A,
3526                           const GrB_Matrix B,
3527                           const GrB_Descriptor desc);
3528
3529     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3530                           const GrB_Matrix Mask,
3531                           const GrB_BinaryOp accum,
3532                           const GrB_BinaryOp op,
3533                           const GrB_Matrix A,
3534                           const GrB_Matrix B,
3535                           const GrB_Descriptor desc);

```

## 3536 Parameters

3537 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3538 that may be accumulated with the result of the element-wise operation. On output,  
3539 the matrix holds the results of the operation.

3540 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3541 stored into the output matrix C. The mask dimensions must match those of the  
3542 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3543 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types  
3544 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
3545 dimensions of C), `GrB_NULL` should be specified.

3546 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3547 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3548 specified.

3549 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
3550 operation. Depending on which type is passed, the following defines the binary  
3551 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:



3580 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3581 corresponding domains of the binary operator ( $\otimes$ ) or accumulation  
 3582 operator, or the mask's domain is not compatible with `bool` (in the  
 3583 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3584 Description

3585 This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:  
 3586  $C = A \otimes B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$ .  
 3587 Logically, this operation occurs in three steps:

3588 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3589 and dimensions are tested for compatibility.

3590 **Compute** The indicated computations are carried out.

3591 **Output** The result is written into the output matrix, possibly under control of a mask.

3592 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3593 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3594 2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)
- 3595 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3596 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3597 The argument matrices, the “product” operator ( $\otimes$ ), and the accumulation operator (if provided)  
 3598 are tested for domain compatibility as follows:

- 3599 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3600 must be from one of the pre-defined types of Table 3.2.
- 3601 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\otimes)$ .
- 3602 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\otimes)$ .
- 3603 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\otimes)$ .
- 3604 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3605 of the accumulation operator and  $\mathbf{D}_{out}(\otimes)$  of  $\otimes$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3606 the accumulation operator.

3607 Two domains are compatible with each other if values from one domain can be cast to values in  
 3608 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3609 compatible with each other. A domain from a user-defined type is only compatible with itself. If any

3610 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3611 error listed above is returned.

3612 From the argument matrices, the internal matrices and mask used in the computation are formed  
 3613 ( $\leftarrow$  denotes copy):

- 3614 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3615 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 3616 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 3617  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 3618 (b) If `Mask  $\neq$  GrB_NULL`,
    - 3619 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 3620  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 3621 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3622  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\text{bool})\mathbf{Mask}(i, j) = \text{true}\} \rangle$ .
  - 3623 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 3624 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 3625 4. Matrix  $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

3626 The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3627 must hold:

- 3628 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
- 3629 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

3630 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
 3631 mismatch error listed above is returned.

3632 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3633 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3634 We are now ready to carry out the element-wise “product” and any additional associated operations.  
 3635 We describe this in terms of two intermediate matrices:

- 3636 •  $\tilde{\mathbf{T}}$ : The matrix holding the element-wise product of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3637 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3638 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$   
 3639 is created. The value of each of its elements is computed by

$$3640 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3641 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:



3642 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

3643 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3644 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3645 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 3646 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$3647 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3648 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3649 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3652 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3653 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3654 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3655 mask which acts as a “write mask”.

3656 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3657 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3658 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3659 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3660 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3661 mask are unchanged:

$$3662 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3663 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 3664 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 3665 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 3666 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3667 sequence.

#### 3668 4.3.5 eWiseAdd: Element-wise addition

3669 **Note:** The difference between `eWiseAdd` and `eWiseMult` is not about the element-wise operation  
 3670 but how the index sets are treated. `eWiseAdd` returns an object whose indices are the “union” of  
 3671 the indices of the inputs whereas `eWiseMult` returns an object whose indices are the “intersection”  
 3672 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3673 the set of values from the resulting index set.

#### 3674 4.3.5.1 eWiseAdd: Vector variant

3675 Perform element-wise (general) addition on the elements of two vectors, producing a third vector  
3676 as result.

#### 3677 C Syntax

```
3678     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3679                          const GrB_Vector mask,  
3680                          const GrB_BinaryOp accum,  
3681                          const GrB_Semiring op,  
3682                          const GrB_Vector u,  
3683                          const GrB_Vector v,  
3684                          const GrB_Descriptor desc);  
3685  
3686     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3687                          const GrB_Vector mask,  
3688                          const GrB_BinaryOp accum,  
3689                          const GrB_Monoid op,  
3690                          const GrB_Vector u,  
3691                          const GrB_Vector v,  
3692                          const GrB_Descriptor desc);  
3693  
3694     GrB_Info GrB_eWiseAdd(GrB_Vector      w,  
3695                          const GrB_Vector mask,  
3696                          const GrB_BinaryOp accum,  
3697                          const GrB_BinaryOp op,  
3698                          const GrB_Vector u,  
3699                          const GrB_Vector v,  
3700                          const GrB_Descriptor desc);
```

#### 3701 Parameters

3702 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3703 that may be accumulated with the result of the element-wise operation. On output,  
3704 this vector holds the results of the operation.

3705 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3706 stored into the output vector **w**. The mask dimensions must match those of the  
3707 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3708 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3709 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3710 dimensions of **w**), **GrB\_NULL** should be specified.

3711 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**

entries. If assignment rather than accumulation is desired, `GrB_NULL` should be specified.

**op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum” operation. Depending on which type is passed, the following defines the binary operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

Monoid:  $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$ ; the identity element is ignored.

Semiring:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus(\text{op}) \rangle$ ; the multiplicative binary op and additive identity are ignored.

**u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the operation.

**v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the operation.

**desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL` should be specified. Non-default field/value pairs are listed as follows:

| Param       | Field                 | Value                      | Description  |
|-------------|-----------------------|----------------------------|--|
| <b>w</b>    | <code>GrB_OUTP</code> | <code>GrB_REPLACE</code>   | Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.  |
| <b>mask</b> | <code>GrB_MASK</code> | <code>GrB_STRUCTURE</code> | The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined. |
| <b>mask</b> | <code>GrB_MASK</code> | <code>GrB_COMP</code>      | Use the complement of <b>mask</b> .  |

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector **w** is ready to be used in the next method of the sequence.

**GrB\_PANIC** Unknown internal error.

**GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call `GrB_error()` to access any error messages generated by the implementation.

**GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3742 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
3743 a call to `new` (or `dup` for vector parameters).

3744 GrB\_DIMENSION\_MISMATCH Mask or vector dimensions are incompatible.

3745 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
3746 responding domains of the binary operator (`op`) or accumulation  
3747 operator, or the mask's domain is not compatible with `bool` (in the  
3748 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3749 Description

3750 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS vectors:  $w =$   
3751  $u \oplus v$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot (u \oplus v)$ . Logically,  
3752 this operation occurs in three steps:

3753 **Setup** The internal vectors and mask used in the computation are formed and their domains  
3754 and dimensions are tested for compatibility.

3755 **Compute** The indicated computations are carried out.

3756 **Output** The result is written into the output vector, possibly under control of a mask.

3757 Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3758 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 3759 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3760 3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$
- 3761 4.  $v = \langle \mathbf{D}(v), \mathbf{size}(v), \mathbf{L}(v) = \{(i, v_i)\} \rangle$

3762 The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are  
3763 tested for domain compatibility as follows:

- 3764 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
3765 must be from one of the pre-defined types of Table 3.2.
- 3766 2.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3767 3.  $\mathbf{D}(v)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3768 4.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3769 5.  $\mathbf{D}(u)$  and  $\mathbf{D}(v)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3770 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3771 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
3772 the accumulation operator.

3773 Two domains are compatible with each other if values from one domain can be cast to values in  
 3774 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3775 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 3776 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch  
 3777 error listed above is returned.

3778 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3779 denotes copy):

- 3780 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3781 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3782 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 3783 (b) If `mask  $\neq$  GrB_NULL`,
    - 3784 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 3785 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 3786 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3787 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 3788 4. Vector  $\tilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3789 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 3790 must hold:

- 3791 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}}) = \text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{v}})$ .

3792 If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension  
 3793 mismatch error listed above is returned.

3794 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3795 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3796 We are now ready to carry out the element-wise “sum” and any additional associated operations.  
 3797 We describe this in terms of two intermediate vectors:

- 3798 •  $\tilde{\mathbf{t}}$ : The vector holding the element-wise “sum” of  $\tilde{\mathbf{u}}$  and vector  $\tilde{\mathbf{v}}$ .
- 3799 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3800 The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\tilde{\mathbf{u}}) \cup \text{ind}(\tilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The  
 3801 value of each of its elements is computed by:

$$\begin{aligned}
 3802 \quad t_i &= (\tilde{\mathbf{u}}(i) \oplus \tilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})) \\
 3803 \\
 3804 \quad t_i &= \tilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\tilde{\mathbf{u}}) - (\text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{v}})))
 \end{aligned}$$

3805  
3806

$$t_i = \tilde{\mathbf{v}}(i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{v}}) - (\mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{v}})))$$

3807

where the difference operator in the previous expressions refers to set difference.

3808

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

3809

- If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

3810

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

3811

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3812

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

3813

3814

$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

3815

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3816

3817

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

3818

3819

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3820

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

3821

3822

3823

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

3824

3825

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3826

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

3827

3828

3829

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3830

In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

3831

3832

3833

3834

3835

#### 4.3.5.2 eWiseAdd: Matrix variant

3836

Perform element-wise (general) addition on the elements of two matrices, producing a third matrix as result.

3837

## 3838 C Syntax

```

3839         GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3840                               const GrB_Matrix Mask,
3841                               const GrB_BinaryOp accum,
3842                               const GrB_Semiring op,
3843                               const GrB_Matrix A,
3844                               const GrB_Matrix B,
3845                               const GrB_Descriptor desc);
3846
3847         GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3848                               const GrB_Matrix Mask,
3849                               const GrB_BinaryOp accum,
3850                               const GrB_Monoid op,
3851                               const GrB_Matrix A,
3852                               const GrB_Matrix B,
3853                               const GrB_Descriptor desc);
3854
3855         GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3856                               const GrB_Matrix Mask,
3857                               const GrB_BinaryOp accum,
3858                               const GrB_BinaryOp op,
3859                               const GrB_Matrix A,
3860                               const GrB_Matrix B,
3861                               const GrB_Descriptor desc);

```

## 3862 Parameters

3863 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3864 that may be accumulated with the result of the element-wise operation. On output,  
3865 the matrix holds the results of the operation.

3866 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3867 stored into the output matrix C. The mask dimensions must match those of the  
3868 matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
3869 of the `Mask` matrix must be of type `bool` or any of the predefined “built-in” types  
3870 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3871 dimensions of C), `GrB_NULL` should be specified.

3872 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3873 entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
3874 specified.

3875 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3876 operation. Depending on which type is passed, the following defines the binary  
3877 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3878 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
 3879 Monoid:  $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$ ; the identity element is ig-  
 3880 nored.  
 3881 Semiring:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus(\text{op}) \rangle$ ; the multiplicative bi-  
 3882 nary op and additive identity are ignored.

3883 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
 3884 operation.

3885 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
 3886 operation.

3887 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 3888 should be specified. Non-default field/value pairs are listed as follows:  
 3889

| Param | Field    | Value         | Description   |
|-------|----------|---------------|---|
| C     | GrB_OUTP | GrB_REPLACE   | Output matrix C is cleared (all elements removed) before the result is stored in it.  |
| Mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined. |
| Mask  | GrB_MASK | GrB_COMP      | Use the complement of Mask.   |
| A     | GrB_INP0 | GrB_TRAN      | Use transpose of A for the operation.   |
| B     | GrB_INP1 | GrB_TRAN      | Use transpose of B for the operation.   |

## 3891 Return Values

3892 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 3893 blocking mode, this indicates that the compatibility tests on di-  
 3894 mensions and domains for the input arguments passed successfully.  
 3895 Either way, output matrix C is ready to be used in the next method  
 3896 of the sequence.

3897 GrB\_PANIC Unknown internal error.

3898 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 3899 GraphBLAS objects (input or output) is in an invalid state caused  
 3900 by a previous execution error. Call GrB\_error() to access any error  
 3901 messages generated by the implementation.

3902 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3903 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 3904 a call to new (or Matrix\_dup for matrix parameters).

3905 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.



3906 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 3907 corresponding domains of the binary operator ( $\text{op}$ ) or accumulation  
 3908 operator, or the mask's domain is not compatible with `bool` (in the  
 3909 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3910 Description

3911 This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:  
 3912  $C = A \oplus B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \oplus B)$ .  
 3913 Logically, this operation occurs in three steps:

3914 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 3915 and dimensions are tested for compatibility.

3916 **Compute** The indicated computations are carried out.

3917 **Output** The result is written into the output matrix, possibly under control of a mask.

3918 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3919 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3920 2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)
- 3921 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3922 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3923 The argument matrices, the “sum” operator ( $\text{op}$ ), and the accumulation operator (if provided) are  
 3924 tested for domain compatibility as follows:

- 3925 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 3926 must be from one of the pre-defined types of Table 3.2.
- 3927 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3928 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3929 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3930 5.  $\mathbf{D}(A)$  and  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3931 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 3932 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of  $\text{op}$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
 3933 the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch error listed above is returned.

From the argument matrices, the internal matrices and mask used in the computation are formed ( $\leftarrow$  denotes copy):

1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - (b) If `Mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
4. Matrix  $\tilde{\mathbf{B}} \leftarrow \text{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

The internal matrices and masks are checked for dimension compatibility. The following conditions must hold:

1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}})$ .
2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}})$ .

If any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the dimension mismatch error listed above is returned.

From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

We are now ready to carry out the element-wise “sum” and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\tilde{\mathbf{T}}$ : The matrix holding the element-wise sum of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3965 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cup \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}$   
 3966 is created. The value of each of its elements is computed by

$$\begin{aligned} 3967 \quad T_{ij} &= (\tilde{\mathbf{A}}(i, j) \oplus \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \\ 3968 \quad T_{ij} &= \tilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{A}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \\ 3969 \quad T_{ij} &= \tilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{B}}) - (\mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}))) \end{aligned}$$

3972 where the difference operator in the previous expressions refers to set difference.

3973 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3974 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3975 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3976 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3977 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 3978 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 3979 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3980 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3981 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \end{aligned}$$

3984 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3985 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3986 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3987 mask which acts as a “write mask”.

- 3988 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3989 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3990 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3991 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3992 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3993 mask are unchanged:

$$3994 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3995 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 3996 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 3997 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 3998 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3999 sequence.

### 4000 4.3.6 extract: Selecting sub-graphs

4001 Extract a subset of a matrix or vector.

#### 4002 4.3.6.1 extract: Standard vector variant

4003 Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector  
4004 whose size is equal to the number of indices.

### 4005 C Syntax

```
4006         GrB_Info GrB_extract(GrB_Vector          w,  
4007                             const GrB_Vector    mask,  
4008                             const GrB_BinaryOp   accum,  
4009                             const GrB_Vector    u,  
4010                             const GrB_Index      *indices,  
4011                             GrB_Index           nindices,  
4012                             const GrB_Descriptor desc);
```

### 4013 Parameters

4014 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4015 that may be accumulated with the result of the extract operation. On output, this  
4016 vector holds the results of the operation.

4017 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4018 stored into the output vector **w**. The mask dimensions must match those of the  
4019 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4020 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4021 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4022 dimensions of **w**), **GrB\_NULL** should be specified.

4023 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4024 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4025 specified.

4026 **u** (IN) The GraphBLAS vector from which the subset is extracted.

4027 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of  
4028 elements from **u** that are extracted. If all elements of **u** are to be extracted in order  
4029 from 0 to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution  
4030 mode and return value, this array may be manipulated by the caller after this  
4031 operation returns without affecting any deferred computations for this operation.

4032 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4033 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4034 should be specified. Non-default field/value pairs are listed as follows:

4035

| Param | Field    | Value         | Description  |
|-------|----------|---------------|--|
| w     | GrB_OUTP | GrB_REPLACE   | Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.  |
| mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined. |
| mask  | GrB_MASK | GrB_COMP      | Use the complement of <b>mask</b> .  |

4036

## 4037 Return Values

4038 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 4039 blocking mode, this indicates that the compatibility tests on  
 4040 dimensions and domains for the input arguments passed suc-  
 4041 cessfully. Either way, output vector **w** is ready to be used in the  
 4042 next method of the sequence.

4043 GrB\_PANIC Unknown internal error.

4044 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 4045 opaque GraphBLAS objects (input or output) is in an invalid  
 4046 state caused by a previous execution error. Call GrB\_error() to  
 4047 access any error messages generated by the implementation.

4048 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

4049 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
 4050 by a call to **new** (or **dup** for vector parameters).

4051 GrB\_INDEX\_OUT\_OF\_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-  
 4052 blocking mode, this error can be deferred.

4053 GrB\_DIMENSION\_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices**  $\neq$  **size(w)**.

4054 GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with each  
 4055 other or the corresponding domains of the accumulation oper-  
 4056 ator, or the mask's domain is not compatible with **bool** (in the  
 4057 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

4058 GrB\_NULL\_POINTER Argument **row\_indices** is a NULL pointer.

## 4059 Description

4060 This variant of GrB\_extract computes the result of extracting a subset of locations from a Graph-  
 4061 BLAS vector in a specific order:  $w = u(\text{indices})$ ; or, if an optional binary accumulation operator

4062  $(\odot)$  is provided,  $w = w \odot u(\text{indices})$ . More explicitly:

$$4063 \quad \begin{aligned} w(i) &= u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ w(i) &= w(i) \odot u(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4064 Logically, this operation occurs in three steps:

4065     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4066             and dimensions are tested for compatibility.

4067     **Compute** The indicated computations are carried out.

4068     **Output** The result is written into the output vector, possibly under control of a mask.

4069 Up to three argument vectors are used in this GrB\_extract operation:

- 4070     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4071     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4072     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4073 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4074 bility as follows:

- 4075     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4076         must be from one of the pre-defined types of Table 3.2.
- 4077     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(u)$ .
- 4078     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4079         of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4080         mulation operator.

4081 Two domains are compatible with each other if values from one domain can be cast to values in  
4082 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4083 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4084 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
4085 error listed above is returned.

4086 From the arguments, the internal vectors, mask, and index array used in the computation are  
4087 formed ( $\leftarrow$  denotes copy):

- 4088     1. Vector  $\tilde{w} \leftarrow w$ .
- 4089     2. One-dimensional mask,  $\tilde{m}$ , is computed from argument `mask` as follows:  
4090         (a) If `mask = GrB_NULL`, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .

- 4091 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
 4092 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
 4093 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
 4094 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
 4095 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .  
 4096 4. The internal index array,  $\widetilde{\mathbf{I}}$ , is computed from argument indices as follows:  
 4097 (a) If  $\text{indices} = \text{GrB\_ALL}$ , then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .  
 4098 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4099 The internal vectors and mask are checked for dimension compatibility. The following conditions  
 4100 must hold:

- 4101 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$   
 4102 2.  $\text{nindices} = \text{size}(\widetilde{\mathbf{w}})$ .

4103 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4104 match error listed above is returned.

4105 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4106 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4107 We are now ready to carry out the extract and any additional associated operations. We describe  
 4108 this in terms of two intermediate vectors:

- 4109 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from  $\widetilde{\mathbf{u}}$  in their destination locations relative to  $\widetilde{\mathbf{w}}$ .
- 4110 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4111 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$4112 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{w}}), \{(i, \widetilde{\mathbf{u}}[\widetilde{\mathbf{I}}[i]]) \mid \forall i, 0 \leq i < \text{nindices} : \widetilde{\mathbf{I}}[i] \in \text{ind}(\widetilde{\mathbf{u}})\} \rangle.$$

4113 At this point, if any value in  $\widetilde{\mathbf{I}}$  is not in the valid range of indices for vector  $\widetilde{\mathbf{u}}$ , the execution of  
 4114 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`  
 4115 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
 4116 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4117 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 4118 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 4119 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$4120 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

#### 4.3.6.2 extract: Standard matrix variant

Extract a sub-matrix from a larger matrix as specified by a set of row indices and a set of column indices. The result is a matrix whose size is equal to size of the sets of indices.

### C Syntax

```
GrB_Info GrB_extract(GrB_Matrix      C,
                    const GrB_Matrix  Mask,
                    const GrB_BinaryOp accum,
                    const GrB_Matrix  A,
                    const GrB_Index   *row_indices,
                    GrB_Index          nrows,
                    const GrB_Index   *col_indices,
                    GrB_Index          ncols,
                    const GrB_Descriptor desc);
```



## Parameters

**C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the extract operation. On output, the matrix holds the results of the operation.

**Mask** (IN) An optional “write” mask that controls which results from this operation are stored into the output matrix **C**. The mask dimensions must match those of the matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the dimensions of **C**), **GrB\_NULL** should be specified.

**accum** (IN) An optional binary operator used for accumulating entries into existing **C** entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be specified.

**A** (IN) The GraphBLAS matrix from which the subset is extracted.

**row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **A** from which elements are extracted. If elements in all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

**nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(C)**.

**col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **A** from which elements are extracted. If elements in all columns of **A** are to be extracted in order, then **GrB\_ALL** should be specified. Regardless of execution mode and return value, this array may be manipulated by the caller after this operation returns without affecting any deferred computations for this operation.

**ncols** (IN) The number of values in the **col\_indices** array. Must be equal to **ncols(C)**.

**desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL** should be specified. Non-default field/value pairs are listed as follows:

| Param       | Field           | Value                | Description  |
|-------------|-----------------|----------------------|--|
| <b>C</b>    | <b>GrB_OUTP</b> | <b>GrB_REPLACE</b>   | Output matrix <b>C</b> is cleared (all elements removed) before the result is stored in it.  |
| <b>Mask</b> | <b>GrB_MASK</b> | <b>GrB_STRUCTURE</b> | The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined. |
| <b>Mask</b> | <b>GrB_MASK</b> | <b>GrB_COMP</b>      | Use the complement of <b>Mask</b> .  |
| <b>A</b>    | <b>GrB_INP0</b> | <b>GrB_TRAN</b>      | Use transpose of <b>A</b> for the operation.   |

## 4187 Return Values

|      |                                 |   |
|------|---------------------------------|---|
| 4188 | <b>GrB_SUCCESS</b>              | In blocking mode, the operation completed successfully. In non-                               |
| 4189 |                                 | blocking mode, this indicates that the compatibility tests on                                 |
| 4190 |                                 | dimensions and domains for the input arguments passed suc-                                    |
| 4191 |                                 | cessfully. Either way, output matrix C is ready to be used in the                             |
| 4192 |                                 | next method of the sequence.  |
| 4193 | <b>GrB_PANIC</b>                | Unknown internal error.   |
| 4194 | <b>GrB_INVALID_OBJECT</b>       | This is returned in any execution mode whenever one of the                                    |
| 4195 |                                 | opaque GraphBLAS objects (input or output) is in an invalid                                   |
| 4196 |                                 | state caused by a previous execution error. Call <code>GrB_error()</code> to                  |
| 4197 |                                 | access any error messages generated by the implementation.                                    |
| 4198 | <b>GrB_OUT_OF_MEMORY</b>        | Not enough memory available for the operation.  |
| 4199 | <b>GrB_UNINITIALIZED_OBJECT</b> | One or more of the GraphBLAS objects has not been initialized                                 |
| 4200 |                                 | by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).             |
| 4201 | <b>GrB_INDEX_OUT_OF_BOUNDS</b>  | A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or    |
| 4202 |                                 | a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In    |
| 4203 |                                 | non-blocking mode, this error can be deferred.  |
| 4204 | <b>GrB_DIMENSION_MISMATCH</b>   | Mask and C dimensions are incompatible, <code>nrows</code> $\neq$ <code>nrows(C)</code> , or  |
| 4205 |                                 | <code>ncols</code> $\neq$ <code>ncols(C)</code> .   |
| 4206 | <b>GrB_DOMAIN_MISMATCH</b>      | The domains of the various matrices are incompatible with each                                |
| 4207 |                                 | other or the corresponding domains of the accumulation oper-                                  |
| 4208 |                                 | ator, or the mask's domain is not compatible with <code>bool</code> (in the                   |
| 4209 |                                 | case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).                             |
| 4210 | <b>GrB_NULL_POINTER</b>         | Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code> |
| 4211 |                                 | is a NULL pointer, or both.   |

## 4212 Description

4213 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified  
 4214 rows and columns of a GraphBLAS matrix in a specific order:  $C = A(\text{row\_indices}, \text{col\_indices})$ ; or,  
 4215 if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A(\text{row\_indices}, \text{col\_indices})$ .  
 4216 More explicitly (not accounting for an optional transpose of A):

$$\begin{aligned}
 &C(i, j) = A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
 &C(i, j) = C(i, j) \odot A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

4218 Logically, this operation occurs in three steps:

4219 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 4220 and dimensions are tested for compatibility.

4221 **Compute** The indicated computations are carried out.

4222 **Output** The result is written into the output matrix, possibly under control of a mask.

4223 Up to three argument matrices are used in the `GrB_extract` operation:

- 4224 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4225 2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)
- 4226 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4227 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
4228 ibility as follows:

- 4229 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
4230 must be from one of the pre-defined types of Table 3.2.
- 4231 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4232 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4233 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4234 mulation operator.

4235 Two domains are compatible with each other if values from one domain can be cast to values in  
4236 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4237 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4238 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4239 error listed above is returned.

4240 From the arguments, the internal matrices, `mask`, and index arrays used in the computation are  
4241 formed ( $\leftarrow$  denotes copy):

- 4242 1. Matrix  $\tilde{C} \leftarrow C$ .
- 4243 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:
  - 4244 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
4245  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 4246 (b) If `Mask`  $\neq$  `GrB_NULL`,
    - 4247 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
4248  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 4249 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
4250  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 4251 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .
- 4252 3. Matrix  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$ .

- 4253 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4254 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .
- 4255 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \text{nrows}$ .
- 4256 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4257 (a) If `col_indices` = `GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .
- 4258 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

4259 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4260 must hold:

- 4261 1.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$ .
- 4262 2.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$ .
- 4263 3.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}$ .
- 4264 4.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}$ .

4265 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4266 match error listed above is returned.

4267 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4268 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4269 We are now ready to carry out the extract and any additional associated operations. We describe  
4270 this in terms of two intermediate matrices:

- 4271 •  $\tilde{\mathbf{T}}$ : The matrix holding the extraction from  $\tilde{\mathbf{A}}$ .
- 4272 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4273 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

4274 
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{ (i, j, \tilde{\mathbf{A}}(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j])) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \text{ind}(\tilde{\mathbf{A}}) \} \rangle.$$

4275 At this point, if any value in the  $\tilde{\mathbf{I}}$  array is not in the range  $[0, \text{nrows}(\tilde{\mathbf{A}}))$  or any value in the  $\tilde{\mathbf{J}}$   
4276 array is not in the range  $[0, \text{ncols}(\tilde{\mathbf{A}}))$ , the execution of `GrB_extract` ends and the index out-of-  
4277 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
4278 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
4279 this point forward in the sequence.

4280 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4281 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .

4282 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4283 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4284 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
4285 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4286 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4287 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4289 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4291 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4292 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
4293 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
4294 mask which acts as a “write mask”.

4295 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
4296 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$4297 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4298 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
4299 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
4300 mask are unchanged:

$$4301 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4302 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
4303 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
4304 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
4305 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
4306 sequence.

#### 4307 4.3.6.3 extract: Column (and row) variant

4308 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the  
4309 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as  
4310 well.

## 4311 C Syntax

```

4312         GrB_Info GrB_extract(GrB_Vector      w,
4313                               const GrB_Vector mask,
4314                               const GrB_BinaryOp accum,
4315                               const GrB_Matrix A,
4316                               const GrB_Index *row_indices,
4317                               GrB_Index      nrows,
4318                               GrB_Index      col_index,
4319                               const GrB_Descriptor desc);

```

## 4320 Parameters

4321        **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4322        that may be accumulated with the result of the extract operation. On output, this  
4323        vector holds the results of the operation.

4324        **mask** (IN) An optional “write” mask that controls which results from this operation are  
4325        stored into the output vector **w**. The mask dimensions must match those of the  
4326        vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4327        of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4328        in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4329        dimensions of **w**), **GrB\_NULL** should be specified.

4330        **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4331        entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4332        specified.

4333        **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4334        **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations  
4335        within the specified column of **A** from which elements are extracted. If elements in  
4336        all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless  
4337        of execution mode and return value, this array may be manipulated by the caller  
4338        after this operation returns without affecting any deferred computations for this  
4339        operation.

4340        **nrows** (IN) The number of indices in the **row\_indices** array. Must be equal to **size(w)**.

4341        **col\_index** (IN) The index of the column of **A** from which to extract values. It must be in the  
4342        range  $[0, \mathbf{ncols}(A))$ .

4343        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4344        should be specified. Non-default field/value pairs are listed as follows:

4345

| Param | Field    | Value         | Description   |
|-------|----------|---------------|---|
| w     | GrB_OUTP | GrB_REPLACE   | Output vector w is cleared (all elements removed) before the result is stored in it.  |
| mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined. |
| mask  | GrB_MASK | GrB_COMP      | Use the complement of mask.   |
| A     | GrB_INP0 | GrB_TRAN      | Use transpose of A for the operation.   |

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector **w** is ready to be used in the next method of the sequence.

**GrB\_PANIC** Unknown internal error.

**GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call **GrB\_error()** to access any error messages generated by the implementation.

**GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

**GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector or matrix parameters).

**GrB\_INVALID\_INDEX** **col\_index** is outside the allowable range (i.e., greater than **ncols(A)**).

**GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** is greater than or equal to **nrows(A)**. In non-blocking mode, this error can be deferred.

**GrB\_DIMENSION\_MISMATCH** **mask** and **w** dimensions are incompatible, or **nrows**  $\neq$  **size(w)**.

**GrB\_DOMAIN\_MISMATCH** The domains of the vector or matrix are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

**GrB\_NULL\_POINTER** Argument **row\_indices** is a NULL pointer.

## Description

This variant of **GrB\_extract** computes the result of extracting a subset of locations (in a specific order) from a specified column of a GraphBLAS matrix: **w** = **A(:, col\_index)(row\_indices)**; or, if

4373 an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot A(:, \text{col\_index})(\text{row\_indices})$ .  
 4374 More explicitly:

$$4375 \quad \begin{aligned} w(i) &= A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows}, \quad \text{or} \\ w(i) &= w(i) \odot A(\text{row\_indices}[i], \text{col\_index}) \quad \forall i : 0 \leq i < \text{nrows} \end{aligned}$$

4376 Logically, this operation occurs in three steps:

4377     **Setup** The internal matrices, vectors, and mask used in the computation are formed and their  
 4378 domains and dimensions are tested for compatibility.

4379     **Compute** The indicated computations are carried out.

4380     **Output** The result is written into the output vector, possibly under control of a mask.

4381 Up to three argument vectors and matrices are used in this GrB\_extract operation:

- 4382     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4383     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4384     3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4385 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain  
 4386 compatibility as follows:

- 4387     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
 4388 must be from one of the pre-defined types of Table 3.2.
- 4389     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}(A)$ .
- 4390     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4391 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4392 mulation operator.

4393 Two domains are compatible with each other if values from one domain can be cast to values in  
 4394 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4395 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4396 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
 4397 error listed above is returned.

4398 From the arguments, the internal vector, matrix, mask, and index array used in the computation  
 4399 are formed ( $\leftarrow$  denotes copy):

- 4400     1. Vector  $\tilde{w} \leftarrow w$ .
- 4401     2. One-dimensional mask,  $\tilde{m}$ , is computed from argument **mask** as follows:  
 4402         (a) If **mask** = GrB\_NULL, then  $\tilde{m} = \langle \text{size}(w), \{i, \forall i : 0 \leq i < \text{size}(w)\} \rangle$ .



4403 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
 4404 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
 4405 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
 4406 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
 4407 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .  
 4408 4. The internal row index array,  $\widetilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:  
 4409 (a) If `indices = GrB_ALL`, then  $\widetilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nrows}$ .  
 4410 (b) Otherwise,  $\widetilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nrows}$ .

4411 The internal vector, `mask`, and index array are checked for dimension compatibility. The following  
 4412 conditions must hold:

- 4413 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 4414 2.  $\text{size}(\widetilde{\mathbf{w}}) = \text{nrows}$ .

4415 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4416 match error listed above is returned.

4417 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4418 1.  $0 \leq \text{col\_index} < \text{ncols}(\mathbf{A})$

4419 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above  
 4420 is returned.

4421 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4422 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4423 We are now ready to carry out the extract and any additional associated operations. We describe  
 4424 this in terms of two intermediate vectors:

- 4425 •  $\widetilde{\mathbf{t}}$ : The vector holding the extraction from a column of  $\widetilde{\mathbf{A}}$ .
- 4426 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4427 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$4428 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \widetilde{\mathbf{A}}(\widetilde{\mathbf{I}}[i], \text{col\_index})) \mid \forall i, 0 \leq i < \text{nrows} : (\widetilde{\mathbf{I}}[i], \text{col\_index}) \in \text{ind}(\widetilde{\mathbf{A}})\} \rangle.$$

4429 At this point, if any value in  $\widetilde{\mathbf{I}}$  is not in the range  $[0, \text{nrows}(\widetilde{\mathbf{A}}))$ , the execution of `GrB_extract`  
 4430 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,  
 4431 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result  
 4432 vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4433 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

4434 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

4435 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4436 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4437 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 4438 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$4439 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})),$$

4440

$$4441 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4442

$$4443 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4444 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4445 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 4446 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 4447 mask which acts as a “write mask”.

4448 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
 4449 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4450 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4451 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 4452 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 4453 mask are unchanged:

$$4454 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4455 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 4456 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 4457 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
 4458 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 4459 sequence.

### 4460 4.3.7 assign: Modifying sub-graphs

4461 Assign the contents of a subset of a matrix or vector.

#### 4462 4.3.7.1 assign: Standard vector variant

4463 Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices.  
 4464 The size of the input vector is the same size as the index array provided.

## 4465 C Syntax

```
4466         GrB_Info GrB_assign(GrB_Vector      w,  
4467                             const GrB_Vector mask,  
4468                             const GrB_BinaryOp accum,  
4469                             const GrB_Vector u,  
4470                             const GrB_Index *indices,  
4471                             GrB_Index nindices,  
4472                             const GrB_Descriptor desc);
```

## 4473 Parameters

4474 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4475 that may be accumulated with the result of the assign operation. On output, this  
4476 vector holds the results of the operation.

4477 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4478 stored into the output vector **w**. The mask dimensions must match those of the  
4479 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4480 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4481 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4482 dimensions of **w**), **GrB\_NULL** should be specified.

4483 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4484 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4485 specified.

4486 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4487 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4488 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0  
4489 to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4490 and return value, this array may be manipulated by the caller after this operation  
4491 returns without affecting any deferred computations for this operation. If this  
4492 array contains duplicate values, it implies in assignment of more than one value to  
4493 the same location which leads to undefined results.

4494 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4495 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4496 should be specified. Non-default field/value pairs are listed as follows:  
4497

| Param | Field    | Value         | Description   |
|-------|----------|---------------|---|
| w     | GrB_OUTP | GrB_REPLACE   | Output vector w is cleared (all elements removed) before the result is stored in it.  |
| mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined. |
| mask  | GrB_MASK | GrB_COMP      | Use the complement of mask.   |

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

**GrB\_PANIC** Unknown internal error.

**GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call **GrB\_error()** to access any error messages generated by the implementation.

**GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

**GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector parameters).

**GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indices** is greater than or equal to **size(w)**. In non-blocking mode, this can be reported as an execution error.

**GrB\_DIMENSION\_MISMATCH** mask and w dimensions are incompatible, or **nindices**  $\neq$  **size(u)**.

**GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

**GrB\_NULL\_POINTER** Argument **indices** is a NULL pointer.

## Description

This variant of **GrB\_assign** computes the result of assigning elements from a source GraphBLAS vector to a destination GraphBLAS vector in a specific order:  $w(\text{indices}) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w(\text{indices}) = w(\text{indices}) \odot u$ . More explicitly:

$$\begin{aligned}
 w(\text{indices}[i]) &= u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
 w(\text{indices}[i]) &= w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
 \end{aligned}$$

4526 Logically, this operation occurs in three steps:

4527     **Setup** The internal vectors and mask used in the computation are formed and their domains  
4528             and dimensions are tested for compatibility.

4529     **Compute** The indicated computations are carried out.

4530     **Output** The result is written into the output vector, possibly under control of a mask.

4531 Up to three argument vectors are used in the `GrB_assign` operation:

- 4532     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4533     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4534     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4535 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4536 bility as follows:

- 4537     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
4538         must be from one of the pre-defined types of Table 3.2.
- 4539     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4540     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
4541         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
4542         mulation operator.

4543 Two domains are compatible with each other if values from one domain can be cast to values in  
4544 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4545 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4546 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
4547 error listed above is returned.

4548 From the arguments, the internal vectors, mask and index array used in the computation are formed  
4549 ( $\leftarrow$  denotes copy):

- 4550     1. Vector  $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 4551     2. One-dimensional mask,  $\widetilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 4552         (a) If `mask` = `GrB_NULL`, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 4553         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 4554             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 4555             ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 4556         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .

4557 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4558 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument indices as follows:

4559 (a) If `indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i$ ,  $\forall i : 0 \leq i < \text{nindices}$ .

4560 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{indices}[i]$ ,  $\forall i : 0 \leq i < \text{nindices}$ .

4561 The internal vector and mask are checked for dimension compatibility. The following conditions  
4562 must hold:

4563 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4564 2.  $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$ .

4565 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4566 match error listed above is returned.

4567 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4568 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4569 We are now ready to carry out the assign and any additional associated operations. We describe  
4570 this in terms of two intermediate vectors:

4571 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{w}}$ .

4572 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4573 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

4574 
$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nindices} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4575 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{w}}$ , computation  
4576 ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING`  
4577 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4578 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4579 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

4580 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

4581 
$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4582 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
4583 of  $\tilde{\mathbf{w}}$  ( $\text{ind}(\tilde{\mathbf{w}})$ ) and remove from it all the indices of  $\tilde{\mathbf{w}}$  that are in the set of indices being  
4584 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4585 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4586 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

4587 
$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{w}}))),$$

4588 
$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4589 where the difference operator refers to set difference.

4591 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4592 \quad \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4593 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 4594 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 4595 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 4596 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4597 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4598 \quad & \\ 4599 \end{aligned}$$

4600 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4601 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 4602 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 4603 mask which acts as a “write mask”.

4604 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
 4605 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4606 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4607 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 4608 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 4609 mask are unchanged:

$$4610 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

4611 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 4612 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 4613 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
 4614 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 4615 sequence.

#### 4616 4.3.7.2 assign: Standard matrix variant

4617 Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices.  
 4618 The dimensions of the input matrix are the same size as the row and column index arrays provided.

### 4619 C Syntax

```
4620      GrB_Info GrB_assign(GrB_Matrix      C,
4621                          const GrB_Matrix Mask,
4622                          const GrB_BinaryOp accum,
4623                          const GrB_Matrix A,
```

```

4624         const GrB_Index      *row_indices,
4625         GrB_Index             nrows,
4626         const GrB_Index      *col_indices,
4627         GrB_Index             ncols,
4628         const GrB_Descriptor desc);

```

## 4629 Parameters

4630     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
4631     that may be accumulated with the result of the assign operation. On output, the  
4632     matrix holds the results of the operation.

4633     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
4634     stored into the output matrix **C**. The mask dimensions must match those of the  
4635     matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4636     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
4637     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4638     dimensions of **C**), **GrB\_NULL** should be specified.

4639     **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4640     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4641     specified.

4642     **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4643     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
4644     that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** – 1,  
4645     then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
4646     this array may be manipulated by the caller after this operation returns without  
4647     affecting any deferred computations for this operation. If this array contains du-  
4648     plicate values, it implies assignment of more than one value to the same location  
4649     which leads to undefined results.

4650     **nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(A)**  
4651     if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4652     **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns  
4653     of **C** that are assigned. If all columns of **C** are to be assigned in order from 0  
4654     to **ncols** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4655     and return value, this array may be manipulated by the caller after this operation  
4656     returns without affecting any deferred computations for this operation. If this  
4657     array contains duplicate values, it implies assignment of more than one value to  
4658     the same location which leads to undefined results.

4659     **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **ncols(A)** if **A** is  
4660     not transposed, or equal to **nrows(A)** if **A** is transposed.



4661  
4662  
4663

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL should be specified. Non-default field/value pairs are listed as follows:

4664

| Param | Field    | Value         | Description   |
|-------|----------|---------------|---|
| C     | GrB_OUTP | GrB_REPLACE   | Output matrix C is cleared (all elements removed) before the result is stored in it.  |
| Mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined. |
| Mask  | GrB_MASK | GrB_COMP      | Use the complement of Mask.   |
| A     | GrB_INP0 | GrB_TRAN      | Use transpose of A for the operation.   |

## 4665 Return Values

4666  
4667  
4668  
4669  
4670

GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

4671

GrB\_PANIC Unknown internal error.

4672  
4673  
4674  
4675

GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB\_error() to access any error messages generated by the implementation.

4676

GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

4677  
4678

GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix\_dup for matrix parameters).

4679  
4680  
4681

GrB\_INDEX\_OUT\_OF\_BOUNDS A value in row\_indices is greater than or equal to nrows(C), or a value in col\_indices is greater than or equal to ncols(C). In non-blocking mode, this can be reported as an execution error.

4682  
4683

GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, nrow  $\neq$  nrow(A), or ncol  $\neq$  ncol(A).

4684  
4685  
4686  
4687

GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

4688  
4689

GrB\_NULL\_POINTER Either argument row\_indices is a NULL pointer, argument col\_indices is a NULL pointer, or both.

## 4690 Description

4691 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows  
 4692 and columns in `C` in a specified order:  $C(\text{row\_indices}, \text{col\_indices}) = A$ ; or, if an optional binary  
 4693 accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot$   
 4694 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned} C(\text{row\_indices}[i], \text{col\_indices}[j]) &= A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ 4695 \quad C(\text{row\_indices}[i], \text{col\_indices}[j]) &= C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot A(i, j), \\ &\quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4696 Logically, this operation occurs in three steps:

4697     Setup The internal matrices and mask used in the computation are formed and their domains  
 4698     and dimensions are tested for compatibility.

4699     Compute The indicated computations are carried out.

4700     Output The result is written into the output matrix, possibly under control of a mask.

4701 Up to three argument matrices are used in the `GrB_assign` operation:

- 4702     1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4703     2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \text{nrows}(\text{Mask}), \text{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)
- 4704     3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4705 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
 4706 ibility as follows:

- 4707     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 4708     must be from one of the pre-defined types of Table 3.2.
- 4709     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 4710     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4711     of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4712     mulation operator.

4713 Two domains are compatible with each other if values from one domain can be cast to values in  
 4714 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4715 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4716 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 4717 error listed above is returned.

4718 From the arguments, the internal matrices, mask, and index arrays used in the computation are  
 4719 formed ( $\leftarrow$  denotes copy):

- 4720 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 4721 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
- 4722 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
4723  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 4724 (b) If `Mask  $\neq$  GrB_NULL`,
- 4725 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
4726  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
- 4727 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
4728  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
- 4729 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4730 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4731 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4732 (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4733 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4734 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4735 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
- 4736 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

4737 The internal matrices and mask are checked for dimension compatibility. The following conditions  
4738 must hold:

- 4739 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 4740 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 4741 3.  $\mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}$ .
- 4742 4.  $\mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}$ .

4743 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4744 match error listed above is returned.

4745 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4746 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4747 We are now ready to carry out the assign and any additional associated operations. We describe  
4748 this in terms of two intermediate vectors:

- 4749 •  $\tilde{\mathbf{T}}$ : The matrix holding the contents from  $\tilde{\mathbf{A}}$  in their destination locations relative to  $\tilde{\mathbf{C}}$ .
- 4750 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4751 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$4752 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4753 At this point, if any value in the  $\tilde{\mathbf{I}}$  array is not in the range  $[0, \mathbf{nrows}(\tilde{\mathbf{C}}))$  or any value in the  
 4754  $\tilde{\mathbf{J}}$  array is not in the range  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-  
 4755 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
 4756 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
 4757 this point forward in the sequence.

4758 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 4759 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$4760 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 4761 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4762 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 4763 of  $\tilde{\mathbf{C}}$  ( $\mathbf{ind}(\tilde{\mathbf{C}})$ ) and remove from it all the indices of  $\tilde{\mathbf{C}}$  that are in the set of indices being  
 4764 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

4765 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4766 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4767 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4768 \\ 4769 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}),$$

4770 where the difference operator refers to set difference.

- 4771 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4772 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4773 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 4774 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4775 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4776 \\ 4777 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4778 \\ 4779 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4780 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4781 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 4782 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 4783 mask which acts as a “write mask”.

- If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in **C** on input to this operation are deleted and the content of the new output matrix, **C**, is defined as,

$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are copied into the result matrix, **C**, and elements of **C** that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of matrix **C** is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of matrix **C** is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

### 4.3.7.3 assign: Column variant

Assign the contents a vector to a subset of elements in one column of a matrix. Note that since the output cannot be transposed, a different variant of **assign** is provided to assign to a row of a matrix.

## C Syntax

```
GrB_Info GrB_assign(GrB_Matrix      C,
                    const GrB_Vector mask,
                    const GrB_BinaryOp accum,
                    const GrB_Vector u,
                    const GrB_Index *row_indices,
                    GrB_Index        nrows,
                    GrB_Index        col_index,
                    const GrB_Descriptor desc);
```

## Parameters

**C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the assign operation. On output, this matrix holds the results of the operation.

**mask** (IN) An optional “write” mask that controls which results from this operation are stored into the specified column of the output matrix **C**. The mask dimensions must match those of a single column of the matrix **C**. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain of the **Mask** matrix must be of type

4817 bool or any of the predefined “built-in” types in Table 3.2. If the default mask  
 4818 is desired (i.e., a mask that is all true with the dimensions of a column of C),  
 4819 GrB\_NULL should be specified.

4820 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
 4821 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
 4822 specified.

4823 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column  
 4824 of C.

4825 **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
 4826 the specified column of C that are to be assigned. If all elements of the column  
 4827 in C are to be assigned in order from index 0 to **nrows** – 1, then GrB\_ALL should  
 4828 be specified. Regardless of execution mode and return value, this array may be  
 4829 manipulated by the caller after this operation returns without affecting any de-  
 4830 ferred computations for this operation. If this array contains duplicate values, it  
 4831 implies in assignment of more than one value to the same location which leads to  
 4832 undefined results.

4833 **nrows** (IN) The number of values in **row\_indices** array. Must be equal to **size(u)**.

4834 **col\_index** (IN) The index of the column in C to assign. Must be in the range [0, **ncols(C)**).

4835 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 4836 should be specified. Non-default field/value pairs are listed as follows:

4837

| Param | Field    | Value         | Description  |
|-------|----------|---------------|--|
| C     | GrB_OUTP | GrB_REPLACE   | Output column in C is cleared (all elements removed) before result is stored in it.  |
| mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined. |
| mask  | GrB_MASK | GrB_COMP      | Use the complement of <b>mask</b> .  |

4838

## 4839 Return Values

4840 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4841 blocking mode, this indicates that the compatibility tests on  
 4842 dimensions and domains for the input arguments passed suc-  
 4843 cessfully. Either way, output matrix C is ready to be used in the  
 4844 next method of the sequence.

4845 **GrB\_PANIC** Unknown internal error.



4877 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4878 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
4879 compatibility as follows:

- 4880 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4881 must be from one of the pre-defined types of Table 3.2.
- 4882 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4883 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4884 of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4885 mulation operator.

4886 Two domains are compatible with each other if values from one domain can be cast to values in  
4887 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4888 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4889 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
4890 error listed above is returned.

4891 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4892 1.  $0 \leq \text{col\_index} < \mathbf{ncols}(\mathbf{C})$

4893 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
4894 is returned.

4895 From the arguments, the internal vectors, `mask`, and index array used in the computation are  
4896 formed ( $\leftarrow$  denotes copy):

- 4897 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a column of  $\mathbf{C}$  as follows:

$$4898 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

- 4899 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

- 4900 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$ .
- 4901 (b) If `mask`  $\neq$  `GrB_NULL`,
  - 4902 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
  - 4903 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
- 4904 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

- 4905 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

- 4906 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:

- 4907 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .



4908 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i]$ ,  $\forall i : 0 \leq i < \text{nrows}$ .

4909 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
4910 conditions must hold:

- 4911 1.  $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$
- 4912 2.  $\text{nrows} = \text{size}(\tilde{\mathbf{u}})$ .

4913 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
4914 match error listed above is returned.

4915 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4916 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4917 We are now ready to carry out the assign and any additional associated operations. We describe  
4918 this in terms of two intermediate vectors:

- 4919 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 4920 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4921 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4922 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \text{nrows} : i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4923 At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
4924 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
4925 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4926 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

4927 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 4928 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$4929 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4930 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
4931 of  $\tilde{\mathbf{c}}$  ( $\text{ind}(\tilde{\mathbf{c}})$ ) and remove from it all the indices of  $\tilde{\mathbf{c}}$  that are in the set of indices being  
4932 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\text{ind}(\tilde{\mathbf{t}})$ ).

4933 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4934 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$4935 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \text{ind}(\tilde{\mathbf{c}}))),$$

$$4936 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \text{ind}(\tilde{\mathbf{t}}),$$

4938 where the difference operator refers to set difference.

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$z_i = \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final result matrix,  $\mathbf{C}(:, \text{col\_index})$ . This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(:, \text{col\_index})$  on input to this operation are deleted and the new contents of the column is given by:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the column of the final result matrix,  $\mathbf{C}(:, \text{col\_index})$ , and elements of this column that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} \mathbf{L}(\mathbf{C}) = & \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \\ & \{(i, \text{col\_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ & \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 4.3.7.4 assign: Row variant

Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the output cannot be transposed, a different variant of `assign` is provided to assign to a column of a matrix.

## 4969 C Syntax

```
4970         GrB_Info GrB_assign(GrB_Matrix      C,  
4971                             const GrB_Vector mask,  
4972                             const GrB_BinaryOp accum,  
4973                             const GrB_Vector u,  
4974                             GrB_Index      row_index,  
4975                             const GrB_Index *col_indices,  
4976                             GrB_Index      ncols,  
4977                             const GrB_Descriptor desc);
```

## 4978 Parameters

4979 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values  
4980 that may be accumulated with the result of the assign operation. On output, this  
4981 matrix holds the results of the operation.

4982 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4983 stored into the specified row of the output matrix **C**. The mask dimensions must  
4984 match those of a single row of the matrix **C**. If the **GrB\_STRUCTURE** descriptor  
4985 is *not* set for the mask, the domain of the **Mask** matrix must be of type **bool** or  
4986 any of the predefined “built-in” types in Table 3.2. If the default mask is desired  
4987 (i.e., a mask that is all **true** with the dimensions of a row of **C**), **GrB\_NULL** should  
4988 be specified.

4989 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4990 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4991 specified.

4992 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of  
4993 **C**.

4994 **row\_index** (IN) The index of the row in **C** to assign. Must be in the range  $[0, \mathbf{nrows}(\mathbf{C})]$ .

4995 **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4996 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to  
4997 be assigned in order from index 0 to  $\mathbf{ncols} - 1$ , then **GrB\_ALL** should be specified.  
4998 Regardless of execution mode and return value, this array may be manipulated by  
4999 the caller after this operation returns without affecting any deferred computations  
5000 for this operation. If this array contains duplicate values, it implies in assignment  
5001 of more than one value to the same location which leads to undefined results.

5002 **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **size(u)**.

5003 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5004 should be specified. Non-default field/value pairs are listed as follows:  
5005

| Param | Field    | Value         | Description  |
|-------|----------|---------------|--|
| C     | GrB_OUTP | GrB_REPLACE   | Output row in C is cleared (all elements removed) before result is stored in it.   |
| mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined. |
| mask  | GrB_MASK | GrB_COMP      | Use the complement of <b>mask</b> .  |

## Return Values

|                          |  |
|--------------------------|--|
| GrB_SUCCESS              | In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence. |
| GrB_PANIC                | Unknown internal error.  |
| GrB_INVALID_OBJECT       | This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call <b>GrB_error()</b> to access any error messages generated by the implementation.                             |
| GrB_OUT_OF_MEMORY        | Not enough memory available for operation.   |
| GrB_UNINITIALIZED_OBJECT | One or more of the GraphBLAS objects has not been initialized by a call to <b>new</b> (or <b>dup</b> for vector or matrix parameters).   |
| GrB_INVALID_INDEX        | <b>row_index</b> is outside the allowable range (i.e., greater than <b>nrows(C)</b> ).   |
| GrB_INDEX_OUT_OF_BOUNDS  | A value in <b>col_indices</b> is greater than or equal to <b>ncols(C)</b> . In non-blocking mode, this can be reported as an execution error.  |
| GrB_DIMENSION_MISMATCH   | <b>mask</b> size and number of columns in C are not the same, or <b>ncols</b> $\neq$ <b>size(u)</b> .  |
| GrB_DOMAIN_MISMATCH      | The domains of the matrix and vector are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with <b>bool</b> (in the case where <b>desc[GrB_MASK].GrB_STRUCTURE</b> is not set).                         |
| GrB_NULL_POINTER         | Argument <b>col_indices</b> is a NULL pointer.   |

## Description

This variant of **GrB\_assign** computes the result of assigning a subset of locations in a row of a GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

5034  $C(\text{row\_index}, :) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_index}, :$   
 5035  $) = C(\text{row\_index}, :) \odot u$ . Taking order of `col_indices` into account it is more explicitly written as:

5036  $C(\text{row\_index}, \text{col\_indices}[j]) = u(j), \forall j : 0 \leq j < \text{ncols}, \text{ or}$   
 $C(\text{row\_index}, \text{col\_indices}[j]) = C(\text{row\_index}, \text{col\_indices}[j]) \odot u(j), \forall j : 0 \leq j < \text{ncols}$

5037 Logically, this operation occurs in three steps:

5038     **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
 5039     domains and dimensions are tested for compatibility.

5040     **Compute** The indicated computations are carried out.

5041     **Output** The result is written into the output matrix, possibly under control of a mask.

5042 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 5043 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5044 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 5045 3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5046 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
 5047 compatibility as follows:

- 5048 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 5049 must be from one of the pre-defined types of Table 3.2.
- 5050 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(u)$ .
- 5051 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5052 of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 5053 mulation operator.

5054 Two domains are compatible with each other if values from one domain can be cast to values in  
 5055 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5056 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5057 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 5058 error listed above is returned.

5059 The `row_index` parameter is checked for a valid value. The following condition must hold:

- 5060 1.  $0 \leq \text{row\_index} < \mathbf{nrows}(C)$

5061 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
 5062 is returned.

5063 From the arguments, the internal vectors, mask, and index array used in the computation are  
 5064 formed ( $\leftarrow$  denotes copy):

5065 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a row of  $\mathbf{C}$  as follows:

$$5066 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \mathbf{ncols}(\mathbf{C}), i = \text{row\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

5067 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

5068 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{ncols}(\mathbf{C})\} \rangle$ .

5069 (b) If `mask  $\neq$  GrB_NULL`,

5070 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

5071 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

5072 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

5073 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5074 4. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:

5075 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .

5076 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

5077 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
5078 conditions must hold:

5079 1.  $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

5080 2.  $\mathbf{ncols} = \mathbf{size}(\tilde{\mathbf{u}})$ .

5081 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
5082 match error listed above is returned.

5083 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5084 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5085 We are now ready to carry out the assign and any additional associated operations. We describe  
5086 this in terms of two intermediate vectors:

- 5087 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- 5088 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5089 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$5090 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \mathbf{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

5091 At this point, if any value of  $\tilde{\mathbf{J}}[j]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation  
5092 ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING`  
5093 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
5094 result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

5095 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

5096 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5097 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5098 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
5099 of  $\tilde{\mathbf{c}}$  ( $\mathbf{ind}(\tilde{\mathbf{c}})$ ) and remove from it all the indices of  $\tilde{\mathbf{c}}$  that are in the set of indices being  
5100 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5101 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5102 indices in  $\tilde{\mathbf{c}}$  and  $\tilde{\mathbf{t}}$ .

$$5103 \quad z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5104 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5106 where the difference operator refers to set difference.

5107 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5108 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5109 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5110 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5111 \quad z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$5112 \quad z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$5113 \quad z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

5115 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5117 Finally, the set of output values that make up the  $\tilde{\mathbf{z}}$  vector are written into the column of the final  
5118 result matrix,  $\mathbf{C}(\text{row\_index}, :)$ . This is carried out under control of the mask which acts as a “write  
5119 mask”.

5120 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}(\text{row\_index}, :)$  on input to this  
5121 operation are deleted and the new contents of the column is given by:

$$5122 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5123 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
5124 copied into the column of the final result matrix,  $\mathbf{C}(\text{row\_index}, :)$ , and elements of this column  
5125 that fall outside the set indicated by the mask are unchanged:

$$5126 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup$$

$$5127 \quad \{(\text{row\_index}, j, \tilde{\mathbf{c}}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup$$

$$5128 \quad \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5129 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5130 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5131 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may  
5132 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 5133 4.3.7.5 assign: Constant vector variant[Scott: NEW CONTENT]

5134 Assign the same value to a specified subset of vector elements. With the use of GrB\_ALL, the entire  
5135 destination vector can be filled with the constant.

#### 5136 C Syntax

```
5137     GrB_Info GrB_assign(GrB_Vector      w,  
5138                        const GrB_Vector mask,  
5139                        const GrB_BinaryOp accum,  
5140                        <type>          val,  
5141                        const GrB_Index  *indices,  
5142                        GrB_Index        nindices,  
5143                        const GrB_Descriptor desc);
```

```
5144     GrB_Info GrB_assign(GrB_Vector      w,  
5145                        const GrB_Vector mask,  
5146                        const GrB_BinaryOp accum,  
5147                        const GrB_Scalar  s,  
5148                        const GrB_Index  *indices,  
5149                        GrB_Index        nindices,  
5150                        const GrB_Descriptor desc);
```

#### 5151 Parameters

5152 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5153 that may be accumulated with the result of the assign operation. On output, this  
5154 vector holds the results of the operation.

5155 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5156 stored into the output vector **w**. The mask dimensions must match those of the  
5157 vector **w**. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
5158 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5159 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5160 dimensions of **w**), GrB\_NULL should be specified.

5161 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5162 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
5163 specified.

5164 **val** (IN) Scalar value to assign to (a subset of) **w**.

5165 **s** (IN) Scalar value to assign to (a subset of) **w**.

5166 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5167 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0



5168 to `nindices - 1`, then `GrB_ALL` should be specified. Regardless of execution mode  
5169 and return value, this array may be manipulated by the caller after this operation  
5170 returns without affecting any deferred computations for this operation. In this  
5171 variant, the specific order of the values in the array has no effect on the result.  
5172 Unlike other variants, if there are duplicated values in this array the result is still  
5173 defined.

5174 **nindices** (IN) The number of values in `indices` array. Must be in the range: `[0, size(w)]`. If  
5175 `nindices` is zero, the operation becomes a NO-OP.

5176 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
5177 should be specified. Non-default field/value pairs are listed as follows:

5178

| Param                  | Field                 | Value                      | Description  |
|------------------------|-----------------------|----------------------------|--|
| <code>w</code>         | <code>GrB_OUTP</code> | <code>GrB_REPLACE</code>   | Output vector <code>w</code> is cleared (all elements removed) before the result is stored in it.  |
| 5179 <code>mask</code> | <code>GrB_MASK</code> | <code>GrB_STRUCTURE</code> | The write mask is constructed from the structure (pattern of stored values) of the input <code>mask</code> vector. The stored values are not examined. |
| <code>mask</code>      | <code>GrB_MASK</code> | <code>GrB_COMP</code>      | Use the complement of <code>mask</code> .  |

## 5180 Return Values

5181 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5182 blocking mode, this indicates that the compatibility tests on  
5183 dimensions and domains for the input arguments passed suc-  
5184 cessfully. Either way, output vector `w` is ready to be used in the  
5185 next method of the sequence.

5186 **GrB\_PANIC** Unknown internal error.

5187 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
5188 opaque GraphBLAS objects (input or output) is in an invalid  
5189 state caused by a previous execution error. Call `GrB_error()` to  
5190 access any error messages generated by the implementation.

5191 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5192 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
5193 by a call to `new` (or `dup` for vector parameters).

5194 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in `indices` is greater than or equal to `size(w)`. In non-  
5195 blocking mode, this can be reported as an execution error.

5196 **GrB\_DIMENSION\_MISMATCH** `mask` and `w` dimensions are incompatible, or `nindices` is not less  
5197 than `size(w)`.



5228 4. If **accum** is not **GrB\_NULL**, then either **D(val)** or **D(s)**, depending on the signature of the  
 5229 method, must be compatible with **D<sub>in2</sub>(accum)** of the accumulation operator.

5230 Two domains are compatible with each other if values from one domain can be cast to values in  
 5231 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5232 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5233 any compatibility rule above is violated, execution of **GrB\_assign** ends and the domain mismatch  
 5234 error listed above is returned.

5235 From the arguments, the internal vectors, mask and index array used in the computation are formed  
 5236 ( $\leftarrow$  denotes copy):

- 5237 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5238 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:
  - 5239 (a) If **mask** = **GrB\_NULL**, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 5240 (b) If **mask**  $\neq$  **GrB\_NULL**,
    - 5241 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 5242 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 5243 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5244 3. Scalar  $\tilde{s} \leftarrow s$  (**GrB\_Scalar** version only).
- 5245 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument **indices** as follows:
  - 5246 (a) If **indices** = **GrB\_ALL**, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$ .
  - 5247 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$ .

5248 The internal vector and mask are checked for dimension compatibility. The following conditions  
 5249 must hold:

- 5250 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5251 2.  $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$ .

5252 If any compatibility rule above is violated, execution of **GrB\_assign** ends and the dimension mis-  
 5253 match error listed above is returned.

5254 From this point forward, in **GrB\_NONBLOCKING** mode, the method can optionally exit with  
 5255 **GrB\_SUCCESS** return code and defer any computation and/or execution error codes.

5256 We are now ready to carry out the assign and any additional associated operations. We describe  
 5257 this in terms of two intermediate vectors:

- 5258 •  $\tilde{\mathbf{t}}$ : The vector holding the copies of the scalar, either **val** or  $\tilde{s}$ , in their destination locations  
 5259 relative to  $\tilde{\mathbf{w}}$ .

5260 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5261 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows. If a non-opaque scalar  $\mathbf{val}$  is provided:

$$5262 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5263 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$5264 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5265 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$5266 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5267 If  $\tilde{\mathbf{I}}$  is empty, this operation results in an empty vector,  $\tilde{\mathbf{t}}$ . Otherwise, if any value in the  $\tilde{\mathbf{I}}$  array  
 5268 is not in the range  $[0, \mathbf{size}(\tilde{\mathbf{w}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds  
 5269 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a  
 5270 sequence-terminating `GrB_wait()` is called. Regardless, the result vector,  $\mathbf{w}$ , is invalid from this  
 5271 point forward in the sequence.

5272 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

5273 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5274 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5275 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
 5276 of  $\tilde{\mathbf{w}}$  ( $\mathbf{ind}(\tilde{\mathbf{w}})$ ) and remove from it all the indices of  $\tilde{\mathbf{w}}$  that are in the set of indices being  
 5277 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5278 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5279 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5280 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5281 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

5283 where the difference operator refers to set difference. We note that in this case of assigning  
 5284 a constant,  $\{\tilde{\mathbf{I}}[k], \forall k\}$  and  $\mathbf{ind}(\tilde{\mathbf{t}})$  are identical.

5285 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5286 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5287 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5288 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$5289 \quad z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$5290 \quad z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$5291 \quad z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

5294 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5295 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 5296 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 5297 mask which acts as a “write mask”.

- 5298 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 5299 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5300 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5301 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 5302 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 5303 mask are unchanged:

$$5304 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5305 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 5306 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 5307 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 5308 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 5309 sequence.

#### 5310 4.3.7.6 assign: Constant matrix variant[Scott: NEW CONTENT]

5311 Assign the same value to a specified subset of matrix elements. With the use of GrB\_ALL, the  
 5312 entire destination matrix can be filled with the constant.

### 5313 C Syntax

```
5314      GrB_Info GrB_assign(GrB_Matrix      C,
5315                          const GrB_Matrix Mask,
5316                          const GrB_BinaryOp accum,
5317                          <type>         val,
5318                          const GrB_Index *row_indices,
5319                          GrB_Index      nrows,
5320                          const GrB_Index *col_indices,
5321                          GrB_Index      ncols,
5322                          const GrB_Descriptor desc);
```

```
5323      GrB_Info GrB_assign(GrB_Matrix      C,
5324                          const GrB_Matrix Mask,
5325                          const GrB_BinaryOp accum,
5326                          const GrB_Scalar s,
5327                          const GrB_Index *row_indices,
5328                          GrB_Index      nrows,
```

```

5329         const GrB_Index      *col_indices,
5330         GrB_Index             ncols,
5331         const GrB_Descriptor  desc);

```

## 5332 Parameters

5333 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
5334 that may be accumulated with the result of the assign operation. On output, the  
5335 matrix holds the results of the operation.

5336 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
5337 stored into the output matrix **C**. The mask dimensions must match those of the  
5338 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5339 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
5340 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5341 dimensions of **C**), **GrB\_NULL** should be specified.

5342 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
5343 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5344 specified.

5345 **val** (IN) Scalar value to assign to (a subset of) **C**.

5346 **s** (IN) Scalar value to assign to (a subset of) **C**.

5347 **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
5348 that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** − 1,  
5349 then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
5350 this array may be manipulated by the caller after this operation returns without  
5351 affecting any deferred computations for this operation. Unlike other variants, if  
5352 there are duplicated values in this array the result is still defined.

5353 **nrows** (IN) The number of values in **row\_indices** array. Must be in the range: [0, **nrows**(**C**)].  
5354 If **nrows** is zero, the operation becomes a NO-OP.

5355 **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**  
5356 that are assigned. If all columns of **C** are to be assigned in order from 0 to **ncols** − 1,  
5357 then **GrB\_ALL** should be specified. Regardless of execution mode and return value,  
5358 this array may be manipulated by the caller after this operation returns without  
5359 affecting any deferred computations for this operation. Unlike other variants, if  
5360 there are duplicated values in this array the result is still defined.

5361 **ncols** (IN) The number of values in **col\_indices** array. Must be in the range: [0, **ncols**(**C**)].  
5362 If **ncols** is zero, the operation becomes a NO-OP.

5363 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5364 should be specified. Non-default field/value pairs are listed as follows:

5365

| Param | Field    | Value         | Description   |
|-------|----------|---------------|---|
| C     | GrB_OUTP | GrB_REPLACE   | Output matrix C is cleared (all elements removed) before the result is stored in it.  |
| Mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined. |
| Mask  | GrB_MASK | GrB_COMP      | Use the complement of Mask.   |

## Return Values

|                          |  |
|--------------------------|--|
| GrB_SUCCESS              | In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence. |
| GrB_PANIC                | Unknown internal error.  |
| GrB_INVALID_OBJECT       | This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call <code>GrB_error()</code> to access any error messages generated by the implementation.                       |
| GrB_OUT_OF_MEMORY        | Not enough memory available for the operation.   |
| GrB_UNINITIALIZED_OBJECT | One or more of the GraphBLAS objects has not been initialized by a call to <code>new</code> (or <code>dup</code> for vector parameters).   |
| GrB_INDEX_OUT_OF_BOUNDS  | A value in <code>row_indices</code> is greater than or equal to <code>nrows(C)</code> , or a value in <code>col_indices</code> is greater than or equal to <code>ncols(C)</code> . In non-blocking mode, this can be reported as an execution error.                                 |
| GrB_DIMENSION_MISMATCH   | Mask and C dimensions are incompatible, <code>nrows</code> is not less than <code>nrows(C)</code> , or <code>ncols</code> is not less than <code>ncols(C)</code> .   |
| GrB_DOMAIN_MISMATCH      | The domains of the matrix and scalar are incompatible with each other or the corresponding domains of the accumulation operator, or the mask's domain is not compatible with <code>bool</code> (in the case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).             |
| GrB_NULL_POINTER         | Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code> is a NULL pointer, or both.  |

## Description

This variant of `GrB_assign` computes the result of assigning a constant scalar value – either `val` or `s` – to locations in a destination GraphBLAS matrix: Either `C(row_indices, col_indices) = val`

5395 or  $C(\text{row\_indices}, \text{col\_indices}) = s$  is performed. If an optional binary accumulation operator  
 5396  $(\odot)$  is provided, then either  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot \text{val}$  or  
 5397  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot s$  is performed. More explicitly, if a  
 5398 non-opaque value  $\text{val}$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = \text{val}, \text{ or} \\ & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot \text{val} \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5400 Correspondingly, if a `GrB_Scalar`  $s$  is provided:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_indices}[j]) = s, \text{ or} \\ & C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot s \\ & \quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

5402 Logically, this operation occurs in three steps:

5403     Setup The internal vectors and mask used in the computation are formed and their domains  
 5404             and dimensions are tested for compatibility.

5405     Compute The indicated computations are carried out.

5406     Output The result is written into the output matrix, possibly under control of a mask.

5407 Up to two argument matrices are used in the `GrB_assign` operation:

- 5408     1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5409     2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \text{nrows}(\text{Mask}), \text{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)

5410 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain  
 5411 compatibility as follows:

- 5412     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 5413         must be from one of the pre-defined types of Table 3.2.
- 5414     2.  $\mathbf{D}(C)$  must be compatible with either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5415         method.
- 5416     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5417         of the accumulation operator.
- 5418     4. If `accum` is not `GrB_NULL`, then either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the  
 5419         method, must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.



Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch error listed above is returned.

From the arguments, the internal matrices, index arrays, and mask used in the computation are formed ( $\leftarrow$  denotes copy):

1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument `Mask` as follows:
  - (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - (b) If `Mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
3. Scalar  $\tilde{s} \leftarrow s$  (`GrB_Scalar` version only).
4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
  - (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
  - (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
  - (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \mathbf{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

The internal matrix and mask are checked for dimension compatibility. The following conditions must hold:

1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
3.  $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$ .
4.  $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$ .

If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mismatch error listed above is returned.

5452 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 5453 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5454 We are now ready to carry out the assign and any additional associated operations. We describe  
 5455 this in terms of two intermediate matrices:

- 5456 •  $\tilde{\mathbf{T}}$ : The matrix holding the copies of the scalar, either `val` or  $\tilde{s}$ , in their destination locations  
 5457 relative to  $\tilde{\mathbf{C}}$ .
- 5458 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5459 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows. If a non-opaque scalar `val` is provided:

$$\begin{aligned} 5460 \quad \tilde{\mathbf{T}} = & \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5461 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e., `size`( $\tilde{s}$ ) = 1):

$$\begin{aligned} 5462 \quad \tilde{\mathbf{T}} = & \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5463 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e., `size`( $\tilde{s}$ ) = 0):

$$5464 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5465 If either  $\tilde{\mathbf{I}}$  or  $\tilde{\mathbf{J}}$  is empty, this operation results in an empty matrix,  $\tilde{\mathbf{T}}$ . Otherwise, if any value  
 5466 in the  $\tilde{\mathbf{I}}$  array is not in the range  $[0, \mathbf{nrows}(\tilde{\mathbf{C}}))$  or any value in the  $\tilde{\mathbf{J}}$  array is not in the range  
 5467  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds error listed above is  
 5468 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating  
 5469 `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from this point forward in the  
 5470 sequence.

5471 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 5472 • If `accum` = `GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$\begin{aligned} 5473 \quad \tilde{\mathbf{Z}} = & \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5474 \quad & \{(i, j, Z_{ij}) \mid (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle. \end{aligned}$$

5475 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 5476 of  $\tilde{\mathbf{C}}$  ( $\mathbf{ind}(\tilde{\mathbf{C}})$ ) and remove from it all the indices of  $\tilde{\mathbf{C}}$  that are in the set of indices being  
 5477 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

5478 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 5479 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 5480 \quad Z_{ij} = & \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5481 \quad & \\ 5482 \quad & \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \end{aligned}$$

5483 where the difference operator refers to set difference. We note that, in this particular case of  
 5484 assigning a constant to a matrix, the sets  $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\}$  and  $\mathbf{ind}(\tilde{\mathbf{T}})$  are identical.

- If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ , using what is called a *standard matrix mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

#### 4.3.8 apply: Apply a function to the elements of an object

Computes the transformation of the values of the elements of a vector or a matrix using a unary function, or a binary function where one argument is bound to a scalar.

##### 4.3.8.1 apply: Vector variant

Computes the transformation of the values of the elements of a vector using a unary function.

## 5515 C Syntax

```

5516      GrB_Info GrB_apply(GrB_Vector      w,
5517                        const GrB_Vector  mask,
5518                        const GrB_BinaryOp accum,
5519                        const GrB_UnaryOp  op,
5520                        const GrB_Vector  u,
5521                        const GrB_Descriptor desc);

```

## 5522 Parameters

5523     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5524     that may be accumulated with the result of the apply operation. On output, this  
5525     vector holds the results of the operation.

5526     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5527     stored into the output vector **w**. The mask dimensions must match those of the  
5528     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5529     of the mask vector must be of type **bool** or any of the predefined “built-in” types  
5530     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5531     dimensions of **w**), **GrB\_NULL** should be specified.

5532     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5533     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5534     specified.

5535     **op** (IN) A unary operator applied to each element of input vector **u**.

5536     **u** (IN) The GraphBLAS vector to which the unary function is applied.

5537     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5538     should be specified. Non-default field/value pairs are listed as follows:

| Param       | Field           | Value                | Description  |
|-------------|-----------------|----------------------|--|
| <b>w</b>    | <b>GrB_OUTP</b> | <b>GrB_REPLACE</b>   | Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.  |
| <b>mask</b> | <b>GrB_MASK</b> | <b>GrB_STRUCTURE</b> | The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined. |
| <b>mask</b> | <b>GrB_MASK</b> | <b>GrB_COMP</b>      | Use the complement of <b>mask</b> .  |

## 5541 Return Values

5542     **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5543     blocking mode, this indicates that the compatibility tests on di-  
5544     mensions and domains for the input arguments passed successfully.

5545 Either way, output vector  $w$  is ready to be used in the next method  
 5546 of the sequence.

5547 **GrB\_PANIC** Unknown internal error.

5548 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 5549 GraphBLAS objects (input or output) is in an invalid state caused  
 5550 by a previous execution error. Call **GrB\_error()** to access any error  
 5551 messages generated by the implementation.

5552 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5553 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 5554 a call to **new** (or **dup** for vector parameters).

5555 **GrB\_DIMENSION\_MISMATCH**  $mask$ ,  $w$  and/or  $u$  dimensions are incompatible.

5556 **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the corre-  
 5557 sponding domains of the accumulation operator or unary function,  
 5558 or the mask's domain is not compatible with **bool** (in the case where  
 5559  $desc[GrB\_MASK].GrB\_STRUCTURE$  is not set).

## 5560 Description

5561 This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a  
 5562 GraphBLAS vector:  $w = f(u)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
 5563  $w = w \odot f(u)$ .

5564 Logically, this operation occurs in three steps:

5565 **Setup** The internal vectors and mask used in the computation are formed and their domains  
 5566 and dimensions are tested for compatibility.

5567 **Compute** The indicated computations are carried out.

5568 **Output** The result is written into the output vector, possibly under control of a mask.

5569 Up to three argument vectors are used in this **GrB\_apply** operation:

- 5570 1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5571 2.  $mask = \langle \mathbf{D}(mask), \mathbf{size}(mask), \mathbf{L}(mask) = \{(i, m_i)\} \rangle$  (optional)
- 5572 3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5573 The argument vectors, unary operator and the accumulation operator (if provided) are tested for  
 5574 domain compatibility as follows:

- 5575 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
5576 must be from one of the pre-defined types of Table 3.2.
- 5577 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5578 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5579 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5580  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5581 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in}(\text{op})$ .

5582 Two domains are compatible with each other if values from one domain can be cast to values in  
5583 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5584 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5585 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
5586 error listed above is returned.

5587 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
5588 denotes copy):

- 5589 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5590 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 5591 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
  - 5592 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 5593 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
    - 5594 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 5595 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5596 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5597 The internal vectors and masks are checked for dimension compatibility. The following conditions  
5598 must hold:

- 5599 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$
- 5600 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

5601 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5602 error listed above is returned.

5603 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5604 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5605 We are now ready to carry out the apply and any additional associated operations. We describe  
5606 this in terms of two intermediate vectors:

- 5607 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the unary operator to the input vector  $\tilde{\mathbf{u}}$ .
- 5608 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5609 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$5610 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle,$$

5611 where  $f = \mathbf{f}(\text{op})$ .

5612 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 5613 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 5614 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5615 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5616 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5617 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 5618 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 5619 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 5620 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 5621 \end{aligned}$$

5622 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5624 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
5625 using what is called a *standard vector mask and replace*. This is carried out under control of the  
5626 mask which acts as a “write mask”.

- 5627 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
5628 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5629 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- 5630 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
5631 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
5632 mask are unchanged:

$$5633 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

5634 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5635 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5636 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
5637 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5638 sequence.

### 4.3.8.2 apply: Matrix variant

Computes the transformation of the values of the elements of a matrix using a unary function.

#### C Syntax

```
GrB_Info GrB_apply(GrB_Matrix      C,
                  const GrB_Matrix  Mask,
                  const GrB_BinaryOp accum,
                  const GrB_UnaryOp  op,
                  const GrB_Matrix  A,
                  const GrB_Descriptor desc);
```

#### Parameters

**C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values that may be accumulated with the result of the apply operation. On output, the matrix holds the results of the operation.

**Mask** (IN) An optional “write” mask that controls which results from this operation are stored into the output matrix C. The mask dimensions must match those of the matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the dimensions of C), `GrB_NULL` should be specified.

**accum** (IN) An optional binary operator used for accumulating entries into existing C entries. If assignment rather than accumulation is desired, `GrB_NULL` should be specified.

**op** (IN) A unary operator applied to each element of input matrix A.

**A** (IN) The GraphBLAS matrix to which the unary function is applied.

**desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL` should be specified. Non-default field/value pairs are listed as follows:

| Param | Field                 | Value                      | Description  |
|-------|-----------------------|----------------------------|--|
| C     | <code>GrB_OUTP</code> | <code>GrB_REPLACE</code>   | Output matrix C is cleared (all elements removed) before the result is stored in it.   |
| Mask  | <code>GrB_MASK</code> | <code>GrB_STRUCTURE</code> | The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined. |
| Mask  | <code>GrB_MASK</code> | <code>GrB_COMP</code>      | Use the complement of <b>Mask</b> .  |
| A     | <code>GrB_INP0</code> | <code>GrB_TRAN</code>      | Use transpose of A for the operation.  |



## 5667 Return Values

5668                   GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
5669                   blocking mode, this indicates that the compatibility tests on  
5670                   dimensions and domains for the input arguments passed suc-  
5671                   cessfully. Either way, output matrix C is ready to be used in the  
5672                   next method of the sequence.

5673                   GrB\_PANIC Unknown internal error.

5674                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
5675                   opaque GraphBLAS objects (input or output) is in an invalid  
5676                   state caused by a previous execution error. Call `GrB_error()` to  
5677                   access any error messages generated by the implementation.

5678                   GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

5679                   GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
5680                   by a call to `new` (or `Matrix_dup` for matrix parameters).

5681                   GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible,  $\text{nrows} \neq \text{nrows}(C)$ , or  
5682                    $\text{ncols} \neq \text{ncols}(C)$ .

5683                   GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
5684                   corresponding domains of the accumulation operator or unary  
5685                   function, or the mask's domain is not compatible with `bool` (in  
5686                   the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 5687 Description

5688 This variant of `GrB_apply` computes the result of applying a unary function to the elements of a  
5689 GraphBLAS matrix:  $C = f(A)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
5690  $C = C \odot f(A)$ .

5691 Logically, this operation occurs in three steps:

5692                   **Setup** The internal matrices and mask used in the computation are formed and their domains  
5693                   and dimensions are tested for compatibility.

5694                   **Compute** The indicated computations are carried out.

5695                   **Output** The result is written into the output matrix, possibly under control of a mask.

5696 Up to three argument matrices are used in the `GrB_apply` operation:

- 5697                   1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$   
5698                   2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \text{nrows}(\text{Mask}), \text{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)

5699 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

5700 The argument matrices, unary operator and the accumulation operator (if provided) are tested for  
5701 domain compatibility as follows:

- 5702 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{Mask})$   
5703 must be from one of the pre-defined types of Table 3.2.
- 5704 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the unary operator.
- 5705 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
5706 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the unary operator must be compatible with  
5707  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 5708 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in}(\mathbf{op})$  of the unary operator.

5709 Two domains are compatible with each other if values from one domain can be cast to values in  
5710 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5711 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5712 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
5713 error listed above is returned.

5714 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
5715 are formed ( $\leftarrow$  denotes copy):

- 5716 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 5717 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument **Mask** as follows:
  - 5718 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
5719  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 5720 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 5721 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
5722  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 5723 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
5724  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 5725 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 5726 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

5727 The internal matrices and mask are checked for dimension compatibility. The following conditions  
5728 must hold:

- 5729 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 5730 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 5731 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

5732 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

5733 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5734 error listed above is returned.

5735 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5736 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5737 We are now ready to carry out the apply and any additional associated operations. We describe  
5738 this in terms of two intermediate matrices:

- 5739 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the unary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 5740 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5741 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$5742 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

5743 where  $f = \mathbf{f}(\mathbf{op})$ .

5744 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 5745 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 5746 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5747 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5748 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5749 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5750 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$5751 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$5752 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

5753 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5756 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5757 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5758 mask which acts as a “write mask”.

- 5759 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5760 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5761 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5762 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 5763 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 5764 mask are unchanged:

$$5765 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\text{ind}(\mathbf{C}) \cap \text{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\text{ind}(\tilde{\mathbf{Z}}) \cap \text{ind}(\tilde{\mathbf{M}}))\}.$$

5766 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 5767 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 5768 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 5769 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 5770 sequence.

#### 5771 4.3.8.3 apply: Vector-BinaryOp variants[Scott: NEW CONTENT]

5772 Computes the transformation of the values of the stored elements of a vector using a binary operator  
 5773 and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the first argument  
 5774 to the binary operator and stored elements of the vector are passed as the second argument. In the  
 5775 *bind-second* variant, the elements of the vector are passed as the first argument and the specified  
 5776 scalar value is passed as the second argument. The scalar can be passed either as a non-opaque  
 5777 variable or as a GrB\_Scalar object.

#### 5778 C Syntax

```
5779 // bind-first + scalar value
5780 GrB_Info GrB_apply(GrB_Vector          w,
5781                   const GrB_Vector      mask,
5782                   const GrB_BinaryOp    accum,
5783                   const GrB_BinaryOp    op,
5784                   <type>                val,
5785                   const GrB_Vector      u,
5786                   const GrB_Descriptor   desc);
```

```
5787 // bind-first + GraphBLAS scalar
5788 GrB_Info GrB_apply(GrB_Vector          w,
5789                   const GrB_Vector      mask,
5790                   const GrB_BinaryOp    accum,
5791                   const GrB_BinaryOp    op,
5792                   const GrB_Scalar      s,
5793                   const GrB_Vector      u,
5794                   const GrB_Descriptor   desc);
```

```
5795 // bind-second + scalar value
5796 GrB_Info GrB_apply(GrB_Vector          w,
5797                   const GrB_Vector      mask,
```

```

5798             const GrB_BinaryOp      accum,
5799             const GrB_BinaryOp      op,
5800             const GrB_Vector        u,
5801             <type>                  val,
5802             const GrB_Descriptor     desc);

5803 // bind-second + GraphBLAS scalar
5804 GrB_Info GrB_apply(GrB_Vector        w,
5805                   const GrB_Vector    mask,
5806                   const GrB_BinaryOp  accum,
5807                   const GrB_BinaryOp  op,
5808                   const GrB_Vector    u,
5809                   const GrB_Scalar    s,
5810                   const GrB_Descriptor desc);

```

## 5811 Parameters

5812     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5813     that may be accumulated with the result of the apply operation. On output, this  
5814     vector holds the results of the operation.

5815     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5816     stored into the output vector **w**. The mask dimensions must match those of the  
5817     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5818     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5819     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5820     dimensions of **w**), **GrB\_NULL** should be specified.

5821     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5822     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5823     specified.

5824     **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar  
5825     value, **val**.

5826     **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as  
5827     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
5828     argument in the *bind-second* variant.

5829     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
5830     argument in the *bind-first* variant, or the right-hand (second) argument in the  
5831     *bind-second* variant.

5832     **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand  
5833     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
5834     the *bind-second* variant. It must not be empty.

5835 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
5836 should be specified. Non-default field/value pairs are listed as follows:

5837

| Param | Field    | Value         | Description  |
|-------|----------|---------------|--|
| w     | GrB_OUTP | GrB_REPLACE   | Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.  |
| mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined. |
| mask  | GrB_MASK | GrB_COMP      | Use the complement of <b>mask</b> .  |

5838

## 5839 Return Values

5840 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
5841 blocking mode, this indicates that the compatibility tests on di-  
5842 mensions and domains for the input arguments passed successfully.  
5843 Either way, output vector **w** is ready to be used in the next method  
5844 of the sequence.

5845 GrB\_PANIC Unknown internal error.

5846 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
5847 GraphBLAS objects (input or output) is in an invalid state caused  
5848 by a previous execution error. Call GrB\_error() to access any error  
5849 messages generated by the implementation.

5850 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

5851 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
5852 a call to new (or dup for vector parameters).

5853 GrB\_DIMENSION\_MISMATCH mask, w and/or u dimensions are incompatible.

5854 GrB\_DOMAIN\_MISMATCH The domains of the various vectors and scalar are incompatible with  
5855 the corresponding domains of the binary operator or accumulation  
5856 operator, or the mask's domain is not compatible with bool (in the  
5857 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

5858 GrB\_EMPTY\_OBJECT The GrB\_Scalar **s** used in the call is empty (**nvals(s) = 0**) and  
5859 therefore a value cannot be passed to the binary operator.

## 5860 Description

5861 This variant of GrB\_apply computes the result of applying a binary operator to the elements of a  
5862 GraphBLAS vector each composed with a scalar constant, either **val** or **s**:

5863                   bind-first:      $w = f(\text{val}, u)$  or  $w = f(s, u)$

5864                   bind-second:    $w = f(u, \text{val})$  or  $w = f(u, s)$ ,

5865 or if an optional binary accumulation operator ( $\odot$ ) is provided:

5866                   bind-first:      $w = w \odot f(\text{val}, u)$  or  $w = w \odot f(s, u)$

5867                   bind-second:     $w = w \odot f(u, \text{val})$  or  $w = w \odot f(u, s)$ .

5868 Logically, this operation occurs in three steps:

5869     **Setup** The internal vectors and mask used in the computation are formed and their domains  
5870             and dimensions are tested for compatibility.

5871     **Compute** The indicated computations are carried out.

5872     **Output** The result is written into the output vector, possibly under control of a mask.

5873 Up to three argument vectors are used in this GrB\_apply operation:

5874     1.  $w = \langle \mathbf{D}(w), \mathbf{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$

5875     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \mathbf{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)

5876     3.  $u = \langle \mathbf{D}(u), \mathbf{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5877 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are  
5878 tested for domain compatibility as follows:

5879     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
5880         must be from one of the pre-defined types of Table 3.2.

5881     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.

5882     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5883         of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
5884          $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

5885     4.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

5886     5. If bind-first:

5887         (a)  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

5888         (b) If the non-opaque scalar **val** is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
5889             of the binary operator.

5890         (c) If the GrB\_Scalar **s** is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
5891             binary operator.

- 5892 6. If bind-second:
- 5893 (a)  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the binary operator.
- 5894 (b) If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$
- 5895 of the binary operator.
- 5896 (c) If the `GrB_Scalar`  $\mathbf{s}$  is provided, then  $\mathbf{D}(\mathbf{s})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the
- 5897 binary operator.

5898 Two domains are compatible with each other if values from one domain can be cast to values in

5899 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

5900 compatible with each other. A domain from a user-defined type is only compatible with itself. If

5901 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

5902 error listed above is returned.

5903 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$

5904 denotes copy):

- 5905 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5906 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 5907 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 5908 (b) If `mask  $\neq$  GrB_NULL`,
- 5909 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 5910 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 5911 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5912 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 5913 4. Scalar  $\tilde{\mathbf{s}} \leftarrow \mathbf{s}$  (GraphBLAS scalar case).

5914 The internal vectors and masks are checked for dimension compatibility. The following conditions

5915 must hold:

- 5916 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5917 2.  $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{size}(\tilde{\mathbf{w}})$ .

5918 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch

5919 error listed above is returned.

5920 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

5921 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5922 If an empty `GrB_Scalar`  $\tilde{\mathbf{s}}$  is provided ( $\mathbf{nvals}(\tilde{\mathbf{s}}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.

5923 If a non-empty `GrB_Scalar`,  $\tilde{\mathbf{s}}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{\mathbf{s}}) = 1$ ), we then create an internal variable

5924 `val` with the same domain as  $\tilde{\mathbf{s}}$  and set `val = val( $\tilde{\mathbf{s}}$ )`.

5925 We are now ready to carry out the apply and any additional associated operations. We describe

5926 this in terms of two intermediate vectors:



- 5927 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the binary operator to the input vector  $\tilde{\mathbf{u}}$ .
- 5928 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5929 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as one of the following:

$$\begin{aligned}
 5930 \quad \text{bind-first:} \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\text{val}, \tilde{\mathbf{u}}(i))) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \\
 5931 \quad \text{bind-second:} \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\text{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i), \text{val})) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,
 \end{aligned}$$

5932 where  $f = \mathbf{f}(\text{op})$ .

5933 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 5934 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 5935 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5936 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5937 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5938 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned}
 5939 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\
 5940 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 5941 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\
 5942 \quad & \\
 5943 \quad &
 \end{aligned}$$

5944 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5945 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
5946 using what is called a *standard vector mask and replace*. This is carried out under control of the  
5947 mask which acts as a “write mask”.

- 5948 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
5949 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5950 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5951 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
5952 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
5953 mask are unchanged:

$$5954 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5955 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5956 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5957 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
5958 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5959 sequence.

#### 5960 4.3.8.4 apply: Matrix-BinaryOp variants[Scott: NEW CONTENT]

5961 Computes the transformation of the values of the stored elements of a matrix using a binary  
5962 operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the  
5963 first argument to the binary operator and stored elements of the matrix are passed as the second  
5964 argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument  
5965 and the specified scalar value is passed as the second argument. The scalar can be passed either as  
5966 a non-opaque variable or as a GrB\_Scalar object.

#### 5967 C Syntax

```
5968 // bind-first + scalar value
5969 GrB_Info GrB_apply(GrB_Matrix      C,
5970                   const GrB_Matrix Mask,
5971                   const GrB_BinaryOp accum,
5972                   const GrB_BinaryOp op,
5973                   <type>           val,
5974                   const GrB_Matrix A,
5975                   const GrB_Descriptor desc);
```

```
5976 // bind-first + GraphBLAS scalar
5977 GrB_Info GrB_apply(GrB_Matrix      C,
5978                   const GrB_Matrix Mask,
5979                   const GrB_BinaryOp accum,
5980                   const GrB_BinaryOp op,
5981                   const GrB_Scalar s,
5982                   const GrB_Matrix A,
5983                   const GrB_Descriptor desc);
```

```
5984 // bind-second + scalar value
5985 GrB_Info GrB_apply(GrB_Matrix      C,
5986                   const GrB_Matrix Mask,
5987                   const GrB_BinaryOp accum,
5988                   const GrB_BinaryOp op,
5989                   const GrB_Matrix A,
5990                   <type>           val,
5991                   const GrB_Descriptor desc);
```

```
5992 // bind-second + GraphBLAS scalar
5993 GrB_Info GrB_apply(GrB_Matrix      C,
5994                   const GrB_Matrix Mask,
5995                   const GrB_BinaryOp accum,
5996                   const GrB_BinaryOp op,
5997                   const GrB_Matrix A,
```

```

5998         const GrB_Scalar      s,
5999         const GrB_Descriptor desc);

```

## 6000 Parameters

6001     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6002     that may be accumulated with the result of the apply operation. On output, the  
6003     matrix holds the results of the operation.

6004     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6005     stored into the output matrix C. The mask dimensions must match those of the  
6006     matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
6007     of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
6008     in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
6009     dimensions of C), `GrB_NULL` should be specified.

6010     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
6011     entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
6012     specified.

6013     **op** (IN) A binary operator applied to each element of input matrix, A, with the element  
6014     of the input matrix used as the left-hand argument, and the scalar value, `val`, used  
6015     as the right-hand argument.

6016     **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as  
6017     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
6018     argument in the *bind-second* variant.

6019     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
6020     argument in the *bind-first* variant, or the right-hand (second) argument in the  
6021     *bind-second* variant.

6022     **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand  
6023     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
6024     the *bind-second* variant. It must not be empty.

6025     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6026     should be specified. Non-default field/value pairs are listed as follows:  
6027

| Param | Field    | Value         | Description   |
|-------|----------|---------------|---|
| C     | GrB_OUTP | GrB_REPLACE   | Output matrix C is cleared (all elements removed) before the result is stored in it.  |
| Mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined. |
| Mask  | GrB_MASK | GrB_COMP      | Use the complement of Mask.   |
| A     | GrB_INP0 | GrB_TRAN      | Use transpose of A for the operation ( <i>bind-second</i> variant only).  |
| A     | GrB_INP1 | GrB_TRAN      | Use transpose of A for the operation ( <i>bind-first</i> variant only).   |

## Return Values

|                          |   |
|--------------------------|---|
| GrB_SUCCESS              | In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.    |
| GrB_PANIC                | Unknown internal error.   |
| GrB_INVALID_OBJECT       | This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call <code>GrB_error()</code> to access any error messages generated by the implementation.                          |
| GrB_OUT_OF_MEMORY        | Not enough memory available for the operation.  |
| GrB_UNINITIALIZED_OBJECT | One or more of the GraphBLAS objects has not been initialized by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).   |
| GrB_INDEX_OUT_OF_BOUNDS  | A value in <code>row_indices</code> is greater than or equal to <code>nrows(A)</code> , or a value in <code>col_indices</code> is greater than or equal to <code>ncols(A)</code> . In non-blocking mode, this can be reported as an execution error.                                    |
| GrB_DIMENSION_MISMATCH   | Mask and C dimensions are incompatible, <code>nrows</code> $\neq$ <code>nrows(C)</code> , or <code>ncols</code> $\neq$ <code>ncols(C)</code> .  |
| GrB_DOMAIN_MISMATCH      | The domains of the various matrices and scalar are incompatible with the corresponding domains of the binary operator or accumulation operator, or the mask's domain is not compatible with <code>bool</code> (in the case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set). |
| GrB_EMPTY_OBJECT         | The <code>GrB_Scalar s</code> used in the call is empty ( <code>nvals(s) = 0</code> ) and therefore a value cannot be passed to the binary operator.  |

## 6055 Description

6056 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a  
6057 GraphBLAS matrix each composed with a scalar constant, `val` or `s`:

6058                   bind-first:      $C = f(\text{val}, A)$  or  $C = f(s, A)$

6059                   bind-second:     $C = f(A, \text{val})$  or  $C = f(A, s)$ ,

6060 or if an optional binary accumulation operator ( $\odot$ ) is provided:

6061                   bind-first:      $C = C \odot f(\text{val}, A)$  or  $C = C \odot f(s, A)$

6062                   bind-second:     $C = C \odot f(A, \text{val})$  or  $C = C \odot f(A, s)$ .

6063 Logically, this operation occurs in three steps:

6064       **Setup** The internal matrices and mask used in the computation are formed and their domains  
6065               and dimensions are tested for compatibility.

6066       **Compute** The indicated computations are carried out.

6067       **Output** The result is written into the output matrix, possibly under control of a mask.

6068 Up to three argument matrices are used in the `GrB_apply` operation:

- 6069     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6070     2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)
- 6071     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6072 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are  
6073 tested for domain compatibility as follows:

- 6074     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
6075       must be from one of the pre-defined types of Table 3.2.
- 6076     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.
- 6077     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6078       of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
6079        $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6080     4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.
- 6081     5. If bind-first:  
6082       (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

6083 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
 6084 of the binary operator.

6085 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
 6086 binary operator.

6087 6. If `bind-second`:

6088 (a)  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

6089 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
 6090 of the binary operator.

6091 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the  
 6092 binary operator.

6093 Two domains are compatible with each other if values from one domain can be cast to values in  
 6094 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6095 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6096 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
 6097 error listed above is returned.

6098 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
 6099 are formed ( $\leftarrow$  denotes copy):

6100 1. Matrix  $\tilde{C} \leftarrow C$ .

6101 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:

6102 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
 6103  $j < \mathbf{ncols}(C)\} \rangle$ .

6104 (b) If `Mask`  $\neq$  `GrB_NULL`,

6105 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
 6106  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,

6107 ii. Otherwise,  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
 6108  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .

6109 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{M} \leftarrow \neg \tilde{M}$ .

6110 3. Matrix  $\tilde{A}$  is computed from argument `A` as follows:

6111 `bind-first`:  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? A^T : A$

6112 `bind-second`:  $\tilde{A} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? A^T : A$

6113 4. Scalar  $\tilde{s} \leftarrow s$  (`GraphBLAS` scalar case).

6114 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 6115 must hold:

6116 1.  $\mathbf{nrows}(\tilde{C}) = \mathbf{nrows}(\tilde{M})$ .

6117 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .

6118 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6119 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6120 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6121 error listed above is returned.

6122 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6123 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6124 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6125 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6126  $\mathbf{val}$  with the same domain as  $\tilde{s}$  and set  $\mathbf{val} = \mathbf{val}(\tilde{s})$ .

6127 We are now ready to carry out the apply and any additional associated operations. We describe  
6128 this in terms of two intermediate matrices:

- 6129 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the binary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 6130 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6131 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as one of the following:

6132 bind-first:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\mathbf{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$ ,

6133 bind-second:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle$ ,

6134 where  $f = \mathbf{f}(\mathbf{op})$ .

6135 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6136 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6137 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6138 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6139 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6140 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6141 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$6142 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$6143 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6146 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6147 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6148 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6149 mask which acts as a “write mask”.

- 6150 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
6151 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6152 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6153 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6154 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6155 mask are unchanged:

$$6156 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6157 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
6158 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
6159 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
6160 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6161 sequence.

#### 6162 4.3.8.5 apply: Vector index unary operator variant[Scott: NEW CONTENT]

6163 Computes the transformation of the values of the stored elements of a vector using an index unary  
6164 operator that is a function of the stored value, its location indices, and an user provided scalar  
6165 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6166 C Syntax

```
6167      GrB_Info GrB_apply(GrB_Vector          w,
6168                        const GrB_Vector     mask,
6169                        const GrB_BinaryOp    accum,
6170                        const GrB_IndexUnaryOp op,
6171                        const GrB_Vector     u,
6172                        <type>               val,
6173                        const GrB_Descriptor  desc);
```

```
6174      GrB_Info GrB_apply(GrB_Vector          w,
6175                        const GrB_Vector     mask,
6176                        const GrB_BinaryOp    accum,
6177                        const GrB_IndexUnaryOp op,
6178                        const GrB_Vector     u,
6179                        const GrB_Scalar     s,
6180                        const GrB_Descriptor  desc);
```



## Parameters

**w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the apply operation. On output, this vector holds the results of the operation.

**mask** (IN) An optional “write” mask that controls which results from this operation are stored into the output vector **w**. The mask dimensions must match those of the vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain of the **mask** vector must be of type **bool** or any of the predefined “built-in” types in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the dimensions of **w**), **GrB\_NULL** should be specified.

**accum** (IN) An optional binary operator used for accumulating entries into existing **w** entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be specified.

**op** (IN) An index unary operator,  $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\mathbf{GrB\_Index}), D_{in_2}, f_i \rangle$ , applied to each element stored in the input vector, **u**. It is a function of the stored element’s value, its location index, and a user supplied scalar value (either **s** or **val**).

**u** (IN) The GraphBLAS vector whose elements are passed to the index unary operator.

**val** (IN) An additional scalar value that is passed to the index unary operator.

**s** (IN) An additional GraphBLAS scalar that is passed to the index unary operator. It must not be empty.

**desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL** should be specified. Non-default field/value pairs are listed as follows:

| Param       | Field           | Value                | Description  |
|-------------|-----------------|----------------------|--|
| <b>w</b>    | <b>GrB_OUTP</b> | <b>GrB_REPLACE</b>   | Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.  |
| <b>mask</b> | <b>GrB_MASK</b> | <b>GrB_STRUCTURE</b> | The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined. |
| <b>mask</b> | <b>GrB_MASK</b> | <b>GrB_COMP</b>      | Use the complement of <b>mask</b> .  |

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector **w** is ready to be used in the next method of the sequence.

6212                   GrB\_PANIC   Unknown internal error.

6213           GrB\_INVALID\_OBJECT   This is returned in any execution mode whenever one of the  
6214                                   opaque GraphBLAS objects (input or output) is in an invalid  
6215                                   state caused by a previous execution error. Call GrB\_error() to  
6216                                   access any error messages generated by the implementation.

6217           GrB\_OUT\_OF\_MEMORY   Not enough memory available for operation.

6218   GrB\_UNINITIALIZED\_OBJECT   One or more of the GraphBLAS objects has not been initialized  
6219                                   by a call to new (or another constructor).

6220   GrB\_DIMENSION\_MISMATCH   mask, w and/or u dimensions are incompatible.

6221   GrB\_DOMAIN\_MISMATCH   The domains of the various vectors are incompatible with the cor-  
6222                                   responding domains of the accumulation operator or index unary  
6223                                   operator, or the mask's domain is not compatible with bool (in  
6224                                   the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

6225           GrB\_EMPTY\_OBJECT   The GrB\_Scalar s used in the call is empty ( $\mathbf{nvals}(s) = 0$ ) and  
6226                                   therefore a value cannot be passed to the index unary operator.

## 6227 Description

6228   This variant of GrB\_apply computes the result of applying an index unary operator to the elements  
6229   of a GraphBLAS vector each composed with the element's index and a scalar constant, val or s:

$$6230 \quad \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \text{ or } \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}),$$

6231   or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6232 \quad \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \text{ or } \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}).$$

6233   Logically, this operation occurs in three steps:

6234       **Setup**   The internal vectors and mask used in the computation are formed and their domains  
6235                   and dimensions are tested for compatibility.

6236       **Compute**   The indicated computations are carried out.

6237       **Output**   The result is written into the output vector, possibly under control of a mask.

6238   Up to three argument vectors are used in this GrB\_apply operation:

- 6239   1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6240   2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)

6241 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6242 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6243 are tested for domain compatibility as follows:

- 6244 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
6245 must be from one of the pre-defined types of Table 3.2.
- 6246 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the index unary operator.
- 6247 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6248 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be compatible  
6249 with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6250 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6251 5. If the non-opaque scalar `val` is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of  
6252 the index unary operator.
- 6253 6. If the `GrB_Scalar s` is provided, then  $\mathbf{D}(\mathbf{s})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the index  
6254 unary operator.

6255 Two domains are compatible with each other if values from one domain can be cast to values in  
6256 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6257 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6258 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
6259 error listed above is returned.

6260 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6261 denotes copy):

- 6262 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6263 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 6264 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 6265 (b) If `mask  $\neq$  GrB_NULL`,
    - 6266 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
    - 6267 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 6268 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6269 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6270 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6271 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6272 must hold:

6273 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6274 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

6275 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6276 error listed above is returned.

6277 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6278 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6279 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6280 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided ( $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable `val`  
6281 with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6282 We are now ready to carry out the apply and any additional associated operations. We describe  
6283 this in terms of two intermediate vectors:

- 6284 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6285  $\tilde{\mathbf{u}}$ .
- 6286 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6287 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6288 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6289 where  $f_i = \mathbf{f}(\text{op})$ .

6290 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6291 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 6292 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6293 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6294 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6295 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6296 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6297 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6298 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6299 \quad & \\ 6300 \end{aligned}$$

6301 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6302 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6303 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6304 mask which acts as a “write mask”.

- If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $w$  on input to this operation are deleted and the content of the new output vector,  $w$ , is defined as,

$$L(w) = \{(i, z_i) : i \in (\text{ind}(\tilde{z}) \cap \text{ind}(\tilde{m}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{z}$  indicated by the mask are copied into the result vector,  $w$ , and elements of  $w$  that fall outside the set indicated by the mask are unchanged:

$$L(w) = \{(i, w_i) : i \in (\text{ind}(w) \cap \text{ind}(\neg\tilde{m}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{z}) \cap \text{ind}(\tilde{m}))\}.$$

In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of vector  $w$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of vector  $w$  is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

#### 6317 4.3.8.6 apply: Matrix index unary operator variant[Scott: NEW CONTENT]

6318 Computes the transformation of the values of the stored elements of a matrix using an index unary  
6319 operator that is a function of the stored value, its location indices, and an user provided scalar  
6320 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6321 C Syntax

```
6322     GrB_Info GrB_apply(GrB_Matrix      C,
6323                       const GrB_Matrix Mask,
6324                       const GrB_BinaryOp accum,
6325                       const GrB_IndexUnaryOp op,
6326                       const GrB_Matrix A,
6327                       <type>          val,
6328                       const GrB_Descriptor desc);
```

```
6329     GrB_Info GrB_apply(GrB_Matrix      C,
6330                       const GrB_Matrix Mask,
6331                       const GrB_BinaryOp accum,
6332                       const GrB_IndexUnaryOp op,
6333                       const GrB_Matrix A,
6334                       const GrB_Scalar s,
6335                       const GrB_Descriptor desc);
```

#### 6336 Parameters

6337 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6338 that may be accumulated with the result of the apply operation. On output, the  
6339 matrix holds the results of the operation.

6340 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6341 stored into the output matrix **C**. The mask dimensions must match those of the  
6342 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6343 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6344 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6345 dimensions of **C**), **GrB\_NULL** should be specified.

6346 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6347 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6348 specified.

6349 **op** (IN) An index unary operator,  $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\mathbf{GrB\_Index}), D_{in_2}, f_i \rangle$ , applied  
6350 to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6351 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6352 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6353 ator.

6354 **val** (IN) An additional scalar value that is passed to the index unary operator.

6355 **s** (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6356 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6357 should be specified. Non-default field/value pairs are listed as follows:

| Param       | Field           | Value                | Description  |
|-------------|-----------------|----------------------|--|
| <b>C</b>    | <b>GrB_OUTP</b> | <b>GrB_REPLACE</b>   | Output matrix <b>C</b> is cleared (all elements removed) before the result is stored in it.  |
| <b>Mask</b> | <b>GrB_MASK</b> | <b>GrB_STRUCTURE</b> | The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined. |
| <b>Mask</b> | <b>GrB_MASK</b> | <b>GrB_COMP</b>      | Use the complement of <b>Mask</b> .  |
| <b>A</b>    | <b>GrB_INP0</b> | <b>GrB_TRAN</b>      | Use transpose of <b>A</b> for the operation.   |

## 6360 Return Values

6361 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
6362 blocking mode, this indicates that the compatibility tests on di-  
6363 mensions and domains for the input arguments passed successfully.  
6364 Either way, output matrix **C** is ready to be used in the next method  
6365 of the sequence.

6366 **GrB\_PANIC** Unknown internal error.

6367 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
6368 GraphBLAS objects (input or output) is in an invalid state caused

6369 by a previous execution error. Call `GrB_error()` to access any error  
 6370 messages generated by the implementation.

6371 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6372 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 6373 a call to `new` (or another constructor).

6374 **GrB\_DIMENSION\_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6375 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the  
 6376 corresponding domains of the accumulation operator or index unary  
 6377 operator, or the mask's domain is not compatible with `bool` (in the  
 6378 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6379 **GrB\_EMPTY\_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
 6380 therefore a value cannot be passed to the index unary operator.

## 6381 Description

6382 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements  
 6383 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar  
 6384 constant, `val` or `s`:

$$6385 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6386 or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6387 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6388 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
 6389 indices, respectively.

6390 Logically, this operation occurs in three steps:

6391 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6392 and dimensions are tested for compatibility.

6393 **Compute** The indicated computations are carried out.

6394 **Output** The result is written into the output matrix, possibly under control of a mask.

6395 Up to three argument matrices are used in the `GrB_apply` operation:

- 6396 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6397 2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)

6398 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6399 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6400 are tested for domain compatibility as follows:

- 6401 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{Mask})$   
6402 must be from one of the pre-defined types of Table 3.2.
- 6403 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator.
- 6404 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6405 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be compatible  
6406 with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 6407 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6408 5. If the non-opaque scalar **val** is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of  
6409 the index unary operator.
- 6410 6. If the **GrB\_Scalar** **s** is provided, then  $\mathbf{D}(\mathbf{s})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the index  
6411 unary operator.

6412 Two domains are compatible with each other if values from one domain can be cast to values in  
6413 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6414 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6415 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
6416 error listed above is returned.

6417 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation  
6418 are formed ( $\leftarrow$  denotes copy):

- 6419 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6420 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument **Mask** as follows:
  - 6421 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
6422  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 6423 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 6424 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
6425  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 6426 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
6427  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 6428 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6429 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument **A** as follows:
  - 6430  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6431 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).



6432 The internal matrices and mask are checked for dimension compatibility. The following conditions  
6433 must hold:

- 6434 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6435 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6436 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6437 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6438 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6439 error listed above is returned.

6440 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6441 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6442 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6443 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6444  $\mathbf{val}$  with the same domain as  $\tilde{s}$  and set  $\mathbf{val} = \mathbf{val}(\tilde{s})$ .

6445 We are now ready to carry out the apply and any additional associated operations. We describe  
6446 this in terms of two intermediate matrices:

- 6447 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6448  $\tilde{\mathbf{A}}$ .
- 6449 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6450 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6451 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6452 where  $f_i = \mathbf{f}(\mathbf{op})$ .

6453 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6454 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6455 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6456 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6457 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6458 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 6459 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6460 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6461 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6462 \end{aligned}$$

6464 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6465 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 6466 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 6467 mask which acts as a “write mask”.

- 6468 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 6469 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6470 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6471 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 6472 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 6473 mask are unchanged:

$$6474 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6475 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 6476 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 6477 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 6478 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 6479 sequence.

#### 6480 4.3.9 select:

6481 Apply a select operator to the stored elements of an object to determine whether or not to keep  
 6482 them.

##### 6483 4.3.9.1 select: Vector variant[Scott: NEW CONTENT]

6484 Apply a select operator (an index unary operator) to the elements of a vector.

#### 6485 C Syntax

```
6486 // scalar value variant
6487 GrB_Info GrB_select(GrB_Vector          w,
6488                    const GrB_Vector     mask,
6489                    const GrB_BinaryOp    accum,
6490                    const GrB_IndexUnaryOp op,
6491                    const GrB_Vector     u,
6492                    <type>                val,
6493                    const GrB_Descriptor  desc);
6494
6495 // GraphBLAS scalar variant
6496 GrB_Info GrB_select(GrB_Vector          w,
6497                    const GrB_Vector     mask,
```

```

6498         const GrB_BinaryOp      accum,
6499         const GrB_IndexUnaryOp  op,
6500         const GrB_Vector        u,
6501         const GrB_Scalar        s,
6502         const GrB_Descriptor    desc);
6503

```

## 6504 Parameters

6505     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6506     that may be accumulated with the result of the select operation. On output, this  
6507     vector holds the results of the operation.

6508     **mask** (IN) An optional “write” mask that controls which results from this operation are  
6509     stored into the output vector **w**. The mask dimensions must match those of the  
6510     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6511     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6512     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6513     dimensions of **w**), **GrB\_NULL** should be specified.

6514     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6515     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6516     specified.

6517     **op** (IN) An index unary operator,  $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\mathbf{GrB\_Index}), D_{in_2}, f_i \rangle$ , applied  
6518     to each element stored in the input vector, **u**. It is a function of the stored element’s  
6519     value, its location index, and a user supplied scalar value (either **s** or **val**).

6520     **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-  
6521     ator.

6522     **val** (IN) An additional scalar value that is passed to the index unary operator.

6523     **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6524     not be empty.

6525     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6526     should be specified. Non-default field/value pairs are listed as follows:

| Param       | Field           | Value                | Description  |
|-------------|-----------------|----------------------|--|
| <b>w</b>    | <b>GrB_OUTP</b> | <b>GrB_REPLACE</b>   | Output vector <b>w</b> is cleared (all elements removed) before the result is stored in it.  |
| <b>mask</b> | <b>GrB_MASK</b> | <b>GrB_STRUCTURE</b> | The write mask is constructed from the structure (pattern of stored values) of the input <b>mask</b> vector. The stored values are not examined. |
| <b>mask</b> | <b>GrB_MASK</b> | <b>GrB_COMP</b>      | Use the complement of <b>mask</b> .  |

## 6529 Return Values

|      |                                 |  |
|------|---------------------------------|--|
| 6530 | <b>GrB_SUCCESS</b>              | In blocking mode, the operation completed successfully. In non-                      |
| 6531 |                                 | blocking mode, this indicates that the compatibility tests on di-                    |
| 6532 |                                 | mensions and domains for the input arguments passed success-                         |
| 6533 |                                 | fully. Either way, output vector <b>w</b> is ready to be used in the next            |
| 6534 |                                 | method of the sequence.  |
| 6535 | <b>GrB_PANIC</b>                | Unknown internal error.  |
| 6536 | <b>GrB_INVALID_OBJECT</b>       | This is returned in any execution mode whenever one of the                           |
| 6537 |                                 | opaque GraphBLAS objects (input or output) is in an invalid                          |
| 6538 |                                 | state caused by a previous execution error. Call <b>GrB_error()</b> to               |
| 6539 |                                 | access any error messages generated by the implementation.                           |
| 6540 | <b>GrB_OUT_OF_MEMORY</b>        | Not enough memory available for operation.   |
| 6541 | <b>GrB_UNINITIALIZED_OBJECT</b> | One or more of the GraphBLAS objects has not been initialized                        |
| 6542 |                                 | by a call to one of its constructors.  |
| 6543 | <b>GrB_DIMENSION_MISMATCH</b>   | <b>mask</b> , <b>w</b> and/or <b>u</b> dimensions are incompatible.                  |
| 6544 | <b>GrB_DOMAIN_MISMATCH</b>      | The domains of the various vectors are incompatible with the cor-                    |
| 6545 |                                 | responding domains of the accumulation operator or index unary                       |
| 6546 |                                 | operator, or the mask's domain is not compatible with <b>bool</b> (in                |
| 6547 |                                 | the case where <b>desc[GrB_MASK].GrB_STRUCTURE</b> is not set).                      |
| 6548 | <b>GrB_EMPTY_OBJECT</b>         | The <b>GrB_Scalar</b> <b>s</b> used in the call is empty ( <b>nvals(s) = 0</b> ) and |
| 6549 |                                 | therefore a value cannot be passed to the index unary operator.                      |

## 6550 Description

6551 This variant of **GrB\_select** computes the result of applying a index unary operator to select the  
6552 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored  
6553 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding  
6554 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.  
6555 This acts like a functional mask on the input vector as follows:

$$6556 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6557 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6558 Correspondingly, if a **GrB\_Scalar**, **s**, is provided:

$$6559 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6560 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6561 Logically, this operation occurs in three steps:

6562     **Setup** The internal vectors and mask used in the computation are formed and their domains  
6563             and dimensions are tested for compatibility.

6564     **Compute** The indicated computations are carried out.

6565     **Output** The result is written into the output vector, possibly under control of a mask.

6566 Up to three argument vectors are used in this `GrB_select` operation:

- 6567     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$   
6568     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)  
6569     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6570 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6571 are tested for domain compatibility as follows:

- 6572     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
6573         must be from one of the pre-defined types of Table 3.2.  
6574     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .  
6575     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6576         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
6577         mulation operator.  
6578     4.  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6579         i.e., castable to `bool`.  
6580     5.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.  
6581     6.  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(\mathbf{s})$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$   
6582         of the index unary operator.

6583 Two domains are compatible with each other if values from one domain can be cast to values in  
6584 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6585 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6586 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch  
6587 error listed above is returned.

6588 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6589 denotes copy):

- 6590     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .  
6591     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

- 6592 (a) If  $\text{mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
- 6593 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,
- 6594 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
- 6595 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
- 6596 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 6597 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6598 4. Scalar  $\widetilde{s} \leftarrow s$  (GrB\_Scalar version only).

6599 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6600 must hold:

- 6601 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6602 2.  $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$ .

6603 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch  
6604 error listed above is returned.

6605 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6606 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6607 If an empty `GrB_Scalar`  $\widetilde{s}$  is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6608 If a non-empty `GrB_Scalar`,  $\widetilde{s}$ , is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 1$ ), we then create an internal variable  
6609 `val` with the same domain as  $\widetilde{s}$  and set `val = val( $\widetilde{s}$ )`.

6610 We are now ready to carry out the `select` and any additional associated operations. We describe  
6611 this in terms of two intermediate vectors:

- 6612 •  $\widetilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6613  $\widetilde{\mathbf{u}}$ .
- 6614 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6615 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$6616 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6617 where  $f_i = \mathbf{f}(\text{op})$ .

6618 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6619 • If `accum = GrB_NULL`, then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 6620 • If `accum` is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$6621 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

#### 4.3.9.2 select: Matrix variant[Scott: NEW CONTENT]

Apply a select operator (an index unary operator) to the elements of a matrix.

#### C Syntax

```
// scalar value variant
GrB_Info GrB_select(GrB_Matrix      C,
                   const GrB_Matrix  Mask,
                   const GrB_BinaryOp accum,
                   const GrB_IndexUnaryOp op,
                   const GrB_Matrix  A,
                   <type>            val,
                   const GrB_Descriptor desc);
```

```

6657 // GraphBLAS scalar variant
6658 GrB_Info GrB_select(GrB_Matrix          C,
6659                    const GrB_Matrix     Mask,
6660                    const GrB_BinaryOp    accum,
6661                    const GrB_IndexUnaryOp op,
6662                    const GrB_Matrix      A,
6663                    const GrB_Scalar      s,
6664                    const GrB_Descriptor  desc);

```

## 6665 Parameters

6666 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6667 that may be accumulated with the result of the select operation. On output, the  
6668 matrix holds the results of the operation.

6669 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6670 stored into the output matrix **C**. The mask dimensions must match those of the  
6671 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6672 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6673 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6674 dimensions of **C**), **GrB\_NULL** should be specified.

6675 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6676 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6677 specified.

6678 **op** (IN) An index unary operator,  $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\mathbf{GrB\_Index}), D_{in_2}, f_i \rangle$ , applied  
6679 to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6680 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6681 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6682 ator.

6683 **val** (IN) An additional scalar value that is passed to the index unary operator.

6684 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6685 not be empty.

6686 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6687 should be specified. Non-default field/value pairs are listed as follows:

6688



| Param | Field    | Value         | Description   |
|-------|----------|---------------|---|
| C     | GrB_OUTP | GrB_REPLACE   | Output matrix C is cleared (all elements removed) before the result is stored in it.  |
| Mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined. |
| Mask  | GrB_MASK | GrB_COMP      | Use the complement of Mask.   |
| A     | GrB_INP0 | GrB_TRAN      | Use transpose of A for the operation.   |

## Return Values

**GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output matrix C is ready to be used in the next method of the sequence.

**GrB\_PANIC** Unknown internal error.

**GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call **GrB\_error()** to access any error messages generated by the implementation.

**GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

**GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to one of its constructors.

**GrB\_DIMENSION\_MISMATCH** Mask, C and/or A dimensions are incompatible.

**GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the corresponding domains of the accumulation operator or index unary operator, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

**GrB\_EMPTY\_OBJECT** The **GrB\_Scalar** s used in the call is empty (**nvals(s) = 0**) and therefore a value cannot be passed to the index unary operator.

## Description

This variant of **GrB\_select** computes the result of applying a index unary operator to select the elements of the input GraphBLAS matrix. The operator takes, as input, the value of each stored element, along with the element's row and column indices and a scalar constant – from either **val** or **s**. The corresponding element of the input matrix is selected (kept) if the function evaluates to **true** when cast to **bool**. This acts like a functional mask on the input matrix as follows when specifying a transparent scalar value:

6718  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ , or  
6719  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ .

6720 Correspondingly, if a GrB\_Scalar,  $s$ , is provided:

6721  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ , or  
6722  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ .

6723 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
6724 indices, respectively.

6725 Logically, this operation occurs in three steps:

6726 **Setup** The internal matrices and mask used in the computation are formed and their domains  
6727 and dimensions are tested for compatibility.

6728 **Compute** The indicated computations are carried out.

6729 **Output** The result is written into the output matrix, possibly under control of a mask.

6730 Up to three argument matrices are used in the GrB\_select operation:

- 6731 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6732 2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)
- 6733 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6734 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6735 are tested for domain compatibility as follows:

- 6736 1. If **Mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
6737 must be from one of the pre-defined types of Table 3.2.
- 6738 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 6739 3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6740 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
6741 mulation operator.
- 6742 4.  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6743 i.e., castable to **bool**.
- 6744 5.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6745 6.  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
6746 of the index unary operator.

6747 Two domains are compatible with each other if values from one domain can be cast to values in  
 6748 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6749 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6750 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch  
 6751 error listed above is returned.

6752 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
 6753 are formed ( $\leftarrow$  denotes copy):

- 6754 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6755 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - 6756 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 6757  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 6758 (b) If `Mask  $\neq$  GrB_NULL`,
    - 6759 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 6760  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 6761 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 6762  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 6763 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6764 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument `A` as follows:  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6765 4. Scalar  $\tilde{s} \leftarrow s$  (`GrB_Scalar` version only).

6766 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 6767 must hold:

- 6768 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6769 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6770 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6771 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6772 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch  
 6773 error listed above is returned.

6774 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 6775 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6776 If an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
 6777 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
 6778 `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6779 We are now ready to carry out the `select` and any additional associated operations. We describe  
 6780 this in terms of two intermediate matrices:

- 6781 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6782  $\tilde{\mathbf{A}}$ .
- 6783 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6784 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6785 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6786 where  $f_i = \mathbf{f}(\text{op})$ .

6787 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6788 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6789 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6790 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6791 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6792 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6793 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6794 \\ 6795 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6796 \\ 6797 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

6798 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6799 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6800 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6801 mask which acts as a “write mask”.

- 6802 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
6803 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6804 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6805 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6806 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6807 mask are unchanged:

$$6808 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6809 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
6810 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
6811 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
6812 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6813 sequence.

#### 6814 4.3.10 reduce: Perform a reduction across the elements of an object

6815 Computes the reduction of the values of the elements of a vector or matrix.

##### 6816 4.3.10.1 reduce: Standard matrix to vector variant

6817 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns  
6818 is desired, the input matrix should be transposed using the descriptor.

#### 6819 C Syntax

```
6820     GrB_Info GrB_reduce(GrB_Vector      w,  
6821                        const GrB_Vector mask,  
6822                        const GrB_BinaryOp accum,  
6823                        const GrB_Monoid op,  
6824                        const GrB_Matrix A,  
6825                        const GrB_Descriptor desc);  
6826  
6827     GrB_Info GrB_reduce(GrB_Vector      w,  
6828                        const GrB_Vector mask,  
6829                        const GrB_BinaryOp accum,  
6830                        const GrB_BinaryOp op,  
6831                        const GrB_Matrix A,  
6832                        const GrB_Descriptor desc);
```

#### 6833 Parameters

6834 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6835 that may be accumulated with the result of the reduction operation. On output,  
6836 this vector holds the results of the operation.

6837 **mask** (IN) An optional “write” mask that controls which results from this operation are  
6838 stored into the output vector **w**. The mask dimensions must match those of the  
6839 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6840 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6841 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6842 dimensions of **w**), **GrB\_NULL** should be specified.

6843 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6844 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6845 specified.

6846 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.  
6847 Depending on which type is passed, the following defines the binary operator with  
6848 one domain,  $F_b = \langle D, D, D, \oplus \rangle$ , that is used:

6849 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
6850 Monoid:  $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$ , the identity element of the  
6851 monoid is ignored.

6852 If  $\text{op}$  is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in  
6853 both cases  $\odot(\text{op})$  must be commutative and associative. Otherwise, the outcome  
6854 of the operation is undefined.

6855  $\mathbf{A}$  (IN) The GraphBLAS matrix on which reduction will be performed.

6856 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6857 should be specified. Non-default field/value pairs are listed as follows:  
6858

| Param | Field    | Value         | Description   |
|-------|----------|---------------|---|
| w     | GrB_OUTP | GrB_REPLACE   | Output vector w is cleared (all elements removed) before the result is stored in it.  |
| mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined. |
| mask  | GrB_MASK | GrB_COMP      | Use the complement of mask.   |
| A     | GrB_INP0 | GrB_TRAN      | Use transpose of A for the operation.   |

## 6860 Return Values

6861 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
6862 blocking mode, this indicates that the compatibility tests on di-  
6863 mensions and domains for the input arguments passed successfully.  
6864 Either way, output vector w is ready to be used in the next method  
6865 of the sequence.

6866 GrB\_PANIC Unknown internal error.

6867 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
6868 GraphBLAS objects (input or output) is in an invalid state caused  
6869 by a previous execution error. Call `GrB_error()` to access any error  
6870 messages generated by the implementation.

6871 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

6872 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
6873 a call to `new` (or `dup` for vector parameters).

6874 GrB\_DIMENSION\_MISMATCH mask, w and/or u dimensions are incompatible.

6875 GrB\_DOMAIN\_MISMATCH Either the domains of the various vectors and matrices are incom-  
6876 patible with the corresponding domains of the accumulation oper-  
6877 ator or reduce function, or the domains of the GraphBLAS binary

operator `op` are not all the same, or the mask's domain is not compatible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 6881 Description

6882 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows  
 6883 of an input matrix:  $w(i) = \bigoplus A(i, :) \forall i$ ; or, if an optional binary accumulation operator is provided,  
 6884  $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$ , where  $\bigoplus = \odot(F_b)$  and  $\odot = \odot(\text{accum})$ .

6885 Logically, this operation occurs in three steps:

6886     **Setup** The internal vector, matrix and mask used in the computation are formed and their  
 6887 domains and dimensions are tested for compatibility.

6888 **Compute** The indicated computations are carried out.

6889 **Output** The result is written into the output vector, possibly under control of a mask.

6890 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6891 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6892 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 6893 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6894 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested  
 6895 for domain compatibility as follows:

- 6896 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 6897 must be from one of the pre-defined types of Table 3.2.
- 6898 2.  $\mathbf{D}(w)$  must be compatible with the domain of the reduction binary operator,  $\mathbf{D}(F_b)$ .
- 6899 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 6900 of the accumulation operator and  $\mathbf{D}(F_b)$ , must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 6901 mulation operator.
- 6902 4.  $\mathbf{D}(A)$  must be compatible with the domain of the binary reduction operator,  $\mathbf{D}(F_b)$ .

6903 Two domains are compatible with each other if values from one domain can be cast to values in  
 6904 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6905 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6906 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
 6907 error listed above is returned.

6908 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 6909 denotes copy):

- 6910 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6911 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 6912 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 6913 (b) If `mask  $\neq$  GrB_NULL`,
- 6914 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 6915 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 6916 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6917 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

6918 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6919 must hold:

- 6920 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6921 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6922 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-  
6923 match error listed above is returned.

6924 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6925 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6926 We carry out the reduce and any additional associated operations. We describe this in terms of  
6927 two intermediate vectors:

- 6928 •  $\tilde{\mathbf{t}}$ : The vector holding the result from reducing along the rows of input matrix  $\tilde{\mathbf{A}}$ .
- 6929 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6930 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6931 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6932 The value of each of its elements is computed by

$$6933 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6934 where  $\bigoplus = \odot(F_b)$ .

6935 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6936 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .



- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ , using what is called a *standard vector mask and replace*. This is carried out under control of the mask which acts as a “write mask”.

- If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but may not be fully computed. However, it can be used in the next GraphBLAS method call in a sequence.

#### 4.3.10.2 reduce: Vector-scalar variant[Scott: NEW CONTENT]

Reduce all stored values into a single scalar.

### C Syntax

```
// scalar value + monoid (only)
GrB_Info GrB_reduce(<type>          *val,
                    const GrB_BinaryOp accum,
                    const GrB_Monoid  op,
                    const GrB_Vector  u,
```

```

6970             const GrB_Descriptor desc);
6971
6972 // GraphBLAS Scalar + monoid
6973 GrB_Info GrB_reduce(GrB_Scalar      s,
6974                   const GrB_BinaryOp accum,
6975                   const GrB_Monoid   op,
6976                   const GrB_Vector   u,
6977                   const GrB_Descriptor desc);
6978
6979 // GraphBLAS Scalar + binary operator
6980 GrB_Info GrB_reduce(GrB_Scalar      s,
6981                   const GrB_BinaryOp accum,
6982                   const GrB_BinaryOp op,
6983                   const GrB_Vector   u,
6984                   const GrB_Descriptor desc);

```

## 6985 Parameters

6986 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
6987 a value that may be accumulated (optionally) with the result of the reduction  
6988 operation. On output, this scalar holds the results of the operation.

6989 **accum** (IN) An optional binary operator used for accumulating entries into an exist-  
6990 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,  
6991 **GrB\_NULL** should be specified.

6992 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
6993 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
6994 otherwise, the outcome of the operation is undefined.

6995 **u** (IN) The GraphBLAS vector on which reduction will be performed.

6996 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6997 should be specified. Non-default field/value pairs are listed as follows:

| 6999 Param | Field | Value | Description |
|------------|-------|-------|-------------|
|------------|-------|-------|-------------|

7000 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7001 tions. There are currently no non-default field/value pairs that can be set for this  
7002 operation.

## 7003 Return Values

7004 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
7005 cessfully, and the output scalar (**s** or **val**) is ready to be used in the  
7006 next method of the sequence.

|      |                                 |  |
|------|---------------------------------|--|
| 7007 | <b>GrB_PANIC</b>                | Unknown internal error.  |
| 7008 | <b>GrB_INVALID_OBJECT</b>       | This is returned in any execution mode whenever one of the opaque          |
| 7009 |                                 | GraphBLAS objects (input or output) is in an invalid state caused          |
| 7010 |                                 | by a previous execution error. Call <b>GrB_error()</b> to access any error |
| 7011 |                                 | messages generated by the implementation.                                  |
| 7012 | <b>GrB_OUT_OF_MEMORY</b>        | Not enough memory available for the operation.                             |
| 7013 | <b>GrB_UNINITIALIZED_OBJECT</b> | One or more of the GraphBLAS objects has not been initialized by           |
| 7014 |                                 | a call to a respective constructor.  |
| 7015 | <b>GrB_NULL_POINTER</b>         | val pointer is NULL.   |
| 7016 | <b>GrB_DOMAIN_MISMATCH</b>      | The domains of input and output arguments are incompatible with            |
| 7017 |                                 | the corresponding domains of the accumulation operator, or reduce          |
| 7018 |                                 | operator.  |

## 7019 Description

This variant of **GrB\_reduce** computes the result of performing a reduction across all of the stored elements of an input vector storing the result into either **s** or **val**. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7020 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7021 Logically, this operation occurs in three steps:

7022 **Setup** The internal vector used in the computation is formed and its domain is tested for  
7023 compatibility.

7024 **Compute** The indicated computations are carried out.

7025 **Output** The result is written into the output scalar.

7026 One vector argument is used in this **GrB\_reduce** operation:

- 7027 1.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7028 The output scalar, argument vector, reduction operator and accumulation operator (if provided)  
7029 are tested for domain compatibility as follows:

- 7030 1. If **accum** is **GrB\_NULL**, then  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\mathbf{s})$  must be compatible with  $\mathbf{D}(\text{op})$  from  $M$  (or with  
7031  $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

- 7032 2. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\text{s})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  
7033  $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
7034 be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 7035 3.  $\mathbf{D}(\text{u})$  must be compatible with  $\mathbf{D}(\text{op})$  from  $M$  (or with  $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

7036 Two domains are compatible with each other if values from one domain can be cast to values in  
7037 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
7038 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
7039 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
7040 error listed above is returned.

7041 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one  
7042 of the following occurs depending on the output variant:

$$7043 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7044 or

$$7045 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7046 where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7047 For all other cases, the internal vector and scalar used in the computation is formed ( $\leftarrow$  denotes  
7048 copy):

- 7049 1. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 7050 2. Scalar  $\tilde{s} \leftarrow \text{s}$  (GraphBLAS scalar case).

7051 We are now ready to carry out the reduction and any additional associated operations. An inter-  
7052 mediate scalar result  $t$  is computed as follows:

$$7053 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

7054 where  $\oplus = \odot(\text{op})$ .

7055 The final reduction value is computed as follows:

$$7056 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7057 or

$$7058 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB\_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7059 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
7060 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7061 4.3.10.3 reduce: Matrix-scalar variant[Scott: NEW CONTENT]

7062 Reduce all stored values into a single scalar.

### 7063 C Syntax

```
7064 // scalar value + monoid (only)
7065 GrB_Info GrB_reduce(<type>          *val,
7066                    const GrB_BinaryOp accum,
7067                    const GrB_Monoid  op,
7068                    const GrB_Matrix  A,
7069                    const GrB_Descriptor desc);
7070
7071 // GraphBLAS Scalar + monoid
7072 GrB_Info GrB_reduce(GrB_Scalar      s,
7073                    const GrB_BinaryOp accum,
7074                    const GrB_Monoid  op,
7075                    const GrB_Matrix  A,
7076                    const GrB_Descriptor desc);
7077
7078 // GraphBLAS Scalar + binary operator
7079 GrB_Info GrB_reduce(GrB_Scalar      s,
7080                    const GrB_BinaryOp accum,
7081                    const GrB_BinaryOp op,
7082                    const GrB_Matrix  A,
7083                    const GrB_Descriptor desc);
```

### 7084 Parameters

7085 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7086 a value that may be accumulated (optionally) with the result of the reduction  
7087 operation. On output, this scalar holds the results of the operation.

7088 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or  
7089 **val**) value. If assignment rather than accumulation is desired, GrB\_NULL should  
7090 be specified.

7091 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7092 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7093 otherwise, the outcome of the operation is undefined.

7094 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.

7095 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7096 should be specified. Non-default field/value pairs are listed as follows:

7097

7098 

| Param | Field | Value | Description |
|-------|-------|-------|-------------|
|-------|-------|-------|-------------|

7099 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7100 tions. There are currently no non-default field/value pairs that can be set for this  
7101 operation.

## 7102 Return Values

7103 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
7104 cessfully, and the output scalar (s or val) is ready to be used in the  
7105 next method of the sequence.

7106 GrB\_PANIC Unknown internal error.

7107 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7108 GraphBLAS objects (input or output) is in an invalid state caused  
7109 by a previous execution error. Call GrB\_error() to access any error  
7110 messages generated by the implementation.

7111 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7112 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7113 a call to a respective constructor.

7114 GrB\_NULL\_POINTER val pointer is NULL.

7115 GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
7116 the corresponding domains of the accumulation operator, or reduce  
7117 operator.

## 7118 Description

This variant of GrB\_reduce computes the result of performing a reduction across all of the stored elements of an input matrix storing the result into either s or val. This corresponds to (shown here for the scalar value case only):

$$\text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7119 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7120 Logically, this operation occurs in three steps:

7121       **Setup** The internal matrix used in the computation is formed and its domain is tested for  
 7122       compatibility.

7123       **Compute** The indicated computations are carried out.

7124       **Output** The result is written into the output scalar.

7125   One matrix argument is used in this GrB\_reduce operation:

7126       1.  $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7127   The output scalar, argument matrix, reduction operator and accumulation operator (if provided)  
 7128   are tested for domain compatibility as follows:

7129       1. If accum is GrB\_NULL, then  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\text{s})$  must be compatible with  $\mathbf{D}(\text{op})$  from  $M$  (or with  
 7130        $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

7131       2. If accum is not GrB\_NULL, then  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\text{s})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  
 7132        $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
 7133       be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

7134       3.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}(\text{op})$  from  $M$  (or with  $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

7135   Two domains are compatible with each other if values from one domain can be cast to values in  
 7136   the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7137   compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7138   any compatibility rule above is violated, execution of GrB\_reduce ends and the domain mismatch  
 7139   error listed above is returned.

7140   The number of values stored in the input,  $A$ , is checked. If there are no stored values in  $A$ , then  
 7141   one of the following occurs depending on the output variant:

$$7142 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7143   or

$$7144 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7145   where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with GrB\_SUCCESS.

7146   For all other cases, the internal matrix and scalar used in the computation is formed ( $\leftarrow$  denotes  
 7147   copy):

7148       1. Matrix  $\tilde{A} \leftarrow A$ .

7149       2. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

7150 We are now ready to carry out the reduce and any additional associated operations. An intermediate  
 7151 scalar result  $t$  is computed as follows:

$$7152 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7153 where  $\oplus = \odot(\text{op})$ .

7154 The final reduction value is computed as follows:

$$7155 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7156 or

$$7157 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB\_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7158 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
 7159 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7160 4.3.11 transpose: Transpose rows and columns of a matrix

7161 This version computes a new matrix that is the transpose of the source matrix.

#### 7162 C Syntax

```
7163      GrB_Info GrB_transpose(GrB_Matrix      C,
7164                           const GrB_Matrix Mask,
7165                           const GrB_BinaryOp accum,
7166                           const GrB_Matrix A,
7167                           const GrB_Descriptor desc);
```

#### 7168 Parameters

7169 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 7170 that may be accumulated with the result of the transpose operation. On output,  
 7171 the matrix holds the results of the operation.

7172 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 7173 stored into the output matrix C. The mask dimensions must match those of the  
 7174 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
 7175 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
 7176 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
 7177 dimensions of C), GrB\_NULL should be specified.



7178       **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
7179           entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
7180           specified.

7181       **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7182       **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7183           should be specified. Non-default field/value pairs are listed as follows:  
7184

| Param       | Field           | Value                | Description  |
|-------------|-----------------|----------------------|--|
| <b>C</b>    | <b>GrB_OUTP</b> | <b>GrB_REPLACE</b>   | Output matrix <b>C</b> is cleared (all elements removed) before the result is stored in it.  |
| <b>Mask</b> | <b>GrB_MASK</b> | <b>GrB_STRUCTURE</b> | The write mask is constructed from the structure (pattern of stored values) of the input <b>Mask</b> matrix. The stored values are not examined. |
| <b>Mask</b> | <b>GrB_MASK</b> | <b>GrB_COMP</b>      | Use the complement of <b>Mask</b> .  |
| <b>A</b>    | <b>GrB_INP0</b> | <b>GrB_TRAN</b>      | Use transpose of <b>A</b> for the operation.   |

## 7186   **Return Values**

7187       **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
7188           blocking mode, this indicates that the compatibility tests on di-  
7189           mensions and domains for the input arguments passed successfully.  
7190           Either way, output matrix **C** is ready to be used in the next method  
7191           of the sequence.

7192       **GrB\_PANIC** Unknown internal error.

7193       **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
7194           GraphBLAS objects (input or output) is in an invalid state caused  
7195           by a previous execution error. Call **GrB\_error()** to access any error  
7196           messages generated by the implementation.

7197       **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

7198       **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
7199           a call to **new** (or **Matrix\_dup** for matrix parameters).

7200       **GrB\_DIMENSION\_MISMATCH** **mask**, **C** and/or **A** dimensions are incompatible.

7201       **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the cor-  
7202           responding domains of the accumulation operator, or the mask's do-  
7203           main is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCT**  
7204           is not set).

7205 **Description**

7206 GrB\_transpose computes the result of performing a transpose of the input matrix:  $C = A^T$ ; or, if an  
 7207 optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A^T$ . We note that the input matrix  
 7208 A can itself be optionally transposed before the operation, which would cause either an assignment  
 7209 from A to C or an accumulation of A into C.

7210 Logically, this operation occurs in three steps:

7211     **Setup** The internal matrix and mask used in the computation are formed and their domains  
 7212             and dimensions are tested for compatibility.

7213     **Compute** The indicated computations are carried out.

7214     **Output** The result is written into the output matrix, possibly under control of a mask.

7215 Up to three matrix arguments are used in this GrB\_transpose operation:

- 7216     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7217     2.  $\text{Mask} = \langle \mathbf{D}(\text{Mask}), \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \mathbf{L}(\text{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)
- 7218     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7219 The argument matrices and accumulation operator (if provided) are tested for domain compatibility  
 7220 as follows:

- 7221     1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 7222         must be from one of the pre-defined types of Table 3.2.
- 7223     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$  of the input matrix.
- 7224     3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 7225         of the accumulation operator and  $\mathbf{D}(A)$  of the input matrix must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 7226         of the accumulation operator.

7227 Two domains are compatible with each other if values from one domain can be cast to values in  
 7228 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7229 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7230 any compatibility rule above is violated, execution of GrB\_transpose ends and the domain mismatch  
 7231 error listed above is returned.

7232 From the argument matrices, the internal matrices and mask used in the computation are formed  
 7233 ( $\leftarrow$  denotes copy):

- 7234     1. Matrix  $\tilde{C} \leftarrow C$ .
- 7235     2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument Mask as follows:

- 7236 (a) If  $\text{Mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
7237  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 7238 (b) If  $\text{Mask} \neq \text{GrB\_NULL}$ ,
- 7239 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
7240  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
- 7241 ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
7242  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
- 7243 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$ .
- 7244 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

7245 The internal matrices and masks are checked for dimension compatibility. The following conditions  
7246 must hold:

- 7247 1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$ .
- 7248 2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$ .
- 7249 3.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$ .
- 7250 4.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$ .

7251 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension  
7252 mismatch error listed above is returned.

7253 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
7254 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7255 We are now ready to carry out the matrix transposition and any additional associated operations.  
7256 We describe this in terms of two intermediate matrices:

- 7257 •  $\widetilde{\mathbf{T}}$ : The matrix holding the transpose of  $\widetilde{\mathbf{A}}$ .
- 7258 •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

7259 The intermediate matrix

$$7260 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7261 is created.

7262 The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7263 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 7264 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$7265 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7266 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 7267 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned}
 7268 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 7269 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7270 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7271 \quad & \\
 7272 \quad &
 \end{aligned}$$

7273 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7274 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 7275 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 7276 mask which acts as a “write mask”.

- 7277 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 7278 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7279 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7280 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 7281 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 7282 mask are unchanged:

$$7283 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7284 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 7285 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 7286 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 7287 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 7288 sequence.

#### 7289 4.3.12 kronecker: Kronecker product of two matrices

7290 Computes the Kronecker product of two matrices. The result is a matrix.

#### 7291 C Syntax

```

7292      GrB_Info GrB_kronecker(GrB_Matrix      C,
7293                           const GrB_Matrix Mask,
7294                           const GrB_BinaryOp accum,
7295                           const GrB_Semiring op,
7296                           const GrB_Matrix A,
7297                           const GrB_Matrix B,
7298                           const GrB_Descriptor desc);
7299
  
```

```

7300     GrB_Info GrB_kronecker(GrB_Matrix      C,
7301                           const GrB_Matrix  Mask,
7302                           const GrB_BinaryOp accum,
7303                           const GrB_Monoid   op,
7304                           const GrB_Matrix  A,
7305                           const GrB_Matrix  B,
7306                           const GrB_Descriptor desc);
7307
7308     GrB_Info GrB_kronecker(GrB_Matrix      C,
7309                           const GrB_Matrix  Mask,
7310                           const GrB_BinaryOp accum,
7311                           const GrB_BinaryOp op,
7312                           const GrB_Matrix  A,
7313                           const GrB_Matrix  B,
7314                           const GrB_Descriptor desc);

```

## 7315 Parameters

7316 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
7317 that may be accumulated with the result of the Kronecker product. On output,  
7318 the matrix holds the results of the operation.

7319 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
7320 stored into the output matrix C. The mask dimensions must match those of the  
7321 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
7322 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
7323 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
7324 dimensions of C), GrB\_NULL should be specified.

7325 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7326 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
7327 specified.

7328 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
7329 operation. Depending on which type is passed, the following defines the binary  
7330 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

7331 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

7332 Monoid:  $F_b = \langle \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \mathbf{D}(\text{op}), \odot(\text{op}) \rangle$ ; the identity element is ig-  
7333 nored.

7334 Semiring:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes(\text{op}) \rangle$ ; the additive monoid  
7335 is ignored.

7336 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
7337 product.

7338 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
7339 product.

7340 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7341 should be specified. Non-default field/value pairs are listed as follows:  
7342

| Param | Field    | Value         | Description   |
|-------|----------|---------------|---|
| C     | GrB_OUTP | GrB_REPLACE   | Output matrix C is cleared (all elements removed) before the result is stored in it.  |
| Mask  | GrB_MASK | GrB_STRUCTURE | The write mask is constructed from the structure (pattern of stored values) of the input Mask matrix. The stored values are not examined. |
| Mask  | GrB_MASK | GrB_COMP      | Use the complement of Mask.   |
| A     | GrB_INP0 | GrB_TRAN      | Use transpose of A for the operation.   |
| B     | GrB_INP1 | GrB_TRAN      | Use transpose of B for the operation.   |

## 7344 Return Values

7345 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
7346 blocking mode, this indicates that the compatibility tests on di-  
7347 mensions and domains for the input arguments passed successfully.  
7348 Either way, output matrix C is ready to be used in the next method  
7349 of the sequence.

7350 GrB\_PANIC Unknown internal error.

7351 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7352 GraphBLAS objects (input or output) is in an invalid state caused  
7353 by a previous execution error. Call GrB\_error() to access any error  
7354 messages generated by the implementation.

7355 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7356 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7357 a call to new (or Matrix\_dup for matrix parameters).

7358 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

7359 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
7360 corresponding domains of the binary operator (op) or accumulation  
7361 operator, or the mask's domain is not compatible with bool (in the  
7362 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 7363 Description

7364 GrB\_kronecker computes the Kronecker product  $C = A \otimes B$  or, if an optional binary accumulation  
7365 operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$  (where matrices A and B can be optionally transposed).

7366 The Kronecker product is defined as follows:

7367

$$7368 \quad \mathbf{C} = \mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{0,0} \otimes \mathbf{B} & A_{0,1} \otimes \mathbf{B} & \dots & A_{0,n_A-1} \otimes \mathbf{B} \\ A_{1,0} \otimes \mathbf{B} & A_{1,1} \otimes \mathbf{B} & \dots & A_{1,n_A-1} \otimes \mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes \mathbf{B} & A_{m_A-1,1} \otimes \mathbf{B} & \dots & A_{m_A-1,n_A-1} \otimes \mathbf{B} \end{bmatrix}$$

7369 where  $\mathbf{A} : \mathbb{S}^{m_A \times n_A}$ ,  $\mathbf{B} : \mathbb{S}^{m_B \times n_B}$ , and  $\mathbf{C} : \mathbb{S}^{m_A m_B \times n_A n_B}$ . More explicitly, the elements of the  
7370 Kronecker product are defined as

$$7371 \quad \mathbf{C}(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7372 where  $\otimes$  is the multiplicative operator specified by the `op` parameter.

7373 Logically, this operation occurs in three steps:

7374 **Setup** The internal matrices and mask used in the computation are formed and their domains  
7375 and dimensions are tested for compatibility.

7376 **Compute** The indicated computations are carried out.

7377 **Output** The result is written into the output matrix, possibly under control of a mask.

7378 Up to four argument matrices are used in the `GrB_kronecker` operation:

- 7379 1.  $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 7380 2.  $\mathbf{Mask} = \langle \mathbf{D}(\mathbf{Mask}), \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \mathbf{L}(\mathbf{Mask}) = \{(i, j, M_{ij})\} \rangle$  (optional)
- 7381 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 7382 4.  $\mathbf{B} = \langle \mathbf{D}(\mathbf{B}), \mathbf{nrows}(\mathbf{B}), \mathbf{ncols}(\mathbf{B}), \mathbf{L}(\mathbf{B}) = \{(i, j, B_{ij})\} \rangle$

7383 The argument matrices, the "product" operator (`op`), and the accumulation operator (if provided)  
7384 are tested for domain compatibility as follows:

- 7385 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{Mask})$   
7386 must be from one of the pre-defined types of Table 3.2.
- 7387 2.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 7388 3.  $\mathbf{D}(\mathbf{B})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 7389 4.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 7390 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
7391 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
7392 the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of `GrB_kronecker` ends and the domain mismatch error listed above is returned.

From the argument matrices, the internal matrices and mask used in the computation are formed ( $\leftarrow$  denotes copy):

1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
  - (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - (b) If `Mask  $\neq$  GrB_NULL`,
    - i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
4. Matrix  $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

The internal matrices and masks are checked for dimension compatibility. The following conditions must hold:

1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) \cdot \mathbf{nrows}(\tilde{\mathbf{B}})$ .
4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) \cdot \mathbf{ncols}(\tilde{\mathbf{B}})$ .

If any compatibility rule above is violated, execution of `GrB_kronecker` ends and the dimension mismatch error listed above is returned.

From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

We are now ready to carry out the Kronecker product and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\tilde{\mathbf{T}}$ : The matrix holding the Kronecker product of matrices  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.



7426 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$   
 7427  $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$  is created. The value of  
 7428 each of its elements is computed by

$$7429 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7430 where  $\otimes$  is the multiplicative operator specified by the `op` parameter.

7431 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7432 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 7433 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$7434 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7435 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 7436 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$7437 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7438 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7440 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7442 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7443 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 7444 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 7445 mask which acts as a “write mask”.

- 7446 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
 7447 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7448 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7449 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 7450 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 7451 mask are unchanged:

$$7452 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7453 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
 7454 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
 7455 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
 7456 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 7457 sequence. s



## Chapter 5

# Nonpolymorphic interface[Scott: NEW CONTENT]

Each polymorphic GraphBLAS method (those with multiple parameter signatures under the same name) has a corresponding set of long-name forms that are specific to each parameter signature. That is show in Tables 5.1 through 5.11.

Table 5.1: Long-name, nonpolymorphic form of GraphBLAS methods.

| Polymorphic signature                    | Nonpolymorphic signature                                 |
|--|--|
| GrB_Monoid_new(GrB_Monoid*,...,bool)     | GrB_Monoid_new_BOOL(GrB_Monoid*,GrB_BinaryOp,bool)       |
| GrB_Monoid_new(GrB_Monoid*,...,int8_t)   | GrB_Monoid_new_INT8(GrB_Monoid*,GrB_BinaryOp,int8_t)     |
| GrB_Monoid_new(GrB_Monoid*,...,uint8_t)  | GrB_Monoid_new_UINT8(GrB_Monoid*,GrB_BinaryOp,uint8_t)   |
| GrB_Monoid_new(GrB_Monoid*,...,int16_t)  | GrB_Monoid_new_INT16(GrB_Monoid*,GrB_BinaryOp,int16_t)   |
| GrB_Monoid_new(GrB_Monoid*,...,uint16_t) | GrB_Monoid_new_UINT16(GrB_Monoid*,GrB_BinaryOp,uint16_t) |
| GrB_Monoid_new(GrB_Monoid*,...,int32_t)  | GrB_Monoid_new_INT32(GrB_Monoid*,GrB_BinaryOp,int32_t)   |
| GrB_Monoid_new(GrB_Monoid*,...,uint32_t) | GrB_Monoid_new_UINT32(GrB_Monoid*,GrB_BinaryOp,uint32_t) |
| GrB_Monoid_new(GrB_Monoid*,...,int64_t)  | GrB_Monoid_new_INT64(GrB_Monoid*,GrB_BinaryOp,int64_t)   |
| GrB_Monoid_new(GrB_Monoid*,...,uint64_t) | GrB_Monoid_new_UINT64(GrB_Monoid*,GrB_BinaryOp,uint64_t) |
| GrB_Monoid_new(GrB_Monoid*,...,float)    | GrB_Monoid_new_FP32(GrB_Monoid*,GrB_BinaryOp,float)      |
| GrB_Monoid_new(GrB_Monoid*,...,double)   | GrB_Monoid_new_FP64(GrB_Monoid*,GrB_BinaryOp,double)     |
| GrB_Monoid_new(GrB_Monoid*,...,other)    | GrB_Monoid_new_UDT(GrB_Monoid*,GrB_BinaryOp,void*)       |

Table 5.2: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

| Polymorphic signature                          | Nonpolymorphic signature                        |
|--|---|
| GrB_Scalar_setElement(..., bool,...)           | GrB_Scalar_setElement_BOOL(..., bool,...)       |
| GrB_Scalar_setElement(..., int8_t,...)         | GrB_Scalar_setElement_INT8(..., int8_t,...)     |
| GrB_Scalar_setElement(..., uint8_t,...)        | GrB_Scalar_setElement_UINT8(..., uint8_t,...)   |
| GrB_Scalar_setElement(..., int16_t,...)        | GrB_Scalar_setElement_INT16(..., int16_t,...)   |
| GrB_Scalar_setElement(..., uint16_t,...)       | GrB_Scalar_setElement_UINT16(..., uint16_t,...) |
| GrB_Scalar_setElement(..., int32_t,...)        | GrB_Scalar_setElement_INT32(..., int32_t,...)   |
| GrB_Scalar_setElement(..., uint32_t,...)       | GrB_Scalar_setElement_UINT32(..., uint32_t,...) |
| GrB_Scalar_setElement(..., int64_t,...)        | GrB_Scalar_setElement_INT64(..., int64_t,...)   |
| GrB_Scalar_setElement(..., uint64_t,...)       | GrB_Scalar_setElement_UINT64(..., uint64_t,...) |
| GrB_Scalar_setElement(..., float,...)          | GrB_Scalar_setElement_FP32(..., float,...)      |
| GrB_Scalar_setElement(..., double,...)         | GrB_Scalar_setElement_FP64(..., double,...)     |
| GrB_Scalar_setElement(..., <i>other</i> ,...)  | GrB_Scalar_setElement_UDT(..., const void*,...) |
| GrB_Scalar_extractElement(bool*,...)           | GrB_Scalar_extractElement_BOOL(bool*,...)       |
| GrB_Scalar_extractElement(int8_t*,...)         | GrB_Scalar_extractElement_INT8(int8_t*,...)     |
| GrB_Scalar_extractElement(uint8_t*,...)        | GrB_Scalar_extractElement_UINT8(uint8_t*,...)   |
| GrB_Scalar_extractElement(int16_t*,...)        | GrB_Scalar_extractElement_INT16(int16_t*,...)   |
| GrB_Scalar_extractElement(uint16_t*,...)       | GrB_Scalar_extractElement_UINT16(uint16_t*,...) |
| GrB_Scalar_extractElement(int32_t*,...)        | GrB_Scalar_extractElement_INT32(int32_t*,...)   |
| GrB_Scalar_extractElement(uint32_t*,...)       | GrB_Scalar_extractElement_UINT32(uint32_t*,...) |
| GrB_Scalar_extractElement(int64_t*,...)        | GrB_Scalar_extractElement_INT64(int64_t*,...)   |
| GrB_Scalar_extractElement(uint64_t*,...)       | GrB_Scalar_extractElement_UINT64(uint64_t*,...) |
| GrB_Scalar_extractElement(float*,...)          | GrB_Scalar_extractElement_FP32(float*,...)      |
| GrB_Scalar_extractElement(double*,...)         | GrB_Scalar_extractElement_FP64(double*,...)     |
| GrB_Scalar_extractElement( <i>other</i> *,...) | GrB_Scalar_extractElement_UDT(void*,...)        |

Table 5.3: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

| Polymorphic signature                             | Nonpolymorphic signature                               |
|---|--|
| GrB_Vector_build(...,const bool*,...)             | GrB_Vector_build_BOOL(...,const bool*,...)             |
| GrB_Vector_build(...,const int8_t*,...)           | GrB_Vector_build_INT8(...,const int8_t*,...)           |
| GrB_Vector_build(...,const uint8_t*,...)          | GrB_Vector_build_UINT8(...,const uint8_t*,...)         |
| GrB_Vector_build(...,const int16_t*,...)          | GrB_Vector_build_INT16(...,const int16_t*,...)         |
| GrB_Vector_build(...,const uint16_t*,...)         | GrB_Vector_build_UINT16(...,const uint16_t*,...)       |
| GrB_Vector_build(...,const int32_t*,...)          | GrB_Vector_build_INT32(...,const int32_t*,...)         |
| GrB_Vector_build(...,const uint32_t*,...)         | GrB_Vector_build_UINT32(...,const uint32_t*,...)       |
| GrB_Vector_build(...,const int64_t*,...)          | GrB_Vector_build_INT64(...,const int64_t*,...)         |
| GrB_Vector_build(...,const uint64_t*,...)         | GrB_Vector_build_UINT64(...,const uint64_t*,...)       |
| GrB_Vector_build(...,const float*,...)            | GrB_Vector_build_FP32(...,const float*,...)            |
| GrB_Vector_build(...,const double*,...)           | GrB_Vector_build_FP64(...,const double*,...)           |
| GrB_Vector_build(...,const <i>other</i> *,...)    | GrB_Vector_build_UDT(...,const void*,...)              |
| GrB_Vector_setElement(...,GrB_Scalar,...)         | GrB_Vector_setElement_Scalar(...,const GrB_Scalar,...) |
| GrB_Vector_setElement(...,bool,...)               | GrB_Vector_setElement_BOOL(..., bool,...)              |
| GrB_Vector_setElement(...,int8_t,...)             | GrB_Vector_setElement_INT8(..., int8_t,...)            |
| GrB_Vector_setElement(...,uint8_t,...)            | GrB_Vector_setElement_UINT8(..., uint8_t,...)          |
| GrB_Vector_setElement(...,int16_t,...)            | GrB_Vector_setElement_INT16(..., int16_t,...)          |
| GrB_Vector_setElement(...,uint16_t,...)           | GrB_Vector_setElement_UINT16(..., uint16_t,...)        |
| GrB_Vector_setElement(...,int32_t,...)            | GrB_Vector_setElement_INT32(..., int32_t,...)          |
| GrB_Vector_setElement(...,uint32_t,...)           | GrB_Vector_setElement_UINT32(..., uint32_t,...)        |
| GrB_Vector_setElement(...,int64_t,...)            | GrB_Vector_setElement_INT64(..., int64_t,...)          |
| GrB_Vector_setElement(...,uint64_t,...)           | GrB_Vector_setElement_UINT64(..., uint64_t,...)        |
| GrB_Vector_setElement(...,float,...)              | GrB_Vector_setElement_FP32(..., float,...)             |
| GrB_Vector_setElement(...,double,...)             | GrB_Vector_setElement_FP64(..., double,...)            |
| GrB_Vector_setElement(..., <i>other</i> ,...)     | GrB_Vector_setElement_UDT(...,const void*,...)         |
| GrB_Vector_extractElement(GrB_Scalar,...)         | GrB_Vector_extractElement_Scalar(GrB_Scalar,...)       |
| GrB_Vector_extractElement(bool*,...)              | GrB_Vector_extractElement_BOOL(bool*,...)              |
| GrB_Vector_extractElement(int8_t*,...)            | GrB_Vector_extractElement_INT8(int8_t*,...)            |
| GrB_Vector_extractElement(uint8_t*,...)           | GrB_Vector_extractElement_UINT8(uint8_t*,...)          |
| GrB_Vector_extractElement(int16_t*,...)           | GrB_Vector_extractElement_INT16(int16_t*,...)          |
| GrB_Vector_extractElement(uint16_t*,...)          | GrB_Vector_extractElement_UINT16(uint16_t*,...)        |
| GrB_Vector_extractElement(int32_t*,...)           | GrB_Vector_extractElement_INT32(int32_t*,...)          |
| GrB_Vector_extractElement(uint32_t*,...)          | GrB_Vector_extractElement_UINT32(uint32_t*,...)        |
| GrB_Vector_extractElement(int64_t*,...)           | GrB_Vector_extractElement_INT64(int64_t*,...)          |
| GrB_Vector_extractElement(uint64_t*,...)          | GrB_Vector_extractElement_UINT64(uint64_t*,...)        |
| GrB_Vector_extractElement(float*,...)             | GrB_Vector_extractElement_FP32(float*,...)             |
| GrB_Vector_extractElement(double*,...)            | GrB_Vector_extractElement_FP64(double*,...)            |
| GrB_Vector_extractElement( <i>other</i> *,...)    | GrB_Vector_extractElement_UDT(void*,...)               |
| GrB_Vector_extractTuples(...,bool*,...)           | GrB_Vector_extractTuples_BOOL(..., bool*,...)          |
| GrB_Vector_extractTuples(...,int8_t*,...)         | GrB_Vector_extractTuples_INT8(..., int8_t*,...)        |
| GrB_Vector_extractTuples(...,uint8_t*,...)        | GrB_Vector_extractTuples_UINT8(..., uint8_t*,...)      |
| GrB_Vector_extractTuples(...,int16_t*,...)        | GrB_Vector_extractTuples_INT16(..., int16_t*,...)      |
| GrB_Vector_extractTuples(...,uint16_t*,...)       | GrB_Vector_extractTuples_UINT16(..., uint16_t*,...)    |
| GrB_Vector_extractTuples(...,int32_t*,...)        | GrB_Vector_extractTuples_INT32(..., int32_t*,...)      |
| GrB_Vector_extractTuples(...,uint32_t*,...)       | GrB_Vector_extractTuples_UINT32(..., uint32_t*,...)    |
| GrB_Vector_extractTuples(...,int64_t*,...)        | GrB_Vector_extractTuples_INT64(..., int64_t*,...)      |
| GrB_Vector_extractTuples(...,uint64_t*,...)       | GrB_Vector_extractTuples_UINT64(..., uint64_t*,...)    |
| GrB_Vector_extractTuples(...,float*,...)          | GrB_Vector_extractTuples_FP32(..., float*,...)         |
| GrB_Vector_extractTuples(...,double*,...)         | GrB_Vector_extractTuples_FP64(..., double*,...)        |
| GrB_Vector_extractTuples(..., <i>other</i> *,...) | GrB_Vector_extractTuples_UDT(..., void*,...)           |

Table 5.4: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

| Polymorphic signature                             | Nonpolymorphic signature                               |
|---|--|
| GrB_Matrix_build(...,const bool*,...)             | GrB_Matrix_build_BOOL(...,const bool*,...)             |
| GrB_Matrix_build(...,const int8_t*,...)           | GrB_Matrix_build_INT8(...,const int8_t*,...)           |
| GrB_Matrix_build(...,const uint8_t*,...)          | GrB_Matrix_build_UINT8(...,const uint8_t*,...)         |
| GrB_Matrix_build(...,const int16_t*,...)          | GrB_Matrix_build_INT16(...,const int16_t*,...)         |
| GrB_Matrix_build(...,const uint16_t*,...)         | GrB_Matrix_build_UINT16(...,const uint16_t*,...)       |
| GrB_Matrix_build(...,const int32_t*,...)          | GrB_Matrix_build_INT32(...,const int32_t*,...)         |
| GrB_Matrix_build(...,const uint32_t*,...)         | GrB_Matrix_build_UINT32(...,const uint32_t*,...)       |
| GrB_Matrix_build(...,const int64_t*,...)          | GrB_Matrix_build_INT64(...,const int64_t*,...)         |
| GrB_Matrix_build(...,const uint64_t*,...)         | GrB_Matrix_build_UINT64(...,const uint64_t*,...)       |
| GrB_Matrix_build(...,const float*,...)            | GrB_Matrix_build_FP32(...,const float*,...)            |
| GrB_Matrix_build(...,const double*,...)           | GrB_Matrix_build_FP64(...,const double*,...)           |
| GrB_Matrix_build(...,const <i>other</i> *,...)    | GrB_Matrix_build_UDT(...,const void*,...)              |
| GrB_Matrix_setElement(...,GrB_Scalar,...)         | GrB_Matrix_setElement_Scalar(...,const GrB_Scalar,...) |
| GrB_Matrix_setElement(...,bool,...)               | GrB_Matrix_setElement_BOOL(..., bool,...)              |
| GrB_Matrix_setElement(...,int8_t,...)             | GrB_Matrix_setElement_INT8(..., int8_t,...)            |
| GrB_Matrix_setElement(...,uint8_t,...)            | GrB_Matrix_setElement_UINT8(..., uint8_t,...)          |
| GrB_Matrix_setElement(...,int16_t,...)            | GrB_Matrix_setElement_INT16(..., int16_t,...)          |
| GrB_Matrix_setElement(...,uint16_t,...)           | GrB_Matrix_setElement_UINT16(..., uint16_t,...)        |
| GrB_Matrix_setElement(...,int32_t,...)            | GrB_Matrix_setElement_INT32(..., int32_t,...)          |
| GrB_Matrix_setElement(...,uint32_t,...)           | GrB_Matrix_setElement_UINT32(..., uint32_t,...)        |
| GrB_Matrix_setElement(...,int64_t,...)            | GrB_Matrix_setElement_INT64(..., int64_t,...)          |
| GrB_Matrix_setElement(...,uint64_t,...)           | GrB_Matrix_setElement_UINT64(..., uint64_t,...)        |
| GrB_Matrix_setElement(...,float,...)              | GrB_Matrix_setElement_FP32(..., float,...)             |
| GrB_Matrix_setElement(...,double,...)             | GrB_Matrix_setElement_FP64(..., double,...)            |
| GrB_Matrix_setElement(..., <i>other</i> ,...)     | GrB_Matrix_setElement_UDT(...,const void*,...)         |
| GrB_Matrix_extractElement(GrB_Scalar,...)         | GrB_Matrix_extractElement_Scalar(GrB_Scalar,...)       |
| GrB_Matrix_extractElement(bool*,...)              | GrB_Matrix_extractElement_BOOL(bool*,...)              |
| GrB_Matrix_extractElement(int8_t*,...)            | GrB_Matrix_extractElement_INT8(int8_t*,...)            |
| GrB_Matrix_extractElement(uint8_t*,...)           | GrB_Matrix_extractElement_UINT8(uint8_t*,...)          |
| GrB_Matrix_extractElement(int16_t*,...)           | GrB_Matrix_extractElement_INT16(int16_t*,...)          |
| GrB_Matrix_extractElement(uint16_t*,...)          | GrB_Matrix_extractElement_UINT16(uint16_t*,...)        |
| GrB_Matrix_extractElement(int32_t*,...)           | GrB_Matrix_extractElement_INT32(int32_t*,...)          |
| GrB_Matrix_extractElement(uint32_t*,...)          | GrB_Matrix_extractElement_UINT32(uint32_t*,...)        |
| GrB_Matrix_extractElement(int64_t*,...)           | GrB_Matrix_extractElement_INT64(int64_t*,...)          |
| GrB_Matrix_extractElement(uint64_t*,...)          | GrB_Matrix_extractElement_UINT64(uint64_t*,...)        |
| GrB_Matrix_extractElement(float*,...)             | GrB_Matrix_extractElement_FP32(float*,...)             |
| GrB_Matrix_extractElement(double*,...)            | GrB_Matrix_extractElement_FP64(double*,...)            |
| GrB_Matrix_extractElement( <i>other</i> ,...)     | GrB_Matrix_extractElement_UDT(void*,...)               |
| GrB_Matrix_extractTuples(..., bool*,...)          | GrB_Matrix_extractTuples_BOOL(..., bool*,...)          |
| GrB_Matrix_extractTuples(..., int8_t*,...)        | GrB_Matrix_extractTuples_INT8(..., int8_t*,...)        |
| GrB_Matrix_extractTuples(..., uint8_t*,...)       | GrB_Matrix_extractTuples_UINT8(..., uint8_t*,...)      |
| GrB_Matrix_extractTuples(..., int16_t*,...)       | GrB_Matrix_extractTuples_INT16(..., int16_t*,...)      |
| GrB_Matrix_extractTuples(..., uint16_t*,...)      | GrB_Matrix_extractTuples_UINT16(..., uint16_t*,...)    |
| GrB_Matrix_extractTuples(..., int32_t*,...)       | GrB_Matrix_extractTuples_INT32(..., int32_t*,...)      |
| GrB_Matrix_extractTuples(..., uint32_t*,...)      | GrB_Matrix_extractTuples_UINT32(..., uint32_t*,...)    |
| GrB_Matrix_extractTuples(..., int64_t*,...)       | GrB_Matrix_extractTuples_INT64(..., int64_t*,...)      |
| GrB_Matrix_extractTuples(..., uint64_t*,...)      | GrB_Matrix_extractTuples_UINT64(..., uint64_t*,...)    |
| GrB_Matrix_extractTuples(..., float*,...)         | GrB_Matrix_extractTuples_FP32(..., float*,...)         |
| GrB_Matrix_extractTuples(..., double*,...)        | GrB_Matrix_extractTuples_FP64(..., double*,...)        |
| GrB_Matrix_extractTuples(..., <i>other</i> *,...) | GrB_Matrix_extractTuples_UDT(..., void*,...)           |

Table 5.5: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

| Polymorphic signature                           | Nonpolymorphic signature                                     |
|---|--|
| GrB_Matrix_import(...,const bool*,...)          | GrB_Matrix_import_BOOL(...,const bool*,...)                  |
| GrB_Matrix_import(...,const int8_t*,...)        | GrB_Matrix_import_INT8(...,const int8_t*,...)                |
| GrB_Matrix_import(...,const uint8_t*,...)       | GrB_Matrix_import_UINT8(...,const uint8_t*,...)              |
| GrB_Matrix_import(...,const int16_t*,...)       | GrB_Matrix_import_INT16(...,const int16_t*,...)              |
| GrB_Matrix_import(...,const uint16_t*,...)      | GrB_Matrix_import_UINT16(...,const uint16_t*,...)            |
| GrB_Matrix_import(...,const int32_t*,...)       | GrB_Matrix_import_INT32(...,const int32_t*,...)              |
| GrB_Matrix_import(...,const uint32_t*,...)      | GrB_Matrix_import_UINT32(...,const uint32_t*,...)            |
| GrB_Matrix_import(...,const int64_t*,...)       | GrB_Matrix_import_INT64(...,const int64_t*,...)              |
| GrB_Matrix_import(...,const uint64_t*,...)      | GrB_Matrix_import_UINT64(...,const uint64_t*,...)            |
| GrB_Matrix_import(...,const float*,...)         | GrB_Matrix_import_FP32(...,const float*,...)                 |
| GrB_Matrix_import(...,const double*,...)        | GrB_Matrix_import_FP64(...,const double*,...)                |
| GrB_Matrix_import(...,const other,...)          | GrB_Matrix_import_UDT(...,const void*,...)                   |
| GrB_Matrix_export(...,bool*,...)                | GrB_Matrix_export_BOOL(...,bool*,...)                        |
| GrB_Matrix_export(...,int8_t*,...)              | GrB_Matrix_export_INT8(...,int8_t*,...)                      |
| GrB_Matrix_export(...,uint8_t*,...)             | GrB_Matrix_export_UINT8(...,uint8_t*,...)                    |
| GrB_Matrix_export(...,int16_t*,...)             | GrB_Matrix_export_INT16(...,int16_t*,...)                    |
| GrB_Matrix_export(...,uint16_t*,...)            | GrB_Matrix_export_UINT16(...,uint16_t*,...)                  |
| GrB_Matrix_export(...,int32_t*,...)             | GrB_Matrix_export_INT32(...,int32_t*,...)                    |
| GrB_Matrix_export(...,uint32_t*,...)            | GrB_Matrix_export_UINT32(...,uint32_t*,...)                  |
| GrB_Matrix_export(...,int64_t*,...)             | GrB_Matrix_export_INT64(...,int64_t*,...)                    |
| GrB_Matrix_export(...,uint64_t*,...)            | GrB_Matrix_export_UINT64(...,uint64_t*,...)                  |
| GrB_Matrix_export(...,float*,...)               | GrB_Matrix_export_FP32(...,float*,...)                       |
| GrB_Matrix_export(...,double*,...)              | GrB_Matrix_export_FP64(...,double*,...)                      |
| GrB_Matrix_export(...,other,...)                | GrB_Matrix_export_UDT(...,void*,...)                         |
| GrB_free(GrB_Type*)                             | GrB_Type_free(GrB_Type*)                                     |
| GrB_free(GrB_UnaryOp*)                          | GrB_UnaryOp_free(GrB_UnaryOp*)                               |
| GrB_free(GrB_IndexUnaryOp*)                     | GrB_IndexUnaryOp_free(GrB_IndexUnaryOp*)                     |
| GrB_free(GrB_BinaryOp*)                         | GrB_BinaryOp_free(GrB_BinaryOp*)                             |
| GrB_free(GrB_Monoid*)                           | GrB_Monoid_free(GrB_Monoid*)                                 |
| GrB_free(GrB_Semiring*)                         | GrB_Semiring_free(GrB_Semiring*)                             |
| GrB_free(GrB_Scalar*)                           | GrB_Scalar_free(GrB_Scalar*)                                 |
| GrB_free(GrB_Vector*)                           | GrB_Vector_free(GrB_Vector*)                                 |
| GrB_free(GrB_Matrix*)                           | GrB_Matrix_free(GrB_Matrix*)                                 |
| GrB_free(GrB_Descriptor*)                       | GrB_Descriptor_free(GrB_Descriptor*)                         |
| GrB_wait(GrB_Type, GrB_WaitMode)                | GrB_Type_wait(GrB_Type, GrB_WaitMode)                        |
| GrB_wait(GrB_UnaryOp, GrB_WaitMode)             | GrB_UnaryOp_wait(GrB_UnaryOp, GrB_WaitMode)                  |
| GrB_wait(GrB_IndexUnaryOp, GrB_WaitMode)        | GrB_IndexUnaryOp_wait(GrB_IndexUnaryOp, GrB_WaitMode)        |
| GrB_wait(GrB_BinaryOp, GrB_WaitMode)            | GrB_BinaryOp_wait(GrB_BinaryOp, GrB_WaitMode)                |
| GrB_wait(GrB_Monoid, GrB_WaitMode)              | GrB_Monoid_wait(GrB_Monoid, GrB_WaitMode)                    |
| GrB_wait(GrB_Semiring, GrB_WaitMode)            | GrB_Semiring_wait(GrB_Semiring, GrB_WaitMode)                |
| GrB_wait(GrB_Scalar, GrB_WaitMode)              | GrB_Scalar_wait(GrB_Scalar, GrB_WaitMode)                    |
| GrB_wait(GrB_Vector, GrB_WaitMode)              | GrB_Vector_wait(GrB_Vector, GrB_WaitMode)                    |
| GrB_wait(GrB_Matrix, GrB_WaitMode)              | GrB_Matrix_wait(GrB_Matrix, GrB_WaitMode)                    |
| GrB_wait(GrB_Descriptor, GrB_WaitMode)          | GrB_Descriptor_wait(GrB_Descriptor, GrB_WaitMode)            |
| GrB_error(const char**, const GrB_Type)         | GrB_Type_error(const char**, const GrB_Type)                 |
| GrB_error(const char**, const GrB_UnaryOp)      | GrB_UnaryOp_error(const char**, const GrB_UnaryOp)           |
| GrB_error(const char**, const GrB_IndexUnaryOp) | GrB_IndexUnaryOp_error(const char**, const GrB_IndexUnaryOp) |
| GrB_error(const char**, const GrB_BinaryOp)     | GrB_BinaryOp_error(const char**, const GrB_BinaryOp)         |
| GrB_error(const char**, const GrB_Monoid)       | GrB_Monoid_error(const char**, const GrB_Monoid)             |
| GrB_error(const char**, const GrB_Semiring)     | GrB_Semiring_error(const char**, const GrB_Semiring)         |
| GrB_error(const char**, const GrB_Scalar)       | GrB_Scalar_error(const char**, const GrB_Scalar)             |
| GrB_error(const char**, const GrB_Vector)       | GrB_Vector_error(const char**, const GrB_Vector)             |
| GrB_error(const char**, const GrB_Matrix)       | GrB_Matrix_error(const char**, const GrB_Matrix)             |
| GrB_error(const char**, const GrB_Descriptor)   | GrB_Descriptor_error(const char**, const GrB_Descriptor)     |

Table 5.6: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

| Polymorphic signature                                      | Nonpolymorphic signature                                       |
|--|--|
| GrB_eWiseMult(GrB_Vector,...,GrB_Semiring,...)             | GrB_Vector_eWiseMult_Semiring(GrB_Vector,...,GrB_Semiring,...) |
| GrB_eWiseMult(GrB_Vector,...,GrB_Monoid,...)               | GrB_Vector_eWiseMult_Monoid(GrB_Vector,...,GrB_Monoid,...)     |
| GrB_eWiseMult(GrB_Vector,...,GrB_BinaryOp,...)             | GrB_Vector_eWiseMult_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...) |
| GrB_eWiseMult(GrB_Matrix,...,GrB_Semiring,...)             | GrB_Matrix_eWiseMult_Semiring(GrB_Matrix,...,GrB_Semiring,...) |
| GrB_eWiseMult(GrB_Matrix,...,GrB_Monoid,...)               | GrB_Matrix_eWiseMult_Monoid(GrB_Matrix,...,GrB_Monoid,...)     |
| GrB_eWiseMult(GrB_Matrix,...,GrB_BinaryOp,...)             | GrB_Matrix_eWiseMult_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...) |
| GrB_eWiseAdd(GrB_Vector,...,GrB_Semiring,...)              | GrB_Vector_eWiseAdd_Semiring(GrB_Vector,...,GrB_Semiring,...)  |
| GrB_eWiseAdd(GrB_Vector,...,GrB_Monoid,...)                | GrB_Vector_eWiseAdd_Monoid(GrB_Vector,...,GrB_Monoid,...)      |
| GrB_eWiseAdd(GrB_Vector,...,GrB_BinaryOp,...)              | GrB_Vector_eWiseAdd_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)  |
| GrB_eWiseAdd(GrB_Matrix,...,GrB_Semiring,...)              | GrB_Matrix_eWiseAdd_Semiring(GrB_Matrix,...,GrB_Semiring,...)  |
| GrB_eWiseAdd(GrB_Matrix,...,GrB_Monoid,...)                | GrB_Matrix_eWiseAdd_Monoid(GrB_Matrix,...,GrB_Monoid,...)      |
| GrB_eWiseAdd(GrB_Matrix,...,GrB_BinaryOp,...)              | GrB_Matrix_eWiseAdd_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)  |
| GrB_extract(GrB_Vector,...,GrB_Vector,...)                 | GrB_Vector_extract(GrB_Vector,...,GrB_Vector,...)              |
| GrB_extract(GrB_Matrix,...,GrB_Matrix,...)                 | GrB_Matrix_extract(GrB_Matrix,...,GrB_Matrix,...)              |
| GrB_extract(GrB_Vector,...,GrB_Matrix,...)                 | GrB_Col_extract(GrB_Vector,...,GrB_Matrix,...)                 |
| GrB_assign(GrB_Vector,...,GrB_Vector,...)                  | GrB_Vector_assign(GrB_Vector,...,GrB_Vector,...)               |
| GrB_assign(GrB_Matrix,...,GrB_Matrix,...)                  | GrB_Matrix_assign(GrB_Matrix,...,GrB_Matrix,...)               |
| GrB_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...) | GrB_Col_assign(GrB_Matrix,...,GrB_Vector,const GrB_Index*,...) |
| GrB_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)        | GrB_Row_assign(GrB_Matrix,...,GrB_Vector,GrB_Index,...)        |
| GrB_assign(GrB_Vector,...,GrB_Scalar,...)                  | GrB_Vector_assign_Scalar(GrB_Vector,...,const GrB_Scalar,...)  |
| GrB_assign(GrB_Vector,...,bool,...)                        | GrB_Vector_assign_BOOL(GrB_Vector,..., bool,...)               |
| GrB_assign(GrB_Vector,...,int8_t,...)                      | GrB_Vector_assign_INT8(GrB_Vector,..., int8_t,...)             |
| GrB_assign(GrB_Vector,...,uint8_t,...)                     | GrB_Vector_assign_UINT8(GrB_Vector,..., uint8_t,...)           |
| GrB_assign(GrB_Vector,...,int16_t,...)                     | GrB_Vector_assign_INT16(GrB_Vector,..., int16_t,...)           |
| GrB_assign(GrB_Vector,...,uint16_t,...)                    | GrB_Vector_assign_UINT16(GrB_Vector,..., uint16_t,...)         |
| GrB_assign(GrB_Vector,...,int32_t,...)                     | GrB_Vector_assign_INT32(GrB_Vector,..., int32_t,...)           |
| GrB_assign(GrB_Vector,...,uint32_t,...)                    | GrB_Vector_assign_UINT32(GrB_Vector,..., uint32_t,...)         |
| GrB_assign(GrB_Vector,...,int64_t,...)                     | GrB_Vector_assign_INT64(GrB_Vector,..., int64_t,...)           |
| GrB_assign(GrB_Vector,...,uint64_t,...)                    | GrB_Vector_assign_UINT64(GrB_Vector,..., uint64_t,...)         |
| GrB_assign(GrB_Vector,...,float,...)                       | GrB_Vector_assign_FP32(GrB_Vector,..., float,...)              |
| GrB_assign(GrB_Vector,...,double,...)                      | GrB_Vector_assign_FP64(GrB_Vector,..., double,...)             |
| GrB_assign(GrB_Vector,...,other,...)                       | GrB_Vector_assign_UDT(GrB_Vector,...,const void*,...)          |
| GrB_assign(GrB_Matrix,...,GrB_Scalar,...)                  | GrB_Matrix_assign_Scalar(GrB_Matrix,...,const GrB_Scalar,...)  |
| GrB_assign(GrB_Matrix,...,bool,...)                        | GrB_Matrix_assign_BOOL(GrB_Matrix,..., bool,...)               |
| GrB_assign(GrB_Matrix,...,int8_t,...)                      | GrB_Matrix_assign_INT8(GrB_Matrix,..., int8_t,...)             |
| GrB_assign(GrB_Matrix,...,uint8_t,...)                     | GrB_Matrix_assign_UINT8(GrB_Matrix,..., uint8_t,...)           |
| GrB_assign(GrB_Matrix,...,int16_t,...)                     | GrB_Matrix_assign_INT16(GrB_Matrix,..., int16_t,...)           |
| GrB_assign(GrB_Matrix,...,uint16_t,...)                    | GrB_Matrix_assign_UINT16(GrB_Matrix,..., uint16_t,...)         |
| GrB_assign(GrB_Matrix,...,int32_t,...)                     | GrB_Matrix_assign_INT32(GrB_Matrix,..., int32_t,...)           |
| GrB_assign(GrB_Matrix,...,uint32_t,...)                    | GrB_Matrix_assign_UINT32(GrB_Matrix,..., uint32_t,...)         |
| GrB_assign(GrB_Matrix,...,int64_t,...)                     | GrB_Matrix_assign_INT64(GrB_Matrix,..., int64_t,...)           |
| GrB_assign(GrB_Matrix,...,uint64_t,...)                    | GrB_Matrix_assign_UINT64(GrB_Matrix,..., uint64_t,...)         |
| GrB_assign(GrB_Matrix,...,float,...)                       | GrB_Matrix_assign_FP32(GrB_Matrix,..., float,...)              |
| GrB_assign(GrB_Matrix,...,double,...)                      | GrB_Matrix_assign_FP64(GrB_Matrix,..., double,...)             |
| GrB_assign(GrB_Matrix,...,other,...)                       | GrB_Matrix_assign_UDT(GrB_Matrix,...,const void*,...)          |



Table 5.7: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

| Polymorphic signature  | Nonpolymorphic signature   |
|--|--|
| GrB_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)                 | GrB_Vector_apply(GrB_Vector,...,GrB_UnaryOp,GrB_Vector,...)                                |
| GrB_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)                 | GrB_Matrix_apply(GrB_Matrix,...,GrB_UnaryOp,GrB_Matrix,...)                                |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...)     | GrB_Vector_apply_BinaryOp1st_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Scalar,GrB_Vector,...) |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)           | GrB_Vector_apply_BinaryOp1st_BOOL(GrB_Vector,...,GrB_BinaryOp,bool,GrB_Vector,...)         |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)         | GrB_Vector_apply_BinaryOp1st_INT8(GrB_Vector,...,GrB_BinaryOp,int8_t,GrB_Vector,...)       |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)        | GrB_Vector_apply_BinaryOp1st_UINT8(GrB_Vector,...,GrB_BinaryOp,uint8_t,GrB_Vector,...)     |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)        | GrB_Vector_apply_BinaryOp1st_INT16(GrB_Vector,...,GrB_BinaryOp,int16_t,GrB_Vector,...)     |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)       | GrB_Vector_apply_BinaryOp1st_UINT16(GrB_Vector,...,GrB_BinaryOp,uint16_t,GrB_Vector,...)   |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)        | GrB_Vector_apply_BinaryOp1st_INT32(GrB_Vector,...,GrB_BinaryOp,int32_t,GrB_Vector,...)     |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)       | GrB_Vector_apply_BinaryOp1st_UINT32(GrB_Vector,...,GrB_BinaryOp,uint32_t,GrB_Vector,...)   |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)        | GrB_Vector_apply_BinaryOp1st_INT64(GrB_Vector,...,GrB_BinaryOp,int64_t,GrB_Vector,...)     |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)       | GrB_Vector_apply_BinaryOp1st_UINT64(GrB_Vector,...,GrB_BinaryOp,uint64_t,GrB_Vector,...)   |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)          | GrB_Vector_apply_BinaryOp1st_FP32(GrB_Vector,...,GrB_BinaryOp,float,GrB_Vector,...)        |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)         | GrB_Vector_apply_BinaryOp1st_FP64(GrB_Vector,...,GrB_BinaryOp,double,GrB_Vector,...)       |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp, <i>other</i> ,GrB_Vector,...) | GrB_Vector_apply_BinaryOp1st_UDT(GrB_Vector,...,GrB_BinaryOp,const void*,GrB_Vector,...)   |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...)     | GrB_Vector_apply_BinaryOp2nd_Scalar(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,GrB_Scalar,...) |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)           | GrB_Vector_apply_BinaryOp2nd_BOOL(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,bool,...)         |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)         | GrB_Vector_apply_BinaryOp2nd_INT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int8_t,...)       |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)        | GrB_Vector_apply_BinaryOp2nd_UINT8(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint8_t,...)     |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)        | GrB_Vector_apply_BinaryOp2nd_INT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int16_t,...)     |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)       | GrB_Vector_apply_BinaryOp2nd_UINT16(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint16_t,...)   |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)        | GrB_Vector_apply_BinaryOp2nd_INT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int32_t,...)     |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)       | GrB_Vector_apply_BinaryOp2nd_UINT32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint32_t,...)   |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)        | GrB_Vector_apply_BinaryOp2nd_INT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,int64_t,...)     |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)       | GrB_Vector_apply_BinaryOp2nd_UINT64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,uint64_t,...)   |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)          | GrB_Vector_apply_BinaryOp2nd_FP32(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,float,...)        |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)         | GrB_Vector_apply_BinaryOp2nd_FP64(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,double,...)       |
| GrB_apply(GrB_Vector,...,GrB_BinaryOp,GrB_Vector, <i>other</i> ,...) | GrB_Vector_apply_BinaryOp2nd_UDT(GrB_Vector,...,GrB_BinaryOp,GrB_Vector,const void*,...)   |

Table 5.8: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

| Polymorphic signature  | Nonpolymorphic signature   |
|--|--|
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...)     | GrB_Matrix_apply_BinaryOp1st_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Scalar,GrB_Matrix,...) |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)           | GrB_Matrix_apply_BinaryOp1st_BOOL(GrB_Matrix,...,GrB_BinaryOp,bool,GrB_Matrix,...)         |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)         | GrB_Matrix_apply_BinaryOp1st_INT8(GrB_Matrix,...,GrB_BinaryOp,int8_t,GrB_Matrix,...)       |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)        | GrB_Matrix_apply_BinaryOp1st_UINT8(GrB_Matrix,...,GrB_BinaryOp,uint8_t,GrB_Matrix,...)     |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)        | GrB_Matrix_apply_BinaryOp1st_INT16(GrB_Matrix,...,GrB_BinaryOp,int16_t,GrB_Matrix,...)     |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)       | GrB_Matrix_apply_BinaryOp1st_UINT16(GrB_Matrix,...,GrB_BinaryOp,uint16_t,GrB_Matrix,...)   |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)        | GrB_Matrix_apply_BinaryOp1st_INT32(GrB_Matrix,...,GrB_BinaryOp,int32_t,GrB_Matrix,...)     |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)       | GrB_Matrix_apply_BinaryOp1st_UINT32(GrB_Matrix,...,GrB_BinaryOp,uint32_t,GrB_Matrix,...)   |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)        | GrB_Matrix_apply_BinaryOp1st_INT64(GrB_Matrix,...,GrB_BinaryOp,int64_t,GrB_Matrix,...)     |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)       | GrB_Matrix_apply_BinaryOp1st_UINT64(GrB_Matrix,...,GrB_BinaryOp,uint64_t,GrB_Matrix,...)   |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)          | GrB_Matrix_apply_BinaryOp1st_FP32(GrB_Matrix,...,GrB_BinaryOp,float,GrB_Matrix,...)        |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)         | GrB_Matrix_apply_BinaryOp1st_FP64(GrB_Matrix,...,GrB_BinaryOp,double,GrB_Matrix,...)       |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp, <i>other</i> ,GrB_Matrix,...) | GrB_Matrix_apply_BinaryOp1st_UDT(GrB_Matrix,...,GrB_BinaryOp,const void*,GrB_Matrix,...)   |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...)     | GrB_Matrix_apply_BinaryOp2nd_Scalar(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,GrB_Scalar,...) |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)           | GrB_Matrix_apply_BinaryOp2nd_BOOL(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,bool,...)         |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)         | GrB_Matrix_apply_BinaryOp2nd_INT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int8_t,...)       |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)        | GrB_Matrix_apply_BinaryOp2nd_UINT8(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint8_t,...)     |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)        | GrB_Matrix_apply_BinaryOp2nd_INT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int16_t,...)     |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)       | GrB_Matrix_apply_BinaryOp2nd_UINT16(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint16_t,...)   |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)        | GrB_Matrix_apply_BinaryOp2nd_INT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int32_t,...)     |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)       | GrB_Matrix_apply_BinaryOp2nd_UINT32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint32_t,...)   |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)        | GrB_Matrix_apply_BinaryOp2nd_INT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,int64_t,...)     |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)       | GrB_Matrix_apply_BinaryOp2nd_UINT64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,uint64_t,...)   |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)          | GrB_Matrix_apply_BinaryOp2nd_FP32(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,float,...)        |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)         | GrB_Matrix_apply_BinaryOp2nd_FP64(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,double,...)       |
| GrB_apply(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix, <i>other</i> ,...) | GrB_Matrix_apply_BinaryOp2nd_UDT(GrB_Matrix,...,GrB_BinaryOp,GrB_Matrix,const void*,...)   |

Table 5.9: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

| Polymorphic signature  | Nonpolymorphic signature   |
|--|--|
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)     | GrB_Vector_apply_IndexOp_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...) |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)           | GrB_Vector_apply_IndexOp_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)         |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)         | GrB_Vector_apply_IndexOp_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)       |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)        | GrB_Vector_apply_IndexOp_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)     |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)        | GrB_Vector_apply_IndexOp_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)     |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)       | GrB_Vector_apply_IndexOp_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)   |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)        | GrB_Vector_apply_IndexOp_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)     |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)       | GrB_Vector_apply_IndexOp_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)   |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)        | GrB_Vector_apply_IndexOp_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)     |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)       | GrB_Vector_apply_IndexOp_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)   |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)          | GrB_Vector_apply_IndexOp_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)        |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)         | GrB_Vector_apply_IndexOp_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)       |
| GrB_apply(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector, <i>other</i> ,...) | GrB_Vector_apply_IndexOp_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)   |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)     | GrB_Matrix_apply_IndexOp_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...) |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)           | GrB_Matrix_apply_IndexOp_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)         |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)         | GrB_Matrix_apply_IndexOp_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)       |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)        | GrB_Matrix_apply_IndexOp_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)     |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)        | GrB_Matrix_apply_IndexOp_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)     |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)       | GrB_Matrix_apply_IndexOp_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)   |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)        | GrB_Matrix_apply_IndexOp_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)     |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)       | GrB_Matrix_apply_IndexOp_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)   |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)        | GrB_Matrix_apply_IndexOp_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)     |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)       | GrB_Matrix_apply_IndexOp_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)   |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)          | GrB_Matrix_apply_IndexOp_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)        |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)         | GrB_Matrix_apply_IndexOp_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)       |
| GrB_apply(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix, <i>other</i> ,...) | GrB_Matrix_apply_IndexOp_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)   |

Table 5.10: Long-name, nonpolymorphic form of GraphBLAS methods (continued).[\[Scott: NEW CONTENT\]](#)

| Polymorphic signature  | Nonpolymorphic signature   |
|--|--|
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code> | <code>GrB_Vector_select_Scalar(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,GrB_Scalar,...)</code> |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>       | <code>GrB_Vector_select_BOOL(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,bool,...)</code>         |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>     | <code>GrB_Vector_select_INT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int8_t,...)</code>       |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>    | <code>GrB_Vector_select_UINT8(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint8_t,...)</code>     |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>    | <code>GrB_Vector_select_INT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int16_t,...)</code>     |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>   | <code>GrB_Vector_select_UINT16(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint16_t,...)</code>   |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>    | <code>GrB_Vector_select_INT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int32_t,...)</code>     |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>   | <code>GrB_Vector_select_UINT32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint32_t,...)</code>   |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>    | <code>GrB_Vector_select_INT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,int64_t,...)</code>     |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>   | <code>GrB_Vector_select_UINT64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,uint64_t,...)</code>   |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>      | <code>GrB_Vector_select_FP32(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,float,...)</code>        |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>     | <code>GrB_Vector_select_FP64(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,double,...)</code>       |
| <code>GrB_select(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,other,...)</code>      | <code>GrB_Vector_select_UDT(GrB_Vector,...,GrB_IndexUnaryOp,GrB_Vector,const void*,...)</code>   |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code> | <code>GrB_Matrix_select_Scalar(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,GrB_Scalar,...)</code> |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>       | <code>GrB_Matrix_select_BOOL(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,bool,...)</code>         |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>     | <code>GrB_Matrix_select_INT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int8_t,...)</code>       |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>    | <code>GrB_Matrix_select_UINT8(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint8_t,...)</code>     |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>    | <code>GrB_Matrix_select_INT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int16_t,...)</code>     |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>   | <code>GrB_Matrix_select_UINT16(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint16_t,...)</code>   |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>    | <code>GrB_Matrix_select_INT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int32_t,...)</code>     |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>   | <code>GrB_Matrix_select_UINT32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint32_t,...)</code>   |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>    | <code>GrB_Matrix_select_INT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,int64_t,...)</code>     |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>   | <code>GrB_Matrix_select_UINT64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,uint64_t,...)</code>   |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>      | <code>GrB_Matrix_select_FP32(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,float,...)</code>        |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>     | <code>GrB_Matrix_select_FP64(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,double,...)</code>       |
| <code>GrB_select(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,other,...)</code>      | <code>GrB_Matrix_select_UDT(GrB_Matrix,...,GrB_IndexUnaryOp,GrB_Matrix,const void*,...)</code>   |

Table 5.11: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

| Polymorphic signature                                  | Nonpolymorphic signature  |
|--|---|
| GrB_reduce(GrB_Vector,...,GrB_Monoid,...)              | GrB_Matrix_reduce_Monoid(GrB_Vector,...,GrB_Monoid,...)                       |
| GrB_reduce(GrB_Vector,...,GrB_BinaryOp,...)            | GrB_Matrix_reduce_BinaryOp(GrB_Vector,...,GrB_BinaryOp,...)                   |
| GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Vector,...)   | GrB_Vector_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Vector,...)                |
| GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Vector,...) | GrB_Vector_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_Vector,...)              |
| GrB_reduce(bool*,...,GrB_Vector,...)                   | GrB_Vector_reduce_BOOL(bool*,...,GrB_Vector,...)                              |
| GrB_reduce(int8_t*,...,GrB_Vector,...)                 | GrB_Vector_reduce_INT8(int8_t*,...,GrB_Vector,...)                            |
| GrB_reduce(uint8_t*,...,GrB_Vector,...)                | GrB_Vector_reduce_UINT8(uint8_t*,...,GrB_Vector,...)                          |
| GrB_reduce(int16_t*,...,GrB_Vector,...)                | GrB_Vector_reduce_INT16(int16_t*,...,GrB_Vector,...)                          |
| GrB_reduce(uint16_t*,...,GrB_Vector,...)               | GrB_Vector_reduce_UINT16(uint16_t*,...,GrB_Vector,...)                        |
| GrB_reduce(int32_t*,...,GrB_Vector,...)                | GrB_Vector_reduce_INT32(int32_t*,...,GrB_Vector,...)                          |
| GrB_reduce(uint32_t*,...,GrB_Vector,...)               | GrB_Vector_reduce_UINT32(uint32_t*,...,GrB_Vector,...)                        |
| GrB_reduce(int64_t*,...,GrB_Vector,...)                | GrB_Vector_reduce_INT64(int64_t*,...,GrB_Vector,...)                          |
| GrB_reduce(uint64_t*,...,GrB_Vector,...)               | GrB_Vector_reduce_UINT64(uint64_t*,...,GrB_Vector,...)                        |
| GrB_reduce(float*,...,GrB_Vector,...)                  | GrB_Vector_reduce_FP32(float*,...,GrB_Vector,...)                             |
| GrB_reduce(double*,...,GrB_Vector,...)                 | GrB_Vector_reduce_FP64(double*,...,GrB_Vector,...)                            |
| GrB_reduce( <i>other</i> *,...,GrB_Vector,...)         | GrB_Vector_reduce_UDT(void*,...,GrB_Vector,...)                               |
| GrB_reduce(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)   | GrB_Matrix_reduce_Monoid_Scalar(GrB_Scalar,...,GrB_Monoid,GrB_Matrix,...)     |
| GrB_reduce(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...) | GrB_Matrix_reduce_BinaryOp_Scalar(GrB_Scalar,...,GrB_BinaryOp,GrB_Matrix,...) |
| GrB_reduce(bool*,...,GrB_Matrix,...)                   | GrB_Matrix_reduce_BOOL(bool*,...,GrB_Matrix,...)                              |
| GrB_reduce(int8_t*,...,GrB_Matrix,...)                 | GrB_Matrix_reduce_INT8(int8_t*,...,GrB_Matrix,...)                            |
| GrB_reduce(uint8_t*,...,GrB_Matrix,...)                | GrB_Matrix_reduce_UINT8(uint8_t*,...,GrB_Matrix,...)                          |
| GrB_reduce(int16_t*,...,GrB_Matrix,...)                | GrB_Matrix_reduce_INT16(int16_t*,...,GrB_Matrix,...)                          |
| GrB_reduce(uint16_t*,...,GrB_Matrix,...)               | GrB_Matrix_reduce_UINT16(uint16_t*,...,GrB_Matrix,...)                        |
| GrB_reduce(int32_t*,...,GrB_Matrix,...)                | GrB_Matrix_reduce_INT32(int32_t*,...,GrB_Matrix,...)                          |
| GrB_reduce(uint32_t*,...,GrB_Matrix,...)               | GrB_Matrix_reduce_UINT32(uint32_t*,...,GrB_Matrix,...)                        |
| GrB_reduce(int64_t*,...,GrB_Matrix,...)                | GrB_Matrix_reduce_INT64(int64_t*,...,GrB_Matrix,...)                          |
| GrB_reduce(uint64_t*,...,GrB_Matrix,...)               | GrB_Matrix_reduce_UINT64(uint64_t*,...,GrB_Matrix,...)                        |
| GrB_reduce(float*,...,GrB_Matrix,...)                  | GrB_Matrix_reduce_FP32(float*,...,GrB_Matrix,...)                             |
| GrB_reduce(double*,...,GrB_Matrix,...)                 | GrB_Matrix_reduce_FP64(double*,...,GrB_Matrix,...)                            |
| GrB_reduce( <i>other</i> *,...,GrB_Matrix,...)         | GrB_Matrix_reduce_UDT(void*,...,GrB_Matrix,...)                               |
| GrB_kronecker(GrB_Matrix,...,GrB_Semiring,...)         | GrB_Matrix_kronecker_Semiring(GrB_Matrix,...,GrB_Semiring,...)                |
| GrB_kronecker(GrB_Matrix,...,GrB_Monoid,...)           | GrB_Matrix_kronecker_Monoid(GrB_Matrix,...,GrB_Monoid,...)                    |
| GrB_kronecker(GrB_Matrix,...,GrB_BinaryOp,...)         | GrB_Matrix_kronecker_BinaryOp(GrB_Matrix,...,GrB_BinaryOp,...)                |



# Appendix A

## Revision history

Changes in 2.0.1 (Released: ## Xxxxx 2022:

- (Issue GH-69) Fix error in description of contents of matrix constructed from `GrB_Matrix_diag`.

Changes in 2.0.0 (Released: 15 November 2021:

- Reorganized Chapters 2 and 3: Chapter 2 contains prose regarding the basic concepts captured in the API; Chapter 3 presents all of the enumerations, literals, data types, and predefined objects required by the API. Made short captions for the List of Tables.
- (Issue BB-49, BB-50) Updated and corrected language regarding multithreading and completion, and requirements regarding acquire-release memory orders. Methods that used to force complete no longer do.
- (Issue BB-74, BB-9) Assigned integer values to all return codes as well as all enumerations in the API to ensure run-time compatibility between libraries.
- (Issues BB-70, BB-67) Changed semantics and signature of `GrB_wait(obj, mode)`. Added wait modes for 'complete' or 'materialize' and removed `GrB_wait(void)`. **This breaks backward compatibility.**
- (Issue GH-51) Removed deprecated `GrB_SCMP` literal from descriptor values. **This breaks backward compatibility.**
- (Issues BB-8, BB-36) Added sparse `GrB_Scalar` object and its use in additional variants of `extract/setElement` methods, and `reduce`, `apply`, `assign` and `select` operations.
- (Issues BB-34, GH-33, GH-45) Added new `select` operation that uses an index unary operator. Added new variants of `apply` that take an index unary operator (matrix and vector variants).
- (Issues BB-68, BB-51) Added `serialize` and `deserialize` methods for matrices to/from implementation defined formats.

- 7488 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified  
7489 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats  
7490 have been deferred.
- 7491 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7492 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return  
7493 error if duplicate locations encountered.
- 7494 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7495 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7496 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type T (not to  
7497 be confused with the proposed unary operator).
- 7498 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added  
7499 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to  
7500 a method that is not supported by the implementation.
- 7501 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque  
7502 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7503 • (Issue BB-45) Removed language about annihilators.
- 7504 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7505 • Updated a number algorithms in the appendix to use new operations and methods.
- 7506 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the  
7507 specification.
- 7508 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and  
7509 `GRB_SUBVERSION`.
- 7510 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7511 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection  
7512 fixed to read `eWiseAdd`/set union.
- 7513 • Typographical error (`NEQ`  $\rightarrow$  `NE`) in Description of Table 3.8.

7514 Changes in 1.3.0 (Released: 25 September 2019):

- 7515 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque  
7516 GraphBLAS object as an argument.
- 7517 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7518 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a  
7519 scalar.



7520  
7521  
7522  
7523  
7524  
7525  
7526  
7527  
7528  
7529  
7530  
7531  
7532  
7533  
7534  
7535  
7536  
7537  
7538  
7539  
7540  
7541  
7542  
7543  
7544  
7545  
7546  
7547  
7548  
7549  
7550  
7551  
7552

- (Issue BB-59) Changed specification about how reductions to scalar (`GrB_reduce`) are to be performed (to minimize dependence on monoid identity).
- (Issue BB-24) Added methods to resize matrices and vectors (`GrB_Matrix_resize` and `GrB_Vector_resize`).
- (Issue BB-47) Added methods to remove single elements from matrices and vectors (`GrB_Matrix_removeElement` and `GrB_Vector_removeElement`).
- (Issue BB-41) Added `GrB_STRUCTURE` descriptor flag for masks (consider only the structure of the mask and not the values).
- (Issue BB-64) Deprecated `GrB_SCMP` in favor of new `GrB_COMP` for descriptor values.
- (Issue BB-46) Added predefined descriptors covering all possible combinations of field, value pairs.
- Added unary operators: absolute value (`GrB_ABS_T`) and bitwise complement of integers (`GrB_BNOT_I`).
- (Issues BB-42, BB-62) Added binary operators: Added boolean exclusive-nor (`GrB_LXNOR`) and bitwise logical operators on integers (`GrB_BOR_I`, `GrB_BAND_I`, `GrB_BXOR_I`, `GrB_BXNOR_I`).
- (Issue BB-11) Added a set of predefined monoids and semirings.
- (Issue BB-57) Updated all examples in the appendix to take advantage of new capabilities and predefined objects.
- (Issue BB-43) Added parent-BFS example.
- (Issue BB-1) Fixed bug in the non-batch betweenness centrality algorithm in Appendix C.4 where source nodes were incorrectly assigned path counts.
- (Issue BB-3) Added compile-time preprocessor defines and runtime method for querying the GraphBLAS API version being used.
- (Issue BB-10) Clarified `GrB_init()` and `GrB_finalize()` errors.
- (Issue BB-16) Clarified behavior of boolean and integer division. **Note that `GrB_MINV` for integer and boolean types was removed from this version of the spec.**
- (Issue BB-19) Clarified aliasing in user-defined operators.
- (Issue BB-20) Clarified language about behavior of `GrB_free()` with predefined objects (implementation defined)
- (Issue BB-55) Clarified that multiplication does not have to distribute over addition in a GraphBLAS semiring.
- (Issue BB-45) Removed unnecessary language about annihilators.
- (Issue BB-61) Removed unnecessary language about implied zeros.
- (Issue BB-60) Added disclaimer against overspecification.

- 7553 • Fixed miscellaneous typographical errors (such as  $\otimes$ ,  $\oplus$ ).
- 7554 Changes in 1.2.0:
- 7555 • Removed "provisional" clause.
- 7556 Changes in 1.1.0:
- 7557 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
  - 7558 `assign` operations.
  - 7559 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
  - 7560 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
  - 7561 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
  - 7562 from the domains of the binary operator provided.
  - 7563 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
  - 7564 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
  - 7565 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
  - 7566 which is returned when the capacities of the output arrays are insufficient to hold all of the
  - 7567 tuples.
  - 7568 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
  - 7569 face.
  - 7570 • Added replace flag (`z`) notation to Table 4.1.
  - 7571 • Updated the "Mathematical Description" of the `assign` operation in Table 4.1.
  - 7572 • Added triangle counting example.
  - 7573 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
  - 7574 sections of GraphBLAS operations when the respective text was the "standard" text (i.e.,
  - 7575 identical in a majority of the operations).
  - 7576 • Fixed typographical errors.
- 7577 Changes in 1.0.2:
- 7578 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
  - 7579 matrices and avoid casting issues in certain implementations.
  - 7580 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
  - 7581 erased outside the assigned area.
  - 7582 • Changes non-polymorphic interface:
    - 7583 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7584           – Renamed GrB\_Vector\_reduce\_BinaryOp to GrB\_Matrix\_reduce\_BinaryOp.
- 7585           – Renamed GrB\_Vector\_reduce\_Monoid to GrB\_Matrix\_reduce\_Monoid.
- 7586       • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7587       • Fixed numerous typographical errors.



## Appendix B

# Non-opaque data format definitions

### B.1 GrB\_Format: Specify the format for input/output of a GraphBLAS matrix.

In this section, the non-opaque matrix formats specified by GrB\_Format and used in matrix import and export methods are defined.

#### B.1.1 GrB\_CSR\_FORMAT

The GrB\_CSR\_FORMAT format indicates that a matrix will be imported or exported using the compressed sparse row (CSR) format. `indptr` is a pointer to an array of GrB\_Index of size `nrows+1` elements, where the `i`'th index will contain the starting index in the `values` and `indices` arrays corresponding to the `i`'th row of the matrix. `indices` is a pointer to an array of number of stored elements (each a GrB\_Index), where each element contains the corresponding element's column index within a row of the matrix. `values` is a pointer to an array of number of stored elements (each the size of the scalar stored in the matrix) containing the corresponding value. The elements of each row are not required to be sorted by column index.

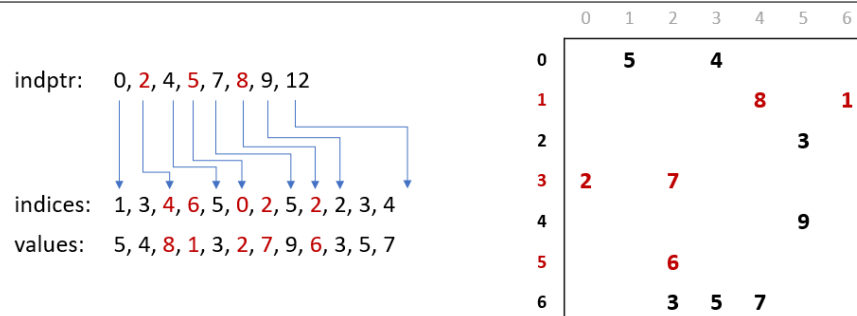


Figure B.1: Data layout for CSR format.

### B.1.2 GrB\_CSC\_FORMAT

The GrB\_CSC\_FORMAT format indicates that a matrix will be imported or exported using the compressed sparse column (CSC) format. `indptr` is a pointer to an array of `GrB_Index` of size `ncols+1` elements, where the *i*'th index will contain the starting index in the `values` and `indices` arrays corresponding to the *i*'th column of the matrix. `indices` is a pointer to an array of number of stored elements (each a `GrB_Index`), where each element contains the corresponding element's row index within a column of the matrix. `values` is a pointer to an array of number of stored elements (each the size of the scalar stored in the matrix) containing the corresponding value. The elements of each column are not required to be sorted by row index.

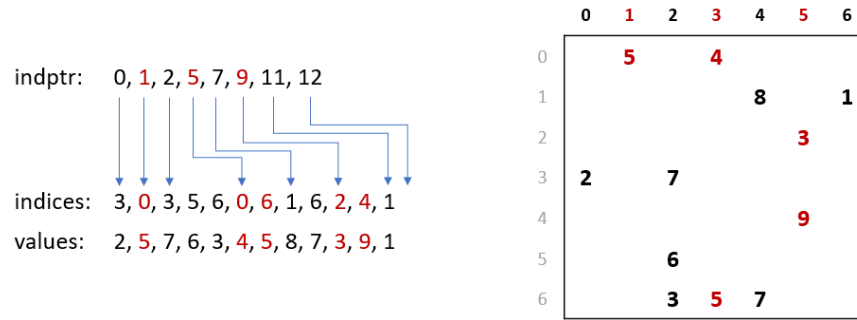


Figure B.2: Data layout for CSC format.

### B.1.3 GrB\_COO\_FORMAT

The GrB\_COO\_FORMAT format indicates that a matrix will be imported or exported using the coordinate list (COO) format. `indptr` is a pointer to an array of `GrB_Index` of size number of stored elements, where each element contains the corresponding element's column index. `indices` will be a pointer to an array of `GrB_Index` of size number of stored elements, where each element contains the corresponding element's row index. `values` will be a pointer to an array of size number of stored elements (each the size of the scalar stored in the matrix) containing the corresponding value. Elements are not required to be sorted in any order.

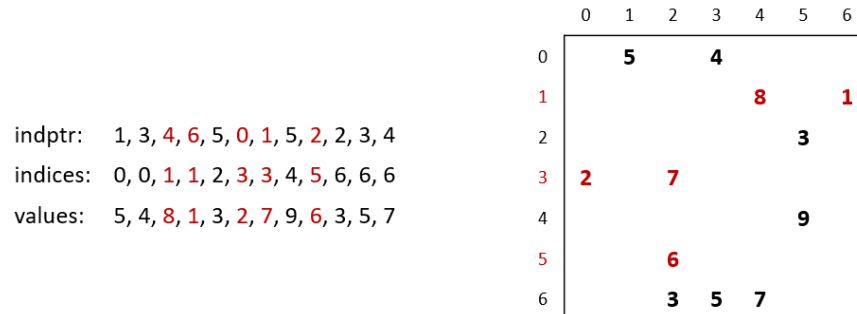


Figure B.3: Data layout for COO format.

7620 **Appendix C**

7621 **Examples**

## C.1 Example: Level breadth-first search (BFS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i] = 0$ . (Vector  $v$  should be empty on input.)
11  */
12  GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
13  {
14      GrB_Index n;
15      GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
16
17      GrB_Vector_new(v,GrB_INT32,n);     // Vector<int32_t>  $v(n)$ 
18
19      GrB_Vector q;                      // vertices visited in each level
20      GrB_Vector_new(&q,GrB_BOOL,n);     // Vector<bool>  $q(n)$ 
21      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
22
23      /*
24       * BFS traversal and label the vertices.
25       */
26      int32_t d = 0;                     //  $d = \text{level in BFS traversal}$ 
27      bool succ = false;                  //  $\text{succ} == \text{true}$  when some successor found
28      do {
29          ++d;                            // next level (start with 1)
30          GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL); //  $v[q] = d$ 
31          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
32                  q,A,GrB_DESC_RC);       //  $q[!v] = q \parallel A$ ; finds all the
33                                          // unvisited successors from current  $q$ 
34          GrB_reduce(&succ,GrB_NULL,GrB_LOR_MONOID_BOOL,
35                    q,GrB_NULL);          //  $\text{succ} = \parallel(q)$ 
36      } while (succ);                     // if there is no successor in  $q$ , we are done.
37
38      GrB_free(&q);                       //  $q$  vector no longer needed
39
40      return GrB_SUCCESS;
41  }

```



## C.2 Example: Level BFS in GraphBLAS using apply

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS traversal
9   * of the graph and sets  $v[i]$  to the level in which vertex  $i$  is visited ( $v[s] == 1$ ).
10  * If  $i$  is not reachable from  $s$ , then  $v[i]$  does not have a stored element.
11  * Vector  $v$  should be uninitialized on input.
12  */
13  GrB_Info BFS(GrB_Vector *v, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index n;
16      GrB_Matrix_nrows(&n,A);           //  $n = \#$  of rows of  $A$ 
17
18      GrB_Vector_new(v,GrB_INT32,n);     // Vector<int32_t>  $v(n) = 0$ 
19
20      GrB_Vector q;                      // vertices visited in each level
21      GrB_Vector_new(&q,GrB_BOOL,n);     // Vector<bool>  $q(n) = \text{false}$ 
22      GrB_Vector_setElement(q,(bool)true,s); //  $q[s] = \text{true}$ , false everywhere else
23
24      /*
25       * BFS traversal and label the vertices.
26       */
27      int32_t level = 0;                  // level = depth in BFS traversal
28      GrB_Index nvals;
29      do {
30          ++level;                        // next level (start with 1)
31          GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,
32                  GrB_SECOND_INT32,q,level,GrB_NULL); //  $v[q] = \text{level}$ 
33          GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
34                  q,A,GrB_DESC_RC);      //  $q[!v] = q \vee A$ ; finds all the
35                                          // unvisited successors from current  $q$ 
36          GrB_Vector_nvals(&nvals, q);
37      } while (nvals);                   // if there is no successor in  $q$ , we are done.
38
39      GrB_free(&q);                      //  $q$  vector no longer needed
40
41      return GrB_SUCCESS;
42  }

```

## C.3 Example: Parent BFS in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a binary  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ , performs a BFS
9   * traversal of the graph and sets  $parents[i]$  to the index of vertex  $i$ 's parent.
10  * The parent of the root vertex,  $s$ , will be set to itself ( $parents[s] = s$ ). If
11  * vertex  $i$  is not reachable from  $s$ ,  $parents[i]$  will not contain a stored value.
12  */
13  GrB_Info BFS(GrB_Vector *parents, const GrB_Matrix A, GrB_Index s)
14  {
15      GrB_Index N;
16      GrB_Matrix_nrows(&N, A);           //  $N = \#$  vertices
17
18      GrB_Vector_new(parents, GrB_UINT64, N);
19      GrB_Vector_setElement(*parents, s, s);           //  $parents[s] = s$ 
20
21      GrB_Vector wavefront;
22      GrB_Vector_new(&wavefront, GrB_UINT64, N);
23      GrB_Vector_setElement(wavefront, 1UL, s);       //  $wavefront[s] = 1$ 
24
25      /*
26       * BFS traversal and label the vertices.
27       */
28      GrB_Index nvals;
29      GrB_Vector_nvals(&nvals, wavefront);
30
31      while (nvals > 0)
32      {
33          // convert all stored values in wavefront to their 0-based index
34          GrB_apply(wavefront, GrB_NULL, GrB_NULL, GrB_ROWINDEX_INT64,
35                  wavefront, 0UL, GrB_NULL);
36
37          // "FIRST" because left-multiplying wavefront rows. Masking out the parent
38          // list ensures wavefront values do not overwrite parents already stored.
39          GrB_vxm(wavefront, *parents, GrB_NULL, GrB_MIN_FIRST_SEMIRING_UINT64,
40                  wavefront, A, GrB_DESC_RSC);
41
42          // Don't need to mask here since we did it in vxm. Merges new parents in
43          // current wavefront with existing parents:  $parents += wavefront$ 
44          GrB_apply(*parents, GrB_NULL, GrB_PLUS_UINT64,
45                  GrB_IDENTITY_UINT64, wavefront, GrB_NULL);
46
47          GrB_Vector_nvals(&nvals, wavefront);
48      }
49
50      GrB_free(&wavefront);
51
52      return GrB_SUCCESS;
53  }

```

## C.4 Example: Betweenness centrality (BC) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  /*
8   * Given a boolean  $n \times n$  adjacency matrix  $A$  and a source vertex  $s$ ,
9   * compute the BC-metric vector  $\delta$ , which should be empty on input.
10  */
11 GrB_Info BC(GrB_Vector *delta, GrB_Matrix A, GrB_Index s)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);                //  $n = \#$  of vertices in graph
15
16     GrB_Vector_new(delta, GrB_FP32, n);      // Vector<float>  $\delta(n)$ 
17
18     GrB_Matrix sigma;
19     GrB_Matrix_new(&sigma, GrB_INT32, n, n); //  $\sigma[d, k] = \#$  shortest paths to node  $k$  at level  $d$ 
20
21     GrB_Vector q;
22     GrB_Vector_new(&q, GrB_INT32, n);        // Vector<int32_t>  $q(n)$  of path counts
23     GrB_Vector_setElement(q, 1, s);          //  $q[s] = 1$ 
24
25     GrB_Vector p;
26     GrB_Vector_dup(&p, q);                  // Vector<int32_t>  $p(n)$  shortest path counts so far
27                                             //  $p = q$ 
28
29     GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
30             q, A, GrB_DESC_RC);              // get the first set of out neighbors
31
32     /*
33     * BFS phase
34     */
35     GrB_Index d = 0;                        // BFS level number
36     int32_t sum = 0;                        // sum == 0 when BFS phase is complete
37
38     do {
39         GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\sigma[d, :] = q$ 
40         GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
41         GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
42                 q, A, GrB_DESC_RC); //  $q = \#$  paths to nodes reachable
43                                     // from current level
44         GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
45         ++d;
46     } while (sum);
47
48     /*
49     * BC computation phase
50     * ( $t1, t2, t3, t4$ ) are temporary vectors
51     */
52     GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
53     GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
54     GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
55     GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
56
57     for (int i=d-1; i>0; i--)
58     {
59         GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t1 = 1 + \delta$ 
60         GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, t1, *delta, GrB_NULL);
61         GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t2 = \sigma[i, :]$ 
62         GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t2 = (1 + \delta) / \sigma[i, :]$ 
63         GrB_mvx(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
64                 // add contributions made by

```

```

63         A, t2, GrB_NULL);
64     GrB_extract(t4, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i-1, GrB_DESC_T0); // t4 = sigma[i-1,:]
65     GrB_eWiseMult(t4, GrB_NULL, GrB_NULL, GrB_TIMES_FP32, t4, t3, GrB_NULL); // t4 = sigma[i-1,:]*t3
66     GrB_eWiseAdd(delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, delta, t4, GrB_NULL); // accumulate into delta
67 }
68
69 GrB_free(&sigma);
70 GrB_free(&q); GrB_free(&p);
71 GrB_free(&t1); GrB_free(&t2); GrB_free(&t3); GrB_free(&t4);
72
73 return GrB_SUCCESS;
74 }

```

## C.5 Example: Batched BC in GraphBLAS

```

1  #include <stdlib.h>
2  #include "GraphBLAS.h" // in addition to other required C headers
3
4  // Compute partial BC metric for a subset of source vertices, s, in graph A
5  GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver)
6  {
7      GrB_Index n;
8      GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
9      GrB_Vector_new(delta, GrB_FP32, n); // Vector<float> delta(n)
10
11     // index and value arrays needed to build numsp
12     GrB_Index *i_nsver = (GrB_Index*) malloc(sizeof(GrB_Index)*nsver);
13     int32_t *ones = (int32_t*) malloc(sizeof(int32_t)*nsver);
14     for(int i=0; i<nsver; ++i) {
15         i_nsver[i] = i;
16         ones[i] = 1;
17     }
18
19     // numsp: structure holds the number of shortest paths for each node and starting vertex
20     // discovered so far. Initialized to source vertices: numsp[s[i],i]=1, i=[0,nsver)
21     GrB_Matrix numsp;
22     GrB_Matrix_new(&numsp, GrB_INT32, n, nsver);
23     GrB_Matrix_build(numsp, s, i_nsver, ones, nsver, GrB_PLUS_INT32);
24     free(i_nsver); free(ones);
25
26     // frontier: Holds the current frontier where values are path counts.
27     // Initialized to out vertices of each source node in s.
28     GrB_Matrix frontier;
29     GrB_Matrix_new(&frontier, GrB_INT32, n, nsver);
30     GrB_extract(frontier, numsp, GrB_NULL, A, GrB_ALL, n, s, nsver, GrB_DESC_RCT0);
31
32     // sigma: stores frontier information for each level of BFS phase. The memory
33     // for an entry in sigmas is only allocated within the do-while loop if needed.
34     // n is an upper bound on diameter.
35     GrB_Matrix *sigmas = (GrB_Matrix*) malloc(sizeof(GrB_Matrix)*n);
36
37     int32_t d = 0; // BFS level number
38     GrB_Index nvals = 0; // nvals == 0 when BFS phase is complete
39
40     // ----- The BFS phase (forward sweep) -----
41     do {
42         // sigmas[d](:,s) = dth level frontier from source vertex s
43         GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
44
45         GrB_apply(sigmas[d], GrB_NULL, GrB_NULL,
46                 GrB_IDENTITY_BOOL, frontier, GrB_NULL); // sigmas[d](:,:) = (Boolean) frontier
47         GrB_eWiseAdd(numsp, GrB_NULL, GrB_NULL, GrB_PLUS_INT32,
48                     numsp, frontier, GrB_NULL); // numsp += frontier (accum path counts)
49         GrB_mxm(frontier, numsp, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
50                 A, frontier, GrB_DESC_RCT0); // f<!numsp> = A' +.* f (update frontier)
51         GrB_Matrix_nvals(&nvals, frontier); // number of nodes in frontier at this level
52         d++;
53     } while (nvals);
54
55     // nspinv: the inverse of the number of shortest paths for each node and starting vertex.
56     GrB_Matrix nspinv;
57     GrB_Matrix_new(&nspinv, GrB_FP32, n, nsver);
58     GrB_apply(nspinv, GrB_NULL, GrB_NULL,
59              GrB_MINV_FP32, numsp, GrB_NULL); // nspinv = 1./numsp
60
61     // bcu: BC updates for each vertex for each starting vertex in s
62     GrB_Matrix bcu;

```

```

63 GrB_Matrix_new(&bcu, GrB_FP32, n, nsver);
64 GrB_assign(bcu, GrB_NULL, GrB_NULL,
65           1.0f, GrB_ALL, n, GrB_ALL, nsver, GrB_NULL); // filled with 1 to avoid sparsity issues
66
67 GrB_Matrix w; // temporary workspace matrix
68 GrB_Matrix_new(&w, GrB_FP32, n, nsver);
69
70 // ----- Tally phase (backward sweep) -----
71 for (int i=d-1; i>0; i--) {
72     GrB_eWiseMult(w, sigmas[i], GrB_NULL,
73                 GrB_TIMES_FP32, bcu, nspinv, GrB_DESC_R); // w<sigmas[i]>=(1 ./ nsp).*bcu
74
75     // add contributions by successors and mask with that BFS level's frontier
76     GrB_mxm(w, sigmas[i-1], GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
77            A, w, GrB_DESC_R); // w<sigmas[i-1]> = (A +.* w)
78     GrB_eWiseMult(bcu, GrB_NULL, GrB_PLUS_FP32, GrB_TIMES_FP32,
79                 w, numsp, GrB_NULL); // bcu += w .* numsp
80 }
81
82 // row reduce bcu and subtract "nsver" from every entry to account
83 // for 1 extra value per bcu row element.
84 GrB_reduce(*delta, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, bcu, GrB_NULL);
85 GrB_apply(*delta, GrB_NULL, GrB_NULL, GrB_MINUS_FP32, *delta, (float)nsver, GrB_NULL);
86
87 // Release resources
88 for (int i=0; i<d; i++) {
89     GrB_free(&(sigmas[i]));
90 }
91 free(sigmas);
92
93 GrB_free(&frontier); GrB_free(&numsp);
94 GrB_free(&nspinv); GrB_free(&bcu); GrB_free(&w);
95
96 return GrB_SUCCESS;
97 }

```

## C.6 Example: Maximal independent set (MIS) in GraphBLAS

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include "GraphBLAS.h"
6
7  // Assign a random number to each element scaled by the inverse of the node's degree.
8  // This will increase the probability that low degree nodes are selected and larger
9  // sets are selected.
10 void setRandom(void *out, const void *in)
11 {
12     uint32_t degree = *(uint32_t*)in;
13     *(float*)out = (0.0001f + random()/(1. + 2.*degree)); // add 1 to prevent divide by zero
14 }
15
16 /*
17  * A variant of Luby's randomized algorithm [Luby 1985].
18  *
19  * Given a numeric n x n adjacency matrix A of an unweighted and undirected graph (where
20  * the value true represents an edge), compute a maximal set of independent vertices and
21  * return it in a boolean n-vector, 'iset' where set[i] == true implies vertex i is a member
22  * of the set (the iset vector should be uninitialized on input.)
23  */
24 GrB_Info MIS(GrB_Vector *iset, const GrB_Matrix A)
25 {
26     GrB_Index n;
27     GrB_Matrix_nrows(&n,A); // n = # of rows of A
28
29     GrB_Vector prob; // holds random probabilities for each node
30     GrB_Vector neighbor_max; // holds value of max neighbor probability
31     GrB_Vector new_members; // holds set of new members to iset
32     GrB_Vector new_neighbors; // holds set of new neighbors to new iset mbrs.
33     GrB_Vector candidates; // candidate members to iset
34
35     GrB_Vector_new(&prob,GrB_FP32,n);
36     GrB_Vector_new(&neighbor_max,GrB_FP32,n);
37     GrB_Vector_new(&new_members,GrB_BOOL,n);
38     GrB_Vector_new(&new_neighbors,GrB_BOOL,n);
39     GrB_Vector_new(&candidates,GrB_BOOL,n);
40     GrB_Vector_new(iset,GrB_BOOL,n); // Initialize independent set vector, bool
41
42     GrB_UnaryOp set_random;
43     GrB_UnaryOp_new(&set_random,setRandom,GrB_FP32,GrB_UINT32);
44
45     // compute the degree of each vertex.
46     GrB_Vector degrees;
47     GrB_Vector_new(&degrees,GrB_FP64,n);
48     GrB_reduce(degrees,GrB_NULL,GrB_NULL,GrB_PLUS_FP64,A,GrB_NULL);
49
50     // Isolated vertices are not candidates: candidates[degrees != 0] = true
51     GrB_assign(candidates,degrees,GrB_NULL,true,GrB_ALL,n,GrB_NULL);
52
53     // add all singletons to iset: iset[degree == 0] = 1
54     GrB_assign(*iset,degrees,GrB_NULL,true,GrB_ALL,n,GrB_DESC_RC) ;
55
56     // Iterate while there are candidates to check.
57     GrB_Index nvals;
58     GrB_Vector_nvals(&nvals, candidates);
59     while (nvals > 0) {
60         // compute a random probability scaled by inverse of degree
61         GrB_apply(prob,candidates,GrB_NULL,set_random,degrees,GrB_DESC_R);
62     }

```

```

63 // compute the max probability of all neighbors
64 GrB_mnv(neighbor_max, candidates, GrB_NULL, GrB_MAX_SECOND_SEMIRING_FP32, A, prob, GrB_DESC_R);
65
66 // select vertex if its probability is larger than all its active neighbors,
67 // and apply a "masked no-op" to remove stored falses
68 GrB_eWiseAdd(new_members, GrB_NULL, GrB_NULL, GrB_GT_FP64, prob, neighbor_max, GrB_NULL);
69 GrB_apply(new_members, new_members, GrB_NULL, GrB_IDENTITY_BOOL, new_members, GrB_DESC_R);
70
71 // add new members to independent set.
72 GrB_eWiseAdd(*iset, GrB_NULL, GrB_NULL, GrB_LOR, *iset, new_members, GrB_NULL);
73
74 // remove new members from set of candidates  $c = c \ominus !new$ 
75 GrB_eWiseMult(candidates, new_members, GrB_NULL,
76               GrB_LAND, candidates, candidates, GrB_DESC_RC);
77
78 GrB_Vector_nvals(&nvals, candidates);
79 if (nvals == 0) { break; } // early exit condition
80
81 // Neighbors of new members can also be removed from candidates
82 GrB_mnv(new_neighbors, candidates, GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL,
83         A, new_members, GrB_NULL);
84 GrB_eWiseMult(candidates, new_neighbors, GrB_NULL, GrB_LAND,
85               candidates, candidates, GrB_DESC_RC);
86
87 GrB_Vector_nvals(&nvals, candidates);
88 }
89
90 GrB_free(&neighbor_max); // free all objects "new'ed"
91 GrB_free(&new_members);
92 GrB_free(&new_neighbors);
93 GrB_free(&prob);
94 GrB_free(&candidates);
95 GrB_free(&set_random);
96 GrB_free(&degrees);
97
98 return GrB_SUCCESS;
99 }

```



## C.7 Example: Counting triangles in GraphBLAS

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given an  $n \times n$  boolean adjacency matrix,  $A$ , of an undirected graph, computes
9  * the number of triangles in the graph.
10 */
11 uint64_t triangle_count(GrB_Matrix A)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);           //  $n = \#$  of vertices
15
16     //  $L$ :  $N \times N$ , lower-triangular, bool
17     GrB_Matrix L;
18     GrB_Matrix_new(&L, GrB_BOOL, n, n);
19     GrB_select(L, GrB_NULL, GrB_NULL, GrB_TRIL, A, 0UL, GrB_NULL);
20
21     GrB_Matrix C;
22     GrB_Matrix_new(&C, GrB_UINT64, n, n);
23
24     GrB_mxm(C, L, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_UINT64, L, L, GrB_NULL); //  $C \langle L \rangle = L +.* L$ 
25
26     uint64_t count;
27     GrB_reduce(&count, GrB_NULL, GrB_PLUS_MONOID_UINT64, C, GrB_NULL); // 1-norm of  $C$ 
28
29     GrB_free(&C);
30     GrB_free(&L);
31
32     return count;
33 }
```