

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

Version 2.1

Benjamin Brock, Aydın Buluç, Raye Kimmerer, Jim Kitchen, Manoj Kumar, Timothy  
Mattson, Scott McMillan, José Moreira, Erik Welch

Generated on 2023/12/10 at 15:28:35 EDT

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

6 Copyright © 2017-2023 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, Nvidia Corporation,  
10 Anaconda Inc., and Massachusetts Institute of Technology.

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, Nvidia Corporation, Anaconda Inc., or 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>15</b>
30	<b>2 Basic concepts</b>	<b>17</b>
31	2.1 Glossary . . . . .	17
32	2.1.1 GraphBLAS API basic definitions . . . . .	17
33	2.1.2 GraphBLAS objects and their structure . . . . .	18
34	2.1.3 Algebraic structures used in the GraphBLAS . . . . .	19
35	2.1.4 The execution of an application using the GraphBLAS C API . . . . .	20
36	2.1.5 GraphBLAS methods: behaviors and error conditions . . . . .	21
37	2.2 Notation . . . . .	23
38	2.3 Mathematical foundations . . . . .	24
39	2.4 GraphBLAS opaque objects . . . . .	25
40	2.5 Execution model . . . . .	26
41	2.5.1 Execution modes . . . . .	27
42	2.5.2 Multi-threaded execution . . . . .	28
43	2.6 Error model . . . . .	30
44	<b>3 Objects</b>	<b>33</b>
45	3.1 Enumerations for <code>init()</code> and <code>wait()</code> . . . . .	33
46	3.2 Indices, index arrays, and scalar arrays . . . . .	33
47	3.3 Types (domains) . . . . .	34

48	3.4	Algebraic objects, operators and associated functions . . . . .	35
49	3.4.1	Operators . . . . .	36
50	3.4.2	Monoids . . . . .	41
51	3.4.3	Semirings . . . . .	41
52	3.5	Collections . . . . .	45
53	3.5.1	Scalars . . . . .	45
54	3.5.2	Vectors . . . . .	45
55	3.5.3	Matrices . . . . .	46
56	3.5.3.1	External matrix formats . . . . .	46
57	3.5.4	Masks . . . . .	46
58	3.6	Descriptors . . . . .	47
59	3.7	Fields . . . . .	48
60	3.7.1	Input Types . . . . .	51
61	3.7.1.1	INT32 Handling . . . . .	51
62	3.7.1.2	GrB_Scalar Handling . . . . .	51
63	3.7.1.3	String (char*) Handling . . . . .	51
64	3.7.1.4	void* Handling . . . . .	51
65	3.7.1.5	SIZE Handling . . . . .	51
66	3.7.2	Hints . . . . .	52
67	3.7.3	GrB_NAME . . . . .	52
68	3.8	GrB_Info return values . . . . .	54
69	<b>4</b>	<b>Methods</b>	<b>57</b>
70	4.1	Context methods . . . . .	57
71	4.1.1	init: Initialize a GraphBLAS context . . . . .	57
72	4.1.2	finalize: Finalize a GraphBLAS context . . . . .	58
73	4.1.3	getVersion: Get the version number of the standard. . . . .	59
74	4.2	Object methods . . . . .	59
75	4.2.1	Get and Set methods . . . . .	60
76	4.2.1.1	get: Query the value of an object . . . . .	60
77	4.2.1.2	set: Set field of an object . . . . .	60

78	4.2.2	Algebra methods . . . . .	61
79	4.2.2.1	Type_new: Construct a new GraphBLAS (user-defined) type . . . .	61
80	4.2.2.2	UnaryOp_new: Construct a new GraphBLAS unary operator . . . .	62
81	4.2.2.3	BinaryOp_new: Construct a new GraphBLAS binary operator . . . .	64
82	4.2.2.4	Monoid_new: Construct a new GraphBLAS monoid . . . . .	66
83	4.2.2.5	Semiring_new: Construct a new GraphBLAS semiring . . . . .	67
84	4.2.2.6	IndexUnaryOp_new: Construct a new GraphBLAS index unary op-	
85		erator . . . . .	68
86	4.2.3	Scalar methods . . . . .	70
87	4.2.3.1	Scalar_new: Construct a new scalar . . . . .	70
88	4.2.3.2	Scalar_dup: Construct a copy of a GraphBLAS scalar . . . . .	71
89	4.2.3.3	Scalar_clear: Clear/remove a stored value from a scalar . . . . .	72
90	4.2.3.4	Scalar_nvals: Number of stored elements in a scalar . . . . .	73
91	4.2.3.5	Scalar_setElement: Set the single element in a scalar . . . . .	74
92	4.2.3.6	Scalar_extractElement: Extract a single element from a scalar. . . .	75
93	4.2.4	Vector methods . . . . .	76
94	4.2.4.1	Vector_new: Construct new vector . . . . .	76
95	4.2.4.2	Vector_dup: Construct a copy of a GraphBLAS vector . . . . .	77
96	4.2.4.3	Vector_resize: Resize a vector . . . . .	78
97	4.2.4.4	Vector_clear: Clear a vector . . . . .	79
98	4.2.4.5	Vector_size: Size of a vector . . . . .	80
99	4.2.4.6	Vector_nvals: Number of stored elements in a vector . . . . .	81
100	4.2.4.7	Vector_build: Store elements from tuples into a vector . . . . .	82
101	4.2.4.8	Vector_setElement: Set a single element in a vector . . . . .	84
102	4.2.4.9	Vector_removeElement: Remove an element from a vector . . . . .	85
103	4.2.4.10	Vector_extractElement: Extract a single element from a vector. . . .	86
104	4.2.4.11	Vector_extractTuples: Extract tuples from a vector . . . . .	88
105	4.2.5	Matrix methods . . . . .	90
106	4.2.5.1	Matrix_new: Construct new matrix . . . . .	90
107	4.2.5.2	Matrix_dup: Construct a copy of a GraphBLAS matrix . . . . .	91
108	4.2.5.3	Matrix_diag: Construct a diagonal GraphBLAS matrix . . . . .	92

109	4.2.5.4	Matrix_resize: Resize a matrix . . . . .	93
110	4.2.5.5	Matrix_clear: Clear a matrix . . . . .	94
111	4.2.5.6	Matrix_nrows: Number of rows in a matrix . . . . .	95
112	4.2.5.7	Matrix_ncols: Number of columns in a matrix . . . . .	96
113	4.2.5.8	Matrix_nvals: Number of stored elements in a matrix . . . . .	97
114	4.2.5.9	Matrix_build: Store elements from tuples into a matrix . . . . .	98
115	4.2.5.10	Matrix_setElement: Set a single element in matrix . . . . .	100
116	4.2.5.11	Matrix_removeElement: Remove an element from a matrix . . . . .	101
117	4.2.5.12	Matrix_extractElement: Extract a single element from a matrix . . .	103
118	4.2.5.13	Matrix_extractTuples: Extract tuples from a matrix . . . . .	105
119	4.2.5.14	Matrix_exportHint: Provide a hint as to which storage format might be most efficient for exporting a matrix . . . . .	107
120			
121	4.2.5.15	Matrix_exportSize: Return the array sizes necessary to export a GraphBLAS matrix object . . . . .	108
122			
123	4.2.5.16	Matrix_export: Export a GraphBLAS matrix to a pre-defined format	109
124	4.2.5.17	Matrix_import: Import a matrix into a GraphBLAS object . . . . .	111
125	4.2.5.18	Matrix_serializeSize: Compute the serialize buffer size . . . . .	112
126	4.2.5.19	Matrix_serialize: Serialize a GraphBLAS matrix. . . . .	113
127	4.2.5.20	Matrix_deserialize: Deserialize a GraphBLAS matrix. . . . .	114
128	4.2.6	Descriptor methods . . . . .	116
129	4.2.6.1	Descriptor_new: Create new descriptor . . . . .	116
130	4.2.6.2	Descriptor_set: Set content of descriptor . . . . .	116
131	4.2.7	free: Destroy an object and release its resources . . . . .	118
132	4.2.8	wait: Return once an object is either <i>complete</i> or <i>materialized</i> . . . . .	119
133	4.2.9	error: Retrieve an error string . . . . .	120
134	4.3	GraphBLAS operations . . . . .	121
135	4.3.1	mxm: Matrix-matrix multiply . . . . .	126
136	4.3.2	vxm: Vector-matrix multiply . . . . .	130
137	4.3.3	mxv: Matrix-vector multiply . . . . .	135
138	4.3.4	eWiseMult: Element-wise multiplication . . . . .	139
139	4.3.4.1	eWiseMult: Vector variant . . . . .	139

140	4.3.4.2	eWiseMult: Matrix variant . . . . .	144
141	4.3.5	eWiseAdd: Element-wise addition . . . . .	149
142	4.3.5.1	eWiseAdd: Vector variant . . . . .	149
143	4.3.5.2	eWiseAdd: Matrix variant . . . . .	154
144	4.3.6	extract: Selecting sub-graphs . . . . .	159
145	4.3.6.1	extract: Standard vector variant . . . . .	159
146	4.3.6.2	extract: Standard matrix variant . . . . .	164
147	4.3.6.3	extract: Column (and row) variant . . . . .	169
148	4.3.7	assign: Modifying sub-graphs . . . . .	174
149	4.3.7.1	assign: Standard vector variant . . . . .	174
150	4.3.7.2	assign: Standard matrix variant . . . . .	179
151	4.3.7.3	assign: Column variant . . . . .	184
152	4.3.7.4	assign: Row variant . . . . .	190
153	4.3.7.5	assign: Constant vector variant . . . . .	195
154	4.3.7.6	assign: Constant matrix variant . . . . .	200
155	4.3.8	apply: Apply a function to the elements of an object . . . . .	207
156	4.3.8.1	apply: Vector variant . . . . .	207
157	4.3.8.2	apply: Matrix variant . . . . .	211
158	4.3.8.3	apply: Vector-BinaryOp variants . . . . .	215
159	4.3.8.4	apply: Matrix-BinaryOp variants . . . . .	221
160	4.3.8.5	apply: Vector index unary operator variant . . . . .	227
161	4.3.8.6	apply: Matrix index unary operator variant . . . . .	232
162	4.3.9	select: . . . . .	237
163	4.3.9.1	select: Vector variant . . . . .	237
164	4.3.9.2	select: Matrix variant . . . . .	242
165	4.3.10	reduce: Perform a reduction across the elements of an object . . . . .	248
166	4.3.10.1	reduce: Standard matrix to vector variant . . . . .	248
167	4.3.10.2	reduce: Vector-scalar variant . . . . .	252
168	4.3.10.3	reduce: Matrix-scalar variant . . . . .	256
169	4.3.11	transpose: Transpose rows and columns of a matrix . . . . .	259

170	4.3.12 kronecker: Kronecker product of two matrices . . . . .	263
171	<b>5 Nonpolymorphic interface</b>	<b>269</b>
172	<b>A Revision history</b>	<b>283</b>
173	<b>B Non-opaque data format definitions</b>	<b>289</b>
174	B.1 GrB_Format: Specify the format for input/output of a GraphBLAS matrix. . . . .	289
175	B.1.1 GrB_CSR_FORMAT . . . . .	289
176	B.1.2 GrB_CSC_FORMAT . . . . .	290
177	B.1.3 GrB_COO_FORMAT . . . . .	290
178	<b>C Examples</b>	<b>291</b>
179	C.1 Example: Level breadth-first search (BFS) in GraphBLAS . . . . .	292
180	C.2 Example: Level BFS in GraphBLAS using apply . . . . .	293
181	C.3 Example: Parent BFS in GraphBLAS . . . . .	294
182	C.4 Example: Betweenness centrality (BC) in GraphBLAS . . . . .	295
183	C.5 Example: Batched BC in GraphBLAS . . . . .	297
184	C.6 Example: Maximal independent set (MIS) in GraphBLAS . . . . .	299
185	C.7 Example: Counting triangles in GraphBLAS . . . . .	301



# List of Tables

186		
187	2.1	Types of GraphBLAS opaque objects. . . . . 25
188	2.2	Methods that forced completion prior to GraphBLAS v2.0. . . . . 30
189	3.1	Enumeration literals and corresponding values input to various GraphBLAS methods. 34
190	3.2	Predefined GrB_Type values. . . . . 35
191	3.3	Operator input for relevant GraphBLAS operations. . . . . 36
192	3.4	Properties and recipes for building GraphBLAS algebraic objects. . . . . 37
193	3.5	Predefined unary and binary operators for GraphBLAS in C. . . . . 39
194	3.6	Predefined index unary operators for GraphBLAS in C. . . . . 40
195	3.7	Predefined monoids for GraphBLAS in C. . . . . 42
196	3.8	Predefined “true” semirings for GraphBLAS in C. . . . . 43
197	3.9	Other useful predefined semirings for GraphBLAS in C. . . . . 44
198	3.10	GrB_Format enumeration literals and corresponding values for matrix import and
199		export methods. . . . . 46
200	3.11	Descriptor types and literals for fields and values. . . . . 49
201	3.12	Predefined GraphBLAS descriptors. . . . . 50
202	3.13	Field values of type GrB_Field enumeration, corresponding types, and the objects
203		which must implement that GrB_Field. Collection refers to GrB_Matrix, GrB_Vector,
204		and GrB_Scalar, Algebraic refers to Operators, Monoids, and Semirings, Type refers to
205		GrB_Type, and Global refers to the GrB_Global context. All fields may be read, some
206		may be written (denoted by W), and some are hints (denoted by H) which may be
207		ignored by the implementation. For * see 3.7 . . . . . 53
208	3.14	Descriptions of select <i>field</i> , <i>value</i> pairs listed in 3.13 . . . . . 54
209	3.15	Field value enumerations. . . . . 55
210	3.16	Enumeration literals and corresponding values returned by GraphBLAS methods
211		and operations. . . . . 56

212	4.1	A mathematical notation for the fundamental GraphBLAS operations supported in	
213		this specification. . . . .	122
214	5.1	Long-name, nonpolymorphic form of GraphBLAS methods. . . . .	269
215	5.2	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	270
216	5.3	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	271
217	5.4	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	272
218	5.5	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	273
219	5.6	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	274
220	5.7	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	275
221	5.8	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	276
222	5.9	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	277
223	5.10	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	278
224	5.11	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	279
225	5.12	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	280
226	5.13	Long-name, nonpolymorphic form of GraphBLAS methods (continued). . . . .	281

# 227 **List of Figures**

228	3.1 Hierarchy of algebraic object classes in GraphBLAS. . . . .	45
229	4.1 Flowchart for the GraphBLAS operations. . . . .	123
230	B.1 Data layout for CSR format. . . . .	289
231	B.2 Data layout for CSC format. . . . .	290
232	B.3 Data layout for COO format. . . . .	290

## 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 2.1 are (in alphabetical order):

- Raye Kimmerer (MIT)
- Jim Kitchen (Anaconda)
- Manoj Kumar (IBM)
- Timothy G. Mattson (Human Learning Group)
- Erik Welch (Nvidia Corporation)

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

- Benjamin Brock (UC Berkeley)
- Aydın 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):

- Aydın 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

- 260     • Intel Corporation
- 261     • Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon Uni-  
262       versity for the operation of the Software Engineering Institute [DM-0003727, DM19-0929,  
263       DM21-0090]
- 264     • International Business Machines Corporation
- 265     • Nvidia Corporation
- 266     • Anaconda Inc.

267   The following people provided valuable input and feedback during the development of the spec-  
268   ification (in alphabetical order): David Bader, Hollen Barmer, Bob Cook, Tim Davis, Jeremy  
269   Kepner, Jim Kitchen, Peter Kogge, Manoj Kumar, Roi Lipman, Andrew Mellinger, Maxim Nau-  
270   mov, Nancy M. Ott, Michel Pelletier, Gabor Szarnyas, Ping Tak Peter Tang, Erik Welch, Michael  
271   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*

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

- 491 • *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as param-  
492 eters to a GraphBLAS method are dimension (or shape) compatible if they have the correct  
493 number of dimensions and sizes for each dimension to satisfy the rules of the mathematical def-  
494 inition of the operation associated with the method. If any *dimension compatibility* rule above  
495 is violated, execution of the GraphBLAS method ends and the GrB\_DIMENSION\_MISMATCH  
496 error is returned.
- 497 • *domain compatible*: Two domains for which values from one domain can be cast to values in  
498 the other domain as per the rules of the C language. In particular, domains from Table 3.2  
499 are all compatible with each other, and a domain from a user-defined type is only compatible  
500 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS  
501 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.
$\mathbf{D}_{out}(*), \mathbf{D}_{in}(*),$ $\mathbf{D}_{in_1}(*), \mathbf{D}_{in_2}(*)$	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 3.16. 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 3.16(a) are non-negative and include `GrB_SUCCESS` (a value of 0) and `GrB_NO_VALUE`.

An API error (listed in Table 3.16(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 3.16(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

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

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

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

762 After a call to any GraphBLAS method that modifies an opaque object, the program can re-  
763 trieve additional error information (beyond the error code returned by the method) though a call  
764 to the function `GrB_error()`, passing the method's output object as described in Section 4.2.9.  
765 The function returns a pointer to a NULL-terminated string, and the contents of that string are  
766 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error  
767 string. `GrB_error()` is a thread-safe function, in the sense that multiple threads can call it simul-  
768 taneously and each will get its own error string back, referring to the object passed as an input  
769 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:

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

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

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

### 806    3.3    Types (domains)

807    In GraphBLAS, domains correspond to the valid values for types from the host language (in our  
808    case, the C programming language). GraphBLAS defines a number of operators that take elements  
809    from one or more domains and produce elements of a (possibly) different domain. GraphBLAS  
810    also defines three kinds of collections: matrices, vectors and scalars. For any given collection, the  
811    elements of the collection belong to a *domain*, which is the set of valid values for the elements. For  
812    any variable or object  $V$  in GraphBLAS we denote as  $\mathbf{D}(V)$  the domain of  $V$ , that is, the set of  
813    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	GrB_Type_Code	Suffix	C type	Domain
-	GrB_UDT_CODE=0	UDT	-	-
GrB_BOOL	GrB_BOOL_CODE=1	BOOL	bool	{false, true}
GrB_INT8	GrB_INT8_CODE=2	INT8	int8_t	$\mathbb{Z} \cap [-2^7, 2^7)$
GrB_UINT8	GrB_UINT8_CODE=3	UINT8	uint8_t	$\mathbb{Z} \cap [0, 2^8)$
GrB_INT16	GrB_INT16_CODE=4	INT16	int16_t	$\mathbb{Z} \cap [-2^{15}, 2^{15})$
GrB_UINT16	GrB_UINT16_CODE=5	UINT16	uint16_t	$\mathbb{Z} \cap [0, 2^{16})$
GrB_INT32	GrB_INT32_CODE=6	INT32	int32_t	$\mathbb{Z} \cap [-2^{31}, 2^{31})$
GrB_UINT32	GrB_UINT32_CODE=7	UINT32	uint32_t	$\mathbb{Z} \cap [0, 2^{32})$
GrB_INT64	GrB_INT64_CODE=8	INT64	int64_t	$\mathbb{Z} \cap [-2^{63}, 2^{63})$
GrB_UINT64	GrB_UINT64_CODE=9	UINT64	uint64_t	$\mathbb{Z} \cap [0, 2^{64})$
GrB_FP32	GrB_FP32_CODE=10	FP32	float	IEEE 754 binary32
GrB_FP64	GrB_FP64_CODE=11	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. UDT stands for user-defined type and is the type code returned for all objects which use a non-predefined type. Implementations which add new types should start their `GrB_Type_Codes` at 100 to avoid possible conflicts with built-in types which may be added in the future.

### 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

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 monoid semiring (add)
eWiseMult	binary operator monoid semiring (times)
reduce (to vector or GrB_Scalar)	binary operator monoid
reduce (to scalar value)	monoid
apply	unary operator binary operator with scalar index unary operator
select	index unary operator
kronecker	binary operator monoid semiring
dup argument (build methods)	binary operator
accum argument (various methods)	binary operator

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 – 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

---

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.

---

852 define  $\mathbf{D}_{out}(F_u) = D_{out}$ ,  $\mathbf{D}_{in}(F_u) = D_{in}$ , and  $\mathbf{f}(F_u) = f$ .

853 A GraphBLAS *binary operator*  $F_b = \langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$  is defined by three domains,  $D_{out}$ ,  $D_{in_1}$ ,  
854  $D_{in_2}$ , and an operation  $\odot : D_{in_1} \times D_{in_2} \rightarrow D_{out}$ . For a given GraphBLAS binary operator  $F_b =$   
855  $\langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$ , 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  $\odot(F_b) =$   
856  $\odot$ . Note that  $\odot$  could be used in place of either  $\oplus$  or  $\otimes$  in other methods and operations.

857 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  
858 domains,  $D_{out}$ ,  $D_{in_1}$ ,  $D_{in_2}$ , the domain of GraphBLAS indices, and an operation  $f_i : D_{in_1} \times I_{U64}^2 \times$   
859  $D_{in_2} \rightarrow D_{out}$  (where  $I_{U64}$  corresponds to the domain of a `GrB_Index`). For a given GraphBLAS  
860 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  $\mathbf{f}(F_i) = f_i$ .

861 User-defined operators can be created with calls to `GrB_UnaryOp_new`, `GrB_BinaryOp_new`, and  
862 `GrB_IndexUnaryOp_new`, respectively. See Section 4.2.2 for information on these methods. The  
863 GraphBLAS C API predefines a number of these operators. These are listed in Tables 3.5 and 3.6.  
864 Note that most entries in these tables represent a “family” of predefined operators for a set of  
865 different types represented by the  $T$ ,  $I$ , or  $F$  in their names. For example, the multiplicative  
866 inverse (`GrB_MINV_F`) function is only defined for floating-point types ( $F = \text{FP32}$  or  $\text{FP64}$ ). The  
867 division (`GrB_DIV_T`) function is defined for all types, but only if  $y \neq 0$  for integral and floating  
868 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 Type	GraphBLAS identifier	Domains (– is don’t care) $A, u$ $i, j$ $s$ 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)
GrB_IndexUnaryOp	GrB_COLINDEX_ $I_{32/64}$	–	$I_{U64}$	$I_{32/64}$	$I_{32/64}$	$f(u_i, i, 0, s) = (i + s)$
GrB_IndexUnaryOp	GrB_DIAGINDEX_ $I_{32/64}$	–	$I_{U64}$	$I_{32/64}$	$I_{32/64}$	$f(A_{ij}, i, j, s) = (j + s)$ replace with its column index (+ s)
GrB_IndexUnaryOp	GrB_TRIL	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j - i + s)$ replace with its diagonal index (+ s)
GrB_IndexUnaryOp	GrB_TRIU	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j \leq i + s)$ triangle on or below diagonal s
GrB_IndexUnaryOp	GrB_DIAG	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j \geq i + s)$ triangle on or above diagonal s
GrB_IndexUnaryOp	GrB_OFFDIAG	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j == i + s)$ diagonal s
GrB_IndexUnaryOp	GrB_COLLE	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j \neq i + s)$ all but diagonal s
GrB_IndexUnaryOp	GrB_COLGT	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j \leq s)$ columns less or equal to s
GrB_IndexUnaryOp	GrB_ROWLE	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j > s)$ columns greater than s
GrB_IndexUnaryOp	GrB_ROWGT	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (i \leq s)$ , rows less or equal to s
GrB_IndexUnaryOp	GrB_ROWGT	–	$I_{U64}$	$I_{64}$	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
GrB_IndexUnaryOp	GrB_ROWGT	–	$I_{U64}$	$I_{64}$	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
GrB_IndexUnaryOp	GrB_VALUEEQ_ $T$	$T$	–	$T$	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
GrB_IndexUnaryOp	GrB_VALUENE_ $T$	$T$	–	$T$	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
GrB_IndexUnaryOp	GrB_VALUELT_ $T$	$T$	–	$T$	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
GrB_IndexUnaryOp	GrB_VALUELE_ $T$	$T$	–	$T$	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
GrB_IndexUnaryOp	GrB_VALUEGT_ $T$	$T$	–	$T$	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
GrB_IndexUnaryOp	GrB_VALUEGE_ $T$	$T$	–	$T$	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 identifier	Domains, $T$ ( $T \times T \rightarrow T$ )	Identity	Description
GrB_PLUS_MONOID_ $T$	UINT $x$ INT $x$ FP $x$	0 0 0	addition
GrB_TIMES_MONOID_ $T$	UINT $x$ INT $x$ FP $x$	1 1 1	multiplication
GrB_MIN_MONOID_ $T$	UINT $x$ INT $x$ FP $x$	UINT $x$ _MAX INT $x$ _MAX INFINITY	minimum
GrB_MAX_MONOID_ $T$	UINT $x$ INT $x$ FP $x$	0 INT $x$ _MIN -INFINITY	maximum
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, $T$ ( $T \times T \rightarrow T$ )	+ identity $\times$ annihilator	Description
<code>GrB_PLUS_TIMES_SEMIRING_T</code>	<code>UINT<math>x</math></code> <code>INT<math>x</math></code> <code>FP<math>x</math></code>	0 0 0	arithmetic semiring
<code>GrB_MIN_PLUS_SEMIRING_T</code>	<code>UINT<math>x</math></code> <code>INT<math>x</math></code> <code>FP<math>x</math></code>	<code>UINT<math>x</math>_MAX</code> <code>INT<math>x</math>_MAX</code> <code>INFINITY</code>	min-plus semiring
<code>GrB_MAX_PLUS_SEMIRING_T</code>	<code>INT<math>x</math></code> <code>FP<math>x</math></code>	<code>INT<math>x</math>_MIN</code> <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> <code>INT<math>x</math></code> <code>FP<math>x</math></code>	<code>UINT<math>x</math>_MAX</code> <code>INT<math>x</math>_MAX</code> <code>INFINITY</code>	min-max semiring
<code>GrB_MAX_MIN_SEMIRING_T</code>	<code>UINT<math>x</math></code> <code>INT<math>x</math></code> <code>FP<math>x</math></code>	0 <code>INT<math>x</math>_MIN</code> <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, $T$ ( $T \times T \rightarrow T$ )	+ 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
	$\text{INT}x$	$\text{INT}x\_MIN$	
	$\text{FP}x$	$-INFINITY$	

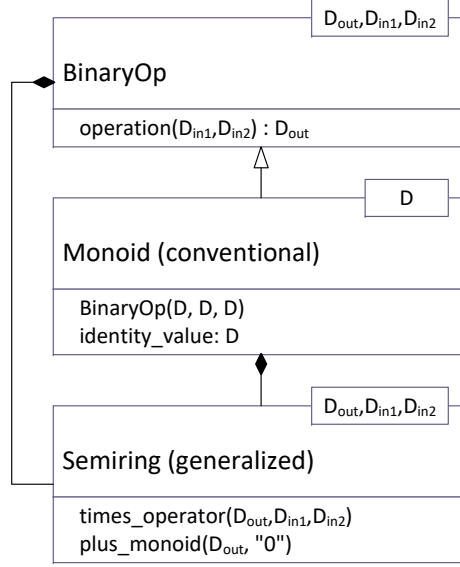


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

## 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 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 3.11.

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 3.12.

## 3.7 Fields

All GraphBLAS objects and implementations contain fields like those in the descriptor, which provide information to users and allow setting runtime parameters and hints. All GraphBLAS objects are required to implement the `GrB_get` and `GrB_set` methods required to query and set these fields. The library itself also contains several *(field, value)* pairs, which provide defaults to object level fields, and implementation information such as the version number or implementation name.

The required *value, field* pairs available for each object are defined in 3.13. Implementations may add their own custom `GrB_Field` enum values to extend the behavior of objects and methods. 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 by other operations, such as `GrB_BLOCKING_MODE` which is determined by `GrB_Init`.

`GrB_INVALID_VALUE` must be returned when attempting to write to fields which are read only.

The `GrB_Field` enumeration is defined by the values in Table 3.13, and selected values are described in Table 3.14.



---

Table 3.11: 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
<code>GrB_DEFAULT</code>	0	The default (unset) value for each field.
<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.
<code>GrB_COMP_STRUCTURE</code>	6	Shorthand for both <code>GrB_COMP</code> and <code>GrB_STRUCTURE</code> .

---

Table 3.12: 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

### 1019 3.7.1 Input Types

1020 Allowable types used in `GrB_get` and `GrB_set` are `INT32`, `GrB_Scalar`, `char*`, `void*`, and `SIZE`. Each  
1021 `GrB_Field` is associated with exactly one of these types as defined in Table 3.13. Implementations  
1022 that add additional `GrB_Fields` must document the type associated with each `GrB_Field`.

#### 1023 3.7.1.1 INT32 Handling

1024 `INT32` types use a 32-bit signed integer type. This can be used both for numeric values as well as  
1025 enumerated C types. Enumerated types must specify the numeric value for each enum, and the  
1026 value specified must fit within the allowable 32-bit signed integer range.

#### 1027 3.7.1.2 GrB\_Scalar Handling

1028 When calling `GrB_get`, the user must provide an already initialized `GrB_Scalar` object to which  
1029 the implementation will write a value of the correct element type. When calling `GrB_set`, the  
1030 `GrB_Scalar` must not be empty, otherwise a `GrB_EMPTY_OBJECT` error is raised.

#### 1031 3.7.1.3 String (char\*) Handling

1032 When the input to `GrB_set` is a `char*` the input array is null terminated. The `GraphBLAS` imple-  
1033 mentation must copy this array into internal data structures. Using `GrB_get` for strings requires  
1034 two calls. First, call `GrB_get` with the field and object, but pass `size_t*` as the value argument.  
1035 The implementation will return the size of the string buffer that the user must create. Second, call  
1036 `GrB_get` with the field and object, this time passing a pointer to the newly created string buffer.  
1037 The `GraphBLAS` implementation will write to this buffer, including a trailing null terminator. The  
1038 size returned in the first call will include enough bytes for the null terminator.

#### 1039 3.7.1.4 void\* Handling

1040 When the input to `GrB_set` is a `void*`, an extra `size_t` argument is passed to indicate the size of the  
1041 buffer. The `GraphBLAS` implementation must copy this many bytes from the buffer into internal  
1042 data structures. Similar to reading strings, `GrB_get` must be called twice for `void*`. The first call  
1043 passes `size_t*` to find the required buffer size. The user must create a buffer and then pass the  
1044 pointer to `GrB_get`. The implementation will write to this buffer. No standard specification or  
1045 protocol is required for the contents of `void*`. It is meant to be a mechanism to allow full freedom  
1046 for `GraphBLAS` implementations with needs that cannot be handled using `INT32`, `GrB_Scalar`, or  
1047 `Strings`.

#### 1048 3.7.1.5 SIZE Handling

1049 `SIZE` types use a `size_t` type. Normally, `SIZE` is used in conjunction with `char*` and `void*` to indicate  
1050 the buffer size. However, it can also be used when the actual return type is `size_t`, as is the case

1051 for the size of a Type.

### 1052 3.7.2 Hints

1053 Several fields are *hints* (marked H in Table 3.13). Hints are used to represent intended use cases  
1054 or best guesses, but do not determine strict behavior. When `GrB_set` is called with a hint, the  
1055 GraphBLAS implementation should return `GrB_SUCCESS`, but is free to use or ignore the hint.  
1056 When `GrB_get` is called, the implementation should return a best guess on the correct answer. If  
1057 there is no clear answer, the implementation should return `GrB_UNKNOWN`.

### 1058 3.7.3 GrB\_NAME

1059 The `GrB_NAME` field is a special case regarding writability. All user-defined objects have a  
1060 `GrB_NAME` field which defaults to an empty string. Collections and `GrB_Descriptors` may have  
1061 their `GrB_NAME` set at any time. User-defined algebraic objects and `GrB_Types` may only have  
1062 their `GrB_NAME` set once to a globally unique value. Attempting to set this field after it has  
1063 already been set will return a `GrB_ALREADY_SET` error code.

1064 Built-in algebraic objects and `GrB_Types` have names which can be read, but not written to. The  
1065 name returned will be the string form of the `GrB_Type` listed in Table 3.2 or the GraphBLAS  
1066 identifier listed in Tables 3.5, 3.6, 3.7, 3.8, and 3.9. For example, the name of `GrB_BOOL` type  
1067 is "`GrB_BOOL`" (8 characters) and the name of `GrB_MIN_FP64` binary op is "`GrB_MIN_FP64`" (12  
1068 characters).

1069 The `GrB_NAME` of the global context is read-only and returns the name of the library implemen-  
1070 tation.

Table 3.13: 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, Type refers to GrB\_Type, and Global refers to the GrB\_Global context. 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. For \* see 3.7

Field Name	W	H	Value	Implementing Objects	Type
GrB_OUTP_FIELD	W	—	0	GrB_Descriptor	INT32 (GrB_Desc_Value)
GrB_MASK_FIELD	W	—	1	GrB_Descriptor	INT32 (GrB_Desc_Value)
GrB_INP0_FIELD	W	—	2	GrB_Descriptor	INT32 (GrB_Desc_Value)
GrB_INP1_FIELD	W	—	3	GrB_Descriptor	INT32 (GrB_Desc_Value)
GrB_NAME	*		10	Global, Collection, Algebraic, Type	Null terminated char*
GrB_LIBRARY_VER_MAJOR	—	—	11	Global	INT32
GrB_LIBRARY_VER_MINOR	—	—	12	Global	INT32
GrB_LIBRARY_VER_PATCH	—	—	13	Global	INT32
GrB_API_VER_MAJOR	—	—	14	Global	INT32
GrB_API_VER_MINOR	—	—	15	Global	INT32
GrB_API_VER_PATCH	—	—	16	Global	INT32
GrB_BLOCKING_MODE	—	—	17	Global	INT32 (GrB_Mode)
GrB_STORAGE_ORIENTATION_HINT	W	H	100	Global, Collection	INT32 (GrB_Orientation)
GrB_ELTYPE_CODE	—	—	102	Collection, Type	INT32 (GrB_Type_Code)
GrB_INPUT1TYPE_CODE	—	—	103	Algebraic	INT32 (GrB_Type_Code)
GrB_INPUT2TYPE_CODE	—	—	104	Algebraic	INT32 (GrB_Type_Code)
GrB_OUTPUTTYPE_CODE	—	—	105	Algebraic	INT32 (GrB_Type_Code)
GrB_ELTYPE_STRING	—	—	106	Collection, Type	Null terminated char*
GrB_INPUT1TYPE_STRING	—	—	107	Algebraic	Null terminated char*
GrB_INPUT2TYPE_STRING	—	—	108	Algebraic	Null terminated char*
GrB_OUTPUTTYPE_STRING	—	—	109	Algebraic	Null terminated char*
GrB_SIZE	—	—	110	Type	SIZE

Table 3.14: Descriptions of select *field*, *value* pairs listed in 3.13

Field Name	Description
GrB_NAME	The name of any GraphBLAS object, or the name of the library implementation.
GrB_BLOCKING_MODE	The blocking mode as set by GrB_init
GrB_STORAGE_ORIENTATION_HINT	Hint to the library that a collection is best stored in a row (lexicographic) or column (colexicographic) major format.
GrB_ELTYPE_(CODE/STRING)	The element type of a collection.
GrB_INPUT1TYPE_(CODE/STRING)	The type of the first argument to an operator. Returns GrB_NO_VALUE for Semirings and IndexUnaryOps which depend only on the index.
GrB_INPUT2TYPE_(CODE/STRING)	The type of the second argument to an operator. Returns GrB_NO_VALUE for Semirings, UnaryOps, and IndexUnaryOps which depend only on the index.
GrB_OUTPUTTYPE_(CODE/STRING)	The type of the output of an operator.
GrB_SIZE	The size of the GrB_Type.

### 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 3.16.

---

Table 3.15: Enumerations not defined elsewhere in the documents and used when getting or setting fields are defined in the following tables.

(a) Field values of type GrB\_Orientation.

Value Name	Value	Description
GrB_ROWMAJOR	0	The majority of iteration over the object will be row-wise.
GrB_COLMAJOR	1	The majority of iteration over the object will be column-wise.
GrB_BOTH	2	Iteration may occur with equal frequency in both directions.
GrB_UNKNOWN	3	No indication is given or is unknown.

---

Table 3.16: 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.
GrB_ALREADY_SET	-9	An attempt was made to write to a field which may only be written to once.

(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`.

## 1093 **Return Values**

1094                   GrB\_SUCCESS operation completed successfully.

1095                   GrB\_PANIC unknown internal error.

1096                   GrB\_INVALID\_VALUE invalid mode specified, or method called multiple times.

## 1097 **Description**

1098 The init method creates and initializes a GraphBLAS C API context. The argument to GrB\_init  
1099 defines the mode for the context. The two available modes are:

- 1100       • GrB\_BLOCKING: In this mode, each method in a sequence returns after its computations have  
1101        completed and output arguments are available to subsequent statements in an application.  
1102        When executing in GrB\_BLOCKING mode, the methods execute in program order.
- 1103       • GrB\_NONBLOCKING: In this mode, methods in a sequence may return after arguments in  
1104        the method have been tested for dimension and domain compatibility within the method  
1105        but potentially before their computations complete. Output arguments are available to sub-  
1106        sequent GraphBLAS methods in an application. When executing in GrB\_NONBLOCKING  
1107        mode, the methods in a sequence may execute in any order that preserves the mathematical  
1108        result defined by the sequence.

1109 An application can only create one context per execution instance. An application may only call  
1110 GrB\_Init once. Calling GrB\_Init more than once results in undefined behavior.

## 1111 **4.1.2 finalize: Finalize a GraphBLAS context**

1112 Terminates and frees any internal resources created to support the GraphBLAS C API context.

## 1113 **C Syntax**

1114                   GrB\_Info GrB\_finalize();

## 1115 **Return Values**

1116                   GrB\_SUCCESS operation completed successfully.

1117                   GrB\_PANIC unknown internal error.

## 1118 **Description**

1119 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS  
1120 C API context. `GrB_finalize` may only be called after a context has been initialized by calling  
1121 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-  
1122 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined  
1123 behavior.

### 1124 **4.1.3 getVersion: Get the version number of the standard.**

1125 Query the library for the version number of the standard that this library implements.

## 1126 **C Syntax**

```
1127         GrB_Info GrB_getVersion(unsigned int *version,  
1128                                unsigned int *subversion);
```

## 1129 **Parameters**

1130 version (OUT) On successful return will hold the value of the major version number.

1131 version (OUT) On successful return will hold the value of the subversion number.

## 1132 **Return Values**

1133 GrB\_SUCCESS operation completed successfully.

1134 GrB\_PANIC unknown internal error.

## 1135 **Description**

1136 The `getVersion` method is used to query the major and minor version number of the GraphBLAS  
1137 C API specification that the library implements at runtime. To support compile time queries the  
1138 following two macros shall also be defined by the library.

```
1139         #define GRB_VERSION      2  
1140         #define GRB_SUBVERSION  0
```

## 1141 **4.2 Object methods**

1142 This section describes methods that setup and operate on GraphBLAS opaque objects but are not  
1143 part of the the GraphBLAS math specification.

## 1144 4.2.1 Get and Set methods

1145 The methods in this section query and, optionally, set internal fields of GraphBLAS objects.

### 1146 4.2.1.1 get: Query the value of an object

#### 1147 C Syntax

```
1148 GrB_Info GrB_get(GrB_<OBJ> o, <type> value, GrB_Field field);
```

#### 1149 Parameters

1150 OBJ (IN) An existing, valid GraphBLAS object (collection, operation, type) which is  
1151 being queried. To indicate the global context, the constant `GrB_Global` is used.

1152 value (OUT) A pointer to or `GrB_Scalar` containing a value whose type is dependent on  
1153 field which will be filled with the current value of the field. type may be `int32_t*`,  
1154 `size_t*`, `GrB_Scalar`, `char*` or `void*`.

1155 field (IN) The field being queried.

#### 1156 Return Value

1157 `GrB_SUCCESS` The method completed successfully.

1158 `GrB_PANIC` unknown internal error.

1159 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1160 `GrB_UNINITIALIZED_OBJECT` the value parameter is `GrB_Scalar` and has not been initialized by  
1161 a call to `new`.

1162 `GrB_INVALID_VALUE` invalid value type provided for the field or invalid field.

#### 1163 Description

1164 Queries a field of an existing GraphBLAS object. The type of the argument is uniquely determined  
1165 by field. For the case of `char*` and `void*`, the value can be replaced with `size_t*` to get the required  
1166 buffer size to hold the response. Fields marked as hints in Table 3.13 will return a hint on how  
1167 best to use the object.

### 1168 4.2.1.2 set: Set field of an object

1169 Set the content for a field for an existing GraphBLAS object.

## 1170 C Syntax

```
1171     GrB_Info GrB_set(GrB_<OBJ> o, <type> value, GrB_Field field);
1172     GrB_Info GrB_set(GrB_<OBJ> o, void *value, GrB_Field field, size_t voidSize);
```

## 1173 Parameters

1174 OBJ (IN) The GraphBLAS object which is having field set. To indicate  
1175 the global context, the constant GrB\_Global is used.

1176 value (IN) A value whose type is dependent on field. type may be a  
1177 int32\_t, GrB\_Scalar, char\* or void\*.

1178 field (IN) The field being set.

1179 voidSize (IN) The size of the void\* buffer. Note that a size is not needed for  
1180 char\* because the string is assumed null-terminated.

## 1181 Return Values

1182 GrB\_SUCCESS The method completed successfully.

1183 GrB\_PANIC unknown internal error.

1184 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1185 GrB\_UNINITIALIZED\_OBJECT the GrB\_Scalar parameter has not been initialized by a call to new.

1186 GrB\_INVALID\_VALUE invalid value set on the field, invalid field, or field is read-only.

1187 GrB\_ALREADY\_SET this field has already been set, and may only be set once.

## 1188 Description

1189 Set a field of OBJ or the Global context to a new value.

## 1190 4.2.2 Algebra methods

### 1191 4.2.2.1 Type\_new: Construct a new GraphBLAS (user-defined) type

1192 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,  
1193 monoids, semirings, vectors and matrices.

## 1194 C Syntax

```
1195         GrB_Info GrB_Type_new(GrB_Type  *utype,  
1196                               size_t     sizeof(ctype));
```

## 1197 Parameters

1198 utype (INOUT) On successful return, contains a handle to the newly created user-defined  
1199 GraphBLAS type object.

1200 ctype (IN) A C type that defines the new GraphBLAS user-defined type.

## 1201 Return Values

1202 GrB\_SUCCESS operation completed successfully.

1203 GrB\_PANIC unknown internal error.

1204 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1205 GrB\_NULL\_POINTER utype pointer is NULL.

## 1206 Description

1207 Given a C type ctype, the Type\_new method returns in utype a handle to a new GraphBLAS type  
1208 that is equivalent to the C type. Variables of this ctype must be a struct, union, or fixed-size array.  
1209 In particular, given two variables, src and dst, of type ctype, the following operation must be a  
1210 valid way to copy the contents of src to dst:

```
1211         memcpy(&dst, &src, sizeof(ctype))
```

1212 A new, user-defined type utype should be destroyed with a call to GrB\_free(utype) when no longer  
1213 needed.

1214 It is not an error to call this method more than once on the same variable; however, the handle to  
1215 the previously created object will be overwritten.

### 1216 4.2.2.2 UnaryOp\_new: Construct a new GraphBLAS unary operator

1217 Initializes a new GraphBLAS unary operator with a specified user-defined function and its types  
1218 (domains).

## 1219 C Syntax

```
1220      GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1221                               void          (*unary_func)(void*, const void*),  
1222                               GrB_Type      d_out,  
1223                               GrB_Type      d_in);
```

## 1224 Parameters

1225     **unary\_op** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1226     unary operator object.

1227     **unary\_func** (IN) a pointer to a user-defined function that takes one input parameter of **d\_in**'s  
1228     type and returns a value of **d\_out**'s type, both passed as **void** pointers. Specifically  
1229     the signature of the function is expected to be of the form:

```
1230           void func(void *out, const void *in);
```

1232     **d\_out** (IN) The **GrB\_Type** of the return value of the unary operator being created. Should  
1233     be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1234     BLAS type.

1235     **d\_in** (IN) The **GrB\_Type** of the input argument of the unary operator being created.  
1236     Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1237     GraphBLAS type.

## 1238 Return Values

1239     **GrB\_SUCCESS** operation completed successfully.

1240     **GrB\_PANIC** unknown internal error.

1241     **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1242     **GrB\_UNINITIALIZED\_OBJECT** any **GrB\_Type** parameter (for user-defined types) has not been ini-  
1243     tialized by a call to **GrB\_Type\_new**.

1244     **GrB\_NULL\_POINTER** **unary\_op** or **unary\_func** pointers are **NULL**.

## 1245 Description

1246     The **UnaryOp\_new** method creates a new GraphBLAS unary operator

1247      $f_u = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in}), \text{unary\_func} \rangle$

1248 and returns a handle to it in `unary_op`.

1249 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments  
1250 are aliased. In other words, for all invocations of the function:

```
1251     unary_func(out,in);
```

1252 the value of `out` must be the same as if the following code was executed:

```
1253     D(d_in) *tmp = malloc(sizeof(D(d_in)));
1254     memcpy(tmp,in,sizeof(D(d_in)));
1255     unary_func(out,tmp);
1256     free(tmp);
```

1257 It is not an error to call this method more than once on the same variable; however, the handle to  
1258 the previously created object will be overwritten.

#### 1259 4.2.2.3 BinaryOp\_new: Construct a new GraphBLAS binary operator

1260 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types  
1261 (domains).

### 1262 C Syntax

```
1263     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,
1264                               void          (*binary_func)(void*,
1265                               const void*,
1266                               const void*),
1267                               GrB_Type      d_out,
1268                               GrB_Type      d_in1,
1269                               GrB_Type      d_in2);
```

### 1270 Parameters

1271 `binary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1272 binary operator object.

1273 `binary_func` (IN) A pointer to a user-defined function that takes two input parameters of types  
1274 `d_in1` and `d_in2` and returns a value of type `d_out`, all passed as void pointers.  
1275 Specifically the signature of the function is expected to be of the form:

```
1276     void func(void *out, const void *in1, const void *in2);
1277
```



1278        **d\_out** (IN) The **GrB\_Type** of the return value of the binary operator being created. Should  
1279        be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1280        BLAS type.

1281        **d\_in1** (IN) The **GrB\_Type** of the left hand argument of the binary operator being created.  
1282        Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1283        GraphBLAS type.

1284        **d\_in2** (IN) The **GrB\_Type** of the right hand argument of the binary operator being cre-  
1285        ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-  
1286        defined GraphBLAS type.

## 1287    **Return Values**

1288                **GrB\_SUCCESS** operation completed successfully.

1289                **GrB\_PANIC** unknown internal error.

1290                **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1291    **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_Type** (for user-defined types) has not been initialized by a  
1292                call to **GrB\_Type\_new**.

1293                **GrB\_NULL\_POINTER** **binary\_op** or **binary\_func** pointer is **NULL**.

## 1294    **Description**

1295    The **BinaryOp\_new** methods creates a new GraphBLAS binary operator

1296                 $f_b = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in1}), \mathbf{D}(\mathbf{d\_in2}), \mathbf{binary\_func} \rangle$

1297    and returns a handle to it in **binary\_op**.

1298    The implementation of **binary\_func** must be such that it works even if any of the **d\_out**, **d\_in1**, and  
1299    **d\_in2** arguments are aliased to each other. In other words, for all invocations of the function:

1300                **binary\_func**(**out**, **in1**, **in2**);

1301    the value of **out** must be the same as if the following code was executed:

```
1302        D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));  
1303        D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));  
1304        memcpy(tmp1, in1, sizeof(D(d_in1)));  
1305        memcpy(tmp2, in2, sizeof(D(d_in2)));  
1306        binary_func(out, tmp1, tmp2);  
1307        free(tmp2);  
1308        free(tmp1);
```

1309 It is not an error to call this method more than once on the same variable; however, the handle to  
1310 the previously created object will be overwritten.

#### 1311 4.2.2.4 Monoid\_new: Construct a new GraphBLAS monoid

1312 Creates a new monoid with specified binary operator and identity value.

#### 1313 C Syntax

```
1314         GrB_Info GrB_Monoid_new(GrB_Monoid    *monoid,  
1315                                GrB_BinaryOp    binary_op,  
1316                                <type>         identity);
```

#### 1317 Parameters

1318 monoid (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1319 monoid object.

1320 binary\_op (IN) An existing GraphBLAS associative binary operator whose input and output  
1321 types are the same.

1322 identity (IN) The value of the identity element of the monoid. Must be the same type as  
1323 the type used by the **binary\_op** operator.

#### 1324 Return Values

1325 GrB\_SUCCESS operation completed successfully.

1326 GrB\_PANIC unknown internal error.

1327 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1328 GrB\_UNINITIALIZED\_OBJECT the GrB\_BinaryOp (for user-defined operators) has not been initial-  
1329 ized by a call to GrB\_BinaryOp\_new.

1330 GrB\_NULL\_POINTER monoid pointer is NULL.

1331 GrB\_DOMAIN\_MISMATCH all three argument types of the binary operator and the type of the  
1332 identity value are not the same.

#### 1333 Description

1334 The **Monoid\_new** method creates a new monoid  $M = \langle \mathbf{D}(\text{binary\_op}), \text{binary\_op}, \text{identity} \rangle$  and re-  
1335 turns a handle to it in **monoid**.

1336 If `binary_op` is not associative, the results of GraphBLAS operations that require associativity of  
1337 this monoid will be undefined.

1338 It is not an error to call this method more than once on the same variable; however, the handle to  
1339 the previously created object will be overwritten.

#### 1340 4.2.2.5 Semiring\_new: Construct a new GraphBLAS semiring

1341 Creates a new semiring with specified domain, operators, and elements.

#### 1342 C Syntax

```
1343         GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1344                                 GrB_Monoid    add_op,  
1345                                 GrB_BinaryOp   mul_op);
```

#### 1346 Parameters

1347 `semiring` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1348 `semiring`.

1349 `add_op` (IN) An existing GraphBLAS commutative monoid that specifies the addition op-  
1350 erator and its identity.

1351 `mul_op` (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-  
1352 plication operator. In addition, `mul_op`'s output domain,  $\mathbf{D}_{out}(\text{mul\_op})$ , must be  
1353 the same as the `add_op`'s domain  $\mathbf{D}(\text{add\_op})$ .

#### 1354 Return Values

1355 `GrB_SUCCESS` operation completed successfully.

1356 `GrB_PANIC` unknown internal error.

1357 `GrB_OUT_OF_MEMORY` not enough memory available for this method to complete.

1358 `GrB_UNINITIALIZED_OBJECT` the `add_op` (for user-define monoids) object has not been initialized  
1359 with a call to `GrB_Monoid_new` or the `mul_op` (for user-defined  
1360 operators) object has not been not been initialized by a call to  
1361 `GrB_BinaryOp_new`.

1362 `GrB_NULL_POINTER` `semiring` pointer is NULL.

1363 `GrB_DOMAIN_MISMATCH` the output domain of `mul_op` does not match the domain of the  
1364 `add_op` monoid.

## 1365 Description

1366 The `Semiring_new` method creates a new semiring:

1367  $S = \langle \mathbf{D}_{out}(\text{mul\_op}), \mathbf{D}_{in_1}(\text{mul\_op}), \mathbf{D}_{in_2}(\text{mul\_op}), \text{add\_op}, \text{mul\_op}, \mathbf{0}(\text{add\_op}) \rangle$

1368 and returns a handle to it in `semiring`. Note that  $\mathbf{D}_{out}(\text{mul\_op})$  must be the same as  $\mathbf{D}(\text{add\_op})$ .

1369 If `add_op` is not commutative, then GraphBLAS operations using this semiring will be undefined.

1370 It is not an error to call this method more than once on the same variable; however, the handle to  
1371 the previously created object will be overwritten.

## 1372 4.2.2.6 IndexUnaryOp\_new: Construct a new GraphBLAS index unary operator

1373 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its  
1374 types (domains).

## 1375 C Syntax

```
1376 GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp *index_unary_op,  
1377                               void (*index_unary_func)(void*,  
1378                                                         const void*,  
1379                                                         GrB_Index,  
1380                                                         GrB_Index,  
1381                                                         const void*),  
1382                               GrB_Type d_out,  
1383                               GrB_Type d_in1,  
1384                               GrB_Type d_in2);
```

## 1385 Parameters

1386 `index_unary_op` (INOUT) On successful return, contains a handle to the newly created Graph-  
1387 BLAS index unary operator object.

1388 `index_unary_func` (IN) A pointer to a user-defined function that takes input parameters of types  
1389 `d_in1`, `GrB_Index`, `GrB_Index` and `d_in2` and returns a value of type `d_out`. Ex-  
1390 cept for the `GrB_Index` parameters, all are passed as `void` pointers. Specifically  
1391 the signature of the function is expected to be of the form:

```
1392 void func(void *out,  
1393           const void *in1,  
1394           GrB_Index row_index,  
1395           GrB_Index col_index,  
1396           const void *in2);  
1397
```

1398           d\_out (IN) The GrB\_Type of the return value of the index unary operator being created.  
 1399           Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
 1400           GraphBLAS type.

1401           d\_in1 (IN) The GrB\_Type of the first input argument of the index unary operator being  
 1402           created and corresponds to the stored values of the GrB\_Vector or GrB\_Matrix  
 1403           being operated on. Should be one of the predefined GraphBLAS types in Ta-  
 1404           ble 3.2, or a user-defined GraphBLAS type.

1405           d\_in2 (IN) The GrB\_Type of the last input argument of the index unary operator be-  
 1406           ing created and corresponds to a scalar provided by the GraphBLAS operation  
 1407           that uses this operator. Should be one of the predefined GraphBLAS types in  
 1408           Table 3.2, or a user-defined GraphBLAS type.

## 1409   Return Values

1410           GrB\_SUCCESS operation completed successfully.

1411           GrB\_PANIC unknown internal error.

1412           GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1413   GrB\_UNINITIALIZED\_OBJECT the GrB\_Type (for user-defined types) has not been initialized by a  
 1414           call to GrB\_Type\_new.

1415           GrB\_NULL\_POINTER index\_unary\_op or index\_unary\_func pointer is NULL.

## 1416   Description

1417   The IndexUnaryOp\_new methods creates a new GraphBLAS index unary operator

1418            $f_i = \langle \mathbf{D}(d\_out), \mathbf{D}(d\_in1), \mathbf{D}(GrB\_Index), \mathbf{D}(GrB\_Index), \mathbf{D}(d\_in2), index\_unary\_func \rangle$

1419   and returns a handle to it in index\_unary\_op.

1420   The implementation of index\_unary\_func must be such that it works even if any of the d\_out,  
 1421   d\_in1, and d\_in2 arguments are aliased to each other. In other words, for all invocations of the  
 1422   function:

1423           index\_unary\_func(out,in1,row\_index,col\_index,n,in2);

1424   the value of out must be the same as if the following code was executed (shown here for matrices):

```
1425       GrB_Index row_index = ...;
1426       GrB_Index col_index = ...;
1427       D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
```

```

1428     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1429     memcpy(tmp1,in1,sizeof(D(d_in1)));
1430     memcpy(tmp2,in2,sizeof(D(d_in2)));
1431     index_unary_func(out,tmp1,row_index,col_index,tmp2);
1432     free(tmp2);
1433     free(tmp1);

```

1434 It is not an error to call this method more than once on the same variable; however, the handle to  
1435 the previously created object will be overwritten.

### 1436 4.2.3 Scalar methods

#### 1437 4.2.3.1 Scalar\_new: Construct a new scalar

1438 Creates a new empty scalar with specified domain.

#### 1439 C Syntax

```

1440     GrB_Info GrB_Scalar_new(GrB_Scalar *s,
1441                             GrB_Type    d);

```

#### 1442 Parameters

1443 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1444 scalar.

1445 **d** (IN) The type corresponding to the domain of the scalar being created. Can be  
1446 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1447 GraphBLAS type.

#### 1448 Return Values

1449 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1450 blocking mode, this indicates that the API checks for the input  
1451 arguments passed successfully. Either way, output scalar **s** is ready  
1452 to be used in the next method of the sequence.

1453 **GrB\_PANIC** Unknown internal error.

1454 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1455 GraphBLAS objects (input or output) is in an invalid state caused  
1456 by a previous execution error. Call **GrB\_error()** to access any error  
1457 messages generated by the implementation.

1458        GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1459 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
1460                                (needed for user-defined types).

1461        GrB\_NULL\_POINTER The s pointer is NULL.

## 1462 Description

1463 Creates a new GraphBLAS scalar  $s$  of domain  $\mathbf{D}(d)$  and empty  $\mathbf{L}(s)$ . The method returns a handle  
1464 to the new scalar in  $s$ .

1465 It is not an error to call this method more than once on the same variable; however, the handle to  
1466 the previously created object will be overwritten.

### 1467 4.2.3.2 Scalar\_dup: Construct a copy of a GraphBLAS scalar

1468 Creates a new scalar with the same domain and contents as another scalar.

## 1469 C Syntax

```
1470            GrB_Info GrB_Scalar_dup(GrB_Scalar        *t,  
1471                                    const GrB_Scalar    s);
```

## 1472 Parameters

1473        t (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1474        scalar.

1475        s (IN) The GraphBLAS scalar to be duplicated.

## 1476 Return Values

1477        GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1478        blocking mode, this indicates that the API checks for the input  
1479        arguments passed successfully. Either way, output scalar  $t$  is ready  
1480        to be used in the next method of the sequence.

1481        GrB\_PANIC Unknown internal error.

1482        GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1483        GraphBLAS objects (input or output) is in an invalid state caused  
1484        by a previous execution error. Call GrB\_error() to access any error  
1485        messages generated by the implementation.

1486        GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1487 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to  
 1488        Scalar\_new or Scalar\_dup.

1489        GrB\_NULL\_POINTER The *t* pointer is NULL.

## 1490 Description

1491 Creates a new scalar *t* of domain  $\mathbf{D}(s)$  and contents  $\mathbf{L}(s)$ . The method returns a handle to the new  
 1492 scalar in *t*.

1493 It is not an error to call this method more than once with the same output variable; however, the  
 1494 handle to the previously created object will be overwritten.

### 1495 4.2.3.3 Scalar\_clear: Clear/remove a stored value from a scalar

1496 Removes the stored value from a scalar.

## 1497 C Syntax

1498        GrB\_Info GrB\_Scalar\_clear(GrB\_Scalar s);

## 1499 Parameters

1500        *s* (INOUT) An existing GraphBLAS scalar to clear.

## 1501 Return Values

1502        GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 1503        blocking mode, this indicates that the API checks for the input  
 1504        arguments passed successfully. Either way, output scalar *s* is ready  
 1505        to be used in the next method of the sequence.

1506        GrB\_PANIC Unknown internal error.

1507        GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1508        GraphBLAS objects (input or output) is in an invalid state caused  
 1509        by a previous execution error. Call GrB\_error() to access any error  
 1510        messages generated by the implementation.

1511        GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1512 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to  
 1513        Scalar\_new or Scalar\_dup.



1514 **Description**

1515 Removes the stored value from an existing scalar. After the call, **L(s)** is empty. The size of the  
1516 scalar does not change.

1517 **4.2.3.4 Scalar\_nvals: Number of stored elements in a scalar**

1518 Retrieve the number of stored elements in a scalar (either zero or one).

1519 **C Syntax**

```
1520         GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,  
1521                                   const GrB_Scalar s);
```

1522 **Parameters**

1523 nvals (OUT) On successful return, this is set to the number of stored elements in the  
1524 scalar (zero or one).

1525 s (IN) An existing GraphBLAS scalar being queried.

1526 **Return Values**

1527 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1528 cessfully and the value of nvals has been set.

1529 GrB\_PANIC Unknown internal error.

1530 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1531 GraphBLAS objects (input or output) is in an invalid state caused  
1532 by a previous execution error. Call GrB\_error() to access any error  
1533 messages generated by the implementation.

1534 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1535 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, s, has not been initialized by a call to  
1536 Scalar\_new or Scalar\_dup.

1537 GrB\_NULL\_POINTER The nvals pointer is NULL.

1538 **Description**

1539 Return **nvals(s)** in nvals. This is the number of stored elements in scalar s, which is the size of  
1540 **L(s)**, and can only be either zero or one (see Section 3.5.1).

#### 1541 4.2.3.5 Scalar\_setElement: Set the single element in a scalar

1542 Set the single element of a scalar to a given value.

#### 1543 C Syntax

```
1544         GrB_Info GrB_Scalar_setElement(GrB_Scalar    s,  
1545                                         <type>      val);
```

#### 1546 Parameters

1547 s (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1548 val (IN) Scalar value to assign. The type must be compatible with the domain of s.

#### 1549 Return Values

1550 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1551 blocking mode, this indicates that the compatibility tests on in-  
1552 dex/dimensions and domains for the input arguments passed suc-  
1553 cessfully. Either way, the output scalar s is ready to be used in the  
1554 next method of the sequence.

1555 GrB\_PANIC Unknown internal error.

1556 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1557 GraphBLAS objects (input or output) is in an invalid state caused  
1558 by a previous execution error. Call GrB\_error() to access any error  
1559 messages generated by the implementation.

1560 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1561 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, s, has not been initialized by a call to  
1562 Scalar\_new or Scalar\_dup.

1563 GrB\_DOMAIN\_MISMATCH The domains of s and val are incompatible.

#### 1564 Description

1565 First, val and output GraphBLAS scalar are tested for domain compatibility as follows: **D**(val) must  
1566 be compatible with **D**(s). Two domains are compatible with each other if values from one domain  
1567 can be cast to values in the other domain as per the rules of the C language. In particular, domains  
1568 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-  
1569 patible with itself. If any compatibility rule above is violated, execution of GrB\_Scalar\_setElement  
1570 ends and the domain mismatch error listed above is returned.

1571 We are now ready to carry out the assignment `val`; that is:

1572  $s(0) = \text{val}$

1573 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1574 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1575 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1576 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be  
1577 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1578 **4.2.3.6 Scalar\_extractElement: Extract a single element from a scalar.**

1579 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

### 1580 **C Syntax**

```
1581      GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1582                                         const GrB_Scalar s);
```

### 1583 **Parameters**

1584 `val` (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain  
1585 of scalar `s`. On successful return, `val` holds the result of the operation, and any  
1586 previous value in `val` is overwritten.

1587 `s` (IN) The GraphBLAS scalar from which an element is extracted.

### 1588 **Return Values**

1589 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-  
1590 cessfully. This indicates that the compatibility tests on dimensions  
1591 and domains for the input arguments passed successfully, and the  
1592 output scalar, `val`, has been computed and is ready to be used in  
1593 the next method of the sequence.

1594 `GrB_PANIC` Unknown internal error.

1595 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
1596 GraphBLAS objects (input or output) is in an invalid state caused  
1597 by a previous execution error. Call `GrB_error()` to access any error  
1598 messages generated by the implementation.

1599 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1600 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, `s`, has not been initialized by a call to  
1601 `Scalar_new` or `Scalar_dup`.

1602 GrB\_NULL\_POINTER `val` pointer is NULL.

1603 GrB\_DOMAIN\_MISMATCH The domains of the scalar or scalar are incompatible.

1604 GrB\_NO\_VALUE There is no stored value in the scalar.

## 1605 Description

1606 First, `val` and input GraphBLAS scalar are tested for domain compatibility as follows: `D(val)`  
1607 must be compatible with `D(s)`. Two domains are compatible with each other if values from  
1608 one domain can be cast to values in the other domain as per the rules of the C language. In  
1609 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1610 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1611 `GrB_Scalar_extractElement` ends and the domain mismatch error listed above is returned.

1612 Then, if no value is currently stored in the GraphBLAS scalar, the method returns `GrB_NO_VALUE`  
1613 and `val` remains unchanged.

1614 Finally the extract into the output argument, `val` can be performed; that is:

1615 
$$val = s(0)$$

1616 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value  
1617 `GrB_SUCCESS`, the new contents of `val` are as defined above.

## 1618 4.2.4 Vector methods

### 1619 4.2.4.1 Vector\_new: Construct new vector

1620 Creates a new vector with specified domain and size.

## 1621 C Syntax

```
1622 GrB_Info GrB_Vector_new(GrB_Vector *v,  
1623                          GrB_Type   d,  
1624                          GrB_Index  nsize);
```

## 1625 Parameters

1626 `v` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1627 vector.

1628                    **d** (IN) The type corresponding to the domain of the vector being created. Can be  
 1629                    one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
 1630                    GraphBLAS type.

1631                    **nsz** (IN) The size of the vector being created.

## 1632    **Return Values**

1633                    **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 1634                    blocking mode, this indicates that the API checks for the input  
 1635                    arguments passed successfully. Either way, output vector **v** is ready  
 1636                    to be used in the next method of the sequence.

1637                    **GrB\_PANIC** Unknown internal error.

1638                    **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 1639                    GraphBLAS objects (input or output) is in an invalid state caused  
 1640                    by a previous execution error. Call **GrB\_error()** to access any error  
 1641                    messages generated by the implementation.

1642                    **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1643                    **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
 1644                    (needed for user-defined types).

1645                    **GrB\_NULL\_POINTER** The **v** pointer is **NULL**.

1646                    **GrB\_INVALID\_VALUE** **nsz** is zero or outside the range of the type **GrB\_Index**.

## 1647    **Description**

1648                    Creates a new vector **v** of domain **D(d)**, size **nsz**, and empty **L(v)**. The method returns a handle  
 1649                    to the new vector in **v**.

1650                    It is not an error to call this method more than once on the same variable; however, the handle to  
 1651                    the previously created object will be overwritten.

### 1652    **4.2.4.2    Vector\_dup: Construct a copy of a GraphBLAS vector**

1653                    Creates a new vector with the same domain, size, and contents as another vector.

## 1654    **C Syntax**

```
1655                    GrB_Info GrB_Vector_dup(GrB_Vector                *w,  
1656                                                const GrB_Vector    u);
```

## 1657 **Parameters**

1658            **w** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1659            vector.

1660            **u** (IN) The GraphBLAS vector to be duplicated.

## 1661 **Return Values**

1662            **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1663            blocking mode, this indicates that the API checks for the input  
1664            arguments passed successfully. Either way, output vector **w** is ready  
1665            to be used in the next method of the sequence.

1666            **GrB\_PANIC** Unknown internal error.

1667            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1668            GraphBLAS objects (input or output) is in an invalid state caused  
1669            by a previous execution error. Call **GrB\_error()** to access any error  
1670            messages generated by the implementation.

1671            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1672            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **u**, has not been initialized by a call to  
1673            **Vector\_new** or **Vector\_dup**.

1674            **GrB\_NULL\_POINTER** The **w** pointer is **NULL**.

## 1675 **Description**

1676            Creates a new vector **w** of domain **D(u)**, size **size(u)**, and contents **L(u)**. The method returns a  
1677            handle to the new vector in **w**.

1678            It is not an error to call this method more than once on the same variable; however, the handle to  
1679            the previously created object will be overwritten.

### 1680 **4.2.4.3 Vector\_resize: Resize a vector**

1681            Changes the size of an existing vector.

## 1682 **C Syntax**

```
1683            GrB_Info GrB_Vector_resize(GrB_Vector w,  
1684                                        GrB_Index nsize);
```

## 1685 Parameters

- 1686        **w** (INOUT) An existing Vector object that is being resized.
- 1687        **nsz** (IN) The new size of the vector. It can be smaller or larger than the current size.

## 1688 Return Values

- 1689        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1690        blocking mode, this indicates that the API checks for the input  
1691        arguments passed successfully. Either way, output vector **w** is ready  
1692        to be used in the next method of the sequence.
- 1693        **GrB\_PANIC** Unknown internal error.
- 1694        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1695        GraphBLAS objects (input or output) is in an invalid state caused  
1696        by a previous execution error. Call **GrB\_error()** to access any error  
1697        messages generated by the implementation.
- 1698        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.
- 1699        **GrB\_NULL\_POINTER** The **w** pointer is NULL.
- 1700        **GrB\_INVALID\_VALUE** **nsz** is zero or outside the range of the type **GrB\_Index**.

## 1701 Description

- 1702        Changes the size of **w** to **nsz**. The domain **D(w)** of vector **w** remains the same. The contents **L(w)**  
1703        are modified as described below.
- 1704        Let  $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$  when the method is called. When the method returns,  $w = \langle \mathbf{D}(w), \text{nsz}, \mathbf{L}'(w) \rangle$   
1705        where  $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < \text{nsz})\}$ . That is, all elements of **w** with index greater  
1706        than or equal to the new vector size (**nsz**) are dropped.

### 1707 4.2.4.4 Vector\_clear: Clear a vector

- 1708        Removes all the elements (tuples) from a vector.

## 1709 C Syntax

1710        **GrB\_Info** **GrB\_Vector\_clear**(**GrB\_Vector** v);

## 1711 Parameters

- 1712        **v** (INOUT) An existing GraphBLAS vector to clear.

## 1713 Return Values

1714           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1715           blocking mode, this indicates that the API checks for the input  
1716           arguments passed successfully. Either way, output vector  $v$  is ready  
1717           to be used in the next method of the sequence.

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 Description

1727 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,  
1728  $L(v) = \emptyset$ . The size of the vector does not change.

### 1729 4.2.4.5 Vector\_size: Size of a vector

1730 Retrieve the size of a vector.

## 1731 C Syntax

```
1732           GrB_Info GrB_Vector_size(GrB_Index           *nsize,  
1733                                   const GrB_Vector v);
```

## 1734 Parameters

1735           nsize (OUT) On successful return, is set to the size of the vector.

1736           v (IN) An existing GraphBLAS vector being queried.

## 1737 Return Values

1738           GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1739           cessfully and the value of `nsize` has been set.

1740           GrB\_PANIC Unknown internal error.



1741        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 1742        GraphBLAS objects (input or output) is in an invalid state caused  
 1743        by a previous execution error. Call **GrB\_error()** to access any error  
 1744        messages generated by the implementation.

1745 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to  
 1746        **Vector\_new** or **Vector\_dup**.

1747        **GrB\_NULL\_POINTER** **nsz** pointer is **NULL**.

## 1748 **Description**

1749 Return **size(v)** in **nsz**.

### 1750 **4.2.4.6 Vector\_nvals: Number of stored elements in a vector**

1751 Retrieve the number of stored elements (tuples) in a vector.

## 1752 **C Syntax**

```
1753        GrB_Info GrB_Vector_nvals(GrB_Index        *nvals,  
1754                                        const GrB_Vector v);
```

## 1755 **Parameters**

1756        **nvals** (OUT) On successful return, this is set to the number of stored elements (tuples)  
 1757        in the vector.

1758        **v** (IN) An existing GraphBLAS vector being queried.

## 1759 **Return Values**

1760        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
 1761        cessfully and the value of **nvals** has been set.

1762        **GrB\_PANIC** Unknown internal error.

1763        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 1764        GraphBLAS objects (input or output) is in an invalid state caused  
 1765        by a previous execution error. Call **GrB\_error()** to access any error  
 1766        messages generated by the implementation.

1767        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1768 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector,  $v$ , has not been initialized by a call to  
 1769 Vector\_new or Vector\_dup.

1770 GrB\_NULL\_POINTER The `nvals` pointer is NULL.

## 1771 Description

1772 Return `nvals(v)` in `nvals`. This is the number of stored elements in vector  $v$ , which is the size of  
 1773  $L(v)$  (see Section 3.5.2).

## 1774 4.2.4.7 Vector\_build: Store elements from tuples into a vector

## 1775 C Syntax

```
1776      GrB_Info GrB_Vector_build(GrB_Vector      w,
1777                               const GrB_Index  *indices,
1778                               const <type>     *values,
1779                               GrB_Index        n,
1780                               const GrB_BinaryOp dup);
```

## 1781 Parameters

1782 `w` (INOUT) An existing Vector object to store the result.

1783 `indices` (IN) Pointer to an array of indices.

1784 `values` (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
 1785 vector `w`.

1786 `n` (IN) The number of entries contained in each array (the same for `indices` and `values`).

1787 `dup` (IN) An associative and commutative binary operator to apply when duplicate  
 1788 values for the same location are present in the input arrays. All three domains of  
 1789 `dup` must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If `dup` is GrB\_NULL,  
 1790 then duplicate locations will result in an error.

## 1791 Return Values

1792 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 1793 blocking mode, this indicates that the API checks for the input  
 1794 arguments passed successfully. Either way, output vector `w` is  
 1795 ready to be used in the next method of the sequence.

1796 GrB\_PANIC Unknown internal error.

1797       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
1798       opaque GraphBLAS objects (input or output) is in an invalid  
1799       state caused by a previous execution error. Call GrB\_error() to  
1800       access any error messages generated by the implementation.

1801       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1802       GrB\_UNINITIALIZED\_OBJECT Either w has not been initialized by a call to by GrB\_Vector\_new  
1803       or by GrB\_Vector\_dup, or dup has not been initialized by a call  
1804       to by GrB\_BinaryOp\_new.

1805       GrB\_NULL\_POINTER indices or values pointer is NULL.

1806       GrB\_INDEX\_OUT\_OF\_BOUNDS A value in indices is outside the allowed range for w.

1807       GrB\_DOMAIN\_MISMATCH Either the domains of the GraphBLAS binary operator dup are  
1808       not all the same, or the domains of values and w are incompatible  
1809       with each other or  $D_{dup}$ .

1810       GrB\_OUTPUT\_NOT\_EMPTY Output vector w already contains valid tuples (elements). In  
1811       other words, GrB\_Vector\_nvals(C) returns a positive value.

1812       GrB\_INVALID\_VALUE indices contains a duplicate location and dup is GrB\_NULL.

## 1813 Description

1814 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  
1815 from w in its domain; otherwise,  $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$ .

1816 Each tuple  $\{\text{indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the output in the form of

$$1817 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1818 If multiple values for the same location are present in the input arrays and dup is not GrB\_NULL,  
1819 dup is used to reduce the values before assignment into  $\tilde{\mathbf{w}}$  as follows:

$$1820 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1821 where  $\oplus$  is the dup binary operator. Finally, the resulting  $\tilde{\mathbf{w}}$  is copied into w via typecasting its  
1822 values to  $\mathbf{D}(\mathbf{w})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

1823 The nonopaque input arrays, indices and values, must be at least as large as n.

1824 It is an error to call this function on an output object with existing elements. In other words,  
1825 GrB\_Vector\_nvals(w) should evaluate to zero prior to calling this function.

1826 After GrB\_Vector\_build returns, it is safe for a programmer to modify or delete the arrays indices  
1827 or values.

#### 1828 4.2.4.8 Vector\_setElement: Set a single element in a vector

1829 Set one element of a vector to a given value.

#### 1830 C Syntax

```
1831 // scalar value
1832 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1833                               <type>         val,
1834                               GrB_Index       index);
1835
1836 // GraphBLAS scalar
1837 GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1838                               const GrB_Scalar s,
1839                               GrB_Index       index);
```

#### 1840 Parameters

1841 w (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1842 val or s (IN) Scalar assign. Its domain (type) must be compatible with the domain of w.

1843 index (IN) The location of the element to be assigned.

#### 1844 Return Values

1845 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1846 blocking mode, this indicates that the compatibility tests on in-  
1847 dex/dimensions and domains for the input arguments passed suc-  
1848 cessfully. Either way, the output vector w is ready to be used in  
1849 the next method of the sequence.

1850 GrB\_PANIC Unknown internal error.

1851 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1852 GraphBLAS objects (input or output) is in an invalid state caused  
1853 by a previous execution error. Call GrB\_error() to access any error  
1854 messages generated by the implementation.

1855 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1856 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, w, or GraphBLAS scalar, s, has not been  
1857 initialized by a call to a respective constructor.

1858 GrB\_INVALID\_INDEX index specifies a location that is outside the dimensions of w.

1859 GrB\_DOMAIN\_MISMATCH The domains of the vector and the scalar are incompatible.

## 1860 Description

1861 First, the scalar and output vector are tested for domain compatibility as follows:  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\mathbf{s})$   
 1862 must be compatible with  $\mathbf{D}(\mathbf{w})$ . Two domains are compatible with each other if values from  
 1863 one domain can be cast to values in the other domain as per the rules of the C language. In  
 1864 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 1865 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 1866 `GrB_Vector_setElement` ends and the domain mismatch error listed above is returned.

1867 Then, the `index` parameter is checked for a valid value where the following condition must hold:

$$1868 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1869 If this condition is violated, execution of `GrB_Vector_setElement` ends and the invalid index error  
 1870 listed above is returned.

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

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

1871 In the case of a transparent scalar or if  $\mathbf{L}(\mathbf{s})$  is not empty, then a value will be stored at the  
 1872 specified location in  $\mathbf{w}$ , overwriting any value that may have been stored there before. In the case  
 1873 of a GraphBLAS scalar, if  $\mathbf{L}(\mathbf{s})$  is empty, then any value stored at the specified location in  $\mathbf{w}$  will  
 1874 be removed.

1875 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
 1876 of  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
 1877 return value `GrB_SUCCESS` and the new contents of vector  $\mathbf{w}$  is as defined above but may not be  
 1878 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1879 4.2.4.9 Vector\_removeElement: Remove an element from a vector

1880 Remove (annihilate) one stored element from a vector.

## 1881 C Syntax

```
1882      GrB_Info GrB_Vector_removeElement(GrB_Vector  w,
1883                                     GrB_Index    index);
```

## 1884 Parameters

1885 `w` (INOUT) An existing GraphBLAS vector from which an element is to be removed.

1886 `index` (IN) The location of the element to be removed.

## 1887 Return Values

1888           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1889                       blocking mode, this indicates that the compatibility tests on in-  
1890                       dex/dimensions and domains for the input arguments passed suc-  
1891                       cessfully. Either way, the output vector **w** is ready to be used in  
1892                       the next method of the sequence.

1893           GrB\_PANIC Unknown internal error.

1894           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1895                       GraphBLAS objects (input or output) is in an invalid state caused  
1896                       by a previous execution error. Call **GrB\_error()** to access any error  
1897                       messages generated by the implementation.

1898           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1899 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, **w**, has not been initialized by a call to  
1900                       **Vector\_new** or **Vector\_dup**.

1901           GrB\_INVALID\_INDEX **index** specifies a location that is outside the dimensions of **w**.

## 1902 Description

1903 First, the **index** parameter is checked for a valid value where the following condition must hold:

$$1904 \qquad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1905 If this condition is violated, execution of **GrB\_Vector\_removeElement** ends and the invalid index  
1906 error listed above is returned.

1907 We are now ready to carry out the removal of a value that may be stored at the location specified  
1908 by **index**. If a value does not exist at the specified location in **w**, no error is reported and the  
1909 operation has no effect on the state of **w**. In either case, the following will be true on return from  
1910 the method:  $\text{index} \notin \text{ind}(\mathbf{w})$ .

1911 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new contents  
1912 of **w** is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method exits with  
1913 return value **GrB\_SUCCESS** and the new content of vector **w** is as defined above but may not be  
1914 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1915 4.2.4.10 Vector\_extractElement: Extract a single element from a vector.

1916 Extract one element of a vector into a scalar.

## 1917 C Syntax

```
1918      // scalar value
1919      GrB_Info GrB_Vector_extractElement(<type>          *val,
1920                                         const GrB_Vector u,
1921                                         GrB_Index      index);
1922
1923      // GraphBLAS scalar
1924      GrB_Info GrB_Vector_extractElement(GrB_Scalar      s,
1925                                         const GrB_Vector u,
1926                                         GrB_Index      index);
```

## 1927 Parameters

1928 **val** or **s** (INOUT) An existing scalar of whose domain is compatible with the domain of vector  
1929 **u**. On successful return, this scalar holds the result of the extract. Any previous  
1930 value stored in **val** or **s** is overwritten.

1931 **u** (IN) The GraphBLAS vector from which an element is extracted.

1932 **index** (IN) The location in **u** to extract.

## 1933 Return Values

1934 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1935 cessfully. This indicates that the compatibility tests on dimensions  
1936 and domains for the input arguments passed successfully, and the  
1937 output scalar, **val** or **s**, has been computed and is ready to be used  
1938 in the next method of the sequence.

1939 **GrB\_NO\_VALUE** When using the transparent scalar, **val**, this is returned when there  
1940 is no stored value at specified location.

1941 **GrB\_PANIC** Unknown internal error.

1942 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1943 GraphBLAS objects (input or output) is in an invalid state caused  
1944 by a previous execution error. Call **GrB\_error()** to access any error  
1945 messages generated by the implementation.

1946 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1947 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **u**, or scalar, **s**, has not been initialized by  
1948 a call to a corresponding constructor.

1949 **GrB\_NULL\_POINTER** **val** pointer is NULL.

1950 **GrB\_INVALID\_INDEX** **index** specifies a location that is outside the dimensions of **w**.

1951     GrB\_DOMAIN\_MISMATCH The domains of the vector and scalar are incompatible.

## 1952 Description

1953 First, the scalar and input vector are tested for domain compatibility as follows:  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\mathbf{s})$   
1954 must be compatible with  $\mathbf{D}(\mathbf{u})$ . Two domains are compatible with each other if values from  
1955 one domain can be cast to values in the other domain as per the rules of the C language. In  
1956 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1957 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1958 `GrB_Vector_extractElement` ends and the domain mismatch error listed above is returned.

1959 Then, the `index` parameter is checked for a valid value where the following condition must hold:

$$1960 \qquad 0 \leq \text{index} < \text{size}(\mathbf{u})$$

1961 If this condition is violated, execution of `GrB_Vector_extractElement` ends and the invalid index  
1962 error listed above is returned.

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

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

1963 If  $\text{index} \in \text{ind}(\mathbf{u})$ , then the corresponding value from  $\mathbf{u}$  is copied into  $\mathbf{s}$  or `val` with casting as  
1964 necessary. If  $\text{index} \notin \text{ind}(\mathbf{u})$ , then one of the follow occurs depending on output scalar type:

- 1965     • The GraphBLAS scalar,  $\mathbf{s}$ , is cleared and `GrB_SUCCESS` is returned.
- 1966     • The non-opaque scalar, `val`, is unchanged, and `GrB_NO_VALUE` is returned.

1967 When using the non-opaque scalar variant (`val`) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
1968 mode, the new contents of `val` are as defined above if the method exits with return value `GrB_SUCCESS`  
1969 or `GrB_NO_VALUE`.

1970 When using the GraphBLAS scalar variant ( $\mathbf{s}$ ) with a `GrB_SUCCESS` return value, the method  
1971 exits and the new contents of  $\mathbf{s}$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
1972 In `GrB_NONBLOCKING` mode, the new contents of  $\mathbf{s}$  is as defined above but may not be fully  
1973 computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1974 4.2.4.11 Vector\_extractTuples: Extract tuples from a vector

1975 Extract the contents of a GraphBLAS vector into non-opaque data structures.

## 1976 C Syntax

1977     GrB\_Info GrB\_Vector\_extractTuples(GrB\_Index                     \*indices,



```

1978                                     <type>                *values,
1979                                     GrB_Index              *n,
1980                                     const GrB_Vector        v);
1981

```

1982       **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored  
1983       values' indices.

1984       **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
1985       the stored values whose type is compatible with **D(v)**.

1986       **n** (INOUT) Pointer to a value indicating (on input) the number of elements the  
1987       **values** and **indices** arrays can hold. Upon return, it will contain the number of  
1988       values written to the arrays.

1989       **v** (IN) An existing GraphBLAS vector.

## 1990 **Return Values**

1991       **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1992       cessfully. This indicates that the compatibility tests on the input  
1993       argument passed successfully, and the output arrays, **indices** and  
1994       **values**, have been computed.

1995       **GrB\_PANIC** Unknown internal error.

1996       **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1997       GraphBLAS objects (input or output) is in an invalid state caused  
1998       by a previous execution error. Call **GrB\_error()** to access any error  
1999       messages generated by the implementation.

2000       **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2001       **GrB\_INSUFFICIENT\_SPACE** Not enough space in **indices** and **values** (as indicated by the **n** pa-  
2002       rameter) to hold all of the tuples that will be extracted.

2003       **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to  
2004       **Vector\_new** or **Vector\_dup**.

2005       **GrB\_NULL\_POINTER** **indices**, **values**, or **n** pointer is **NULL**.

2006       **GrB\_DOMAIN\_MISMATCH** The domains of the **v** vector or **values** array are incompatible with  
2007       one another.

## 2008 **Description**

2009       This method will extract all the tuples from the GraphBLAS vector **v**. The values associated  
2010       with those tuples are placed in the **values** array and the indices are placed in the **indices** array.

Both `indices` and `values` must be pre-allocated by the user to have enough space to hold at least `GrB_Vector_nvals(v)` elements before calling this function.

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

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

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

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

In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value `GrB_SUCCESS`, the new contents of the arrays `indices` and `values` are as defined above.

## 4.2.5 Matrix methods

### 4.2.5.1 Matrix\_new: Construct new matrix

Creates a new matrix with specified domain and dimensions.

## C Syntax

```
GrB_Info GrB_Matrix_new(GrB_Matrix *A,
                        GrB_Type    d,
                        GrB_Index    nrows,
                        GrB_Index    ncols);
```

## Parameters

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

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

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

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

## 2039 Return Values

2040           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2041                       blocking mode, this indicates that the API checks for the input ar-  
2042                       guments passed successfully. Either way, output matrix **A** is ready  
2043                       to be used in the next method of the sequence.

2044           GrB\_PANIC Unknown internal error.

2045           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2046                       GraphBLAS objects (input or output) is in an invalid state caused  
2047                       by a previous execution error. Call GrB\_error() to access any error  
2048                       messages generated by the implementation.

2049           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2050 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
2051                       (needed for user-defined types).

2052           GrB\_NULL\_POINTER The **A** pointer is NULL.

2053           GrB\_INVALID\_VALUE nrows or ncols is zero or outside the range of the type GrB\_Index.

## 2054 Description

2055 Creates a new matrix **A** of domain **D**(**d**), size **nrows**  $\times$  **ncols**, and empty **L**(**A**). The method returns  
2056 a handle to the new matrix in **A**.

2057 It is not an error to call this method more than once on the same variable; however, the handle to  
2058 the previously created object will be overwritten.

### 2059 4.2.5.2 Matrix\_dup: Construct a copy of a GraphBLAS matrix

2060 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

## 2061 C Syntax

```
2062           GrB_Info GrB_Matrix_dup(GrB_Matrix        *C,  
2063                                   const GrB_Matrix A);
```

## 2064 Parameters

2065           C (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2066                       matrix.

2067           A (IN) The GraphBLAS matrix to be duplicated.

## 2068 Return Values

2069           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2070           blocking mode, this indicates that the API checks for the input  
2071           arguments passed successfully. Either way, output matrix **C** is ready  
2072           to be used in the next method of the sequence.

2073           GrB\_PANIC Unknown internal error.

2074           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2075           GraphBLAS objects (input or output) is in an invalid state caused  
2076           by a previous execution error. Call **GrB\_error()** to access any error  
2077           messages generated by the implementation.

2078           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2079           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, **A**, has not been initialized by a call to  
2080           any matrix constructor.

2081           GrB\_NULL\_POINTER The **C** pointer is **NULL**.

## 2082 Description

2083           Creates a new matrix **C** of domain **D(A)**, size **nrows(A) × ncols(A)**, and contents **L(A)**. It returns  
2084           a handle to it in **C**.

2085           It is not an error to call this method more than once on the same variable; however, the handle to  
2086           the previously created object will be overwritten.

### 2087 4.2.5.3 Matrix\_diag: Construct a diagonal GraphBLAS matrix

2088           Creates a new matrix with the same domain and contents as a **GrB\_Vector**, and square dimensions  
2089           appropriate for placing the contents of the vector along the specified diagonal of the matrix.

## 2090 C Syntax

```
2091           GrB_Info GrB_Matrix_diag(GrB_Matrix        *C,  
2092                                    const GrB_Vector   v,  
2093                                    int64_t           k);
```

## 2094 Parameters

2095           **C** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2096           matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

2097            **v** (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the  
 2098            matrix.

2099            **k** (IN) The diagonal to which the vector is assigned.  $k = 0$  represents the main  
 2100            diagonal,  $k > 0$  is above the main diagonal, and  $k < 0$  is below.

## 2101 Return Values

2102            **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 2103            blocking mode, this indicates that the API checks for the input  
 2104            arguments passed successfully. Either way, output matrix **C** is ready  
 2105            to be used in the next method of the sequence.

2106            **GrB\_PANIC** Unknown internal error.

2107            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 2108            GraphBLAS objects (input or output) is in an invalid state caused  
 2109            by a previous execution error. Call **GrB\_error()** to access any error  
 2110            messages generated by the implementation.

2111            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

2112            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to  
 2113            **Vector\_new** or **Vector\_dup**.

2114            **GrB\_NULL\_POINTER** The C pointer is NULL.

## 2115 Description

2116            Creates a new matrix **C** of domain **D(v)**, size  $(\text{size}(\mathbf{v}) + |k|) \times (\text{size}(\mathbf{v}) + |k|)$ , and contents

$$2117 \quad \mathbf{L}(\mathbf{C}) = \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or}$$

$$2118 \quad \mathbf{L}(\mathbf{C}) = \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0.$$

2119            It returns a handle to it in **C**. It is not an error to call this method more than once on the same  
 2120            variable; however, the handle to the previously created object will be overwritten.

### 2121 4.2.5.4 Matrix\_resize: Resize a matrix

2122            Changes the dimensions of an existing matrix.

## 2123 C Syntax

```
2124            GrB_Info GrB_Matrix_resize(GrB_Matrix C,
2125                                        GrB_Index nrows,
2126                                        GrB_Index ncols);
```

## 2127 Parameters

2128           C (INOUT) An existing Matrix object that is being resized.

2129           nrows (IN) The new number of rows of the matrix. It can be smaller or larger than the  
2130           current number of rows.

2131           ncols (IN) The new number of columns of the matrix. It can be smaller or larger than  
2132           the current number of columns.

## 2133 Return Values

2134           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2135           blocking mode, this indicates that the API checks for the input  
2136           arguments passed successfully. Either way, output matrix C is ready  
2137           to be used in the next method of the sequence.

2138           GrB\_PANIC Unknown internal error.

2139           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2140           GraphBLAS objects (input or output) is in an invalid state caused  
2141           by a previous execution error. Call GrB\_error() to access any error  
2142           messages generated by the implementation.

2143           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2144           GrB\_NULL\_POINTER The C pointer is NULL.

2145           GrB\_INVALID\_VALUE nrows or ncols is zero or outside the range of the type GrB\_Index.

## 2146 Description

2147 Changes the number of rows and columns of C to nrows and ncols, respectively. The domain  $\mathbf{D}(\mathbf{C})$   
2148 of matrix C remains the same. The contents  $\mathbf{L}(\mathbf{C})$  are modified as described below.

2149 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  
2150 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  
2151 greater than or equal to ncols are dropped.  
2152

### 2153 4.2.5.5 Matrix\_clear: Clear a matrix

2154 Removes all elements (tuples) from a matrix.

## 2155 C Syntax

2156           GrB\_Info GrB\_Matrix\_clear(GrB\_Matrix A);

## 2157 Parameters

2158           A (IN) An existing GraphBLAS matrix to clear.

## 2159 Return Values

2160           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2161                       blocking mode, this indicates that the API checks for the input ar-  
2162                       guments passed successfully. Either way, output matrix A is ready  
2163                       to be used in the next method of the sequence.

2164           GrB\_PANIC Unknown internal error.

2165           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2166                       GraphBLAS objects (input or output) is in an invalid state caused  
2167                       by a previous execution error. Call GrB\_error() to access any error  
2168                       messages generated by the implementation.

2169           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2170           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2171                       any matrix constructor.

## 2172 Description

2173 Removes all elements (tuples) from an existing matrix. After the call to GrB\_Matrix\_clear(A),  
2174  $L(A) = \emptyset$ . The dimensions of the matrix do not change.

### 2175 4.2.5.6 Matrix\_nrows: Number of rows in a matrix

2176 Retrieve the number of rows in a matrix.

## 2177 C Syntax

```
2178           GrB_Info GrB_Matrix_nrows(GrB_Index           *nrows,  
2179                                       const GrB_Matrix  A);
```

## 2180 Parameters

2181           nrows (OUT) On successful return, contains the number of rows in the matrix.

2182           A (IN) An existing GraphBLAS matrix being queried.

## 2183 **Return Values**

2184                   GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2185                   cessfully and the value of `nrows` has been set.

2186                   GrB\_PANIC Unknown internal error.

2187           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2188           GraphBLAS objects (input or output) is in an invalid state caused  
2189           by a previous execution error. Call `GrB_error()` to access any error  
2190           messages generated by the implementation.

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

2193           GrB\_NULL\_POINTER `nrows` pointer is NULL.

## 2194 **Description**

2195 Return `nrows(A)` in `nrows` (the number of rows).

## 2196 **4.2.5.7 Matrix\_ncols: Number of columns in a matrix**

2197 Retrieve the number of columns in a matrix.

## 2198 **C Syntax**

```
2199           GrB_Info GrB_Matrix_ncols(GrB_Index           *ncols,  
2200                                    const GrB_Matrix A);
```

## 2201 **Parameters**

2202           ncols (OUT) On successful return, contains the number of columns in the matrix.

2203           A (IN) An existing GraphBLAS matrix being queried.

## 2204 **Return Values**

2205                   GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2206                   cessfully and the value of `ncols` has been set.

2207                   GrB\_PANIC Unknown internal error.



2208           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 2209           GraphBLAS objects (input or output) is in an invalid state caused  
 2210           by a previous execution error. Call GrB\_error() to access any error  
 2211           messages generated by the implementation.

2212 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
 2213           any matrix constructor.

2214           GrB\_NULL\_POINTER ncols pointer is NULL.

## 2215 Description

2216 Return **ncols(A)** in **ncols** (the number of columns).

### 2217 4.2.5.8 Matrix\_nvals: Number of stored elements in a matrix

2218 Retrieve the number of stored elements (tuples) in a matrix.

## 2219 C Syntax

```
2220      GrB_Info GrB_Matrix_nvals(GrB_Index      *nvals,
2221                               const GrB_Matrix A);
```

## 2222 Parameters

2223           **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in  
 2224           the matrix.

2225           **A** (IN) An existing GraphBLAS matrix being queried.

## 2226 Return Values

2227           GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 2228           cessfully and the value of **nvals** has been set.

2229           GrB\_PANIC Unknown internal error.

2230           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 2231           GraphBLAS objects (input or output) is in an invalid state caused  
 2232           by a previous execution error. Call GrB\_error() to access any error  
 2233           messages generated by the implementation.

2234           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2235 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
 2236 any matrix constructor.

2237 GrB\_NULL\_POINTER The nvals pointer is NULL.

## 2238 Description

2239 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix A, which is the size of  
 2240 **L(A)** (see Section 3.5.3).

## 2241 4.2.5.9 Matrix\_build: Store elements from tuples into a matrix

### 2242 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);
```

### 2243 Parameters

2244 C (INOUT) An existing Matrix object to store the result.

2245 row\_indices (IN) Pointer to an array of row indices.

2246 col\_indices (IN) Pointer to an array of column indices.

2247 values (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
 2248 matrix, C.

2249 n (IN) The number of entries contained in each array (the same for row\_indices,  
 2250 col\_indices, and values).

2251 dup (IN) An associative and commutative binary operator to apply when duplicate  
 2252 values for the same location are present in the input arrays. All three domains of  
 2253 dup must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If dup is GrB\_NULL,  
 2254 then duplicate locations will result in an error.

### 2255 Return Values

2256 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 2257 blocking mode, this indicates that the API checks for the input  
 2258 arguments passed successfully. Either way, output matrix C is  
 2259 ready to be used in the next method of the sequence.

2260                   GrB\_PANIC Unknown internal error.

2261           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2262                                   opaque GraphBLAS objects (input or output) is in an invalid  
2263                                   state caused by a previous execution error. Call GrB\_error() to  
2264                                   access any error messages generated by the implementation.

2265           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2266   GrB\_UNINITIALIZED\_OBJECT Either C has not been initialized by a call to any matrix construc-  
2267                                   tor, or dup has not been initialized by a call to by GrB\_BinaryOp\_new.

2268           GrB\_NULL\_POINTER row\_indices, col\_indices or values pointer is NULL.

2269 GrB\_INDEX\_OUT\_OF\_BOUNDS A value in row\_indices or col\_indices is outside the allowed range  
2270                                   for C.

2271           GrB\_DOMAIN\_MISMATCH Either the domains of the GraphBLAS binary operator dup are  
2272                                   not all the same, or the domains of values and C are incompatible  
2273                                   with each other or  $D_{dup}$ .

2274   GrB\_OUTPUT\_NOT\_EMPTY Output matrix C already contains valid tuples (elements). In  
2275                                   other words, GrB\_Matrix\_nvals(C) returns a positive value.

2276           GrB\_INVALID\_VALUE indices contains a duplicate location and dup is GrB\_NULL.

## 2277 Description

2278 If dup is not GrB\_NULL, an internal matrix  $\tilde{C} = \langle D_{dup}, \mathbf{nrows}(C), \mathbf{ncols}(C), \emptyset \rangle$  is created, which  
2279 only differs from C in its domain; otherwise,  $\tilde{C} = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \emptyset \rangle$ .

2280 Each tuple  $\{\text{row\_indices}[k], \text{col\_indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the  
2281 output in the form of

$$2282 \quad \tilde{C}(\text{row\_indices}[k], \text{col\_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(C)) \text{values}[k] & \text{otherwise.} \end{cases}$$

2283 If multiple values for the same location are present in the input arrays and dup is not GrB\_NULL,  
2284 dup is used to reduce the values before assignment into  $\tilde{C}$  as follows:

$$2285 \quad \tilde{C}_{ij} = \bigoplus_{k: \text{row\_indices}[k]=i \wedge \text{col\_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2286 where  $\oplus$  is the dup binary operator. Finally, the resulting  $\tilde{C}$  is copied into C via typecasting its  
2287 values to  $\mathbf{D}(C)$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

2288 The nonopaque input arrays row\_indices, col\_indices, and values must be at least as large as n.

2289 It is an error to call this function on an output object with existing elements. In other words,  
2290 GrB\_Matrix\_nvals(C) should evaluate to zero prior to calling this function.

2291 After GrB\_Matrix\_build returns, it is safe for a programmer to modify or delete the arrays row\_indices,  
2292 col\_indices, or values.

#### 2293 4.2.5.10 Matrix\_setElement: Set a single element in matrix

2294 Set one element of a matrix to a given value.

#### 2295 C Syntax

```
2296         // scalar value
2297         GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2298                                     <type>           val,
2299                                     GrB_Index          row_index,
2300                                     GrB_Index          col_index);
2301
2302         // GraphBLAS scalar
2303         GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2304                                     const GrB_Scalar    s,
2305                                     GrB_Index          row_index,
2306                                     GrB_Index          col_index);
```

#### 2307 Parameters

2308 C (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.

2309 val or s (IN) Scalar to assign. Its domain (type) must be compatible with the domain of  
2310 C.

2311 row\_index (IN) Row index of element to be assigned

2312 col\_index (IN) Column index of element to be assigned

#### 2313 Return Values

2314 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2315 blocking mode, this indicates that the compatibility tests on in-  
2316 dex/dimensions and domains for the input arguments passed suc-  
2317 cessfully. Either way, the output matrix C is ready to be used in  
2318 the next method of the sequence.

2319 GrB\_PANIC Unknown internal error.

2320 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2321 GraphBLAS objects (input or output) is in an invalid state caused

2322 by a previous execution error. Call `GrB_error()` to access any error  
 2323 messages generated by the implementation.

2324 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2325 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been  
 2326 initialized by a call to a respective constructor.

2327 **GrB\_INVALID\_INDEX** `row_index` or `col_index` is outside the allowable range (i.e., not less  
 2328 than `nrows(C)` or `ncols(C)`, respectively).

2329 **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and the scalar are incompatible.

## 2330 Description

2331 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or  
 2332 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from  
 2333 one domain can be cast to values in the other domain as per the rules of the C language. In  
 2334 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 2335 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 2336 `GrB_Matrix_setElement` ends and the domain mismatch error listed above is returned.

2337 Then, both index parameters are checked for valid values where following conditions must hold:

$$2338 \begin{aligned} 0 &\leq \text{row\_index} < \text{nrows}(\mathbf{C}), \\ 0 &\leq \text{col\_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2339 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid  
 2340 index error listed above is returned.

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

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

2341 In the case of a transparent scalar or if **L(s)** is not empty, then a value will be stored at the  
 2342 specified location in **C**, overwriting any value that may have been stored there before. In the case  
 2343 of a GraphBLAS scalar and if **L(s)** is empty, then any value stored at the specified location in **C**  
 2344 will be removed.

2345 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new contents  
 2346 of **C** is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method exits with  
 2347 return value **GrB\_SUCCESS** and the new content of vector **C** is as defined above but may not be  
 2348 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2349 4.2.5.11 `Matrix_removeElement`: Remove an element from a matrix

2350 Remove (annihilate) one stored element from a matrix.

## 2351 C Syntax

```
2352         GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,  
2353                                           GrB_Index    row_index,  
2354                                           GrB_Index    col_index);
```

## 2355 Parameters

2356       C (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2357       row\_index (IN) Row index of element to be removed

2358       col\_index (IN) Column index of element to be removed

## 2359 Return Values

2360       GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2361                   blocking mode, this indicates that the compatibility tests on in-  
2362                   dex/dimensions and domains for the input arguments passed suc-  
2363                   cessfully. Either way, the output matrix C is ready to be used in  
2364                   the next method of the sequence.

2365       GrB\_PANIC Unknown internal error.

2366       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2367                           GraphBLAS objects (input or output) is in an invalid state caused  
2368                           by a previous execution error. Call GrB\_error() to access any error  
2369                           messages generated by the implementation.

2370       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2371       GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to  
2372                                  any matrix constructor.

2373       GrB\_INVALID\_INDEX row\_index or col\_index is outside the allowable range (i.e., not less  
2374                           than **nrows**(C) or **ncols**(C), respectively).

## 2375 Description

2376       First, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2377 \quad & 0 \leq \text{row\_index} < \text{nrows}(\text{C}), \\ & 0 \leq \text{col\_index} < \text{ncols}(\text{C}) \end{aligned}$$

2378       If either of these conditions is violated, execution of GrB\_Matrix\_removeElement ends and the  
2379       invalid index error listed above is returned.

2380 We are now ready to carry out the removal of a value that may be stored at the location specified by  
 2381 (row\_index, col\_index). If a value does not exist at the specified location in C, no error is reported  
 2382 and the operation has no effect on the state of C. In either case, the following will be true on return  
 2383 from this method: (row\_index, col\_index)  $\notin$  ind(C)

2384 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 2385 of C is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 2386 return value GrB\_SUCCESS and the new content of vector C is as defined above but may not be  
 2387 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 2388 4.2.5.12 Matrix\_extractElement: Extract a single element from a matrix

2389 Extract one element of a matrix into a scalar.

#### 2390 C Syntax

```
2391      // scalar value
2392      GrB_Info GrB_Matrix_extractElement(<type>          *val,
2393                                         const GrB_Matrix A,
2394                                         GrB_Index      row_index,
2395                                         GrB_Index      col_index);
2396
2397      // GraphBLAS scalar
2398      GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2399                                         const GrB_Matrix A,
2400                                         GrB_Index      row_index,
2401                                         GrB_Index      col_index);
2402
```

#### 2403 Parameters

2404 val or s (INOUT) An existing scalar whose domain is compatible with the domain of matrix  
 2405 A. On successful return, this scalar holds the result of the extract. Any previous  
 2406 value stored in val or s is overwritten.

2407 A (IN) The GraphBLAS matrix from which an element is extracted.

2408 row\_index (IN) The row index of location in A to extract.

2409 col\_index (IN) The column index of location in A to extract.

#### 2410 Return Values

2411 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 2412 cessfully. This indicates that the compatibility tests on dimensions

2413 and domains for the input arguments passed successfully, and the  
 2414 output scalar, **val** or **s**, has been computed and is ready to be used  
 2415 in the next method of the sequence.

2416 **GrB\_NO\_VALUE** When using the transparent scalar, **val**, this is returned when there  
 2417 is no stored value at specified location.

2418 **GrB\_PANIC** Unknown internal error.

2419 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 2420 GraphBLAS objects (input or output) is in an invalid state caused  
 2421 by a previous execution error. Call **GrB\_error()** to access any error  
 2422 messages generated by the implementation.

2423 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2424 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by  
 2425 a call to a corresponding constructor.

2426 **GrB\_NULL\_POINTER** **val** pointer is NULL.

2427 **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e. less than  
 2428 zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-  
 2429 tively).

2430 **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and scalar are incompatible.

## 2431 Description

2432 First, the scalar and input matrix are tested for domain compatibility as follows: **D(val)** or **D(s)**  
 2433 must be compatible with **D(A)**. Two domains are compatible with each other if values from  
 2434 one domain can be cast to values in the other domain as per the rules of the C language. In  
 2435 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 2436 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 2437 **GrB\_Matrix\_extractElement** ends and the domain mismatch error listed above is returned.

2438 Then, both index parameters are checked for valid values where following conditions must hold:

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

2440 If either condition is violated, execution of **GrB\_Matrix\_extractElement** ends and the invalid index  
 2441 error listed above is returned.

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

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



2442 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  $\mathbf{val}$   
 2443 with casting as necessary. If  $(\text{row\_index}, \text{col\_index}) \notin \mathbf{ind}(\mathbf{A})$ , then one of the follow occurs  
 2444 depending on output scalar type:

- 2445 • The GraphBLAS scalar,  $\mathbf{s}$ , is cleared and  $\text{GrB\_SUCCESS}$  is returned.
- 2446 • The non-opaque scalar,  $\mathbf{val}$ , is unchanged, and  $\text{GrB\_NO\_VALUE}$  is returned.

2447 When using the non-opaque scalar variant ( $\mathbf{val}$ ) in both  $\text{GrB\_BLOCKING}$  mode  $\text{GrB\_NONBLOCKING}$   
 2448 mode, the new contents of  $\mathbf{val}$  are as defined above if the method exits with return value  $\text{GrB\_SUCCESS}$   
 2449 or  $\text{GrB\_NO\_VALUE}$ .

2450 When using the GraphBLAS scalar variant ( $\mathbf{s}$ ) with a  $\text{GrB\_SUCCESS}$  return value, the method  
 2451 exits and the new contents of  $\mathbf{s}$  is as defined above and fully computed in  $\text{GrB\_BLOCKING}$  mode.  
 2452 In  $\text{GrB\_NONBLOCKING}$  mode, the new contents of  $\mathbf{s}$  is as defined above but may not be fully  
 2453 computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 2454 4.2.5.13 Matrix\_extractTuples: Extract tuples from a matrix

2455 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

#### 2456 C Syntax

```
2457      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2458                                     GrB_Index      *col_indices,
2459                                     <type>          *values,
2460                                     GrB_Index      *n,
2461                                     const GrB_Matrix A);
```

#### 2462 Parameters

2463 **row\_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the  
 2464 row indices.

2465 **col\_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the  
 2466 column indices.

2467 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
 2468 the stored values whose type is compatible with  $\mathbf{D}(\mathbf{A})$ .

2469 **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,  
 2470 **row\_indices**, and **col\_indices** arrays can hold. Upon return, it will contain the  
 2471 number of values written to the arrays.

2472 **A** (IN) An existing GraphBLAS matrix.

## 2473 Return Values

2474	<b>GrB_SUCCESS</b>	In blocking or non-blocking mode, the operation completed successfully. This indicates that the compatibility tests on the input
2475		argument passed successfully, and the output arrays, <b>indices</b> and
2476		<b>values</b> , have been computed.
2477		
2478	<b>GrB_PANIC</b>	Unknown internal error.
2479	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the opaque
2480		GraphBLAS objects (input or output) is in an invalid state caused
2481		by a previous execution error. Call <b>GrB_error()</b> to access any error
2482		messages generated by the implementation.
2483	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for operation.
2484	<b>GrB_INSUFFICIENT_SPACE</b>	Not enough space in <b>row_indices</b> , <b>col_indices</b> , and <b>values</b> (as indi-
2485		cated by the <b>n</b> parameter) to hold all of the tuples that will be
2486		extracted.
2487	<b>GrB_UNINITIALIZED_OBJECT</b>	The GraphBLAS matrix, <b>A</b> , has not been initialized by a call to
2488		any matrix constructor.
2489	<b>GrB_NULL_POINTER</b>	<b>row_indices</b> , <b>col_indices</b> , <b>values</b> or <b>n</b> pointer is <b>NULL</b> .
2490	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the <b>A</b> matrix and <b>values</b> array are incompatible
2491		with one another.

## 2492 Description

2493 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with  
2494 those tuples are placed in the **values** array, the column indices are placed in the **col\_indices** array,  
2495 and the row indices are placed in the **row\_indices** array. These output arrays are pre-allocated by  
2496 the user before calling this function such that each output array has enough space to hold at least  
2497 **GrB\_Matrix\_nvals(A)** elements.

2498 Upon return of this function, a pair of  $\{\text{row\_indices}[k], \text{col\_indices}[k]\}$  are unique for every valid  
2499  $k$ , but they are not required to be sorted in any particular order. Each tuple  $(i, j, A_{ij})$  in **A** is  
2500 unzipped and copied into a distinct  $k$ th location in output vectors:

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

2501 where  $0 \leq k < \text{GrB\_Matrix\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if **row\_indices**[ $k$ ],  
2502 **col\_indices**[ $k$ ] and **values**[ $k$ ] exist upon return, so does **row\_indices**[ $j$ ], **col\_indices**[ $j$ ] and **values**[ $j$ ] for  
2503 all  $j$  such that  $0 \leq j < k$ .

2504 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,  
2505 then a **GrB\_INSUFFICIENT\_SPACE** error is returned since it is undefined which subset of values  
2506 would be extracted.

2507 In both GrB\_BLOCKING mode GrB\_NONBLOCKING mode if the method exits with return value  
2508 GrB\_SUCCESS, the new contents of the arrays row\_indices, col\_indices and values are as defined  
2509 above.

2510 **4.2.5.14 Matrix\_exportHint: Provide a hint as to which storage format might be most**  
2511 **efficient for exporting a matrix**

## 2512 C Syntax

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

## 2513 Parameters

2514 hint (OUT) Pointer to a value of type GrB\_Format.

2515 A (IN) A GraphBLAS matrix object.

## 2516 Return Values

2517 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2518 cessfully and the value of hint has been set.

2519 GrB\_PANIC Unknown internal error.

2520 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2521 opaque GraphBLAS objects (input or output) is in an invalid  
2522 state caused by a previous execution error. Call GrB\_error() to  
2523 access any error messages generated by the implementation.

2524 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2525 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2526 any matrix constructor.

2527 GrB\_NULL\_POINTER hint is NULL.

2528 GrB\_NO\_VALUE If the implementation does not have a preferred format, it may  
2529 return the value GrB\_NO\_VALUE.

## 2530 Description

2531 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for  
2532 exporting the matrix A. GraphBLAS implementations might return the current storage format of  
2533 the matrix, or the format to which it could most efficiently be exported. However, implementations  
2534 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is  
2535 free to refuse to provide a format hint, returning GrB\_NO\_VALUE.

2536 **4.2.5.15 Matrix\_exportSize:** Return the array sizes necessary to export a GraphBLAS  
 2537 matrix object

## 2538 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);
```

## 2539 Parameters

2540 **n\_indptr** (OUT) Pointer to a value of type GrB\_Index.

2541 **n\_indices** (OUT) Pointer to a value of type GrB\_Index.

2542 **n\_values** (OUT) Pointer to a value of type GrB\_Index.

2543 **format** (IN) a value indicating the format in which the matrix will be exported, as defined  
 2544 in Section 3.5.3.1.

2545 **A** (IN) A GraphBLAS matrix object.

## 2546 Return Values

2547 **GrB\_SUCCESS** In blocking mode or non-blocking mode, the operation com-  
 2548 pleted successfully. This indicates that the API checks for the  
 2549 input arguments passed successfully, and the number of elements  
 2550 necessary for the export buffers have been written to **n\_indptr**,  
 2551 **n\_indices**, and **n\_values**, respectively.

2552 **GrB\_PANIC** Unknown internal error.

2553 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 2554 opaque GraphBLAS objects (input or output) is in an invalid  
 2555 state caused by a previous execution error. Call **GrB\_error()** to  
 2556 access any error messages generated by the implementation.

2557 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2558 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS Matrix, **A**, has not been initialized by a call to  
 2559 any matrix constructor.

2560 **GrB\_NULL\_POINTER** **n\_indptr**, **n\_indices**, or **n\_values** is NULL.

## 2561 Description

2562 Given a matrix **A**, returns the required capacities of arrays **values**, **indptr**, and **indices** necessary to  
2563 export the matrix in the format specified by **format**. The output values **n\_values**, **n\_indptr**, and  
2564 **indices** will contain the corresponding sizes of the arrays (in number of elements) that must be  
2565 allocated to hold the exported matrix. The argument **format** can be chosen arbitrarily by the user  
2566 as one of the values defined in Section 3.5.3.1.

### 2567 4.2.5.16 Matrix\_export: Export a GraphBLAS matrix to a pre-defined format

## 2568 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);
```

## 2569 Parameters

2570 **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-  
2571 dices, depending on the value of **format**. It must be large enough to hold at  
2572 least **n\_indptr** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2573 **GrB\_Matrix\_exportSize()** method.

2574 **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements  
2575 in **values**, depending on the value of **format**. It must be large enough to hold at  
2576 least **n\_indices** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2577 **GrB\_Matrix\_exportSize()** method.

2578 **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-  
2579 ment must match the type of the values stored in **A**. It must be large enough  
2580 to hold at least **n\_values** elements of that type, where **n\_values** was returned from  
2581 **GrB\_Matrix\_exportSize**.

2582 **n\_indptr** (INOUT) Pointer to a value indicating (on input) the number of elements the **indptr**  
2583 array can hold. Upon return, it will contain the number of elements written to the  
2584 array.

2585 **n\_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**  
2586 array can hold. Upon return, it will contain the number of elements written to the  
2587 array.

2588        **n\_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**  
2589        array can hold. Upon return, it will contain the number of elements written to the  
2590        array.

2591        **format** (IN) a value indicating the format in which the matrix will be exported, as defined  
2592        in Section 3.5.3.1.

2593        **A** (IN) A GraphBLAS matrix object.

## 2594 **Return Values**

2595        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2596        cessfully. This indicates that the compatibility tests on the input  
2597        argument passed successfully, and the output arrays, **indptr**, **in-**  
2598        **indices** and **values**, have been computed.

2599        **GrB\_PANIC** Unknown internal error.

2600        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2601        opaque GraphBLAS objects (input or output) is in an invalid  
2602        state caused by a previous execution error. Call **GrB\_error()** to  
2603        access any error messages generated by the implementation.

2604        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2605        **GrB\_INSUFFICIENT\_SPACE** Not enough space in **indptr**, **indices**, and/or **values** (as indicated  
2606        by the corresponding **n\_\*** parameter) to hold all of the corre-  
2607        sponding elements that will be extacted.

2608        **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2609        any matrix constructor.

2610        **GrB\_NULL\_POINTER** **indptr**, **indices**, **values** **n\_indptr**, **n\_indices**, **n\_values** pointer is  
2611        **NULL**.

2612        **GrB\_DOMAIN\_MISMATCH** The domain of **A** does not match with the type of **values**.

## 2613 **Description**

2614        Given a matrix **A**, this method exports the contents of the matrix into one of the pre-defined  
2615        **GrB\_Format** formats from Section 3.5.3.1. The user-allocated arrays pointed to by **indptr**, **indices**,  
2616        and **values** must be at least large enough to hold the corresponding number of elements returned  
2617        by calling **GrB\_Matrix\_exportSize**. The value of **format** can be chosen arbitrarily, but a call to  
2618        **GrB\_Matrix\_exportHint** may suggest a format that results in the most efficient export. Details  
2619        of the contents of **indptr**, **indices**, and **values** corresponding to each supported format is given in  
2620        Appendix B.

#### 2621 4.2.5.17 Matrix\_import: Import a matrix into a GraphBLAS object

#### 2622 C Syntax

```
GrB_Info GrB_Matrix_import(GrB_Matrix      *A,  
                           GrB_Type        d,  
                           GrB_Index       nrows,  
                           GrB_Index       ncols,  
                           const GrB_Index *indptr,  
                           const GrB_Index *indices,  
                           const <type>   *values,  
                           GrB_Index       n_indptr,  
                           GrB_Index       n_indices,  
                           GrB_Index       n_values,  
                           GrB_Format      format);
```

#### 2623 Parameters

2624       A (INOUT) On a successful return, contains a handle to the newly created Graph-  
2625       BLAS matrix.

2626       d (IN) The type corresponding to the domain of the matrix being created. Can be  
2627       one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
2628       GraphBLAS type.

2629       nrows (IN) Integer value holding the number of rows in the matrix.

2630       ncols (IN) Integer value holding the number of columns in the matrix.

2631       indptr (IN) Pointer to an array of row or column offsets, or row indices, depending on the  
2632       value of `format`.

2633       indices (IN) Pointer to an array row or column indices of the elements in `values`, depending  
2634       on the value of `format`.

2635       values (IN) Pointer to an array of values. Type must match the type of `d`.

2636       n\_indptr (IN) Integer value holding the number of elements in the array pointed to by `indptr`.

2637       n\_indices (IN) Integer value holding the number of elements in the array pointed to by `indices`.

2638       n\_values (IN) Integer value holding the number of elements in the array pointed to by `values`.

2639       format (IN) a value indicating the format of the matrix being imported, as defined in  
2640       Section 3.5.3.1.

## 2641 Return Values

2642	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
2643		blocking mode, this indicates that the API checks for the input
2644		arguments passed successfully and the input arrays have been
2645		consumed. Either way, output matrix <b>A</b> is ready to be used in
2646		the next method of the sequence.
2647	GrB_PANIC	Unknown internal error.
2648	GrB_OUT_OF_MEMORY	Not enough memory available for operation.
2649	GrB_UNINITIALIZED_OBJECT	The GrB_Type object has not been initialized by a call to GrB_Type_new
2650		(needed for user-defined types).
2651	GrB_NULL_POINTER	<b>A</b> , <b>indptr</b> , <b>indices</b> or <b>values</b> pointer is NULL.
2652	GrB_INDEX_OUT_OF_BOUNDS	A value in <b>indptr</b> or <b>indices</b> is outside the allowed range for indices
2653		in <b>A</b> and or the size of <b>values</b> , <b>n_values</b> , depending on the value
2654		of <b>format</b> .
2655	GrB_INVALID_VALUE	<b>nrows</b> or <b>ncols</b> is zero or outside the range of the type GrB_Index.
2656	GrB_DOMAIN_MISMATCH	The domain given in parameter <b>d</b> does not match the element
2657		type of <b>values</b> .

## 2658 Description

2659 Creates a new matrix **A** of domain **D(d)** and dimension **nrows**  $\times$  **ncols**. The new GraphBLAS  
2660 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.  
2661 The method returns a handle to the new matrix in **A**. The structure of the data being imported is  
2662 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of  
2663 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2664 It is not an error to call this method more than once on the same output matrix; however, the  
2665 handle to the previously created object will be overwritten.

### 2666 4.2.5.18 Matrix\_serializeSize: Compute the serialize buffer size

2667 Compute the buffer size (in bytes) necessary to serialize a GrB\_Matrix using GrB\_Matrix\_serialize.

## 2668 C Syntax

```
GrB_Info GrB_Matrix_serializeSize(GrB_Index *size,  
                                  GrB_Matrix A);
```



## 2669 Parameters

2670           size (OUT) Pointer to GrB\_Index value where size in bytes of serialized object will be  
2671           written.

2672           A (IN) A GraphBLAS matrix object.

## 2673 Return Values

2674           GrB\_SUCCESS The operation completed successfully and the value pointed to  
2675           by \*size has been computed and is ready to use.

2676           GrB\_PANIC Unknown internal error.

2677           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2678           GrB\_NULL\_POINTER size is NULL.

## 2679 Description

2680 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object A.  
2681 Users may then allocate a buffer of size bytes to pass as a parameter to GrB\_Matrix\_serialize.

### 2682 4.2.5.19 Matrix\_serialize: Serialize a GraphBLAS matrix.

2683 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

## 2684 C Syntax

```
GrB_Info GrB_Matrix_serialize(void      *serialized_data,  
                               GrB_Index *serialized_size,  
                               GrB_Matrix A);
```

## 2685 Parameters

2686   serialized\_data (INOUT) Pointer to the preallocated buffer where the serialized matrix will be  
2687   written.

2688   serialized\_size (INOUT) On input, the size in bytes of the buffer pointed to by serialized\_data.  
2689   On output, the number of bytes written to serialized\_data.

2690   A (IN) A GraphBLAS matrix object.

## 2691 Return Values

2692                   GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2693                   cessfully. This indicates that the compatibility tests on the in-  
2694                   put argument passed successfully, and the output buffer serial-  
2695                   ized\_data and serialized\_size, have been computed and are ready  
2696                   to use.

2697                   GrB\_PANIC Unknown internal error.

2698                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2699                   opaque GraphBLAS objects (input or output) is in an invalid  
2700                   state caused by a previous execution error. Call GrB\_error() to  
2701                   access any error messages generated by the implementation.

2702                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2703                   GrB\_NULL\_POINTER serialized\_data or serialize\_size is NULL.

2704                   GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2705                   any matrix constructor.

2706                   GrB\_INSUFFICIENT\_SPACE The size of the buffer serialized\_data (provided as an input seri-  
2707                   alized\_size) was not large enough.

## 2708 Description

2709 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,  
2710 the size of the buffer pointed to by serialized\_data, provided as an input by serialized\_size, must  
2711 be of at least the number of bytes returned from GrB\_Matrix\_serializeSize. The actual size of the  
2712 serialized matrix written to serialized\_data is provided upon completion as an output written to  
2713 serialized\_size.

2714 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created  
2715 with one library implementation is not necessarily valid for deserialization with another implemen-  
2716 tation.

### 2717 4.2.5.20 Matrix\_deserialize: Deserialize a GraphBLAS matrix.

2718 Construct a new GraphBLAS matrix from a serialized object.

## 2719 C Syntax

```
GrB_Info GrB_Matrix_deserialize(GrB_Matrix *A,  
                                GrB_Type   d,  
                                const void *serialized_data,  
                                GrB_Index   serialized_size);
```

## 2720 Parameters

2721           **A** (INOUT) On a successful return, contains a handle to the newly created Graph-  
2722           BLAS matrix.

2723           **d** (IN) the type of the matrix that was serialized in `serialized_data`.  
2724           If `d` is `GrB_NULL`, the implementation must attempt to deserialize the matrix  
2725           without a provided type object.

2726   **serialized\_data** (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2727   **serialized\_size** (IN) the size of the buffer pointed to by `serialized_data` in bytes.

## 2728 Return Values

2729           **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
2730           blocking mode, this indicates that the API checks for the input  
2731           arguments passed successfully. Either way, output matrix **A** is  
2732           ready to be used in the next method of the sequence.

2733           **GrB\_PANIC** Unknown internal error.

2734           **GrB\_INVALID\_OBJECT** This is returned if `serialized_data` is invalid or corrupted.

2735           **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2736           **GrB\_UNINITIALIZED\_OBJECT** The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`  
2737           (needed for user-defined types).

2738           **GrB\_NULL\_POINTER** `serialized_data` or **A** is `NULL`.

2739           **GrB\_DOMAIN\_MISMATCH** The type given in `d` does not match the type of the matrix  
2740           serialized in `serialized_data`, or `GrB_NULL` was passed in and  
2741           the implementation is unable to construct the matrix without  
2742           the explicitly provided `GrB_Type`.

## 2743 Description

2744   Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object  
2745   pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The  
2746   domain of the matrix is given as an input in `d`, which must match the domain of the matrix serialized  
2747   in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2748   Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix  
2749   serialized in one library implementation can be deserialized by another.

2750   It is not an error to call this method more than once on the same output matrix; however, the  
2751   handle to the previously created object will be overwritten.

## 2752 4.2.6 Descriptor methods

2753 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-  
2754 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

### 2755 4.2.6.1 Descriptor\_new: Create new descriptor

2756 Creates a new (empty or default) descriptor.

## 2757 C Syntax

```
2758      GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);
```

## 2759 Parameters

2760 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2761 descriptor.

## 2762 Return Value

2763 GrB\_SUCCESS The method completed successfully.

2764 GrB\_PANIC unknown internal error.

2765 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2766 GrB\_NULL\_POINTER desc pointer is NULL.

## 2767 Description

2768 Creates a new descriptor object and returns a handle to it in `desc`. A newly created descriptor can  
2769 be populated by calls to `Descriptor_set`.

2770 It is not an error to call this method more than once on the same variable; however, the handle to  
2771 the previously created object will be overwritten.

### 2772 4.2.6.2 Descriptor\_set: Set content of descriptor

2773 Sets the content for a field for an existing descriptor.

## 2774 C Syntax

```
2775         GrB_Info GrB_Descriptor_set(GrB_Descriptor      desc,  
2776                                     GrB_Desc_Field      field,  
2777                                     GrB_Desc_Value      val);
```

## 2778 Parameters

2779 desc (IN) An existing GraphBLAS descriptor to be modified.

2780 field (IN) The field being set.

2781 val (IN) New value for the field being set.

## 2782 Return Values

2783 GrB\_SUCCESS operation completed successfully.

2784 GrB\_PANIC unknown internal error.

2785 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2786 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

2787 GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 2788 Description

2789 For a given descriptor, the GrB\_Descriptor\_set method can be called for each field in the descriptor  
2790 to set the value associated with that field. Valid values for the field parameter include the following:

2791 GrB\_OUTP refers to the output parameter (result) of the operation.

2792 GrB\_MASK refers to the mask parameter of the operation.

2793 GrB\_INP0 refers to the first input parameters of the operation (matrices and vectors).

2794 GrB\_INP1 refers to the second input parameters of the operation (matrices and vectors).

2795 Valid values for the val parameter are:

2796 GrB\_STRUCTURE Use only the structure of the stored values of the corresponding mask  
2797 (GrB\_MASK) parameter.

2798 GrB\_COMP Use the complement of the corresponding mask (GrB\_MASK) param-  
2799 eter. When combined with GrB\_STRUCTURE, the complement of the  
2800 structure of the mask is used without evaluating the values stored.

2801           GrB\_TRAN Use the transpose of the corresponding matrix parameter (valid for input  
2802           matrix parameters only).

2803           GrB\_REPLACE When assigning the masked values to the output matrix or vector, clear  
2804           the matrix first (or clear the non-masked entries). The default behavior  
2805           is to leave non-masked locations unchanged. Valid for the GrB\_OUTP  
2806           parameter only.

2807 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of GrB\_MASK,  
2808 multiple values can be set and all will apply (for example, both GrB\_COMP and GrB\_STRUCTURE).  
2809 A value for a given field may be set multiple times but will have no additional effect. Fields that  
2810 have no values set result in their default behavior, as defined in Section 3.6.

#### 2811 **4.2.7 free: Destroy an object and release its resources**

2812 Destroys a previously created GraphBLAS object and releases any resources associated with the  
2813 object.

#### 2814 **C Syntax**

2815           GrB\_Info GrB\_free(<GrB\_Object> \*obj);

#### 2816 **Parameters**

2817           obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have  
2818           been created by an explicit call to a GraphBLAS constructor. It can be any of the  
2819           opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,  
2820           binary op, unary op, or type. On successful completion of GrB\_free, obj behaves  
2821           as an uninitialized object.

#### 2822 **Return Values**

2823           GrB\_SUCCESS operation completed successfully

2824           GrB\_PANIC unknown internal error. If this return value is encountered when  
2825           in nonblocking mode, the error responsible for the panic condition  
2826           could be from any method involved in the computation of the input  
2827           object. The GrB\_error() method should be called for additional  
2828           information.

#### 2829 **Description**

2830 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime  
2831 system. A call to GrB\_free frees those resources so they are available for use by other GraphBLAS

2832 objects.

2833 The parameter passed into `GrB_free` is a handle referencing a GraphBLAS opaque object of a data  
2834 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-  
2835 structor. The behavior of a program that calls `GrB_free` on a pre-defined object is implementation  
2836 defined.

2837 After the `GrB_free` method returns, the object referenced by the input handle is destroyed and the  
2838 handle has the value `GrB_INVALID_HANDLE`. The handle can be used in subsequent GraphBLAS  
2839 methods but only after the handle has been reinitialized with a call the the appropriate `_new` or  
2840 `_dup` method.

2841 Note that unlike other GraphBLAS methods, calling `GrB_free` with an object with an invalid handle  
2842 is legal. The system may attempt to free resources that might be associated with that object, if  
2843 possible, and return normally.

2844 When using `GrB_free` it is possible to create a dangling reference to an object. This would occur  
2845 when a handle is assigned to a second variable of the same opaque type. This creates two handles  
2846 that reference the same object. If `GrB_free` is called with one of the variables, the object is destroyed  
2847 and the handle associated with the other variable no longer references a valid object. This is not an  
2848 error condition that the implementation of the GraphBLAS API can be expected to catch, hence  
2849 programmers must take care to prevent this situation from occurring.

#### 2850 4.2.8 wait: Return once an object is either *complete* or *materialized*

2851 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

#### 2852 C Syntax

```
2853      GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);
```

#### 2854 Parameters

2855 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an  
2856 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2857 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2858 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another  
2859 thread (completion) or all computing to produce `obj` by all GraphBLAS operations  
2860 in its sequence have finished (materialization).

2861 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the  
2862 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or  
2863 `GrB_MATERIALIZE`.

## 2864 Return values

2865                   GrB\_SUCCESS operation completed successfully.

2866 GrB\_INDEX\_OUT\_OF\_BOUNDS an index out-of-bounds execution error happened during com-  
2867 pletion of pending operations.

2868                   GrB\_OUT\_OF\_MEMORY and out-of-memory execution error happened during completion  
2869 of pending operations.

2870 GrB\_UNINITIALIZED\_OBJECT object has not been initialized by a call to the respective \*\_new,  
2871 or other constructor, method.

2872                   GrB\_PANIC unknown internal error.

2873                   GrB\_INVALID\_VALUE method called with a GrB\_WaitMode other than GrB\_COMPLETE  
2874 GrB\_MATERIALIZE.

## 2875 Description

2876 On successful return from GrB\_wait(), the input object, **obj** is in one of two states depending on  
2877 the mode of GrB\_wait:

- 2878       • *complete*: **obj** can be used in a happens-before relation, so in a properly synchronized program  
2879       it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another  
2880       thread. This result occurs when the mode parameter is set to GrB\_COMPLETE.
- 2881       • *materialized*: **obj** is *complete*, but in addition, no further computing will be carried out on  
2882       behalf of **obj** and error information is available. This result occurs when the mode parameter  
2883       is set to GrB\_MATERIALIZE.

2884 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,  
2885 GrB\_wait(**obj**,mode) has no effect when called in blocking mode.

2886 In non-blocking mode, the status of any pending method calls, other than those associated with pro-  
2887 ducing the *complete* or *materialized* state of **obj**, are not impacted by the call to GrB\_wait(**obj**,mode).  
2888 Methods in the sequence for **obj**, however, most likely would be impacted by a call to GrB\_wait(**obj**,mode);  
2889 especially in the case of the *materialized* mode for which any computing on behalf of **obj** must be  
2890 finished prior to the return from GrB\_wait(**obj**,mode).

## 2891 4.2.9 error: Retrieve an error string

2892 Retrieve an error-message about any errors encountered during the processing associated with an  
2893 object.



## 2894 C Syntax

```
2895         GrB_Info GrB_error(const char      **error,  
2896                             const GrB_Object  obj);
```

## 2897 Parameters

2898       error (OUT) A pointer to a null-terminated string. The contents of the string are im-  
2899       plementation defined.

2900       obj (IN) An existing GraphBLAS object. The object must have been created by an  
2901       explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2902       objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2903       or type.

## 2904 Return value

2905       GrB\_SUCCESS operation completed successfully.

2906       GrB\_UNINITIALIZED\_OBJECT object has not been initialized by a call to the respective \*\_new,  
2907       or other constructor, method.

2908       GrB\_PANIC unknown internal error.

## 2909 Description

2910 This method retrieves a message related to any errors that were encountered during the last Graph-  
2911 BLAS method that had the opaque GraphBLAS object, obj, as an OUT or INOUT parameter.  
2912 The function returns a pointer to a null-terminated string and the contents of that string are  
2913 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error  
2914 string. The string that is returned is owned by obj and will be valid until the next time obj is  
2915 used as an OUT or INOUT parameter or the object is freed by a call to GrB\_free(obj). This is a  
2916 thread-safe function. It can be safely called by multiple threads for the same object in a race-free  
2917 program.

## 2918 4.3 GraphBLAS operations

2919 The GraphBLAS operations are defined in the GraphBLAS math specification and summarized in  
2920 Table 4.1. In addition to methods that implement these fundamental GraphBLAS operations, we  
2921 support a number of variants that have been found to be especially useful in algorithm development.  
2922 A flowchart of the overall behavior of a GraphBLAS operation is shown in Figure 4.1.

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	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

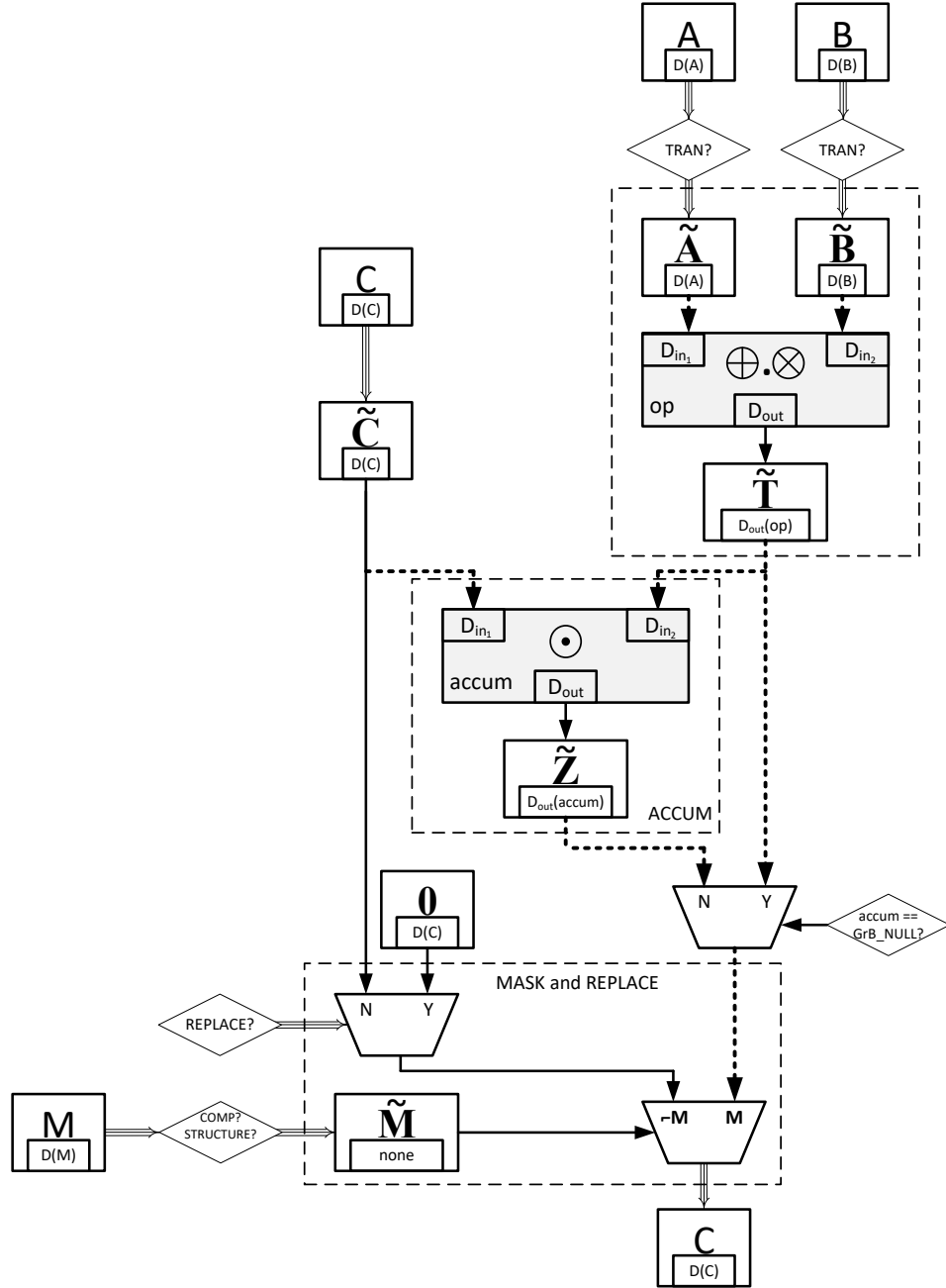


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows ( $\Rightarrow$ ) denote where “as if copy” takes place (including both collections and descriptor settings). The bold, dotted arrows indicate where casting may occur between different domains.

## 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 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:

$$\mathbf{m} = \begin{cases} \langle N, \{\mathbf{ind}(\mathbf{v})\} \rangle, & \text{if GrB\_STRUCTURE is specified,} \\ \langle N, \{i : (\mathbf{bool})v_i = \mathbf{true}\} \rangle, & \text{otherwise} \end{cases}$$

where  $(\mathbf{bool})v_i$  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:

$$\mathbf{M} = \begin{cases} \langle M, N, \{\mathbf{ind}(\mathbf{A})\} \rangle, & \text{if GrB\_STRUCTURE is specified,} \\ \langle M, N, \{(i, j) : (\mathbf{bool})A_{ij} = \mathbf{true}\} \rangle, & \text{otherwise} \end{cases}$$

where  $(\mathbf{bool})A_{ij}$  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**).

To support the detection of as many cases of uninitialized objects as possible, it is strongly recommended to initialize all GraphBLAS objects to the predefined value **GrB\_INVALID\_HANDLE** at the point of their declaration, as shown in the following examples:

```
GrB_Type      type = GrB_INVALID_HANDLE;
GrB_Semiring  semiring = GrB_INVALID_HANDLE;
GrB_Matrix    matrix = GrB_INVALID_HANDLE;
```

## Compliance

We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations. That is, for each operation we give a recipe for producing its outcome. Any implementation that produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error model (Section 2.6) is a conforming implementation.

### 2989 4.3.1 mxm: Matrix-matrix multiply

2990 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

#### 2991 C Syntax

```
2992         GrB_Info GrB_mxm(GrB_Matrix      C,  
2993                         const GrB_Matrix  Mask,  
2994                         const GrB_BinaryOp accum,  
2995                         const GrB_Semiring op,  
2996                         const GrB_Matrix  A,  
2997                         const GrB_Matrix  B,  
2998                         const GrB_Descriptor desc);
```

#### 2999 Parameters

3000 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3001 that may be accumulated with the result of the matrix product. On output, the  
3002 matrix holds the results of the operation.

3003 Mask (IN) An optional “write” mask that controls which results from this operation are  
3004 stored into the output matrix C. The mask dimensions must match those of the  
3005 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
3006 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
3007 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3008 dimensions of C), GrB\_NULL should be specified.

3009 accum (IN) An optional binary operator used for accumulating entries into existing C  
3010 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3011 specified.

3012 op (IN) The semiring used in the matrix-matrix multiply.

3013 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
3014 multiplication.

3015 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
3016 multiplication.

3017 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
3018 should be specified. Non-default field/value pairs are listed as follows:

3019

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.

## 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 the operation.

**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).

**GrB\_DIMENSION\_MISMATCH** Mask and/or matrix dimensions are incompatible.

**GrB\_DOMAIN\_MISMATCH** The domains of the various 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\_mxm** computes the matrix product  $C = A \oplus . \otimes B$  or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \oplus . \otimes B)$  (where matrices A and B can be optionally transposed). Logically, this operation occurs in three steps:

**Setup** The internal matrices and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

3046 **Compute** The indicated computations are carried out.

3047 **Output** The result is written into the output matrix, possibly under control of a mask.

3048 Up to four argument matrices are used in the `GrB_mxm` operation:

- 3049 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3050 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)
- 3051 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3052 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3053 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for  
3054 domain compatibility as follows:

- 3055 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
3056 must be from one of the pre-defined types of Table 3.2.
- 3057 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3058 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3059 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3060 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3061 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
3062 of the accumulation operator.

3063 Two domains are compatible with each other if values from one domain can be cast to values in  
3064 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
3065 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
3066 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch  
3067 error listed above is returned.

3068 From the argument matrices, the internal matrices and mask used in the computation are formed  
3069 ( $\leftarrow$  denotes copy):

- 3070 1. Matrix  $\tilde{C} \leftarrow C$ .
- 3071 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:
  - 3072 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
3073  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 3074 (b) If `Mask`  $\neq$  `GrB_NULL`,
    - 3075 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
3076  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,



3077           ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
3078            $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\}\rangle.$

3079           (c) If  $\mathbf{desc}[\mathbf{GrB\_MASK}].\mathbf{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}.$

3080       3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}.$

3081       4. Matrix  $\widetilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}.$

3082       The internal matrices and masks are checked for dimension compatibility. The following conditions  
3083       must hold:

3084       1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}}).$

3085       2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}}).$

3086       3.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}}).$

3087       4.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{B}}).$

3088       5.  $\mathbf{ncols}(\widetilde{\mathbf{A}}) = \mathbf{nrows}(\widetilde{\mathbf{B}}).$

3089       If any compatibility rule above is violated, execution of  $\mathbf{GrB\_mxm}$  ends and the dimension mismatch  
3090       error listed above is returned.

3091       From this point forward, in  $\mathbf{GrB\_NONBLOCKING}$  mode, the method can optionally exit with  
3092        $\mathbf{GrB\_SUCCESS}$  return code and defer any computation and/or execution error codes.

3093       We are now ready to carry out the matrix multiplication and any additional associated operations.  
3094       We describe this in terms of two intermediate matrices:

- 3095       •  $\widetilde{\mathbf{T}}$ : The matrix holding the product of matrices  $\widetilde{\mathbf{A}}$  and  $\widetilde{\mathbf{B}}$ .
- 3096       •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3097       The intermediate matrix  $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{B}}), \{(i, j, T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:, j)) \neq \emptyset\}\rangle$  is created. The value of each of its elements is computed by

$$3099 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\widetilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:, j))} (\widetilde{\mathbf{A}}(i, k) \otimes \widetilde{\mathbf{B}}(k, j)),$$

3100       where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring  $\mathbf{op}$ , respectively.

3101       The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3102       • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 3103       • If  $\mathbf{accum}$  is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$3104 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\}\rangle.$$

3105 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 3106 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned}
 3107 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 3108 \\
 3109 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 3110 \\
 3111 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),
 \end{aligned}$$

3112 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3113 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3114 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3115 mask which acts as a “write mask”.

- 3116 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3117 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3118 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3119 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3120 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3121 mask are unchanged:

$$3122 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3123 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3124 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3125 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 3126 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3127 sequence.

### 3128 4.3.2 vxm: Vector-matrix multiply

3129 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

### 3130 C Syntax

```

3131      GrB_Info GrB_vxm(GrB_Vector      w,
3132                      const GrB_Vector mask,
3133                      const GrB_BinaryOp accum,
3134                      const GrB_Semiring op,
3135                      const GrB_Vector u,
3136                      const GrB_Matrix A,
3137                      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 vector-matrix product. 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) Semiring used in the vector-matrix multiply.

**u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the multiplication.

**A** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the multiplication.

**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> .
<b>A</b>	<b>GrB_INP1</b>	<b>GrB_TRAN</b>	Use transpose of <b>A</b> 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.

3167        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 3168        GraphBLAS objects (input or output) is in an invalid state caused  
 3169        by a previous execution error. Call `GrB_error()` to access any error  
 3170        messages generated by the implementation.

3171        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3172 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 3173        a call to `new` (or `dup` for matrix or vector parameters).

3174 **GrB\_DIMENSION\_MISMATCH** Mask, vector, and/or matrix dimensions are incompatible.

3175        **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors/matrices are incompatible with  
 3176        the corresponding domains of the semiring or accumulation opera-  
 3177        tor, or the mask's domain is not compatible with `bool` (in the case  
 3178        where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3179 **Description**

3180 **GrB\_vxm** computes the vector-matrix product  $\mathbf{w}^T = \mathbf{u}^T \oplus . \otimes \mathbf{A}$ , or, if an optional binary accu-  
 3181        mulation operator ( $\odot$ ) is provided,  $\mathbf{w}^T = \mathbf{w}^T \odot (\mathbf{u}^T \oplus . \otimes \mathbf{A})$  (where matrix  $\mathbf{A}$  can be optionally  
 3182        transposed). Logically, this operation occurs in three steps:

3183        **Setup** The internal vectors, matrices and mask used in the computation are formed and their  
 3184        domains/dimensions are tested for compatibility.

3185        **Compute** The indicated computations are carried out.

3186        **Output** The result is written into the output vector, possibly under control of a mask.

3187 Up to four argument vectors or matrices are used in the **GrB\_vxm** operation:

- 3188        1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3189        2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3190        3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3191        4.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3192 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
 3193 tested for domain compatibility as follows:

- 3194        1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3195        must be from one of the pre-defined types of Table 3.2.
- 3196        2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the semiring.

- 3197 3.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the semiring.
- 3198 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring.
- 3199 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$
- 3200 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$
- 3201 of the accumulation operator.

3202 Two domains are compatible with each other if values from one domain can be cast to values in  
 3203 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3204 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
 3205 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch  
 3206 error listed above is returned.

3207 From the argument vectors and matrices, the internal matrices and mask used in the computation  
 3208 are formed ( $\leftarrow$  denotes copy):

- 3209 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3210 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3211 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3212 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3213 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 3214 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 3215 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3216 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 3217 4. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

3218 The internal matrices and masks are checked for shape compatibility. The following conditions  
 3219 must hold:

- 3220 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$ .
- 3221 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .
- 3222 3.  $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

3223 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch  
 3224 error listed above is returned.

3225 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 3226 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3227 We are now ready to carry out the vector-matrix multiplication and any additional associated  
 3228 operations. We describe this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$ : The vector holding the product of vector  $\tilde{\mathbf{u}}^T$  and matrix  $\tilde{\mathbf{A}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{ncols}(\tilde{\mathbf{A}}), \{(j, t_j) : \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j)) \neq \emptyset\} \rangle$  is created. The value of each of its elements is computed by

$$t_j = \bigoplus_{k \in \text{ind}(\tilde{\mathbf{u}}) \cap \text{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring  $\text{op}$ , respectively.

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- If  $\text{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- If  $\text{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  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  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  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  $\text{GrB\_BLOCKING}$  mode, the method exits with return value  $\text{GrB\_SUCCESS}$  and the new content of vector  $\mathbf{w}$  is as defined above and fully computed. In  $\text{GrB\_NONBLOCKING}$  mode, the method exits with return value  $\text{GrB\_SUCCESS}$  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.

### 3262 4.3.3 mxv: Matrix-vector multiply

3263 Multiplies a matrix by a vector on a semiring. The result is a vector.

#### 3264 C Syntax

```
3265     GrB_Info GrB_mxv(GrB_Vector      w,  
3266                     const GrB_Vector mask,  
3267                     const GrB_BinaryOp accum,  
3268                     const GrB_Semiring op,  
3269                     const GrB_Matrix  A,  
3270                     const GrB_Vector  u,  
3271                     const GrB_Descriptor desc);
```

#### 3272 Parameters

3273 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3274 that may be accumulated with the result of the matrix-vector product. On output,  
3275 this vector holds the results of the operation.

3276 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3277 stored into the output vector **w**. The mask dimensions must match those of the  
3278 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3279 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3280 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3281 dimensions of **w**), **GrB\_NULL** should be specified.

3282 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3283 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3284 specified.

3285 **op** (IN) Semiring used in the vector-matrix multiply.

3286 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
3287 multiplication.

3288 **u** (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
3289 multiplication.

3290 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
3291 should be specified. Non-default field/value pairs are listed as follows:

3292

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 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\_mvx** computes the matrix-vector product  $w = A \oplus . \otimes u$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot (A \oplus . \otimes u)$  (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.



3320     **Output** The result is written into the output vector, possibly under control of a mask.

3321 Up to four argument vectors or matrices are used in the `GrB_mvx` operation:

- 3322     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3323     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3324     3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 3325     4.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

3326 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
 3327 tested for domain compatibility as follows:

- 3328     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3329         must be from one of the pre-defined types of Table 3.2.
- 3330     2.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the semiring.
- 3331     3.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the semiring.
- 3332     4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring.
- 3333     5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3334         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$   
 3335         of the accumulation operator.

3336 Two domains are compatible with each other if values from one domain can be cast to values in  
 3337 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3338 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
 3339 If any compatibility rule above is violated, execution of `GrB_mvx` ends and the domain mismatch  
 3340 error listed above is returned.

3341 From the argument vectors and matrices, the internal matrices and mask used in the computation  
 3342 are formed ( $\leftarrow$  denotes copy):

- 3343     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3344     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3345         (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3346         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3347             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 3348             ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 3349         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3350     3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

3351 4. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3352 The internal matrices and masks are checked for shape compatibility. The following conditions  
3353 must hold:

- 3354 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$ .
- 3355 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 3356 3.  $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

3357 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the dimension mismatch  
3358 error listed above is returned.

3359 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3360 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3361 We are now ready to carry out the matrix-vector multiplication and any additional associated  
3362 operations. We describe this in terms of two intermediate vectors:

- 3363 •  $\tilde{\mathbf{t}}$ : The vector holding the product of matrix  $\tilde{\mathbf{A}}$  and vector  $\tilde{\mathbf{u}}$ .
- 3364 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3365 The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \{(i, t_i) : \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{u}}) \neq \emptyset\} \rangle$  is created.  
3366 The value of each of its elements is computed by

$$3367 \quad t_i = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3368 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring `op`, respectively.

3369 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3370 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3371 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3372 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3373 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3374 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3375 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3376 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3377 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3378 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3379 \end{aligned}$$

3380 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3381 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3382 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3383 mask which acts as a “write mask”.

- 3384 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3385 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3386 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3387 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3388 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3389 mask are unchanged:

$$3390 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3391 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3392 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3393 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3394 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3395 sequence.

#### 3396 4.3.4 eWiseMult: Element-wise multiplication

3397 **Note:** The difference between eWiseAdd and eWiseMult is not about the element-wise operation  
 3398 but how the index sets are treated. eWiseAdd returns an object whose indices are the “union” of  
 3399 the indices of the inputs whereas eWiseMult returns an object whose indices are the “intersection”  
 3400 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3401 the set of values from the resulting index set.

##### 3402 4.3.4.1 eWiseMult: Vector variant

3403 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-  
 3404 ducing a third vector as result.

#### 3405 C Syntax

```
3406      GrB_Info GrB_eWiseMult(GrB_Vector      w,
3407                             const GrB_Vector mask,
3408                             const GrB_BinaryOp accum,
3409                             const GrB_Semiring op,
3410                             const GrB_Vector u,
3411                             const GrB_Vector v,
3412                             const GrB_Descriptor desc);
3413
```

```

3414 GrB_Info GrB_eWiseMult(GrB_Vector      w,
3415                        const GrB_Vector  mask,
3416                        const GrB_BinaryOp accum,
3417                        const GrB_Monoid   op,
3418                        const GrB_Vector  u,
3419                        const GrB_Vector  v,
3420                        const GrB_Descriptor desc);
3421
3422 GrB_Info GrB_eWiseMult(GrB_Vector      w,
3423                        const GrB_Vector  mask,
3424                        const GrB_BinaryOp accum,
3425                        const GrB_BinaryOp op,
3426                        const GrB_Vector  u,
3427                        const GrB_Vector  v,
3428                        const GrB_Descriptor desc);

```

## 3429 Parameters

3430 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3431 that may be accumulated with the result of the element-wise operation. On output,  
3432 this vector holds the results of the operation.

3433 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3434 stored into the output vector **w**. The mask dimensions must match those of the  
3435 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3436 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3437 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3438 dimensions of **w**), **GrB\_NULL** should be specified.

3439 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3440 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3441 specified.

3442 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
3443 operation. Depending on which type is passed, the following defines the binary  
3444 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3445 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3446 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-  
3447 nored.

3448 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  
3449 is ignored.

3450 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3451 operation.

3452 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3453 operation.

3454 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 3455 should be specified. Non-default field/value pairs are listed as follows:  
 3456

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> .

## 3458 Return Values

3459 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 3460 blocking mode, this indicates that the compatibility tests on di-  
 3461 mensions and domains for the input arguments passed successfully.  
 3462 Either way, output vector **w** is ready to be used in the next method  
 3463 of the sequence.

3464 **GrB\_PANIC** Unknown internal error.

3465 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 3466 GraphBLAS objects (input or output) is in an invalid state caused  
 3467 by a previous execution error. Call **GrB\_error()** to access any error  
 3468 messages generated by the implementation.

3469 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3470 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 3471 a call to **new** (or **dup** for vector parameters).

3472 **GrB\_DIMENSION\_MISMATCH** Mask or vector dimensions are incompatible.

3473 **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the cor-  
 3474 responding domains of the binary operator (**op**) or accumulation  
 3475 operator, or the mask's domain is not compatible with **bool** (in the  
 3476 case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

## 3477 Description

3478 This variant of **GrB\_eWiseMult** computes the element-wise “product” of two GraphBLAS vectors:  
 3479  $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$ .  
 3480 Logically, this operation occurs in three steps:

3481     **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3482             and dimensions are tested for compatibility.

3483     **Compute** The indicated computations are carried out.

3484     **Output** The result is written into the output vector, possibly under control of a mask.

3485     Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3486     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3487     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3488     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3489     4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3490     The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)  
 3491     are tested for domain compatibility as follows:

- 3492     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3493         must be from one of the pre-defined types of Table 3.2.
- 3494     2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3495     3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3496     4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3497     5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3498         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
 3499         the accumulation operator.

3500     Two domains are compatible with each other if values from one domain can be cast to values in  
 3501     the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3502     compatible with each other. A domain from a user-defined type is only compatible with itself. If any  
 3503     compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3504     error listed above is returned.

3505     From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3506     denotes copy):

- 3507     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3508     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3509         (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3510         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3511             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,

3512 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .

3513 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .

3514 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3515 4. Vector  $\widetilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3516 The internal vectors and mask are checked for dimension compatibility. The following conditions  
3517 must hold:

3518 1.  $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}}) = \mathbf{size}(\widetilde{\mathbf{u}}) = \mathbf{size}(\widetilde{\mathbf{v}})$ .

3519 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
3520 mismatch error listed above is returned.

3521 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3522 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3523 We are now ready to carry out the element-wise “product” and any additional associated operations.  
3524 We describe this in terms of two intermediate vectors:

- 3525 •  $\widetilde{\mathbf{t}}$ : The vector holding the element-wise “product” of  $\widetilde{\mathbf{u}}$  and vector  $\widetilde{\mathbf{v}}$ .
- 3526 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3527 The intermediate vector  $\widetilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\widetilde{\mathbf{u}}), \{(i, t_i) : \mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The  
3528 value of each of its elements is computed by:

$$3529 \quad t_i = (\widetilde{\mathbf{u}}(i) \otimes \widetilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{v}}))$$

3530 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3531 • If `accum = GrB_NULL`, then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 3532 • If `accum` is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$3533 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

3534 The values of the elements of  $\widetilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3535 indices in  $\widetilde{\mathbf{w}}$  and  $\widetilde{\mathbf{t}}$ .

$$\begin{aligned} 3536 \quad z_i &= \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})), \\ 3537 \quad z_i &= \widetilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))), \\ 3538 \quad z_i &= \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))), \\ 3539 \quad & \\ 3540 \end{aligned}$$

3541 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3542 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3543 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3544 mask which acts as a “write mask”.

- 3545 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3546 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3547 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- 3548 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3549 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3550 mask are unchanged:

$$3551 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

3552 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3553 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3554 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3555 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3556 sequence.

#### 3557 4.3.4.2 eWiseMult: Matrix variant

3558 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-  
 3559 ducing a third matrix as result.

### 3560 C Syntax

```

3561     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3562                           const GrB_Matrix Mask,
3563                           const GrB_BinaryOp accum,
3564                           const GrB_Semiring op,
3565                           const GrB_Matrix A,
3566                           const GrB_Matrix B,
3567                           const GrB_Descriptor desc);
3568
3569     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3570                           const GrB_Matrix Mask,
3571                           const GrB_BinaryOp accum,
3572                           const GrB_Monoid op,
3573                           const GrB_Matrix A,
3574                           const GrB_Matrix B,
3575                           const GrB_Descriptor desc);
3576
3577     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
```



```

3578         const GrB_Matrix      Mask,
3579         const GrB_BinaryOp    accum,
3580         const GrB_BinaryOp    op,
3581         const GrB_Matrix      A,
3582         const GrB_Matrix      B,
3583         const GrB_Descriptor   desc);

```

## 3584 Parameters

3585 **C (INOUT)** An existing GraphBLAS matrix. On input, the matrix provides values  
3586 that may be accumulated with the result of the element-wise operation. On output,  
3587 the matrix holds the results of the operation.

3588 **Mask (IN)** An optional “write” mask that controls which results from this operation are  
3589 stored into the output matrix C. The mask dimensions must match those of the  
3590 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
3591 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
3592 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
3593 dimensions of C), GrB\_NULL should be specified.

3594 **accum (IN)** An optional binary operator used for accumulating entries into existing C  
3595 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3596 specified.

3597 **op (IN)** The semiring, monoid, or binary operator used in the element-wise “product”  
3598 operation. Depending on which type is passed, the following defines the binary  
3599 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3600 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3601 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-  
3602 nored.

3603 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  
3604 is ignored.

3605 **A (IN)** The GraphBLAS matrix holding the values for the left-hand matrix in the  
3606 operation.

3607 **B (IN)** The GraphBLAS matrix holding the values for the right-hand matrix in the  
3608 operation.

3609 **desc (IN)** An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
3610 should be specified. Non-default field/value pairs are listed as follows:  
3611

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.

## 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 the operation.

**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).

**GrB\_DIMENSION\_MISMATCH** Mask and/or matrix dimensions are incompatible.

**GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the corresponding domains of the binary operator (`op`) 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

This variant of `GrB_eWiseMult` computes the element-wise “product” of two GraphBLAS matrices:  $C = A \otimes B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$ . Logically, this operation occurs in three steps:

**Setup** The internal matrices and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

3638 **Compute** The indicated computations are carried out.

3639 **Output** The result is written into the output matrix, possibly under control of a mask.

3640 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3641 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3642 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)
- 3643 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3644 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3645 The argument matrices, the “product” operator (`op`), and the accumulation operator (if provided)  
3646 are tested for domain compatibility as follows:

- 3647 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
3648 must be from one of the pre-defined types of Table 3.2.
- 3649 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 3650 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 3651 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 3652 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3653 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
3654 the accumulation operator.

3655 Two domains are compatible with each other if values from one domain can be cast to values in  
3656 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
3657 compatible with each other. A domain from a user-defined type is only compatible with itself. If any  
3658 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
3659 error listed above is returned.

3660 From the argument matrices, the internal matrices and mask used in the computation are formed  
3661 ( $\leftarrow$  denotes copy):

- 3662 1. Matrix  $\tilde{C} \leftarrow C$ .
- 3663 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:
  - 3664 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
3665  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 3666 (b) If `Mask`  $\neq$  `GrB_NULL`,
    - 3667 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
3668  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,

3669           ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
3670            $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\}\rangle.$   
3671       (c) If  $\mathbf{desc}[\mathbf{GrB\_MASK}].\mathbf{GrB\_COMP}$  is set, then  $\tilde{\mathbf{M}} \leftarrow \neg\tilde{\mathbf{M}}.$

3672   3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}.$

3673   4. Matrix  $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}.$

3674   The internal matrices and masks are checked for dimension compatibility. The following conditions  
3675   must hold:

3676       1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}}).$

3677       2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}}).$

3678   If any compatibility rule above is violated, execution of  $\mathbf{GrB\_eWiseMult}$  ends and the dimension  
3679   mismatch error listed above is returned.

3680   From this point forward, in  $\mathbf{GrB\_NONBLOCKING}$  mode, the method can optionally exit with  
3681    $\mathbf{GrB\_SUCCESS}$  return code and defer any computation and/or execution error codes.

3682   We are now ready to carry out the element-wise “product” and any additional associated operations.  
3683   We describe this in terms of two intermediate matrices:

- 3684       •  $\tilde{\mathbf{T}}$ : The matrix holding the element-wise product of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3685       •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3686   The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\}\rangle$   
3687   is created. The value of each of its elements is computed by

$$3688 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3689   The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3690       • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3691       • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3692 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

3693   The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
3694   indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$3695 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$3696 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$3697 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

3700   where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3701 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3702 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3703 mask which acts as a “write mask”.

- 3704 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3705 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3706 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3707 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3708 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3709 mask are unchanged:

$$3710 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3711 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3712 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3713 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 3714 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3715 sequence.

### 3716 4.3.5 eWiseAdd: Element-wise addition

3717 **Note:** The difference between eWiseAdd and eWiseMult is not about the element-wise operation  
 3718 but how the index sets are treated. eWiseAdd returns an object whose indices are the “union” of  
 3719 the indices of the inputs whereas eWiseMult returns an object whose indices are the “intersection”  
 3720 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3721 the set of values from the resulting index set.

#### 3722 4.3.5.1 eWiseAdd: Vector variant

3723 Perform element-wise (general) addition on the elements of two vectors, producing a third vector  
 3724 as result.

### 3725 C Syntax

```
3726      GrB_Info GrB_eWiseAdd(GrB_Vector      w,
3727                           const GrB_Vector mask,
3728                           const GrB_BinaryOp accum,
3729                           const GrB_Semiring op,
3730                           const GrB_Vector u,
3731                           const GrB_Vector v,
3732                           const GrB_Descriptor desc);
3733
```

```

3734 GrB_Info GrB_eWiseAdd(GrB_Vector      w,
3735                        const GrB_Vector mask,
3736                        const GrB_BinaryOp accum,
3737                        const GrB_Monoid op,
3738                        const GrB_Vector u,
3739                        const GrB_Vector v,
3740                        const GrB_Descriptor desc);
3741
3742 GrB_Info GrB_eWiseAdd(GrB_Vector      w,
3743                        const GrB_Vector mask,
3744                        const GrB_BinaryOp accum,
3745                        const GrB_BinaryOp op,
3746                        const GrB_Vector u,
3747                        const GrB_Vector v,
3748                        const GrB_Descriptor desc);

```

## 3749 Parameters

3750 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3751 that may be accumulated with the result of the element-wise operation. On output,  
3752 this vector holds the results of the operation.

3753 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3754 stored into the output vector **w**. The mask dimensions must match those of the  
3755 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3756 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3757 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
3758 dimensions of **w**), **GrB\_NULL** should be specified.

3759 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3760 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3761 specified.

3762 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3763 operation. Depending on which type is passed, the following defines the binary  
3764 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3765 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3766 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-  
3767 nored.

3768 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-  
3769 nary op and additive identity are ignored.

3770 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3771 operation.

3772  $v$  (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3773 operation.

3774 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 3775 should be specified. Non-default field/value pairs are listed as follows:  
 3776

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> .

## 3778 Return Values

3779 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 3780 blocking mode, this indicates that the compatibility tests on di-  
 3781 mensions and domains for the input arguments passed successfully.  
 3782 Either way, output vector **w** is ready to be used in the next method  
 3783 of the sequence.

3784 **GrB\_PANIC** Unknown internal error.

3785 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 3786 GraphBLAS objects (input or output) is in an invalid state caused  
 3787 by a previous execution error. Call **GrB\_error()** to access any error  
 3788 messages generated by the implementation.

3789 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3790 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 3791 a call to **new** (or **dup** for vector parameters).

3792 **GrB\_DIMENSION\_MISMATCH** Mask or vector dimensions are incompatible.

3793 **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the cor-  
 3794 responding domains of the binary operator (**op**) or accumulation  
 3795 operator, or the mask's domain is not compatible with **bool** (in the  
 3796 case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

## 3797 Description

3798 This variant of **GrB\_eWiseAdd** computes the element-wise “sum” of two GraphBLAS vectors:  $w =$   
 3799  $u \oplus v$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot (u \oplus v)$ . Logically,  
 3800 this operation occurs in three steps:

3801     **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3802             and dimensions are tested for compatibility.

3803     **Compute** The indicated computations are carried out.

3804     **Output** The result is written into the output vector, possibly under control of a mask.

3805     Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3806     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3807     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3808     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3809     4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3810     The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are  
 3811     tested for domain compatibility as follows:

- 3812     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3813         must be from one of the pre-defined types of Table 3.2.
- 3814     2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3815     3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3816     4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3817     5.  $\mathbf{D}(\mathbf{u})$  and  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3818     6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3819         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
 3820         the accumulation operator.

3821     Two domains are compatible with each other if values from one domain can be cast to values in  
 3822     the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3823     compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 3824     any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch  
 3825     error listed above is returned.

3826     From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3827     denotes copy):

- 3828     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3829     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:  
 3830         (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .



3831 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
3832 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
3833 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
3834 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
3835 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .  
3836 4. Vector  $\widetilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3837 The internal vectors and mask are checked for dimension compatibility. The following conditions  
3838 must hold:

3839 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}}) = \text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{v}})$ .

3840 If any compatibility rule above is violated, execution of  $\text{GrB\_eWiseAdd}$  ends and the dimension  
3841 mismatch error listed above is returned.

3842 From this point forward, in  $\text{GrB\_NONBLOCKING}$  mode, the method can optionally exit with  
3843  $\text{GrB\_SUCCESS}$  return code and defer any computation and/or execution error codes.

3844 We are now ready to carry out the element-wise “sum” and any additional associated operations.  
3845 We describe this in terms of two intermediate vectors:

- 3846 •  $\widetilde{\mathbf{t}}$ : The vector holding the element-wise “sum” of  $\widetilde{\mathbf{u}}$  and vector  $\widetilde{\mathbf{v}}$ .
- 3847 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3848 The intermediate vector  $\widetilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\widetilde{\mathbf{u}}), \{(i, t_i) : \text{ind}(\widetilde{\mathbf{u}}) \cup \text{ind}(\widetilde{\mathbf{v}}) \neq \emptyset\} \rangle$  is created. The  
3849 value of each of its elements is computed by:

$$\begin{aligned}
3850 \quad t_i &= (\widetilde{\mathbf{u}}(i) \oplus \widetilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\widetilde{\mathbf{u}}) \cap \text{ind}(\widetilde{\mathbf{v}})) \\
3851 \quad t_i &= \widetilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\widetilde{\mathbf{u}}) - (\text{ind}(\widetilde{\mathbf{u}}) \cap \text{ind}(\widetilde{\mathbf{v}}))) \\
3852 \quad t_i &= \widetilde{\mathbf{v}}(i), \forall i \in (\text{ind}(\widetilde{\mathbf{v}}) - (\text{ind}(\widetilde{\mathbf{u}}) \cap \text{ind}(\widetilde{\mathbf{v}})))
\end{aligned}$$

3855 where the difference operator in the previous expressions refers to set difference.

3856 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3857 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 3858 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$3859 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

3860 The values of the elements of  $\widetilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3861 indices in  $\widetilde{\mathbf{w}}$  and  $\widetilde{\mathbf{t}}$ .

$$\begin{aligned}
3862 \quad z_i &= \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\widetilde{\mathbf{t}}) \cap \text{ind}(\widetilde{\mathbf{w}})), \\
3863 \quad z_i &= \widetilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\widetilde{\mathbf{w}}) - (\text{ind}(\widetilde{\mathbf{t}}) \cap \text{ind}(\widetilde{\mathbf{w}}))), \\
3864 \quad z_i &= \widetilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\widetilde{\mathbf{t}}) - (\text{ind}(\widetilde{\mathbf{t}}) \cap \text{ind}(\widetilde{\mathbf{w}}))), \\
3865 \quad & \\
3866 \quad &
\end{aligned}$$

3867 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3868 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3869 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3870 mask which acts as a “write mask”.

- 3871 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3872 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3873 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3874 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3875 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3876 mask are unchanged:

$$3877 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3878 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3879 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3880 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3881 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3882 sequence.

#### 3883 4.3.5.2 eWiseAdd: Matrix variant

3884 Perform element-wise (general) addition on the elements of two matrices, producing a third matrix  
 3885 as result.

#### 3886 C Syntax

```

3887     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3888                           const GrB_Matrix Mask,
3889                           const GrB_BinaryOp accum,
3890                           const GrB_Semiring op,
3891                           const GrB_Matrix A,
3892                           const GrB_Matrix B,
3893                           const GrB_Descriptor desc);
3894
3895     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3896                           const GrB_Matrix Mask,
3897                           const GrB_BinaryOp accum,
3898                           const GrB_Monoid op,
3899                           const GrB_Matrix A,
3900                           const GrB_Matrix B,
3901                           const GrB_Descriptor desc);
3902
3903     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
```

```

3904         const GrB_Matrix      Mask,
3905         const GrB_BinaryOp     accum,
3906         const GrB_BinaryOp     op,
3907         const GrB_Matrix      A,
3908         const GrB_Matrix      B,
3909         const GrB_Descriptor   desc);

```

## 3910 Parameters

3911 **C (INOUT)** An existing GraphBLAS matrix. On input, the matrix provides values  
3912 that may be accumulated with the result of the element-wise operation. On output,  
3913 the matrix holds the results of the operation.

3914 **Mask (IN)** An optional “write” mask that controls which results from this operation are  
3915 stored into the output matrix C. The mask dimensions must match those of the  
3916 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
3917 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
3918 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
3919 dimensions of C), GrB\_NULL should be specified.

3920 **accum (IN)** An optional binary operator used for accumulating entries into existing C  
3921 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3922 specified.

3923 **op (IN)** The semiring, monoid, or binary operator used in the element-wise “sum”  
3924 operation. Depending on which type is passed, the following defines the binary  
3925 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3926 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3927 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-  
3928 nored.

3929 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-  
3930 nary op and additive identity are ignored.

3931 **A (IN)** The GraphBLAS matrix holding the values for the left-hand matrix in the  
3932 operation.

3933 **B (IN)** The GraphBLAS matrix holding the values for the right-hand matrix in the  
3934 operation.

3935 **desc (IN)** An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
3936 should be specified. Non-default field/value pairs are listed as follows:

3937

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.

## 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 the operation.

**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).

**GrB\_DIMENSION\_MISMATCH** Mask and/or matrix dimensions are incompatible.

**GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the corresponding domains of the binary operator (`op`) 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

This variant of `GrB_eWiseAdd` computes the element-wise “sum” of two GraphBLAS matrices:  $C = A \oplus B$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot (A \oplus B)$ . Logically, this operation occurs in three steps:

**Setup** The internal matrices and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

3964 **Compute** The indicated computations are carried out.

3965 **Output** The result is written into the output matrix, possibly under control of a mask.

3966 Up to four argument matrices are used in the `GrB_eWiseAdd` operation:

- 3967 1.  $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 3968 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)
- 3969 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 3970 4.  $\mathbf{B} = \langle \mathbf{D}(\mathbf{B}), \mathbf{nrows}(\mathbf{B}), \mathbf{ncols}(\mathbf{B}), \mathbf{L}(\mathbf{B}) = \{(i, j, B_{ij})\} \rangle$

3971 The argument matrices, the “sum” operator (`op`), and the accumulation operator (if provided) are  
3972 tested for domain compatibility as follows:

- 3973 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{Mask})$   
3974 must be from one of the pre-defined types of Table 3.2.
- 3975 2.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3976 3.  $\mathbf{D}(\mathbf{B})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3977 4.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3978 5.  $\mathbf{D}(\mathbf{A})$  and  $\mathbf{D}(\mathbf{B})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3979 6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
3980 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
3981 the accumulation operator.

3982 Two domains are compatible with each other if values from one domain can be cast to values in  
3983 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
3984 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
3985 any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch  
3986 error listed above is returned.

3987 From the argument matrices, the internal matrices and mask used in the computation are formed  
3988 ( $\leftarrow$  denotes copy):

- 3989 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3990 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
- 3991 (a) If `Mask` = `GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
3992  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 3993 (b) If `Mask`  $\neq$  `GrB_NULL`,

3994 i. If desc[GrB\_MASK].GrB\_STRUCTURE is set, then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
3995  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,  
3996 ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
3997  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .  
3998 (c) If desc[GrB\_MASK].GrB\_COMP is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$ .  
3999 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .  
4000 4. Matrix  $\widetilde{\mathbf{B}} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

4001 The internal matrices and masks are checked for dimension compatibility. The following conditions  
4002 must hold:

- 4003 1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}}) = \mathbf{nrows}(\widetilde{\mathbf{A}}) = \mathbf{nrows}(\widetilde{\mathbf{B}})$ .  
4004 2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}}) = \mathbf{ncols}(\widetilde{\mathbf{A}}) = \mathbf{ncols}(\widetilde{\mathbf{B}})$ .

4005 If any compatibility rule above is violated, execution of GrB\_eWiseAdd ends and the dimension  
4006 mismatch error listed above is returned.

4007 From this point forward, in GrB\_NONBLOCKING mode, the method can optionally exit with  
4008 GrB\_SUCCESS return code and defer any computation and/or execution error codes.

4009 We are now ready to carry out the element-wise “sum” and any additional associated operations.  
4010 We describe this in terms of two intermediate matrices:

- 4011 •  $\widetilde{\mathbf{T}}$ : The matrix holding the element-wise sum of  $\widetilde{\mathbf{A}}$  and  $\widetilde{\mathbf{B}}$ .  
4012 •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4013 The intermediate matrix  $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}) \cup \mathbf{ind}(\widetilde{\mathbf{B}}) \neq \emptyset\} \rangle$   
4014 is created. The value of each of its elements is computed by

$$\begin{aligned}
4015 \quad T_{ij} &= (\widetilde{\mathbf{A}}(i, j) \oplus \widetilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}}) \\
4016 \quad T_{ij} &= \widetilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{A}}) - (\mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}}))) \\
4017 \quad T_{ij} &= \widetilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{B}}) - (\mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}})))
\end{aligned}$$

4020 where the difference operator in the previous expressions refers to set difference.

4021 The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4022 • If accum = GrB\_NULL, then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .  
4023 • If accum is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$4024 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{C}}$  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.

### 4.3.6 extract: Selecting sub-graphs

Extract a subset of a matrix or vector.

#### 4.3.6.1 extract: Standard vector variant

Extract a sub-vector from a larger vector as specified by a set of indices. The result is a vector whose size is equal to the number of indices.

### C Syntax

```
GrB_Info GrB_extract(GrB_Vector w,
                    const GrB_Vector mask,
```

```

4056         const GrB_BinaryOp      accum,
4057         const GrB_Vector        u,
4058         const GrB_Index         *indices,
4059         GrB_Index               nindices,
4060         const GrB_Descriptor     desc);

```

## 4061 Parameters

4062     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4063             that may be accumulated with the result of the extract operation. On output, this  
4064             vector holds the results of the operation.

4065     **mask** (IN) An optional “write” mask that controls which results from this operation are  
4066             stored into the output vector **w**. The mask dimensions must match those of the  
4067             vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4068             of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4069             in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4070             dimensions of **w**), **GrB\_NULL** should be specified.

4071     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4072             entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4073             specified.

4074     **u** (IN) The GraphBLAS vector from which the subset is extracted.

4075     **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of  
4076             elements from **u** that are extracted. If all elements of **u** are to be extracted in order  
4077             from 0 to **nindices** − 1, then **GrB\_ALL** should be specified. Regardless of execution  
4078             mode and return value, this array may be manipulated by the caller after this  
4079             operation returns without affecting any deferred computations for this operation.

4080     **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4081     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4082             should be specified. Non-default field/value pairs are listed as follows:  
4083

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> .



## 4085 Return Values

4086                   GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 4087                   blocking mode, this indicates that the compatibility tests on  
 4088                   dimensions and domains for the input arguments passed suc-  
 4089                   cessfully. Either way, output vector **w** is ready to be used in the  
 4090                   next method of the sequence.

4091                   GrB\_PANIC Unknown internal error.

4092                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 4093                   opaque GraphBLAS objects (input or output) is in an invalid  
 4094                   state caused by a previous execution error. Call **GrB\_error()** to  
 4095                   access any error messages generated by the implementation.

4096                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

4097                   GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
 4098                   by a call to **new** (or **dup** for vector parameters).

4099                   GrB\_INDEX\_OUT\_OF\_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-  
 4100                   blocking mode, this error can be deferred.

4101                   GrB\_DIMENSION\_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices**  $\neq$  **size(w)**.

4102                   GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with each  
 4103                   other or the corresponding domains of the accumulation oper-  
 4104                   ator, or the mask's domain is not compatible with **bool** (in the  
 4105                   case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

4106                   GrB\_NULL\_POINTER Argument **row\_indices** is a NULL pointer.

## 4107 Description

4108 This variant of **GrB\_extract** computes the result of extracting a subset of locations from a Graph-  
 4109 BLAS vector in a specific order: **w** = **u(indices)**; or, if an optional binary accumulation operator  
 4110 ( $\odot$ ) is provided, **w** = **w**  $\odot$  **u(indices)**. More explicitly:

$$4111 \quad \begin{aligned} \mathbf{w}(i) &= \mathbf{u}(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ \mathbf{w}(i) &= \mathbf{w}(i) \odot \mathbf{u}(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4112 Logically, this operation occurs in three steps:

4113       **Setup** The internal vectors and mask used in the computation are formed and their domains  
 4114       and dimensions are tested for compatibility.

4115       **Compute** The indicated computations are carried out.

4116       **Output** The result is written into the output vector, possibly under control of a mask.

4117 Up to three argument vectors are used in this GrB\_extract operation:

- 4118 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4119 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4120 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4121 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4122 bility as follows:

- 4123 1. If  $\mathbf{mask}$  is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\mathbf{mask})$   
4124 must be from one of the pre-defined types of Table 3.2.
- 4125 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4126 3. If  $\mathbf{accum}$  is not GrB\_NULL, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
4127 of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
4128 mulation operator.

4129 Two domains are compatible with each other if values from one domain can be cast to values in  
4130 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4131 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4132 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
4133 error listed above is returned.

4134 From the arguments, the internal vectors,  $\mathbf{mask}$ , and index array used in the computation are  
4135 formed ( $\leftarrow$  denotes copy):

- 4136 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 4137 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument  $\mathbf{mask}$  as follows:
  - 4138 (a) If  $\mathbf{mask} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 4139 (b) If  $\mathbf{mask} \neq \text{GrB\_NULL}$ ,
    - 4140 i. If desc[GrB\_MASK].GrB\_STRUCTURE is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 4141 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\text{bool})\mathbf{mask}(i) = \text{true}\} \rangle$ .
  - 4142 (c) If desc[GrB\_MASK].GrB\_COMP is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 4143 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 4144 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument indices as follows:
  - 4145 (a) If  $\mathbf{indices} = \text{GrB\_ALL}$ , then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$ .
  - 4146 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$ .

4147 The internal vectors and mask are checked for dimension compatibility. The following conditions  
4148 must hold:

4149 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4150 2.  $\text{nindices} = \text{size}(\tilde{\mathbf{w}})$ .

4151 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4152 match error listed above is returned.

4153 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4154 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4155 We are now ready to carry out the extract and any additional associated operations. We describe  
4156 this in terms of two intermediate vectors:

- 4157 •  $\tilde{\mathbf{t}}$ : The vector holding the extraction from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{w}}$ .
- 4158 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4159 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4160 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(i, \tilde{\mathbf{u}}[\tilde{\mathbf{I}}[i])) \mid \forall i, 0 \leq i < \text{nindices} : \tilde{\mathbf{I}}[i] \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4161 At this point, if any value in  $\tilde{\mathbf{I}}$  is not in the valid range of indices for vector  $\tilde{\mathbf{u}}$ , the execution of  
4162 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`  
4163 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4164 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4165 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 4166 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 4167 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4168 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4169 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4170 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 4171 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 4172 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4173 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4174 \end{aligned}$$

4175 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4177 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
4178 using what is called a *standard vector mask and replace*. This is carried out under control of the  
4179 mask which acts as a “write mask”.

- 4180 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
4181 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4182 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- 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.

#### 4.3.6.2 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.

4218           A (IN) The GraphBLAS matrix from which the subset is extracted.

4219   row\_indices (IN) Pointer to the ordered set (array) of indices corresponding to the rows of A  
4220           from which elements are extracted. If elements in all rows of A are to be extracted  
4221           in order, GrB\_ALL should be specified. Regardless of execution mode and return  
4222           value, this array may be manipulated by the caller after this operation returns  
4223           without affecting any deferred computations for this operation.

4224   nrows (IN) The number of values in the row\_indices array. Must be equal to **nrows(C)**.

4225   col\_indices (IN) Pointer to the ordered set (array) of indices corresponding to the columns  
4226           of A from which elements are extracted. If elements in all columns of A are to  
4227           be extracted in order, then GrB\_ALL should be specified. Regardless of execution  
4228           mode and return value, this array may be manipulated by the caller after this  
4229           operation returns without affecting any deferred computations for this operation.

4230   ncols (IN) The number of values in the col\_indices array. Must be equal to **ncols(C)**.

4231   desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
4232           should be specified. Non-default field/value pairs are listed as follows:

4233

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.

4234

## 4235 Return Values

4236           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
4237           blocking mode, this indicates that the compatibility tests on  
4238           dimensions and domains for the input arguments passed suc-  
4239           cessfully. Either way, output matrix C is ready to be used in the  
4240           next method of the sequence.

4241           GrB\_PANIC Unknown internal error.

4242           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
4243           opaque GraphBLAS objects (input or output) is in an invalid  
4244           state caused by a previous execution error. Call GrB\_error() to  
4245           access any error messages generated by the implementation.

4246           GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

4247 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
4248 by a call to `new` (or `Matrix_dup` for matrix parameters).

4249 GrB\_INDEX\_OUT\_OF\_BOUNDS A value in `row_indices` is greater than or equal to `nrows(A)`, or  
4250 a value in `col_indices` is greater than or equal to `ncols(A)`. In  
4251 non-blocking mode, this error can be deferred.

4252 GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, `nrows`  $\neq$  `nrows(C)`, or  
4253 `ncols`  $\neq$  `ncols(C)`.

4254 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with each  
4255 other or the corresponding domains of the accumulation oper-  
4256 ator, or the mask's domain is not compatible with `bool` (in the  
4257 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4258 GrB\_NULL\_POINTER Either argument `row_indices` is a NULL pointer, argument `col_indices`  
4259 is a NULL pointer, or both.

## 4260 Description

4261 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified  
4262 rows and columns of a GraphBLAS matrix in a specific order:  $C = A(\text{row\_indices}, \text{col\_indices})$ ; or,  
4263 if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A(\text{row\_indices}, \text{col\_indices})$ .  
4264 More explicitly (not accounting for an optional transpose of A):

$$4265 \quad \begin{aligned} C(i, j) &= A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ C(i, j) &= C(i, j) \odot A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4266 Logically, this operation occurs in three steps:

4267 **Setup** The internal matrices and mask used in the computation are formed and their domains  
4268 and dimensions are tested for compatibility.

4269 **Compute** The indicated computations are carried out.

4270 **Output** The result is written into the output matrix, possibly under control of a mask.

4271 Up to three argument matrices are used in the `GrB_extract` operation:

- 4272 1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4273 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)
- 4274 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4275 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
4276 ibility as follows:

- 4277 1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 4278 must be from one of the pre-defined types of Table 3.2.
- 4279 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}(\mathbf{A})$ .
- 4280 3. If accum is not GrB\_NULL, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4281 of the accumulation operator and  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4282 mulation operator.

4283 Two domains are compatible with each other if values from one domain can be cast to values in  
 4284 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4285 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4286 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
 4287 error listed above is returned.

4288 From the arguments, the internal matrices, mask, and index arrays used in the computation are  
 4289 formed ( $\leftarrow$  denotes copy):

- 4290 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 4291 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument Mask as follows:
- 4292 (a) If Mask = GrB\_NULL, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 4293  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 4294 (b) If Mask  $\neq$  GrB\_NULL,
- 4295 i. If desc[GrB\_MASK].GrB\_STRUCTURE is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
 4296  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
- 4297 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
 4298  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
- 4299 (c) If desc[GrB\_MASK].GrB\_COMP is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4300 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4301 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument row\_indices as follows:
- 4302 (a) If row\_indices = GrB\_ALL, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4303 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4304 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument col\_indices as follows:
- 4305 (a) If col\_indices = GrB\_ALL, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
- 4306 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

4307 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 4308 must hold:

- 4309 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .

4310 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .

4311 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}$ .

4312 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}$ .

4313 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4314 match error listed above is returned.

4315 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4316 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4317 We are now ready to carry out the extract and any additional associated operations. We describe  
4318 this in terms of two intermediate matrices:

- 4319 •  $\tilde{\mathbf{T}}$ : The matrix holding the extraction from  $\tilde{\mathbf{A}}$ .
- 4320 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4321 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$4322 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(i, j, \tilde{\mathbf{A}}[\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]]) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4323 At this point, if any value in the  $\tilde{\mathbf{I}}$  array is not in the range  $[0, \mathbf{nrows}(\tilde{\mathbf{A}}))$  or any value in the  $\tilde{\mathbf{J}}$   
4324 array is not in the range  $[0, \mathbf{ncols}(\tilde{\mathbf{A}}))$ , the execution of `GrB_extract` ends and the index out-of-  
4325 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
4326 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
4327 this point forward in the sequence.

4328 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4329 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 4330 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4331 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4332 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
4333 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4334 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 4335 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4336 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 4337 \quad Z_{ij} = 0, \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{C}}) \cap \mathbf{ind}(\tilde{\mathbf{T}}))), \\ 4338 \quad Z_{ij} = 0, \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{C}}) \cap \mathbf{ind}(\tilde{\mathbf{T}}))),$$

4339 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.



4340 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 4341 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 4342 mask which acts as a “write mask”.

- 4343 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 4344 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$4345 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4346 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 4347 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 4348 mask are unchanged:

$$4349 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4350 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 4351 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 4352 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 4353 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 4354 sequence.

#### 4355 4.3.6.3 extract: Column (and row) variant

4356 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the  
 4357 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as  
 4358 well.

#### 4359 C Syntax

```
4360      GrB_Info GrB_extract(GrB_Vector      w,
4361                          const GrB_Vector mask,
4362                          const GrB_BinaryOp accum,
4363                          const GrB_Matrix A,
4364                          const GrB_Index *row_indices,
4365                          GrB_Index      nrows,
4366                          GrB_Index      col_index,
4367                          const GrB_Descriptor desc);
```

#### 4368 Parameters

4369 w (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
 4370 that may be accumulated with the result of the extract operation. On output, this  
 4371 vector holds the results of the operation.

4372        **mask** (IN) An optional “write” mask that controls which results from this operation are  
 4373        stored into the output vector **w**. The mask dimensions must match those of the  
 4374        vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
 4375        of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
 4376        in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
 4377        dimensions of **w**), **GrB\_NULL** should be specified.

4378        **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
 4379        entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 4380        specified.

4381        **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4382        **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations  
 4383        within the specified column of **A** from which elements are extracted. If elements in  
 4384        all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless  
 4385        of execution mode and return value, this array may be manipulated by the caller  
 4386        after this operation returns without affecting any deferred computations for this  
 4387        operation.

4388        **nrows** (IN) The number of indices in the **row\_indices** array. Must be equal to **size(w)**.

4389        **col\_index** (IN) The index of the column of **A** from which to extract values. It must be in the  
 4390        range  $[0, \mathbf{ncols}(\mathbf{A})]$ .

4391        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 4392        should be specified. Non-default field/value pairs are listed as follows:  
 4393

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> .
<b>A</b>	<b>GrB_INP0</b>	<b>GrB_TRAN</b>	Use transpose of <b>A</b> for the operation.

## 4395 Return Values

4396        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4397        blocking mode, this indicates that the compatibility tests on  
 4398        dimensions and domains for the input arguments passed suc-  
 4399        cessfully. Either way, output vector **w** is ready to be used in the  
 4400        next method of the sequence.

4401        **GrB\_PANIC** Unknown internal error.



4432 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

4433 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain  
4434 compatibility as follows:

- 4435 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4436 must be from one of the pre-defined types of Table 3.2.
- 4437 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{A})$ .
- 4438 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4439 of the accumulation operator and  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4440 mulation operator.

4441 Two domains are compatible with each other if values from one domain can be cast to values in  
4442 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4443 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4444 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4445 error listed above is returned.

4446 From the arguments, the internal vector, matrix, mask, and index array used in the computation  
4447 are formed ( $\leftarrow$  denotes copy):

- 4448 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 4449 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 4450 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 4451 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 4452 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
    - 4453 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 4454 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 4455 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4456 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
  - 4457 (a) If `indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - 4458 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .

4459 The internal vector, mask, and index array are checked for dimension compatibility. The following  
4460 conditions must hold:

- 4461 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 4462 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}$ .

4463 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
 4464 match error listed above is returned.

4465 The `col_index` parameter is checked for a valid value. The following condition must hold:

4466 1.  $0 \leq \text{col\_index} < \text{ncols}(\mathbf{A})$

4467 If the rule above is violated, execution of `GrB_extract` ends and the invalid index error listed above  
 4468 is returned.

4469 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4470 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4471 We are now ready to carry out the extract and any additional associated operations. We describe  
 4472 this in terms of two intermediate vectors:

- 4473 •  $\tilde{\mathbf{t}}$ : The vector holding the extraction from a column of  $\tilde{\mathbf{A}}$ .
- 4474 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4475 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4476 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \tilde{\mathbf{A}}[\tilde{\mathbf{I}}[i], \text{col\_index}]) \mid \forall i, 0 \leq i < \text{nrows} : (\tilde{\mathbf{I}}[i], \text{col\_index}) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4477 At this point, if any value in  $\tilde{\mathbf{I}}$  is not in the range  $[0, \text{nrows}(\tilde{\mathbf{A}}))$ , the execution of `GrB_extract`  
 4478 ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING` mode,  
 4479 the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result  
 4480 vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4481 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 4482 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 4483 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4484 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4485 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 4486 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 4487 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 4488 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4489 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4490 \end{aligned}$$

4491 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4492 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 4493 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 4494 mask which acts as a “write mask”.  
 4495

- 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.

### 4.3.7 assign: Modifying sub-graphs

Assign the contents of a subset of a matrix or vector.

#### 4.3.7.1 assign: Standard vector variant

Assign values from one GraphBLAS vector to a subset of a vector as specified by a set of indices. The size of the input vector is the same size as the index array provided.

## C Syntax

```
GrB_Info GrB_assign(GrB_Vector      w,
                    const GrB_Vector mask,
                    const GrB_BinaryOp accum,
                    const GrB_Vector u,
                    const GrB_Index *indices,
                    GrB_Index      nindices,
                    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 assign 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

4527 vector **w** If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
 4528 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
 4529 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
 4530 dimensions of **w**), **GrB\_NULL** should be specified.

4531 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
 4532 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 4533 specified.

4534 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4535 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
 4536 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0  
 4537 to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
 4538 and return value, this array may be manipulated by the caller after this operation  
 4539 returns without affecting any deferred computations for this operation. If this  
 4540 array contains duplicate values, it implies in assignment of more than one value to  
 4541 the same location which leads to undefined results.

4542 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4543 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 4544 should be specified. Non-default field/value pairs are listed as follows:

4545

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> .

4546

## 4547 Return Values

4548 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4549 blocking mode, this indicates that the compatibility tests on  
 4550 dimensions and domains for the input arguments passed suc-  
 4551 cessfully. Either way, output vector **w** is ready to be used in the  
 4552 next method of the sequence.

4553 **GrB\_PANIC** Unknown internal error.

4554 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 4555 opaque GraphBLAS objects (input or output) is in an invalid  
 4556 state caused by a previous execution error. Call **GrB\_error()** to  
 4557 access any error messages generated by the implementation.

4558        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

4559        **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
4560                by a call to **new** (or **dup** for vector parameters).

4561        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indices** is greater than or equal to **size(w)**. In non-  
4562                blocking mode, this can be reported as an execution error.

4563        **GrB\_DIMENSION\_MISMATCH** **mask** and **w** dimensions are incompatible, or **nindices**  $\neq$  **size(u)**.

4564        **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with each  
4565                other or the corresponding domains of the accumulation oper-  
4566                ator, or the mask's domain is not compatible with **bool** (in the  
4567                case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

4568        **GrB\_NULL\_POINTER** Argument **indices** is a **NULL** pointer.

## 4569 Description

4570 This variant of **GrB\_assign** computes the result of assigning elements from a source GraphBLAS  
4571 vector to a destination GraphBLAS vector in a specific order:  $w(\text{indices}) = u$ ; or, if an optional  
4572 binary accumulation operator ( $\odot$ ) is provided,  $w(\text{indices}) = w(\text{indices}) \odot u$ . More explicitly:

$$\begin{aligned}
&w(\text{indices}[i]) = u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
&w(\text{indices}[i]) = w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
\end{aligned}$$

4574 Logically, this operation occurs in three steps:

4575        **Setup** The internal vectors and mask used in the computation are formed and their domains  
4576                and dimensions are tested for compatibility.

4577        **Compute** The indicated computations are carried out.

4578        **Output** The result is written into the output vector, possibly under control of a mask.

4579 Up to three argument vectors are used in the **GrB\_assign** operation:

- 4580 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4581 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4582 3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4583 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4584 bility as follows:

- 4585 1. If **mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then **D(mask)**  
4586        must be from one of the pre-defined types of Table 3.2.



4587 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .

4588 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4589 of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4590 mulation operator.

4591 Two domains are compatible with each other if values from one domain can be cast to values in  
 4592 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4593 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4594 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 4595 error listed above is returned.

4596 From the arguments, the internal vectors, mask and index array used in the computation are formed  
 4597 ( $\leftarrow$  denotes copy):

4598 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .

4599 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

4600 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .

4601 (b) If `mask`  $\neq$  `GrB_NULL`,

4602 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,

4603 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

4604 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

4605 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4606 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument `indices` as follows:

4607 (a) If `indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .

4608 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4609 The internal vector and mask are checked for dimension compatibility. The following conditions  
 4610 must hold:

4611 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4612 2.  $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$ .

4613 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 4614 match error listed above is returned.

4615 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4616 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4617 We are now ready to carry out the assign and any additional associated operations. We describe  
 4618 this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{w}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid 0 \leq i < \text{nindices} : i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{w}}$ , computation ends and the method returns the index-out-of-bounds error listed above. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure 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 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

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  $\tilde{\mathbf{t}}$ .

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

where the difference operator refers to set difference.

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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  $\tilde{\mathbf{t}}$ .

$$z_i = \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})),$$

$$z_i = \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))),$$

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  $w$  on input to this operation are deleted and the content of the new output vector,  $w$ , is defined as,

$$\mathbf{L}(w) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{z}) \cap \mathbf{ind}(\tilde{m}))\}.$$

- 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:

$$\mathbf{L}(w) = \{(i, w_i) : i \in (\mathbf{ind}(w) \cap \mathbf{ind}(\neg\tilde{m}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{z}) \cap \mathbf{ind}(\tilde{m}))\}.$$

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.

#### 4.3.7.2 assign: Standard matrix variant

Assign values from one GraphBLAS matrix to a subset of a matrix as specified by a set of indices. The dimensions of the input matrix are the same size as the row and column index arrays provided.

### C Syntax

```
GrB_Info GrB_assign(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 assign 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.

4687        **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4688        entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4689        specified.

4690        **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4691        **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
4692        that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** - 1,  
4693        then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
4694        this array may be manipulated by the caller after this operation returns without  
4695        affecting any deferred computations for this operation. If this array contains du-  
4696        plicate values, it implies assignment of more than one value to the same location  
4697        which leads to undefined results.

4698        **nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(A)**  
4699        if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4700        **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns  
4701        of **C** that are assigned. If all columns of **C** are to be assigned in order from 0  
4702        to **ncols** - 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
4703        and return value, this array may be manipulated by the caller after this operation  
4704        returns without affecting any deferred computations for this operation. If this  
4705        array contains duplicate values, it implies assignment of more than one value to  
4706        the same location which leads to undefined results.

4707        **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **ncols(A)** if **A** is  
4708        not transposed, or equal to **nrows(A)** if **A** is transposed.

4709        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4710        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.

## 4713 Return Values

4714        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
4715        blocking mode, this indicates that the compatibility tests on  
4716        dimensions and domains for the input arguments passed suc-  
4717        cessfully. Either way, output matrix **C** is ready to be used in the  
4718        next method of the sequence.

4719	<b>GrB_PANIC</b>	Unknown internal error.
4720	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the
4721		opaque GraphBLAS objects (input or output) is in an invalid
4722		state caused by a previous execution error. Call <code>GrB_error()</code> to
4723		access any error messages generated by the implementation.
4724	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for the operation.
4725	<b>GrB_UNINITIALIZED_OBJECT</b>	One or more of the GraphBLAS objects has not been initialized
4726		by a call to <code>new</code> (or <code>Matrix_dup</code> for matrix parameters).
4727	<b>GrB_INDEX_OUT_OF_BOUNDS</b>	A value in <code>row_indices</code> is greater than or equal to <code>nrows(C)</code> , or
4728		a value in <code>col_indices</code> is greater than or equal to <code>ncols(C)</code> . In
4729		non-blocking mode, this can be reported as an execution error.
4730	<b>GrB_DIMENSION_MISMATCH</b>	Mask and C dimensions are incompatible, <code>nrows</code> $\neq$ <code>nrows(A)</code> ,
4731		or <code>ncols</code> $\neq$ <code>ncols(A)</code> .
4732	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the various matrices are incompatible with each
4733		other or the corresponding domains of the accumulation oper-
4734		ator, or the mask's domain is not compatible with <code>bool</code> (in the
4735		case where <code>desc[GrB_MASK].GrB_STRUCTURE</code> is not set).
4736	<b>GrB_NULL_POINTER</b>	Either argument <code>row_indices</code> is a NULL pointer, argument <code>col_indices</code>
4737		is a NULL pointer, or both.

## 4738 Description

4739 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows  
4740 and columns in `C` in a specified order:  $C(\text{row\_indices}, \text{col\_indices}) = A$ ; or, if an optional binary  
4741 accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot$   
4742 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned}
&C(\text{row\_indices}[i], \text{col\_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
&C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot A(i, j), \\
&\quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
\end{aligned}$$

4744 Logically, this operation occurs in three steps:

4745     **Setup** The internal matrices and mask used in the computation are formed and their domains  
4746     and dimensions are tested for compatibility.

4747     **Compute** The indicated computations are carried out.

4748     **Output** The result is written into the output matrix, possibly under control of a mask.

4749 Up to three argument matrices are used in the `GrB_assign` operation:

- 4750 1.  $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 4751 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)
- 4752 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

4753 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
 4754 ibility as follows:

- 4755 1. If  $\mathbf{Mask}$  is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{Mask})$   
 4756 must be from one of the pre-defined types of Table 3.2.
- 4757 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}(\mathbf{A})$ .
- 4758 3. If  $\mathbf{accum}$  is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 4759 of the accumulation operator and  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
 4760 mulation operator.

4761 Two domains are compatible with each other if values from one domain can be cast to values in  
 4762 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4763 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4764 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 4765 error listed above is returned.

4766 From the arguments, the internal matrices,  $\mathbf{mask}$ , and index arrays used in the computation are  
 4767 formed ( $\leftarrow$  denotes copy):

- 4768 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 4769 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument  $\mathbf{Mask}$  as follows:
  - 4770 (a) If  $\mathbf{Mask} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 4771  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 4772 (b) If  $\mathbf{Mask} \neq \mathbf{GrB\_NULL}$ ,
    - 4773 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 4774  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 4775 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 4776  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 4777 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4778 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4779 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
  - 4780 (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - 4781 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4782 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:

- 4783 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .  
 4784 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

4785 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 4786 must hold:

- 4787 1.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$ .  
 4788 2.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$ .  
 4789 3.  $\text{nrows}(\tilde{\mathbf{A}}) = \text{nrows}$ .  
 4790 4.  $\text{ncols}(\tilde{\mathbf{A}}) = \text{ncols}$ .

4791 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 4792 match error listed above is returned.

4793 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4794 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4795 We are now ready to carry out the assign and any additional associated operations. We describe  
 4796 this in terms of two intermediate vectors:

- 4797 •  $\tilde{\mathbf{T}}$ : The matrix holding the contents from  $\tilde{\mathbf{A}}$  in their destination locations relative to  $\tilde{\mathbf{C}}$ .
- 4798 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4799 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$4800 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (i, j) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4801 At this point, if any value in the  $\tilde{\mathbf{I}}$  array is not in the range  $[0, \text{nrows}(\tilde{\mathbf{C}}))$  or any value in the  
 4802  $\tilde{\mathbf{J}}$  array is not in the range  $[0, \text{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-  
 4803 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
 4804 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
 4805 this point forward in the sequence.

4806 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 4807 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$4808 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \\ 4809 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\text{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \text{ind}(\tilde{\mathbf{C}}))) \cup \text{ind}(\tilde{\mathbf{T}}))\} \rangle.$$

4810 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 4811 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  
 4812 assigned ( $\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \text{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\text{ind}(\tilde{\mathbf{T}})$ ).

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  $\tilde{\mathbf{T}}$ .

$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{T}}[k], \tilde{\mathbf{T}}[l]), \forall k, l\} \cap \text{ind}(\tilde{\mathbf{C}}))),$$

$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \text{ind}(\tilde{\mathbf{T}}),$$

where the difference operator refers to set difference.

- If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \text{ind}(\tilde{\mathbf{C}}) \cup \text{ind}(\tilde{\mathbf{T}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}})),$$

$$Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{C}}) - (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}}))),$$

$$Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\text{ind}(\tilde{\mathbf{T}}) - (\text{ind}(\tilde{\mathbf{T}}) \cap \text{ind}(\tilde{\mathbf{C}}))),$$

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,

$$\mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\text{ind}(\tilde{\mathbf{Z}}) \cap \text{ind}(\tilde{\mathbf{M}}))\}.$$

- If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the mask are unchanged:

$$\mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\text{ind}(\mathbf{C}) \cap \text{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\text{ind}(\tilde{\mathbf{Z}}) \cap \text{ind}(\tilde{\mathbf{M}}))\}.$$

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.

#### 4.3.7.3 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.



## 4848 C Syntax

```
4849         GrB_Info GrB_assign(GrB_Matrix      C,  
4850                             const GrB_Vector mask,  
4851                             const GrB_BinaryOp accum,  
4852                             const GrB_Vector u,  
4853                             const GrB_Index *row_indices,  
4854                             GrB_Index      nrows,  
4855                             GrB_Index      col_index,  
4856                             const GrB_Descriptor desc);
```

## 4857 Parameters

4858 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
4859 that may be accumulated with the result of the assign operation. On output, this  
4860 matrix holds the results of the operation.

4861 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4862 stored into the specified column of the output matrix **C**. The mask dimensions  
4863 must match those of a single column of the matrix **C**. If the **GrB\_STRUCTURE**  
4864 descriptor is *not* set for the mask, the domain of the **Mask** matrix must be of type  
4865 **bool** or any of the predefined “built-in” types in Table 3.2. If the default mask  
4866 is desired (i.e., a mask that is all true with the dimensions of a column of **C**),  
4867 **GrB\_NULL** should be specified.

4868 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4869 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4870 specified.

4871 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column  
4872 of **C**.

4873 **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4874 the specified column of **C** that are to be assigned. If all elements of the column  
4875 in **C** are to be assigned in order from index 0 to **nrows** – 1, then **GrB\_ALL** should  
4876 be specified. Regardless of execution mode and return value, this array may be  
4877 manipulated by the caller after this operation returns without affecting any de-  
4878 ferred computations for this operation. If this array contains duplicate values, it  
4879 implies in assignment of more than one value to the same location which leads to  
4880 undefined results.

4881 **nrows** (IN) The number of values in **row\_indices** array. Must be equal to **size(u)**.

4882 **col\_index** (IN) The index of the column in **C** to assign. Must be in the range [0, **ncols(C)**).

4883 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4884 should be specified. Non-default field/value pairs are listed as follows:  
4885

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> .

## 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>col_index</b> is outside the allowable range (i.e., greater than <b>ncols(C)</b> ).
GrB_INDEX_OUT_OF_BOUNDS	A value in <b>row_indices</b> is greater than or equal to <b>nrows(C)</b> . In non-blocking mode, this can be reported as an execution error.
GrB_DIMENSION_MISMATCH	<b>mask</b> size and number of rows in C are not the same, or <b>nrows</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>row_indices</b> is a NULL pointer.

## 4911 Description

4912 This variant of GrB\_assign computes the result of assigning a subset of locations in a column of a  
4913 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

4914  $C(:, \text{col\_index}) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(:, \text{col\_index}) =$   
4915  $C(:, \text{col\_index}) \odot u$ . Taking order of row\_indices into account, it is more explicitly written as:

$$4916 \quad C(\text{row\_indices}[i], \text{col\_index}) = u(i), \forall i : 0 \leq i < \text{nrows}, \text{ or}$$

$$C(\text{row\_indices}[i], \text{col\_index}) = C(\text{row\_indices}[i], \text{col\_index}) \odot u(i), \forall i : 0 \leq i < \text{nrows}.$$

4917 Logically, this operation occurs in three steps:

4918     **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
4919 domains and dimensions are tested for compatibility.

4920     **Compute** The indicated computations are carried out.

4921     **Output** The result is written into the output matrix, possibly under control of a mask.

4922 Up to three argument vectors and matrices are used in this GrB\_assign operation:

- 4923 1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4924 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4925 3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4926 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
4927 compatibility as follows:

- 4928 1. If mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{mask})$   
4929 must be from one of the pre-defined types of Table 3.2.
- 4930 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(u)$ .
- 4931 3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4932 of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4933 mulation operator.

4934 Two domains are compatible with each other if values from one domain can be cast to values in  
4935 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4936 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4937 any compatibility rule above is violated, execution of GrB\_assign ends and the domain mismatch  
4938 error listed above is returned.

4939 The col\_index parameter is checked for a valid value. The following condition must hold:

- 4940 1.  $0 \leq \text{col\_index} < \text{ncols}(C)$

4941 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
 4942 is returned.

4943 From the arguments, the internal vectors, mask, and index array used in the computation are  
 4944 formed ( $\leftarrow$  denotes copy):

4945 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a column of  $\mathbf{C}$  as follows:

$$4946 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4947 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

4948 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$ .

4949 (b) If `mask  $\neq$  GrB_NULL`,

4950 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

4951 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

4952 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

4953 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4954 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:

4955 (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .

4956 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .

4957 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
 4958 conditions must hold:

4959 1.  $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

4960 2.  $\mathbf{nrows} = \mathbf{size}(\tilde{\mathbf{u}})$ .

4961 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 4962 match error listed above is returned.

4963 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4964 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4965 We are now ready to carry out the assign and any additional associated operations. We describe  
 4966 this in terms of two intermediate vectors:

4967 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .

4968 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4969 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4970 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \mathbf{nrows} : i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

At this point, if any value of  $\tilde{\mathbf{I}}[i]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure 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 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

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  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}), \end{aligned}$$

where the difference operator refers to set difference.

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

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  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} z_i &= \tilde{\mathbf{c}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})), \\ z_i &= \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))), \\ z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))), \end{aligned}$$

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:

$$\begin{aligned} \mathbf{L}(\mathbf{C}) &= \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \\ &\quad \{(i, \text{col\_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ &\quad \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

5009 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
5010 of vector w is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
5011 exits with return value GrB\_SUCCESS and the new content of vector w is as defined above but may  
5012 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 5013 4.3.7.4 assign: Row variant

5014 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the  
5015 output cannot be transposed, a different variant of assign is provided to assign to a column of a  
5016 matrix.

#### 5017 C Syntax

```
5018         GrB_Info GrB_assign(GrB_Matrix      C,
5019                             const GrB_Vector mask,
5020                             const GrB_BinaryOp accum,
5021                             const GrB_Vector u,
5022                             GrB_Index      row_index,
5023                             const GrB_Index *col_indices,
5024                             GrB_Index      ncols,
5025                             const GrB_Descriptor desc);
```

#### 5026 Parameters

5027 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values  
5028 that may be accumulated with the result of the assign operation. On output, this  
5029 matrix holds the results of the operation.

5030 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5031 stored into the specified row of the output matrix C. The mask dimensions must  
5032 match those of a single row of the matrix C. If the GrB\_STRUCTURE descriptor  
5033 is *not* set for the mask, the domain of the Mask matrix must be of type **bool** or  
5034 any of the predefined “built-in” types in Table 3.2. If the default mask is desired  
5035 (i.e., a mask that is all true with the dimensions of a row of C), GrB\_NULL should  
5036 be specified.

5037 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
5038 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
5039 specified.

5040 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of  
5041 C.

5042 **row\_index** (IN) The index of the row in C to assign. Must be in the range [0, **nrows**(C)).

5043 **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5044 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to  
5045 be assigned in order from index 0 to **ncols** - 1, then **GrB\_ALL** should be specified.  
5046 Regardless of execution mode and return value, this array may be manipulated by  
5047 the caller after this operation returns without affecting any deferred computations  
5048 for this operation. If this array contains duplicate values, it implies in assignment  
5049 of more than one value to the same location which leads to undefined results.

5050 **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **size(u)**.

5051 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5052 should be specified. Non-default field/value pairs are listed as follows:  
5053

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output row in <b>C</b> 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> .

## 5055 Return Values

5056 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5057 blocking mode, this indicates that the compatibility tests on  
5058 dimensions and domains for the input arguments passed suc-  
5059 cessfully. Either way, output matrix **C** is ready to be used in the  
5060 next method of the sequence.

5061 **GrB\_PANIC** Unknown internal error.

5062 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
5063 opaque GraphBLAS objects (input or output) is in an invalid  
5064 state caused by a previous execution error. Call **GrB\_error()** to  
5065 access any error messages generated by the implementation.

5066 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5067 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
5068 by a call to **new** (or **dup** for vector or matrix parameters).

5069 **GrB\_INVALID\_INDEX** **row\_index** is outside the allowable range (i.e., greater than **nrows(C)**).

5070 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **col\_indices** is greater than or equal to **ncols(C)**. In  
5071 non-blocking mode, this can be reported as an execution error.

5072 **GrB\_DIMENSION\_MISMATCH** **mask** size and number of columns in **C** are not the same, or  
5073 **ncols**  $\neq$  **size(u)**.





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.

The `row_index` parameter is checked for a valid value. The following condition must hold:

1.  $0 \leq \text{row\_index} < \text{nrows}(\mathbf{C})$

If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above is returned.

From the arguments, the internal vectors, mask, and index array used in the computation are formed ( $\leftarrow$  denotes copy):

1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a row of  $\mathbf{C}$  as follows:

$$\tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \text{ncols}(\mathbf{C}), \{(j, C_{ij}) \mid \forall j : 0 \leq j < \text{ncols}(\mathbf{C}), i = \text{row\_index}, (i, j) \in \text{ind}(\mathbf{C})\} \rangle$$

2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

(a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{ncols}(\mathbf{C}), \{i, \forall i : 0 \leq i < \text{ncols}(\mathbf{C})\} \rangle$ .

(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. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:

(a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .

(b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

The internal vectors, matrices, and masks are checked for dimension compatibility. The following conditions must hold:

1.  $\text{size}(\tilde{\mathbf{c}}) = \text{size}(\tilde{\mathbf{m}})$

2.  $\text{ncols} = \text{size}(\tilde{\mathbf{u}})$ .

If any compatibility rule above is violated, execution of `GrB_assign` 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 assign and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .
- $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$\tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{J}}[j], \tilde{\mathbf{u}}(j)) \mid \forall j, 0 \leq j < \text{ncols} : j \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

At this point, if any value of  $\tilde{\mathbf{J}}[j]$  is outside the valid range of indices for vector  $\tilde{\mathbf{c}}$ , computation ends and the method returns the index out-of-bounds error listed above. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix,  $\mathbf{C}$ , is invalid from this point forward in the sequence.

The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$\tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{size}(\tilde{\mathbf{c}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure 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 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

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  $\tilde{\mathbf{t}}$ .

$$z_i = \tilde{\mathbf{c}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$z_i = \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}),$$

where the difference operator refers to set difference.

- If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\langle \mathbf{D}_{out}(\text{accum}), \mathbf{size}(\tilde{\mathbf{c}}), \{(j, z_j) \mid \forall j \in \mathbf{ind}(\tilde{\mathbf{c}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$z_j = \tilde{\mathbf{c}}(j) \odot \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}})),$$

$$z_j = \tilde{\mathbf{c}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{c}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

$$z_j = \tilde{\mathbf{t}}(j), \text{ if } j \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{c}}))),$$

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,  $\mathbf{C}(\text{row\_index}, :)$ . 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  $C(\text{row\_index}, :)$  on input to this operation are deleted and the new contents of the column is given by:

$$\mathbf{L}(C) = \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

- 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,  $C(\text{row\_index}, :)$ , and elements of this column that fall outside the set indicated by the mask are unchanged:

$$\begin{aligned} \mathbf{L}(C) = & \{(i, j, C_{ij}) : i \neq \text{row\_index}\} \cup \\ & \{(\text{row\_index}, j, \tilde{c}(j)) : j \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg\widetilde{\mathbf{m}}))\} \cup \\ & \{(\text{row\_index}, j, z_j) : j \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}. \end{aligned}$$

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.

#### 4.3.7.5 assign: Constant vector variant

Assign the same value to a specified subset of vector elements. With the use of GrB\_ALL, the entire destination vector can be filled with the constant.

### C Syntax

```
GrB_Info GrB_assign(GrB_Vector      w,
                    const GrB_Vector mask,
                    const GrB_BinaryOp accum,
                    <type>          val,
                    const GrB_Index *indices,
                    GrB_Index      nindices,
                    const GrB_Descriptor desc);
```

```
GrB_Info GrB_assign(GrB_Vector      w,
                    const GrB_Vector mask,
                    const GrB_BinaryOp accum,
                    const GrB_Scalar s,
                    const GrB_Index *indices,
                    GrB_Index      nindices,
                    const GrB_Descriptor desc);
```

### Parameters

$\mathbf{w}$  (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the assign operation. On output, this

5202  
5203  
5204  
5205  
5206  
5207  
5208  
  
5209  
5210  
5211  
  
5212  
  
5213  
  
5214  
5215  
5216  
5217  
5218  
5219  
5220  
5221  
  
5222  
5223  
  
5224  
5225  
5226

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.

**val** (IN) Scalar value to assign to (a subset of) **w**.

**s** (IN) Scalar value to assign to (a subset of) **w**.

**indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0 to **nindices** – 1, 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. In this variant, the specific order of the values in the array has no effect on the result. Unlike other variants, if there are duplicated values in this array the result is still defined.

**nindices** (IN) The number of values in **indices** array. Must be in the range: [0, **size(w)**]. If **nindices** is zero, the operation becomes a NO-OP.

**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> .

5227

## Return Values

5229  
5230  
5231  
5232  
5233

**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.

5234                   GrB\_PANIC Unknown internal error.

5235                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
5236   opaque GraphBLAS objects (input or output) is in an invalid  
5237   state caused by a previous execution error. Call GrB\_error() to  
5238   access any error messages generated by the implementation.

5239                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

5240                   GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
5241   by a call to new (or dup for vector parameters).

5242                   GrB\_INDEX\_OUT\_OF\_BOUNDS A value in indices is greater than or equal to size(w). In non-  
5243   blocking mode, this can be reported as an execution error.

5244                   GrB\_DIMENSION\_MISMATCH mask and w dimensions are incompatible, or nindices is not less  
5245   than size(w).

5246                   GrB\_DOMAIN\_MISMATCH The domains of the vector and scalar are incompatible with each  
5247   other or the corresponding domains of the accumulation oper-  
5248   ator, or the mask's domain is not compatible with bool (in the  
5249   case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

5250                   GrB\_NULL\_POINTER Argument indices is a NULL pointer.

## 5251 Description

5252 This variant of GrB\_assign computes the result of assigning a constant scalar value – either val or  
5253 s – to locations in a destination GraphBLAS vector. Either w(indices) = val or w(indices) = s is  
5254 performed. If an optional binary accumulation operator ( $\odot$ ) is provided, then either w(indices) =  
5255 w(indices)  $\odot$  val or w(indices) = w(indices)  $\odot$  s is performed. More explicitly, if a non-opaque value  
5256 val is provided:

$$5257 \quad \begin{aligned} &w(\text{indices}[i]) = \text{val}, \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ &w(\text{indices}[i]) = w(\text{indices}[i]) \odot \text{val}, \forall i : 0 \leq i < \text{nindices}. \end{aligned}$$

5258 Correspondingly, if a GrB\_Scalar s is provided:

$$5259 \quad \begin{aligned} &w(\text{indices}[i]) = s, \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ &w(\text{indices}[i]) = w(\text{indices}[i]) \odot s, \forall i : 0 \leq i < \text{nindices}. \end{aligned}$$

5260 Logically, this operation occurs in three steps:

5261       **Setup** The internal vectors and mask used in the computation are formed and their domains  
5262                   and dimensions are tested for compatibility.

5263       **Compute** The indicated computations are carried out.

5264       **Output** The result is written into the output vector, possibly under control of a mask.

5265 Up to two argument vectors are used in the GrB\_assign operation:

- 5266 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 5267 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)

5268 The argument scalar, vectors, and the accumulation operator (if provided) are tested for domain  
5269 compatibility as follows:

- 5270 1. If  $\mathbf{mask}$  is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\mathbf{mask})$   
5271 must be from one of the pre-defined types of Table 3.2.
- 5272 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with either  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(\mathbf{s})$ , depending on the signature of the  
5273 method.
- 5274 3. If  $\mathbf{accum}$  is not GrB\_NULL, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
5275 of the accumulation operator.
- 5276 4. If  $\mathbf{accum}$  is not GrB\_NULL, then either  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(\mathbf{s})$ , depending on the signature of the  
5277 method, must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.

5278 Two domains are compatible with each other if values from one domain can be cast to values in  
5279 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5280 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5281 any compatibility rule above is violated, execution of GrB\_assign ends and the domain mismatch  
5282 error listed above is returned.

5283 From the arguments, the internal vectors, mask and index array used in the computation are formed  
5284 ( $\leftarrow$  denotes copy):

- 5285 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5286 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument  $\mathbf{mask}$  as follows:
  - 5287 (a) If  $\mathbf{mask} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 5288 (b) If  $\mathbf{mask} \neq \text{GrB\_NULL}$ ,
    - 5289 i. If desc[GrB\_MASK].GrB\_STRUCTURE is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 5290 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\text{bool})\mathbf{mask}(i) = \text{true}\} \rangle$ .
  - 5291 (c) If desc[GrB\_MASK].GrB\_COMP is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5292 3. Scalar  $\tilde{s} \leftarrow \mathbf{s}$  (GrB\_Scalar version only).
- 5293 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument indices as follows:
  - 5294 (a) If  $\mathbf{indices} = \text{GrB\_ALL}$ , then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$ .
  - 5295 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$ .

5296 The internal vector and mask are checked for dimension compatibility. The following conditions  
5297 must hold:

- 5298 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5299 2.  $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$ .

5300 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
5301 match error listed above is returned.

5302 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5303 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5304 We are now ready to carry out the assign and any additional associated operations. We describe  
5305 this in terms of two intermediate vectors:

- 5306 •  $\tilde{\mathbf{t}}$ : The vector holding the copies of the scalar, either `val` or  $\tilde{s}$ , in their destination locations  
5307 relative to  $\tilde{\mathbf{w}}$ .
- 5308 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5309 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows. If a non-opaque scalar `val` is provided:

$$5310 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5311 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$5312 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5313 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$5314 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5315 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  
5316 is not in the range  $[0, \mathbf{size}(\tilde{\mathbf{w}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds  
5317 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a  
5318 sequence-terminating `GrB_wait()` is called. Regardless, the result vector,  $\mathbf{w}$ , is invalid from this  
5319 point forward in the sequence.

5320 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 5321 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5322 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5323 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
5324 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  
5325 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5326 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5327 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 5328 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 5329 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}), \\ 5330 \end{aligned}$$

5331 where the difference operator refers to set difference. We note that in this case of assigning  
 5332 a constant,  $\{\tilde{\mathbf{I}}[k], \forall k\}$  and  $\mathbf{ind}(\tilde{\mathbf{t}})$  are identical.

- 5333 • If **accum** is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5334 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5335 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5336 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 5337 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 5338 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 5339 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 5340 \end{aligned}$$

5342 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5343 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 5344 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 5345 mask which acts as a “write mask”.

- 5346 • If **desc[GrB\_OUTP].GrB\_REPLACE** is set, then any values in  $\mathbf{w}$  on input to this operation are  
 5347 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5348 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5349 • If **desc[GrB\_OUTP].GrB\_REPLACE** is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 5350 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 5351 mask are unchanged:

$$5352 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5353 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 5354 of vector  $\mathbf{w}$  is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 5355 exits with return value **GrB\_SUCCESS** and the new content of vector  $\mathbf{w}$  is as defined above but  
 5356 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 5357 sequence.

#### 5358 4.3.7.6 assign: Constant matrix variant

5359 Assign the same value to a specified subset of matrix elements. With the use of **GrB\_ALL**, the  
 5360 entire destination matrix can be filled with the constant.



## 5361 C Syntax

```

5362         GrB_Info GrB_assign(GrB_Matrix      C,
5363                             const GrB_Matrix Mask,
5364                             const GrB_BinaryOp accum,
5365                             <type>          val,
5366                             const GrB_Index  *row_indices,
5367                             GrB_Index        nrows,
5368                             const GrB_Index  *col_indices,
5369                             GrB_Index        ncols,
5370                             const GrB_Descriptor desc);

```

```

5371         GrB_Info GrB_assign(GrB_Matrix      C,
5372                             const GrB_Matrix Mask,
5373                             const GrB_BinaryOp accum,
5374                             const GrB_Scalar  s,
5375                             const GrB_Index  *row_indices,
5376                             GrB_Index        nrows,
5377                             const GrB_Index  *col_indices,
5378                             GrB_Index        ncols,
5379                             const GrB_Descriptor desc);

```

## 5380 Parameters

5381     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
5382     that may be accumulated with the result of the assign operation. On output, the  
5383     matrix holds the results of the operation.

5384     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
5385     stored into the output matrix C. The mask dimensions must match those of the  
5386     matrix C. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5387     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
5388     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5389     dimensions of C), **GrB\_NULL** should be specified.

5390     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
5391     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5392     specified.

5393     **val** (IN) Scalar value to assign to (a subset of) C.

5394     **s** (IN) Scalar value to assign to (a subset of) C.

5395     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of C  
5396     that are assigned. If all rows of C are to be assigned in order from 0 to **nrows** − 1,  
5397     then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
5398     this array may be manipulated by the caller after this operation returns without

5399 affecting any deferred computations for this operation. Unlike other variants, if  
5400 there are duplicated values in this array the result is still defined.

5401 **nrows** (IN) The number of values in **row\_indices** array. Must be in the range:  $[0, \mathbf{nrows}(C)]$ .  
5402 If **nrows** is zero, the operation becomes a NO-OP.

5403 **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**  
5404 that are assigned. If all columns of **C** are to be assigned in order from 0 to  $\mathbf{ncols} - 1$ ,  
5405 then **GrB\_ALL** should be specified. Regardless of execution mode and return value,  
5406 this array may be manipulated by the caller after this operation returns without  
5407 affecting any deferred computations for this operation. Unlike other variants, if  
5408 there are duplicated values in this array the result is still defined.

5409 **ncols** (IN) The number of values in **col\_indices** array. Must be in the range:  $[0, \mathbf{ncols}(C)]$ .  
5410 If **ncols** is zero, the operation becomes a NO-OP.

5411 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5412 should be specified. Non-default field/value pairs are listed as follows:  
5413

Param	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix <b>C</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> matrix. The stored values are not examined.
Mask	GrB_MASK	GrB_COMP	Use the complement of <b>Mask</b> .

## 5415 Return Values

5416 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5417 blocking mode, this indicates that the compatibility tests on  
5418 dimensions and domains for the input arguments passed suc-  
5419 cessfully. Either way, output matrix **C** is ready to be used in the  
5420 next method of the sequence.

5421 **GrB\_PANIC** Unknown internal error.

5422 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
5423 opaque GraphBLAS objects (input or output) is in an invalid  
5424 state caused by a previous execution error. Call **GrB\_error()** to  
5425 access any error messages generated by the implementation.

5426 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

5427 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
5428 by a call to **new** (or **dup** for vector parameters).

5429 GrB\_INDEX\_OUT\_OF\_BOUNDS A value in `row_indices` is greater than or equal to `nrows(C)`, or  
5430 a value in `col_indices` is greater than or equal to `ncols(C)`. In  
5431 non-blocking mode, this can be reported as an execution error.

5432 GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, `nrows` is not less than  
5433 `nrows(C)`, or `ncols` is not less than `ncols(C)`.

5434 GrB\_DOMAIN\_MISMATCH The domains of the matrix and scalar are incompatible with  
5435 each other or the corresponding domains of the accumulation  
5436 operator, or the mask's domain is not compatible with `bool` (in  
5437 the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

5438 GrB\_NULL\_POINTER Either argument `row_indices` is a NULL pointer, argument `col_indices`  
5439 is a NULL pointer, or both.

## 5440 Description

5441 This variant of `GrB_assign` computes the result of assigning a constant scalar value – either `val` or  
5442 `s` – to locations in a destination GraphBLAS matrix: Either `C(row_indices, col_indices) = val`  
5443 or `C(row_indices, col_indices) = s` is performed. If an optional binary accumulation operator  
5444 ( $\odot$ ) is provided, then either `C(row_indices, col_indices) = C(row_indices, col_indices)  $\odot$  val` or  
5445 `C(row_indices, col_indices) = C(row_indices, col_indices)  $\odot$  s` is performed. More explicitly, if a  
5446 non-opaque value `val` is provided:

$$\begin{aligned}
 &C(\text{row\_indices}[i], \text{col\_indices}[j]) = \text{val}, \text{ or} \\
 5447 \quad &C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot \text{val} \\
 &\quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

5448 Correspondingly, if a `GrB_Scalar s` is provided:

$$\begin{aligned}
 &C(\text{row\_indices}[i], \text{col\_indices}[j]) = s, \text{ or} \\
 5449 \quad &C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot s \\
 &\quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

5450 Logically, this operation occurs in three steps:

5451     Setup The internal vectors and mask used in the computation are formed and their domains  
5452     and dimensions are tested for compatibility.

5453     Compute The indicated computations are carried out.

5454     Output The result is written into the output matrix, possibly under control of a mask.

5455 Up to two argument matrices are used in the `GrB_assign` operation:

5456     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

5457 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)

5458 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain  
5459 compatibility as follows:

- 5460 1. If  $\text{Mask}$  is not  $\text{GrB\_NULL}$ , and  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is not set, then  $\mathbf{D}(\text{Mask})$   
5461 must be from one of the pre-defined types of Table 3.2.
- 5462 2.  $\mathbf{D}(\text{C})$  must be compatible with either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\text{val})$ , depending on the signature of the  
5463 method.
- 5464 3. If  $\text{accum}$  is not  $\text{GrB\_NULL}$ , then  $\mathbf{D}(\text{C})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5465 of the accumulation operator.
- 5466 4. If  $\text{accum}$  is not  $\text{GrB\_NULL}$ , then either  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(\text{s})$ , depending on the signature of the  
5467 method, must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

5468 Two domains are compatible with each other if values from one domain can be cast to values in  
5469 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5470 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5471 any compatibility rule above is violated, execution of  $\text{GrB\_assign}$  ends and the domain mismatch  
5472 error listed above is returned.

5473 From the arguments, the internal matrices, index arrays, and mask used in the computation are  
5474 formed ( $\leftarrow$  denotes copy):

- 5475 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 5476 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument  $\text{Mask}$  as follows:
  - 5477 (a) If  $\text{Mask} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{C}), \mathbf{ncols}(\text{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\text{C}), 0 \leq$   
5478  $j < \mathbf{ncols}(\text{C})\} \rangle$ .
  - 5479 (b) If  $\text{Mask} \neq \text{GrB\_NULL}$ ,
    - 5480 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
5481  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 5482 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
5483  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 5484 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 5485 3. Scalar  $\tilde{s} \leftarrow s$  ( $\text{GrB\_Scalar}$  version only).
- 5486 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument  $\text{row\_indices}$  as follows:
  - 5487 (a) If  $\text{row\_indices} = \text{GrB\_ALL}$ , then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - 5488 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 5489 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument  $\text{col\_indices}$  as follows:

- 5490 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .  
 5491 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

5492 The internal matrix and mask are checked for dimension compatibility. The following conditions  
 5493 must hold:

- 5494 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .  
 5495 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .  
 5496 3.  $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$ .  
 5497 4.  $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$ .

5498 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 5499 match error listed above is returned.

5500 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 5501 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5502 We are now ready to carry out the assign and any additional associated operations. We describe  
 5503 this in terms of two intermediate matrices:

- 5504 •  $\tilde{\mathbf{T}}$ : The matrix holding the copies of the scalar, either `val` or  $\tilde{s}$ , in their destination locations  
 5505 relative to  $\tilde{\mathbf{C}}$ .
- 5506 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5507 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows. If a non-opaque scalar `val` is provided:

$$\begin{aligned} 5508 \quad \tilde{\mathbf{T}} = & \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5509 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$\begin{aligned} 5510 \quad \tilde{\mathbf{T}} = & \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ & \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle. \end{aligned}$$

5511 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$5512 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5513 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  
 5514 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  
 5515  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds error listed above is  
 5516 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating  
 5517 `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from this point forward in the  
 5518 sequence.

5519 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

5520 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$\begin{aligned} 5521 \quad \tilde{\mathbf{Z}} &= \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5522 \quad &\{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l], \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle. \end{aligned}$$

5523 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
5524 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  
5525 assigned ( $\{\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l], \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

5526 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5527 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 5528 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l], \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5529 \quad & \\ 5530 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \end{aligned}$$

5531 where the difference operator refers to set difference. We note that, in this particular case of  
5532 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.

5533 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5534 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5535 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5536 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 5537 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 5538 \quad & \\ 5539 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5540 \quad & \\ 5541 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \end{aligned}$$

5542 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5543 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5544 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5545 mask which acts as a “write mask”.

5546 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5547 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5548 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5549 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
5550 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
5551 mask are unchanged:

$$5552 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5553 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5554 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5555 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
5556 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5557 sequence.

### 5558 4.3.8 apply: Apply a function to the elements of an object

5559 Computes the transformation of the values of the elements of a vector or a matrix using a unary  
5560 function, or a binary function where one argument is bound to a scalar.

#### 5561 4.3.8.1 apply: Vector variant

5562 Computes the transformation of the values of the elements of a vector using a unary function.

### 5563 C Syntax

```
5564      GrB_Info GrB_apply(GrB_Vector      w,  
5565                        const GrB_Vector mask,  
5566                        const GrB_BinaryOp accum,  
5567                        const GrB_UnaryOp op,  
5568                        const GrB_Vector u,  
5569                        const GrB_Descriptor desc);
```

### 5570 Parameters

5571 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5572 that may be accumulated with the result of the apply operation. On output, this  
5573 vector holds the results of the operation.

5574 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5575 stored into the output vector **w**. The mask dimensions must match those of the  
5576 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5577 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5578 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5579 dimensions of **w**), **GrB\_NULL** should be specified.

5580 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5581 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5582 specified.

5583 **op** (IN) A unary operator applied to each element of input vector **u**.

5584 **u** (IN) The GraphBLAS vector to which the unary function is applied.

5585 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5586 should be specified. Non-default field/value pairs are listed as follows:  
5587

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\_DIMENSION\_MISMATCH** mask, w and/or u dimensions are incompatible.

**GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the corresponding domains of the accumulation operator or unary function, or the mask's domain is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

## Description

This variant of **GrB\_apply** computes the result of applying a unary function to the elements of a GraphBLAS vector:  $w = f(u)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot f(u)$ .

Logically, this operation occurs in three steps:

**Setup** The internal vectors and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

**Compute** The indicated computations are carried out.



5616 **Output** The result is written into the output vector, possibly under control of a mask.

5617 Up to three argument vectors are used in this **GrB\_apply** operation:

- 5618 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 5619 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 5620 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

5621 The argument vectors, unary operator and the accumulation operator (if provided) are tested for  
5622 domain compatibility as follows:

- 5623 1. If **mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{mask})$   
5624 must be from one of the pre-defined types of Table 3.2.
- 5625 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the unary operator.
- 5626 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
5627 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the unary operator must be compatible with  
5628  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 5629 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in}(\mathbf{op})$ .

5630 Two domains are compatible with each other if values from one domain can be cast to values in  
5631 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5632 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5633 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
5634 error listed above is returned.

5635 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
5636 denotes copy):

- 5637 1. Vector  $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5638 2. One-dimensional mask,  $\widetilde{\mathbf{m}}$ , is computed from argument **mask** as follows:
  - 5639 (a) If **mask** = **GrB\_NULL**, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 5640 (b) If **mask**  $\neq$  **GrB\_NULL**,
    - 5641 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 5642 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 5643 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 5644 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5645 The internal vectors and masks are checked for dimension compatibility. The following conditions  
5646 must hold:

5647 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

5648 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

5649 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5650 error listed above is returned.

5651 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5652 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5653 We are now ready to carry out the apply and any additional associated operations. We describe  
5654 this in terms of two intermediate vectors:

- 5655 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the unary operator to the input vector  $\tilde{\mathbf{u}}$ .
- 5656 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5657 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$5658 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle,$$

5659 where  $f = \mathbf{f}(\text{op})$ .

5660 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 5661 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 5662 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5663 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5664 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5665 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 5666 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 5667 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 5668 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 5669 \quad & \\ 5670 \end{aligned}$$

5671 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5672 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
5673 using what is called a *standard vector mask and replace*. This is carried out under control of the  
5674 mask which acts as a “write mask”.

- 5675 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
5676 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5677 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- 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:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

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.

#### 4.3.8.2 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.

5711 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
5712 should be specified. Non-default field/value pairs are listed as follows:

5713

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.

5714

## 5715 Return Values

5716 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
5717 blocking mode, this indicates that the compatibility tests on  
5718 dimensions and domains for the input arguments passed suc-  
5719 cessfully. Either way, output matrix C is ready to be used in the  
5720 next method of the sequence.

5721 GrB\_PANIC Unknown internal error.

5722 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
5723 opaque GraphBLAS objects (input or output) is in an invalid  
5724 state caused by a previous execution error. Call GrB\_error() to  
5725 access any error messages generated by the implementation.

5726 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

5727 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
5728 by a call to new (or Matrix\_dup for matrix parameters).

5729 GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, nrow  $\neq$  nrow(C), or  
5730 ncol  $\neq$  ncol(C).

5731 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
5732 corresponding domains of the accumulation operator or unary  
5733 function, or the mask's domain is not compatible with bool (in  
5734 the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 5735 Description

5736 This variant of GrB\_apply computes the result of applying a unary function to the elements of a  
5737 GraphBLAS matrix:  $C = f(A)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
5738  $C = C \odot f(A)$ .

5739 Logically, this operation occurs in three steps:

5740     **Setup** The internal matrices and mask used in the computation are formed and their domains  
5741             and dimensions are tested for compatibility.

5742     **Compute** The indicated computations are carried out.

5743     **Output** The result is written into the output matrix, possibly under control of a mask.

5744 Up to three argument matrices are used in the `GrB_apply` operation:

- 5745 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5746 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)
- 5747 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5748 The argument matrices, unary operator and the accumulation operator (if provided) are tested for  
5749 domain compatibility as follows:

- 5750 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
5751     must be from one of the pre-defined types of Table 3.2.
- 5752 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5753 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5754     of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5755      $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5756 4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in}(\text{op})$  of the unary operator.

5757 Two domains are compatible with each other if values from one domain can be cast to values in  
5758 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5759 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5760 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
5761 error listed above is returned.

5762 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
5763 are formed ( $\leftarrow$  denotes copy):

- 5764 1. Matrix  $\tilde{C} \leftarrow C$ .
- 5765 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:
  - 5766 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
5767      $j < \mathbf{ncols}(C)\} \rangle$ .
  - 5768 (b) If `Mask`  $\neq$  `GrB_NULL`,
    - 5769 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
5770      $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,

5771           ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
5772            $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\}\rangle.$   
5773       (c) If  $\mathbf{desc}[\mathbf{GrB\_MASK}].\mathbf{GrB\_COMP}$  is set, then  $\tilde{\mathbf{M}} \leftarrow \neg\tilde{\mathbf{M}}.$   
5774   3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}.$

5775 The internal matrices and mask are checked for dimension compatibility. The following conditions  
5776 must hold:

- 5777   1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}).$
- 5778   2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}).$
- 5779   3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}}).$
- 5780   4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}}).$

5781 If any compatibility rule above is violated, execution of  $\mathbf{GrB\_apply}$  ends and the dimension mismatch  
5782 error listed above is returned.

5783 From this point forward, in  $\mathbf{GrB\_NONBLOCKING}$  mode, the method can optionally exit with  
5784  $\mathbf{GrB\_SUCCESS}$  return code and defer any computation and/or execution error codes.

5785 We are now ready to carry out the apply and any additional associated operations. We describe  
5786 this in terms of two intermediate matrices:

- 5787   •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the unary operator to the input matrix  $\tilde{\mathbf{A}}.$
- 5788   •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5789 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$5790 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\}\rangle,$$

5791 where  $f = \mathbf{f}(\mathbf{op}).$

5792 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 5793   • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}.$
- 5794   • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5795 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

5796 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5797 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}.$

$$\begin{aligned} 5798 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 5799 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5800 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5801 \end{aligned}$$

5803 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5804 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5805 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5806 mask which acts as a “write mask”.

- 5807 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
5808 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5809 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 5810 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
5811 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
5812 mask are unchanged:

$$5813 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5814 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
5815 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
5816 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
5817 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5818 sequence.

#### 5819 4.3.8.3 apply: Vector-BinaryOp variants

5820 Computes the transformation of the values of the stored elements of a vector using a binary operator  
5821 and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the first argument  
5822 to the binary operator and stored elements of the vector are passed as the second argument. In the  
5823 *bind-second* variant, the elements of the vector are passed as the first argument and the specified  
5824 scalar value is passed as the second argument. The scalar can be passed either as a non-opaque  
5825 variable or as a GrB\_Scalar object.

#### 5826 C Syntax

```
5827 // bind-first + scalar value
5828 GrB_Info GrB_apply(GrB_Vector          w,
5829                   const GrB_Vector      mask,
5830                   const GrB_BinaryOp    accum,
5831                   const GrB_BinaryOp    op,
5832                   <type>                val,
5833                   const GrB_Vector      u,
5834                   const GrB_Descriptor   desc);

5835 // bind-first + GraphBLAS scalar
5836 GrB_Info GrB_apply(GrB_Vector          w,
5837                   const GrB_Vector      mask,
```

```

5838         const GrB_BinaryOp    accum,
5839         const GrB_BinaryOp    op,
5840         const GrB_Scalar      s,
5841         const GrB_Vector      u,
5842         const GrB_Descriptor  desc);

5843 // bind-second + scalar value
5844 GrB_Info GrB_apply(GrB_Vector      w,
5845                   const GrB_Vector  mask,
5846                   const GrB_BinaryOp accum,
5847                   const GrB_BinaryOp op,
5848                   const GrB_Vector  u,
5849                   <type>           val,
5850                   const GrB_Descriptor desc);

5851 // bind-second + GraphBLAS scalar
5852 GrB_Info GrB_apply(GrB_Vector      w,
5853                   const GrB_Vector  mask,
5854                   const GrB_BinaryOp accum,
5855                   const GrB_BinaryOp op,
5856                   const GrB_Vector  u,
5857                   const GrB_Scalar  s,
5858                   const GrB_Descriptor desc);

```

## 5859 Parameters

5860 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5861 that may be accumulated with the result of the apply operation. On output, this  
5862 vector holds the results of the operation.

5863 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5864 stored into the output vector **w**. The mask dimensions must match those of the  
5865 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5866 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5867 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5868 dimensions of **w**), **GrB\_NULL** should be specified.

5869 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5870 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5871 specified.

5872 **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar  
5873 value, **val**.

5874 **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as  
5875 the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
5876 argument in the *bind-second* variant.



5877        **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
 5878        argument in the *bind-first* variant, or the right-hand (second) argument in the  
 5879        *bind-second* variant.

5880        **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand  
 5881        (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
 5882        the *bind-second* variant. It must not be empty.

5883        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 5884        should be specified. Non-default field/value pairs are listed as follows:  
 5885

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> .

## 5887 Return Values

5888        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 5889        blocking mode, this indicates that the compatibility tests on di-  
 5890        mensions and domains for the input arguments passed successfully.  
 5891        Either way, output vector **w** is ready to be used in the next method  
 5892        of the sequence.

5893        **GrB\_PANIC** Unknown internal error.

5894        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 5895        GraphBLAS objects (input or output) is in an invalid state caused  
 5896        by a previous execution error. Call **GrB\_error()** to access any error  
 5897        messages generated by the implementation.

5898        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5899        **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 5900        a call to **new** (or **dup** for vector parameters).

5901        **GrB\_DIMENSION\_MISMATCH** **mask**, **w** and/or **u** dimensions are incompatible.

5902        **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors and scalar are incompatible with  
 5903        the corresponding domains of the binary operator or accumulation  
 5904        operator, or the mask's domain is not compatible with **bool** (in the  
 5905        case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

5906        **GrB\_EMPTY\_OBJECT** The **GrB\_Scalar s** used in the call is empty (**nvals(s) = 0**) and  
 5907        therefore a value cannot be passed to the binary operator.

## 5908 Description

5909 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a  
 5910 GraphBLAS vector each composed with a scalar constant, either `val` or `s`:

5911                   bind-first:      $w = f(\text{val}, u)$  or  $w = f(s, u)$

5912                   bind-second:     $w = f(u, \text{val})$  or  $w = f(u, s)$ ,

5913 or if an optional binary accumulation operator ( $\odot$ ) is provided:

5914                   bind-first:      $w = w \odot f(\text{val}, u)$  or  $w = w \odot f(s, u)$

5915                   bind-second:     $w = w \odot f(u, \text{val})$  or  $w = w \odot f(u, s)$ .

5916 Logically, this operation occurs in three steps:

5917     **Setup** The internal vectors and mask used in the computation are formed and their domains  
 5918             and dimensions are tested for compatibility.

5919 **Compute** The indicated computations are carried out.

5920 **Output** The result is written into the output vector, possibly under control of a mask.

5921 Up to three argument vectors are used in this `GrB_apply` operation:

- 5922     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5923     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 5924     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5925 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are  
 5926 tested for domain compatibility as follows:

- 5927     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 5928         must be from one of the pre-defined types of Table 3.2.
- 5929     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.
- 5930     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5931         of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
 5932          $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5933     4.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.
- 5934     5. If bind-first:  
 5935         (a)  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

5936 (b) If the non-opaque scalar **val** is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
 5937 of the binary operator.

5938 (c) If the **GrB\_Scalar** **s** is provided, then  $\mathbf{D}(\text{s})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
 5939 binary operator.

5940 6. If bind-second:

5941 (a)  $\mathbf{D}(\text{u})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

5942 (b) If the non-opaque scalar **val** is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
 5943 of the binary operator.

5944 (c) If the **GrB\_Scalar** **s** is provided, then  $\mathbf{D}(\text{s})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the  
 5945 binary operator.

5946 Two domains are compatible with each other if values from one domain can be cast to values in  
 5947 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5948 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5949 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
 5950 error listed above is returned.

5951 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 5952 denotes copy):

5953 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .

5954 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:

5955 (a) If **mask** = **GrB\_NULL**, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .

5956 (b) If **mask**  $\neq$  **GrB\_NULL**,

5957 i. If desc[**GrB\_MASK**].**GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,

5958 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

5959 (c) If desc[**GrB\_MASK**].**GrB\_COMP** is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

5960 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5961 4. Scalar  $\tilde{s} \leftarrow \mathbf{s}$  (GraphBLAS scalar case).

5962 The internal vectors and masks are checked for dimension compatibility. The following conditions  
 5963 must hold:

5964 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

5965 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

5966 If any compatibility rule above is violated, execution of **GrB\_apply** ends and the dimension mismatch  
 5967 error listed above is returned.

5968 From this point forward, in GrB\_NONBLOCKING mode, the method can optionally exit with  
 5969 GrB\_SUCCESS return code and defer any computation and/or execution error codes.

5970 If an empty GrB\_Scalar  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code GrB\_EMPTY\_OBJECT.  
 5971 If a non-empty GrB\_Scalar,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
 5972  $\mathbf{val}$  with the same domain as  $\tilde{s}$  and set  $\mathbf{val} = \mathbf{val}(\tilde{s})$ .

5973 We are now ready to carry out the apply and any additional associated operations. We describe  
 5974 this in terms of two intermediate vectors:

- 5975 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the binary operator to the input vector  $\tilde{\mathbf{u}}$ .
- 5976 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5977 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as one of the following:

$$\begin{aligned} \text{5978} \quad \text{bind-first:} \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\mathbf{val}, \tilde{\mathbf{u}}(i))) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \\ \text{5979} \quad \text{bind-second:} \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i), \mathbf{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \end{aligned}$$

5980 where  $f = \mathbf{f}(\mathbf{op})$ .

5981 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 5982 • If  $\mathbf{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 5983 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\text{5984} \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5985 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5986 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} \text{5987} \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ \text{5988} \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ \text{5989} \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ \text{5990} \quad & \\ \text{5991} \end{aligned}$$

5992 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5993 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 5994 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 5995 mask which acts as a “write mask”.

- 5996 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is set, then any values in  $\mathbf{w}$  on input to this operation are  
 5997 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\text{5998} \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 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:

$$\mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \widetilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\widetilde{\mathbf{m}}))\}.$$

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.

#### 4.3.8.4 apply: Matrix-BinaryOp variants

Computes the transformation of the values of the stored elements of a matrix using a binary operator and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the first argument to the binary operator and stored elements of the matrix are passed as the second argument. In the *bind-second* variant, the elements of the matrix are passed as the first argument and the specified scalar value is passed as the second argument. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

### C Syntax

```
// bind-first + scalar value
GrB_Info GrB_apply(GrB_Matrix      C,
                   const GrB_Matrix Mask,
                   const GrB_BinaryOp accum,
                   const GrB_BinaryOp op,
                   <type>           val,
                   const GrB_Matrix A,
                   const GrB_Descriptor desc);
```

```
// bind-first + GraphBLAS scalar
GrB_Info GrB_apply(GrB_Matrix      C,
                   const GrB_Matrix Mask,
                   const GrB_BinaryOp accum,
                   const GrB_BinaryOp op,
                   const GrB_Scalar s,
                   const GrB_Matrix A,
                   const GrB_Descriptor desc);
```

```
// bind-second + scalar value
GrB_Info GrB_apply(GrB_Matrix      C,
                   const GrB_Matrix Mask,
```

```

6035         const GrB_BinaryOp    accum,
6036         const GrB_BinaryOp    op,
6037         const GrB_Matrix      A,
6038         <type>                val,
6039         const GrB_Descriptor   desc);

6040 // bind-second + GraphBLAS scalar
6041 GrB_Info GrB_apply(GrB_Matrix      C,
6042                   const GrB_Matrix Mask,
6043                   const GrB_BinaryOp accum,
6044                   const GrB_BinaryOp op,
6045                   const GrB_Matrix  A,
6046                   const GrB_Scalar  s,
6047                   const GrB_Descriptor desc);

```

## 6048 Parameters

6049     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6050     that may be accumulated with the result of the apply operation. On output, the  
6051     matrix holds the results of the operation.

6052     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6053     stored into the output matrix C. The mask dimensions must match those of the  
6054     matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
6055     of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types  
6056     in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
6057     dimensions of C), `GrB_NULL` should be specified.

6058     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
6059     entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
6060     specified.

6061     **op** (IN) A binary operator applied to each element of input matrix, A, with the element  
6062     of the input matrix used as the left-hand argument, and the scalar value, `val`, used  
6063     as the right-hand argument.

6064     **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as  
6065     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
6066     argument in the *bind-second* variant.

6067     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
6068     argument in the *bind-first* variant, or the right-hand (second) argument in the  
6069     *bind-second* variant.

6070     **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand  
6071     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
6072     the *bind-second* variant. 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
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 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 Matrix\_dup for matrix parameters).

GrB\_INDEX\_OUT\_OF\_BOUNDS A value in row\_indices is greater than or equal to nrows(A), or a value in col\_indices is greater than or equal to ncols(A). In non-blocking mode, this can be reported as an execution error.

GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, nrows  $\neq$  nrows(C), or ncols  $\neq$  ncols(C).

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 bool (in the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

6101                   GrB\_EMPTY\_OBJECT The GrB\_Scalar  $s$  used in the call is empty ( $\mathbf{nvals}(s) = 0$ ) and  
 6102                   therefore a value cannot be passed to the binary operator.

### 6103 Description

6104 This variant of GrB\_apply computes the result of applying a binary operator to the elements of a  
 6105 GraphBLAS matrix each composed with a scalar constant,  $\mathbf{val}$  or  $\mathbf{s}$ :

6106                   bind-first:      $C = f(\mathbf{val}, A)$  or  $C = f(\mathbf{s}, A)$

6107                   bind-second:    $C = f(A, \mathbf{val})$  or  $C = f(A, \mathbf{s})$ ,

6108 or if an optional binary accumulation operator ( $\odot$ ) is provided:

6109                   bind-first:      $C = C \odot f(\mathbf{val}, A)$  or  $C = C \odot f(\mathbf{s}, A)$

6110                   bind-second:    $C = C \odot f(A, \mathbf{val})$  or  $C = C \odot f(A, \mathbf{s})$ .

6111 Logically, this operation occurs in three steps:

6112           **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6113           and dimensions are tested for compatibility.

6114           **Compute** The indicated computations are carried out.

6115           **Output** The result is written into the output matrix, possibly under control of a mask.

6116 Up to three argument matrices are used in the GrB\_apply operation:

- 6117     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6118     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)
- 6119     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6120 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are  
 6121 tested for domain compatibility as follows:

- 6122     1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 6123       must be from one of the pre-defined types of Table 3.2.
- 6124     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the binary operator.
- 6125     3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 6126       of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the binary operator must be compatible with  
 6127        $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.



- 6128 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the binary operator.
- 6129 5. If bind-first:
- 6130 (a)  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the binary operator.
- 6131 (b) If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$
- 6132 of the binary operator.
- 6133 (c) If the `GrB_Scalar`  $\mathbf{s}$  is provided, then  $\mathbf{D}(\mathbf{s})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the
- 6134 binary operator.

- 6135 6. If bind-second:
- 6136 (a)  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the binary operator.
- 6137 (b) If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$
- 6138 of the binary operator.
- 6139 (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
- 6140 binary operator.

6141 Two domains are compatible with each other if values from one domain can be cast to values in

6142 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

6143 compatible with each other. A domain from a user-defined type is only compatible with itself. If

6144 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

6145 error listed above is returned.

6146 From the argument matrices, the internal matrices, mask, and index arrays used in the computation

6147 are formed ( $\leftarrow$  denotes copy):

- 6148 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6149 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
- 6150 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
- 6151  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 6152 (b) If `Mask  $\neq$  GrB_NULL`,
- 6153 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
- 6154  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
- 6155 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
- 6156  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
- 6157 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6158 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument `A` as follows:
- 6159 bind-first:  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6160 bind-second:  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6161 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6162 The internal matrices and mask are checked for dimension compatibility. The following conditions  
6163 must hold:

- 6164 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6165 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6166 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6167 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6168 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6169 error listed above is returned.

6170 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6171 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6172 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6173 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6174 `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6175 We are now ready to carry out the apply and any additional associated operations. We describe  
6176 this in terms of two intermediate matrices:

- 6177 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the binary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 6178 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6179 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as one of the following:

6180 bind-first:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\text{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6181 bind-second:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \text{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6182 where  $f = \mathbf{f}(\text{op})$ .

6183 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6184 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6185 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6186 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6187 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6188 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 6189 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6190 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6191 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6192 \end{aligned}$$

6194 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6195 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 6196 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 6197 mask which acts as a “write mask”.

- 6198 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 6199 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6200 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6201 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 6202 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 6203 mask are unchanged:

$$6204 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6205 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 6206 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 6207 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 6208 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 6209 sequence.

#### 6210 4.3.8.5 apply: Vector index unary operator variant

6211 Computes the transformation of the values of the stored elements of a vector using an index unary  
 6212 operator that is a function of the stored value, its location indices, and an user provided scalar  
 6213 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6214 C Syntax

```
6215     GrB_Info GrB_apply(GrB_Vector          w,
6216                       const GrB_Vector    mask,
6217                       const GrB_BinaryOp   accum,
6218                       const GrB_IndexUnaryOp op,
6219                       const GrB_Vector    u,
6220                       <type>              val,
6221                       const GrB_Descriptor desc);
```

```
6222     GrB_Info GrB_apply(GrB_Vector          w,
6223                       const GrB_Vector    mask,
6224                       const GrB_BinaryOp   accum,
6225                       const GrB_IndexUnaryOp op,
6226                       const GrB_Vector    u,
6227                       const GrB_Scalar    s,
6228                       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.

6260                   GrB\_PANIC   Unknown internal error.

6261           GrB\_INVALID\_OBJECT   This is returned in any execution mode whenever one of the  
6262                                   opaque GraphBLAS objects (input or output) is in an invalid  
6263                                   state caused by a previous execution error. Call GrB\_error() to  
6264                                   access any error messages generated by the implementation.

6265           GrB\_OUT\_OF\_MEMORY   Not enough memory available for operation.

6266   GrB\_UNINITIALIZED\_OBJECT   One or more of the GraphBLAS objects has not been initialized  
6267                                   by a call to new (or another constructor).

6268   GrB\_DIMENSION\_MISMATCH   mask, w and/or u dimensions are incompatible.

6269   GrB\_DOMAIN\_MISMATCH   The domains of the various vectors are incompatible with the cor-  
6270                                   responding domains of the accumulation operator or index unary  
6271                                   operator, or the mask's domain is not compatible with bool (in  
6272                                   the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

6273           GrB\_EMPTY\_OBJECT   The GrB\_Scalar s used in the call is empty ( $\mathbf{nvals}(s) = 0$ ) and  
6274                                   therefore a value cannot be passed to the index unary operator.

## 6275 Description

6276   This variant of GrB\_apply computes the result of applying an index unary operator to the elements  
6277   of a GraphBLAS vector each composed with the element's index and a scalar constant, val or s:

$$6278 \quad \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \text{val}) \text{ or } \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}),$$

6279   or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6280 \quad \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \text{val}) \text{ or } \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}).$$

6281   Logically, this operation occurs in three steps:

6282       **Setup**   The internal vectors and mask used in the computation are formed and their domains  
6283                   and dimensions are tested for compatibility.

6284       **Compute**   The indicated computations are carried out.

6285       **Output**   The result is written into the output vector, possibly under control of a mask.

6286   Up to three argument vectors are used in this GrB\_apply operation:

- 6287   1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6288   2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)

6289 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6290 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6291 are tested for domain compatibility as follows:

- 6292 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
6293 must be from one of the pre-defined types of Table 3.2.
- 6294 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the index unary operator.
- 6295 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6296 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be compatible  
6297 with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6298 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6299 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  
6300 the index unary operator.
- 6301 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  
6302 unary operator.

6303 Two domains are compatible with each other if values from one domain can be cast to values in  
6304 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6305 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6306 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
6307 error listed above is returned.

6308 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6309 denotes copy):

- 6310 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6311 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 6312 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 6313 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 6314 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
    - 6315 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 6316 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6317 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6318 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6319 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6320 must hold:

6321 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6322 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

6323 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6324 error listed above is returned.

6325 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6326 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6327 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6328 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided ( $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable `val`  
6329 with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6330 We are now ready to carry out the apply and any additional associated operations. We describe  
6331 this in terms of two intermediate vectors:

- 6332 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6333  $\tilde{\mathbf{u}}$ .
- 6334 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6335 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6336 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6337 where  $f_i = \mathbf{f}(\text{op})$ .

6338 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6339 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 6340 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6341 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6342 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6343 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6344 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6345 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6346 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

6349 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6350 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6351 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6352 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.

#### 4.3.8.6 apply: Matrix index unary operator variant

Computes the transformation of the values of the stored elements of a matrix using an index unary operator that is a function of the stored value, its location indices, and an user provided scalar value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### C Syntax

```

GrB_Info GrB_apply(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);

GrB_Info GrB_apply(GrB_Matrix      C,
                   const GrB_Matrix Mask,
                   const GrB_BinaryOp accum,
                   const GrB_IndexUnaryOp op,
                   const GrB_Matrix A,
                   const GrB_Scalar s,
                   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.



6388       Mask (IN) An optional “write” mask that controls which results from this operation are  
6389       stored into the output matrix C. The mask dimensions must match those of the  
6390       matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
6391       of the Mask matrix must be of type **bool** or any of the predefined “built-in” types  
6392       in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6393       dimensions of C), GrB\_NULL should be specified.

6394       accum (IN) An optional binary operator used for accumulating entries into existing C  
6395       entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
6396       specified.

6397       op (IN) An index unary operator,  $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\text{GrB\_Index}), D_{in_2}, f_i \rangle$ , applied  
6398       to each element stored in the input matrix, A. It is a function of the stored element’s  
6399       value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6400       A (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6401       ator.

6402       val (IN) An additional scalar value that is passed to the index unary operator.

6403       s (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6404       desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
6405       should be specified. Non-default field/value pairs are listed as follows:

6406

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.

6407

## 6408 Return Values

6409       GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
6410       blocking mode, this indicates that the compatibility tests on di-  
6411       mensions and domains for the input arguments passed successfully.  
6412       Either way, output matrix C is ready to be used in the next method  
6413       of the sequence.

6414       GrB\_PANIC Unknown internal error.

6415       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
6416       GraphBLAS objects (input or output) is in an invalid state caused

6417 by a previous execution error. Call `GrB_error()` to access any error  
 6418 messages generated by the implementation.

6419 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6420 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 6421 a call to `new` (or another constructor).

6422 **GrB\_DIMENSION\_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6423 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the  
 6424 corresponding domains of the accumulation operator or index unary  
 6425 operator, or the mask's domain is not compatible with `bool` (in the  
 6426 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6427 **GrB\_EMPTY\_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
 6428 therefore a value cannot be passed to the index unary operator.

## 6429 Description

6430 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements  
 6431 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar  
 6432 constant, `val` or `s`:

$$6433 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6434 or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6435 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6436 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
 6437 indices, respectively.

6438 Logically, this operation occurs in three steps:

6439 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6440 and dimensions are tested for compatibility.

6441 **Compute** The indicated computations are carried out.

6442 **Output** The result is written into the output matrix, possibly under control of a mask.

6443 Up to three argument matrices are used in the `GrB_apply` operation:

- 6444 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6445 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)

6446 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6447 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6448 are tested for domain compatibility as follows:

- 6449 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{Mask})$   
6450 must be from one of the pre-defined types of Table 3.2.
- 6451 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator.
- 6452 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6453 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be compatible  
6454 with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 6455 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6456 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  
6457 the index unary operator.
- 6458 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  
6459 unary operator.

6460 Two domains are compatible with each other if values from one domain can be cast to values in  
6461 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6462 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6463 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
6464 error listed above is returned.

6465 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation  
6466 are formed ( $\leftarrow$  denotes copy):

- 6467 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6468 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument **Mask** as follows:
  - 6469 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
6470  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 6471 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 6472 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
6473  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 6474 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
6475  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 6476 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6477 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument **A** as follows:
 
$$6478 \quad \tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$$
- 6479 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6480 The internal matrices and mask are checked for dimension compatibility. The following conditions  
6481 must hold:

- 6482 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6483 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6484 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6485 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6486 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6487 error listed above is returned.

6488 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6489 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6490 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6491 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6492 `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6493 We are now ready to carry out the apply and any additional associated operations. We describe  
6494 this in terms of two intermediate matrices:

- 6495 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6496  $\tilde{\mathbf{A}}$ .
- 6497 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6498 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6499 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6500 where  $f_i = \mathbf{f}(\mathbf{op})$ .

6501 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6502 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6503 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6504 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6505 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6506 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 6507 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6508 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6509 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6510 \quad & \\ 6511 \end{aligned}$$

6512 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6513 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6514 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6515 mask which acts as a “write mask”.

- 6516 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
6517 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6518 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6519 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6520 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6521 mask are unchanged:

$$6522 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6523 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
6524 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
6525 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
6526 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6527 sequence.

#### 6528 4.3.9 select:

6529 Apply a select operator to the stored elements of an object to determine whether or not to keep  
6530 them.

##### 6531 4.3.9.1 select: Vector variant

6532 Apply a select operator (an index unary operator) to the elements of a vector.

#### 6533 C Syntax

```
6534 // scalar value variant
6535 GrB_Info GrB_select(GrB_Vector          w,
6536                    const GrB_Vector      mask,
6537                    const GrB_BinaryOp    accum,
6538                    const GrB_IndexUnaryOp op,
6539                    const GrB_Vector      u,
6540                    <type>                val,
6541                    const GrB_Descriptor   desc);
6542
6543 // GraphBLAS scalar variant
6544 GrB_Info GrB_select(GrB_Vector          w,
6545                    const GrB_Vector      mask,
```

```

6546         const GrB_BinaryOp      accum,
6547         const GrB_IndexUnaryOp  op,
6548         const GrB_Vector        u,
6549         const GrB_Scalar        s,
6550         const GrB_Descriptor    desc);
6551

```

## 6552 Parameters

6553     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6554     that may be accumulated with the result of the select operation. On output, this  
6555     vector holds the results of the operation.

6556     **mask** (IN) An optional “write” mask that controls which results from this operation are  
6557     stored into the output vector **w**. The mask dimensions must match those of the  
6558     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6559     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6560     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6561     dimensions of **w**), **GrB\_NULL** should be specified.

6562     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6563     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6564     specified.

6565     **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  
6566     to each element stored in the input vector, **u**. It is a function of the stored element’s  
6567     value, its location index, and a user supplied scalar value (either **s** or **val**).

6568     **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-  
6569     ator.

6570     **val** (IN) An additional scalar value that is passed to the index unary operator.

6571     **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6572     not be empty.

6573     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6574     should be specified. Non-default field/value pairs are listed as follows:

6575

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> .

6576

## 6577 Return Values

6578           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
6579                       blocking mode, this indicates that the compatibility tests on di-  
6580                       mensions and domains for the input arguments passed success-  
6581                       fully. Either way, output vector **w** is ready to be used in the next  
6582                       method of the sequence.

6583           GrB\_PANIC Unknown internal error.

6584           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
6585                       opaque GraphBLAS objects (input or output) is in an invalid  
6586                       state caused by a previous execution error. Call **GrB\_error()** to  
6587                       access any error messages generated by the implementation.

6588           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

6589           GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
6590                       by a call to one of its constructors.

6591           GrB\_DIMENSION\_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

6592           GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
6593                       responding domains of the accumulation operator or index unary  
6594                       operator, or the **mask**'s domain is not compatible with **bool** (in  
6595                       the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

6596           GrB\_EMPTY\_OBJECT The **GrB\_Scalar s** used in the call is empty (**nvals(s) = 0**) and  
6597                       therefore a value cannot be passed to the index unary operator.

## 6598 Description

6599 This variant of **GrB\_select** computes the result of applying a index unary operator to select the  
6600 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored  
6601 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding  
6602 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.  
6603 This acts like a functional mask on the input vector as follows:

$$6604 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6605 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6606 Correspondingly, if a **GrB\_Scalar s**, is provided:

$$6607 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6608 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6609 Logically, this operation occurs in three steps:

6610     **Setup** The internal vectors and mask used in the computation are formed and their domains  
6611             and dimensions are tested for compatibility.

6612     **Compute** The indicated computations are carried out.

6613     **Output** The result is written into the output vector, possibly under control of a mask.

6614 Up to three argument vectors are used in this GrB\_select operation:

- 6615     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$   
6616     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)  
6617     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6618 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6619 are tested for domain compatibility as follows:

- 6620     1. If **mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\mathbf{mask})$   
6621         must be from one of the pre-defined types of Table 3.2.  
6622     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .  
6623     3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6624         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
6625         mulation operator.  
6626     4.  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6627         i.e., castable to **bool**.  
6628     5.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.  
6629     6.  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(\mathbf{s})$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$   
6630         of the index unary operator.

6631 Two domains are compatible with each other if values from one domain can be cast to values in  
6632 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6633 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6634 any compatibility rule above is violated, execution of GrB\_select ends and the domain mismatch  
6635 error listed above is returned.

6636 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6637 denotes copy):

- 6638     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .  
6639     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument **mask** as follows:



- 6640 (a) If  $\text{mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
- 6641 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,
- 6642 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
- 6643 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
- 6644 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 6645 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6646 4. Scalar  $\widetilde{\mathbf{s}} \leftarrow \mathbf{s}$  (GrB\_Scalar version only).

6647 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6648 must hold:

- 6649 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6650 2.  $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$ .

6651 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch  
6652 error listed above is returned.

6653 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6654 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6655 If an empty `GrB_Scalar`  $\widetilde{\mathbf{s}}$  is provided (i.e.,  $\text{nvals}(\widetilde{\mathbf{s}}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6656 If a non-empty `GrB_Scalar`,  $\widetilde{\mathbf{s}}$ , is provided (i.e.,  $\text{nvals}(\widetilde{\mathbf{s}}) = 1$ ), we then create an internal variable  
6657 `val` with the same domain as  $\widetilde{\mathbf{s}}$  and set  $\text{val} = \text{val}(\widetilde{\mathbf{s}})$ .

6658 We are now ready to carry out the `select` and any additional associated operations. We describe  
6659 this in terms of two intermediate vectors:

- 6660 •  $\widetilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6661  $\widetilde{\mathbf{u}}$ .
- 6662 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6663 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$6664 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6665 where  $f_i = \mathbf{f}(\text{op})$ .

6666 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6667 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 6668 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$6669 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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

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);
```

```

6705 // GraphBLAS scalar variant
6706 GrB_Info GrB_select(GrB_Matrix          C,
6707                    const GrB_Matrix     Mask,
6708                    const GrB_BinaryOp    accum,
6709                    const GrB_IndexUnaryOp op,
6710                    const GrB_Matrix      A,
6711                    const GrB_Scalar      s,
6712                    const GrB_Descriptor   desc);

```

## 6713 Parameters

6714 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6715 that may be accumulated with the result of the select operation. On output, the  
6716 matrix holds the results of the operation.

6717 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6718 stored into the output matrix **C**. The mask dimensions must match those of the  
6719 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6720 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6721 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6722 dimensions of **C**), **GrB\_NULL** should be specified.

6723 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6724 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6725 specified.

6726 **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  
6727 to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6728 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6729 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6730 ator.

6731 **val** (IN) An additional scalar value that is passed to the index unary operator.

6732 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6733 not be empty.

6734 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6735 should be specified. Non-default field/value pairs are listed as follows:

6736

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:

6766  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ , or  
6767  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ .

6768 Correspondingly, if a GrB\_Scalar, s, is provided:

6769  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ , or  
6770  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ .

6771 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
6772 indices, respectively.

6773 Logically, this operation occurs in three steps:

6774 **Setup** The internal matrices and mask used in the computation are formed and their domains  
6775 and dimensions are tested for compatibility.

6776 **Compute** The indicated computations are carried out.

6777 **Output** The result is written into the output matrix, possibly under control of a mask.

6778 Up to three argument matrices are used in the GrB\_select operation:

- 6779 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6780 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)
- 6781 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6782 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6783 are tested for domain compatibility as follows:

- 6784 1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
6785 must be from one of the pre-defined types of Table 3.2.
- 6786 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 6787 3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6788 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
6789 mulation operator.
- 6790 4.  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6791 i.e., castable to bool.
- 6792 5.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6793 6.  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
6794 of the index unary 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_select` ends and the domain mismatch error listed above is returned.

From the argument matrices, the internal matrices, mask, and index arrays 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  $\tilde{\mathbf{A}}$  is computed from argument `A` as follows:  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$
4. Scalar  $\tilde{s} \leftarrow s$  (`GrB_Scalar` version only).

The internal matrices 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.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

If any compatibility rule above is violated, execution of `GrB_select` 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.

If an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`. If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

We are now ready to carry out the `select` and any additional associated operations. We describe this in terms of two intermediate matrices:

- 6829 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6830  $\tilde{\mathbf{A}}$ .
- 6831 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6832 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6833 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6834 where  $f_i = \mathbf{f}(\text{op})$ .

6835 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6836 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6837 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6838 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6839 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6840 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6841 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6842 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6843 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6844 \quad 6845$$

6846 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6847 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6848 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6849 mask which acts as a “write mask”.

- 6850 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
6851 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6852 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6853 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6854 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6855 mask are unchanged:

$$6856 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6857 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
6858 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
6859 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
6860 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6861 sequence.

### 6862 4.3.10 reduce: Perform a reduction across the elements of an object

6863 Computes the reduction of the values of the elements of a vector or matrix.

#### 6864 4.3.10.1 reduce: Standard matrix to vector variant

6865 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns  
6866 is desired, the input matrix should be transposed using the descriptor.

### 6867 C Syntax

```
6868     GrB_Info GrB_reduce(GrB_Vector      w,  
6869                        const GrB_Vector mask,  
6870                        const GrB_BinaryOp accum,  
6871                        const GrB_Monoid op,  
6872                        const GrB_Matrix A,  
6873                        const GrB_Descriptor desc);  
6874  
6875     GrB_Info GrB_reduce(GrB_Vector      w,  
6876                        const GrB_Vector mask,  
6877                        const GrB_BinaryOp accum,  
6878                        const GrB_BinaryOp op,  
6879                        const GrB_Matrix A,  
6880                        const GrB_Descriptor desc);
```

### 6881 Parameters

6882 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6883 that may be accumulated with the result of the reduction operation. On output,  
6884 this vector holds the results of the operation.

6885 **mask** (IN) An optional “write” mask that controls which results from this operation are  
6886 stored into the output vector **w**. The mask dimensions must match those of the  
6887 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6888 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6889 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6890 dimensions of **w**), **GrB\_NULL** should be specified.

6891 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6892 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6893 specified.

6894 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.  
6895 Depending on which type is passed, the following defines the binary operator with  
6896 one domain,  $F_b = \langle D, D, D, \oplus \rangle$ , that is used:



6897 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
6898 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  
6899 monoid is ignored.

6900 If `op` is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in  
6901 both cases  $\odot(\text{op})$  must be commutative and associative. Otherwise, the outcome  
6902 of the operation is undefined.

6903 `A` (IN) The GraphBLAS matrix on which reduction will be performed.

6904 `desc` (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6905 should be specified. Non-default field/value pairs are listed as follows:  
6906

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.

## 6908 Return Values

6909 `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
6910 blocking mode, this indicates that the compatibility tests on di-  
6911 mensions and domains for the input arguments passed successfully.  
6912 Either way, output vector `w` is ready to be used in the next method  
6913 of the sequence.

6914 `GrB_PANIC` Unknown internal error.

6915 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
6916 GraphBLAS objects (input or output) is in an invalid state caused  
6917 by a previous execution error. Call `GrB_error()` to access any error  
6918 messages generated by the implementation.

6919 `GrB_OUT_OF_MEMORY` Not enough memory available for the operation.

6920 `GrB_UNINITIALIZED_OBJECT` One or more of the GraphBLAS objects has not been initialized by  
6921 a call to `new` (or `dup` for vector parameters).

6922 `GrB_DIMENSION_MISMATCH` `mask`, `w` and/or `u` dimensions are incompatible.

6923 `GrB_DOMAIN_MISMATCH` Either the domains of the various vectors and matrices are incom-  
6924 patible with the corresponding domains of the accumulation oper-  
6925 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).

## 6929 Description

6930 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows  
 6931 of an input matrix:  $w(i) = \bigoplus A(i, :) \forall i$ ; or, if an optional binary accumulation operator is provided,  
 6932  $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$ , where  $\bigoplus = \odot(F_b)$  and  $\odot = \odot(\text{accum})$ .

6933 Logically, this operation occurs in three steps:

6934     **Setup** The internal vector, matrix and mask used in the computation are formed and their  
 6935 domains and dimensions are tested for compatibility.

6936     **Compute** The indicated computations are carried out.

6937     **Output** The result is written into the output vector, possibly under control of a mask.

6938 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6939 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6940 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 6941 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6942 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested  
 6943 for domain compatibility as follows:

- 6944 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 6945 must be from one of the pre-defined types of Table 3.2.
- 6946 2.  $\mathbf{D}(w)$  must be compatible with the domain of the reduction binary operator,  $\mathbf{D}(F_b)$ .
- 6947 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 6948 of the accumulation operator and  $\mathbf{D}(F_b)$ , must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 6949 mulation operator.
- 6950 4.  $\mathbf{D}(A)$  must be compatible with the domain of the binary reduction operator,  $\mathbf{D}(F_b)$ .

6951 Two domains are compatible with each other if values from one domain can be cast to values in  
 6952 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6953 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6954 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
 6955 error listed above is returned.

6956 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 6957 denotes copy):

- 6958 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6959 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 6960 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 6961 (b) If `mask  $\neq$  GrB_NULL`,
- 6962 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 6963 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 6964 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6965 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

6966 The internal vectors and masks are checked for dimension compatibility. The following conditions  
 6967 must hold:

- 6968 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6969 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6970 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-  
 6971 match error listed above is returned.

6972 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 6973 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6974 We carry out the reduce and any additional associated operations. We describe this in terms of  
 6975 two intermediate vectors:

- 6976 •  $\tilde{\mathbf{t}}$ : The vector holding the result from reducing along the rows of input matrix  $\tilde{\mathbf{A}}$ .
- 6977 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6978 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6979 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6980 The value of each of its elements is computed by

$$6981 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6982 where  $\bigoplus = \odot(F_b)$ .

6983 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6984 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

6985 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6986 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6987 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6988 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6989 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 6990 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6991 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6992 \quad & \\ 6993 \end{aligned}$$

6994 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6995 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6996 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6997 mask which acts as a “write mask”.

6998 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
6999 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$7000 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7001 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
7002 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
7003 mask are unchanged:

$$7004 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7005 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
7006 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
7007 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
7008 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7009 sequence.

#### 7010 4.3.10.2 reduce: Vector-scalar variant

7011 Reduce all stored values into a single scalar.

#### 7012 C Syntax

```
7013 // scalar value + monoid (only)
7014 GrB_Info GrB_reduce(<type>          *val,
7015                      const GrB_BinaryOp accum,
7016                      const GrB_Monoid  op,
7017                      const GrB_Vector  u,
```

```

7018             const GrB_Descriptor desc);
7019
7020 // GraphBLAS Scalar + monoid
7021 GrB_Info GrB_reduce(GrB_Scalar      s,
7022                   const GrB_BinaryOp accum,
7023                   const GrB_Monoid  op,
7024                   const GrB_Vector  u,
7025                   const GrB_Descriptor desc);
7026
7027 // GraphBLAS Scalar + binary operator
7028 GrB_Info GrB_reduce(GrB_Scalar      s,
7029                   const GrB_BinaryOp accum,
7030                   const GrB_BinaryOp op,
7031                   const GrB_Vector  u,
7032                   const GrB_Descriptor desc);

```

### 7033 Parameters

7034 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7035 a value that may be accumulated (optionally) with the result of the reduction  
7036 operation. On output, this scalar holds the results of the operation.

7037 **accum** (IN) An optional binary operator used for accumulating entries into an exist-  
7038 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,  
7039 **GrB\_NULL** should be specified.

7040 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7041 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7042 otherwise, the outcome of the operation is undefined.

7043 **u** (IN) The GraphBLAS vector on which reduction will be performed.

7044 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7045 should be specified. Non-default field/value pairs are listed as follows:

7047	Param	Field	Value	Description
------	-------	-------	-------	-------------

7048 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7049 tions. There are currently no non-default field/value pairs that can be set for this  
7050 operation.

### 7051 Return Values

7052 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
7053 cessfully, and the output scalar (**s** or **val**) is ready to be used in the  
7054 next method of the sequence.

7055                   GrB\_PANIC Unknown internal error.

7056           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7057                   GraphBLAS objects (input or output) is in an invalid state caused  
7058                   by a previous execution error. Call GrB\_error() to access any error  
7059                   messages generated by the implementation.

7060           GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7061 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7062                   a call to a respective constructor.

7063           GrB\_NULL\_POINTER val pointer is NULL.

7064           GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
7065                   the corresponding domains of the accumulation operator, or reduce  
7066                   operator.

## 7067 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):

$$\text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7068 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7069 Logically, this operation occurs in three steps:

7070       **Setup** The internal vector used in the computation is formed and its domain is tested for  
7071                   compatibility.

7072       **Compute** The indicated computations are carried out.

7073       **Output** The result is written into the output scalar.

7074 One vector argument is used in this GrB\_reduce operation:

- 7075       1.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7076 The output scalar, argument vector, reduction operator and accumulation operator (if provided)  
7077 are tested for domain compatibility as follows:

- 7078       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  
7079            $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

- 7080 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  
 7081  $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
 7082 be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 7083 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  $F_b$ ).

7084 Two domains are compatible with each other if values from one domain can be cast to values in  
 7085 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7086 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7087 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
 7088 error listed above is returned.

7089 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one  
 7090 of the following occurs depending on the output variant:

$$7091 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7092 or

$$7093 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7094 where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7095 For all other cases, the internal vector and scalar used in the computation is formed ( $\leftarrow$  denotes  
 7096 copy):

- 7097 1. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 7098 2. Scalar  $\tilde{s} \leftarrow \text{s}$  (GraphBLAS scalar case).

7099 We are now ready to carry out the reduction and any additional associated operations. An inter-  
 7100 mediate scalar result  $t$  is computed as follows:

$$7101 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

7102 where  $\oplus = \odot(\text{op})$ .

7103 The final reduction value is computed as follows:

$$7104 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7105 or

$$7106 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB\_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7107 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
7108 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7109 4.3.10.3 reduce: Matrix-scalar variant

7110 Reduce all stored values into a single scalar.

#### 7111 C Syntax

```
7112 // scalar value + monoid (only)
7113 GrB_Info GrB_reduce(<type>          *val,
7114                   const GrB_BinaryOp accum,
7115                   const GrB_Monoid   op,
7116                   const GrB_Matrix   A,
7117                   const GrB_Descriptor desc);
7118
7119 // GraphBLAS Scalar + monoid
7120 GrB_Info GrB_reduce(GrB_Scalar      s,
7121                   const GrB_BinaryOp accum,
7122                   const GrB_Monoid   op,
7123                   const GrB_Matrix   A,
7124                   const GrB_Descriptor desc);
7125
7126 // GraphBLAS Scalar + binary operator
7127 GrB_Info GrB_reduce(GrB_Scalar      s,
7128                   const GrB_BinaryOp accum,
7129                   const GrB_BinaryOp op,
7130                   const GrB_Matrix   A,
7131                   const GrB_Descriptor desc);
```

#### 7132 Parameters

7133 val or s (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7134 a value that may be accumulated (optionally) with the result of the reduction  
7135 operation. On output, this scalar holds the results of the operation.

7136 accum (IN) An optional binary operator used for accumulating entries into existing (s or  
7137 val) value. If assignment rather than accumulation is desired, GrB\_NULL should  
7138 be specified.

7139 op (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7140 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7141 otherwise, the outcome of the operation is undefined.

7142 A (IN) The GraphBLAS matrix on which the reduction will be performed.



7143 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 7144 should be specified. Non-default field/value pairs are listed as follows:  
 7145

7146	Param	Field	Value	Description
------	-------	-------	-------	-------------

7147 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
 7148 tions. There are currently no non-default field/value pairs that can be set for this  
 7149 operation.

## 7150 Return Values

7151 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 7152 cessfully, and the output scalar (s or val) is ready to be used in the  
 7153 next method of the sequence.

7154 GrB\_PANIC Unknown internal error.

7155 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 7156 GraphBLAS objects (input or output) is in an invalid state caused  
 7157 by a previous execution error. Call GrB\_error() to access any error  
 7158 messages generated by the implementation.

7159 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7160 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 7161 a call to a respective constructor.

7162 GrB\_NULL\_POINTER val pointer is NULL.

7163 GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
 7164 the corresponding domains of the accumulation operator, or reduce  
 7165 operator.

## 7166 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):

$$\text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7167 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7168 Logically, this operation occurs in three steps:

7169       **Setup** The internal matrix used in the computation is formed and its domain is tested for  
 7170       compatibility.

7171       **Compute** The indicated computations are carried out.

7172       **Output** The result is written into the output scalar.

7173   One matrix argument is used in this GrB\_reduce operation:

7174       1.  $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7175   The output scalar, argument matrix, reduction operator and accumulation operator (if provided)  
 7176   are tested for domain compatibility as follows:

7177       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  
 7178        $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

7179       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  
 7180        $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
 7181       be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

7182       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  $F_b$ ).

7183   Two domains are compatible with each other if values from one domain can be cast to values in  
 7184   the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7185   compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7186   any compatibility rule above is violated, execution of GrB\_reduce ends and the domain mismatch  
 7187   error listed above is returned.

7188   The number of values stored in the input,  $A$ , is checked. If there are no stored values in  $A$ , then  
 7189   one of the following occurs depending on the output variant:

$$7190 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7191   or

$$7192 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7193   where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with GrB\_SUCCESS.

7194   For all other cases, the internal matrix and scalar used in the computation is formed ( $\leftarrow$  denotes  
 7195   copy):

7196       1. Matrix  $\tilde{A} \leftarrow A$ .

7197       2. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

7198 We are now ready to carry out the reduce and any additional associated operations. An intermediate  
7199 scalar result  $t$  is computed as follows:

$$7200 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7201 where  $\oplus = \odot(\text{op})$ .

7202 The final reduction value is computed as follows:

$$7203 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7204 or

$$7205 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB\_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7206 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
7207 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7208 4.3.11 transpose: Transpose rows and columns of a matrix

7209 This version computes a new matrix that is the transpose of the source matrix.

#### 7210 C Syntax

```
7211      GrB_Info GrB_transpose(GrB_Matrix      C,
7212                           const GrB_Matrix Mask,
7213                           const GrB_BinaryOp accum,
7214                           const GrB_Matrix A,
7215                           const GrB_Descriptor desc);
```

#### 7216 Parameters

7217 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
7218 that may be accumulated with the result of the transpose operation. On output,  
7219 the matrix holds the results of the operation.

7220 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
7221 stored into the output matrix C. The mask dimensions must match those of the  
7222 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
7223 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
7224 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
7225 dimensions of C), GrB\_NULL should be specified.

7226 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7227 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
7228 specified.

7229 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7230 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7231 should be specified. Non-default field/value pairs are listed as follows:  
7232

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.

## 7234 Return Values

7235 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
7236 blocking mode, this indicates that the compatibility tests on di-  
7237 mensions and domains for the input arguments passed successfully.  
7238 Either way, output matrix C is ready to be used in the next method  
7239 of the sequence.

7240 **GrB\_PANIC** Unknown internal error.

7241 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
7242 GraphBLAS objects (input or output) is in an invalid state caused  
7243 by a previous execution error. Call **GrB\_error()** to access any error  
7244 messages generated by the implementation.

7245 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

7246 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
7247 a call to **new** (or **Matrix\_dup** for matrix parameters).

7248 **GrB\_DIMENSION\_MISMATCH** mask, C and/or A dimensions are incompatible.

7249 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the cor-  
7250 responding domains of the accumulation operator, or the mask's do-  
7251 main is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE**  
7252 is not set).

## 7253 Description

7254 GrB\_transpose computes the result of performing a transpose of the input matrix:  $C = A^T$ ; or, if an  
 7255 optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A^T$ . We note that the input matrix  
 7256 A can itself be optionally transposed before the operation, which would cause either an assignment  
 7257 from A to C or an accumulation of A into C.

7258 Logically, this operation occurs in three steps:

7259     **Setup** The internal matrix and mask used in the computation are formed and their domains  
 7260             and dimensions are tested for compatibility.

7261     **Compute** The indicated computations are carried out.

7262     **Output** The result is written into the output matrix, possibly under control of a mask.

7263 Up to three matrix arguments are used in this GrB\_transpose operation:

- 7264     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7265     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)
- 7266     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7267 The argument matrices and accumulation operator (if provided) are tested for domain compatibility  
 7268 as follows:

- 7269     1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 7270         must be from one of the pre-defined types of Table 3.2.
- 7271     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$  of the input matrix.
- 7272     3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 7273         of the accumulation operator and  $\mathbf{D}(A)$  of the input matrix must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 7274         of the accumulation operator.

7275 Two domains are compatible with each other if values from one domain can be cast to values in  
 7276 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7277 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7278 any compatibility rule above is violated, execution of GrB\_transpose ends and the domain mismatch  
 7279 error listed above is returned.

7280 From the argument matrices, the internal matrices and mask used in the computation are formed  
 7281 ( $\leftarrow$  denotes copy):

- 7282     1. Matrix  $\tilde{C} \leftarrow C$ .
- 7283     2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument Mask as follows:

- 7284 (a) If  $\text{Mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
7285  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 7286 (b) If  $\text{Mask} \neq \text{GrB\_NULL}$ ,
- 7287 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
7288  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
- 7289 ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
7290  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
- 7291 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$ .
- 7292 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

7293 The internal matrices and masks are checked for dimension compatibility. The following conditions  
7294 must hold:

- 7295 1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$ .
- 7296 2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$ .
- 7297 3.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$ .
- 7298 4.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$ .

7299 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension  
7300 mismatch error listed above is returned.

7301 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
7302 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7303 We are now ready to carry out the matrix transposition and any additional associated operations.  
7304 We describe this in terms of two intermediate matrices:

- 7305 •  $\widetilde{\mathbf{T}}$ : The matrix holding the transpose of  $\widetilde{\mathbf{A}}$ .
- 7306 •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

7307 The intermediate matrix

$$7308 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7309 is created.

7310 The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7311 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 7312 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$7313 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7314 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 7315 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned}
 7316 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 7317 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7318 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 7319 \quad & \\
 7320 \quad &
 \end{aligned}$$

7321 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7322 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 7323 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 7324 mask which acts as a “write mask”.

- 7325 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 7326 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7327 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7328 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 7329 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 7330 mask are unchanged:

$$7331 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7332 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 7333 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 7334 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 7335 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 7336 sequence.

#### 7337 4.3.12 kronecker: Kronecker product of two matrices

7338 Computes the Kronecker product of two matrices. The result is a matrix.

#### 7339 C Syntax

```

7340      GrB_Info GrB_kronecker(GrB_Matrix      C,
7341                             const GrB_Matrix  Mask,
7342                             const GrB_BinaryOp accum,
7343                             const GrB_Semiring op,
7344                             const GrB_Matrix  A,
7345                             const GrB_Matrix  B,
7346                             const GrB_Descriptor desc);
7347
```

```

7348     GrB_Info GrB_kronecker(GrB_Matrix      C,
7349                           const GrB_Matrix  Mask,
7350                           const GrB_BinaryOp accum,
7351                           const GrB_Monoid   op,
7352                           const GrB_Matrix  A,
7353                           const GrB_Matrix  B,
7354                           const GrB_Descriptor desc);
7355
7356     GrB_Info GrB_kronecker(GrB_Matrix      C,
7357                           const GrB_Matrix  Mask,
7358                           const GrB_BinaryOp accum,
7359                           const GrB_BinaryOp op,
7360                           const GrB_Matrix  A,
7361                           const GrB_Matrix  B,
7362                           const GrB_Descriptor desc);

```

### 7363 Parameters

7364 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
7365 that may be accumulated with the result of the Kronecker product. On output,  
7366 the matrix holds the results of the operation.

7367 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
7368 stored into the output matrix C. The mask dimensions must match those of the  
7369 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
7370 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
7371 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
7372 dimensions of C), GrB\_NULL should be specified.

7373 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7374 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
7375 specified.

7376 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
7377 operation. Depending on which type is passed, the following defines the binary  
7378 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

7379 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

7380 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-  
7381 nored.

7382 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  
7383 is ignored.

7384 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
7385 product.



7386 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
7387 product.

7388 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7389 should be specified. Non-default field/value pairs are listed as follows:  
7390

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.

## 7392 Return Values

7393 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
7394 blocking mode, this indicates that the compatibility tests on di-  
7395 mensions and domains for the input arguments passed successfully.  
7396 Either way, output matrix C is ready to be used in the next method  
7397 of the sequence.

7398 GrB\_PANIC Unknown internal error.

7399 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7400 GraphBLAS objects (input or output) is in an invalid state caused  
7401 by a previous execution error. Call GrB\_error() to access any error  
7402 messages generated by the implementation.

7403 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7404 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7405 a call to new (or Matrix\_dup for matrix parameters).

7406 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

7407 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
7408 corresponding domains of the binary operator (op) or accumulation  
7409 operator, or the mask's domain is not compatible with bool (in the  
7410 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 7411 Description

7412 GrB\_kronecker computes the Kronecker product  $C = A \otimes B$  or, if an optional binary accumulation  
7413 operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$  (where matrices A and B can be optionally transposed).

7414 The Kronecker product is defined as follows:

7415

$$7416 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

7417 where  $A : \mathbb{S}^{m_A \times n_A}$ ,  $B : \mathbb{S}^{m_B \times n_B}$ , and  $C : \mathbb{S}^{m_A m_B \times n_A n_B}$ . More explicitly, the elements of the  
7418 Kronecker product are defined as

$$7419 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7420 where  $\otimes$  is the multiplicative operator specified by the **op** parameter.

7421 Logically, this operation occurs in three steps:

7422     **Setup** The internal matrices and mask used in the computation are formed and their domains  
7423             and dimensions are tested for compatibility.

7424     **Compute** The indicated computations are carried out.

7425     **Output** The result is written into the output matrix, possibly under control of a mask.

7426 Up to four argument matrices are used in the **GrB\_kronecker** operation:

- 7427 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7428 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)
- 7429 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 7430 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

7431 The argument matrices, the "product" operator (**op**), and the accumulation operator (if provided)  
7432 are tested for domain compatibility as follows:

- 7433 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
7434     must be from one of the pre-defined types of Table 3.2.
- 7435 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 7436 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 7437 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 7438 5. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
7439     of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of **op** must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
7440     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.

7474 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$   
7475  $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$  is created. The value of  
7476 each of its elements is computed by

$$7477 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7478 where  $\otimes$  is the multiplicative operator specified by the `op` parameter.

7479 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7480 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 7481 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$7482 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7483 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
7484 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 7485 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 7486 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 7487 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 7488 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 7489 \end{aligned}$$

7490 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7491 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
7492 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
7493 mask which acts as a “write mask”.

- 7494 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
7495 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7496 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7497 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
7498 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
7499 mask are unchanged:

$$7500 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7501 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
7502 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
7503 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
7504 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7505 sequence. s

## Chapter 5

# Nonpolymorphic interface

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
<code>GrB_Monoid_new(GrB_Monoid*,...,bool)</code>	<code>GrB_Monoid_new_BOOL(GrB_Monoid*,GrB_BinaryOp,bool)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,int8_t)</code>	<code>GrB_Monoid_new_INT8(GrB_Monoid*,GrB_BinaryOp,int8_t)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,uint8_t)</code>	<code>GrB_Monoid_new_UINT8(GrB_Monoid*,GrB_BinaryOp,uint8_t)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,int16_t)</code>	<code>GrB_Monoid_new_INT16(GrB_Monoid*,GrB_BinaryOp,int16_t)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,uint16_t)</code>	<code>GrB_Monoid_new_UINT16(GrB_Monoid*,GrB_BinaryOp,uint16_t)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,int32_t)</code>	<code>GrB_Monoid_new_INT32(GrB_Monoid*,GrB_BinaryOp,int32_t)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,uint32_t)</code>	<code>GrB_Monoid_new_UINT32(GrB_Monoid*,GrB_BinaryOp,uint32_t)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,int64_t)</code>	<code>GrB_Monoid_new_INT64(GrB_Monoid*,GrB_BinaryOp,int64_t)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,uint64_t)</code>	<code>GrB_Monoid_new_UINT64(GrB_Monoid*,GrB_BinaryOp,uint64_t)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,float)</code>	<code>GrB_Monoid_new_FP32(GrB_Monoid*,GrB_BinaryOp,float)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,double)</code>	<code>GrB_Monoid_new_FP64(GrB_Monoid*,GrB_BinaryOp,double)</code>
<code>GrB_Monoid_new(GrB_Monoid*,...,other)</code>	<code>GrB_Monoid_new_UDT(GrB_Monoid*,GrB_BinaryOp,void*)</code>

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).

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).

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,...)

Table 5.12: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_get(GrB_Scalar,GrB_Scalar,GrB_Field)	GrB_Scalar_get_Scalar(GrB_Scalar,GrB_Scalar,GrB_Field)
GrB_get(GrB_Scalar,char*,GrB_Field)	GrB_Scalar_get_String(GrB_Scalar,char*,GrB_Field)
GrB_get(GrB_Scalar,int32_t*,GrB_Field)	GrB_Scalar_get_INT32(GrB_Scalar,int32_t*,GrB_Field)
GrB_get(GrB_Scalar,size_t*,GrB_Field)	GrB_Scalar_get_SIZE(GrB_Scalar,size_t*,GrB_Field)
GrB_get(GrB_Scalar,void*,GrB_Field)	GrB_Scalar_get_VOID(GrB_Scalar,void*,GrB_Field)
GrB_get(GrB_Vector,GrB_Scalar,GrB_Field)	GrB_Vector_get_Scalar(GrB_Vector,GrB_Scalar,GrB_Field)
GrB_get(GrB_Vector,char*,GrB_Field)	GrB_Vector_get_String(GrB_Vector,char*,GrB_Field)
GrB_get(GrB_Vector,int32_t*,GrB_Field)	GrB_Vector_get_INT32(GrB_Vector,int32_t*,GrB_Field)
GrB_get(GrB_Vector,size_t*,GrB_Field)	GrB_Vector_get_SIZE(GrB_Vector,size_t*,GrB_Field)
GrB_get(GrB_Vector,void*,GrB_Field)	GrB_Vector_get_VOID(GrB_Vector,void*,GrB_Field)
GrB_get(GrB_Matrix,GrB_Scalar,GrB_Field)	GrB_Matrix_get_Scalar(GrB_Matrix,GrB_Scalar,GrB_Field)
GrB_get(GrB_Matrix,char*,GrB_Field)	GrB_Matrix_get_String(GrB_Matrix,char*,GrB_Field)
GrB_get(GrB_Matrix,int32_t*,GrB_Field)	GrB_Matrix_get_INT32(GrB_Matrix,int32_t*,GrB_Field)
GrB_get(GrB_Matrix,size_t*,GrB_Field)	GrB_Matrix_get_SIZE(GrB_Matrix,size_t*,GrB_Field)
GrB_get(GrB_Matrix,void*,GrB_Field)	GrB_Matrix_get_VOID(GrB_Matrix,void*,GrB_Field)
GrB_get(GrB_UnaryOp,GrB_Scalar,GrB_Field)	GrB_UnaryOp_get_Scalar(GrB_UnaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_UnaryOp,char*,GrB_Field)	GrB_UnaryOp_get_String(GrB_UnaryOp,char*,GrB_Field)
GrB_get(GrB_UnaryOp,int32_t*,GrB_Field)	GrB_UnaryOp_get_INT32(GrB_UnaryOp,int32_t*,GrB_Field)
GrB_get(GrB_UnaryOp,size_t*,GrB_Field)	GrB_UnaryOp_get_SIZE(GrB_UnaryOp,size_t*,GrB_Field)
GrB_get(GrB_UnaryOp,void*,GrB_Field)	GrB_UnaryOp_get_VOID(GrB_UnaryOp,void*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)	GrB_IndexUnaryOp_get_Scalar(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_IndexUnaryOp,char*,GrB_Field)	GrB_IndexUnaryOp_get_String(GrB_IndexUnaryOp,char*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,int32_t*,GrB_Field)	GrB_IndexUnaryOp_get_INT32(GrB_IndexUnaryOp,int32_t*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,size_t*,GrB_Field)	GrB_IndexUnaryOp_get_SIZE(GrB_IndexUnaryOp,size_t*,GrB_Field)
GrB_get(GrB_IndexUnaryOp,void*,GrB_Field)	GrB_IndexUnaryOp_get_VOID(GrB_IndexUnaryOp,void*,GrB_Field)
GrB_get(GrB_BinaryOp,GrB_Scalar,GrB_Field)	GrB_BinaryOp_get_Scalar(GrB_BinaryOp,GrB_Scalar,GrB_Field)
GrB_get(GrB_BinaryOp,char*,GrB_Field)	GrB_BinaryOp_get_String(GrB_BinaryOp,char*,GrB_Field)
GrB_get(GrB_BinaryOp,int32_t*,GrB_Field)	GrB_BinaryOp_get_INT32(GrB_BinaryOp,int32_t*,GrB_Field)
GrB_get(GrB_BinaryOp,size_t*,GrB_Field)	GrB_BinaryOp_get_SIZE(GrB_BinaryOp,size_t*,GrB_Field)
GrB_get(GrB_BinaryOp,void*,GrB_Field)	GrB_BinaryOp_get_VOID(GrB_BinaryOp,void*,GrB_Field)
GrB_get(GrB_Monoid,GrB_Scalar,GrB_Field)	GrB_Monoid_get_Scalar(GrB_Monoid,GrB_Scalar,GrB_Field)
GrB_get(GrB_Monoid,char*,GrB_Field)	GrB_Monoid_get_String(GrB_Monoid,char*,GrB_Field)
GrB_get(GrB_Monoid,int32_t*,GrB_Field)	GrB_Monoid_get_INT32(GrB_Monoid,int32_t*,GrB_Field)
GrB_get(GrB_Monoid,size_t*,GrB_Field)	GrB_Monoid_get_SIZE(GrB_Monoid,size_t*,GrB_Field)
GrB_get(GrB_Monoid,void*,GrB_Field)	GrB_Monoid_get_VOID(GrB_Monoid,void*,GrB_Field)
GrB_get(GrB_Semiring,GrB_Scalar,GrB_Field)	GrB_Semiring_get_Scalar(GrB_Semiring,GrB_Scalar,GrB_Field)
GrB_get(GrB_Semiring,char*,GrB_Field)	GrB_Semiring_get_String(GrB_Semiring,char*,GrB_Field)
GrB_get(GrB_Semiring,int32_t*,GrB_Field)	GrB_Semiring_get_INT32(GrB_Semiring,int32_t*,GrB_Field)
GrB_get(GrB_Semiring,size_t*,GrB_Field)	GrB_Semiring_get_SIZE(GrB_Semiring,size_t*,GrB_Field)
GrB_get(GrB_Semiring,void*,GrB_Field)	GrB_Semiring_get_VOID(GrB_Semiring,void*,GrB_Field)
GrB_get(GrB_Descriptor,GrB_Scalar,GrB_Field)	GrB_Descriptor_get_Scalar(GrB_Descriptor,GrB_Scalar,GrB_Field)
GrB_get(GrB_Descriptor,char*,GrB_Field)	GrB_Descriptor_get_String(GrB_Descriptor,char*,GrB_Field)
GrB_get(GrB_Descriptor,int32_t*,GrB_Field)	GrB_Descriptor_get_INT32(GrB_Descriptor,int32_t*,GrB_Field)
GrB_get(GrB_Descriptor,size_t*,GrB_Field)	GrB_Descriptor_get_SIZE(GrB_Descriptor,size_t*,GrB_Field)
GrB_get(GrB_Descriptor,void*,GrB_Field)	GrB_Descriptor_get_VOID(GrB_Descriptor,void*,GrB_Field)
GrB_get(GrB_Type,GrB_Scalar,GrB_Field)	GrB_Type_get_Scalar(GrB_Type,GrB_Scalar,GrB_Field)
GrB_get(GrB_Type,char*,GrB_Field)	GrB_Type_get_String(GrB_Type,char*,GrB_Field)
GrB_get(GrB_Type,int32_t*,GrB_Field)	GrB_Type_get_INT32(GrB_Type,int32_t*,GrB_Field)
GrB_get(GrB_Type,size_t*,GrB_Field)	GrB_Type_get_SIZE(GrB_Type,size_t*,GrB_Field)
GrB_get(GrB_Type,void*,GrB_Field)	GrB_Type_get_VOID(GrB_Type,void*,GrB_Field)
GrB_get(GrB_Global,GrB_Scalar,GrB_Field)	GrB_Global_get_Scalar(GrB_Global,GrB_Scalar,GrB_Field)
GrB_get(GrB_Global,char*,GrB_Field)	GrB_Global_get_String(GrB_Global,char*,GrB_Field)
GrB_get(GrB_Global,int32_t*,GrB_Field)	GrB_Global_get_INT32(GrB_Global,int32_t*,GrB_Field)
GrB_get(GrB_Global,size_t*,GrB_Field)	GrB_Global_get_SIZE(GrB_Global,size_t*,GrB_Field)
GrB_get(GrB_Global,void*,GrB_Field)	GrB_Global_get_VOID(GrB_Global,void*,GrB_Field)



Table 5.13: Long-name, nonpolymorphic form of GraphBLAS methods (continued).

Polymorphic signature	Nonpolymorphic signature
GrB_set(GrB_Scalar,GrB_Scalar,GrB_Field)	GrB_Scalar_set_Scalar(GrB_Scalar,GrB_Scalar,GrB_Field)
GrB_set(GrB_Scalar,char*,GrB_Field)	GrB_Scalar_set_String(GrB_Scalar,char*,GrB_Field)
GrB_set(GrB_Scalar,int32_t,GrB_Field)	GrB_Scalar_set_INT32(GrB_Scalar,int32_t,GrB_Field)
GrB_set(GrB_Scalar,void*,GrB_Field,size_t)	GrB_Scalar_set_VOID(GrB_Scalar,void*,GrB_Field,size_t)
GrB_set(GrB_Vector,GrB_Scalar,GrB_Field)	GrB_Vector_set_Scalar(GrB_Vector,GrB_Scalar,GrB_Field)
GrB_set(GrB_Vector,char*,GrB_Field)	GrB_Vector_set_String(GrB_Vector,char*,GrB_Field)
GrB_set(GrB_Vector,int32_t,GrB_Field)	GrB_Vector_set_INT32(GrB_Vector,int32_t,GrB_Field)
GrB_set(GrB_Vector,void*,GrB_Field,size_t)	GrB_Vector_set_VOID(GrB_Vector,void*,GrB_Field,size_t)
GrB_set(GrB_Matrix,GrB_Scalar,GrB_Field)	GrB_Matrix_set_Scalar(GrB_Matrix,GrB_Scalar,GrB_Field)
GrB_set(GrB_Matrix,char*,GrB_Field)	GrB_Matrix_set_String(GrB_Matrix,char*,GrB_Field)
GrB_set(GrB_Matrix,int32_t,GrB_Field)	GrB_Matrix_set_INT32(GrB_Matrix,int32_t,GrB_Field)
GrB_set(GrB_Matrix,void*,GrB_Field,size_t)	GrB_Matrix_set_VOID(GrB_Matrix,void*,GrB_Field,size_t)
GrB_set(GrB_UnaryOp,GrB_Scalar,GrB_Field)	GrB_UnaryOp_set_Scalar(GrB_UnaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_UnaryOp,char*,GrB_Field)	GrB_UnaryOp_set_String(GrB_UnaryOp,char*,GrB_Field)
GrB_set(GrB_UnaryOp,int32_t,GrB_Field)	GrB_UnaryOp_set_INT32(GrB_UnaryOp,int32_t,GrB_Field)
GrB_set(GrB_UnaryOp,void*,GrB_Field,size_t)	GrB_UnaryOp_set_VOID(GrB_UnaryOp,void*,GrB_Field,size_t)
GrB_set(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)	GrB_IndexUnaryOp_set_Scalar(GrB_IndexUnaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_IndexUnaryOp,char*,GrB_Field)	GrB_IndexUnaryOp_set_String(GrB_IndexUnaryOp,char*,GrB_Field)
GrB_set(GrB_IndexUnaryOp,int32_t,GrB_Field)	GrB_IndexUnaryOp_set_INT32(GrB_IndexUnaryOp,int32_t,GrB_Field)
GrB_set(GrB_IndexUnaryOp,void*,GrB_Field,size_t)	GrB_IndexUnaryOp_set_VOID(GrB_IndexUnaryOp,void*,GrB_Field,size_t)
GrB_set(GrB_BinaryOp,GrB_Scalar,GrB_Field)	GrB_BinaryOp_set_Scalar(GrB_BinaryOp,GrB_Scalar,GrB_Field)
GrB_set(GrB_BinaryOp,char*,GrB_Field)	GrB_BinaryOp_set_String(GrB_BinaryOp,char*,GrB_Field)
GrB_set(GrB_BinaryOp,int32_t,GrB_Field)	GrB_BinaryOp_set_INT32(GrB_BinaryOp,int32_t,GrB_Field)
GrB_set(GrB_BinaryOp,void*,GrB_Field,size_t)	GrB_BinaryOp_set_VOID(GrB_BinaryOp,void*,GrB_Field,size_t)
GrB_set(GrB_Monoid,GrB_Scalar,GrB_Field)	GrB_Monoid_set_Scalar(GrB_Monoid,GrB_Scalar,GrB_Field)
GrB_set(GrB_Monoid,char*,GrB_Field)	GrB_Monoid_set_String(GrB_Monoid,char*,GrB_Field)
GrB_set(GrB_Monoid,int32_t,GrB_Field)	GrB_Monoid_set_INT32(GrB_Monoid,int32_t,GrB_Field)
GrB_set(GrB_Monoid,void*,GrB_Field,size_t)	GrB_Monoid_set_VOID(GrB_Monoid,void*,GrB_Field,size_t)
GrB_set(GrB_Semiring,GrB_Scalar,GrB_Field)	GrB_Semiring_set_Scalar(GrB_Semiring,GrB_Scalar,GrB_Field)
GrB_set(GrB_Semiring,char*,GrB_Field)	GrB_Semiring_set_String(GrB_Semiring,char*,GrB_Field)
GrB_set(GrB_Semiring,int32_t,GrB_Field)	GrB_Semiring_set_INT32(GrB_Semiring,int32_t,GrB_Field)
GrB_set(GrB_Semiring,void*,GrB_Field,size_t)	GrB_Semiring_set_VOID(GrB_Semiring,void*,GrB_Field,size_t)
GrB_set(GrB_Descriptor,GrB_Scalar,GrB_Field)	GrB_Descriptor_set_Scalar(GrB_Descriptor,GrB_Scalar,GrB_Field)
GrB_set(GrB_Descriptor,char*,GrB_Field)	GrB_Descriptor_set_String(GrB_Descriptor,char*,GrB_Field)
GrB_set(GrB_Descriptor,int32_t,GrB_Field)	GrB_Descriptor_set_INT32(GrB_Descriptor,int32_t,GrB_Field)
GrB_set(GrB_Descriptor,void*,GrB_Field,size_t)	GrB_Descriptor_set_VOID(GrB_Descriptor,void*,GrB_Field,size_t)
GrB_set(GrB_Type,GrB_Scalar,GrB_Field)	GrB_Type_set_Scalar(GrB_Type,GrB_Scalar,GrB_Field)
GrB_set(GrB_Type,char*,GrB_Field)	GrB_Type_set_String(GrB_Type,char*,GrB_Field)
GrB_set(GrB_Type,int32_t,GrB_Field)	GrB_Type_set_INT32(GrB_Type,int32_t,GrB_Field)
GrB_set(GrB_Type,void*,GrB_Field,size_t)	GrB_Type_set_VOID(GrB_Type,void*,GrB_Field,size_t)
GrB_set(GrB_Global,GrB_Scalar,GrB_Field)	GrB_Global_set_Scalar(GrB_Global,GrB_Scalar,GrB_Field)
GrB_set(GrB_Global,char*,GrB_Field)	GrB_Global_set_String(GrB_Global,char*,GrB_Field)
GrB_set(GrB_Global,int32_t,GrB_Field)	GrB_Global_set_INT32(GrB_Global,int32_t,GrB_Field)
GrB_set(GrB_Global,void*,GrB_Field,size_t)	GrB_Global_set_VOID(GrB_Global,void*,GrB_Field,size_t)



# Appendix A

## Revision history

Changes in 2.1.0 (Released: ## Xxxxx 2023):

- Added GrB\_get and GrB\_set methods, and associated field values to GrB\_field.
- Added GrB\_Type\_Code.
- Added GrB\_DEFAULT.
- Added GrB\_COMP\_STRUCTURE.
- Added GrB\_ALREADY\_SET.
- Allow deserialization when input type parameter is NULL.

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.**

- 7534 • (Issue GH-51) Removed deprecated `GrB_SCMP` literal from descriptor values. **This breaks**  
7535 **backward compatibility.**
- 7536 • (Issues BB-8, BB-36) Added sparse `GrB_Scalar` object and its use in additional variants of  
7537 `extract/setElement` methods, and `reduce`, `apply`, `assign` and `select` operations.
- 7538 • (Issues BB-34, GH-33, GH-45) Added new `select` operation that uses an index unary operator.  
7539 Added new variants of `apply` that take an index unary operator (matrix and vector variants).
- 7540 • (Issues BB-68, BB-51) Added `serialize` and `deserialize` methods for matrices to/from imple-  
7541 mentation defined formats.
- 7542 • (Issues BB-25, GH-42) Added `import` and `export` methods for matrices to/from API specified  
7543 formats. Three formats have been specified: `CSC`, `CSR`, `COO`. Dense row and column formats  
7544 have been deferred.
- 7545 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7546 • (Issue BB-73) Allow `GrB_NULL` for `dup` operator in matrix and vector `build` methods. Return  
7547 error if duplicate locations encountered.
- 7548 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7549 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7550 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type `T` (not to  
7551 be confused with the proposed unary operator).
- 7552 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added  
7553 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to  
7554 a method that is not supported by the implementation.
- 7555 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque  
7556 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7557 • (Issue BB-45) Removed language about annihilators.
- 7558 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7559 • Updated a number algorithms in the appendix to use new operations and methods.
- 7560 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the  
7561 specification.
- 7562 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and  
7563 `GRB_SUBVERSION`.
- 7564 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7565 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult/set` intersection  
7566 fixed to read `eWiseAdd/set` union.
- 7567 • Typographical error (`NEQ`  $\rightarrow$  `NE`) in Description of Table 3.8.

7568 Changes in 1.3.0 (Released: 25 September 2019):

- 7569 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque  
7570 GraphBLAS object as an argument.
- 7571 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7572 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a  
7573 scalar.
- 7574 • (Issue BB-59) Changed specification about how reductions to scalar (`GrB_reduce`) are to be  
7575 performed (to minimize dependence on monoid identity).
- 7576 • (Issue BB-24) Added methods to resize matrices and vectors (`GrB_Matrix_resize` and `GrB_Vector_resize`).
- 7577 • (Issue BB-47) Added methods to remove single elements from matrices and vectors (`GrB_Matrix_removeElement`  
7578 and `GrB_Vector_removeElement`).
- 7579 • (Issue BB-41) Added `GrB_STRUCTURE` descriptor flag for masks (consider only the structure  
7580 of the mask and not the values).
- 7581 • (Issue BB-64) Deprecated `GrB_SCMP` in favor of new `GrB_COMP` for descriptor values.
- 7582 • (Issue BB-46) Added predefined descriptors covering all possible combinations of field, value  
7583 pairs.
- 7584 • Added unary operators: absolute value (`GrB_ABS_T`) and bitwise complement of integers  
7585 (`GrB_BNOT_I`).
- 7586 • (Issues BB-42, BB-62) Added binary operators: Added boolean exclusive-nor (`GrB_LXNOR`)  
7587 and bitwise logical operators on integers (`GrB_BOR_I`, `GrB_BAND_I`, `GrB_BXOR_I`, `GrB_BXNOR_I`).
- 7588 • (Issue BB-11) Added a set of predefined monoids and semirings.
- 7589 • (Issue BB-57) Updated all examples in the appendix to take advantage of new capabilities  
7590 and predefined objects.
- 7591 • (Issue BB-43) Added parent-BFS example.
- 7592 • (Issue BB-1) Fixed bug in the non-batch betweenness centrality algorithm in Appendix C.4  
7593 where source nodes were incorrectly assigned path counts.
- 7594 • (Issue BB-3) Added compile-time preprocessor defines and runtime method for querying the  
7595 GraphBLAS API version being used.
- 7596 • (Issue BB-10) Clarified `GrB_init()` and `GrB_finalize()` errors.
- 7597 • (Issue BB-16) Clarified behavior of boolean and integer division. **Note that `GrB_MINV` for  
7598 integer and boolean types was removed from this version of the spec.**
- 7599 • (Issue BB-19) Clarified aliasing in user-defined operators.

- 7600 • (Issue BB-20) Clarified language about behavior of `GrB_free()` with predefined objects (im-
  - 7601 plementation defined)
  - 7602 • (Issue BB-55) Clarified that multiplication does not have to distribute over addition in a
  - 7603 GraphBLAS semiring.
  - 7604 • (Issue BB-45) Removed unnecessary language about annihilators.
  - 7605 • (Issue BB-61) Removed unnecessary language about implied zeros.
  - 7606 • (Issue BB-60) Added disclaimer against overspecification.
  - 7607 • Fixed miscellaneous typographical errors (such as  $\otimes.\oplus$ ).
- 7608 Changes in 1.2.0:
- 7609 • Removed "provisional" clause.
- 7610 Changes in 1.1.0:
- 7611 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
  - 7612 `assign` operations.
  - 7613 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
  - 7614 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
  - 7615 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
  - 7616 from the domains of the binary operator provided.
  - 7617 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
  - 7618 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
  - 7619 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
  - 7620 which is returned when the capacities of the output arrays are insufficient to hold all of the
  - 7621 tuples.
  - 7622 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
  - 7623 face.
  - 7624 • Added replace flag ( $z$ ) notation to Table 4.1.
  - 7625 • Updated the “Mathematical Description” of the `assign` operation in Table 4.1.
  - 7626 • Added triangle counting example.
  - 7627 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
  - 7628 sections of GraphBLAS operations when the respective text was the “standard” text (i.e.,
  - 7629 identical in a majority of the operations).
  - 7630 • Fixed typographical errors.

7631 Changes in 1.0.2:

- 7632 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate  
7633 matrices and avoid casting issues in certain implementations.
- 7634 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being  
7635 erased outside the assigned area.
- 7636 • Changes non-polymorphic interface:
  - 7637 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.
  - 7638 – Renamed `GrB_Vector_reduce_BinaryOp` to `GrB_Matrix_reduce_BinaryOp`.
  - 7639 – Renamed `GrB_Vector_reduce_Monoid` to `GrB_Matrix_reduce_Monoid`.
- 7640 • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7641 • 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.

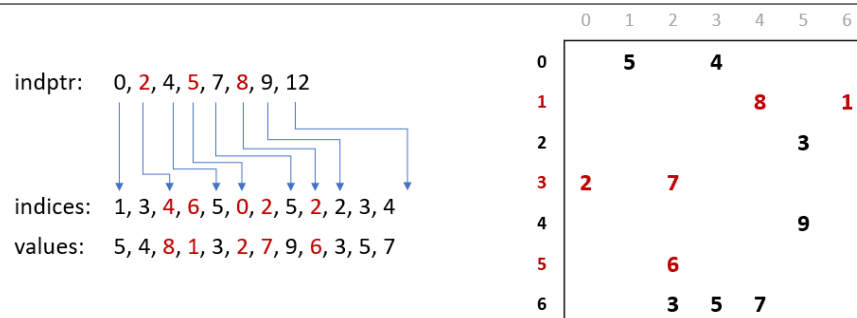


Figure B.1: Data layout for CSR format.

### 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.

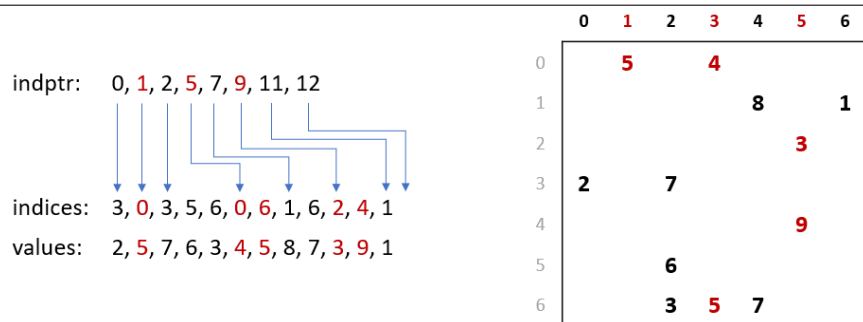


Figure B.2: Data layout for CSC format.

### 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.

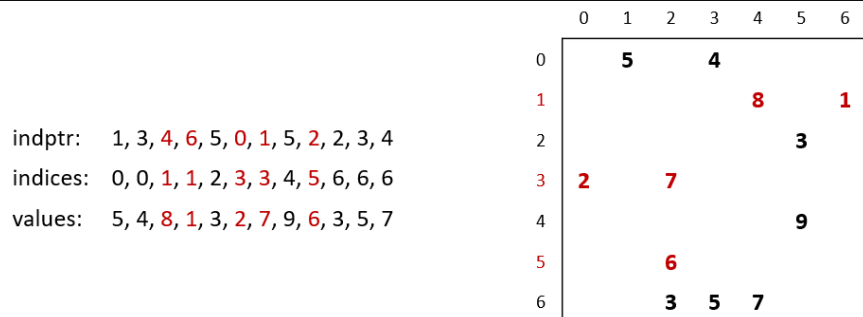


Figure B.3: Data layout for COO format.

7674 **Appendix C**

7675 **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); //  $\text{Matrix}<\text{int32}_t> \text{sigma}(n, n)$ 
20                                           //  $\text{sigma}[d, k] = \#$  shortest paths to node  $k$  at level  $d$ 
21
22     GrB_Vector q;
23     GrB_Vector_new(&q, GrB_INT32, n);        // Vector<int32_t>  $q(n)$  of path counts
24     GrB_Vector_setElement(q, 1, s);          //  $q[s] = 1$ 
25
26     GrB_Vector p;
27     GrB_Vector_dup(&p, q);                   // Vector<int32_t>  $p(n)$  shortest path counts so far
28                                           //  $p = q$ 
29
30     GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
31             q, A, GrB_DESC_RC);              // get the first set of out neighbors
32
33     /*
34     * BFS phase
35     */
36     GrB_Index d = 0;                          // BFS level number
37     int32_t sum = 0;                          //  $\text{sum} == 0$  when BFS phase is complete
38
39     do {
40         GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\text{sigma}[d, :] = q$ 
41         GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
42         GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
43                 q, A, GrB_DESC_RC);        //  $q = \#$  paths to nodes reachable
44                                           // from current level
45         GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
46         ++d;
47     } while (sum);
48
49     /*
50     * BC computation phase
51     * ( $t_1, t_2, t_3, t_4$ ) are temporary vectors
52     */
53     GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
54     GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
55     GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
56     GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
57
58     for (int i=d-1; i>0; i--)
59     {
60         GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t_1 = 1 + \delta$ 
61         GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, t1, *delta, GrB_NULL);
62         GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t_2 = \text{sigma}[i, :]$ 
63         GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t_2 = (1 + \delta) / \text{sigma}[i, :]$ 
64         GrB_mvx(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
65                 // 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) = d^th 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_m xv(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_m xv(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 }
```