

# 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 2023/11/01 at 10:14:52 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, and Massachusetts  
10 Institute of Technology Lincoln Laboratory.

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

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

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

# Contents

25		
26	List of Tables . . . . .	9
27	List of Figures . . . . .	11
28	Acknowledgments . . . . .	12
29	<b>1 Introduction</b>	<b>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 (?)
- Timothy G. Mattson (Intel Corporation)
- 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

- Intel Corporation
- Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute [DM-0003727, DM19-0929, DM21-0090]
- International Business Machines Corporation

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



# Chapter 1

## Introduction

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

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

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

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

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

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

is not possible to express the signature of that method.

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

The remainder of this document is organized as follows:

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



## Chapter 2

# Basic concepts

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

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

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

## 2.1 Glossary

### 2.1.1 GraphBLAS API basic definitions

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

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

### 2.1.2 GraphBLAS objects and their structure

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

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

- *mask*: An internal GraphBLAS object used to control how values are stored in a method's output object. The mask exists only inside a method; hence, it is called an *internal opaque object*. A mask is formed from the elements of a collection object (vector or matrix) input as a mask parameter to a method. GraphBLAS allows two types of masks:

1. In the default case, an element of the mask exists for each element that exists in the input collection object when the value of that element, when cast to a Boolean type, evaluates to `true`.

2. In the *structure only* case, masks have structure but no values. The input collection describes a structure whereby an element of the mask exists for each element stored in the input collection regardless of its value.

- *complement*: The *complement* of a GraphBLAS mask,  $M$ , is another mask,  $M'$ , where the elements of  $M'$  are those elements from  $M$  that *do not* exist.

### 2.1.3 Algebraic structures used in the GraphBLAS

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

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

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

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

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

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

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

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

### 2.1.5 GraphBLAS methods: behaviors and error conditions

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

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

489 • *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as param-  
490 eters to a GraphBLAS method are dimension (or shape) compatible if they have the correct  
491 number of dimensions and sizes for each dimension to satisfy the rules of the mathematical def-  
492 inition of the operation associated with the method. If any *dimension compatibility* rule above  
493 is violated, execution of the GraphBLAS method ends and the GrB\_DIMENSION\_MISMATCH  
494 error is returned.

495 • *domain compatible*: Two domains for which values from one domain can be cast to values in  
496 the other domain as per the rules of the C language. In particular, domains from Table 3.2  
497 are all compatible with each other, and a domain from a user-defined type is only compatible  
498 with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS  
499 method ends and the GrB\_DOMAIN\_MISMATCH error is returned.

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

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

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

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

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

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

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

712 Completion is the central concept that allows GraphBLAS objects to be used in happens-before  
713 relations between threads. In earlier versions of GraphBLAS (1.X) completion was implied by  
714 any operation that produced non-opaque values from a GraphBLAS object. These operations are  
715 summarized in Table 2.2). In GraphBLAS 2.0, these methods no longer imply completion. This  
716 change was made since there are cases where the non-opaque value is needed but the object from  
717 which it is computed is not. We want implementations of the GraphBLAS to be able to exploit  
718 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

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

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

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

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

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

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

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

### 804    3.3    Types (domains)

805    In GraphBLAS, domains correspond to the valid values for types from the host language (in our  
806    case, the C programming language). GraphBLAS defines a number of operators that take elements  
807    from one or more domains and produce elements of a (possibly) different domain. GraphBLAS  
808    also defines three kinds of collections: matrices, vectors and scalars. For any given collection, the  
809    elements of the collection belong to a *domain*, which is the set of valid values for the elements. For  
810    any variable or object  $V$  in GraphBLAS we denote as  $\mathbf{D}(V)$  the domain of  $V$ , that is, the set of  
811    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.

---

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

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

855 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  
856 domains,  $D_{out}$ ,  $D_{in_1}$ ,  $D_{in_2}$ , the domain of GraphBLAS indices, and an operation  $f_i : D_{in_1} \times I_{U64}^2 \times$   
857  $D_{in_2} \rightarrow D_{out}$  (where  $I_{U64}$  corresponds to the domain of a  $\text{GrB\_Index}$ ). For a given GraphBLAS  
858 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  $\mathbf{f}(F_i) = f_i$ .

859 User-defined operators can be created with calls to  $\text{GrB\_UnaryOp\_new}$ ,  $\text{GrB\_BinaryOp\_new}$ , and  
860  $\text{GrB\_IndexUnaryOp\_new}$ , respectively. See Section 4.2.2 for information on these methods. The  
861 GraphBLAS C API predefines a number of these operators. These are listed in Tables 3.5 and 3.6.  
862 Note that most entries in these tables represent a “family” of predefined operators for a set of  
863 different types represented by the  $T$ ,  $I$ , or  $F$  in their names. For example, the multiplicative  
864 inverse ( $\text{GrB\_MINV\_F}$ ) function is only defined for floating-point types ( $F = \text{FP32}$  or  $\text{FP64}$ ). The  
865 division ( $\text{GrB\_DIV\_T}$ ) function is defined for all types, but only if  $y \neq 0$  for integral and floating  
866 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)
		–	$I_{U64}$	$I_{32/64}$	$I_{32/64}$	$f(u_i, i, 0, s) = (i + s)$
GrB_IndexUnaryOp	GrB_COLINDEX_ $I_{32/64}$	–	$I_{U64}$	$I_{32/64}$	$I_{32/64}$	$f(A_{ij}, i, j, s) = (j + s)$ replace with its column index (+ s)
GrB_IndexUnaryOp	GrB_DIAGINDEX_ $I_{32/64}$	–	$I_{U64}$	$I_{32/64}$	$I_{32/64}$	$f(A_{ij}, i, j, s) = (j - i + s)$ replace with its diagonal index (+ s)
GrB_IndexUnaryOp	GrB_TRIL	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j \leq i + s)$ triangle on or below diagonal s
GrB_IndexUnaryOp	GrB_TRIU	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j \geq i + s)$ triangle on or above diagonal s
GrB_IndexUnaryOp	GrB_DIAG	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j == i + s)$ diagonal s
GrB_IndexUnaryOp	GrB_OFFDIAG	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j \neq i + s)$ all but diagonal s
GrB_IndexUnaryOp	GrB_COLLE	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j \leq s)$ columns less or equal to s
GrB_IndexUnaryOp	GrB_COLGT	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (j > s)$ columns greater than s
GrB_IndexUnaryOp	GrB_ROWLE	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (i \leq s)$ , rows less or equal to s
		–	$I_{U64}$	$I_{64}$	bool	$f(u_i, i, 0, s) = (i \leq s)$
GrB_IndexUnaryOp	GrB_ROWGT	–	$I_{U64}$	$I_{64}$	bool	$f(A_{ij}, i, j, s) = (i > s)$ , rows greater than s
		–	$I_{U64}$	$I_{64}$	bool	$f(u_i, i, 0, s) = (i > s)$
GrB_IndexUnaryOp	GrB_VALUEEQ_ $T$	$T$	–	$T$	bool	$f(A_{ij}, i, j, s) = (A_{ij} == s)$ , elements equal to value s
		$T$	–	$T$	bool	$f(u_i, i, 0, s) = (u_i == s)$
GrB_IndexUnaryOp	GrB_VALUENE_ $T$	$T$	–	$T$	bool	$f(A_{ij}, i, j, s) = (A_{ij} \neq s)$ , elements not equal to value s
		$T$	–	$T$	bool	$f(u_i, i, 0, s) = (u_i \neq s)$
GrB_IndexUnaryOp	GrB_VALUELT_ $T$	$T$	–	$T$	bool	$f(A_{ij}, i, j, s) = (A_{ij} < s)$ , elements less than value s
		$T$	–	$T$	bool	$f(u_i, i, 0, s) = (u_i < s)$
GrB_IndexUnaryOp	GrB_VALUELE_ $T$	$T$	–	$T$	bool	$f(A_{ij}, i, j, s) = (A_{ij} \leq s)$ , elements less or equal to value s
		$T$	–	$T$	bool	$f(u_i, i, 0, s) = (u_i \leq s)$
GrB_IndexUnaryOp	GrB_VALUEGT_ $T$	$T$	–	$T$	bool	$f(A_{ij}, i, j, s) = (A_{ij} > s)$ , elements greater than value s
		$T$	–	$T$	bool	$f(u_i, i, 0, s) = (u_i > s)$
GrB_IndexUnaryOp	GrB_VALUEGE_ $T$	$T$	–	$T$	bool	$f(A_{ij}, i, j, s) = (A_{ij} \geq s)$ , elements greater or equal to value s
		$T$	–	$T$	bool	$f(u_i, i, 0, s) = (u_i \geq s)$



### 3.4.2 Monoids

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

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

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

### 3.4.3 Semirings

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

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

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

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

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

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

---

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

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

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

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

GraphBLAS identifier	Domains, $T$ ( $T \times T \rightarrow T$ )	+ identity $\times$ annihilator	Description
GrB_PLUS_TIMES_SEMIRING_ $T$	$\text{UINT}x$ $\text{INT}x$ $\text{FP}x$	0 0 0	arithmetic semiring
GrB_MIN_PLUS_SEMIRING_ $T$	$\text{UINT}x$ $\text{INT}x$ $\text{FP}x$	$\text{UINT}x\_MAX$ $\text{INT}x\_MAX$ $INFINITY$	min-plus semiring
GrB_MAX_PLUS_SEMIRING_ $T$	$\text{INT}x$ $\text{FP}x$	$\text{INT}x\_MIN$ $-INFINITY$	max-plus semiring
GrB_MIN_TIMES_SEMIRING_ $T$	$\text{UINT}x$	$\text{UINT}x\_MAX$	min-times semiring
GrB_MIN_MAX_SEMIRING_ $T$	$\text{UINT}x$ $\text{INT}x$ $\text{FP}x$	$\text{UINT}x\_MAX$ $\text{INT}x\_MAX$ $INFINITY$	min-max semiring
GrB_MAX_MIN_SEMIRING_ $T$	$\text{UINT}x$ $\text{INT}x$ $\text{FP}x$	0 $\text{INT}x\_MIN$ $-INFINITY$	max-min semiring
GrB_MAX_TIMES_SEMIRING_ $T$	$\text{UINT}x$	0	max-times semiring
GrB_PLUS_MIN_SEMIRING_ $T$	$\text{UINT}x$	0	plus-min semiring
GrB_LOR_LAND_SEMIRING_BOOL	BOOL	false	Logical semiring
GrB_LAND_LOR_SEMIRING_BOOL	BOOL	true	"and-or" semiring
GrB_LXOR_LAND_SEMIRING_BOOL	BOOL	false	same as NE_LAND
GrB_LXNOR_LOR_SEMIRING_BOOL	BOOL	true	same as EQ_LOR

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
	$\text{INT}x$	$\text{INT}x\_MIN$	
	$\text{FP}x$	$-INFINITY$	

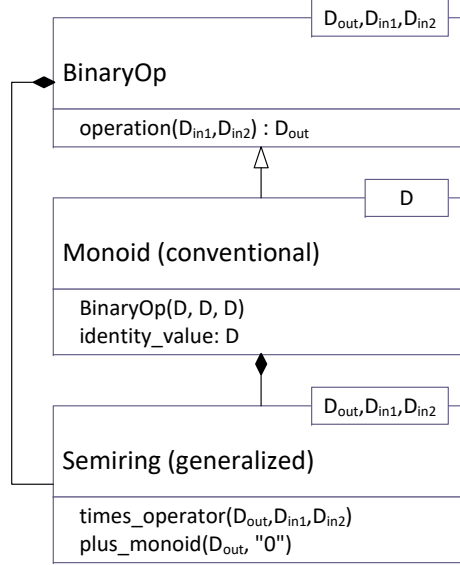


Figure 3.1: Hierarchy of algebraic object classes in GraphBLAS. GraphBLAS semirings consist of a conventional monoid with one domain for the addition function, and a binary operator with three domains for the multiplication function.

## 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

### 1017 3.7.1 Input Types

1018 Allowable types used in `GrB_get` and `GrB_set` are `INT32`, `GrB_Scalar`, `char*`, `void*`, and `SIZE`. Each  
1019 `GrB_Field` is associated with exactly one of these types as defined in Table 3.13. Implementations  
1020 that add additional `GrB_Fields` must document the type associated with each `GrB_Field`.

#### 1021 3.7.1.1 INT32 Handling

1022 `INT32` types use a 32-bit signed integer type. This can be used both for numeric values as well as  
1023 enumerated C types. Enumerated types must specify the numeric value for each enum, and the  
1024 value specified must fit within the allowable 32-bit signed integer range.

#### 1025 3.7.1.2 GrB\_Scalar Handling

1026 When calling `GrB_get`, the user must provide an already initialized `GrB_Scalar` object to which  
1027 the implementation will write a value of the correct element type. When calling `GrB_set`, the  
1028 `GrB_Scalar` must not be empty, otherwise a `GrB_EMPTY_OBJECT` error is raised.

#### 1029 3.7.1.3 String (char\*) Handling

1030 When the input to `GrB_set` is a `char*` the input array is null terminated. The `GraphBLAS` imple-  
1031 mentation must copy this array into internal data structures. Using `GrB_get` for strings requires  
1032 two calls. First, call `GrB_get` with the field and object, but pass `size_t*` as the value argument.  
1033 The implementation will return the size of the string buffer that the user must create. Second, call  
1034 `GrB_get` with the field and object, this time passing a pointer to the newly created string buffer.  
1035 The `GraphBLAS` implementation will write to this buffer, including a trailing null terminator. The  
1036 size returned in the first call will include enough bytes for the null terminator.

#### 1037 3.7.1.4 void\* Handling

1038 When the input to `GrB_set` is a `void*`, an extra `size_t` argument is passed to indicate the size of the  
1039 buffer. The `GraphBLAS` implementation must copy this many bytes from the buffer into internal  
1040 data structures. Similar to reading strings, `GrB_get` must be called twice for `void*`. The first call  
1041 passes `size_t*` to find the required buffer size. The user must create a buffer and then pass the  
1042 pointer to `GrB_get`. The implementation will write to this buffer. No standard specification or  
1043 protocol is required for the contents of `void*`. It is meant to be a mechanism to allow full freedom  
1044 for `GraphBLAS` implementations with needs that cannot be handled using `INT32`, `GrB_Scalar`, or  
1045 `Strings`.

#### 1046 3.7.1.5 SIZE Handling

1047 `SIZE` types use a `size_t` type. Normally, `SIZE` is used in conjunction with `char*` and `void*` to indicate  
1048 the buffer size. However, it can also be used when the actual return type is `size_t`, as is the case

1049 for the size of a Type.

### 1050 3.7.2 Hints

1051 Several fields are *hints* (marked H in Table 3.13). Hints are used to represent intended use cases  
1052 or best guesses, but do not determine strict behavior. When `GrB_set` is called with a hint, the  
1053 GraphBLAS implementation should return `GrB_SUCCESS`, but is free to use or ignore the hint.  
1054 When `GrB_get` is called, the implementation should return a best guess on the correct answer. If  
1055 there is no clear answer, the implementation should return `GrB_UNKNOWN`.

### 1056 3.7.3 GrB\_NAME

1057 The `GrB_NAME` field is a special case regarding writability. All user-defined objects have a  
1058 `GrB_NAME` field which defaults to an empty string. Collections and `GrB_Descriptors` may have  
1059 their `GrB_NAME` set at any time. User-defined algebraic objects and `GrB_Types` may only have  
1060 their `GrB_NAME` set once to a globally unique value. Attempting to set this field after it has  
1061 already been set will return a `GrB_ALREADY_SET` error code.

1062 Built-in algebraic objects and `GrB_Types` have names which can be read, but not written to. The  
1063 name returned will be the string form of the `GrB_Type` listed in Table 3.2 or the GraphBLAS  
1064 identifier listed in Tables 3.5, 3.6, 3.7, 3.8, and 3.9. For example, the name of `GrB_BOOL` type  
1065 is "`GrB_BOOL`" (8 characters) and the name of `GrB_MIN_FP64` binary op is "`GrB_MIN_FP64`" (12  
1066 characters).

1067 The `GrB_NAME` of the global context is read-only and returns the name of the library implemen-  
1068 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`.

## 1091 **Return Values**

1092 `GrB_SUCCESS` operation completed successfully.

1093 `GrB_PANIC` unknown internal error.

1094 `GrB_INVALID_VALUE` invalid mode specified, or method called multiple times.

## 1095 **Description**

1096 The `init` method creates and initializes a GraphBLAS C API context. The argument to `GrB_init`  
1097 defines the mode for the context. The two available modes are:

- 1098 • `GrB_BLOCKING`: In this mode, each method in a sequence returns after its computations have  
1099 completed and output arguments are available to subsequent statements in an application.  
1100 When executing in `GrB_BLOCKING` mode, the methods execute in program order.
- 1101 • `GrB_NONBLOCKING`: In this mode, methods in a sequence may return after arguments in  
1102 the method have been tested for dimension and domain compatibility within the method  
1103 but potentially before their computations complete. Output arguments are available to sub-  
1104 sequent GraphBLAS methods in an application. When executing in `GrB_NONBLOCKING`  
1105 mode, the methods in a sequence may execute in any order that preserves the mathematical  
1106 result defined by the sequence.

1107 An application can only create one context per execution instance. An application may only call  
1108 `GrB_Init` once. Calling `GrB_Init` more than once results in undefined behavior.

### 1109 **4.1.2 finalize: Finalize a GraphBLAS context**

1110 Terminates and frees any internal resources created to support the GraphBLAS C API context.

## 1111 **C Syntax**

1112 `GrB_Info GrB_finalize();`

## 1113 **Return Values**

1114 `GrB_SUCCESS` operation completed successfully.

1115 `GrB_PANIC` unknown internal error.

## 1116 **Description**

1117 The `finalize` method terminates and frees any internal resources created to support the GraphBLAS  
1118 C API context. `GrB_finalize` may only be called after a context has been initialized by calling  
1119 `GrB_init`, or else undefined behavior occurs. After `GrB_finalize` has been called to finalize a Graph-  
1120 BLAS context, calls to any GraphBLAS methods, including `GrB_finalize`, will result in undefined  
1121 behavior.

### 1122 **4.1.3 getVersion: Get the version number of the standard.**

1123 Query the library for the version number of the standard that this library implements.

## 1124 **C Syntax**

```
1125         GrB_Info GrB_getVersion(unsigned int *version,  
1126                                unsigned int *subversion);
```

## 1127 **Parameters**

1128 version (OUT) On successful return will hold the value of the major version number.

1129 version (OUT) On successful return will hold the value of the subversion number.

## 1130 **Return Values**

1131 GrB\_SUCCESS operation completed successfully.

1132 GrB\_PANIC unknown internal error.

## 1133 **Description**

1134 The `getVersion` method is used to query the major and minor version number of the GraphBLAS  
1135 C API specification that the library implements at runtime. To support compile time queries the  
1136 following two macros shall also be defined by the library.

```
1137         #define GRB_VERSION      2  
1138         #define GRB_SUBVERSION  0
```

## 1139 **4.2 Object methods**

1140 This section describes methods that setup and operate on GraphBLAS opaque objects but are not  
1141 part of the the GraphBLAS math specification.

## 1142 4.2.1 Get and Set methods

1143 The methods in this section query and, optionally, set internal fields of GraphBLAS objects.

### 1144 4.2.1.1 get: Query the value of an object

#### 1145 C Syntax

```
1146 GrB_Info GrB_get(GrB_<OBJ> o, <type> value, GrB_Field field);
```

#### 1147 Parameters

1148 OBJ (IN) An existing, valid GraphBLAS object (collection, operation, type) which is  
1149 being queried. To indicate the global context, the constant `GrB_Global` is used.

1150 value (OUT) A pointer to or `GrB_Scalar` containing a value whose type is dependent on  
1151 field which will be filled with the current value of the field. type may be `int32_t*`,  
1152 `size_t*`, `GrB_Scalar`, `char*` or `void*`.

1153 field (IN) The field being queried.

#### 1154 Return Value

1155 `GrB_SUCCESS` The method completed successfully.

1156 `GrB_PANIC` unknown internal error.

1157 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1158 `GrB_UNINITIALIZED_OBJECT` the value parameter is `GrB_Scalar` and has not been initialized by  
1159 a call to `new`.

1160 `GrB_INVALID_VALUE` invalid value type provided for the field or invalid field.

#### 1161 Description

1162 Queries a field of an existing GraphBLAS object. The type of the argument is uniquely determined  
1163 by field. For the case of `char*` and `void*`, the value can be replaced with `size_t*` to get the required  
1164 buffer size to hold the response. Fields marked as hints in Table 3.13 will return a hint on how  
1165 best to use the object.

### 1166 4.2.1.2 set: Set field of an object

1167 Set the content for a field for an existing GraphBLAS object.

## 1168 C Syntax

```
1169     GrB_Info GrB_set(GrB_<OBJ> o, <type> value, GrB_Field field);
1170     GrB_Info GrB_set(GrB_<OBJ> o, void *value, GrB_Field field, size_t voidSize);
```

## 1171 Parameters

1172 OBJ (IN) The GraphBLAS object which is having field set. To indicate  
1173 the global context, the constant `GrB_Global` is used.

1174 value (IN) A value whose type is dependent on field. type may be a  
1175 `int32_t`, `GrB_Scalar`, `char*` or `void*`.

1176 field (IN) The field being set.

1177 voidSize (IN) The size of the `void*` buffer. Note that a size is not needed for  
1178 `char*` because the string is assumed null-terminated.

## 1179 Return Values

1180 `GrB_SUCCESS` The method completed successfully.

1181 `GrB_PANIC` unknown internal error.

1182 `GrB_OUT_OF_MEMORY` not enough memory available for operation.

1183 `GrB_UNINITIALIZED_OBJECT` the `GrB_Scalar` parameter has not been initialized by a call to `new`.

1184 `GrB_INVALID_VALUE` invalid value set on the field, invalid field, or field is read-only.

1185 `GrB_ALREADY_SET` this field has already been set, and may only be set once.

## 1186 Description

1187 Set a field of OBJ or the Global context to a new value.

## 1188 4.2.2 Algebra methods

### 1189 4.2.2.1 Type\_new: Construct a new GraphBLAS (user-defined) type

1190 Creates a new user-defined GraphBLAS type. This type can then be used to create new operators,  
1191 monoids, semirings, vectors and matrices.

## 1192 C Syntax

```
1193         GrB_Info GrB_Type_new(GrB_Type  *utype,  
1194                               size_t     sizeof(ctype));
```

## 1195 Parameters

1196 **utype** (INOUT) On successful return, contains a handle to the newly created user-defined  
1197 GraphBLAS type object.

1198 **ctype** (IN) A C type that defines the new GraphBLAS user-defined type.

## 1199 Return Values

1200 **GrB\_SUCCESS** operation completed successfully.

1201 **GrB\_PANIC** unknown internal error.

1202 **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1203 **GrB\_NULL\_POINTER** utype pointer is NULL.

## 1204 Description

1205 Given a C type **ctype**, the **Type\_new** method returns in **utype** a handle to a new GraphBLAS type  
1206 that is equivalent to the C type. Variables of this **ctype** must be a struct, union, or fixed-size array.  
1207 In particular, given two variables, **src** and **dst**, of type **ctype**, the following operation must be a  
1208 valid way to copy the contents of **src** to **dst**:

```
1209         memcpy(&dst, &src, sizeof(ctype))
```

1210 A new, user-defined type **utype** should be destroyed with a call to **GrB\_free(utype)** when no longer  
1211 needed.

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

### 1214 4.2.2.2 UnaryOp\_new: Construct a new GraphBLAS unary operator

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

## 1217 C Syntax

```
1218     GrB_Info GrB_UnaryOp_new(GrB_UnaryOp *unary_op,  
1219                             void          (*unary_func)(void*, const void*),  
1220                             GrB_Type      d_out,  
1221                             GrB_Type      d_in);
```

## 1222 Parameters

1223 unary\_op (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1224 unary operator object.

1225 unary\_func (IN) a pointer to a user-defined function that takes one input parameter of d\_in's  
1226 type and returns a value of d\_out's type, both passed as void pointers. Specifically  
1227 the signature of the function is expected to be of the form:

```
1228         void func(void *out, const void *in);
```

1230 d\_out (IN) The GrB\_Type of the return value of the unary operator being created. Should  
1231 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1232 BLAS type.

1233 d\_in (IN) The GrB\_Type of the input argument of the unary operator being created.  
1234 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1235 GraphBLAS type.

## 1236 Return Values

1237 GrB\_SUCCESS operation completed successfully.

1238 GrB\_PANIC unknown internal error.

1239 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1240 GrB\_UNINITIALIZED\_OBJECT any GrB\_Type parameter (for user-defined types) has not been ini-  
1241 tialized by a call to GrB\_Type\_new.

1242 GrB\_NULL\_POINTER unary\_op or unary\_func pointers are NULL.

## 1243 Description

1244 The UnaryOp\_new method creates a new GraphBLAS unary operator

1245  $f_u = \langle \mathbf{D}(d\_out), \mathbf{D}(d\_in), unary\_func \rangle$

1246 and returns a handle to it in `unary_op`.

1247 The implementation of `unary_func` must be such that it works even if the `d_out` and `d_in` arguments  
1248 are aliased. In other words, for all invocations of the function:

```
1249     unary_func(out,in);
```

1250 the value of `out` must be the same as if the following code was executed:

```
1251     D(d_in) *tmp = malloc(sizeof(D(d_in)));  
1252     memcpy(tmp,in,sizeof(D(d_in)));  
1253     unary_func(out,tmp);  
1254     free(tmp);
```

1255 It is not an error to call this method more than once on the same variable; however, the handle to  
1256 the previously created object will be overwritten.

#### 1257 4.2.2.3 BinaryOp\_new: Construct a new GraphBLAS binary operator

1258 Initializes a new GraphBLAS binary operator with a specified user-defined function and its types  
1259 (domains).

### 1260 C Syntax

```
1261     GrB_Info GrB_BinaryOp_new(GrB_BinaryOp *binary_op,  
1262                               void          (*binary_func)(void*,  
1263                               const void*,  
1264                               const void*),  
1265                               GrB_Type      d_out,  
1266                               GrB_Type      d_in1,  
1267                               GrB_Type      d_in2);
```

### 1268 Parameters

1269 `binary_op` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1270 binary operator object.

1271 `binary_func` (IN) A pointer to a user-defined function that takes two input parameters of types  
1272 `d_in1` and `d_in2` and returns a value of type `d_out`, all passed as void pointers.  
1273 Specifically the signature of the function is expected to be of the form:

```
1274     void func(void *out, const void *in1, const void *in2);  
1275
```



1276 **d\_out** (IN) The **GrB\_Type** of the return value of the binary operator being created. Should  
1277 be one of the predefined GraphBLAS types in Table 3.2, or a user-defined Graph-  
1278 BLAS type.

1279 **d\_in1** (IN) The **GrB\_Type** of the left hand argument of the binary operator being created.  
1280 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
1281 GraphBLAS type.

1282 **d\_in2** (IN) The **GrB\_Type** of the right hand argument of the binary operator being cre-  
1283 ated. Should be one of the predefined GraphBLAS types in Table 3.2, or a user-  
1284 defined GraphBLAS type.

## 1285 **Return Values**

1286 **GrB\_SUCCESS** operation completed successfully.

1287 **GrB\_PANIC** unknown internal error.

1288 **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1289 **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_Type** (for user-defined types) has not been initialized by a  
1290 call to **GrB\_Type\_new**.

1291 **GrB\_NULL\_POINTER** **binary\_op** or **binary\_func** pointer is **NULL**.

## 1292 **Description**

1293 The **BinaryOp\_new** methods creates a new GraphBLAS binary operator

1294  $f_b = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in1}), \mathbf{D}(\mathbf{d\_in2}), \mathbf{binary\_func} \rangle$

1295 and returns a handle to it in **binary\_op**.

1296 The implementation of **binary\_func** must be such that it works even if any of the **d\_out**, **d\_in1**, and  
1297 **d\_in2** arguments are aliased to each other. In other words, for all invocations of the function:

1298 **binary\_func(out, in1, in2);**

1299 the value of **out** must be the same as if the following code was executed:

```
1300 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));  
1301 D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));  
1302 memcpy(tmp1, in1, sizeof(D(d_in1)));  
1303 memcpy(tmp2, in2, sizeof(D(d_in2)));  
1304 binary_func(out, tmp1, tmp2);  
1305 free(tmp2);  
1306 free(tmp1);
```

1307 It is not an error to call this method more than once on the same variable; however, the handle to  
1308 the previously created object will be overwritten.

#### 1309 4.2.2.4 Monoid\_new: Construct a new GraphBLAS monoid

1310 Creates a new monoid with specified binary operator and identity value.

### 1311 C Syntax

```
1312         GrB_Info GrB_Monoid_new(GrB_Monoid    *monoid,  
1313                                GrB_BinaryOp    binary_op,  
1314                                <type>         identity);
```

### 1315 Parameters

1316 monoid (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1317 monoid object.

1318 binary\_op (IN) An existing GraphBLAS associative binary operator whose input and output  
1319 types are the same.

1320 identity (IN) The value of the identity element of the monoid. Must be the same type as  
1321 the type used by the **binary\_op** operator.

### 1322 Return Values

1323 GrB\_SUCCESS operation completed successfully.

1324 GrB\_PANIC unknown internal error.

1325 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

1326 GrB\_UNINITIALIZED\_OBJECT the GrB\_BinaryOp (for user-defined operators) has not been initial-  
1327 ized by a call to GrB\_BinaryOp\_new.

1328 GrB\_NULL\_POINTER monoid pointer is NULL.

1329 GrB\_DOMAIN\_MISMATCH all three argument types of the binary operator and the type of the  
1330 identity value are not the same.

### 1331 Description

1332 The **Monoid\_new** method creates a new monoid  $M = \langle \mathbf{D}(\text{binary\_op}), \text{binary\_op}, \text{identity} \rangle$  and re-  
1333 turns a handle to it in **monoid**.

1334 If `binary_op` is not associative, the results of GraphBLAS operations that require associativity of  
1335 this monoid will be undefined.

1336 It is not an error to call this method more than once on the same variable; however, the handle to  
1337 the previously created object will be overwritten.

#### 1338 4.2.2.5 Semiring\_new: Construct a new GraphBLAS semiring

1339 Creates a new semiring with specified domain, operators, and elements.

#### 1340 C Syntax

```
1341         GrB_Info GrB_Semiring_new(GrB_Semiring *semiring,  
1342                                 GrB_Monoid    add_op,  
1343                                 GrB_BinaryOp   mul_op);
```

#### 1344 Parameters

1345 `semiring` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1346 `semiring`.

1347 `add_op` (IN) An existing GraphBLAS commutative monoid that specifies the addition op-  
1348 erator and its identity.

1349 `mul_op` (IN) An existing GraphBLAS binary operator that specifies the semiring's multi-  
1350 plication operator. In addition, `mul_op`'s output domain,  $\mathbf{D}_{out}(\text{mul\_op})$ , must be  
1351 the same as the `add_op`'s domain  $\mathbf{D}(\text{add\_op})$ .

#### 1352 Return Values

1353 `GrB_SUCCESS` operation completed successfully.

1354 `GrB_PANIC` unknown internal error.

1355 `GrB_OUT_OF_MEMORY` not enough memory available for this method to complete.

1356 `GrB_UNINITIALIZED_OBJECT` the `add_op` (for user-define monoids) object has not been initialized  
1357 with a call to `GrB_Monoid_new` or the `mul_op` (for user-defined  
1358 operators) object has not been not been initialized by a call to  
1359 `GrB_BinaryOp_new`.

1360 `GrB_NULL_POINTER` `semiring` pointer is NULL.

1361 `GrB_DOMAIN_MISMATCH` the output domain of `mul_op` does not match the domain of the  
1362 `add_op` monoid.

## 1363 Description

1364 The `Semiring_new` method creates a new semiring:

1365  $S = \langle \mathbf{D}_{out}(\text{mul\_op}), \mathbf{D}_{in_1}(\text{mul\_op}), \mathbf{D}_{in_2}(\text{mul\_op}), \text{add\_op}, \text{mul\_op}, \mathbf{0}(\text{add\_op}) \rangle$

1366 and returns a handle to it in `semiring`. Note that  $\mathbf{D}_{out}(\text{mul\_op})$  must be the same as  $\mathbf{D}(\text{add\_op})$ .

1367 If `add_op` is not commutative, then GraphBLAS operations using this semiring will be undefined.

1368 It is not an error to call this method more than once on the same variable; however, the handle to  
1369 the previously created object will be overwritten.

## 1370 4.2.2.6 IndexUnaryOp\_new: Construct a new GraphBLAS index unary operator

1371 Initializes a new GraphBLAS index unary operator with a specified user-defined function and its  
1372 types (domains).

## 1373 C Syntax

```
1374 GrB_Info GrB_IndexUnaryOp_new(GrB_IndexUnaryOp *index_unary_op,  
1375                               void (*index_unary_func)(void*,  
1376                                                         const void*,  
1377                                                         GrB_Index,  
1378                                                         GrB_Index,  
1379                                                         const void*),  
1380                               GrB_Type d_out,  
1381                               GrB_Type d_in1,  
1382                               GrB_Type d_in2);
```

## 1383 Parameters

1384 `index_unary_op` (INOUT) On successful return, contains a handle to the newly created Graph-  
1385 BLAS index unary operator object.

1386 `index_unary_func` (IN) A pointer to a user-defined function that takes input parameters of types  
1387 `d_in1`, `GrB_Index`, `GrB_Index` and `d_in2` and returns a value of type `d_out`. Ex-  
1388 cept for the `GrB_Index` parameters, all are passed as `void` pointers. Specifically  
1389 the signature of the function is expected to be of the form:

```
1390 void func(void *out,  
1391           const void *in1,  
1392           GrB_Index row_index,  
1393           GrB_Index col_index,  
1394           const void *in2);  
1395
```

1396 **d\_out** (IN) The **GrB\_Type** of the return value of the index unary operator being created.  
 1397 Should be one of the predefined GraphBLAS types in Table 3.2, or a user-defined  
 1398 GraphBLAS type.

1399 **d\_in1** (IN) The **GrB\_Type** of the first input argument of the index unary operator being  
 1400 created and corresponds to the stored values of the **GrB\_Vector** or **GrB\_Matrix**  
 1401 being operated on. Should be one of the predefined GraphBLAS types in Ta-  
 1402 ble 3.2, or a user-defined GraphBLAS type.

1403 **d\_in2** (IN) The **GrB\_Type** of the last input argument of the index unary operator be-  
 1404 ing created and corresponds to a scalar provided by the GraphBLAS operation  
 1405 that uses this operator. Should be one of the predefined GraphBLAS types in  
 1406 Table 3.2, or a user-defined GraphBLAS type.

## 1407 **Return Values**

1408 **GrB\_SUCCESS** operation completed successfully.

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

1410 **GrB\_OUT\_OF\_MEMORY** not enough memory available for operation.

1411 **GrB\_UNINITIALIZED\_OBJECT** the **GrB\_Type** (for user-defined types) has not been initialized by a  
 1412 call to **GrB\_Type\_new**.

1413 **GrB\_NULL\_POINTER** **index\_unary\_op** or **index\_unary\_func** pointer is **NULL**.

## 1414 **Description**

1415 The **IndexUnaryOp\_new** methods creates a new GraphBLAS index unary operator

1416  $f_i = \langle \mathbf{D}(\mathbf{d\_out}), \mathbf{D}(\mathbf{d\_in1}), \mathbf{D}(\mathbf{GrB\_Index}), \mathbf{D}(\mathbf{GrB\_Index}), \mathbf{D}(\mathbf{d\_in2}), \mathbf{index\_unary\_func} \rangle$

1417 and returns a handle to it in **index\_unary\_op**.

1418 The implementation of **index\_unary\_func** must be such that it works even if any of the **d\_out**,  
 1419 **d\_in1**, and **d\_in2** arguments are aliased to each other. In other words, for all invocations of the  
 1420 function:

1421 **index\_unary\_func**(**out**, **in1**, **row\_index**, **col\_index**, **n**, **in2**);

1422 the value of **out** must be the same as if the following code was executed (shown here for matrices):

```
1423 GrB_Index row_index = ...;
1424 GrB_Index col_index = ...;
1425 D(d_in1) *tmp1 = malloc(sizeof(D(d_in1)));
```

```

1426     D(d_in2) *tmp2 = malloc(sizeof(D(d_in2)));
1427     memcpy(tmp1,in1,sizeof(D(d_in1)));
1428     memcpy(tmp2,in2,sizeof(D(d_in2)));
1429     index_unary_func(out,tmp1,row_index,col_index,tmp2);
1430     free(tmp2);
1431     free(tmp1);

```

1432 It is not an error to call this method more than once on the same variable; however, the handle to  
1433 the previously created object will be overwritten.

### 1434 4.2.3 Scalar methods

#### 1435 4.2.3.1 Scalar\_new: Construct a new scalar

1436 Creates a new empty scalar with specified domain.

#### 1437 C Syntax

```

1438     GrB_Info GrB_Scalar_new(GrB_Scalar *s,
1439                             GrB_Type    d);

```

#### 1440 Parameters

1441 **s** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1442 scalar.

1443 **d** (IN) The type corresponding to the domain of the scalar being created. Can be  
1444 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
1445 GraphBLAS type.

#### 1446 Return Values

1447 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1448 blocking mode, this indicates that the API checks for the input  
1449 arguments passed successfully. Either way, output scalar **s** is ready  
1450 to be used in the next method of the sequence.

1451 **GrB\_PANIC** Unknown internal error.

1452 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1453 GraphBLAS objects (input or output) is in an invalid state caused  
1454 by a previous execution error. Call **GrB\_error()** to access any error  
1455 messages generated by the implementation.

1456        GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1457 GrB\_UNINITIALIZED\_OBJECT The GrB\_Type object has not been initialized by a call to GrB\_Type\_new  
1458                                (needed for user-defined types).

1459        GrB\_NULL\_POINTER The s pointer is NULL.

## 1460 Description

1461 Creates a new GraphBLAS scalar  $s$  of domain  $\mathbf{D}(d)$  and empty  $\mathbf{L}(s)$ . The method returns a handle  
1462 to the new scalar in  $s$ .

1463 It is not an error to call this method more than once on the same variable; however, the handle to  
1464 the previously created object will be overwritten.

### 1465 4.2.3.2 Scalar\_dup: Construct a copy of a GraphBLAS scalar

1466 Creates a new scalar with the same domain and contents as another scalar.

## 1467 C Syntax

```
1468            GrB_Info GrB_Scalar_dup(GrB_Scalar            *t,  
1469                                        const GrB_Scalar    s);
```

## 1470 Parameters

1471        t (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1472        scalar.

1473        s (IN) The GraphBLAS scalar to be duplicated.

## 1474 Return Values

1475        GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1476        blocking mode, this indicates that the API checks for the input  
1477        arguments passed successfully. Either way, output scalar  $t$  is ready  
1478        to be used in the next method of the sequence.

1479        GrB\_PANIC Unknown internal error.

1480        GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1481        GraphBLAS objects (input or output) is in an invalid state caused  
1482        by a previous execution error. Call GrB\_error() to access any error  
1483        messages generated by the implementation.

1484        GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1485 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to  
 1486        Scalar\_new or Scalar\_dup.

1487        GrB\_NULL\_POINTER The *t* pointer is NULL.

## 1488 Description

1489 Creates a new scalar *t* of domain  $\mathbf{D}(\mathbf{s})$  and contents  $\mathbf{L}(\mathbf{s})$ . The method returns a handle to the new  
 1490 scalar in *t*.

1491 It is not an error to call this method more than once with the same output variable; however, the  
 1492 handle to the previously created object will be overwritten.

### 1493 4.2.3.3 Scalar\_clear: Clear/remove a stored value from a scalar

1494 Removes the stored value from a scalar.

## 1495 C Syntax

1496        GrB\_Info GrB\_Scalar\_clear(GrB\_Scalar s);

## 1497 Parameters

1498        *s* (INOUT) An existing GraphBLAS scalar to clear.

## 1499 Return Values

1500        GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 1501        blocking mode, this indicates that the API checks for the input  
 1502        arguments passed successfully. Either way, output scalar *s* is ready  
 1503        to be used in the next method of the sequence.

1504        GrB\_PANIC Unknown internal error.

1505        GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1506        GraphBLAS objects (input or output) is in an invalid state caused  
 1507        by a previous execution error. Call GrB\_error() to access any error  
 1508        messages generated by the implementation.

1509        GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1510 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, *s*, has not been initialized by a call to  
 1511        Scalar\_new or Scalar\_dup.



## 1512 Description

1513 Removes the stored value from an existing scalar. After the call, **L(s)** is empty. The size of the  
1514 scalar does not change.

### 1515 4.2.3.4 Scalar\_nvals: Number of stored elements in a scalar

1516 Retrieve the number of stored elements in a scalar (either zero or one).

## 1517 C Syntax

```
1518         GrB_Info GrB_Scalar_nvals(GrB_Index      *nvals,  
1519                                   const GrB_Scalar s);
```

## 1520 Parameters

1521 nvals (OUT) On successful return, this is set to the number of stored elements in the  
1522 scalar (zero or one).

1523 s (IN) An existing GraphBLAS scalar being queried.

## 1524 Return Values

1525 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1526 cessfully and the value of nvals has been set.

1527 GrB\_PANIC Unknown internal error.

1528 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1529 GraphBLAS objects (input or output) is in an invalid state caused  
1530 by a previous execution error. Call GrB\_error() to access any error  
1531 messages generated by the implementation.

1532 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1533 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, s, has not been initialized by a call to  
1534 Scalar\_new or Scalar\_dup.

1535 GrB\_NULL\_POINTER The nvals pointer is NULL.

## 1536 Description

1537 Return **nvals(s)** in nvals. This is the number of stored elements in scalar s, which is the size of  
1538 **L(s)**, and can only be either zero or one (see Section 3.5.1).

### 1539 4.2.3.5 Scalar\_setElement: Set the single element in a scalar

1540 Set the single element of a scalar to a given value.

#### 1541 C Syntax

```
1542         GrB_Info GrB_Scalar_setElement(GrB_Scalar    s,  
1543                                         <type>      val);
```

#### 1544 Parameters

1545 s (INOUT) An existing GraphBLAS scalar for which the element is to be assigned.

1546 val (IN) Scalar value to assign. The type must be compatible with the domain of s.

#### 1547 Return Values

1548 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1549 blocking mode, this indicates that the compatibility tests on in-  
1550 dex/dimensions and domains for the input arguments passed suc-  
1551 cessfully. Either way, the output scalar s is ready to be used in the  
1552 next method of the sequence.

1553 GrB\_PANIC Unknown internal error.

1554 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1555 GraphBLAS objects (input or output) is in an invalid state caused  
1556 by a previous execution error. Call GrB\_error() to access any error  
1557 messages generated by the implementation.

1558 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1559 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, s, has not been initialized by a call to  
1560 Scalar\_new or Scalar\_dup.

1561 GrB\_DOMAIN\_MISMATCH The domains of s and val are incompatible.

#### 1562 Description

1563 First, val and output GraphBLAS scalar are tested for domain compatibility as follows: **D**(val) must  
1564 be compatible with **D**(s). Two domains are compatible with each other if values from one domain  
1565 can be cast to values in the other domain as per the rules of the C language. In particular, domains  
1566 from Table 3.2 are all compatible with each other. A domain from a user-defined type is only com-  
1567 patible with itself. If any compatibility rule above is violated, execution of GrB\_Scalar\_setElement  
1568 ends and the domain mismatch error listed above is returned.

1569 We are now ready to carry out the assignment `val`; that is:

1570  $s(0) = \text{val}$

1571 If `s` already had a stored value, it will be overwritten; otherwise, the new value is stored in `s`.

1572 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new contents  
1573 of `s` is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method exits with  
1574 return value `GrB_SUCCESS` and the new content of scalar `s` is as defined above but may not be  
1575 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 1576 **4.2.3.6 Scalar\_extractElement: Extract a single element from a scalar.**

1577 Assign a non-opaque scalar with the value of the element stored in a GraphBLAS scalar.

### 1578 **C Syntax**

```
1579     GrB_Info GrB_Scalar_extractElement(<type>          *val,  
1580                                     const GrB_Scalar s);
```

### 1581 **Parameters**

1582 `val` (INOUT) Pointer to a non-opaque scalar of type that is compatible with the domain  
1583 of scalar `s`. On successful return, `val` holds the result of the operation, and any  
1584 previous value in `val` is overwritten.

1585 `s` (IN) The GraphBLAS scalar from which an element is extracted.

### 1586 **Return Values**

1587 `GrB_SUCCESS` In blocking or non-blocking mode, the operation completed suc-  
1588 cessfully. This indicates that the compatibility tests on dimensions  
1589 and domains for the input arguments passed successfully, and the  
1590 output scalar, `val`, has been computed and is ready to be used in  
1591 the next method of the sequence.

1592 `GrB_PANIC` Unknown internal error.

1593 `GrB_INVALID_OBJECT` This is returned in any execution mode whenever one of the opaque  
1594 GraphBLAS objects (input or output) is in an invalid state caused  
1595 by a previous execution error. Call `GrB_error()` to access any error  
1596 messages generated by the implementation.

1597 `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

1598 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS scalar, `s`, has not been initialized by a call to  
1599 `Scalar_new` or `Scalar_dup`.

1600 GrB\_NULL\_POINTER `val` pointer is NULL.

1601 GrB\_DOMAIN\_MISMATCH The domains of the scalar or scalar are incompatible.

1602 GrB\_NO\_VALUE There is no stored value in the scalar.

## 1603 Description

1604 First, `val` and input GraphBLAS scalar are tested for domain compatibility as follows: **D(val)**  
1605 must be compatible with **D(s)**. Two domains are compatible with each other if values from  
1606 one domain can be cast to values in the other domain as per the rules of the C language. In  
1607 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
1608 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
1609 `GrB_Scalar_extractElement` ends and the domain mismatch error listed above is returned.

1610 Then, if no value is currently stored in the GraphBLAS scalar, the method returns `GrB_NO_VALUE`  
1611 and `val` remains unchanged.

1612 Finally the extract into the output argument, `val` can be performed; that is:

1613 
$$\text{val} = \text{s}(0)$$

1614 In both `GrB_BLOCKING` mode `GrB_NONBLOCKING` mode if the method exits with return value  
1615 `GrB_SUCCESS`, the new contents of `val` are as defined above.

## 1616 4.2.4 Vector methods

### 1617 4.2.4.1 Vector\_new: Construct new vector

1618 Creates a new vector with specified domain and size.

## 1619 C Syntax

```
1620 GrB_Info GrB_Vector_new(GrB_Vector *v,  
1621                          GrB_Type   d,  
1622                          GrB_Index  nsize);
```

## 1623 Parameters

1624 `v` (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1625 vector.

1626                    **d** (IN) The type corresponding to the domain of the vector being created. Can be  
 1627                    one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
 1628                    GraphBLAS type.

1629                    **nsz** (IN) The size of the vector being created.

## 1630 **Return Values**

1631                    **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 1632                    blocking mode, this indicates that the API checks for the input  
 1633                    arguments passed successfully. Either way, output vector **v** is ready  
 1634                    to be used in the next method of the sequence.

1635                    **GrB\_PANIC** Unknown internal error.

1636                    **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 1637                    GraphBLAS objects (input or output) is in an invalid state caused  
 1638                    by a previous execution error. Call **GrB\_error()** to access any error  
 1639                    messages generated by the implementation.

1640                    **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1641 **GrB\_UNINITIALIZED\_OBJECT** The **GrB\_Type** object has not been initialized by a call to **GrB\_Type\_new**  
 1642                    (needed for user-defined types).

1643                    **GrB\_NULL\_POINTER** The **v** pointer is **NULL**.

1644                    **GrB\_INVALID\_VALUE** **nsz** is zero or outside the range of the type **GrB\_Index**.

## 1645 **Description**

1646                    Creates a new vector **v** of domain **D(d)**, size **nsz**, and empty **L(v)**. The method returns a handle  
 1647                    to the new vector in **v**.

1648                    It is not an error to call this method more than once on the same variable; however, the handle to  
 1649                    the previously created object will be overwritten.

### 1650 **4.2.4.2 Vector\_dup: Construct a copy of a GraphBLAS vector**

1651                    Creates a new vector with the same domain, size, and contents as another vector.

## 1652 **C Syntax**

```
1653                    GrB_Info GrB_Vector_dup(GrB_Vector                *w,  
1654                                                   const GrB_Vector    u);
```

## 1655 Parameters

1656            **w** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
1657            vector.

1658            **u** (IN) The GraphBLAS vector to be duplicated.

## 1659 Return Values

1660            **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1661            blocking mode, this indicates that the API checks for the input  
1662            arguments passed successfully. Either way, output vector **w** is ready  
1663            to be used in the next method of the sequence.

1664            **GrB\_PANIC** Unknown internal error.

1665            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1666            GraphBLAS objects (input or output) is in an invalid state caused  
1667            by a previous execution error. Call **GrB\_error()** to access any error  
1668            messages generated by the implementation.

1669            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1670            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **u**, has not been initialized by a call to  
1671            **Vector\_new** or **Vector\_dup**.

1672            **GrB\_NULL\_POINTER** The **w** pointer is **NULL**.

## 1673 Description

1674            Creates a new vector **w** of domain **D(u)**, size **size(u)**, and contents **L(u)**. The method returns a  
1675            handle to the new vector in **w**.

1676            It is not an error to call this method more than once on the same variable; however, the handle to  
1677            the previously created object will be overwritten.

### 1678 4.2.4.3 Vector\_resize: Resize a vector

1679            Changes the size of an existing vector.

## 1680 C Syntax

```
1681            GrB_Info GrB_Vector_resize(GrB_Vector w,  
1682                                        GrB_Index nsize);
```

## 1683 Parameters

1684            **w** (INOUT) An existing Vector object that is being resized.  
1685            **nsiz**e (IN) The new size of the vector. It can be smaller or larger than the current size.

## 1686 Return Values

1687            **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
1688            blocking mode, this indicates that the API checks for the input  
1689            arguments passed successfully. Either way, output vector **w** is ready  
1690            to be used in the next method of the sequence.

1691            **GrB\_PANIC** Unknown internal error.

1692            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1693            GraphBLAS objects (input or output) is in an invalid state caused  
1694            by a previous execution error. Call **GrB\_error()** to access any error  
1695            messages generated by the implementation.

1696            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

1697            **GrB\_NULL\_POINTER** The **w** pointer is **NULL**.

1698            **GrB\_INVALID\_VALUE** **nsiz**e is zero or outside the range of the type **GrB\_Index**.

## 1699 Description

1700 Changes the size of **w** to **nsiz**e. The domain **D(w)** of vector **w** remains the same. The contents **L(w)**  
1701 are modified as described below.

1702 Let  $w = \langle \mathbf{D}(w), N, \mathbf{L}(w) \rangle$  when the method is called. When the method returns,  $w = \langle \mathbf{D}(w), \text{nsiz}, \mathbf{L}'(w) \rangle$   
1703 where  $\mathbf{L}'(w) = \{(i, w_i) : (i, w_i) \in \mathbf{L}(w) \wedge (i < \text{nsiz})\}$ . That is, all elements of **w** with index greater  
1704 than or equal to the new vector size (**nsiz**) are dropped.

### 1705 4.2.4.4 Vector\_clear: Clear a vector

1706 Removes all the elements (tuples) from a vector.

## 1707 C Syntax

1708            **GrB\_Info** **GrB\_Vector\_clear**(**GrB\_Vector** v);

## 1709 Parameters

1710            **v** (INOUT) An existing GraphBLAS vector to clear.

## 1711 Return Values

1712                   GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1713                   blocking mode, this indicates that the API checks for the input  
1714                   arguments passed successfully. Either way, output vector  $v$  is ready  
1715                   to be used in the next method of the sequence.

1716                   GrB\_PANIC Unknown internal error.

1717                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1718                   GraphBLAS objects (input or output) is in an invalid state caused  
1719                   by a previous execution error. Call `GrB_error()` to access any error  
1720                   messages generated by the implementation.

1721                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1722 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector,  $v$ , has not been initialized by a call to  
1723                   Vector\_new or Vector\_dup.

## 1724 Description

1725 Removes all elements (tuples) from an existing vector. After the call to `GrB_Vector_clear(v)`,  
1726  $L(v) = \emptyset$ . The size of the vector does not change.

### 1727 4.2.4.5 Vector\_size: Size of a vector

1728 Retrieve the size of a vector.

## 1729 C Syntax

```
1730                   GrB_Info GrB_Vector_size(GrB_Index                *nsiz,
1731                                               const GrB_Vector    v);
```

## 1732 Parameters

1733                   nsiz (OUT) On successful return, is set to the size of the vector.

1734                   v (IN) An existing GraphBLAS vector being queried.

## 1735 Return Values

1736                   GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
1737                   cessfully and the value of `nsiz` has been set.

1738                   GrB\_PANIC Unknown internal error.



1739       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1740       GraphBLAS objects (input or output) is in an invalid state caused  
 1741       by a previous execution error. Call GrB\_error() to access any error  
 1742       messages generated by the implementation.

1743 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, *v*, has not been initialized by a call to  
 1744       Vector\_new or Vector\_dup.

1745       GrB\_NULL\_POINTER *nsz* pointer is NULL.

## 1746 Description

1747 Return **size**(*v*) in *nsz*.

### 1748 4.2.4.6 Vector\_nvals: Number of stored elements in a vector

1749 Retrieve the number of stored elements (tuples) in a vector.

## 1750 C Syntax

```
1751      GrB_Info GrB_Vector_nvals(GrB_Index      *nvals,
1752                               const GrB_Vector v);
```

## 1753 Parameters

1754       *nvals* (OUT) On successful return, this is set to the number of stored elements (tuples)  
 1755       in the vector.

1756       *v* (IN) An existing GraphBLAS vector being queried.

## 1757 Return Values

1758       GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 1759       cessfully and the value of **nvals** has been set.

1760       GrB\_PANIC Unknown internal error.

1761       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 1762       GraphBLAS objects (input or output) is in an invalid state caused  
 1763       by a previous execution error. Call GrB\_error() to access any error  
 1764       messages generated by the implementation.

1765       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1766 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector,  $v$ , has not been initialized by a call to  
 1767 Vector\_new or Vector\_dup.

1768 GrB\_NULL\_POINTER The nvals pointer is NULL.

## 1769 Description

1770 Return  $\mathbf{nvals}(v)$  in  $\mathbf{nvals}$ . This is the number of stored elements in vector  $v$ , which is the size of  
 1771  $\mathbf{L}(v)$  (see Section 3.5.2).

## 1772 4.2.4.7 Vector\_build: Store elements from tuples into a vector

## 1773 C Syntax

```
1774      GrB_Info GrB_Vector_build(GrB_Vector      w,
1775                               const GrB_Index  *indices,
1776                               const <type>      *values,
1777                               GrB_Index         n,
1778                               const GrB_BinaryOp dup);
```

## 1779 Parameters

1780  $w$  (INOUT) An existing Vector object to store the result.

1781  $\mathbf{indices}$  (IN) Pointer to an array of indices.

1782  $\mathbf{values}$  (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
 1783 vector  $w$ .

1784  $n$  (IN) The number of entries contained in each array (the same for  $\mathbf{indices}$  and  $\mathbf{values}$ ).

1785  $\mathbf{dup}$  (IN) An associative and commutative binary operator to apply when duplicate  
 1786 values for the same location are present in the input arrays. All three domains of  
 1787  $\mathbf{dup}$  must be the same; hence  $\mathbf{dup} = \langle D_{\mathbf{dup}}, D_{\mathbf{dup}}, D_{\mathbf{dup}}, \oplus \rangle$ . If  $\mathbf{dup}$  is GrB\_NULL,  
 1788 then duplicate locations will result in an error.

## 1789 Return Values

1790 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 1791 blocking mode, this indicates that the API checks for the input  
 1792 arguments passed successfully. Either way, output vector  $w$  is  
 1793 ready to be used in the next method of the sequence.

1794 GrB\_PANIC Unknown internal error.

1795           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
1796           opaque GraphBLAS objects (input or output) is in an invalid  
1797           state caused by a previous execution error. Call GrB\_error() to  
1798           access any error messages generated by the implementation.

1799           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1800           GrB\_UNINITIALIZED\_OBJECT Either w has not been initialized by a call to by GrB\_Vector\_new  
1801           or by GrB\_Vector\_dup, or dup has not been initialized by a call  
1802           to by GrB\_BinaryOp\_new.

1803           GrB\_NULL\_POINTER indices or values pointer is NULL.

1804           GrB\_INDEX\_OUT\_OF\_BOUNDS A value in indices is outside the allowed range for w.

1805           GrB\_DOMAIN\_MISMATCH Either the domains of the GraphBLAS binary operator dup are  
1806           not all the same, or the domains of values and w are incompatible  
1807           with each other or  $D_{dup}$ .

1808           GrB\_OUTPUT\_NOT\_EMPTY Output vector w already contains valid tuples (elements). In  
1809           other words, GrB\_Vector\_nvals(C) returns a positive value.

1810           GrB\_INVALID\_VALUE indices contains a duplicate location and dup is GrB\_NULL.

## 1811 Description

1812 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  
1813 from w in its domain; otherwise,  $\tilde{\mathbf{w}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \emptyset \rangle$ .

1814 Each tuple  $\{\text{indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the output in the form of

$$1815 \quad \tilde{\mathbf{w}}(\text{indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(\mathbf{w})) \text{values}[k] & \text{otherwise.} \end{cases}$$

1816 If multiple values for the same location are present in the input arrays and dup is not GrB\_NULL,  
1817 dup is used to reduce the values before assignment into  $\tilde{\mathbf{w}}$  as follows:

$$1818 \quad \tilde{\mathbf{w}}_i = \bigoplus_{k: \text{indices}[k]=i} (D_{dup}) \text{values}[k],$$

1819 where  $\oplus$  is the dup binary operator. Finally, the resulting  $\tilde{\mathbf{w}}$  is copied into w via typecasting its  
1820 values to  $\mathbf{D}(\mathbf{w})$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

1821 The nonopaque input arrays, indices and values, must be at least as large as n.

1822 It is an error to call this function on an output object with existing elements. In other words,  
1823 GrB\_Vector\_nvals(w) should evaluate to zero prior to calling this function.

1824 After GrB\_Vector\_build returns, it is safe for a programmer to modify or delete the arrays indices  
1825 or values.

#### 1826 4.2.4.8 Vector\_setElement: Set a single element in a vector

1827 Set one element of a vector to a given value.

#### 1828 C Syntax

```
1829         // scalar value
1830         GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1831                                     <type>          val,
1832                                     GrB_Index         index);
1833
1834         // GraphBLAS scalar
1835         GrB_Info GrB_Vector_setElement(GrB_Vector      w,
1836                                     const GrB_Scalar  s,
1837                                     GrB_Index         index);
```

#### 1838 Parameters

1839 w (INOUT) An existing GraphBLAS vector for which an element is to be assigned.

1840 val or s (IN) Scalar assign. Its domain (type) must be compatible with the domain of w.

1841 index (IN) The location of the element to be assigned.

#### 1842 Return Values

1843 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
1844 blocking mode, this indicates that the compatibility tests on in-  
1845 dex/dimensions and domains for the input arguments passed suc-  
1846 cessfully. Either way, the output vector w is ready to be used in  
1847 the next method of the sequence.

1848 GrB\_PANIC Unknown internal error.

1849 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
1850 GraphBLAS objects (input or output) is in an invalid state caused  
1851 by a previous execution error. Call GrB\_error() to access any error  
1852 messages generated by the implementation.

1853 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

1854 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS vector, w, or GraphBLAS scalar, s, has not been  
1855 initialized by a call to a respective constructor.

1856 GrB\_INVALID\_INDEX index specifies a location that is outside the dimensions of w.

1857 GrB\_DOMAIN\_MISMATCH The domains of the vector and the scalar are incompatible.

## 1858 Description

1859 First, the scalar and output vector are tested for domain compatibility as follows: **D(val)** or **D(s)**  
 1860 must be compatible with **D(w)**. Two domains are compatible with each other if values from  
 1861 one domain can be cast to values in the other domain as per the rules of the C language. In  
 1862 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 1863 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 1864 **GrB\_Vector\_setElement** ends and the domain mismatch error listed above is returned.

1865 Then, the **index** parameter is checked for a valid value where the following condition must hold:

$$1866 \quad 0 \leq \text{index} < \text{size}(\mathbf{w})$$

1867 If this condition is violated, execution of **GrB\_Vector\_setElement** ends and the invalid index error  
 1868 listed above is returned.

1869 We are now ready to carry out the assignment; that is:

$$1870 \quad \mathbf{w}(\text{index}) = \begin{cases} \mathbf{L}(\mathbf{s}), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

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

1875 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new contents  
 1876 of **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 **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.

1963 We are now ready to carry out the extract into the output scalar; that is:

$$1964 \qquad \left. \begin{array}{l} \mathbf{L}(\mathbf{s}) \\ \text{val} \end{array} \right\} = \mathbf{u}(\text{index})$$

1965 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  
1966 necessary. If  $\text{index} \notin \text{ind}(\mathbf{u})$ , then one of the follow occurs depending on output scalar type:

- 1967     • The GraphBLAS scalar,  $\mathbf{s}$ , is cleared and `GrB_SUCCESS` is returned.
- 1968     • The non-opaque scalar, `val`, is unchanged, and `GrB_NO_VALUE` is returned.

1969 When using the non-opaque scalar variant (`val`) in both `GrB_BLOCKING` mode `GrB_NONBLOCKING`  
1970 mode, the new contents of `val` are as defined above if the method exits with return value `GrB_SUCCESS`  
1971 or `GrB_NO_VALUE`.

1972 When using the GraphBLAS scalar variant ( $\mathbf{s}$ ) with a `GrB_SUCCESS` return value, the method  
1973 exits and the new contents of  $\mathbf{s}$  is as defined above and fully computed in `GrB_BLOCKING` mode.  
1974 In `GrB_NONBLOCKING` mode, the new contents of  $\mathbf{s}$  is as defined above but may not be fully  
1975 computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 1976 4.2.4.11 Vector\_extractTuples: Extract tuples from a vector

1977 Extract the contents of a GraphBLAS vector into non-opaque data structures.

## 1978 C Syntax

1979     GrB\_Info GrB\_Vector\_extractTuples(GrB\_Index                     \*indices,



```

1980                                     <type>                *values,
1981                                     GrB_Index              *n,
1982                                     const GrB_Vector        v);
1983

```

1984 **indices** (OUT) Pointer to an array of indices that is large enough to hold all of the stored  
1985 values' indices.

1986 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
1987 the stored values whose type is compatible with **D(v)**.

1988 **n** (INOUT) Pointer to a value indicating (on input) the number of elements the  
1989 **values** and **indices** arrays can hold. Upon return, it will contain the number of  
1990 values written to the arrays.

1991 **v** (IN) An existing GraphBLAS vector.

## 1992 Return Values

1993 **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
1994 cessfully. This indicates that the compatibility tests on the input  
1995 argument passed successfully, and the output arrays, **indices** and  
1996 **values**, have been computed.

1997 **GrB\_PANIC** Unknown internal error.

1998 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
1999 GraphBLAS objects (input or output) is in an invalid state caused  
2000 by a previous execution error. Call **GrB\_error()** to access any error  
2001 messages generated by the implementation.

2002 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2003 **GrB\_INSUFFICIENT\_SPACE** Not enough space in **indices** and **values** (as indicated by the **n** pa-  
2004 rameter) to hold all of the tuples that will be extracted.

2005 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to  
2006 **Vector\_new** or **Vector\_dup**.

2007 **GrB\_NULL\_POINTER** **indices**, **values**, or **n** pointer is NULL.

2008 **GrB\_DOMAIN\_MISMATCH** The domains of the **v** vector or **values** array are incompatible with  
2009 one another.

## 2010 Description

2011 This method will extract all the tuples from the GraphBLAS vector **v**. The values associated  
2012 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.

## 2041 Return Values

- 2042           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2043           blocking mode, this indicates that the API checks for the input ar-  
2044           guments passed successfully. Either way, output matrix **A** is ready  
2045           to be used in the next method of the sequence.
- 2046           GrB\_PANIC Unknown internal error.
- 2047           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2048           GraphBLAS objects (input or output) is in an invalid state caused  
2049           by a previous execution error. Call `GrB_error()` to access any error  
2050           messages generated by the implementation.
- 2051           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.
- 2052           GrB\_UNINITIALIZED\_OBJECT The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`  
2053           (needed for user-defined types).
- 2054           GrB\_NULL\_POINTER The **A** pointer is NULL.
- 2055           GrB\_INVALID\_VALUE `nrows` or `ncols` is zero or outside the range of the type `GrB_Index`.

## 2056 Description

- 2057 Creates a new matrix **A** of domain **D**(**d**), size `nrows`  $\times$  `ncols`, and empty **L**(**A**). The method returns  
2058 a handle to the new matrix in **A**.
- 2059 It is not an error to call this method more than once on the same variable; however, the handle to  
2060 the previously created object will be overwritten.

### 2061 4.2.5.2 Matrix\_dup: Construct a copy of a GraphBLAS matrix

- 2062 Creates a new matrix with the same domain, dimensions, and contents as another matrix.

## 2063 C Syntax

```
2064           GrB_Info GrB_Matrix_dup(GrB_Matrix        *C,  
2065                                   const GrB_Matrix  A);
```

## 2066 Parameters

- 2067           **C** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2068           matrix.
- 2069           **A** (IN) The GraphBLAS matrix to be duplicated.

## 2070 Return Values

2071           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2072           blocking mode, this indicates that the API checks for the input  
2073           arguments passed successfully. Either way, output matrix **C** is ready  
2074           to be used in the next method of the sequence.

2075           GrB\_PANIC Unknown internal error.

2076           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2077           GraphBLAS objects (input or output) is in an invalid state caused  
2078           by a previous execution error. Call **GrB\_error()** to access any error  
2079           messages generated by the implementation.

2080           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2081           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, **A**, has not been initialized by a call to  
2082           any matrix constructor.

2083           GrB\_NULL\_POINTER The **C** pointer is **NULL**.

## 2084 Description

2085           Creates a new matrix **C** of domain **D(A)**, size **nrows(A) × ncols(A)**, and contents **L(A)**. It returns  
2086           a handle to it in **C**.

2087           It is not an error to call this method more than once on the same variable; however, the handle to  
2088           the previously created object will be overwritten.

### 2089 4.2.5.3 Matrix\_diag: Construct a diagonal GraphBLAS matrix

2090           Creates a new matrix with the same domain and contents as a **GrB\_Vector**, and square dimensions  
2091           appropriate for placing the contents of the vector along the specified diagonal of the matrix.

## 2092 C Syntax

```
2093           GrB_Info GrB_Matrix_diag(GrB_Matrix        *C,  
2094                                    const GrB_Vector   v,  
2095                                    int64_t           k);
```

## 2096 Parameters

2097           **C** (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2098           matrix. The matrix is square with each dimension equal to **size(v) + |k|**.

2099            **v** (IN) The GraphBLAS vector whose contents will be copied to the diagonal of the  
 2100            matrix.

2101            **k** (IN) The diagonal to which the vector is assigned.  $k = 0$  represents the main  
 2102            diagonal,  $k > 0$  is above the main diagonal, and  $k < 0$  is below.

## 2103    **Return Values**

2104            **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 2105            blocking mode, this indicates that the API checks for the input  
 2106            arguments passed successfully. Either way, output matrix **C** is ready  
 2107            to be used in the next method of the sequence.

2108            **GrB\_PANIC** Unknown internal error.

2109            **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 2110            GraphBLAS objects (input or output) is in an invalid state caused  
 2111            by a previous execution error. Call **GrB\_error()** to access any error  
 2112            messages generated by the implementation.

2113            **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

2114            **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS vector, **v**, has not been initialized by a call to  
 2115            **Vector\_new** or **Vector\_dup**.

2116            **GrB\_NULL\_POINTER** The C pointer is NULL.

## 2117    **Description**

2118    Creates a new matrix **C** of domain **D(v)**, size  $(\text{size}(\mathbf{v}) + |k|) \times (\text{size}(\mathbf{v}) + |k|)$ , and contents

$$2119 \qquad \mathbf{L}(\mathbf{C}) = \{(i, i + k, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k \geq 0 \text{ or}$$

$$2120 \qquad \mathbf{L}(\mathbf{C}) = \{(i - k, i, v_i) : (i, v_i) \in \mathbf{L}(\mathbf{v})\} \text{ if } k < 0.$$

2121    It returns a handle to it in **C**. It is not an error to call this method more than once on the same  
 2122    variable; however, the handle to the previously created object will be overwritten.

### 2123    **4.2.5.4    Matrix\_resize: Resize a matrix**

2124    Changes the dimensions of an existing matrix.

## 2125    **C Syntax**

```
2126            GrB_Info GrB_Matrix_resize(GrB_Matrix C,
2127                                            GrB_Index nrows,
2128                                            GrB_Index ncols);
```

## 2129 Parameters

2130           C (INOUT) An existing Matrix object that is being resized.

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

2133           ncols (IN) The new number of columns of the matrix. It can be smaller or larger than  
2134           the current number of columns.

## 2135 Return Values

2136           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2137           blocking mode, this indicates that the API checks for the input  
2138           arguments passed successfully. Either way, output matrix C is ready  
2139           to be used in the next method of the sequence.

2140           GrB\_PANIC Unknown internal error.

2141           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2142           GraphBLAS objects (input or output) is in an invalid state caused  
2143           by a previous execution error. Call GrB\_error() to access any error  
2144           messages generated by the implementation.

2145           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2146           GrB\_NULL\_POINTER The C pointer is NULL.

2147           GrB\_INVALID\_VALUE nrows or ncols is zero or outside the range of the type GrB\_Index.

## 2148 Description

2149 Changes the number of rows and columns of C to nrows and ncols, respectively. The domain  $\mathbf{D}(\mathbf{C})$   
2150 of matrix C remains the same. The contents  $\mathbf{L}(\mathbf{C})$  are modified as described below.

2151 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  
2152 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  
2153 greater than or equal to ncols are dropped.  
2154

### 2155 4.2.5.5 Matrix\_clear: Clear a matrix

2156 Removes all elements (tuples) from a matrix.

## 2157 C Syntax

2158           GrB\_Info GrB\_Matrix\_clear(GrB\_Matrix A);

## 2159 Parameters

2160           A (IN) An existing GraphBLAS matrix to clear.

## 2161 Return Values

2162           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2163                       blocking mode, this indicates that the API checks for the input ar-  
2164                       guments passed successfully. Either way, output matrix A is ready  
2165                       to be used in the next method of the sequence.

2166           GrB\_PANIC Unknown internal error.

2167           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2168                       GraphBLAS objects (input or output) is in an invalid state caused  
2169                       by a previous execution error. Call GrB\_error() to access any error  
2170                       messages generated by the implementation.

2171           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2172           GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2173                       any matrix constructor.

## 2174 Description

2175 Removes all elements (tuples) from an existing matrix. After the call to GrB\_Matrix\_clear(A),  
2176  $\mathbf{L}(\mathbf{A}) = \emptyset$ . The dimensions of the matrix do not change.

### 2177 4.2.5.6 Matrix\_nrows: Number of rows in a matrix

2178 Retrieve the number of rows in a matrix.

## 2179 C Syntax

```
2180           GrB_Info GrB_Matrix_nrows(GrB_Index           *nrows,  
2181                                       const GrB_Matrix  A);
```

## 2182 Parameters

2183           nrows (OUT) On successful return, contains the number of rows in the matrix.

2184           A (IN) An existing GraphBLAS matrix being queried.

## 2185 **Return Values**

2186                   GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2187                   cessfully and the value of `nrows` has been set.

2188                   GrB\_PANIC Unknown internal error.

2189           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2190           GraphBLAS objects (input or output) is in an invalid state caused  
2191           by a previous execution error. Call `GrB_error()` to access any error  
2192           messages generated by the implementation.

2193 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, `A`, has not been initialized by a call to  
2194           any matrix constructor.

2195           GrB\_NULL\_POINTER `nrows` pointer is NULL.

## 2196 **Description**

2197 Return `nrows(A)` in `nrows` (the number of rows).

## 2198 **4.2.5.7 Matrix\_ncols: Number of columns in a matrix**

2199 Retrieve the number of columns in a matrix.

## 2200 **C Syntax**

```
2201           GrB_Info GrB_Matrix_ncols(GrB_Index           *ncols,  
2202                                    const GrB_Matrix A);
```

## 2203 **Parameters**

2204           ncols (OUT) On successful return, contains the number of columns in the matrix.

2205           A (IN) An existing GraphBLAS matrix being queried.

## 2206 **Return Values**

2207                   GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2208                   cessfully and the value of `ncols` has been set.

2209                   GrB\_PANIC Unknown internal error.



2210           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 2211           GraphBLAS objects (input or output) is in an invalid state caused  
 2212           by a previous execution error. Call GrB\_error() to access any error  
 2213           messages generated by the implementation.

2214 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
 2215           any matrix constructor.

2216           GrB\_NULL\_POINTER ncols pointer is NULL.

## 2217 Description

2218 Return **ncols(A)** in **ncols** (the number of columns).

### 2219 4.2.5.8 Matrix\_nvals: Number of stored elements in a matrix

2220 Retrieve the number of stored elements (tuples) in a matrix.

## 2221 C Syntax

```
2222           GrB_Info GrB_Matrix_nvals(GrB_Index           *nvals,  
2223                                    const GrB_Matrix A);
```

## 2224 Parameters

2225           **nvals** (OUT) On successful return, contains the number of stored elements (tuples) in  
 2226           the matrix.

2227           **A** (IN) An existing GraphBLAS matrix being queried.

## 2228 Return Values

2229           GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 2230           cessfully and the value of **nvals** has been set.

2231           GrB\_PANIC Unknown internal error.

2232           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 2233           GraphBLAS objects (input or output) is in an invalid state caused  
 2234           by a previous execution error. Call GrB\_error() to access any error  
 2235           messages generated by the implementation.

2236           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2237 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
 2238 any matrix constructor.

2239 GrB\_NULL\_POINTER The nvals pointer is NULL.

## 2240 Description

2241 Return **nvals(A)** in **nvals**. This is the number of tuples stored in matrix A, which is the size of  
 2242 **L(A)** (see Section 3.5.3).

## 2243 4.2.5.9 Matrix\_build: Store elements from tuples into a matrix

### 2244 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);
```

### 2245 Parameters

2246 C (INOUT) An existing Matrix object to store the result.

2247 row\_indices (IN) Pointer to an array of row indices.

2248 col\_indices (IN) Pointer to an array of column indices.

2249 values (IN) Pointer to an array of scalars of a type that is compatible with the domain of  
 2250 matrix, C.

2251 n (IN) The number of entries contained in each array (the same for row\_indices,  
 2252 col\_indices, and values).

2253 dup (IN) An associative and commutative binary operator to apply when duplicate  
 2254 values for the same location are present in the input arrays. All three domains of  
 2255 dup must be the same; hence  $dup = \langle D_{dup}, D_{dup}, D_{dup}, \oplus \rangle$ . If dup is GrB\_NULL,  
 2256 then duplicate locations will result in an error.

### 2257 Return Values

2258 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 2259 blocking mode, this indicates that the API checks for the input  
 2260 arguments passed successfully. Either way, output matrix C is  
 2261 ready to be used in the next method of the sequence.

2262                   GrB\_PANIC Unknown internal error.

2263           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2264                   opaque GraphBLAS objects (input or output) is in an invalid  
2265                   state caused by a previous execution error. Call GrB\_error() to  
2266                   access any error messages generated by the implementation.

2267           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2268           GrB\_UNINITIALIZED\_OBJECT Either C has not been initialized by a call to any matrix construc-  
2269                   tor, or dup has not been initialized by a call to by GrB\_BinaryOp\_new.

2270           GrB\_NULL\_POINTER row\_indices, col\_indices or values pointer is NULL.

2271   GrB\_INDEX\_OUT\_OF\_BOUNDS A value in row\_indices or col\_indices is outside the allowed range  
2272                   for C.

2273           GrB\_DOMAIN\_MISMATCH Either the domains of the GraphBLAS binary operator dup are  
2274                   not all the same, or the domains of values and C are incompatible  
2275                   with each other or  $D_{dup}$ .

2276           GrB\_OUTPUT\_NOT\_EMPTY Output matrix C already contains valid tuples (elements). In  
2277                   other words, GrB\_Matrix\_nvals(C) returns a positive value.

2278           GrB\_INVALID\_VALUE indices contains a duplicate location and dup is GrB\_NULL.

## 2279 Description

2280 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  
2281 only differs from C in its domain; otherwise,  $\tilde{C} = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \emptyset \rangle$ .

2282 Each tuple  $\{\text{row\_indices}[k], \text{col\_indices}[k], \text{values}[k]\}$ , where  $0 \leq k < n$ , is a contribution to the  
2283 output in the form of

$$2284 \quad \tilde{C}(\text{row\_indices}[k], \text{col\_indices}[k]) = \begin{cases} (D_{dup}) \text{values}[k] & \text{if } \text{dup} \neq \text{GrB\_NULL} \\ (\mathbf{D}(C)) \text{values}[k] & \text{otherwise.} \end{cases}$$

2285 If multiple values for the same location are present in the input arrays and dup is not GrB\_NULL,  
2286 dup is used to reduce the values before assignment into  $\tilde{C}$  as follows:

$$2287 \quad \tilde{C}_{ij} = \bigoplus_{k: \text{row\_indices}[k]=i \wedge \text{col\_indices}[k]=j} (D_{dup}) \text{values}[k],$$

2288 where  $\oplus$  is the dup binary operator. Finally, the resulting  $\tilde{C}$  is copied into C via typecasting its  
2289 values to  $\mathbf{D}(C)$  if necessary. If  $\oplus$  is not associative or not commutative, the result is undefined.

2290 The nonopaque input arrays row\_indices, col\_indices, and values must be at least as large as n.

2291 It is an error to call this function on an output object with existing elements. In other words,  
2292 GrB\_Matrix\_nvals(C) should evaluate to zero prior to calling this function.

2293 After GrB\_Matrix\_build returns, it is safe for a programmer to modify or delete the arrays row\_indices,  
2294 col\_indices, or values.

#### 2295 4.2.5.10 Matrix\_setElement: Set a single element in matrix

2296 Set one element of a matrix to a given value.

#### 2297 C Syntax

```
2298         // scalar value
2299         GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2300                                     <type>           val,
2301                                     GrB_Index          row_index,
2302                                     GrB_Index          col_index);
2303
2304         // GraphBLAS scalar
2305         GrB_Info GrB_Matrix_setElement(GrB_Matrix      C,
2306                                     const GrB_Scalar   s,
2307                                     GrB_Index          row_index,
2308                                     GrB_Index          col_index);
```

#### 2309 Parameters

2310 C (INOUT) An existing GraphBLAS matrix for which an element is to be assigned.

2311 val or s (IN) Scalar to assign. Its domain (type) must be compatible with the domain of  
2312 C.

2313 row\_index (IN) Row index of element to be assigned

2314 col\_index (IN) Column index of element to be assigned

#### 2315 Return Values

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

2321 GrB\_PANIC Unknown internal error.

2322 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2323 GraphBLAS objects (input or output) is in an invalid state caused

2324 by a previous execution error. Call `GrB_error()` to access any error  
 2325 messages generated by the implementation.

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

2327 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or GraphBLAS scalar, **s**, has not been  
 2328 initialized by a call to a respective constructor.

2329 **GrB\_INVALID\_INDEX** `row_index` or `col_index` is outside the allowable range (i.e., not less  
 2330 than `nrows(C)` or `ncols(C)`, respectively).

2331 **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and the scalar are incompatible.

## 2332 Description

2333 First, the scalar and output matrix are tested for domain compatibility as follows: **D(val)** or  
 2334 **D(s)** must be compatible with **D(C)**. Two domains are compatible with each other if values from  
 2335 one domain can be cast to values in the other domain as per the rules of the C language. In  
 2336 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 2337 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 2338 `GrB_Matrix_setElement` ends and the domain mismatch error listed above is returned.

2339 Then, both index parameters are checked for valid values where following conditions must hold:

$$2340 \quad \begin{aligned} 0 &\leq \text{row\_index} < \text{nrows}(\mathbf{C}), \\ 0 &\leq \text{col\_index} < \text{ncols}(\mathbf{C}) \end{aligned}$$

2341 If either of these conditions is violated, execution of `GrB_Matrix_setElement` ends and the invalid  
 2342 index error listed above is returned.

2343 We are now ready to carry out the assignment; that is:

$$2344 \quad \mathbf{C}(\text{row\_index}, \text{col\_index}) = \begin{cases} \mathbf{L}(\mathbf{s}), & \text{GraphBLAS scalar.} \\ \text{val}, & \text{otherwise.} \end{cases}$$

2345 In the case of a transparent scalar or if **L(s)** is not empty, then a value will be stored at the  
 2346 specified location in **C**, overwriting any value that may have been stored there before. In the case  
 2347 of a GraphBLAS scalar and if **L(s)** is empty, then any value stored at the specified location in **C**  
 2348 will be removed.

2349 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new contents  
 2350 of **C** is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method exits with  
 2351 return value **GrB\_SUCCESS** and the new content of vector **C** is as defined above but may not be  
 2352 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

### 2353 4.2.5.11 `Matrix_removeElement`: Remove an element from a matrix

2354 Remove (annihilate) one stored element from a matrix.

## 2355 C Syntax

```
2356         GrB_Info GrB_Matrix_removeElement(GrB_Matrix  C,  
2357                                           GrB_Index    row_index,  
2358                                           GrB_Index    col_index);
```

## 2359 Parameters

2360       C (INOUT) An existing GraphBLAS matrix from which an element is to be removed.

2361       row\_index (IN) Row index of element to be removed

2362       col\_index (IN) Column index of element to be removed

## 2363 Return Values

2364       GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
2365       blocking mode, this indicates that the compatibility tests on in-  
2366       dex/dimensions and domains for the input arguments passed suc-  
2367       cessfully. Either way, the output matrix C is ready to be used in  
2368       the next method of the sequence.

2369       GrB\_PANIC Unknown internal error.

2370       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
2371       GraphBLAS objects (input or output) is in an invalid state caused  
2372       by a previous execution error. Call GrB\_error() to access any error  
2373       messages generated by the implementation.

2374       GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2375       GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, C, has not been initialized by a call to  
2376       any matrix constructor.

2377       GrB\_INVALID\_INDEX row\_index or col\_index is outside the allowable range (i.e., not less  
2378       than **nrows**(C) or **ncols**(C), respectively).

## 2379 Description

2380 First, both index parameters are checked for valid values where following conditions must hold:

$$\begin{aligned} 2381 \quad & 0 \leq \text{row\_index} < \text{nrows}(\text{C}), \\ & 0 \leq \text{col\_index} < \text{ncols}(\text{C}) \end{aligned}$$

2382 If either of these conditions is violated, execution of GrB\_Matrix\_removeElement ends and the  
2383 invalid index error listed above is returned.

2384 We are now ready to carry out the removal of a value that may be stored at the location specified by  
 2385 (row\_index, col\_index). If a value does not exist at the specified location in C, no error is reported  
 2386 and the operation has no effect on the state of C. In either case, the following will be true on return  
 2387 from this method: (row\_index, col\_index)  $\notin$  ind(C)

2388 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new contents  
 2389 of C is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method exits with  
 2390 return value GrB\_SUCCESS and the new content of vector C is as defined above but may not be  
 2391 fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 2392 4.2.5.12 Matrix\_extractElement: Extract a single element from a matrix

2393 Extract one element of a matrix into a scalar.

#### 2394 C Syntax

```
2395         // scalar value
2396         GrB_Info GrB_Matrix_extractElement(<type>          *val,
2397                                         const GrB_Matrix  A,
2398                                         GrB_Index         row_index,
2399                                         GrB_Index         col_index);
2400
2401         // GraphBLAS scalar
2402         GrB_Info GrB_Matrix_extractElement(GrB_Scalar      s,
2403                                         const GrB_Matrix  A,
2404                                         GrB_Index         row_index,
2405                                         GrB_Index         col_index);
2406
```

#### 2407 Parameters

2408 val or s (INOUT) An existing scalar whose domain is compatible with the domain of matrix  
 2409 A. On successful return, this scalar holds the result of the extract. Any previous  
 2410 value stored in val or s is overwritten.

2411 A (IN) The GraphBLAS matrix from which an element is extracted.

2412 row\_index (IN) The row index of location in A to extract.

2413 col\_index (IN) The column index of location in A to extract.

#### 2414 Return Values

2415 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 2416 cessfully. This indicates that the compatibility tests on dimensions

2417 and domains for the input arguments passed successfully, and the  
 2418 output scalar, **val** or **s**, has been computed and is ready to be used  
 2419 in the next method of the sequence.

2420 **GrB\_NO\_VALUE** When using the transparent scalar, **val**, this is returned when there  
 2421 is no stored value at specified location.

2422 **GrB\_PANIC** Unknown internal error.

2423 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 2424 GraphBLAS objects (input or output) is in an invalid state caused  
 2425 by a previous execution error. Call **GrB\_error()** to access any error  
 2426 messages generated by the implementation.

2427 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2428 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, or scalar, **s**, has not been initialized by  
 2429 a call to a corresponding constructor.

2430 **GrB\_NULL\_POINTER** **val** pointer is NULL.

2431 **GrB\_INVALID\_INDEX** **row\_index** or **col\_index** is outside the allowable range (i.e. less than  
 2432 zero or greater than or equal to **nrows(A)** or **ncols(A)**, respec-  
 2433 tively).

2434 **GrB\_DOMAIN\_MISMATCH** The domains of the matrix and scalar are incompatible.

## 2435 Description

2436 First, the scalar and input matrix are tested for domain compatibility as follows: **D(val)** or **D(s)**  
 2437 must be compatible with **D(A)**. Two domains are compatible with each other if values from  
 2438 one domain can be cast to values in the other domain as per the rules of the C language. In  
 2439 particular, domains from Table 3.2 are all compatible with each other. A domain from a user-  
 2440 defined type is only compatible with itself. If any compatibility rule above is violated, execution of  
 2441 **GrB\_Matrix\_extractElement** ends and the domain mismatch error listed above is returned.

2442 Then, both index parameters are checked for valid values where following conditions must hold:

$$2443 \begin{aligned} 0 &\leq \text{row\_index} < \text{nrows}(\mathbf{A}), \\ 0 &\leq \text{col\_index} < \text{ncols}(\mathbf{A}) \end{aligned}$$

2444 If either condition is violated, execution of **GrB\_Matrix\_extractElement** ends and the invalid index  
 2445 error listed above is returned.

2446 We are now ready to carry out the extract into the output scalar; that is,

$$2447 \left. \begin{array}{l} \mathbf{L}(\mathbf{s}) \\ \mathbf{val} \end{array} \right\} = \mathbf{A}(\text{row\_index}, \text{col\_index})$$



2448 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  $\mathbf{val}$   
 2449 with casting as necessary. If  $(\text{row\_index}, \text{col\_index}) \notin \mathbf{ind}(\mathbf{A})$ , then one of the follow occurs  
 2450 depending on output scalar type:

- 2451 • The GraphBLAS scalar,  $\mathbf{s}$ , is cleared and  $\text{GrB\_SUCCESS}$  is returned.
- 2452 • The non-opaque scalar,  $\mathbf{val}$ , is unchanged, and  $\text{GrB\_NO\_VALUE}$  is returned.

2453 When using the non-opaque scalar variant ( $\mathbf{val}$ ) in both  $\text{GrB\_BLOCKING}$  mode  $\text{GrB\_NONBLOCKING}$   
 2454 mode, the new contents of  $\mathbf{val}$  are as defined above if the method exits with return value  $\text{GrB\_SUCCESS}$   
 2455 or  $\text{GrB\_NO\_VALUE}$ .

2456 When using the GraphBLAS scalar variant ( $\mathbf{s}$ ) with a  $\text{GrB\_SUCCESS}$  return value, the method  
 2457 exits and the new contents of  $\mathbf{s}$  is as defined above and fully computed in  $\text{GrB\_BLOCKING}$  mode.  
 2458 In  $\text{GrB\_NONBLOCKING}$  mode, the new contents of  $\mathbf{s}$  is as defined above but may not be fully  
 2459 computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 2460 4.2.5.13 Matrix\_extractTuples: Extract tuples from a matrix

2461 Extract the contents of a GraphBLAS matrix into non-opaque data structures.

#### 2462 C Syntax

```
2463      GrB_Info GrB_Matrix_extractTuples(GrB_Index      *row_indices,
2464                                      GrB_Index      *col_indices,
2465                                      <type>          *values,
2466                                      GrB_Index      *n,
2467                                      const GrB_Matrix A);
```

#### 2468 Parameters

2469 **row\_indices** (OUT) Pointer to an array of row indices that is large enough to hold all of the  
 2470 row indices.

2471 **col\_indices** (OUT) Pointer to an array of column indices that is large enough to hold all of the  
 2472 column indices.

2473 **values** (OUT) Pointer to an array of scalars of a type that is large enough to hold all of  
 2474 the stored values whose type is compatible with  $\mathbf{D}(\mathbf{A})$ .

2475 **n** (INOUT) Pointer to a value indicating (in input) the number of elements the **values**,  
 2476 **row\_indices**, and **col\_indices** arrays can hold. Upon return, it will contain the  
 2477 number of values written to the arrays.

2478 **A** (IN) An existing GraphBLAS matrix.

## 2479 Return Values

2480	<b>GrB_SUCCESS</b>	In blocking or non-blocking mode, the operation completed successfully. This indicates that the compatibility tests on the input
2481		argument passed successfully, and the output arrays, <b>indices</b> and
2482		<b>values</b> , have been computed.
2483		
2484	<b>GrB_PANIC</b>	Unknown internal error.
2485	<b>GrB_INVALID_OBJECT</b>	This is returned in any execution mode whenever one of the opaque
2486		GraphBLAS objects (input or output) is in an invalid state caused
2487		by a previous execution error. Call <b>GrB_error()</b> to access any error
2488		messages generated by the implementation.
2489	<b>GrB_OUT_OF_MEMORY</b>	Not enough memory available for operation.
2490	<b>GrB_INSUFFICIENT_SPACE</b>	Not enough space in <b>row_indices</b> , <b>col_indices</b> , and <b>values</b> (as indi-
2491		cated by the <b>n</b> parameter) to hold all of the tuples that will be
2492		extracted.
2493	<b>GrB_UNINITIALIZED_OBJECT</b>	The GraphBLAS matrix, <b>A</b> , has not been initialized by a call to
2494		any matrix constructor.
2495	<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> .
2496	<b>GrB_DOMAIN_MISMATCH</b>	The domains of the <b>A</b> matrix and <b>values</b> array are incompatible
2497		with one another.

## 2498 Description

2499 This method will extract all the tuples from the GraphBLAS matrix **A**. The values associated with  
2500 those tuples are placed in the **values** array, the column indices are placed in the **col\_indices** array,  
2501 and the row indices are placed in the **row\_indices** array. These output arrays are pre-allocated by  
2502 the user before calling this function such that each output array has enough space to hold at least  
2503 **GrB\_Matrix\_nvals(A)** elements.

2504 Upon return of this function, a pair of  $\{\text{row\_indices}[k], \text{col\_indices}[k]\}$  are unique for every valid  
2505  $k$ , but they are not required to be sorted in any particular order. Each tuple  $(i, j, A_{ij})$  in **A** is  
2506 unzipped and copied into a distinct  $k$ th location in output vectors:

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

2507 where  $0 \leq k < \text{GrB\_Matrix\_nvals}(v)$ . No gaps in output vectors are allowed; that is, if **row\_indices**[ $k$ ],  
2508 **col\_indices**[ $k$ ] and **values**[ $k$ ] exist upon return, so does **row\_indices**[ $j$ ], **col\_indices**[ $j$ ] and **values**[ $j$ ] for  
2509 all  $j$  such that  $0 \leq j < k$ .

2510 Note that if the value in **n** on input is less than the number of values contained in the matrix **A**,  
2511 then a **GrB\_INSUFFICIENT\_SPACE** error is returned since it is undefined which subset of values  
2512 would be extracted.

2513 In both GrB\_BLOCKING mode GrB\_NONBLOCKING mode if the method exits with return value  
2514 GrB\_SUCCESS, the new contents of the arrays row\_indices, col\_indices and values are as defined  
2515 above.

2516 **4.2.5.14 Matrix\_exportHint: Provide a hint as to which storage format might be most**  
2517 **efficient for exporting a matrix**

## 2518 C Syntax

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

## 2519 Parameters

2520 hint (OUT) Pointer to a value of type GrB\_Format.

2521 A (IN) A GraphBLAS matrix object.

## 2522 Return Values

2523 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2524 cessfully and the value of hint has been set.

2525 GrB\_PANIC Unknown internal error.

2526 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2527 opaque GraphBLAS objects (input or output) is in an invalid  
2528 state caused by a previous execution error. Call GrB\_error() to  
2529 access any error messages generated by the implementation.

2530 GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2531 GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2532 any matrix constructor.

2533 GrB\_NULL\_POINTER hint is NULL.

2534 GrB\_NO\_VALUE If the implementation does not have a preferred format, it may  
2535 return the value GrB\_NO\_VALUE.

## 2536 Description

2537 Given a GraphBLAS matrix A, provide a hint as to which format might be most efficient for  
2538 exporting the matrix A. GraphBLAS implementations might return the current storage format of  
2539 the matrix, or the format to which it could most efficiently be exported. However, implementations  
2540 are free to return any value for format defined in Section 3.5.3.1. Note that an implementation is  
2541 free to refuse to provide a format hint, returning GrB\_NO\_VALUE.

2542 **4.2.5.15 Matrix\_exportSize:** Return the array sizes necessary to export a GraphBLAS  
 2543 matrix object

#### 2544 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);

```

#### 2545 Parameters

2546 **n\_indptr** (OUT) Pointer to a value of type GrB\_Index.

2547 **n\_indices** (OUT) Pointer to a value of type GrB\_Index.

2548 **n\_values** (OUT) Pointer to a value of type GrB\_Index.

2549 **format** (IN) a value indicating the format in which the matrix will be exported, as defined  
 2550 in Section 3.5.3.1.

2551 **A** (IN) A GraphBLAS matrix object.

#### 2552 Return Values

2553 **GrB\_SUCCESS** In blocking mode or non-blocking mode, the operation com-  
 2554 pleted successfully. This indicates that the API checks for the  
 2555 input arguments passed successfully, and the number of elements  
 2556 necessary for the export buffers have been written to **n\_indptr**,  
 2557 **n\_indices**, and **n\_values**, respectively.

2558 **GrB\_PANIC** Unknown internal error.

2559 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 2560 opaque GraphBLAS objects (input or output) is in an invalid  
 2561 state caused by a previous execution error. Call **GrB\_error()** to  
 2562 access any error messages generated by the implementation.

2563 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2564 **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS Matrix, **A**, has not been initialized by a call to  
 2565 any matrix constructor.

2566 **GrB\_NULL\_POINTER** **n\_indptr**, **n\_indices**, or **n\_values** is NULL.

2567 **Description**

2568 Given a matrix **A**, returns the required capacities of arrays **values**, **indptr**, and **indices** necessary to  
2569 export the matrix in the format specified by **format**. The output values **n\_values**, **n\_indptr**, and  
2570 **indices** will contain the corresponding sizes of the arrays (in number of elements) that must be  
2571 allocated to hold the exported matrix. The argument **format** can be chosen arbitrarily by the user  
2572 as one of the values defined in Section 3.5.3.1.

2573 **4.2.5.16 Matrix\_export: Export a GraphBLAS matrix to a pre-defined format**

2574 **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);
```

2575 **Parameters**

2576 **indptr** (INOUT) Pointer to an array that will hold row or column offsets, or row in-  
2577 dices, depending on the value of **format**. It must be large enough to hold at  
2578 least **n\_indptr** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2579 **GrB\_Matrix\_exportSize()** method.

2580 **indices** (INOUT) Pointer to an array that will hold row or column indices of the elements  
2581 in **values**, depending on the value of **format**. It must be large enough to hold at  
2582 least **n\_indices** elements of type **GrB\_Index**, where **n\_indices** was returned from  
2583 **GrB\_Matrix\_exportSize()** method.

2584 **values** (INOUT) Pointer to an array that will hold stored values. The type of ele-  
2585 ment must match the type of the values stored in **A**. It must be large enough  
2586 to hold at least **n\_values** elements of that type, where **n\_values** was returned from  
2587 **GrB\_Matrix\_exportSize**.

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

2591 **n\_indices** (INOUT) Pointer to a value indicating (on input) the number of elements the **indices**  
2592 array can hold. Upon return, it will contain the number of elements written to the  
2593 array.

2594        **n\_values** (INOUT) Pointer to a value indicating (on input) the number of elements the **values**  
2595        array can hold. Upon return, it will contain the number of elements written to the  
2596        array.

2597        **format** (IN) a value indicating the format in which the matrix will be exported, as defined  
2598        in Section 3.5.3.1.

2599        **A** (IN) A GraphBLAS matrix object.

## 2600 **Return Values**

2601        **GrB\_SUCCESS** In blocking or non-blocking mode, the operation completed suc-  
2602        cessfully. This indicates that the compatibility tests on the input  
2603        argument passed successfully, and the output arrays, **indptr**, **in-**  
2604        **indices** and **values**, have been computed.

2605        **GrB\_PANIC** Unknown internal error.

2606        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
2607        opaque GraphBLAS objects (input or output) is in an invalid  
2608        state caused by a previous execution error. Call **GrB\_error()** to  
2609        access any error messages generated by the implementation.

2610        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

2611        **GrB\_INSUFFICIENT\_SPACE** Not enough space in **indptr**, **indices**, and/or **values** (as indicated  
2612        by the corresponding **n\_\*** parameter) to hold all of the corre-  
2613        sponding elements that will be extacted.

2614        **GrB\_UNINITIALIZED\_OBJECT** The GraphBLAS matrix, **A**, has not been initialized by a call to  
2615        any matrix constructor.

2616        **GrB\_NULL\_POINTER** **indptr**, **indices**, **values** **n\_indptr**, **n\_indices**, **n\_values** pointer is  
2617        **NULL**.

2618        **GrB\_DOMAIN\_MISMATCH** The domain of **A** does not match with the type of **values**.

## 2619 **Description**

2620        Given a matrix **A**, this method exports the contents of the matrix into one of the pre-defined  
2621        **GrB\_Format** formats from Section 3.5.3.1. The user-allocated arrays pointed to by **indptr**, **indices**,  
2622        and **values** must be at least large enough to hold the corresponding number of elements returned  
2623        by calling **GrB\_Matrix\_exportSize**. The value of **format** can be chosen arbitrarily, but a call to  
2624        **GrB\_Matrix\_exportHint** may suggest a format that results in the most efficient export. Details  
2625        of the contents of **indptr**, **indices**, and **values** corresponding to each supported format is given in  
2626        Appendix B.

#### 2627 4.2.5.17 Matrix\_import: Import a matrix into a GraphBLAS object

#### 2628 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);
```

#### 2629 Parameters

2630 A (INOUT) On a successful return, contains a handle to the newly created Graph-  
2631 BLAS matrix.

2632 d (IN) The type corresponding to the domain of the matrix being created. Can be  
2633 one of the predefined GraphBLAS types in Table 3.2, or an existing user-defined  
2634 GraphBLAS type.

2635 nrows (IN) Integer value holding the number of rows in the matrix.

2636 ncols (IN) Integer value holding the number of columns in the matrix.

2637 indptr (IN) Pointer to an array of row or column offsets, or row indices, depending on the  
2638 value of format.

2639 indices (IN) Pointer to an array row or column indices of the elements in values, depending  
2640 on the value of format.

2641 values (IN) Pointer to an array of values. Type must match the type of d.

2642 n\_indptr (IN) Integer value holding the number of elements in the array pointed to by indptr.

2643 n\_indices (IN) Integer value holding the number of elements in the array pointed to by indices.

2644 n\_values (IN) Integer value holding the number of elements in the array pointed to by values.

2645 format (IN) a value indicating the format of the matrix being imported, as defined in  
2646 Section 3.5.3.1.

## 2647 Return Values

2648	GrB_SUCCESS	In blocking mode, the operation completed successfully. In non-
2649		blocking mode, this indicates that the API checks for the input
2650		arguments passed successfully and the input arrays have been
2651		consumed. Either way, output matrix <b>A</b> is ready to be used in
2652		the next method of the sequence.
2653	GrB_PANIC	Unknown internal error.
2654	GrB_OUT_OF_MEMORY	Not enough memory available for operation.
2655	GrB_UNINITIALIZED_OBJECT	The GrB_Type object has not been initialized by a call to GrB_Type_new
2656		(needed for user-defined types).
2657	GrB_NULL_POINTER	<b>A</b> , <b>indptr</b> , <b>indices</b> or <b>values</b> pointer is NULL.
2658	GrB_INDEX_OUT_OF_BOUNDS	A value in <b>indptr</b> or <b>indices</b> is outside the allowed range for indices
2659		in <b>A</b> and or the size of <b>values</b> , <b>n_values</b> , depending on the value
2660		of <b>format</b> .
2661	GrB_INVALID_VALUE	<b>nrows</b> or <b>ncols</b> is zero or outside the range of the type GrB_Index.
2662	GrB_DOMAIN_MISMATCH	The domain given in parameter <b>d</b> does not match the element
2663		type of <b>values</b> .

## 2664 Description

2665 Creates a new matrix **A** of domain **D(d)** and dimension **nrows**  $\times$  **ncols**. The new GraphBLAS  
2666 matrix will be filled with the contents of the matrix pointed to by **indptr**, and **indices**, and **values**.  
2667 The method returns a handle to the new matrix in **A**. The structure of the data being imported is  
2668 defined by **format**, which must be equal to one of the values defined in Section 3.5.3.1. Details of  
2669 the contents of **indptr**, **indices** and **values** for each supported format is given in Appendix B.

2670 It is not an error to call this method more than once on the same output matrix; however, the  
2671 handle to the previously created object will be overwritten.

### 2672 4.2.5.18 Matrix\_serializeSize: Compute the serialize buffer size

2673 Compute the buffer size (in bytes) necessary to serialize a GrB\_Matrix using GrB\_Matrix\_serialize.

## 2674 C Syntax

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



2675 **Parameters**

2676           size (OUT) Pointer to GrB\_Index value where size in bytes of serialized object will be  
2677           written.

2678           A (IN) A GraphBLAS matrix object.

2679 **Return Values**

2680           GrB\_SUCCESS The operation completed successfully and the value pointed to  
2681           by \*size has been computed and is ready to use.

2682           GrB\_PANIC Unknown internal error.

2683           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2684           GrB\_NULL\_POINTER size is NULL.

2685 **Description**

2686 Returns the size in bytes of the data buffer necessary to serialize the GraphBLAS matrix object A.  
2687 Users may then allocate a buffer of size bytes to pass as a parameter to GrB\_Matrix\_serialize.

2688 **4.2.5.19 Matrix\_serialize: Serialize a GraphBLAS matrix.**

2689 Serialize a GraphBLAS Matrix object into an opaque stream of bytes.

2690 **C Syntax**

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

2691 **Parameters**

2692   serialized\_data (INOUT) Pointer to the preallocated buffer where the serialized matrix will be  
2693   written.

2694   serialized\_size (INOUT) On input, the size in bytes of the buffer pointed to by serialized\_data.  
2695   On output, the number of bytes written to serialized\_data.

2696   A (IN) A GraphBLAS matrix object.

## 2697 Return Values

2698                   GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
2699                   cessfully. This indicates that the compatibility tests on the in-  
2700                   put argument passed successfully, and the output buffer serial-  
2701                   ized\_data and serialized\_size, have been computed and are ready  
2702                   to use.

2703                   GrB\_PANIC Unknown internal error.

2704                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
2705                   opaque GraphBLAS objects (input or output) is in an invalid  
2706                   state caused by a previous execution error. Call GrB\_error() to  
2707                   access any error messages generated by the implementation.

2708                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

2709                   GrB\_NULL\_POINTER serialized\_data or serialize\_size is NULL.

2710                   GrB\_UNINITIALIZED\_OBJECT The GraphBLAS matrix, A, has not been initialized by a call to  
2711                   any matrix constructor.

2712                   GrB\_INSUFFICIENT\_SPACE The size of the buffer serialized\_data (provided as an input seri-  
2713                   alized\_size) was not large enough.

## 2714 Description

2715 Serializes a GraphBLAS matrix object to an opaque buffer. To guarantee successful execution,  
2716 the size of the buffer pointed to by serialized\_data, provided as an input by serialized\_size, must  
2717 be of at least the number of bytes returned from GrB\_Matrix\_serializeSize. The actual size of the  
2718 serialized matrix written to serialized\_data is provided upon completion as an output written to  
2719 serialized\_size.

2720 The contents of the serialized buffer are implementation defined. Thus, a serialized matrix created  
2721 with one library implementation is not necessarily valid for deserialization with another implemen-  
2722 tation.

### 2723 4.2.5.20 Matrix\_deserialize: Deserialize a GraphBLAS matrix.

2724 Construct a new GraphBLAS matrix from a serialized object.

## 2725 C Syntax

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

## 2726 Parameters

2727           A (INOUT) On a successful return, contains a handle to the newly created Graph-  
2728           BLAS matrix.

2729           d (IN) the type of the matrix that was serialized in `serialized_data`.  
2730           If d is `GrB_NULL`, the implementation must attempt to deserialize the matrix  
2731           without a provided type object.

2732    `serialized_data` (IN) a pointer to a serialized GraphBLAS matrix created with `GrB_Matrix_serialize`.

2733    `serialized_size` (IN) the size of the buffer pointed to by `serialized_data` in bytes.

## 2734 Return Values

2735           `GrB_SUCCESS` In blocking mode, the operation completed successfully. In non-  
2736           blocking mode, this indicates that the API checks for the input  
2737           arguments passed successfully. Either way, output matrix A is  
2738           ready to be used in the next method of the sequence.

2739           `GrB_PANIC` Unknown internal error.

2740           `GrB_INVALID_OBJECT` This is returned if `serialized_data` is invalid or corrupted.

2741           `GrB_OUT_OF_MEMORY` Not enough memory available for operation.

2742           `GrB_UNINITIALIZED_OBJECT` The `GrB_Type` object has not been initialized by a call to `GrB_Type_new`  
2743           (needed for user-defined types).

2744           `GrB_NULL_POINTER` `serialized_data` or A is `NULL`.

2745           `GrB_DOMAIN_MISMATCH` The type given in d does not match the type of the matrix  
2746           serialized in `serialized_data`, or `GrB_NULL` was passed in and  
2747           the implementation is unable to construct the matrix without  
2748           the explicitly provided `GrB_Type`.

## 2749 Description

2750    Creates a new matrix **A** using the serialized matrix object pointed to by `serialized_data`. The object  
2751    pointed to by `serialized_data` must have been created using the method `GrB_Matrix_serialize`. The  
2752    domain of the matrix is given as an input in d, which must match the domain of the matrix serialized  
2753    in `serialized_data`. Note that for user-defined types, only the size of the type will be checked.

2754    Since the format of a serialized matrix is implementation-defined, it is not guaranteed that a matrix  
2755    serialized in one library implementation can be deserialized by another.

2756    It is not an error to call this method more than once on the same output matrix; however, the  
2757    handle to the previously created object will be overwritten.

## 2758 4.2.6 Descriptor methods

2759 The methods in this section create and set values in descriptors. A descriptor is an opaque Graph-  
2760 BLAS object the values of which are used to modify the behavior of GraphBLAS operations.

### 2761 4.2.6.1 Descriptor\_new: Create new descriptor

2762 Creates a new (empty or default) descriptor.

## 2763 C Syntax

```
2764 GrB_Info GrB_Descriptor_new(GrB_Descriptor *desc);
```

## 2765 Parameters

2766 desc (INOUT) On successful return, contains a handle to the newly created GraphBLAS  
2767 descriptor.

## 2768 Return Value

2769 GrB\_SUCCESS The method completed successfully.

2770 GrB\_PANIC unknown internal error.

2771 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2772 GrB\_NULL\_POINTER desc pointer is NULL.

## 2773 Description

2774 Creates a new descriptor object and returns a handle to it in desc. A newly created descriptor can  
2775 be populated by calls to Descriptor\_set.

2776 It is not an error to call this method more than once on the same variable; however, the handle to  
2777 the previously created object will be overwritten.

### 2778 4.2.6.2 Descriptor\_set: Set content of descriptor

2779 Sets the content for a field for an existing descriptor.

## 2780 C Syntax

```
2781         GrB_Info GrB_Descriptor_set(GrB_Descriptor      desc,  
2782                                     GrB_Desc_Field      field,  
2783                                     GrB_Desc_Value      val);
```

## 2784 Parameters

2785 desc (IN) An existing GraphBLAS descriptor to be modified.

2786 field (IN) The field being set.

2787 val (IN) New value for the field being set.

## 2788 Return Values

2789 GrB\_SUCCESS operation completed successfully.

2790 GrB\_PANIC unknown internal error.

2791 GrB\_OUT\_OF\_MEMORY not enough memory available for operation.

2792 GrB\_UNINITIALIZED\_OBJECT the desc parameter has not been initialized by a call to new.

2793 GrB\_INVALID\_VALUE invalid value set on the field, or invalid field.

## 2794 Description

2795 For a given descriptor, the GrB\_Descriptor\_set method can be called for each field in the descriptor  
2796 to set the value associated with that field. Valid values for the field parameter include the following:

2797 GrB\_OUTP refers to the output parameter (result) of the operation.

2798 GrB\_MASK refers to the mask parameter of the operation.

2799 GrB\_INP0 refers to the first input parameters of the operation (matrices and vectors).

2800 GrB\_INP1 refers to the second input parameters of the operation (matrices and vectors).

2801 Valid values for the val parameter are:

2802 GrB\_STRUCTURE Use only the structure of the stored values of the corresponding mask  
2803 (GrB\_MASK) parameter.

2804 GrB\_COMP Use the complement of the corresponding mask (GrB\_MASK) param-  
2805 eter. When combined with GrB\_STRUCTURE, the complement of the  
2806 structure of the mask is used without evaluating the values stored.

2807           GrB\_TRAN Use the transpose of the corresponding matrix parameter (valid for input  
2808           matrix parameters only).

2809           GrB\_REPLACE When assigning the masked values to the output matrix or vector, clear  
2810           the matrix first (or clear the non-masked entries). The default behavior  
2811           is to leave non-masked locations unchanged. Valid for the GrB\_OUTP  
2812           parameter only.

2813 Descriptor values can only be set, and once set, cannot be cleared. As, in the case of GrB\_MASK,  
2814 multiple values can be set and all will apply (for example, both GrB\_COMP and GrB\_STRUCTURE).  
2815 A value for a given field may be set multiple times but will have no additional effect. Fields that  
2816 have no values set result in their default behavior, as defined in Section 3.6.

#### 2817 **4.2.7 free: Destroy an object and release its resources**

2818 Destroys a previously created GraphBLAS object and releases any resources associated with the  
2819 object.

#### 2820 **C Syntax**

2821           GrB\_Info GrB\_free(<GrB\_Object> \*obj);

#### 2822 **Parameters**

2823           obj (INOUT) An existing GraphBLAS object to be destroyed. The object must have  
2824           been created by an explicit call to a GraphBLAS constructor. It can be any of the  
2825           opaque GraphBLAS objects such as matrix, vector, descriptor, semiring, monoid,  
2826           binary op, unary op, or type. On successful completion of GrB\_free, obj behaves  
2827           as an uninitialized object.

#### 2828 **Return Values**

2829           GrB\_SUCCESS operation completed successfully

2830           GrB\_PANIC unknown internal error. If this return value is encountered when  
2831           in nonblocking mode, the error responsible for the panic condition  
2832           could be from any method involved in the computation of the input  
2833           object. The GrB\_error() method should be called for additional  
2834           information.

#### 2835 **Description**

2836 GraphBLAS objects consume memory and other resources managed by the GraphBLAS runtime  
2837 system. A call to GrB\_free frees those resources so they are available for use by other GraphBLAS

2838 objects.

2839 The parameter passed into `GrB_free` is a handle referencing a GraphBLAS opaque object of a data  
2840 type from table 2.1. The object must have been created by an explicit call to a GraphBLAS con-  
2841 structor. The behavior of a program that calls `GrB_free` on a pre-defined object is implementation  
2842 defined.

2843 After the `GrB_free` method returns, the object referenced by the input handle is destroyed and the  
2844 handle has the value `GrB_INVALID_HANDLE`. The handle can be used in subsequent GraphBLAS  
2845 methods but only after the handle has been reinitialized with a call the the appropriate `_new` or  
2846 `_dup` method.

2847 Note that unlike other GraphBLAS methods, calling `GrB_free` with an object with an invalid handle  
2848 is legal. The system may attempt to free resources that might be associated with that object, if  
2849 possible, and return normally.

2850 When using `GrB_free` it is possible to create a dangling reference to an object. This would occur  
2851 when a handle is assigned to a second variable of the same opaque type. This creates two handles  
2852 that reference the same object. If `GrB_free` is called with one of the variables, the object is destroyed  
2853 and the handle associated with the other variable no longer references a valid object. This is not an  
2854 error condition that the implementation of the GraphBLAS API can be expected to catch, hence  
2855 programmers must take care to prevent this situation from occurring.

#### 2856 4.2.8 wait: Return once an object is either *complete* or *materialized*

2857 Wait until method calls in a sequence put an object into a state of *completion* or *materialization*.

#### 2858 C Syntax

```
2859 GrB_Info GrB_wait(GrB_Object obj, GrB_WaitMode mode);
```

#### 2860 Parameters

2861 `obj` (INOUT) An existing GraphBLAS object. The object must have been created by an  
2862 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2863 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2864 or type. On successful return of `GrB_wait`, the `obj` can be safely read from another  
2865 thread (completion) or all computing to produce `obj` by all GraphBLAS operations  
2866 in its sequence have finished (materialization).

2867 `mode` (IN) Set's the mode for `GrB_wait` for whether it is waiting for `obj` to be in the  
2868 state of *completion* or *materialization*. Acceptable values are `GrB_COMPLETE` or  
2869 `GrB_MATERIALIZE`.

## 2870 Return values

2871                   GrB\_SUCCESS operation completed successfully.

2872 GrB\_INDEX\_OUT\_OF\_BOUNDS an index out-of-bounds execution error happened during com-  
2873 pletion of pending operations.

2874                   GrB\_OUT\_OF\_MEMORY and out-of-memory execution error happened during completion  
2875 of pending operations.

2876 GrB\_UNINITIALIZED\_OBJECT object has not been initialized by a call to the respective \*\_new,  
2877 or other constructor, method.

2878                   GrB\_PANIC unknown internal error.

2879                   GrB\_INVALID\_VALUE method called with a GrB\_WaitMode other than GrB\_COMPLETE  
2880 GrB\_MATERIALIZE.

## 2881 Description

2882 On successful return from GrB\_wait(), the input object, **obj** is in one of two states depending on  
2883 the mode of GrB\_wait:

- 2884       • *complete*: **obj** can be used in a happens-before relation, so in a properly synchronized program  
2885       it can be safely used as an IN or INOUT parameter in a GraphBLAS method call from another  
2886       thread. This result occurs when the mode parameter is set to GrB\_COMPLETE.
- 2887       • *materialized*: **obj** is *complete*, but in addition, no further computing will be carried out on  
2888       behalf of **obj** and error information is available. This result occurs when the mode parameter  
2889       is set to GrB\_MATERIALIZE.

2890 Since in blocking mode OUT or INOUT parameters to any method call are materialized upon return,  
2891 GrB\_wait(**obj**,mode) has no effect when called in blocking mode.

2892 In non-blocking mode, the status of any pending method calls, other than those associated with pro-  
2893 ducing the *complete* or *materialized* state of **obj**, are not impacted by the call to GrB\_wait(**obj**,mode).  
2894 Methods in the sequence for **obj**, however, most likely would be impacted by a call to GrB\_wait(**obj**,mode);  
2895 especially in the case of the *materialized* mode for which any computing on behalf of **obj** must be  
2896 finished prior to the return from GrB\_wait(**obj**,mode).

## 2897 4.2.9 error: Retrieve an error string

2898 Retrieve an error-message about any errors encountered during the processing associated with an  
2899 object.



## 2900 C Syntax

```
2901      GrB_Info GrB_error(const char      **error,  
2902                        const GrB_Object  obj);
```

## 2903 Parameters

2904 error (OUT) A pointer to a null-terminated string. The contents of the string are im-  
2905 plementation defined.

2906 obj (IN) An existing GraphBLAS object. The object must have been created by an  
2907 explicit call to a GraphBLAS constructor. Can be any of the opaque GraphBLAS  
2908 objects such as matrix, vector, descriptor, semiring, monoid, binary op, unary op,  
2909 or type.

## 2910 Return value

2911 GrB\_SUCCESS operation completed successfully.

2912 GrB\_UNINITIALIZED\_OBJECT object has not been initialized by a call to the respective \*\_new,  
2913 or other constructor, method.

2914 GrB\_PANIC unknown internal error.

## 2915 Description

2916 This method retrieves a message related to any errors that were encountered during the last Graph-  
2917 BLAS method that had the opaque GraphBLAS object, obj, as an OUT or INOUT parameter.  
2918 The function returns a pointer to a null-terminated string and the contents of that string are  
2919 implementation-dependent. In particular, a null string (not a NULL pointer) is always a valid error  
2920 string. The string that is returned is owned by obj and will be valid until the next time obj is  
2921 used as an OUT or INOUT parameter or the object is freed by a call to GrB\_free(obj). This is a  
2922 thread-safe function. It can be safely called by multiple threads for the same object in a race-free  
2923 program.

## 2924 4.3 GraphBLAS operations

2925 The GraphBLAS operations are defined in the GraphBLAS math specification and summarized in  
2926 Table 4.1. In addition to methods that implement these fundamental GraphBLAS operations, we  
2927 support a number of variants that have been found to be especially useful in algorithm development.  
2928 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	$\mathbf{C}\langle\mathbf{M}, r\rangle$	=	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$

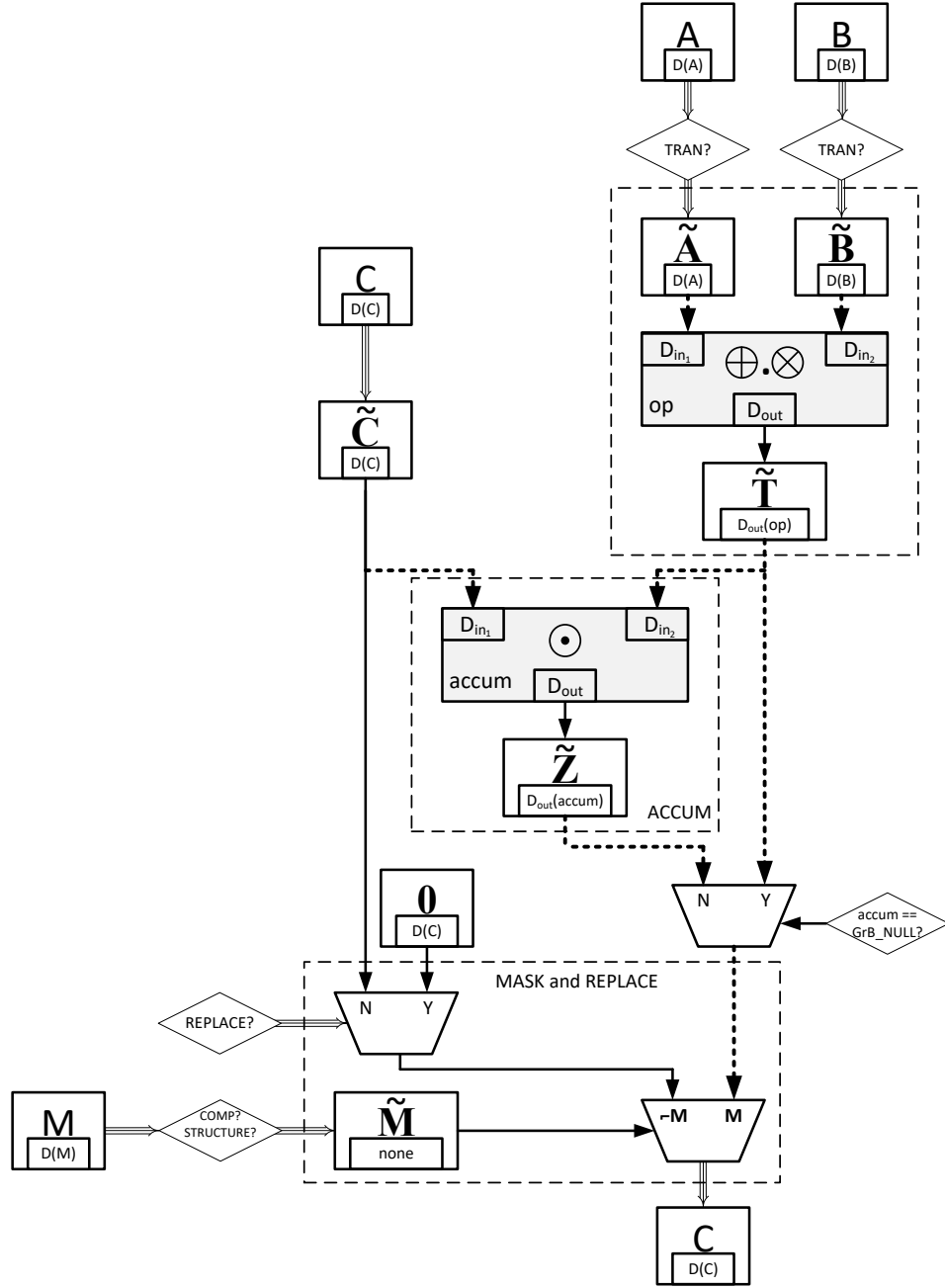


Figure 4.1: Flowchart for the GraphBLAS operations. Although shown specifically for the mxm operation, many elements are common to all operations: such as the “ACCUM” and “MASK and REPLACE” blocks. The triple arrows ( $\Rightarrow$ ) denote where “as if copy” takes place (including both collections and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

2964 operation as follows:

$$2965 \quad \mathbf{m} = \begin{cases} \langle N, \{\mathbf{ind}(\mathbf{v})\} \rangle, & \text{if GrB\_STRUCTURE is specified,} \\ \langle N, \{i : (\mathbf{bool})v_i = \mathbf{true}\} \rangle, & \text{otherwise} \end{cases}$$

2966 where  $(\mathbf{bool})v_i$  denotes casting the value  $v_i$  to a Boolean value (**true** or **false**). Likewise, given a  
2967 GraphBLAS matrix  $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$ , a two-dimensional mask is derived for use in the  
2968 operation as follows:

$$2969 \quad \mathbf{M} = \begin{cases} \langle M, N, \{\mathbf{ind}(\mathbf{A})\} \rangle, & \text{if GrB\_STRUCTURE is specified,} \\ \langle M, N, \{(i, j) : (\mathbf{bool})A_{ij} = \mathbf{true}\} \rangle, & \text{otherwise} \end{cases}$$

2970 where  $(\mathbf{bool})A_{ij}$  denotes casting the value  $A_{ij}$  to a Boolean value. (**true** or **false**)

2971 In both the one- and two-dimensional cases, the mask may also have a subsequent complement  
2972 operation applied (Section 3.5.4) as specified in the descriptor, before a final mask is generated for  
2973 use in the operation.

2974 When the descriptor of an operation with a mask has specified that the GrB\_REPLACE value is  
2975 to be applied to the output (GrB\_OUTP), then anywhere the mask is not **true**, the corresponding  
2976 location in the output is cleared.

## 2977 Invalid and uninitialized objects

2978 Upon entering a GraphBLAS operation, the first step is a check that all objects are valid and ini-  
2979 tialized. (Optional parameters can be set to GrB\_NULL, which always counts as a valid object.) An  
2980 invalid object is one that could not be computed due to a previous execution error. An uninitialized  
2981 object is one that has not yet been created by a corresponding **new** or **dup** method. Appropriate  
2982 error codes are returned if an object is not initialized (GrB\_UNINITIALIZED\_OBJECT) or invalid  
2983 (GrB\_INVALID\_OBJECT).

2984 To support the detection of as many cases of uninitialized objects as possible, it is strongly rec-  
2985 ommended to initialize all GraphBLAS objects to the predefined value GrB\_INVALID\_HANDLE at  
2986 the point of their declaration, as shown in the following examples:

```
2987         GrB_Type          type = GrB_INVALID_HANDLE;
2988         GrB_Semiring      semiring = GrB_INVALID_HANDLE;
2989         GrB_Matrix       matrix = GrB_INVALID_HANDLE;
```

## 2990 Compliance

2991 We follow a *prescriptive* approach to the definition of the semantics of GraphBLAS operations.  
2992 That is, for each operation we give a recipe for producing its outcome. Any implementation that  
2993 produces the same outcome, and follows the GraphBLAS execution model (Section 2.5) and error  
2994 model (Section 2.6) is a conforming implementation.

### 2995 4.3.1 mxm: Matrix-matrix multiply

2996 Multiplies a matrix with another matrix on a semiring. The result is a matrix.

#### 2997 C Syntax

```
2998         GrB_Info GrB_mxm(GrB_Matrix      C,  
2999                         const GrB_Matrix  Mask,  
3000                         const GrB_BinaryOp accum,  
3001                         const GrB_Semiring op,  
3002                         const GrB_Matrix  A,  
3003                         const GrB_Matrix  B,  
3004                         const GrB_Descriptor desc);
```

#### 3005 Parameters

3006 C (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3007 that may be accumulated with the result of the matrix product. On output, the  
3008 matrix holds the results of the operation.

3009 Mask (IN) An optional “write” mask that controls which results from this operation are  
3010 stored into the output matrix C. The mask dimensions must match those of the  
3011 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
3012 of the Mask matrix must be of type `bool` or any of the predefined “built-in” types  
3013 in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
3014 dimensions of C), GrB\_NULL should be specified.

3015 accum (IN) An optional binary operator used for accumulating entries into existing C  
3016 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3017 specified.

3018 op (IN) The semiring used in the matrix-matrix multiply.

3019 A (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
3020 multiplication.

3021 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
3022 multiplication.

3023 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
3024 should be specified. Non-default field/value pairs are listed as follows:

3025

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.

3052 **Compute** The indicated computations are carried out.

3053 **Output** The result is written into the output matrix, possibly under control of a mask.

3054 Up to four argument matrices are used in the `GrB_mxm` operation:

- 3055 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 3056 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)
- 3057 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 3058 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3059 The argument matrices, the semiring, and the accumulation operator (if provided) are tested for  
3060 domain compatibility as follows:

- 3061 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
3062 must be from one of the pre-defined types of Table 3.2.
- 3063 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the semiring.
- 3064 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the semiring.
- 3065 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the semiring.
- 3066 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3067 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
3068 of the accumulation operator.

3069 Two domains are compatible with each other if values from one domain can be cast to values in  
3070 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
3071 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
3072 If any compatibility rule above is violated, execution of `GrB_mxm` ends and the domain mismatch  
3073 error listed above is returned.

3074 From the argument matrices, the internal matrices and mask used in the computation are formed  
3075 ( $\leftarrow$  denotes copy):

- 3076 1. Matrix  $\tilde{C} \leftarrow C$ .
- 3077 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:
  - 3078 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
3079  $j < \mathbf{ncols}(C)\} \rangle$ .
  - 3080 (b) If `Mask`  $\neq$  `GrB_NULL`,
    - 3081 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
3082  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,



3083                   ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 3084                    $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\}\rangle.$

3085                   (c) If  $\mathbf{desc}[\mathbf{GrB\_MASK}].\mathbf{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg\widetilde{\mathbf{M}}.$

3086       3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}.$

3087       4. Matrix  $\widetilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}.$

3088   The internal matrices and masks are checked for dimension compatibility. The following conditions  
 3089   must hold:

3090       1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}}).$

3091       2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}}).$

3092       3.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}}).$

3093       4.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{B}}).$

3094       5.  $\mathbf{ncols}(\widetilde{\mathbf{A}}) = \mathbf{nrows}(\widetilde{\mathbf{B}}).$

3095   If any compatibility rule above is violated, execution of  $\mathbf{GrB\_mxm}$  ends and the dimension mismatch  
 3096   error listed above is returned.

3097   From this point forward, in  $\mathbf{GrB\_NONBLOCKING}$  mode, the method can optionally exit with  
 3098    $\mathbf{GrB\_SUCCESS}$  return code and defer any computation and/or execution error codes.

3099   We are now ready to carry out the matrix multiplication and any additional associated operations.  
 3100   We describe this in terms of two intermediate matrices:

- 3101       •  $\widetilde{\mathbf{T}}$ : The matrix holding the product of matrices  $\widetilde{\mathbf{A}}$  and  $\widetilde{\mathbf{B}}$ .
- 3102       •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3103   The intermediate matrix  $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{B}}), \{(i, j, T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:, j)) \neq \emptyset\} \rangle$   
 3104   is created. The value of each of its elements is computed by

$$3105 \quad T_{ij} = \bigoplus_{k \in \mathbf{ind}(\widetilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:, j))} (\widetilde{\mathbf{A}}(i, k) \otimes \widetilde{\mathbf{B}}(k, j)),$$

3106   where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring  $\mathbf{op}$ , respectively.

3107   The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3108       • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 3109       • If  $\mathbf{accum}$  is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$3110 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

3111 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
 3112 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned}
 3113 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
 3114 \\
 3115 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
 3116 \\
 3117 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),
 \end{aligned}$$

3118 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3119 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3120 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3121 mask which acts as a “write mask”.

- 3122 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3123 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3124 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3125 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3126 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3127 mask are unchanged:

$$3128 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3129 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3130 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3131 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 3132 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3133 sequence.

### 3134 4.3.2 vxm: Vector-matrix multiply

3135 Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

#### 3136 C Syntax

```

3137      GrB_Info GrB_vxm(GrB_Vector      w,
3138                      const GrB_Vector mask,
3139                      const GrB_BinaryOp accum,
3140                      const GrB_Semiring op,
3141                      const GrB_Vector u,
3142                      const GrB_Matrix A,
3143                      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.

3173        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 3174        GraphBLAS objects (input or output) is in an invalid state caused  
 3175        by a previous execution error. Call `GrB_error()` to access any error  
 3176        messages generated by the implementation.

3177        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3178 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 3179        a call to `new` (or `dup` for matrix or vector parameters).

3180 **GrB\_DIMENSION\_MISMATCH** Mask, vector, and/or matrix dimensions are incompatible.

3181        **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors/matrices are incompatible with  
 3182        the corresponding domains of the semiring or accumulation opera-  
 3183        tor, or the mask's domain is not compatible with `bool` (in the case  
 3184        where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

## 3185 Description

3186 **GrB\_vxm** computes the vector-matrix product  $\mathbf{w}^T = \mathbf{u}^T \oplus . \otimes \mathbf{A}$ , or, if an optional binary accu-  
 3187        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  
 3188        transposed). Logically, this operation occurs in three steps:

3189        **Setup** The internal vectors, matrices and mask used in the computation are formed and their  
 3190        domains/dimensions are tested for compatibility.

3191        **Compute** The indicated computations are carried out.

3192        **Output** The result is written into the output vector, possibly under control of a mask.

3193 Up to four argument vectors or matrices are used in the **GrB\_vxm** operation:

- 3194 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3195 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3196 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3197 4.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

3198 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
 3199 tested for domain compatibility as follows:

- 3200 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3201        must be from one of the pre-defined types of Table 3.2.
- 3202 2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the semiring.

- 3203 3.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the semiring.
- 3204 4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring.
- 3205 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$
- 3206 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$
- 3207 of the accumulation operator.

3208 Two domains are compatible with each other if values from one domain can be cast to values in

3209 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are

3210 all compatible with each other. A domain from a user-defined type is only compatible with itself.

3211 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the domain mismatch

3212 error listed above is returned.

3213 From the argument vectors and matrices, the internal matrices and mask used in the computation

3214 are formed ( $\leftarrow$  denotes copy):

- 3215 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3216 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 3217 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 3218 (b) If `mask  $\neq$  GrB_NULL`,
- 3219 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 3220 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
- 3221 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3222 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 3223 4. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

3224 The internal matrices and masks are checked for shape compatibility. The following conditions

3225 must hold:

- 3226 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$ .
- 3227 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .
- 3228 3.  $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

3229 If any compatibility rule above is violated, execution of `GrB_vxm` ends and the dimension mismatch

3230 error listed above is returned.

3231 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with

3232 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3233 We are now ready to carry out the vector-matrix multiplication and any additional associated

3234 operations. We describe this in terms of two intermediate vectors:

- 3235 •  $\tilde{\mathbf{t}}$ : The vector holding the product of vector  $\tilde{\mathbf{u}}^T$  and matrix  $\tilde{\mathbf{A}}$ .
- 3236 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3237 The intermediate vector  $\tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(j, t_j) : \mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{A}}(:, j)) \neq \emptyset\} \rangle$  is created.  
 3238 The value of each of its elements is computed by

$$3239 \quad t_j = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{u}}) \cap \mathbf{ind}(\tilde{\mathbf{A}}(:, j))} (\tilde{\mathbf{u}}(k) \otimes \tilde{\mathbf{A}}(k, j)),$$

3240 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring  $\mathbf{op}$ , respectively.

3241 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3242 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3243 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3244 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3245 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 3246 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3247 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3248 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3249 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3250 \quad & \\ 3251 \end{aligned}$$

3252 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3253 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3254 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3255 mask which acts as a “write mask”.

- 3256 • If  $\mathbf{desc}[\mathbf{GrB\_OUTP}].\mathbf{GrB\_REPLACE}$  is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3257 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3258 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3259 • If  $\mathbf{desc}[\mathbf{GrB\_OUTP}].\mathbf{GrB\_REPLACE}$  is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3260 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3261 mask are unchanged:

$$3262 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3263 In  $\mathbf{GrB\_BLOCKING}$  mode, the method exits with return value  $\mathbf{GrB\_SUCCESS}$  and the new content  
 3264 of vector  $\mathbf{w}$  is as defined above and fully computed. In  $\mathbf{GrB\_NONBLOCKING}$  mode, the method  
 3265 exits with return value  $\mathbf{GrB\_SUCCESS}$  and the new content of vector  $\mathbf{w}$  is as defined above but  
 3266 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3267 sequence.

### 3268 4.3.3 mxv: Matrix-vector multiply

3269 Multiplies a matrix by a vector on a semiring. The result is a vector.

#### 3270 C Syntax

```
3271     GrB_Info GrB_mxv(GrB_Vector      w,  
3272                     const GrB_Vector mask,  
3273                     const GrB_BinaryOp accum,  
3274                     const GrB_Semiring op,  
3275                     const GrB_Matrix  A,  
3276                     const GrB_Vector  u,  
3277                     const GrB_Descriptor desc);
```

#### 3278 Parameters

3279 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3280 that may be accumulated with the result of the matrix-vector product. On output,  
3281 this vector holds the results of the operation.

3282 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3283 stored into the output vector **w**. The mask dimensions must match those of the  
3284 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3285 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3286 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3287 dimensions of **w**), **GrB\_NULL** should be specified.

3288 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3289 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3290 specified.

3291 **op** (IN) Semiring used in the vector-matrix multiply.

3292 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
3293 multiplication.

3294 **u** (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
3295 multiplication.

3296 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
3297 should be specified. Non-default field/value pairs are listed as follows:  
3298

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.

3299

## 3300 Return Values

3301 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
3302 blocking mode, this indicates that the compatibility tests on di-  
3303 mensions and domains for the input arguments passed successfully.  
3304 Either way, output vector w is ready to be used in the next method  
3305 of the sequence.

3306 GrB\_PANIC Unknown internal error.

3307 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
3308 GraphBLAS objects (input or output) is in an invalid state caused  
3309 by a previous execution error. Call GrB\_error() to access any error  
3310 messages generated by the implementation.

3311 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

3312 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
3313 a call to new (or dup for matrix or vector parameters).

3314 GrB\_DIMENSION\_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

3315 GrB\_DOMAIN\_MISMATCH The domains of the various vectors/matrices are incompatible with  
3316 the corresponding domains of the semiring or accumulation opera-  
3317 tor, or the mask's domain is not compatible with bool (in the case  
3318 where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 3319 Description

3320 GrB\_mvx computes the matrix-vector product  $w = A \oplus . \otimes u$ , or, if an optional binary accumulation  
3321 operator ( $\odot$ ) is provided,  $w = w \odot (A \oplus . \otimes u)$  (where matrix A can be optionally transposed).  
3322 Logically, this operation occurs in three steps:

3323 **Setup** The internal vectors, matrices and mask used in the computation are formed and their  
3324 domains/dimensions are tested for compatibility.

3325 **Compute** The indicated computations are carried out.



3326     **Output** The result is written into the output vector, possibly under control of a mask.

3327 Up to four argument vectors or matrices are used in the `GrB_mvx` operation:

- 3328     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3329     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3330     3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 3331     4.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

3332 The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are  
 3333 tested for domain compatibility as follows:

- 3334     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3335         must be from one of the pre-defined types of Table 3.2.
- 3336     2.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the semiring.
- 3337     3.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the semiring.
- 3338     4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring.
- 3339     5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3340         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the semiring must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$   
 3341         of the accumulation operator.

3342 Two domains are compatible with each other if values from one domain can be cast to values in  
 3343 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are  
 3344 all compatible with each other. A domain from a user-defined type is only compatible with itself.  
 3345 If any compatibility rule above is violated, execution of `GrB_mvx` ends and the domain mismatch  
 3346 error listed above is returned.

3347 From the argument vectors and matrices, the internal matrices and mask used in the computation  
 3348 are formed ( $\leftarrow$  denotes copy):

- 3349     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3350     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3351         (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3352         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3353             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 3354             ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool}(\mathbf{mask})(i) = \mathbf{true})\} \rangle$ .
  - 3355         (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 3356     3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

3357 4. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3358 The internal matrices and masks are checked for shape compatibility. The following conditions  
3359 must hold:

- 3360 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$ .
- 3361 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 3362 3.  $\mathbf{size}(\tilde{\mathbf{u}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

3363 If any compatibility rule above is violated, execution of `GrB_m xv` ends and the dimension mismatch  
3364 error listed above is returned.

3365 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3366 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3367 We are now ready to carry out the matrix-vector multiplication and any additional associated  
3368 operations. We describe this in terms of two intermediate vectors:

- 3369 •  $\tilde{\mathbf{t}}$ : The vector holding the product of matrix  $\tilde{\mathbf{A}}$  and vector  $\tilde{\mathbf{u}}$ .
- 3370 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3371 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.  
3372 The value of each of its elements is computed by

$$3373 \quad t_i = \bigoplus_{k \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :)) \cap \mathbf{ind}(\tilde{\mathbf{u}})} (\tilde{\mathbf{A}}(i, k) \otimes \tilde{\mathbf{u}}(k)),$$

3374 where  $\oplus$  and  $\otimes$  are the additive and multiplicative operators of semiring `op`, respectively.

3375 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3376 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 3377 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$3378 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

3379 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3380 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 3381 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 3382 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3383 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3384 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 3385 \end{aligned}$$

3386 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3387 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3388 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3389 mask which acts as a “write mask”.

- 3390 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3391 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3392 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3393 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3394 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3395 mask are unchanged:

$$3396 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3397 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3398 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3399 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3400 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3401 sequence.

#### 3402 4.3.4 eWiseMult: Element-wise multiplication

3403 **Note:** The difference between eWiseAdd and eWiseMult is not about the element-wise operation  
 3404 but how the index sets are treated. eWiseAdd returns an object whose indices are the “union” of  
 3405 the indices of the inputs whereas eWiseMult returns an object whose indices are the “intersection”  
 3406 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3407 the set of values from the resulting index set.

##### 3408 4.3.4.1 eWiseMult: Vector variant

3409 Perform element-wise (general) multiplication on the intersection of elements of two vectors, pro-  
 3410 ducing a third vector as result.

#### 3411 C Syntax

```
3412      GrB_Info GrB_eWiseMult(GrB_Vector      w,
3413                           const GrB_Vector  mask,
3414                           const GrB_BinaryOp accum,
3415                           const GrB_Semiring op,
3416                           const GrB_Vector  u,
3417                           const GrB_Vector  v,
3418                           const GrB_Descriptor desc);
3419
```

```

3420 GrB_Info GrB_eWiseMult(GrB_Vector          w,
3421                        const GrB_Vector      mask,
3422                        const GrB_BinaryOp     accum,
3423                        const GrB_Monoid       op,
3424                        const GrB_Vector       u,
3425                        const GrB_Vector       v,
3426                        const GrB_Descriptor    desc);
3427
3428 GrB_Info GrB_eWiseMult(GrB_Vector          w,
3429                        const GrB_Vector      mask,
3430                        const GrB_BinaryOp     accum,
3431                        const GrB_BinaryOp     op,
3432                        const GrB_Vector       u,
3433                        const GrB_Vector       v,
3434                        const GrB_Descriptor    desc);

```

### 3435 Parameters

3436 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3437 that may be accumulated with the result of the element-wise operation. On output,  
3438 this vector holds the results of the operation.

3439 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3440 stored into the output vector **w**. The mask dimensions must match those of the  
3441 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3442 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3443 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
3444 dimensions of **w**), **GrB\_NULL** should be specified.

3445 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3446 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3447 specified.

3448 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
3449 operation. Depending on which type is passed, the following defines the binary  
3450 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3451 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3452 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-  
3453 nored.

3454 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  
3455 is ignored.

3456 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3457 operation.

3458 **v** (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3459 operation.

3460 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 3461 should be specified. Non-default field/value pairs are listed as follows:  
 3462

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> .

## 3464 Return Values

3465 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 3466 blocking mode, this indicates that the compatibility tests on di-  
 3467 mensions and domains for the input arguments passed successfully.  
 3468 Either way, output vector **w** is ready to be used in the next method  
 3469 of the sequence.

3470 **GrB\_PANIC** Unknown internal error.

3471 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 3472 GraphBLAS objects (input or output) is in an invalid state caused  
 3473 by a previous execution error. Call **GrB\_error()** to access any error  
 3474 messages generated by the implementation.

3475 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3476 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 3477 a call to **new** (or **dup** for vector parameters).

3478 **GrB\_DIMENSION\_MISMATCH** Mask or vector dimensions are incompatible.

3479 **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the cor-  
 3480 responding domains of the binary operator (**op**) or accumulation  
 3481 operator, or the mask's domain is not compatible with **bool** (in the  
 3482 case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

## 3483 Description

3484 This variant of **GrB\_eWiseMult** computes the element-wise “product” of two GraphBLAS vectors:  
 3485  $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $\mathbf{w} = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$ .  
 3486 Logically, this operation occurs in three steps:

3487     **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3488             and dimensions are tested for compatibility.

3489     **Compute** The indicated computations are carried out.

3490     **Output** The result is written into the output vector, possibly under control of a mask.

3491     Up to four argument vectors are used in the `GrB_eWiseMult` operation:

- 3492     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3493     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3494     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3495     4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3496     The argument vectors, the “product” operator (`op`), and the accumulation operator (if provided)  
 3497     are tested for domain compatibility as follows:

- 3498     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3499         must be from one of the pre-defined types of Table 3.2.
- 3500     2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3501     3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3502     4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3503     5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3504         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
 3505         the accumulation operator.

3506     Two domains are compatible with each other if values from one domain can be cast to values in  
 3507     the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3508     compatible with each other. A domain from a user-defined type is only compatible with itself. If any  
 3509     compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
 3510     error listed above is returned.

3511     From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3512     denotes copy):

- 3513     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3514     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 3515         (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 3516         (b) If `mask`  $\neq$  `GrB_NULL`,
    - 3517             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,

3518 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .

3519 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .

3520 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

3521 4. Vector  $\widetilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3522 The internal vectors and mask are checked for dimension compatibility. The following conditions  
3523 must hold:

3524 1.  $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}}) = \mathbf{size}(\widetilde{\mathbf{u}}) = \mathbf{size}(\widetilde{\mathbf{v}})$ .

3525 If any compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the dimension  
3526 mismatch error listed above is returned.

3527 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
3528 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

3529 We are now ready to carry out the element-wise “product” and any additional associated operations.  
3530 We describe this in terms of two intermediate vectors:

- 3531 •  $\widetilde{\mathbf{t}}$ : The vector holding the element-wise “product” of  $\widetilde{\mathbf{u}}$  and vector  $\widetilde{\mathbf{v}}$ .
- 3532 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3533 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  
3534 value of each of its elements is computed by:

$$3535 \quad t_i = (\widetilde{\mathbf{u}}(i) \otimes \widetilde{\mathbf{v}}(i)), \forall i \in (\mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{v}}))$$

3536 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3537 • If `accum = GrB_NULL`, then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 3538 • If `accum` is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$3539 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\widetilde{\mathbf{w}}) \cup \mathbf{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

3540 The values of the elements of  $\widetilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3541 indices in  $\widetilde{\mathbf{w}}$  and  $\widetilde{\mathbf{t}}$ .

$$\begin{aligned} 3542 \quad z_i &= \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}})), \\ 3543 \quad z_i &= \widetilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{w}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))), \\ 3544 \quad z_i &= \widetilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\widetilde{\mathbf{t}}) - (\mathbf{ind}(\widetilde{\mathbf{t}}) \cap \mathbf{ind}(\widetilde{\mathbf{w}}))), \\ 3545 \quad & \\ 3546 \end{aligned}$$

3547 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3548 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3549 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3550 mask which acts as a “write mask”.

- 3551 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3552 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3553 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3554 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3555 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3556 mask are unchanged:

$$3557 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3558 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3559 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3560 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3561 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3562 sequence.

#### 3563 4.3.4.2 eWiseMult: Matrix variant

3564 Perform element-wise (general) multiplication on the intersection of elements of two matrices, pro-  
 3565 ducing a third matrix as result.

### 3566 C Syntax

```

3567     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3568                           const GrB_Matrix Mask,
3569                           const GrB_BinaryOp accum,
3570                           const GrB_Semiring op,
3571                           const GrB_Matrix A,
3572                           const GrB_Matrix B,
3573                           const GrB_Descriptor desc);
3574
3575     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
3576                           const GrB_Matrix Mask,
3577                           const GrB_BinaryOp accum,
3578                           const GrB_Monoid op,
3579                           const GrB_Matrix A,
3580                           const GrB_Matrix B,
3581                           const GrB_Descriptor desc);
3582
3583     GrB_Info GrB_eWiseMult(GrB_Matrix      C,
```



```

3584         const GrB_Matrix      Mask,
3585         const GrB_BinaryOp     accum,
3586         const GrB_BinaryOp     op,
3587         const GrB_Matrix      A,
3588         const GrB_Matrix      B,
3589         const GrB_Descriptor   desc);

```

## 3590 Parameters

3591 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
3592 that may be accumulated with the result of the element-wise operation. On output,  
3593 the matrix holds the results of the operation.

3594 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
3595 stored into the output matrix C. The mask dimensions must match those of the  
3596 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
3597 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
3598 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
3599 dimensions of C), GrB\_NULL should be specified.

3600 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
3601 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3602 specified.

3603 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
3604 operation. Depending on which type is passed, the following defines the binary  
3605 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

3606 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3607 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-  
3608 nored.

3609 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  
3610 is ignored.

3611 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
3612 operation.

3613 **B** (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
3614 operation.

3615 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
3616 should be specified. Non-default field/value pairs are listed as follows:

3617

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.

3644 **Compute** The indicated computations are carried out.

3645 **Output** The result is written into the output matrix, possibly under control of a mask.

3646 Up to four argument matrices are used in the `GrB_eWiseMult` operation:

- 3647 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$   
3648 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)  
3649 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$   
3650 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

3651 The argument matrices, the “product” operator (`op`), and the accumulation operator (if provided)  
3652 are tested for domain compatibility as follows:

- 3653 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
3654 must be from one of the pre-defined types of Table 3.2.  
3655 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .  
3656 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .  
3657 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .  
3658 5. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
3659 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
3660 the accumulation operator.

3661 Two domains are compatible with each other if values from one domain can be cast to values in  
3662 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
3663 compatible with each other. A domain from a user-defined type is only compatible with itself. If any  
3664 compatibility rule above is violated, execution of `GrB_eWiseMult` ends and the domain mismatch  
3665 error listed above is returned.

3666 From the argument matrices, the internal matrices and mask used in the computation are formed  
3667 ( $\leftarrow$  denotes copy):

- 3668 1. Matrix  $\tilde{C} \leftarrow C$ .  
3669 2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:  
3670 (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
3671  $j < \mathbf{ncols}(C)\} \rangle$ .  
3672 (b) If `Mask`  $\neq$  `GrB_NULL`,  
3673 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
3674  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,

3675                   ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
3676                    $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\}\rangle.$   
3677                   (c) If  $\mathbf{desc}[\mathbf{GrB\_MASK}].\mathbf{GrB\_COMP}$  is set, then  $\tilde{\mathbf{M}} \leftarrow \neg\tilde{\mathbf{M}}.$

3678   3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}.$

3679   4. Matrix  $\tilde{\mathbf{B}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}.$

3680   The internal matrices and masks are checked for dimension compatibility. The following conditions  
3681   must hold:

3682   1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}) = \mathbf{nrows}(\tilde{\mathbf{A}}) = \mathbf{nrows}(\tilde{\mathbf{B}}).$

3683   2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}) = \mathbf{ncols}(\tilde{\mathbf{A}}) = \mathbf{ncols}(\tilde{\mathbf{B}}).$

3684   If any compatibility rule above is violated, execution of  $\mathbf{GrB\_eWiseMult}$  ends and the dimension  
3685   mismatch error listed above is returned.

3686   From this point forward, in  $\mathbf{GrB\_NONBLOCKING}$  mode, the method can optionally exit with  
3687    $\mathbf{GrB\_SUCCESS}$  return code and defer any computation and/or execution error codes.

3688   We are now ready to carry out the element-wise “product” and any additional associated operations.  
3689   We describe this in terms of two intermediate matrices:

- 3690   •  $\tilde{\mathbf{T}}$ : The matrix holding the element-wise product of  $\tilde{\mathbf{A}}$  and  $\tilde{\mathbf{B}}$ .
- 3691   •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

3692   The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}}) \neq \emptyset\} \rangle$   
3693   is created. The value of each of its elements is computed by

$$3694 \quad T_{ij} = (\tilde{\mathbf{A}}(i, j) \otimes \tilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}}) \cap \mathbf{ind}(\tilde{\mathbf{B}})$$

3695   The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 3696   • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 3697   • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$3698 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

3699   The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
3700   indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 3701 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 3702 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3703 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 3704 \quad & \\ 3705 \quad & \end{aligned}$$

3706   where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

3707 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 3708 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 3709 mask which acts as a “write mask”.

- 3710 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 3711 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$3712 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 3713 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 3714 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 3715 mask are unchanged:

$$3716 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

3717 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3718 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3719 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 3720 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3721 sequence.

### 3722 4.3.5 eWiseAdd: Element-wise addition

3723 **Note:** The difference between eWiseAdd and eWiseMult is not about the element-wise operation  
 3724 but how the index sets are treated. eWiseAdd returns an object whose indices are the “union” of  
 3725 the indices of the inputs whereas eWiseMult returns an object whose indices are the “intersection”  
 3726 of the indices of the inputs. In both cases, the passed semiring, monoid, or operator operates on  
 3727 the set of values from the resulting index set.

#### 3728 4.3.5.1 eWiseAdd: Vector variant

3729 Perform element-wise (general) addition on the elements of two vectors, producing a third vector  
 3730 as result.

### 3731 C Syntax

```
3732      GrB_Info GrB_eWiseAdd(GrB_Vector      w,
3733                           const GrB_Vector mask,
3734                           const GrB_BinaryOp accum,
3735                           const GrB_Semiring op,
3736                           const GrB_Vector u,
3737                           const GrB_Vector v,
3738                           const GrB_Descriptor desc);
3739
```

```

3740     GrB_Info GrB_eWiseAdd(GrB_Vector      w,
3741                          const GrB_Vector mask,
3742                          const GrB_BinaryOp accum,
3743                          const GrB_Monoid op,
3744                          const GrB_Vector u,
3745                          const GrB_Vector v,
3746                          const GrB_Descriptor desc);
3747
3748     GrB_Info GrB_eWiseAdd(GrB_Vector      w,
3749                          const GrB_Vector mask,
3750                          const GrB_BinaryOp accum,
3751                          const GrB_BinaryOp op,
3752                          const GrB_Vector u,
3753                          const GrB_Vector v,
3754                          const GrB_Descriptor desc);

```

## 3755 Parameters

3756 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
3757 that may be accumulated with the result of the element-wise operation. On output,  
3758 this vector holds the results of the operation.

3759 **mask** (IN) An optional “write” mask that controls which results from this operation are  
3760 stored into the output vector **w**. The mask dimensions must match those of the  
3761 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
3762 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
3763 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
3764 dimensions of **w**), **GrB\_NULL** should be specified.

3765 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
3766 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
3767 specified.

3768 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “sum”  
3769 operation. Depending on which type is passed, the following defines the binary  
3770 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3771 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3772 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-  
3773 nored.

3774 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-  
3775 nary op and additive identity are ignored.

3776 **u** (IN) The GraphBLAS vector holding the values for the left-hand vector in the  
3777 operation.

3778  $v$  (IN) The GraphBLAS vector holding the values for the right-hand vector in the  
 3779 operation.

3780 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 3781 should be specified. Non-default field/value pairs are listed as follows:  
 3782

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> .

## 3784 Return Values

3785 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 3786 blocking mode, this indicates that the compatibility tests on di-  
 3787 mensions and domains for the input arguments passed successfully.  
 3788 Either way, output vector **w** is ready to be used in the next method  
 3789 of the sequence.

3790 **GrB\_PANIC** Unknown internal error.

3791 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 3792 GraphBLAS objects (input or output) is in an invalid state caused  
 3793 by a previous execution error. Call **GrB\_error()** to access any error  
 3794 messages generated by the implementation.

3795 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

3796 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 3797 a call to **new** (or **dup** for vector parameters).

3798 **GrB\_DIMENSION\_MISMATCH** Mask or vector dimensions are incompatible.

3799 **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with the cor-  
 3800 responding domains of the binary operator (**op**) or accumulation  
 3801 operator, or the mask's domain is not compatible with **bool** (in the  
 3802 case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

## 3803 Description

3804 This variant of **GrB\_eWiseAdd** computes the element-wise “sum” of two GraphBLAS vectors:  $w =$   
 3805  $u \oplus v$ , or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $w = w \odot (u \oplus v)$ . Logically,  
 3806 this operation occurs in three steps:

3807     **Setup** The internal vectors and mask used in the computation are formed and their domains  
 3808             and dimensions are tested for compatibility.

3809     **Compute** The indicated computations are carried out.

3810     **Output** The result is written into the output vector, possibly under control of a mask.

3811     Up to four argument vectors are used in the `GrB_eWiseAdd` operation:

- 3812     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 3813     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 3814     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$
- 3815     4.  $\mathbf{v} = \langle \mathbf{D}(\mathbf{v}), \mathbf{size}(\mathbf{v}), \mathbf{L}(\mathbf{v}) = \{(i, v_i)\} \rangle$

3816     The argument vectors, the “sum” operator (`op`), and the accumulation operator (if provided) are  
 3817     tested for domain compatibility as follows:

- 3818     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
 3819         must be from one of the pre-defined types of Table 3.2.
- 3820     2.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3821     3.  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3822     4.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3823     5.  $\mathbf{D}(\mathbf{u})$  and  $\mathbf{D}(\mathbf{v})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3824     6. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 3825         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of `op` must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
 3826         the accumulation operator.

3827     Two domains are compatible with each other if values from one domain can be cast to values in  
 3828     the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 3829     compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 3830     any compatibility rule above is violated, execution of `GrB_eWiseAdd` ends and the domain mismatch  
 3831     error listed above is returned.

3832     From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 3833     denotes copy):

- 3834     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 3835     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:  
 3836         (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .



3837 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,  
3838 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,  
3839 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .  
3840 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .  
3841 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .  
3842 4. Vector  $\widetilde{\mathbf{v}} \leftarrow \mathbf{v}$ .

3843 The internal vectors and mask are checked for dimension compatibility. The following conditions  
3844 must hold:

3845 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}}) = \text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{v}})$ .

3846 If any compatibility rule above is violated, execution of  $\text{GrB\_eWiseAdd}$  ends and the dimension  
3847 mismatch error listed above is returned.

3848 From this point forward, in  $\text{GrB\_NONBLOCKING}$  mode, the method can optionally exit with  
3849  $\text{GrB\_SUCCESS}$  return code and defer any computation and/or execution error codes.

3850 We are now ready to carry out the element-wise “sum” and any additional associated operations.  
3851 We describe this in terms of two intermediate vectors:

- 3852 •  $\widetilde{\mathbf{t}}$ : The vector holding the element-wise “sum” of  $\widetilde{\mathbf{u}}$  and vector  $\widetilde{\mathbf{v}}$ .
- 3853 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

3854 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  
3855 value of each of its elements is computed by:

$$\begin{aligned}
3856 \quad t_i &= (\widetilde{\mathbf{u}}(i) \oplus \widetilde{\mathbf{v}}(i)), \forall i \in (\text{ind}(\widetilde{\mathbf{u}}) \cap \text{ind}(\widetilde{\mathbf{v}})) \\
3857 \\
3858 \quad t_i &= \widetilde{\mathbf{u}}(i), \forall i \in (\text{ind}(\widetilde{\mathbf{u}}) - (\text{ind}(\widetilde{\mathbf{u}}) \cap \text{ind}(\widetilde{\mathbf{v}}))) \\
3859 \\
3860 \quad t_i &= \widetilde{\mathbf{v}}(i), \forall i \in (\text{ind}(\widetilde{\mathbf{v}}) - (\text{ind}(\widetilde{\mathbf{u}}) \cap \text{ind}(\widetilde{\mathbf{v}})))
\end{aligned}$$

3861 where the difference operator in the previous expressions refers to set difference.

3862 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 3863 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 3864 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$3865 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

3866 The values of the elements of  $\widetilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
3867 indices in  $\widetilde{\mathbf{w}}$  and  $\widetilde{\mathbf{t}}$ .

$$\begin{aligned}
3868 \quad z_i &= \widetilde{\mathbf{w}}(i) \odot \widetilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\widetilde{\mathbf{t}}) \cap \text{ind}(\widetilde{\mathbf{w}})), \\
3869 \\
3870 \quad z_i &= \widetilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\widetilde{\mathbf{w}}) - (\text{ind}(\widetilde{\mathbf{t}}) \cap \text{ind}(\widetilde{\mathbf{w}}))), \\
3871 \\
3872 \quad z_i &= \widetilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\widetilde{\mathbf{t}}) - (\text{ind}(\widetilde{\mathbf{t}}) \cap \text{ind}(\widetilde{\mathbf{w}}))),
\end{aligned}$$

3873 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

3874 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 3875 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 3876 mask which acts as a “write mask”.

- 3877 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{w}$  on input to this operation are  
 3878 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$3879 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 3880 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 3881 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 3882 mask are unchanged:

$$3883 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg\tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

3884 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 3885 of vector  $\mathbf{w}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 3886 exits with return value GrB\_SUCCESS and the new content of vector  $\mathbf{w}$  is as defined above but  
 3887 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 3888 sequence.

#### 3889 4.3.5.2 eWiseAdd: Matrix variant

3890 Perform element-wise (general) addition on the elements of two matrices, producing a third matrix  
 3891 as result.

#### 3892 C Syntax

```

3893     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3894                           const GrB_Matrix Mask,
3895                           const GrB_BinaryOp accum,
3896                           const GrB_Semiring op,
3897                           const GrB_Matrix A,
3898                           const GrB_Matrix B,
3899                           const GrB_Descriptor desc);
3900
3901     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
3902                           const GrB_Matrix Mask,
3903                           const GrB_BinaryOp accum,
3904                           const GrB_Monoid op,
3905                           const GrB_Matrix A,
3906                           const GrB_Matrix B,
3907                           const GrB_Descriptor desc);
3908
3909     GrB_Info GrB_eWiseAdd(GrB_Matrix      C,
```

```

3910         const GrB_Matrix      Mask,
3911         const GrB_BinaryOp    accum,
3912         const GrB_BinaryOp    op,
3913         const GrB_Matrix      A,
3914         const GrB_Matrix      B,
3915         const GrB_Descriptor   desc);

```

## 3916 Parameters

3917 **C (INOUT)** An existing GraphBLAS matrix. On input, the matrix provides values  
3918 that may be accumulated with the result of the element-wise operation. On output,  
3919 the matrix holds the results of the operation.

3920 **Mask (IN)** An optional “write” mask that controls which results from this operation are  
3921 stored into the output matrix C. The mask dimensions must match those of the  
3922 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
3923 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
3924 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
3925 dimensions of C), GrB\_NULL should be specified.

3926 **accum (IN)** An optional binary operator used for accumulating entries into existing C  
3927 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
3928 specified.

3929 **op (IN)** The semiring, monoid, or binary operator used in the element-wise “sum”  
3930 operation. Depending on which type is passed, the following defines the binary  
3931 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \oplus \rangle$ , used:

3932 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

3933 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-  
3934 nored.

3935 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-  
3936 nary op and additive identity are ignored.

3937 **A (IN)** The GraphBLAS matrix holding the values for the left-hand matrix in the  
3938 operation.

3939 **B (IN)** The GraphBLAS matrix holding the values for the right-hand matrix in the  
3940 operation.

3941 **desc (IN)** An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
3942 should be specified. Non-default field/value pairs are listed as follows:

3943

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.

3970 **Compute** The indicated computations are carried out.

3971 **Output** The result is written into the output matrix, possibly under control of a mask.

3972 Up to four argument matrices are used in the GrB\_eWiseAdd operation:

- 3973 1.  $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 3974 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)
- 3975 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$
- 3976 4.  $\mathbf{B} = \langle \mathbf{D}(\mathbf{B}), \mathbf{nrows}(\mathbf{B}), \mathbf{ncols}(\mathbf{B}), \mathbf{L}(\mathbf{B}) = \{(i, j, B_{ij})\} \rangle$

3977 The argument matrices, the “sum” operator (**op**), and the accumulation operator (if provided) are  
3978 tested for domain compatibility as follows:

- 3979 1. If **Mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\mathbf{Mask})$   
3980 must be from one of the pre-defined types of Table 3.2.
- 3981 2.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$ .
- 3982 3.  $\mathbf{D}(\mathbf{B})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$ .
- 3983 4.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3984 5.  $\mathbf{D}(\mathbf{A})$  and  $\mathbf{D}(\mathbf{B})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$ .
- 3985 6. If **accum** is not GrB\_NULL, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
3986 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of **op** must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of  
3987 the accumulation operator.

3988 Two domains are compatible with each other if values from one domain can be cast to values in  
3989 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
3990 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
3991 any compatibility rule above is violated, execution of GrB\_eWiseAdd ends and the domain mismatch  
3992 error listed above is returned.

3993 From the argument matrices, the internal matrices and mask used in the computation are formed  
3994 ( $\leftarrow$  denotes copy):

- 3995 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 3996 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument **Mask** as follows:
- 3997 (a) If **Mask** = GrB\_NULL, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
3998  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 3999 (b) If **Mask**  $\neq$  GrB\_NULL,

4000 i. If desc[GrB\_MASK].GrB\_STRUCTURE is set, then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
4001  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,  
4002 ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
4003  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .  
4004 (c) If desc[GrB\_MASK].GrB\_COMP is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$ .  
4005 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .  
4006 4. Matrix  $\widetilde{\mathbf{B}} \leftarrow \text{desc}[\text{GrB\_INP1}].\text{GrB\_TRAN} ? \mathbf{B}^T : \mathbf{B}$ .

4007 The internal matrices and masks are checked for dimension compatibility. The following conditions  
4008 must hold:

- 4009 1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}}) = \mathbf{nrows}(\widetilde{\mathbf{A}}) = \mathbf{nrows}(\widetilde{\mathbf{B}})$ .  
4010 2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}}) = \mathbf{ncols}(\widetilde{\mathbf{A}}) = \mathbf{ncols}(\widetilde{\mathbf{B}})$ .

4011 If any compatibility rule above is violated, execution of GrB\_eWiseAdd ends and the dimension  
4012 mismatch error listed above is returned.

4013 From this point forward, in GrB\_NONBLOCKING mode, the method can optionally exit with  
4014 GrB\_SUCCESS return code and defer any computation and/or execution error codes.

4015 We are now ready to carry out the element-wise “sum” and any additional associated operations.  
4016 We describe this in terms of two intermediate matrices:

- 4017 •  $\widetilde{\mathbf{T}}$ : The matrix holding the element-wise sum of  $\widetilde{\mathbf{A}}$  and  $\widetilde{\mathbf{B}}$ .
- 4018 •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4019 The intermediate matrix  $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \{(i, j, T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}) \cup \mathbf{ind}(\widetilde{\mathbf{B}}) \neq \emptyset\} \rangle$   
4020 is created. The value of each of its elements is computed by

$$\begin{aligned}
4021 \quad T_{ij} &= (\widetilde{\mathbf{A}}(i, j) \oplus \widetilde{\mathbf{B}}(i, j)), \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}}) \\
4022 \quad T_{ij} &= \widetilde{\mathbf{A}}(i, j), \forall (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{A}}) - (\mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}}))) \\
4023 \quad T_{ij} &= \widetilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{B}}) - (\mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}}))) \\
4024 \quad T_{ij} &= \widetilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{B}}) - (\mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}}))) \\
4025 \quad T_{ij} &= \widetilde{\mathbf{B}}(i, j), \forall (i, j) \in (\mathbf{ind}(\widetilde{\mathbf{B}}) - (\mathbf{ind}(\widetilde{\mathbf{A}}) \cap \mathbf{ind}(\widetilde{\mathbf{B}})))
\end{aligned}$$

4026 where the difference operator in the previous expressions refers to set difference.

4027 The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4028 • If accum = GrB\_NULL, then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 4029 • If accum is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$4030 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

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,
```

```

4062         const GrB_BinaryOp      accum,
4063         const GrB_Vector        u,
4064         const GrB_Index         *indices,
4065         GrB_Index               nindices,
4066         const GrB_Descriptor     desc);

```

## 4067 Parameters

4068     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
4069             that may be accumulated with the result of the extract operation. On output, this  
4070             vector holds the results of the operation.

4071     **mask** (IN) An optional “write” mask that controls which results from this operation are  
4072             stored into the output vector **w**. The mask dimensions must match those of the  
4073             vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
4074             of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
4075             in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
4076             dimensions of **w**), **GrB\_NULL** should be specified.

4077     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
4078             entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4079             specified.

4080     **u** (IN) The GraphBLAS vector from which the subset is extracted.

4081     **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations of  
4082             elements from **u** that are extracted. If all elements of **u** are to be extracted in order  
4083             from 0 to **nindices** − 1, then **GrB\_ALL** should be specified. Regardless of execution  
4084             mode and return value, this array may be manipulated by the caller after this  
4085             operation returns without affecting any deferred computations for this operation.

4086     **nindices** (IN) The number of values in **indices** array. Must be equal to **size(w)**.

4087     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4088             should be specified. Non-default field/value pairs are listed as follows:  
4089

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> .



## 4091 Return Values

4092                   GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 4093                   blocking mode, this indicates that the compatibility tests on  
 4094                   dimensions and domains for the input arguments passed suc-  
 4095                   cessfully. Either way, output vector **w** is ready to be used in the  
 4096                   next method of the sequence.

4097                   GrB\_PANIC Unknown internal error.

4098                   GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 4099                   opaque GraphBLAS objects (input or output) is in an invalid  
 4100                   state caused by a previous execution error. Call **GrB\_error()** to  
 4101                   access any error messages generated by the implementation.

4102                   GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

4103                   GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
 4104                   by a call to **new** (or **dup** for vector parameters).

4105                   GrB\_INDEX\_OUT\_OF\_BOUNDS A value in **indices** is greater than or equal to **size(u)**. In non-  
 4106                   blocking mode, this error can be deferred.

4107                   GrB\_DIMENSION\_MISMATCH **mask** and **w** dimensions are incompatible, or **nindices**  $\neq$  **size(w)**.

4108                   GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with each  
 4109                   other or the corresponding domains of the accumulation oper-  
 4110                   ator, or the mask's domain is not compatible with **bool** (in the  
 4111                   case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

4112                   GrB\_NULL\_POINTER Argument **row\_indices** is a NULL pointer.

## 4113 Description

4114 This variant of **GrB\_extract** computes the result of extracting a subset of locations from a Graph-  
 4115 BLAS vector in a specific order: **w** = **u(indices)**; or, if an optional binary accumulation operator  
 4116 ( $\odot$ ) is provided, **w** = **w**  $\odot$  **u(indices)**. More explicitly:

$$4117 \quad \begin{aligned} \mathbf{w}(i) &= \mathbf{u}(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ \mathbf{w}(i) &= \mathbf{w}(i) \odot \mathbf{u}(\text{indices}[i]), \forall i : 0 \leq i < \text{nindices} \end{aligned}$$

4118 Logically, this operation occurs in three steps:

4119       **Setup** The internal vectors and mask used in the computation are formed and their domains  
 4120       and dimensions are tested for compatibility.

4121       **Compute** The indicated computations are carried out.

4122       **Output** The result is written into the output vector, possibly under control of a mask.

4123 Up to three argument vectors are used in this GrB\_extract operation:

- 4124 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 4125 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4126 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

4127 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4128 bility as follows:

- 4129 1. If  $\mathbf{mask}$  is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\mathbf{mask})$   
4130 must be from one of the pre-defined types of Table 3.2.
- 4131 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 4132 3. If  $\mathbf{accum}$  is not GrB\_NULL, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
4133 of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
4134 mulation operator.

4135 Two domains are compatible with each other if values from one domain can be cast to values in  
4136 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4137 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4138 any compatibility rule above is violated, execution of GrB\_extract ends and the domain mismatch  
4139 error listed above is returned.

4140 From the arguments, the internal vectors,  $\mathbf{mask}$ , and index array used in the computation are  
4141 formed ( $\leftarrow$  denotes copy):

- 4142 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 4143 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument  $\mathbf{mask}$  as follows:  
4144 (a) If  $\mathbf{mask} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .  
4145 (b) If  $\mathbf{mask} \neq \text{GrB\_NULL}$ ,  
4146 i. If desc[GrB\_MASK].GrB\_STRUCTURE is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,  
4147 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\text{bool})\mathbf{mask}(i) = \text{true}\} \rangle$ .  
4148 (c) If desc[GrB\_MASK].GrB\_COMP is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 4149 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 4150 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument indices as follows:  
4151 (a) If  $\mathbf{indices} = \text{GrB\_ALL}$ , then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$ .  
4152 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$ .

4153 The internal vectors and mask are checked for dimension compatibility. The following conditions  
4154 must hold:

4155 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4156 2.  $\text{nindices} = \text{size}(\tilde{\mathbf{w}})$ .

4157 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4158 match error listed above is returned.

4159 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4160 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4161 We are now ready to carry out the extract and any additional associated operations. We describe  
4162 this in terms of two intermediate vectors:

- 4163 •  $\tilde{\mathbf{t}}$ : The vector holding the extraction from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{w}}$ .
- 4164 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4165 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4166 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\tilde{\mathbf{w}}), \{(i, \tilde{\mathbf{u}}[\tilde{\mathbf{I}}[i])) \mid \forall i, 0 \leq i < \text{nindices} : \tilde{\mathbf{I}}[i] \in \text{ind}(\tilde{\mathbf{u}})\} \rangle.$$

4167 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  
4168 `GrB_extract` ends and the index-out-of-bounds error listed above is generated. In `GrB_NONBLOCKING`  
4169 mode, the error can be deferred until a sequence-terminating `GrB_wait()` is called. Regardless, the  
4170 result vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4171 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 4172 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 4173 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4174 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4175 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
4176 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 4177 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 4178 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4179 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4180 \end{aligned}$$

4181 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4183 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
4184 using what is called a *standard vector mask and replace*. This is carried out under control of the  
4185 mask which acts as a “write mask”.

- 4186 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
4187 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$4188 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- 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.

4224           A (IN) The GraphBLAS matrix from which the subset is extracted.

4225   row\_indices (IN) Pointer to the ordered set (array) of indices corresponding to the rows of A  
4226           from which elements are extracted. If elements in all rows of A are to be extracted  
4227           in order, GrB\_ALL should be specified. Regardless of execution mode and return  
4228           value, this array may be manipulated by the caller after this operation returns  
4229           without affecting any deferred computations for this operation.

4230   nrows (IN) The number of values in the row\_indices array. Must be equal to **nrows**(C).

4231   col\_indices (IN) Pointer to the ordered set (array) of indices corresponding to the columns  
4232           of A from which elements are extracted. If elements in all columns of A are to  
4233           be extracted in order, then GrB\_ALL should be specified. Regardless of execution  
4234           mode and return value, this array may be manipulated by the caller after this  
4235           operation returns without affecting any deferred computations for this operation.

4236   ncols (IN) The number of values in the col\_indices array. Must be equal to **ncols**(C).

4237   desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
4238           should be specified. Non-default field/value pairs are listed as follows:

4239

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.

4240

## 4241 Return Values

4242           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
4243           blocking mode, this indicates that the compatibility tests on  
4244           dimensions and domains for the input arguments passed suc-  
4245           cessfully. Either way, output matrix C is ready to be used in the  
4246           next method of the sequence.

4247           GrB\_PANIC Unknown internal error.

4248           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
4249           opaque GraphBLAS objects (input or output) is in an invalid  
4250           state caused by a previous execution error. Call GrB\_error() to  
4251           access any error messages generated by the implementation.

4252           GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

4253 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
4254 by a call to `new` (or `Matrix_dup` for matrix parameters).

4255 GrB\_INDEX\_OUT\_OF\_BOUNDS A value in `row_indices` is greater than or equal to `nrows(A)`, or  
4256 a value in `col_indices` is greater than or equal to `ncols(A)`. In  
4257 non-blocking mode, this error can be deferred.

4258 GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, `nrows`  $\neq$  `nrows(C)`, or  
4259 `ncols`  $\neq$  `ncols(C)`.

4260 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with each  
4261 other or the corresponding domains of the accumulation oper-  
4262 ator, or the mask's domain is not compatible with `bool` (in the  
4263 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4264 GrB\_NULL\_POINTER Either argument `row_indices` is a NULL pointer, argument `col_indices`  
4265 is a NULL pointer, or both.

## 4266 Description

4267 This variant of `GrB_extract` computes the result of extracting a subset of locations from specified  
4268 rows and columns of a GraphBLAS matrix in a specific order:  $C = A(\text{row\_indices}, \text{col\_indices})$ ; or,  
4269 if an optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A(\text{row\_indices}, \text{col\_indices})$ .  
4270 More explicitly (not accounting for an optional transpose of A):

$$4271 \quad \begin{aligned} C(i, j) &= A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\ C(i, j) &= C(i, j) \odot A(\text{row\_indices}[i], \text{col\_indices}[j]) \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} \end{aligned}$$

4272 Logically, this operation occurs in three steps:

4273 **Setup** The internal matrices and mask used in the computation are formed and their domains  
4274 and dimensions are tested for compatibility.

4275 **Compute** The indicated computations are carried out.

4276 **Output** The result is written into the output matrix, possibly under control of a mask.

4277 Up to three argument matrices are used in the `GrB_extract` operation:

- 4278 1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4279 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)
- 4280 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

4281 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
4282 ibility as follows:

- 4283 1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
 4284 must be from one of the pre-defined types of Table 3.2.
- 4285 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}(\mathbf{A})$ .
- 4286 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4287 of the accumulation operator and  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4288 mulation operator.

4289 Two domains are compatible with each other if values from one domain can be cast to values in  
 4290 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4291 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4292 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
 4293 error listed above is returned.

4294 From the arguments, the internal matrices, mask, and index arrays used in the computation are  
 4295 formed ( $\leftarrow$  denotes copy):

- 4296 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 4297 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
- 4298 (a) If `Mask` = `GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 4299  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 4300 (b) If `Mask`  $\neq$  `GrB_NULL`,
- 4301 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
 4302  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
- 4303 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
 4304  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
- 4305 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4306 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4307 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
- 4308 (a) If `row_indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4309 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4310 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument `col_indices` as follows:
- 4311 (a) If `col_indices` = `GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \mathbf{ncols}$ .
- 4312 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \mathbf{ncols}$ .

4313 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 4314 must hold:

- 4315 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .

4316 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .

4317 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}$ .

4318 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}$ .

4319 If any compatibility rule above is violated, execution of `GrB_extract` ends and the dimension mis-  
4320 match error listed above is returned.

4321 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
4322 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4323 We are now ready to carry out the extract and any additional associated operations. We describe  
4324 this in terms of two intermediate matrices:

- 4325 •  $\tilde{\mathbf{T}}$ : The matrix holding the extraction from  $\tilde{\mathbf{A}}$ .
- 4326 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4327 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$4328 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(i, j, \tilde{\mathbf{A}}[\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]]) \mid \forall (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols} : (\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j]) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4329 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  $\tilde{\mathbf{J}}$   
4330 array is not in the range  $[0, \mathbf{ncols}(\tilde{\mathbf{A}}))$ , the execution of `GrB_extract` ends and the index out-of-  
4331 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
4332 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
4333 this point forward in the sequence.

4334 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 4335 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 4336 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$4337 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4338 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
4339 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$4340 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$4341 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4342 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$4343 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

4345 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.



4346 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 4347 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 4348 mask which acts as a “write mask”.

- 4349 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 4350 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$4351 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 4352 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 4353 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 4354 mask are unchanged:

$$4355 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

4356 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 4357 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 4358 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 4359 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 4360 sequence.

#### 4361 4.3.6.3 extract: Column (and row) variant

4362 Extract from one column of a matrix into a vector. Note that with the transpose descriptor for the  
 4363 source matrix, elements of an arbitrary row of the matrix can be extracted with this function as  
 4364 well.

#### 4365 C Syntax

```
4366      GrB_Info GrB_extract(GrB_Vector      w,
4367                          const GrB_Vector mask,
4368                          const GrB_BinaryOp accum,
4369                          const GrB_Matrix A,
4370                          const GrB_Index *row_indices,
4371                          GrB_Index      nrows,
4372                          GrB_Index      col_index,
4373                          const GrB_Descriptor desc);
```

#### 4374 Parameters

4375 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
 4376 that may be accumulated with the result of the extract operation. On output, this  
 4377 vector holds the results of the operation.

4378        **mask** (IN) An optional “write” mask that controls which results from this operation are  
 4379        stored into the output vector **w**. The mask dimensions must match those of the  
 4380        vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
 4381        of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
 4382        in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
 4383        dimensions of **w**), **GrB\_NULL** should be specified.

4384        **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
 4385        entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 4386        specified.

4387        **A** (IN) The GraphBLAS matrix from which the column subset is extracted.

4388        **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations  
 4389        within the specified column of **A** from which elements are extracted. If elements in  
 4390        all rows of **A** are to be extracted in order, **GrB\_ALL** should be specified. Regardless  
 4391        of execution mode and return value, this array may be manipulated by the caller  
 4392        after this operation returns without affecting any deferred computations for this  
 4393        operation.

4394        **nrows** (IN) The number of indices in the **row\_indices** array. Must be equal to **size(w)**.

4395        **col\_index** (IN) The index of the column of **A** from which to extract values. It must be in the  
 4396        range  $[0, \mathbf{ncols}(\mathbf{A})]$ .

4397        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 4398        should be specified. Non-default field/value pairs are listed as follows:  
 4399

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.

## 4401 Return Values

4402        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4403        blocking mode, this indicates that the compatibility tests on  
 4404        dimensions and domains for the input arguments passed suc-  
 4405        cessfully. Either way, output vector **w** is ready to be used in the  
 4406        next method of the sequence.

4407        **GrB\_PANIC** Unknown internal error.



4438 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

4439 The argument vectors, matrix and the accumulation operator (if provided) are tested for domain  
4440 compatibility as follows:

- 4441 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
4442 must be from one of the pre-defined types of Table 3.2.
- 4443 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{A})$ .
- 4444 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
4445 of the accumulation operator and  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
4446 mulation operator.

4447 Two domains are compatible with each other if values from one domain can be cast to values in  
4448 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
4449 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
4450 any compatibility rule above is violated, execution of `GrB_extract` ends and the domain mismatch  
4451 error listed above is returned.

4452 From the arguments, the internal vector, matrix, mask, and index array used in the computation  
4453 are formed ( $\leftarrow$  denotes copy):

- 4454 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 4455 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 4456 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 4457 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 4458 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
    - 4459 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 4460 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 4461 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4462 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:
  - 4463 (a) If `indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - 4464 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .

4465 The internal vector, mask, and index array are checked for dimension compatibility. The following  
4466 conditions must hold:

- 4467 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 4468 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}$ .

4469 If any compatibility rule above is violated, execution of GrB\_extract ends and the dimension mis-  
 4470 match error listed above is returned.

4471 The col\_index parameter is checked for a valid value. The following condition must hold:

4472 1.  $0 \leq \text{col\_index} < \text{ncols}(\mathbf{A})$

4473 If the rule above is violated, execution of GrB\_extract ends and the invalid index error listed above  
 4474 is returned.

4475 From this point forward, in GrB\_NONBLOCKING mode, the method can optionally exit with  
 4476 GrB\_SUCCESS return code and defer any computation and/or execution error codes.

4477 We are now ready to carry out the extract and any additional associated operations. We describe  
 4478 this in terms of two intermediate vectors:

- 4479 •  $\tilde{\mathbf{t}}$ : The vector holding the extraction from a column of  $\tilde{\mathbf{A}}$ .
- 4480 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4481 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4482 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}, \{(i, \tilde{\mathbf{A}}[\tilde{\mathbf{I}}[i], \text{col\_index}]) \mid \forall i, 0 \leq i < \text{nrows} : (\tilde{\mathbf{I}}[i], \text{col\_index}) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4483 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  
 4484 ends and the index-out-of-bounds error listed above is generated. In GrB\_NONBLOCKING mode,  
 4485 the error can be deferred until a sequence-terminating GrB\_wait() is called. Regardless, the result  
 4486 vector,  $\mathbf{w}$ , is invalid from this point forward in the sequence.

4487 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 4488 • If accum = GrB\_NULL, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 4489 • If accum is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$4490 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

4491 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 4492 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 4493 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 4494 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4495 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 4496 \quad & \\ 4497 \end{aligned}$$

4498 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

4499 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 4500 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 4501 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.

### 4.3.7 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

4533 vector **w** If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
 4534 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
 4535 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
 4536 dimensions of **w**), **GrB\_NULL** should be specified.

4537 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
 4538 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 4539 specified.

4540 **u** (IN) The GraphBLAS vector whose contents are assigned to a subset of **w**.

4541 **indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
 4542 **w** that are to be assigned. If all elements of **w** are to be assigned in order from 0  
 4543 to **nindices** – 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
 4544 and return value, this array may be manipulated by the caller after this operation  
 4545 returns without affecting any deferred computations for this operation. If this  
 4546 array contains duplicate values, it implies in assignment of more than one value to  
 4547 the same location which leads to undefined results.

4548 **nindices** (IN) The number of values in **indices** array. Must be equal to **size(u)**.

4549 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 4550 should be specified. Non-default field/value pairs are listed as follows:

4551

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> .

4552

## 4553 Return Values

4554 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4555 blocking mode, this indicates that the compatibility tests on  
 4556 dimensions and domains for the input arguments passed suc-  
 4557 cessfully. Either way, output vector **w** is ready to be used in the  
 4558 next method of the sequence.

4559 **GrB\_PANIC** Unknown internal error.

4560 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 4561 opaque GraphBLAS objects (input or output) is in an invalid  
 4562 state caused by a previous execution error. Call **GrB\_error()** to  
 4563 access any error messages generated by the implementation.

4564        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

4565        **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
4566                by a call to **new** (or **dup** for vector parameters).

4567        **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **indices** is greater than or equal to **size(w)**. In non-  
4568                blocking mode, this can be reported as an execution error.

4569        **GrB\_DIMENSION\_MISMATCH** **mask** and **w** dimensions are incompatible, or **nindices**  $\neq$  **size(u)**.

4570        **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors are incompatible with each  
4571                other or the corresponding domains of the accumulation oper-  
4572                ator, or the mask's domain is not compatible with **bool** (in the  
4573                case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

4574        **GrB\_NULL\_POINTER** Argument **indices** is a **NULL** pointer.

## 4575 Description

4576 This variant of **GrB\_assign** computes the result of assigning elements from a source GraphBLAS  
4577 vector to a destination GraphBLAS vector in a specific order:  $w(\text{indices}) = u$ ; or, if an optional  
4578 binary accumulation operator ( $\odot$ ) is provided,  $w(\text{indices}) = w(\text{indices}) \odot u$ . More explicitly:

$$\begin{aligned}
&w(\text{indices}[i]) = u(i), \forall i : 0 \leq i < \text{nindices}, \text{ or} \\
&w(\text{indices}[i]) = w(\text{indices}[i]) \odot u(i), \forall i : 0 \leq i < \text{nindices}.
\end{aligned}$$

4580 Logically, this operation occurs in three steps:

4581        **Setup** The internal vectors and mask used in the computation are formed and their domains  
4582                and dimensions are tested for compatibility.

4583        **Compute** The indicated computations are carried out.

4584        **Output** The result is written into the output vector, possibly under control of a mask.

4585 Up to three argument vectors are used in the **GrB\_assign** operation:

- 4586 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 4587 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4588 3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4589 The argument vectors and the accumulation operator (if provided) are tested for domain compati-  
4590 bility as follows:

- 4591 1. If **mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then **D(mask)**  
4592        must be from one of the pre-defined types of Table 3.2.



4593 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .

4594 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4595 of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4596 mulation operator.

4597 Two domains are compatible with each other if values from one domain can be cast to values in  
 4598 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4599 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4600 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 4601 error listed above is returned.

4602 From the arguments, the internal vectors, mask and index array used in the computation are formed  
 4603 ( $\leftarrow$  denotes copy):

4604 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .

4605 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

4606 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .

4607 (b) If `mask`  $\neq$  `GrB_NULL`,

4608 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,

4609 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

4610 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

4611 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4612 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument `indices` as follows:

4613 (a) If `indices` = `GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \text{nindices}$ .

4614 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{indices}[i], \forall i : 0 \leq i < \text{nindices}$ .

4615 The internal vector and mask are checked for dimension compatibility. The following conditions  
 4616 must hold:

4617 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

4618 2.  $\text{nindices} = \text{size}(\tilde{\mathbf{u}})$ .

4619 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 4620 match error listed above is returned.

4621 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4622 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4623 We are now ready to carry out the assign and any additional associated operations. We describe  
 4624 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,

$$L(w) = \{(i, z_i) : i \in (\text{ind}(\tilde{z}) \cap \text{ind}(\tilde{m}))\}.$$

- 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.

#### 4.3.7.2 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.

4693        **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
 4694        entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
 4695        specified.

4696        **A** (IN) The GraphBLAS matrix whose contents are assigned to a subset of **C**.

4697        **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of **C**  
 4698        that are assigned. If all rows of **C** are to be assigned in order from 0 to **nrows** - 1,  
 4699        then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
 4700        this array may be manipulated by the caller after this operation returns without  
 4701        affecting any deferred computations for this operation. If this array contains du-  
 4702        plicate values, it implies assignment of more than one value to the same location  
 4703        which leads to undefined results.

4704        **nrows** (IN) The number of values in the **row\_indices** array. Must be equal to **nrows(A)**  
 4705        if **A** is not transposed, or equal to **ncols(A)** if **A** is transposed.

4706        **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns  
 4707        of **C** that are assigned. If all columns of **C** are to be assigned in order from 0  
 4708        to **ncols** - 1, then **GrB\_ALL** should be specified. Regardless of execution mode  
 4709        and return value, this array may be manipulated by the caller after this operation  
 4710        returns without affecting any deferred computations for this operation. If this  
 4711        array contains duplicate values, it implies assignment of more than one value to  
 4712        the same location which leads to undefined results.

4713        **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **ncols(A)** if **A** is  
 4714        not transposed, or equal to **nrows(A)** if **A** is transposed.

4715        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 4716        should be specified. Non-default field/value pairs are listed as follows:

4717

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.

4718

## 4719 Return Values

4720        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 4721        blocking mode, this indicates that the compatibility tests on  
 4722        dimensions and domains for the input arguments passed suc-  
 4723        cessfully. Either way, output matrix **C** is ready to be used in the  
 4724        next method of the sequence.

4725                   GrB\_PANIC Unknown internal error.

4726           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
4727                   opaque GraphBLAS objects (input or output) is in an invalid  
4728                   state caused by a previous execution error. Call `GrB_error()` to  
4729                   access any error messages generated by the implementation.

4730           GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

4731   GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
4732                   by a call to `new` (or `Matrix_dup` for matrix parameters).

4733   GrB\_INDEX\_OUT\_OF\_BOUNDS A value in `row_indices` is greater than or equal to `nrows(C)`, or  
4734                   a value in `col_indices` is greater than or equal to `ncols(C)`. In  
4735                   non-blocking mode, this can be reported as an execution error.

4736   GrB\_DIMENSION\_MISMATCH Mask and `C` dimensions are incompatible, `nrows`  $\neq$  `nrows(A)`,  
4737                   or `ncols`  $\neq$  `ncols(A)`.

4738           GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with each  
4739                   other or the corresponding domains of the accumulation oper-  
4740                   ator, or the mask's domain is not compatible with `bool` (in the  
4741                   case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

4742           GrB\_NULL\_POINTER Either argument `row_indices` is a NULL pointer, argument `col_indices`  
4743                   is a NULL pointer, or both.

## 4744 Description

4745 This variant of `GrB_assign` computes the result of assigning the contents of `A` to a subset of rows  
4746 and columns in `C` in a specified order:  $C(\text{row\_indices}, \text{col\_indices}) = A$ ; or, if an optional binary  
4747 accumulation operator ( $\odot$ ) is provided,  $C(\text{row\_indices}, \text{col\_indices}) = C(\text{row\_indices}, \text{col\_indices}) \odot$   
4748 `A`. More explicitly (not accounting for an optional transpose of `A`):

$$\begin{aligned}
&C(\text{row\_indices}[i], \text{col\_indices}[j]) = A(i, j), \quad \forall i, j : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}, \text{ or} \\
&C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot A(i, j), \\
&\quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
\end{aligned}$$

4750 Logically, this operation occurs in three steps:

4751       Setup The internal matrices and mask used in the computation are formed and their domains  
4752                   and dimensions are tested for compatibility.

4753       Compute The indicated computations are carried out.

4754       Output The result is written into the output matrix, possibly under control of a mask.

4755 Up to three argument matrices are used in the `GrB_assign` operation:

- 4756 1.  $\mathbf{C} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij})\} \rangle$
- 4757 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)
- 4758 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

4759 The argument matrices and the accumulation operator (if provided) are tested for domain compat-  
 4760 ibility as follows:

- 4761 1. If  $\mathbf{Mask}$  is not  $\mathbf{GrB\_NULL}$ , and  $\text{desc}[\mathbf{GrB\_MASK}].\mathbf{GrB\_STRUCTURE}$  is not set, then  $\mathbf{D}(\mathbf{Mask})$   
 4762 must be from one of the pre-defined types of Table 3.2.
- 4763 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}(\mathbf{A})$ .
- 4764 3. If  $\mathbf{accum}$  is not  $\mathbf{GrB\_NULL}$ , then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
 4765 of the accumulation operator and  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
 4766 mulation operator.

4767 Two domains are compatible with each other if values from one domain can be cast to values in  
 4768 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4769 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4770 any compatibility rule above is violated, execution of  $\mathbf{GrB\_assign}$  ends and the domain mismatch  
 4771 error listed above is returned.

4772 From the arguments, the internal matrices,  $\mathbf{mask}$ , and index arrays used in the computation are  
 4773 formed ( $\leftarrow$  denotes copy):

- 4774 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 4775 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument  $\mathbf{Mask}$  as follows:
  - 4776 (a) If  $\mathbf{Mask} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
 4777  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 4778 (b) If  $\mathbf{Mask} \neq \mathbf{GrB\_NULL}$ ,
    - 4779 i. If  $\text{desc}[\mathbf{GrB\_MASK}].\mathbf{GrB\_STRUCTURE}$  is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
 4780  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 4781 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
 4782  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 4783 (c) If  $\text{desc}[\mathbf{GrB\_MASK}].\mathbf{GrB\_COMP}$  is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 4784 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \text{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .
- 4785 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument  $\mathbf{row\_indices}$  as follows:
  - 4786 (a) If  $\mathbf{row\_indices} = \mathbf{GrB\_ALL}$ , then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - 4787 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 4788 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument  $\mathbf{col\_indices}$  as follows:

- 4789 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .  
 4790 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

4791 The internal matrices and mask are checked for dimension compatibility. The following conditions  
 4792 must hold:

- 4793 1.  $\text{nrows}(\tilde{\mathbf{C}}) = \text{nrows}(\tilde{\mathbf{M}})$ .  
 4794 2.  $\text{ncols}(\tilde{\mathbf{C}}) = \text{ncols}(\tilde{\mathbf{M}})$ .  
 4795 3.  $\text{nrows}(\tilde{\mathbf{A}}) = \text{nrows}$ .  
 4796 4.  $\text{ncols}(\tilde{\mathbf{A}}) = \text{ncols}$ .

4797 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 4798 match error listed above is returned.

4799 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4800 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4801 We are now ready to carry out the assign and any additional associated operations. We describe  
 4802 this in terms of two intermediate vectors:

- 4803 •  $\tilde{\mathbf{T}}$ : The matrix holding the contents from  $\tilde{\mathbf{A}}$  in their destination locations relative to  $\tilde{\mathbf{C}}$ .  
 4804 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

4805 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

4806 
$$\tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \tilde{\mathbf{A}}(i, j)) \mid \forall (i, j), 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols} : (i, j) \in \text{ind}(\tilde{\mathbf{A}})\} \rangle.$$

4807 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  
 4808  $\tilde{\mathbf{J}}$  array is not in the range  $[0, \text{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-  
 4809 bounds error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred  
 4810 until a sequence-terminating `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from  
 4811 this point forward in the sequence.

4812 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

- 4813 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

4814 
$$\tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \text{nrows}(\tilde{\mathbf{C}}), \text{ncols}(\tilde{\mathbf{C}}), \\ \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\text{ind}(\tilde{\mathbf{C}}) - (\{(\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l]), \forall k, l\} \cap \text{ind}(\tilde{\mathbf{C}}))) \cup \text{ind}(\tilde{\mathbf{T}})\} \rangle.$$

4816 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
 4817 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  
 4818 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.



## 4854 C Syntax

```
4855         GrB_Info GrB_assign(GrB_Matrix      C,  
4856                             const GrB_Vector mask,  
4857                             const GrB_BinaryOp accum,  
4858                             const GrB_Vector u,  
4859                             const GrB_Index *row_indices,  
4860                             GrB_Index nrows,  
4861                             GrB_Index col_index,  
4862                             const GrB_Descriptor desc);
```

## 4863 Parameters

4864 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
4865 that may be accumulated with the result of the assign operation. On output, this  
4866 matrix holds the results of the operation.

4867 **mask** (IN) An optional “write” mask that controls which results from this operation are  
4868 stored into the specified column of the output matrix **C**. The mask dimensions  
4869 must match those of a single column of the matrix **C**. If the **GrB\_STRUCTURE**  
4870 descriptor is *not* set for the mask, the domain of the **Mask** matrix must be of type  
4871 **bool** or any of the predefined “built-in” types in Table 3.2. If the default mask  
4872 is desired (i.e., a mask that is all true with the dimensions of a column of **C**),  
4873 **GrB\_NULL** should be specified.

4874 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
4875 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
4876 specified.

4877 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a column  
4878 of **C**.

4879 **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
4880 the specified column of **C** that are to be assigned. If all elements of the column  
4881 in **C** are to be assigned in order from index 0 to **nrows** – 1, then **GrB\_ALL** should  
4882 be specified. Regardless of execution mode and return value, this array may be  
4883 manipulated by the caller after this operation returns without affecting any de-  
4884 ferred computations for this operation. If this array contains duplicate values, it  
4885 implies in assignment of more than one value to the same location which leads to  
4886 undefined results.

4887 **nrows** (IN) The number of values in **row\_indices** array. Must be equal to **size(u)**.

4888 **col\_index** (IN) The index of the column in **C** to assign. Must be in the range [0, **ncols(C)**).

4889 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
4890 should be specified. Non-default field/value pairs are listed as follows:

4891

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 **GrB\_error()** to access any error messages generated by the implementation.

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

**GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by a call to **new** (or **dup** for vector or matrix parameters).

**GrB\_INVALID\_INDEX** **col\_index** is outside the allowable range (i.e., greater than **ncols(C)**).

**GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **row\_indices** is greater than or equal to **nrows(C)**. In non-blocking mode, this can be reported as an execution error.

**GrB\_DIMENSION\_MISMATCH** **mask** size and number of rows in C are not the same, or **nrows**  $\neq$  **size(u)**.

**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 **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

**GrB\_NULL\_POINTER** Argument **row\_indices** is a NULL pointer.

## 4917 Description

4918 This variant of `GrB_assign` computes the result of assigning a subset of locations in a column of a  
 4919 GraphBLAS matrix (in a specific order) from the contents of a GraphBLAS vector:

4920  $C(:, \text{col\_index}) = u$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  $C(:, \text{col\_index}) =$   
 4921  $C(:, \text{col\_index}) \odot u$ . Taking order of `row_indices` into account, it is more explicitly written as:

$$\begin{aligned} & C(\text{row\_indices}[i], \text{col\_index}) = u(i), \forall i : 0 \leq i < \text{nrows}, \text{ or} \\ & C(\text{row\_indices}[i], \text{col\_index}) = C(\text{row\_indices}[i], \text{col\_index}) \odot u(i), \forall i : 0 \leq i < \text{nrows}. \end{aligned}$$

4923 Logically, this operation occurs in three steps:

4924     **Setup** The internal matrices, vectors and mask used in the computation are formed and their  
 4925               domains and dimensions are tested for compatibility.

4926     **Compute** The indicated computations are carried out.

4927     **Output** The result is written into the output matrix, possibly under control of a mask.

4928 Up to three argument vectors and matrices are used in this `GrB_assign` operation:

- 4929 1.  $C = \langle \mathbf{D}(C), \text{nrows}(C), \text{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 4930 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 4931 3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

4932 The argument vectors, matrix, and the accumulation operator (if provided) are tested for domain  
 4933 compatibility as follows:

- 4934 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 4935     must be from one of the pre-defined types of Table 3.2.
- 4936 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(u)$ .
- 4937 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 4938     of the accumulation operator and  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 4939     mulation operator.

4940 Two domains are compatible with each other if values from one domain can be cast to values in  
 4941 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 4942 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 4943 any compatibility rule above is violated, execution of `GrB_assign` ends and the domain mismatch  
 4944 error listed above is returned.

4945 The `col_index` parameter is checked for a valid value. The following condition must hold:

- 4946 1.  $0 \leq \text{col\_index} < \text{ncols}(C)$

4947 If the rule above is violated, execution of `GrB_assign` ends and the invalid index error listed above  
 4948 is returned.

4949 From the arguments, the internal vectors, mask, and index array used in the computation are  
 4950 formed ( $\leftarrow$  denotes copy):

4951 1. The vector,  $\tilde{\mathbf{c}}$ , is extracted from a column of  $\mathbf{C}$  as follows:

$$4952 \quad \tilde{\mathbf{c}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\mathbf{C}), \{(i, C_{ij}) \mid \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C}), j = \text{col\_index}, (i, j) \in \mathbf{ind}(\mathbf{C})\} \rangle$$

4953 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

4954 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{nrows}(\mathbf{C}), \{i, \forall i : 0 \leq i < \mathbf{nrows}(\mathbf{C})\} \rangle$ .

4955 (b) If `mask  $\neq$  GrB_NULL`,

4956 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,

4957 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

4958 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

4959 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

4960 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument `row_indices` as follows:

4961 (a) If `row_indices = GrB_ALL`, then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .

4962 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .

4963 The internal vectors, matrices, and masks are checked for dimension compatibility. The following  
 4964 conditions must hold:

4965 1.  $\mathbf{size}(\tilde{\mathbf{c}}) = \mathbf{size}(\tilde{\mathbf{m}})$

4966 2.  $\mathbf{nrows} = \mathbf{size}(\tilde{\mathbf{u}})$ .

4967 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 4968 match error listed above is returned.

4969 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 4970 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

4971 We are now ready to carry out the assign and any additional associated operations. We describe  
 4972 this in terms of two intermediate vectors:

4973 •  $\tilde{\mathbf{t}}$ : The vector holding the elements from  $\tilde{\mathbf{u}}$  in their destination locations relative to  $\tilde{\mathbf{c}}$ .

4974 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

4975 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$4976 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\tilde{\mathbf{c}}), \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{u}}(i)) \mid \forall i, 0 \leq i < \mathbf{nrows} : i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle.$$

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:

$$\begin{aligned} \mathbf{L}(\mathbf{C}) &= \{(i, j, C_{ij}) : j \neq \text{col\_index}\} \cup \\ &\quad \{(i, \text{col\_index}, \tilde{\mathbf{c}}(i)) : i \in (\mathbf{ind}(\tilde{\mathbf{c}}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \\ &\quad \{(i, \text{col\_index}, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}. \end{aligned}$$

5015 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
5016 of vector w is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
5017 exits with return value GrB\_SUCCESS and the new content of vector w is as defined above but may  
5018 not be fully computed; however, it can be used in the next GraphBLAS method call in a sequence.

#### 5019 4.3.7.4 assign: Row variant

5020 Assign the contents a vector to a subset of elements in one row of a matrix. Note that since the  
5021 output cannot be transposed, a different variant of assign is provided to assign to a column of a  
5022 matrix.

#### 5023 C Syntax

```
5024      GrB_Info GrB_assign(GrB_Matrix      C,
5025                          const GrB_Vector mask,
5026                          const GrB_BinaryOp accum,
5027                          const GrB_Vector u,
5028                          GrB_Index      row_index,
5029                          const GrB_Index *col_indices,
5030                          GrB_Index      ncols,
5031                          const GrB_Descriptor desc);
```

#### 5032 Parameters

5033 **C** (INOUT) An existing GraphBLAS Matrix. On input, the matrix provides values  
5034 that may be accumulated with the result of the assign operation. On output, this  
5035 matrix holds the results of the operation.

5036 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5037 stored into the specified row of the output matrix C. The mask dimensions must  
5038 match those of a single row of the matrix C. If the GrB\_STRUCTURE descriptor  
5039 is *not* set for the mask, the domain of the Mask matrix must be of type **bool** or  
5040 any of the predefined “built-in” types in Table 3.2. If the default mask is desired  
5041 (i.e., a mask that is all true with the dimensions of a row of C), GrB\_NULL should  
5042 be specified.

5043 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
5044 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
5045 specified.

5046 **u** (IN) The GraphBLAS vector whose contents are assigned to (a subset of) a row of  
5047 C.

5048 **row\_index** (IN) The index of the row in C to assign. Must be in the range [0, n<sub>rows</sub>(C)).

5049 **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the locations in  
5050 the specified row of **C** that are to be assigned. If all elements of the row in **C** are to  
5051 be assigned in order from index 0 to **ncols** – 1, then **GrB\_ALL** should be specified.  
5052 Regardless of execution mode and return value, this array may be manipulated by  
5053 the caller after this operation returns without affecting any deferred computations  
5054 for this operation. If this array contains duplicate values, it implies in assignment  
5055 of more than one value to the same location which leads to undefined results.

5056 **ncols** (IN) The number of values in **col\_indices** array. Must be equal to **size(u)**.

5057 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5058 should be specified. Non-default field/value pairs are listed as follows:  
5059

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> .

## 5061 Return Values

5062 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5063 blocking mode, this indicates that the compatibility tests on  
5064 dimensions and domains for the input arguments passed suc-  
5065 cessfully. Either way, output matrix **C** is ready to be used in the  
5066 next method of the sequence.

5067 **GrB\_PANIC** Unknown internal error.

5068 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
5069 opaque GraphBLAS objects (input or output) is in an invalid  
5070 state caused by a previous execution error. Call **GrB\_error()** to  
5071 access any error messages generated by the implementation.

5072 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5073 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
5074 by a call to **new** (or **dup** for vector or matrix parameters).

5075 **GrB\_INVALID\_INDEX** **row\_index** is outside the allowable range (i.e., greater than **nrows(C)**).

5076 **GrB\_INDEX\_OUT\_OF\_BOUNDS** A value in **col\_indices** is greater than or equal to **ncols(C)**. In  
5077 non-blocking mode, this can be reported as an execution error.

5078 **GrB\_DIMENSION\_MISMATCH** **mask** size and number of columns in **C** are not the same, or  
5079 **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

5208  
5209  
5210  
5211  
5212  
5213  
5214  
  
5215  
5216  
5217  
  
5218  
  
5219  
  
5220  
5221  
5222  
5223  
5224  
5225  
5226  
5227  
  
5228  
5229  
  
5230  
5231  
5232

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> .

5234 **Return Values**

5235 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
5236 blocking mode, this indicates that the compatibility tests on  
5237 dimensions and domains for the input arguments passed suc-  
5238 cessfully. Either way, output vector **w** is ready to be used in the  
5239 next method of the sequence.

5240                   GrB\_PANIC Unknown internal error.

5241           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
5242                                   opaque GraphBLAS objects (input or output) is in an invalid  
5243                                   state caused by a previous execution error. Call GrB\_error() to  
5244                                   access any error messages generated by the implementation.

5245           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

5246   GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
5247                                   by a call to new (or dup for vector parameters).

5248   GrB\_INDEX\_OUT\_OF\_BOUNDS A value in indices is greater than or equal to **size(w)**. In non-  
5249                                   blocking mode, this can be reported as an execution error.

5250   GrB\_DIMENSION\_MISMATCH mask and w dimensions are incompatible, or nindices is not less  
5251                                   than **size(w)**.

5252   GrB\_DOMAIN\_MISMATCH The domains of the vector and scalar are incompatible with each  
5253                                   other or the corresponding domains of the accumulation oper-  
5254                                   ator, or the mask's domain is not compatible with bool (in the  
5255                                   case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

5256           GrB\_NULL\_POINTER Argument indices is a NULL pointer.

## 5257 Description

5258 This variant of GrB\_assign computes the result of assigning a constant scalar value – either val or  
5259 s – to locations in a destination GraphBLAS vector. Either w(indices) = val or w(indices) = s is  
5260 performed. If an optional binary accumulation operator ( $\odot$ ) is provided, then either w(indices) =  
5261 w(indices)  $\odot$  val or w(indices) = w(indices)  $\odot$  s is performed. More explicitly, if a non-opaque value  
5262 val is provided:

$$5263 \quad \begin{aligned} &w(\text{indices}[i]) = \text{val}, \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ &w(\text{indices}[i]) = w(\text{indices}[i]) \odot \text{val}, \forall i : 0 \leq i < \text{nindices}. \end{aligned}$$

5264 Correspondingly, if a GrB\_Scalar s is provided:

$$5265 \quad \begin{aligned} &w(\text{indices}[i]) = s, \forall i : 0 \leq i < \text{nindices}, \text{ or} \\ &w(\text{indices}[i]) = w(\text{indices}[i]) \odot s, \forall i : 0 \leq i < \text{nindices}. \end{aligned}$$

5266 Logically, this operation occurs in three steps:

5267   **Setup** The internal vectors and mask used in the computation are formed and their domains  
5268                   and dimensions are tested for compatibility.

5269 **Compute** The indicated computations are carried out.

5270 **Output** The result is written into the output vector, possibly under control of a mask.

5271 Up to two argument vectors are used in the GrB\_assign operation:

- 5272 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 5273 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)

5274 The argument scalar, vectors, and the accumulation operator (if provided) are tested for domain  
5275 compatibility as follows:

- 5276 1. If  $\mathbf{mask}$  is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\mathbf{mask})$   
5277 must be from one of the pre-defined types of Table 3.2.
- 5278 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  
5279 method.
- 5280 3. If  $\mathbf{accum}$  is not GrB\_NULL, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
5281 of the accumulation operator.
- 5282 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  
5283 method, must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.

5284 Two domains are compatible with each other if values from one domain can be cast to values in  
5285 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5286 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5287 any compatibility rule above is violated, execution of GrB\_assign ends and the domain mismatch  
5288 error listed above is returned.

5289 From the arguments, the internal vectors, mask and index array used in the computation are formed  
5290 ( $\leftarrow$  denotes copy):

- 5291 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5292 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument  $\mathbf{mask}$  as follows:
  - 5293 (a) If  $\mathbf{mask} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 5294 (b) If  $\mathbf{mask} \neq \text{GrB\_NULL}$ ,
    - 5295 i. If desc[GrB\_MASK].GrB\_STRUCTURE is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 5296 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\text{bool})\mathbf{mask}(i) = \text{true}\} \rangle$ .
  - 5297 (c) If desc[GrB\_MASK].GrB\_COMP is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 5298 3. Scalar  $\tilde{s} \leftarrow \mathbf{s}$  (GrB\_Scalar version only).
- 5299 4. The internal index array,  $\tilde{\mathbf{I}}$ , is computed from argument indices as follows:
  - 5300 (a) If  $\mathbf{indices} = \text{GrB\_ALL}$ , then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nindices}$ .
  - 5301 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \mathbf{indices}[i], \forall i : 0 \leq i < \mathbf{nindices}$ .

5302 The internal vector and mask are checked for dimension compatibility. The following conditions  
 5303 must hold:

- 5304 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 5305 2.  $0 \leq \mathbf{nindices} \leq \mathbf{size}(\tilde{\mathbf{w}})$ .

5306 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 5307 match error listed above is returned.

5308 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 5309 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5310 We are now ready to carry out the assign and any additional associated operations. We describe  
 5311 this in terms of two intermediate vectors:

- 5312 •  $\tilde{\mathbf{t}}$ : The vector holding the copies of the scalar, either `val` or  $\tilde{s}$ , in their destination locations  
 5313 relative to  $\tilde{\mathbf{w}}$ .
- 5314 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5315 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows. If a non-opaque scalar `val` is provided:

$$5316 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{val}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5317 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$5318 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \{(\tilde{\mathbf{I}}[i], \mathbf{val}(\tilde{s})) \mid \forall i, 0 \leq i < \mathbf{nindices}\} \rangle.$$

5319 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$5320 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\tilde{s}), \mathbf{size}(\tilde{\mathbf{w}}), \emptyset \rangle.$$

5321 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  
 5322 is not in the range  $[0, \mathbf{size}(\tilde{\mathbf{w}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds  
 5323 error listed above is generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a  
 5324 sequence-terminating `GrB_wait()` is called. Regardless, the result vector,  $\mathbf{w}$ , is invalid from this  
 5325 point forward in the sequence.

5326 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows:

- 5327 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}}$  is defined as

$$5328 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i), \forall i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5329 The above expression defines the structure of vector  $\tilde{\mathbf{z}}$  as follows: We start with the structure  
 5330 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  
 5331 assigned ( $\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}})$ ). Finally, we add the structure of  $\tilde{\mathbf{t}}$  ( $\mathbf{ind}(\tilde{\mathbf{t}})$ ).

5332 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5333 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 5334 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\{\tilde{\mathbf{I}}[k], \forall k\} \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 5335 \quad & \\ 5336 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in \mathbf{ind}(\tilde{\mathbf{t}}), \end{aligned}$$

5337 where the difference operator refers to set difference. We note that in this case of assigning  
 5338 a constant,  $\{\tilde{\mathbf{I}}[k], \forall k\}$  and  $\mathbf{ind}(\tilde{\mathbf{t}})$  are identical.

- 5339 • If **accum** is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5340 \quad \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5341 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5342 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 5343 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 5344 \quad & \\ 5345 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 5346 \quad & \\ 5347 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \end{aligned}$$

5348 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5349 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 5350 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 5351 mask which acts as a “write mask”.

- 5352 • If **desc[GrB\_OUTP].GrB\_REPLACE** is set, then any values in  $\mathbf{w}$  on input to this operation are  
 5353 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5354 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 5355 • If **desc[GrB\_OUTP].GrB\_REPLACE** is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
 5356 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
 5357 mask are unchanged:

$$5358 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\mathbf{ind}(\mathbf{w}) \cap \mathbf{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

5359 In **GrB\_BLOCKING** mode, the method exits with return value **GrB\_SUCCESS** and the new content  
 5360 of vector  $\mathbf{w}$  is as defined above and fully computed. In **GrB\_NONBLOCKING** mode, the method  
 5361 exits with return value **GrB\_SUCCESS** and the new content of vector  $\mathbf{w}$  is as defined above but  
 5362 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 5363 sequence.

#### 5364 4.3.7.6 assign: Constant matrix variant

5365 Assign the same value to a specified subset of matrix elements. With the use of **GrB\_ALL**, the  
 5366 entire destination matrix can be filled with the constant.



## 5367 C Syntax

```

5368         GrB_Info GrB_assign(GrB_Matrix      C,
5369                             const GrB_Matrix Mask,
5370                             const GrB_BinaryOp accum,
5371                             <type>          val,
5372                             const GrB_Index  *row_indices,
5373                             GrB_Index        nrows,
5374                             const GrB_Index  *col_indices,
5375                             GrB_Index        ncols,
5376                             const GrB_Descriptor desc);

```

```

5377         GrB_Info GrB_assign(GrB_Matrix      C,
5378                             const GrB_Matrix Mask,
5379                             const GrB_BinaryOp accum,
5380                             const GrB_Scalar  s,
5381                             const GrB_Index  *row_indices,
5382                             GrB_Index        nrows,
5383                             const GrB_Index  *col_indices,
5384                             GrB_Index        ncols,
5385                             const GrB_Descriptor desc);

```

## 5386 Parameters

5387     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
5388     that may be accumulated with the result of the assign operation. On output, the  
5389     matrix holds the results of the operation.

5390     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
5391     stored into the output matrix C. The mask dimensions must match those of the  
5392     matrix C. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5393     of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
5394     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5395     dimensions of C), **GrB\_NULL** should be specified.

5396     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
5397     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5398     specified.

5399     **val** (IN) Scalar value to assign to (a subset of) C.

5400     **s** (IN) Scalar value to assign to (a subset of) C.

5401     **row\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the rows of C  
5402     that are assigned. If all rows of C are to be assigned in order from 0 to **nrows** − 1,  
5403     then **GrB\_ALL** can be specified. Regardless of execution mode and return value,  
5404     this array may be manipulated by the caller after this operation returns without

5405 affecting any deferred computations for this operation. Unlike other variants, if  
 5406 there are duplicated values in this array the result is still defined.

5407 **nrows** (IN) The number of values in **row\_indices** array. Must be in the range:  $[0, \mathbf{nrows}(C)]$ .  
 5408 If **nrows** is zero, the operation becomes a NO-OP.

5409 **col\_indices** (IN) Pointer to the ordered set (array) of indices corresponding to the columns of **C**  
 5410 that are assigned. If all columns of **C** are to be assigned in order from 0 to  $\mathbf{ncols} - 1$ ,  
 5411 then **GrB\_ALL** should be specified. Regardless of execution mode and return value,  
 5412 this array may be manipulated by the caller after this operation returns without  
 5413 affecting any deferred computations for this operation. Unlike other variants, if  
 5414 there are duplicated values in this array the result is still defined.

5415 **ncols** (IN) The number of values in **col\_indices** array. Must be in the range:  $[0, \mathbf{ncols}(C)]$ .  
 5416 If **ncols** is zero, the operation becomes a NO-OP.

5417 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 5418 should be specified. Non-default field/value pairs are listed as follows:  
 5419

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> .

## 5421 Return Values

5422 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 5423 blocking mode, this indicates that the compatibility tests on  
 5424 dimensions and domains for the input arguments passed suc-  
 5425 cessfully. Either way, output matrix **C** is ready to be used in the  
 5426 next method of the sequence.

5427 **GrB\_PANIC** Unknown internal error.

5428 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the  
 5429 opaque GraphBLAS objects (input or output) is in an invalid  
 5430 state caused by a previous execution error. Call **GrB\_error()** to  
 5431 access any error messages generated by the implementation.

5432 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

5433 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized  
 5434 by a call to **new** (or **dup** for vector parameters).

5435 GrB\_INDEX\_OUT\_OF\_BOUNDS A value in `row_indices` is greater than or equal to `nrows(C)`, or  
5436 a value in `col_indices` is greater than or equal to `ncols(C)`. In  
5437 non-blocking mode, this can be reported as an execution error.

5438 GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, `nrows` is not less than  
5439 `nrows(C)`, or `ncols` is not less than `ncols(C)`.

5440 GrB\_DOMAIN\_MISMATCH The domains of the matrix and scalar are incompatible with  
5441 each other or the corresponding domains of the accumulation  
5442 operator, or the mask's domain is not compatible with `bool` (in  
5443 the case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

5444 GrB\_NULL\_POINTER Either argument `row_indices` is a NULL pointer, argument `col_indices`  
5445 is a NULL pointer, or both.

## 5446 Description

5447 This variant of `GrB_assign` computes the result of assigning a constant scalar value – either `val` or  
5448 `s` – to locations in a destination GraphBLAS matrix: Either `C(row_indices, col_indices) = val`  
5449 or `C(row_indices, col_indices) = s` is performed. If an optional binary accumulation operator  
5450 ( $\odot$ ) is provided, then either `C(row_indices, col_indices) = C(row_indices, col_indices)  $\odot$  val` or  
5451 `C(row_indices, col_indices) = C(row_indices, col_indices)  $\odot$  s` is performed. More explicitly, if a  
5452 non-opaque value `val` is provided:

$$\begin{aligned}
 &C(\text{row\_indices}[i], \text{col\_indices}[j]) = \text{val}, \text{ or} \\
 5453 \quad &C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot \text{val} \\
 &\quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

5454 Correspondingly, if a `GrB_Scalar s` is provided:

$$\begin{aligned}
 &C(\text{row\_indices}[i], \text{col\_indices}[j]) = s, \text{ or} \\
 5455 \quad &C(\text{row\_indices}[i], \text{col\_indices}[j]) = C(\text{row\_indices}[i], \text{col\_indices}[j]) \odot s \\
 &\quad \forall (i, j) : 0 \leq i < \text{nrows}, 0 \leq j < \text{ncols}
 \end{aligned}$$

5456 Logically, this operation occurs in three steps:

5457     Setup The internal vectors and mask used in the computation are formed and their domains  
5458     and dimensions are tested for compatibility.

5459     Compute The indicated computations are carried out.

5460     Output The result is written into the output matrix, possibly under control of a mask.

5461 Up to two argument matrices are used in the `GrB_assign` operation:

- 5462 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

5463 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)

5464 The argument scalar, matrices, and the accumulation operator (if provided) are tested for domain  
5465 compatibility as follows:

- 5466 1. If  $\text{Mask}$  is not  $\text{GrB\_NULL}$ , and  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is not set, then  $\mathbf{D}(\text{Mask})$   
5467 must be from one of the pre-defined types of Table 3.2.
- 5468 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  
5469 method.
- 5470 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  $\mathbf{D}_{out}(\text{accum})$   
5471 of the accumulation operator.
- 5472 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  
5473 method, must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

5474 Two domains are compatible with each other if values from one domain can be cast to values in  
5475 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5476 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5477 any compatibility rule above is violated, execution of  $\text{GrB\_assign}$  ends and the domain mismatch  
5478 error listed above is returned.

5479 From the arguments, the internal matrices, index arrays, and mask used in the computation are  
5480 formed ( $\leftarrow$  denotes copy):

- 5481 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 5482 2. Two-dimensional mask  $\tilde{\mathbf{M}}$  is computed from argument  $\text{Mask}$  as follows:
  - 5483 (a) If  $\text{Mask} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{C}), \mathbf{ncols}(\text{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\text{C}), 0 \leq$   
5484  $j < \mathbf{ncols}(\text{C})\} \rangle$ .
  - 5485 (b) If  $\text{Mask} \neq \text{GrB\_NULL}$ ,
    - 5486 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
5487  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
    - 5488 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
5489  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
  - 5490 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 5491 3. Scalar  $\tilde{s} \leftarrow s$  ( $\text{GrB\_Scalar}$  version only).
- 5492 4. The internal row index array,  $\tilde{\mathbf{I}}$ , is computed from argument  $\text{row\_indices}$  as follows:
  - 5493 (a) If  $\text{row\_indices} = \text{GrB\_ALL}$ , then  $\tilde{\mathbf{I}}[i] = i, \forall i : 0 \leq i < \mathbf{nrows}$ .
  - 5494 (b) Otherwise,  $\tilde{\mathbf{I}}[i] = \text{row\_indices}[i], \forall i : 0 \leq i < \mathbf{nrows}$ .
- 5495 5. The internal column index array,  $\tilde{\mathbf{J}}$ , is computed from argument  $\text{col\_indices}$  as follows:

- 5496 (a) If `col_indices = GrB_ALL`, then  $\tilde{\mathbf{J}}[j] = j, \forall j : 0 \leq j < \text{ncols}$ .  
 5497 (b) Otherwise,  $\tilde{\mathbf{J}}[j] = \text{col\_indices}[j], \forall j : 0 \leq j < \text{ncols}$ .

5498 The internal matrix and mask are checked for dimension compatibility. The following conditions  
 5499 must hold:

- 5500 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .  
 5501 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .  
 5502 3.  $0 \leq \mathbf{nrows} \leq \mathbf{nrows}(\tilde{\mathbf{C}})$ .  
 5503 4.  $0 \leq \mathbf{ncols} \leq \mathbf{ncols}(\tilde{\mathbf{C}})$ .

5504 If any compatibility rule above is violated, execution of `GrB_assign` ends and the dimension mis-  
 5505 match error listed above is returned.

5506 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 5507 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5508 We are now ready to carry out the assign and any additional associated operations. We describe  
 5509 this in terms of two intermediate matrices:

- 5510 •  $\tilde{\mathbf{T}}$ : The matrix holding the copies of the scalar, either `val` or  $\tilde{s}$ , in their destination locations  
 5511 relative to  $\tilde{\mathbf{C}}$ .
- 5512 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5513 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows. If a non-opaque scalar `val` is provided:

$$5514 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\text{val}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5515 Correspondingly, if a non-empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 1$ ):

$$5516 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ \{(\tilde{\mathbf{I}}[i], \tilde{\mathbf{J}}[j], \text{val}(\tilde{s})) \mid (i, j), 0 \leq i < \mathbf{nrows}, 0 \leq j < \mathbf{ncols}\} \rangle.$$

5517 Finally, if an empty `GrB_Scalar`  $\tilde{s}$  is provided (i.e.,  $\mathbf{size}(\tilde{s}) = 0$ ):

$$5518 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\tilde{s}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \emptyset \rangle.$$

5519 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  
 5520 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  
 5521  $[0, \mathbf{ncols}(\tilde{\mathbf{C}}))$ , the execution of `GrB_assign` ends and the index out-of-bounds error listed above is  
 5522 generated. In `GrB_NONBLOCKING` mode, the error can be deferred until a sequence-terminating  
 5523 `GrB_wait()` is called. Regardless, the result matrix  $\mathbf{C}$  is invalid from this point forward in the  
 5524 sequence.

5525 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows:

5526 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}}$  is defined as

$$5527 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}(\mathbf{C}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \\ 5528 \quad \{(i, j, Z_{ij}) \mid \forall (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l], \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5529 The above expression defines the structure of matrix  $\tilde{\mathbf{Z}}$  as follows: We start with the structure  
5530 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  
5531 assigned ( $\{\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l], \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}})$ ). Finally, we add the structure of  $\tilde{\mathbf{T}}$  ( $\mathbf{ind}(\tilde{\mathbf{T}})$ ).

5532 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5533 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5534 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\{\tilde{\mathbf{I}}[k], \tilde{\mathbf{J}}[l], \forall k, l\} \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5535 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in \mathbf{ind}(\tilde{\mathbf{T}}), \\ 5536$$

5537 where the difference operator refers to set difference. We note that, in this particular case of  
5538 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.

5539 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5540 \quad \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

5541 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5542 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$5543 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 5544 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5545 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5546 \\ 5547$$

5548 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5549 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5550 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5551 mask which acts as a “write mask”.

5552 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
5553 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5554 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5555 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
5556 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
5557 mask are unchanged:

$$5558 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5559 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
5560 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
5561 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
5562 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5563 sequence.

### 5564 4.3.8 apply: Apply a function to the elements of an object

5565 Computes the transformation of the values of the elements of a vector or a matrix using a unary  
5566 function, or a binary function where one argument is bound to a scalar.

#### 5567 4.3.8.1 apply: Vector variant

5568 Computes the transformation of the values of the elements of a vector using a unary function.

### 5569 C Syntax

```
5570      GrB_Info GrB_apply(GrB_Vector      w,  
5571                        const GrB_Vector mask,  
5572                        const GrB_BinaryOp accum,  
5573                        const GrB_UnaryOp op,  
5574                        const GrB_Vector u,  
5575                        const GrB_Descriptor desc);
```

### 5576 Parameters

5577 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5578 that may be accumulated with the result of the apply operation. On output, this  
5579 vector holds the results of the operation.

5580 **mask** (IN) An optional “write” mask that controls which results from this operation are  
5581 stored into the output vector **w**. The mask dimensions must match those of the  
5582 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5583 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5584 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5585 dimensions of **w**), **GrB\_NULL** should be specified.

5586 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5587 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5588 specified.

5589 **op** (IN) A unary operator applied to each element of input vector **u**.

5590 **u** (IN) The GraphBLAS vector to which the unary function is applied.

5591 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
5592 should be specified. Non-default field/value pairs are listed as follows:  
5593

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.



5622 **Output** The result is written into the output vector, possibly under control of a mask.

5623 Up to three argument vectors are used in this **GrB\_apply** operation:

- 5624 1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 5625 2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 5626 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

5627 The argument vectors, unary operator and the accumulation operator (if provided) are tested for  
5628 domain compatibility as follows:

- 5629 1. If **mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{mask})$   
5630 must be from one of the pre-defined types of Table 3.2.
- 5631 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the unary operator.
- 5632 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
5633 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the unary operator must be compatible with  
5634  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 5635 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in}(\mathbf{op})$ .

5636 Two domains are compatible with each other if values from one domain can be cast to values in  
5637 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5638 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5639 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
5640 error listed above is returned.

5641 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
5642 denotes copy):

- 5643 1. Vector  $\widetilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 5644 2. One-dimensional mask,  $\widetilde{\mathbf{m}}$ , is computed from argument **mask** as follows:
  - 5645 (a) If **mask** = **GrB\_NULL**, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 5646 (b) If **mask**  $\neq$  **GrB\_NULL**,
    - 5647 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
    - 5648 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
  - 5649 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 5650 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5651 The internal vectors and masks are checked for dimension compatibility. The following conditions  
5652 must hold:

5653 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

5654 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

5655 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
5656 error listed above is returned.

5657 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
5658 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

5659 We are now ready to carry out the apply and any additional associated operations. We describe  
5660 this in terms of two intermediate vectors:

- 5661 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the unary operator to the input vector  $\tilde{\mathbf{u}}$ .
- 5662 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5663 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$5664 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i))) \mid \forall i \in \text{ind}(\tilde{\mathbf{u}})\} \rangle,$$

5665 where  $f = \mathbf{f}(\text{op})$ .

5666 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 5667 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 5668 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$5669 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5670 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
5671 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 5672 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 5673 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 5674 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 5675 \quad & \\ 5676 \end{aligned}$$

5677 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

5678 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
5679 using what is called a *standard vector mask and replace*. This is carried out under control of the  
5680 mask which acts as a “write mask”.

- 5681 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
5682 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$5683 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

- 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.

5717 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 5718 should be specified. Non-default field/value pairs are listed as follows:

5719

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.

5720

## 5721 Return Values

5722 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
 5723 blocking mode, this indicates that the compatibility tests on  
 5724 dimensions and domains for the input arguments passed suc-  
 5725 cessfully. Either way, output matrix C is ready to be used in the  
 5726 next method of the sequence.

5727 GrB\_PANIC Unknown internal error.

5728 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
 5729 opaque GraphBLAS objects (input or output) is in an invalid  
 5730 state caused by a previous execution error. Call GrB\_error() to  
 5731 access any error messages generated by the implementation.

5732 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

5733 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
 5734 by a call to new (or Matrix\_dup for matrix parameters).

5735 GrB\_DIMENSION\_MISMATCH Mask and C dimensions are incompatible, nrow  $\neq$  nrow(C), or  
 5736 ncol  $\neq$  ncol(C).

5737 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
 5738 corresponding domains of the accumulation operator or unary  
 5739 function, or the mask's domain is not compatible with bool (in  
 5740 the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 5741 Description

5742 This variant of GrB\_apply computes the result of applying a unary function to the elements of a  
 5743 GraphBLAS matrix:  $C = f(A)$ ; or, if an optional binary accumulation operator ( $\odot$ ) is provided,  
 5744  $C = C \odot f(A)$ .

5745 Logically, this operation occurs in three steps:

5746     **Setup** The internal matrices and mask used in the computation are formed and their domains  
5747             and dimensions are tested for compatibility.

5748     **Compute** The indicated computations are carried out.

5749     **Output** The result is written into the output matrix, possibly under control of a mask.

5750 Up to three argument matrices are used in the `GrB_apply` operation:

- 5751     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 5752     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)
- 5753     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

5754 The argument matrices, unary operator and the accumulation operator (if provided) are tested for  
5755 domain compatibility as follows:

- 5756     1. If `Mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{Mask})$   
5757         must be from one of the pre-defined types of Table 3.2.
- 5758     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the unary operator.
- 5759     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
5760         of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the unary operator must be compatible with  
5761          $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5762     4.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in}(\text{op})$  of the unary operator.

5763 Two domains are compatible with each other if values from one domain can be cast to values in  
5764 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
5765 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
5766 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
5767 error listed above is returned.

5768 From the argument matrices, the internal matrices, mask, and index arrays used in the computation  
5769 are formed ( $\leftarrow$  denotes copy):

- 5770     1. Matrix  $\tilde{C} \leftarrow C$ .
- 5771     2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument `Mask` as follows:
  - 5772         (a) If `Mask` = `GrB_NULL`, then  $\tilde{M} = \langle \mathbf{nrows}(C), \mathbf{ncols}(C), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(C), 0 \leq$   
5773              $j < \mathbf{ncols}(C)\} \rangle$ .
  - 5774         (b) If `Mask`  $\neq$  `GrB_NULL`,
    - 5775             i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{M} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
5776                  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,

5777           ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
5778            $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\}\rangle.$   
5779       (c) If  $\mathbf{desc}[\mathbf{GrB\_MASK}].\mathbf{GrB\_COMP}$  is set, then  $\tilde{\mathbf{M}} \leftarrow \neg\tilde{\mathbf{M}}.$   
5780   3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}.$

5781 The internal matrices and mask are checked for dimension compatibility. The following conditions  
5782 must hold:

- 5783   1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}}).$
- 5784   2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}}).$
- 5785   3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}}).$
- 5786   4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}}).$

5787 If any compatibility rule above is violated, execution of  $\mathbf{GrB\_apply}$  ends and the dimension mismatch  
5788 error listed above is returned.

5789 From this point forward, in  $\mathbf{GrB\_NONBLOCKING}$  mode, the method can optionally exit with  
5790  $\mathbf{GrB\_SUCCESS}$  return code and defer any computation and/or execution error codes.

5791 We are now ready to carry out the apply and any additional associated operations. We describe  
5792 this in terms of two intermediate matrices:

- 5793   •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the unary operator to the input matrix  $\tilde{\mathbf{A}}.$
- 5794   •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

5795 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$5796 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\}\rangle,$$

5797 where  $f = \mathbf{f}(\mathbf{op}).$

5798 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 5799   • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}.$
- 5800   • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$5801 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

5802 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
5803 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}.$

$$\begin{aligned} 5804 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 5805 \\ 5806 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 5807 \\ 5808 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \end{aligned}$$

5809 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5810 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
5811 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
5812 mask which acts as a “write mask”.

- 5813 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
5814 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$5815 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 5816 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
5817 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
5818 mask are unchanged:

$$5819 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

5820 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
5821 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
5822 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
5823 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
5824 sequence.

#### 5825 4.3.8.3 apply: Vector-BinaryOp variants

5826 Computes the transformation of the values of the stored elements of a vector using a binary operator  
5827 and a scalar value. In the *bind-first* variant, the specified scalar value is passed as the first argument  
5828 to the binary operator and stored elements of the vector are passed as the second argument. In the  
5829 *bind-second* variant, the elements of the vector are passed as the first argument and the specified  
5830 scalar value is passed as the second argument. The scalar can be passed either as a non-opaque  
5831 variable or as a GrB\_Scalar object.

#### 5832 C Syntax

```
5833 // bind-first + scalar value
5834 GrB_Info GrB_apply(GrB_Vector          w,
5835                   const GrB_Vector      mask,
5836                   const GrB_BinaryOp    accum,
5837                   const GrB_BinaryOp    op,
5838                   <type>                val,
5839                   const GrB_Vector      u,
5840                   const GrB_Descriptor   desc);
```

```
5841 // bind-first + GraphBLAS scalar
5842 GrB_Info GrB_apply(GrB_Vector          w,
5843                   const GrB_Vector      mask,
```

```

5844         const GrB_BinaryOp      accum,
5845         const GrB_BinaryOp      op,
5846         const GrB_Scalar        s,
5847         const GrB_Vector        u,
5848         const GrB_Descriptor    desc);

5849     // bind-second + scalar value
5850     GrB_Info GrB_apply(GrB_Vector      w,
5851                       const GrB_Vector mask,
5852                       const GrB_BinaryOp accum,
5853                       const GrB_BinaryOp op,
5854                       const GrB_Vector u,
5855                       <type>         val,
5856                       const GrB_Descriptor desc);

5857     // bind-second + GraphBLAS scalar
5858     GrB_Info GrB_apply(GrB_Vector      w,
5859                       const GrB_Vector mask,
5860                       const GrB_BinaryOp accum,
5861                       const GrB_BinaryOp op,
5862                       const GrB_Vector u,
5863                       const GrB_Scalar s,
5864                       const GrB_Descriptor desc);

```

## 5865 Parameters

5866     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
5867     that may be accumulated with the result of the apply operation. On output, this  
5868     vector holds the results of the operation.

5869     **mask** (IN) An optional “write” mask that controls which results from this operation are  
5870     stored into the output vector **w**. The mask dimensions must match those of the  
5871     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
5872     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
5873     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
5874     dimensions of **w**), **GrB\_NULL** should be specified.

5875     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
5876     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
5877     specified.

5878     **op** (IN) A binary operator applied to each element of input vector, **u**, and the scalar  
5879     value, **val**.

5880     **u** (IN) The GraphBLAS vector whose elements are passed to the binary operator as  
5881     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
5882     argument in the *bind-second* variant.



5883        **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
 5884        argument in the *bind-first* variant, or the right-hand (second) argument in the  
 5885        *bind-second* variant.

5886        **s** (IN) A GraphBLAS scalar that is passed to the binary operator as the left-hand  
 5887        (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
 5888        the *bind-second* variant. It must not be empty.

5889        **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
 5890        should be specified. Non-default field/value pairs are listed as follows:  
 5891

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> .

## 5893 Return Values

5894        **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
 5895        blocking mode, this indicates that the compatibility tests on di-  
 5896        mensions and domains for the input arguments passed successfully.  
 5897        Either way, output vector **w** is ready to be used in the next method  
 5898        of the sequence.

5899        **GrB\_PANIC** Unknown internal error.

5900        **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
 5901        GraphBLAS objects (input or output) is in an invalid state caused  
 5902        by a previous execution error. Call **GrB\_error()** to access any error  
 5903        messages generated by the implementation.

5904        **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

5905        **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 5906        a call to **new** (or **dup** for vector parameters).

5907        **GrB\_DIMENSION\_MISMATCH** **mask**, **w** and/or **u** dimensions are incompatible.

5908        **GrB\_DOMAIN\_MISMATCH** The domains of the various vectors and scalar are incompatible with  
 5909        the corresponding domains of the binary operator or accumulation  
 5910        operator, or the mask's domain is not compatible with **bool** (in the  
 5911        case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

5912        **GrB\_EMPTY\_OBJECT** The **GrB\_Scalar s** used in the call is empty (**nvals(s) = 0**) and  
 5913        therefore a value cannot be passed to the binary operator.

## 5914 Description

5915 This variant of `GrB_apply` computes the result of applying a binary operator to the elements of a  
 5916 GraphBLAS vector each composed with a scalar constant, either `val` or `s`:

5917                   bind-first:      $w = f(\text{val}, u)$  or  $w = f(s, u)$

5918                   bind-second:    $w = f(u, \text{val})$  or  $w = f(u, s)$ ,

5919 or if an optional binary accumulation operator ( $\odot$ ) is provided:

5920                   bind-first:      $w = w \odot f(\text{val}, u)$  or  $w = w \odot f(s, u)$

5921                   bind-second:    $w = w \odot f(u, \text{val})$  or  $w = w \odot f(u, s)$ .

5922 Logically, this operation occurs in three steps:

5923     **Setup** The internal vectors and mask used in the computation are formed and their domains  
 5924             and dimensions are tested for compatibility.

5925     **Compute** The indicated computations are carried out.

5926     **Output** The result is written into the output vector, possibly under control of a mask.

5927 Up to three argument vectors are used in this `GrB_apply` operation:

- 5928     1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 5929     2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 5930     3.  $u = \langle \mathbf{D}(u), \text{size}(u), \mathbf{L}(u) = \{(i, u_i)\} \rangle$

5931 The argument scalar, vectors, binary operator and the accumulation operator (if provided) are  
 5932 tested for domain compatibility as follows:

- 5933     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 5934         must be from one of the pre-defined types of Table 3.2.
- 5935     2.  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the binary operator.
- 5936     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 5937         of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the binary operator must be compatible with  
 5938          $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 5939     4.  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.
- 5940     5. If bind-first:  
 5941         (a)  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the binary operator.

5942 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$   
 5943 of the binary operator.

5944 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the  
 5945 binary operator.

5946 6. If `bind-second`:

5947 (a)  $\mathbf{D}(u)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the binary operator.

5948 (b) If the non-opaque scalar  $\text{val}$  is provided, then  $\mathbf{D}(\text{val})$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
 5949 of the binary operator.

5950 (c) If the `GrB_Scalar`  $s$  is provided, then  $\mathbf{D}(s)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$  of the  
 5951 binary operator.

5952 Two domains are compatible with each other if values from one domain can be cast to values in  
 5953 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 5954 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 5955 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
 5956 error listed above is returned.

5957 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 5958 denotes copy):

5959 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .

5960 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:

5961 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .

5962 (b) If `mask  $\neq$  GrB_NULL`,

5963 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,

5964 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .

5965 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .

5966 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

5967 4. Scalar  $\tilde{s} \leftarrow s$  (`GraphBLAS scalar case`).

5968 The internal vectors and masks are checked for dimension compatibility. The following conditions  
 5969 must hold:

5970 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

5971 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

5972 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
 5973 error listed above is returned.

5974 From this point forward, in GrB\_NONBLOCKING mode, the method can optionally exit with  
 5975 GrB\_SUCCESS return code and defer any computation and/or execution error codes.

5976 If an empty GrB\_Scalar  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code GrB\_EMPTY\_OBJECT.  
 5977 If a non-empty GrB\_Scalar,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
 5978  $\mathbf{val}$  with the same domain as  $\tilde{s}$  and set  $\mathbf{val} = \mathbf{val}(\tilde{s})$ .

5979 We are now ready to carry out the apply and any additional associated operations. We describe  
 5980 this in terms of two intermediate vectors:

- 5981 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the binary operator to the input vector  $\tilde{\mathbf{u}}$ .
- 5982 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

5983 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as one of the following:

$$\begin{aligned} \text{5984} \quad \text{bind-first:} \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\mathbf{val}, \tilde{\mathbf{u}}(i))) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \\ \text{5985} \quad \text{bind-second:} \quad \tilde{\mathbf{t}} &= \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{u}}), \{(i, f(\tilde{\mathbf{u}}(i), \mathbf{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle, \end{aligned}$$

5986 where  $f = \mathbf{f}(\mathbf{op})$ .

5987 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 5988 • If  $\mathbf{accum} = \text{GrB\_NULL}$ , then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 5989 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$\text{5990} \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

5991 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
 5992 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} \text{5993} \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ \text{5994} \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ \text{5995} \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ \text{5996} \quad & \\ \text{5997} \quad & \end{aligned}$$

5998 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

5999 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
 6000 using what is called a *standard vector mask and replace*. This is carried out under control of the  
 6001 mask which acts as a “write mask”.

- 6002 • If  $\text{desc}[\text{GrB\_OUTP}].\text{GrB\_REPLACE}$  is set, then any values in  $\mathbf{w}$  on input to this operation are  
 6003 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$\text{6004} \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\mathbf{ind}(\tilde{\mathbf{z}}) \cap \mathbf{ind}(\tilde{\mathbf{m}}))\}.$$

- 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

```

6022 // bind-first + scalar value
6023 GrB_Info GrB_apply(GrB_Matrix      C,
6024                   const GrB_Matrix Mask,
6025                   const GrB_BinaryOp accum,
6026                   const GrB_BinaryOp op,
6027                   <type>            val,
6028                   const GrB_Matrix  A,
6029                   const GrB_Descriptor desc);

6030 // bind-first + GraphBLAS scalar
6031 GrB_Info GrB_apply(GrB_Matrix      C,
6032                   const GrB_Matrix Mask,
6033                   const GrB_BinaryOp accum,
6034                   const GrB_BinaryOp op,
6035                   const GrB_Scalar  s,
6036                   const GrB_Matrix  A,
6037                   const GrB_Descriptor desc);

6038 // bind-second + scalar value
6039 GrB_Info GrB_apply(GrB_Matrix      C,
6040                   const GrB_Matrix Mask,
```

```

6041         const GrB_BinaryOp    accum,
6042         const GrB_BinaryOp    op,
6043         const GrB_Matrix      A,
6044         <type>                val,
6045         const GrB_Descriptor   desc);

6046 // bind-second + GraphBLAS scalar
6047 GrB_Info GrB_apply(GrB_Matrix      C,
6048                   const GrB_Matrix Mask,
6049                   const GrB_BinaryOp accum,
6050                   const GrB_BinaryOp op,
6051                   const GrB_Matrix  A,
6052                   const GrB_Scalar  s,
6053                   const GrB_Descriptor desc);

```

## 6054 Parameters

6055     **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6056     that may be accumulated with the result of the apply operation. On output, the  
6057     matrix holds the results of the operation.

6058     **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6059     stored into the output matrix C. The mask dimensions must match those of the  
6060     matrix C. If the `GrB_STRUCTURE` descriptor is *not* set for the mask, the domain  
6061     of the **Mask** matrix must be of type `bool` or any of the predefined “built-in” types  
6062     in Table 3.2. If the default mask is desired (i.e., a mask that is all `true` with the  
6063     dimensions of C), `GrB_NULL` should be specified.

6064     **accum** (IN) An optional binary operator used for accumulating entries into existing C  
6065     entries. If assignment rather than accumulation is desired, `GrB_NULL` should be  
6066     specified.

6067     **op** (IN) A binary operator applied to each element of input matrix, A, with the element  
6068     of the input matrix used as the left-hand argument, and the scalar value, `val`, used  
6069     as the right-hand argument.

6070     **A** (IN) The GraphBLAS matrix whose elements are passed to the binary operator as  
6071     the right-hand (second) argument in the *bind-first* variant, or the left-hand (first)  
6072     argument in the *bind-second* variant.

6073     **val** (IN) Scalar value that is passed to the binary operator as the left-hand (first)  
6074     argument in the *bind-first* variant, or the right-hand (second) argument in the  
6075     *bind-second* variant.

6076     **s** (IN) GraphBLAS scalar value that is passed to the binary operator as the left-hand  
6077     (first) argument in the *bind-first* variant, or the right-hand (second) argument in  
6078     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).

6107                   GrB\_EMPTY\_OBJECT The GrB\_Scalar  $s$  used in the call is empty ( $\mathbf{nvals}(s) = 0$ ) and  
6108                   therefore a value cannot be passed to the binary operator.

## 6109 Description

6110 This variant of GrB\_apply computes the result of applying a binary operator to the elements of a  
6111 GraphBLAS matrix each composed with a scalar constant,  $\mathbf{val}$  or  $\mathbf{s}$ :

6112                   bind-first:      $C = f(\mathbf{val}, A)$  or  $C = f(\mathbf{s}, A)$

6113                   bind-second:    $C = f(A, \mathbf{val})$  or  $C = f(A, \mathbf{s})$ ,

6114 or if an optional binary accumulation operator ( $\odot$ ) is provided:

6115                   bind-first:      $C = C \odot f(\mathbf{val}, A)$  or  $C = C \odot f(\mathbf{s}, A)$

6116                   bind-second:    $C = C \odot f(A, \mathbf{val})$  or  $C = C \odot f(A, \mathbf{s})$ .

6117 Logically, this operation occurs in three steps:

6118           **Setup** The internal matrices and mask used in the computation are formed and their domains  
6119           and dimensions are tested for compatibility.

6120           **Compute** The indicated computations are carried out.

6121           **Output** The result is written into the output matrix, possibly under control of a mask.

6122 Up to three argument matrices are used in the GrB\_apply operation:

6123     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$

6124     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)

6125     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6126 The argument scalar, matrices, binary operator and the accumulation operator (if provided) are  
6127 tested for domain compatibility as follows:

6128     1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
6129         must be from one of the pre-defined types of Table 3.2.

6130     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the binary operator.

6131     3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6132         of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the binary operator must be compatible with  
6133          $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.



- 6134 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the binary operator.
- 6135 5. If bind-first:
- 6136 (a)  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$  of the binary operator.
- 6137 (b) If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$
- 6138 of the binary operator.
- 6139 (c) If the `GrB_Scalar`  $\mathbf{s}$  is provided, then  $\mathbf{D}(\mathbf{s})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the
- 6140 binary operator.

- 6141 6. If bind-second:
- 6142 (a)  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the binary operator.
- 6143 (b) If the non-opaque scalar  $\mathbf{val}$  is provided, then  $\mathbf{D}(\mathbf{val})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$
- 6144 of the binary operator.
- 6145 (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
- 6146 binary operator.

6147 Two domains are compatible with each other if values from one domain can be cast to values in

6148 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all

6149 compatible with each other. A domain from a user-defined type is only compatible with itself. If

6150 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch

6151 error listed above is returned.

6152 From the argument matrices, the internal matrices, mask, and index arrays used in the computation

6153 are formed ( $\leftarrow$  denotes copy):

- 6154 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6155 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument `Mask` as follows:
- 6156 (a) If `Mask = GrB_NULL`, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$
- 6157  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 6158 (b) If `Mask  $\neq$  GrB_NULL`,
- 6159 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$
- 6160  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
- 6161 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$
- 6162  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
- 6163 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6164 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument `A` as follows:
- 6165 bind-first:  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP1}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6166 bind-second:  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$
- 6167 4. Scalar  $\tilde{s} \leftarrow s$  (`GraphBLAS` scalar case).

6168 The internal matrices and mask are checked for dimension compatibility. The following conditions  
6169 must hold:

- 6170 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6171 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6172 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6173 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6174 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6175 error listed above is returned.

6176 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6177 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6178 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6179 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6180 `val` with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6181 We are now ready to carry out the apply and any additional associated operations. We describe  
6182 this in terms of two intermediate matrices:

- 6183 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the binary operator to the input matrix  $\tilde{\mathbf{A}}$ .
- 6184 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6185 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as one of the following:

6186 bind-first:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\text{val}, \tilde{\mathbf{A}}(i, j))) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6187 bind-second:  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f(\tilde{\mathbf{A}}(i, j), \text{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$

6188 where  $f = \mathbf{f}(\text{op})$ .

6189 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6190 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6191 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6192 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6193 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6194 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 6195 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6196 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6197 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6198 \quad & \\ 6199 \end{aligned}$$

6200 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6201 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
 6202 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
 6203 mask which acts as a “write mask”.

- 6204 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
 6205 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6206 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6207 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
 6208 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
 6209 mask are unchanged:

$$6210 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6211 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
 6212 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
 6213 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
 6214 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
 6215 sequence.

#### 6216 4.3.8.5 apply: Vector index unary operator variant

6217 Computes the transformation of the values of the stored elements of a vector using an index unary  
 6218 operator that is a function of the stored value, its location indices, and an user provided scalar  
 6219 value. The scalar can be passed either as a non-opaque variable or as a GrB\_Scalar object.

#### 6220 C Syntax

```
6221      GrB_Info GrB_apply(GrB_Vector          w,
6222                        const GrB_Vector     mask,
6223                        const GrB_BinaryOp    accum,
6224                        const GrB_IndexUnaryOp op,
6225                        const GrB_Vector     u,
6226                        <type>                val,
6227                        const GrB_Descriptor  desc);
```

```
6228      GrB_Info GrB_apply(GrB_Vector          w,
6229                        const GrB_Vector     mask,
6230                        const GrB_BinaryOp    accum,
6231                        const GrB_IndexUnaryOp op,
6232                        const GrB_Vector     u,
6233                        const GrB_Scalar     s,
6234                        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.

6266                   GrB\_PANIC Unknown internal error.

6267           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
6268                   opaque GraphBLAS objects (input or output) is in an invalid  
6269                   state caused by a previous execution error. Call GrB\_error() to  
6270                   access any error messages generated by the implementation.

6271           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

6272   GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
6273                   by a call to new (or another constructor).

6274   GrB\_DIMENSION\_MISMATCH mask, w and/or u dimensions are incompatible.

6275   GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
6276                   responding domains of the accumulation operator or index unary  
6277                   operator, or the mask's domain is not compatible with bool (in  
6278                   the case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

6279           GrB\_EMPTY\_OBJECT The GrB\_Scalar s used in the call is empty ( $\mathbf{nvals}(s) = 0$ ) and  
6280                   therefore a value cannot be passed to the index unary operator.

## 6281 Description

6282 This variant of GrB\_apply computes the result of applying an index unary operator to the elements  
6283 of a GraphBLAS vector each composed with the element's index and a scalar constant, val or s:

$$6284 \quad \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \text{ or } \mathbf{w} = f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}),$$

6285 or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6286 \quad \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \text{ or } \mathbf{w} = \mathbf{w} \odot f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}).$$

6287 Logically, this operation occurs in three steps:

6288       **Setup** The internal vectors and mask used in the computation are formed and their domains  
6289                   and dimensions are tested for compatibility.

6290       **Compute** The indicated computations are carried out.

6291       **Output** The result is written into the output vector, possibly under control of a mask.

6292 Up to three argument vectors are used in this GrB\_apply operation:

- 6293   1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6294   2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)

6295 3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6296 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6297 are tested for domain compatibility as follows:

- 6298 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
6299 must be from one of the pre-defined types of Table 3.2.
- 6300 2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{out}(\text{op})$  of the index unary operator.
- 6301 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6302 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be compatible  
6303 with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 6304 4.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6305 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  
6306 the index unary operator.
- 6307 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  
6308 unary operator.

6309 Two domains are compatible with each other if values from one domain can be cast to values in  
6310 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6311 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6312 any compatibility rule above is violated, execution of `GrB_apply` ends and the domain mismatch  
6313 error listed above is returned.

6314 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6315 denotes copy):

- 6316 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6317 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
  - 6318 (a) If `mask` = `GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
  - 6319 (b) If `mask`  $\neq$  `GrB_NULL`,
    - 6320 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask})\} \rangle$ ,
    - 6321 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\text{mask}), \{i : i \in \mathbf{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
  - 6322 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6323 3. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6324 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6325 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6326 must hold:

6327 1.  $\text{size}(\tilde{\mathbf{w}}) = \text{size}(\tilde{\mathbf{m}})$

6328 2.  $\text{size}(\tilde{\mathbf{u}}) = \text{size}(\tilde{\mathbf{w}})$ .

6329 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6330 error listed above is returned.

6331 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6332 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6333 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6334 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided ( $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable `val`  
6335 with the same domain as  $\tilde{s}$  and set `val = val( $\tilde{s}$ )`.

6336 We are now ready to carry out the apply and any additional associated operations. We describe  
6337 this in terms of two intermediate vectors:

- 6338 •  $\tilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6339  $\tilde{\mathbf{u}}$ .
- 6340 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6341 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6342 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\text{op}), \text{size}(\tilde{\mathbf{u}}), \{(i, f_i(\tilde{\mathbf{u}}(i), [i], 0, \text{val})) \mid i \in \mathbf{ind}(\tilde{\mathbf{u}})\} \rangle,$$

6343 where  $f_i = \mathbf{f}(\text{op})$ .

6344 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6345 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .
- 6346 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6347 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \mathbf{ind}(\tilde{\mathbf{w}}) \cup \mathbf{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6348 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6349 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6350 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}})), \\ 6351 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{w}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6352 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\mathbf{ind}(\tilde{\mathbf{t}}) - (\mathbf{ind}(\tilde{\mathbf{t}}) \cap \mathbf{ind}(\tilde{\mathbf{w}}))), \\ 6353 \quad & \\ 6354 \end{aligned}$$

6355 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6356 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
6357 using what is called a *standard vector mask and replace*. This is carried out under control of the  
6358 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.



6394       Mask (IN) An optional “write” mask that controls which results from this operation are  
6395       stored into the output matrix C. The mask dimensions must match those of the  
6396       matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
6397       of the Mask matrix must be of type **bool** or any of the predefined “built-in” types  
6398       in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6399       dimensions of C), GrB\_NULL should be specified.

6400       accum (IN) An optional binary operator used for accumulating entries into existing C  
6401       entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
6402       specified.

6403       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  
6404       to each element stored in the input matrix, A. It is a function of the stored element’s  
6405       value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6406       A (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6407       ator.

6408       val (IN) An additional scalar value that is passed to the index unary operator.

6409       s (IN) An additional GraphBLAS scalar that is passed to the index unary operator.

6410       desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
6411       should be specified. Non-default field/value pairs are listed as follows:

6412

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.

6413

## 6414   Return Values

6415       GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
6416       blocking mode, this indicates that the compatibility tests on di-  
6417       mensions and domains for the input arguments passed successfully.  
6418       Either way, output matrix C is ready to be used in the next method  
6419       of the sequence.

6420       GrB\_PANIC Unknown internal error.

6421       GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
6422       GraphBLAS objects (input or output) is in an invalid state caused

6423 by a previous execution error. Call `GrB_error()` to access any error  
 6424 messages generated by the implementation.

6425 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for operation.

6426 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
 6427 a call to `new` (or another constructor).

6428 **GrB\_DIMENSION\_MISMATCH** `mask`, `w` and/or `u` dimensions are incompatible.

6429 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the  
 6430 corresponding domains of the accumulation operator or index unary  
 6431 operator, or the mask's domain is not compatible with `bool` (in the  
 6432 case where `desc[GrB_MASK].GrB_STRUCTURE` is not set).

6433 **GrB\_EMPTY\_OBJECT** The `GrB_Scalar s` used in the call is empty (`nvals(s) = 0`) and  
 6434 therefore a value cannot be passed to the index unary operator.

## 6435 Description

6436 This variant of `GrB_apply` computes the result of applying a index unary operator to the elements  
 6437 of a GraphBLAS matrix each composed with the elements row and column indices, and a scalar  
 6438 constant, `val` or `s`:

$$6439 \quad C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s),$$

6440 or if an optional binary accumulation operator ( $\odot$ ) is provided:

$$6441 \quad C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), \mathbf{val}) \text{ or } C = C \odot f_i(A, \mathbf{row}(\mathbf{ind}(A)), \mathbf{col}(\mathbf{ind}(A)), s).$$

6442 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
 6443 indices, respectively.

6444 Logically, this operation occurs in three steps:

6445 **Setup** The internal matrices and mask used in the computation are formed and their domains  
 6446 and dimensions are tested for compatibility.

6447 **Compute** The indicated computations are carried out.

6448 **Output** The result is written into the output matrix, possibly under control of a mask.

6449 Up to three argument matrices are used in the `GrB_apply` operation:

- 6450 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6451 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)

6452 3.  $\mathbf{A} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\mathbf{A}), \mathbf{ncols}(\mathbf{A}), \mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\} \rangle$

6453 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6454 are tested for domain compatibility as follows:

- 6455 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\mathbf{Mask})$   
6456 must be from one of the pre-defined types of Table 3.2.
- 6457 2.  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator.
- 6458 3. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(\mathbf{C})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6459 of the accumulation operator and  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be compatible  
6460 with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accumulation operator.
- 6461 4.  $\mathbf{D}(\mathbf{A})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6462 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  
6463 the index unary operator.
- 6464 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  
6465 unary operator.

6466 Two domains are compatible with each other if values from one domain can be cast to values in  
6467 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6468 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6469 any compatibility rule above is violated, execution of **GrB\_apply** ends and the domain mismatch  
6470 error listed above is returned.

6471 From the argument matrices, the internal matrices, **mask**, and index arrays used in the computation  
6472 are formed ( $\leftarrow$  denotes copy):

- 6473 1. Matrix  $\tilde{\mathbf{C}} \leftarrow \mathbf{C}$ .
- 6474 2. Two-dimensional mask,  $\tilde{\mathbf{M}}$ , is computed from argument **Mask** as follows:
  - 6475 (a) If **Mask** = **GrB\_NULL**, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
6476  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
  - 6477 (b) If **Mask**  $\neq$  **GrB\_NULL**,
    - 6478 i. If **desc[GrB\_MASK].GrB\_STRUCTURE** is set, then  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}), \{(i, j) :$   
6479  $(i, j) \in \mathbf{ind}(\mathbf{Mask})\} \rangle$ ,
    - 6480 ii. Otherwise,  $\tilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{Mask}), \mathbf{ncols}(\mathbf{Mask}),$   
6481  $\{(i, j) : (i, j) \in \mathbf{ind}(\mathbf{Mask}) \wedge (\mathbf{bool})\mathbf{Mask}(i, j) = \mathbf{true}\} \rangle$ .
  - 6482 (c) If **desc[GrB\_MASK].GrB\_COMP** is set, then  $\tilde{\mathbf{M}} \leftarrow \neg \tilde{\mathbf{M}}$ .
- 6483 3. Matrix  $\tilde{\mathbf{A}}$  is computed from argument **A** as follows:
 
$$6484 \quad \tilde{\mathbf{A}} \leftarrow \mathbf{desc[GrB_INP0].GrB_TRAN} ? \mathbf{A}^T : \mathbf{A}$$
- 6485 4. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

6486 The internal matrices and mask are checked for dimension compatibility. The following conditions  
6487 must hold:

- 6488 1.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{M}})$ .
- 6489 2.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{M}})$ .
- 6490 3.  $\mathbf{nrows}(\tilde{\mathbf{C}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .
- 6491 4.  $\mathbf{ncols}(\tilde{\mathbf{C}}) = \mathbf{ncols}(\tilde{\mathbf{A}})$ .

6492 If any compatibility rule above is violated, execution of `GrB_apply` ends and the dimension mismatch  
6493 error listed above is returned.

6494 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6495 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6496 If an empty `GrB_Scalar`  $\tilde{s}$  is provided ( $\mathbf{nvals}(\tilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6497 If a non-empty `GrB_Scalar`,  $\tilde{s}$ , is provided (i.e.,  $\mathbf{nvals}(\tilde{s}) = 1$ ), we then create an internal variable  
6498  $\mathbf{val}$  with the same domain as  $\tilde{s}$  and set  $\mathbf{val} = \mathbf{val}(\tilde{s})$ .

6499 We are now ready to carry out the apply and any additional associated operations. We describe  
6500 this in terms of two intermediate matrices:

- 6501 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6502  $\tilde{\mathbf{A}}$ .
- 6503 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6504 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6505 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathbf{op}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, f_i(\tilde{\mathbf{A}}(i, j), i, j, \mathbf{val})) \mid (i, j) \in \mathbf{ind}(\tilde{\mathbf{A}})\} \rangle,$$

6506 where  $f_i = \mathbf{f}(\mathbf{op})$ .

6507 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6508 • If  $\mathbf{accum} = \mathbf{GrB\_NULL}$ , then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6509 • If  $\mathbf{accum}$  is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6510 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\mathbf{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

6511 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6512 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned} 6513 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6514 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6515 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6516 \quad & \\ 6517 \end{aligned}$$

6518 where  $\odot = \odot(\mathbf{accum})$ , and the difference operator refers to set difference.

6519 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6520 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6521 mask which acts as a “write mask”.

- 6522 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
6523 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6524 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6525 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6526 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6527 mask are unchanged:

$$6528 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6529 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
6530 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
6531 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
6532 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6533 sequence.

#### 6534 4.3.9 select:

6535 Apply a select operator to the stored elements of an object to determine whether or not to keep  
6536 them.

##### 6537 4.3.9.1 select: Vector variant

6538 Apply a select operator (an index unary operator) to the elements of a vector.

#### 6539 C Syntax

```
6540 // scalar value variant
6541 GrB_Info GrB_select(GrB_Vector          w,
6542                    const GrB_Vector      mask,
6543                    const GrB_BinaryOp    accum,
6544                    const GrB_IndexUnaryOp op,
6545                    const GrB_Vector      u,
6546                    <type>                val,
6547                    const GrB_Descriptor   desc);
6548
6549 // GraphBLAS scalar variant
6550 GrB_Info GrB_select(GrB_Vector          w,
6551                    const GrB_Vector      mask,
```

```

6552         const GrB_BinaryOp      accum,
6553         const GrB_IndexUnaryOp  op,
6554         const GrB_Vector        u,
6555         const GrB_Scalar        s,
6556         const GrB_Descriptor    desc);
6557

```

## 6558 Parameters

6559     **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6560     that may be accumulated with the result of the select operation. On output, this  
6561     vector holds the results of the operation.

6562     **mask** (IN) An optional “write” mask that controls which results from this operation are  
6563     stored into the output vector **w**. The mask dimensions must match those of the  
6564     vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6565     of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6566     in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6567     dimensions of **w**), **GrB\_NULL** should be specified.

6568     **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6569     entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6570     specified.

6571     **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  
6572     to each element stored in the input vector, **u**. It is a function of the stored element’s  
6573     value, its location index, and a user supplied scalar value (either **s** or **val**).

6574     **u** (IN) The GraphBLAS vector whose elements are passed to the index unary oper-  
6575     ator.

6576     **val** (IN) An additional scalar value that is passed to the index unary operator.

6577     **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6578     not be empty.

6579     **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6580     should be specified. Non-default field/value pairs are listed as follows:

6581

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> .

6582

## 6583 Return Values

6584           GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
6585                       blocking mode, this indicates that the compatibility tests on di-  
6586                       mensions and domains for the input arguments passed success-  
6587                       fully. Either way, output vector **w** is ready to be used in the next  
6588                       method of the sequence.

6589           GrB\_PANIC Unknown internal error.

6590           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the  
6591                       opaque GraphBLAS objects (input or output) is in an invalid  
6592                       state caused by a previous execution error. Call **GrB\_error()** to  
6593                       access any error messages generated by the implementation.

6594           GrB\_OUT\_OF\_MEMORY Not enough memory available for operation.

6595           GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized  
6596                       by a call to one of its constructors.

6597           GrB\_DIMENSION\_MISMATCH **mask**, **w** and/or **u** dimensions are incompatible.

6598           GrB\_DOMAIN\_MISMATCH The domains of the various vectors are incompatible with the cor-  
6599                       responding domains of the accumulation operator or index unary  
6600                       operator, or the **mask**'s domain is not compatible with **bool** (in  
6601                       the case where **desc[GrB\_MASK].GrB\_STRUCTURE** is not set).

6602           GrB\_EMPTY\_OBJECT The **GrB\_Scalar s** used in the call is empty (**nvals(s) = 0**) and  
6603                       therefore a value cannot be passed to the index unary operator.

## 6604 Description

6605 This variant of **GrB\_select** computes the result of applying a index unary operator to select the  
6606 elements of the input GraphBLAS vector. The operator takes, as input, the value of each stored  
6607 element, along with the element's index and a scalar constant – either **val** or **s**. The corresponding  
6608 element of the input vector is selected (kept) if the function evaluates to **true** when cast to **bool**.  
6609 This acts like a functional mask on the input vector as follows:

$$6610 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle,$$

$$6611 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{val}) \rangle.$$

6612 Correspondingly, if a **GrB\_Scalar s**, is provided:

$$6613 \quad \mathbf{w} = \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle,$$

$$6614 \quad \mathbf{w} = \mathbf{w} \odot \mathbf{u} \langle f_i(\mathbf{u}, \mathbf{ind}(\mathbf{u}), 0, \mathbf{s}) \rangle.$$

6615 Logically, this operation occurs in three steps:

6616     **Setup** The internal vectors and mask used in the computation are formed and their domains  
6617             and dimensions are tested for compatibility.

6618     **Compute** The indicated computations are carried out.

6619     **Output** The result is written into the output vector, possibly under control of a mask.

6620 Up to three argument vectors are used in this `GrB_select` operation:

- 6621     1.  $\mathbf{w} = \langle \mathbf{D}(\mathbf{w}), \mathbf{size}(\mathbf{w}), \mathbf{L}(\mathbf{w}) = \{(i, w_i)\} \rangle$
- 6622     2.  $\mathbf{mask} = \langle \mathbf{D}(\mathbf{mask}), \mathbf{size}(\mathbf{mask}), \mathbf{L}(\mathbf{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 6623     3.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \mathbf{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

6624 The argument scalar, vectors, index unary operator and the accumulation operator (if provided)  
6625 are tested for domain compatibility as follows:

- 6626     1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\mathbf{mask})$   
6627         must be from one of the pre-defined types of Table 3.2.
- 6628     2.  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}(\mathbf{u})$ .
- 6629     3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(\mathbf{w})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{accum})$  and  $\mathbf{D}_{out}(\mathbf{accum})$   
6630         of the accumulation operator and  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_2}(\mathbf{accum})$  of the accu-  
6631         mulation operator.
- 6632     4.  $\mathbf{D}_{out}(\mathbf{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6633         i.e., castable to `bool`.
- 6634     5.  $\mathbf{D}(\mathbf{u})$  must be compatible with  $\mathbf{D}_{in_1}(\mathbf{op})$  of the index unary operator.
- 6635     6.  $\mathbf{D}(\mathbf{val})$  or  $\mathbf{D}(\mathbf{s})$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\mathbf{op})$   
6636         of the index unary operator.

6637 Two domains are compatible with each other if values from one domain can be cast to values in  
6638 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
6639 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
6640 any compatibility rule above is violated, execution of `GrB_select` ends and the domain mismatch  
6641 error listed above is returned.

6642 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
6643 denotes copy):

- 6644     1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6645     2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:



- 6646 (a) If  $\text{mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{m}} = \langle \text{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \text{size}(\mathbf{w})\} \rangle$ .
- 6647 (b) If  $\text{mask} \neq \text{GrB\_NULL}$ ,
- 6648 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask})\} \rangle$ ,
- 6649 ii. Otherwise,  $\widetilde{\mathbf{m}} = \langle \text{size}(\text{mask}), \{i : i \in \text{ind}(\text{mask}) \wedge (\text{bool})\text{mask}(i) = \text{true}\} \rangle$ .
- 6650 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{m}} \leftarrow \neg \widetilde{\mathbf{m}}$ .
- 6651 3. Vector  $\widetilde{\mathbf{u}} \leftarrow \mathbf{u}$ .
- 6652 4. Scalar  $\widetilde{s} \leftarrow s$  (GrB\_Scalar version only).

6653 The internal vectors and masks are checked for dimension compatibility. The following conditions  
6654 must hold:

- 6655 1.  $\text{size}(\widetilde{\mathbf{w}}) = \text{size}(\widetilde{\mathbf{m}})$
- 6656 2.  $\text{size}(\widetilde{\mathbf{u}}) = \text{size}(\widetilde{\mathbf{w}})$ .

6657 If any compatibility rule above is violated, execution of `GrB_select` ends and the dimension mismatch  
6658 error listed above is returned.

6659 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
6660 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6661 If an empty `GrB_Scalar`  $\widetilde{s}$  is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 0$ ), the method returns with code `GrB_EMPTY_OBJECT`.  
6662 If a non-empty `GrB_Scalar`,  $\widetilde{s}$ , is provided (i.e.,  $\text{nvals}(\widetilde{s}) = 1$ ), we then create an internal variable  
6663 `val` with the same domain as  $\widetilde{s}$  and set  $\text{val} = \text{val}(\widetilde{s})$ .

6664 We are now ready to carry out the `select` and any additional associated operations. We describe  
6665 this in terms of two intermediate vectors:

- 6666 •  $\widetilde{\mathbf{t}}$ : The vector holding the result from applying the index unary operator to the input vector  
6667  $\widetilde{\mathbf{u}}$ .
- 6668 •  $\widetilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6669 The intermediate vector,  $\widetilde{\mathbf{t}}$ , is created as follows:

$$6670 \quad \widetilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\widetilde{\mathbf{u}}), \{(i, \widetilde{\mathbf{u}}(i), : i \in \text{ind}(\widetilde{\mathbf{u}}) \wedge (\text{bool})f_i(\widetilde{\mathbf{u}}(i), i, 0, \text{val}) = \text{true})\} \rangle,$$

6671 where  $f_i = \mathbf{f}(\text{op})$ .

6672 The intermediate vector  $\widetilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6673 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{z}} = \widetilde{\mathbf{t}}$ .
- 6674 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{z}}$  is defined as

$$6675 \quad \widetilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\widetilde{\mathbf{w}}), \{(i, z_i) \mid \forall i \in \text{ind}(\widetilde{\mathbf{w}}) \cup \text{ind}(\widetilde{\mathbf{t}})\} \rangle.$$

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);
```

```

6711 // GraphBLAS scalar variant
6712 GrB_Info GrB_select(GrB_Matrix          C,
6713                    const GrB_Matrix     Mask,
6714                    const GrB_BinaryOp    accum,
6715                    const GrB_IndexUnaryOp op,
6716                    const GrB_Matrix     A,
6717                    const GrB_Scalar      s,
6718                    const GrB_Descriptor  desc);

```

## 6719 Parameters

6720 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
6721 that may be accumulated with the result of the select operation. On output, the  
6722 matrix holds the results of the operation.

6723 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
6724 stored into the output matrix **C**. The mask dimensions must match those of the  
6725 matrix **C**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6726 of the **Mask** matrix must be of type **bool** or any of the predefined “built-in” types  
6727 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6728 dimensions of **C**), **GrB\_NULL** should be specified.

6729 **accum** (IN) An optional binary operator used for accumulating entries into existing **C**  
6730 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6731 specified.

6732 **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  
6733 to each element stored in the input matrix, **A**. It is a function of the stored element’s  
6734 value, its row and column indices, and a user supplied scalar value (either **s** or **val**).

6735 **A** (IN) The GraphBLAS matrix whose elements are passed to the index unary oper-  
6736 ator.

6737 **val** (IN) An additional scalar value that is passed to the index unary operator.

6738 **s** (IN) An GraphBLAS scalar that is passed to the index unary operator. It must  
6739 not be empty.

6740 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
6741 should be specified. Non-default field/value pairs are listed as follows:  
6742

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:

6772  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ , or  
6773  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), \text{val}) \rangle$ .

6774 Correspondingly, if a GrB\_Scalar,  $s$ , is provided:

6775  $C = A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ , or  
6776  $C = C \odot A \langle f_i(A, \text{row}(\text{ind}(A)), \text{col}(\text{ind}(A)), s) \rangle$ .

6777 Where the **row** and **col** functions extract the row and column indices from a list of two-dimensional  
6778 indices, respectively.

6779 Logically, this operation occurs in three steps:

6780 **Setup** The internal matrices and mask used in the computation are formed and their domains  
6781 and dimensions are tested for compatibility.

6782 **Compute** The indicated computations are carried out.

6783 **Output** The result is written into the output matrix, possibly under control of a mask.

6784 Up to three argument matrices are used in the GrB\_select operation:

- 6785 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 6786 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)
- 6787 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6788 The argument scalar, matrices, index unary operator and the accumulation operator (if provided)  
6789 are tested for domain compatibility as follows:

- 6790 1. If **Mask** is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
6791 must be from one of the pre-defined types of Table 3.2.
- 6792 2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$ .
- 6793 3. If **accum** is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
6794 of the accumulation operator and  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
6795 mulation operator.
- 6796 4.  $\mathbf{D}_{out}(\text{op})$  of the index unary operator must be from one of the pre-defined types of Table 3.2;  
6797 i.e., castable to **bool**.
- 6798 5.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$  of the index unary operator.
- 6799 6.  $\mathbf{D}(\text{val})$  or  $\mathbf{D}(s)$ , depending on the signature of the method, must be compatible with  $\mathbf{D}_{in_2}(\text{op})$   
6800 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:

- 6835 •  $\tilde{\mathbf{T}}$ : The matrix holding the result from applying the index unary operator to the input matrix  
6836  $\tilde{\mathbf{A}}$ .
- 6837 •  $\tilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

6838 The intermediate matrix,  $\tilde{\mathbf{T}}$ , is created as follows:

$$6839 \quad \tilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{nrows}(\tilde{\mathbf{A}}), \mathbf{ncols}(\tilde{\mathbf{A}}), \\ \{(i, j, \tilde{\mathbf{A}}(i, j) : i, j \in \mathbf{ind}(\tilde{\mathbf{A}}) \wedge (\text{bool})f_i(\tilde{\mathbf{A}}(i, j), i, j, \text{val}) = \text{true})\},$$

6840 where  $f_i = \mathbf{f}(\text{op})$ .

6841 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 6842 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 6843 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$6844 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\}\rangle.$$

6845 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
6846 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$6847 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\ 6848 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6849 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\ 6850 \quad 6851$$

6852 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

6853 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
6854 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
6855 mask which acts as a “write mask”.

- 6856 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
6857 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$6858 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 6859 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
6860 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
6861 mask are unchanged:

$$6862 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

6863 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
6864 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
6865 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
6866 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
6867 sequence.

### 6868 4.3.10 reduce: Perform a reduction across the elements of an object

6869 Computes the reduction of the values of the elements of a vector or matrix.

#### 6870 4.3.10.1 reduce: Standard matrix to vector variant

6871 This performs a reduction across rows of a matrix to produce a vector. If reduction down columns  
6872 is desired, the input matrix should be transposed using the descriptor.

### 6873 C Syntax

```
6874     GrB_Info GrB_reduce(GrB_Vector      w,  
6875                        const GrB_Vector mask,  
6876                        const GrB_BinaryOp accum,  
6877                        const GrB_Monoid op,  
6878                        const GrB_Matrix A,  
6879                        const GrB_Descriptor desc);  
6880  
6881     GrB_Info GrB_reduce(GrB_Vector      w,  
6882                        const GrB_Vector mask,  
6883                        const GrB_BinaryOp accum,  
6884                        const GrB_BinaryOp op,  
6885                        const GrB_Matrix A,  
6886                        const GrB_Descriptor desc);
```

### 6887 Parameters

6888 **w** (INOUT) An existing GraphBLAS vector. On input, the vector provides values  
6889 that may be accumulated with the result of the reduction operation. On output,  
6890 this vector holds the results of the operation.

6891 **mask** (IN) An optional “write” mask that controls which results from this operation are  
6892 stored into the output vector **w**. The mask dimensions must match those of the  
6893 vector **w**. If the **GrB\_STRUCTURE** descriptor is *not* set for the mask, the domain  
6894 of the **mask** vector must be of type **bool** or any of the predefined “built-in” types  
6895 in Table 3.2. If the default mask is desired (i.e., a mask that is all **true** with the  
6896 dimensions of **w**), **GrB\_NULL** should be specified.

6897 **accum** (IN) An optional binary operator used for accumulating entries into existing **w**  
6898 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
6899 specified.

6900 **op** (IN) The monoid or binary operator used in the element-wise reduction operation.  
6901 Depending on which type is passed, the following defines the binary operator with  
6902 one domain,  $F_b = \langle D, D, D, \oplus \rangle$ , that is used:



6903 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .  
6904 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  
6905 monoid is ignored.

6906 If  $\text{op}$  is a `GrB_BinaryOp`, then all its domains must be the same. Furthermore, in  
6907 both cases  $\odot(\text{op})$  must be commutative and associative. Otherwise, the outcome  
6908 of the operation is undefined.

6909 **A** (IN) The GraphBLAS matrix on which reduction will be performed.

6910 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, `GrB_NULL`  
6911 should be specified. Non-default field/value pairs are listed as follows:  
6912

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.

## 6914 Return Values

6915 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
6916 blocking mode, this indicates that the compatibility tests on di-  
6917 mensions and domains for the input arguments passed successfully.  
6918 Either way, output vector w is ready to be used in the next method  
6919 of the sequence.

6920 **GrB\_PANIC** Unknown internal error.

6921 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
6922 GraphBLAS objects (input or output) is in an invalid state caused  
6923 by a previous execution error. Call `GrB_error()` to access any error  
6924 messages generated by the implementation.

6925 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

6926 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
6927 a call to `new` (or `dup` for vector parameters).

6928 **GrB\_DIMENSION\_MISMATCH** mask, w and/or u dimensions are incompatible.

6929 **GrB\_DOMAIN\_MISMATCH** Either the domains of the various vectors and matrices are incom-  
6930 patible with the corresponding domains of the accumulation oper-  
6931 ator or reduce function, or the domains of the GraphBLAS binary

6932 operator `op` are not all the same, or the mask's domain is not com-  
 6933 patible with `bool` (in the case where `desc[GrB_MASK].GrB_STRUCTURE`  
 6934 is not set).

## 6935 Description

6936 This variant of `GrB_reduce` computes the result of performing a reduction across each of the rows  
 6937 of an input matrix:  $w(i) = \bigoplus A(i, :) \forall i$ ; or, if an optional binary accumulation operator is provided,  
 6938  $w(i) = w(i) \odot (\bigoplus A(i, :)) \forall i$ , where  $\bigoplus = \odot(F_b)$  and  $\odot = \odot(\text{accum})$ .

6939 Logically, this operation occurs in three steps:

6940     **Setup** The internal vector, matrix and mask used in the computation are formed and their  
 6941 domains and dimensions are tested for compatibility.

6942 **Compute** The indicated computations are carried out.

6943 **Output** The result is written into the output vector, possibly under control of a mask.

6944 Up to two vector and one matrix argument are used in this `GrB_reduce` operation:

- 6945 1.  $w = \langle \mathbf{D}(w), \text{size}(w), \mathbf{L}(w) = \{(i, w_i)\} \rangle$
- 6946 2.  $\text{mask} = \langle \mathbf{D}(\text{mask}), \text{size}(\text{mask}), \mathbf{L}(\text{mask}) = \{(i, m_i)\} \rangle$  (optional)
- 6947 3.  $A = \langle \mathbf{D}(A), \text{nrows}(A), \text{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

6948 The argument vector, matrix, reduction operator and accumulation operator (if provided) are tested  
 6949 for domain compatibility as follows:

- 6950 1. If `mask` is not `GrB_NULL`, and `desc[GrB_MASK].GrB_STRUCTURE` is not set, then  $\mathbf{D}(\text{mask})$   
 6951 must be from one of the pre-defined types of Table 3.2.
- 6952 2.  $\mathbf{D}(w)$  must be compatible with the domain of the reduction binary operator,  $\mathbf{D}(F_b)$ .
- 6953 3. If `accum` is not `GrB_NULL`, then  $\mathbf{D}(w)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 6954 of the accumulation operator and  $\mathbf{D}(F_b)$ , must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accu-  
 6955 mulation operator.
- 6956 4.  $\mathbf{D}(A)$  must be compatible with the domain of the binary reduction operator,  $\mathbf{D}(F_b)$ .

6957 Two domains are compatible with each other if values from one domain can be cast to values in  
 6958 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 6959 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 6960 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
 6961 error listed above is returned.

6962 From the argument vectors, the internal vectors and mask used in the computation are formed ( $\leftarrow$   
 6963 denotes copy):

- 6964 1. Vector  $\tilde{\mathbf{w}} \leftarrow \mathbf{w}$ .
- 6965 2. One-dimensional mask,  $\tilde{\mathbf{m}}$ , is computed from argument `mask` as follows:
- 6966 (a) If `mask = GrB_NULL`, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{w}), \{i, \forall i : 0 \leq i < \mathbf{size}(\mathbf{w})\} \rangle$ .
- 6967 (b) If `mask  $\neq$  GrB_NULL`,
- 6968 i. If `desc[GrB_MASK].GrB_STRUCTURE` is set, then  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask})\} \rangle$ ,
- 6969 ii. Otherwise,  $\tilde{\mathbf{m}} = \langle \mathbf{size}(\mathbf{mask}), \{i : i \in \mathbf{ind}(\mathbf{mask}) \wedge (\mathbf{bool})\mathbf{mask}(i) = \mathbf{true}\} \rangle$ .
- 6970 (c) If `desc[GrB_MASK].GrB_COMP` is set, then  $\tilde{\mathbf{m}} \leftarrow \neg \tilde{\mathbf{m}}$ .
- 6971 3. Matrix  $\tilde{\mathbf{A}} \leftarrow \mathbf{desc}[\mathbf{GrB\_INP0}].\mathbf{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

6972 The internal vectors and masks are checked for dimension compatibility. The following conditions  
 6973 must hold:

- 6974 1.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{size}(\tilde{\mathbf{m}})$
- 6975 2.  $\mathbf{size}(\tilde{\mathbf{w}}) = \mathbf{nrows}(\tilde{\mathbf{A}})$ .

6976 If any compatibility rule above is violated, execution of `GrB_reduce` ends and the dimension mis-  
 6977 match error listed above is returned.

6978 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
 6979 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

6980 We carry out the reduce and any additional associated operations. We describe this in terms of  
 6981 two intermediate vectors:

- 6982 •  $\tilde{\mathbf{t}}$ : The vector holding the result from reducing along the rows of input matrix  $\tilde{\mathbf{A}}$ .
- 6983 •  $\tilde{\mathbf{z}}$ : The vector holding the result after application of the (optional) accumulation operator.

6984 The intermediate vector,  $\tilde{\mathbf{t}}$ , is created as follows:

$$6985 \quad \tilde{\mathbf{t}} = \langle \mathbf{D}(\mathbf{op}), \mathbf{size}(\tilde{\mathbf{w}}), \{(i, t_i) : \mathbf{ind}(\mathbf{A}(i, :)) \neq \emptyset\} \rangle.$$

6986 The value of each of its elements is computed by

$$6987 \quad t_i = \bigoplus_{j \in \mathbf{ind}(\tilde{\mathbf{A}}(i, :))} \tilde{\mathbf{A}}(i, j),$$

6988 where  $\bigoplus = \odot(F_b)$ .

6989 The intermediate vector  $\tilde{\mathbf{z}}$  is created as follows, using what is called a *standard vector accumulate*:

- 6990 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{z}} = \tilde{\mathbf{t}}$ .

6991 • If `accum` is a binary operator, then  $\tilde{\mathbf{z}}$  is defined as

$$6992 \quad \tilde{\mathbf{z}} = \langle \mathbf{D}_{out}(\text{accum}), \text{size}(\tilde{\mathbf{w}}), \{(i, z_i) \mid i \in \text{ind}(\tilde{\mathbf{w}}) \cup \text{ind}(\tilde{\mathbf{t}})\} \rangle.$$

6993 The values of the elements of  $\tilde{\mathbf{z}}$  are computed based on the relationships between the sets of  
6994 indices in  $\tilde{\mathbf{w}}$  and  $\tilde{\mathbf{t}}$ .

$$\begin{aligned} 6995 \quad z_i &= \tilde{\mathbf{w}}(i) \odot \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}})), \\ 6996 \quad z_i &= \tilde{\mathbf{w}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{w}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6997 \quad z_i &= \tilde{\mathbf{t}}(i), \text{ if } i \in (\text{ind}(\tilde{\mathbf{t}}) - (\text{ind}(\tilde{\mathbf{t}}) \cap \text{ind}(\tilde{\mathbf{w}}))), \\ 6998 \quad & \\ 6999 \end{aligned}$$

7000 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7001 Finally, the set of output values that make up vector  $\tilde{\mathbf{z}}$  are written into the final result vector  $\mathbf{w}$ ,  
7002 using what is called a *standard vector mask and replace*. This is carried out under control of the  
7003 mask which acts as a “write mask”.

7004 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{w}$  on input to this operation are  
7005 deleted and the content of the new output vector,  $\mathbf{w}$ , is defined as,

$$7006 \quad \mathbf{L}(\mathbf{w}) = \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7007 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{z}}$  indicated by the mask are  
7008 copied into the result vector,  $\mathbf{w}$ , and elements of  $\mathbf{w}$  that fall outside the set indicated by the  
7009 mask are unchanged:

$$7010 \quad \mathbf{L}(\mathbf{w}) = \{(i, w_i) : i \in (\text{ind}(\mathbf{w}) \cap \text{ind}(\neg \tilde{\mathbf{m}}))\} \cup \{(i, z_i) : i \in (\text{ind}(\tilde{\mathbf{z}}) \cap \text{ind}(\tilde{\mathbf{m}}))\}.$$

7011 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
7012 of vector  $\mathbf{w}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
7013 exits with return value `GrB_SUCCESS` and the new content of vector  $\mathbf{w}$  is as defined above but  
7014 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7015 sequence.

#### 7016 4.3.10.2 reduce: Vector-scalar variant

7017 Reduce all stored values into a single scalar.

#### 7018 C Syntax

```
7019 // scalar value + monoid (only)
7020 GrB_Info GrB_reduce(<type>          *val,
7021                      const GrB_BinaryOp accum,
7022                      const GrB_Monoid  op,
7023                      const GrB_Vector  u,
```

```

7024             const GrB_Descriptor desc);
7025
7026 // GraphBLAS Scalar + monoid
7027 GrB_Info GrB_reduce(GrB_Scalar      s,
7028                   const GrB_BinaryOp accum,
7029                   const GrB_Monoid  op,
7030                   const GrB_Vector  u,
7031                   const GrB_Descriptor desc);
7032
7033 // GraphBLAS Scalar + binary operator
7034 GrB_Info GrB_reduce(GrB_Scalar      s,
7035                   const GrB_BinaryOp accum,
7036                   const GrB_BinaryOp op,
7037                   const GrB_Vector  u,
7038                   const GrB_Descriptor desc);

```

## 7039 Parameters

7040 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7041 a value that may be accumulated (optionally) with the result of the reduction  
7042 operation. On output, this scalar holds the results of the operation.

7043 **accum** (IN) An optional binary operator used for accumulating entries into an exist-  
7044 ing scalar (**s** or **val**) value. If assignment rather than accumulation is desired,  
7045 GrB\_NULL should be specified.

7046 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7047 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7048 otherwise, the outcome of the operation is undefined.

7049 **u** (IN) The GraphBLAS vector on which reduction will be performed.

7050 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7051 should be specified. Non-default field/value pairs are listed as follows:

7053 Param	Field	Value	Description
------------	-------	-------	-------------

7054 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
7055 tions. There are currently no non-default field/value pairs that can be set for this  
7056 operation.

## 7057 Return Values

7058 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
7059 cessfully, and the output scalar (**s** or **val**) is ready to be used in the  
7060 next method of the sequence.

7061                   GrB\_PANIC Unknown internal error.

7062           GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7063                   GraphBLAS objects (input or output) is in an invalid state caused  
7064                   by a previous execution error. Call GrB\_error() to access any error  
7065                   messages generated by the implementation.

7066           GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7067 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7068                   a call to a respective constructor.

7069           GrB\_NULL\_POINTER val pointer is NULL.

7070   GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
7071                   the corresponding domains of the accumulation operator, or reduce  
7072                   operator.

### 7073 Description

7074 This variant of GrB\_reduce computes the result of performing a reduction across all of the stored  
7075 elements of an input vector storing the result into either s or val. This corresponds to (shown here  
7076 for the scalar value case only):

$$7077 \quad \text{val} = \begin{cases} \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{i \in \text{ind}(\mathbf{u})} \mathbf{u}(i) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7078 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7079 Logically, this operation occurs in three steps:

7080       **Setup** The internal vector used in the computation is formed and its domain is tested for  
7081                   compatibility.

7082       **Compute** The indicated computations are carried out.

7083       **Output** The result is written into the output scalar.

7084 One vector argument is used in this GrB\_reduce operation:

- 7085 1.  $\mathbf{u} = \langle \mathbf{D}(\mathbf{u}), \text{size}(\mathbf{u}), \mathbf{L}(\mathbf{u}) = \{(i, u_i)\} \rangle$

7086 The output scalar, argument vector, reduction operator and accumulation operator (if provided)  
7087 are tested for domain compatibility as follows:

- 7088 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  
7089  $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

- 7090 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  
7091  $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
7092 be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.
- 7093 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  $F_b$ ).

7094 Two domains are compatible with each other if values from one domain can be cast to values in  
7095 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
7096 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
7097 any compatibility rule above is violated, execution of `GrB_reduce` ends and the domain mismatch  
7098 error listed above is returned.

7099 The number of values stored in the input, `u`, is checked. If there are no stored values in `u`, then one  
7100 of the following occurs depending on the output variant:

$$7101 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7102 or

$$7103 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if } \text{accum} = \text{GrB\_NULL}, \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7104 where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with `GrB_SUCCESS`.

7105 For all other cases, the internal vector and scalar used in the computation is formed ( $\leftarrow$  denotes  
7106 copy):

7107 1. Vector  $\tilde{\mathbf{u}} \leftarrow \mathbf{u}$ .

7108 2. Scalar  $\tilde{s} \leftarrow \text{s}$  (GraphBLAS scalar case).

7109 We are now ready to carry out the reduction and any additional associated operations. An inter-  
7110 mediate scalar result  $t$  is computed as follows:

$$7111 \quad t = \bigoplus_{i \in \text{ind}(\tilde{\mathbf{u}})} \tilde{\mathbf{u}}(i),$$

7112 where  $\oplus = \odot(\text{op})$ .

7113 The final reduction value is computed as follows:

$$7114 \quad \mathbf{L}(\text{s}) \leftarrow \begin{cases} \{t\}, & \text{when } \text{accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\text{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7115 or

$$7116 \quad \text{val} \leftarrow \begin{cases} t, & \text{when } \text{accum} = \text{GrB\_NULL, or} \\ \text{val} \odot t, & \text{otherwise;} \end{cases}$$

7117 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
7118 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7119 4.3.10.3 reduce: Matrix-scalar variant

7120 Reduce all stored values into a single scalar.

#### 7121 C Syntax

```
7122 // scalar value + monoid (only)
7123 GrB_Info GrB_reduce(<type>          *val,
7124                   const GrB_BinaryOp accum,
7125                   const GrB_Monoid   op,
7126                   const GrB_Matrix   A,
7127                   const GrB_Descriptor desc);
7128
7129 // GraphBLAS Scalar + monoid
7130 GrB_Info GrB_reduce(GrB_Scalar      s,
7131                   const GrB_BinaryOp accum,
7132                   const GrB_Monoid   op,
7133                   const GrB_Matrix   A,
7134                   const GrB_Descriptor desc);
7135
7136 // GraphBLAS Scalar + binary operator
7137 GrB_Info GrB_reduce(GrB_Scalar      s,
7138                   const GrB_BinaryOp accum,
7139                   const GrB_BinaryOp op,
7140                   const GrB_Matrix   A,
7141                   const GrB_Descriptor desc);
```

#### 7142 Parameters

7143 **val** or **s** (INOUT) Scalar to store final reduced value into. On input, the scalar provides  
7144 a value that may be accumulated (optionally) with the result of the reduction  
7145 operation. On output, this scalar holds the results of the operation.

7146 **accum** (IN) An optional binary operator used for accumulating entries into existing (**s** or  
7147 **val**) value. If assignment rather than accumulation is desired, GrB\_NULL should  
7148 be specified.

7149 **op** (IN) The monoid ( $M = \langle D, \oplus, 0 \rangle$ ) or binary operator ( $F_b = \langle D, D, D, \oplus \rangle$ ) used in  
7150 the reduction operation. The  $\oplus$  operator must be commutative and associative;  
7151 otherwise, the outcome of the operation is undefined.

7152 **A** (IN) The GraphBLAS matrix on which the reduction will be performed.



7153 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
 7154 should be specified. Non-default field/value pairs are listed as follows:  
 7155

7156	Param	Field	Value	Description
------	-------	-------	-------	-------------

7157 *Note:* This argument is defined for consistency with the other GraphBLAS opera-  
 7158 tions. There are currently no non-default field/value pairs that can be set for this  
 7159 operation.

## 7160 Return Values

7161 GrB\_SUCCESS In blocking or non-blocking mode, the operation completed suc-  
 7162 cessfully, and the output scalar (s or val) is ready to be used in the  
 7163 next method of the sequence.

7164 GrB\_PANIC Unknown internal error.

7165 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
 7166 GraphBLAS objects (input or output) is in an invalid state caused  
 7167 by a previous execution error. Call GrB\_error() to access any error  
 7168 messages generated by the implementation.

7169 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7170 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
 7171 a call to a respective constructor.

7172 GrB\_NULL\_POINTER val pointer is NULL.

7173 GrB\_DOMAIN\_MISMATCH The domains of input and output arguments are incompatible with  
 7174 the corresponding domains of the accumulation operator, or reduce  
 7175 operator.

## 7176 Description

7177 This variant of GrB\_reduce computes the result of performing a reduction across all of the stored  
 7178 elements of an input matrix storing the result into either s or val. This corresponds to (shown here  
 7179 for the scalar value case only):

$$7180 \quad \text{val} = \begin{cases} \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j), & \text{or} \\ \text{val} \odot \left[ \bigoplus_{(i,j) \in \text{ind}(\mathbf{A})} \mathbf{A}(i,j) \right], & \text{if the optional accumulator is specified.} \end{cases}$$

7181 where  $\bigoplus = \odot(\text{op})$  and  $\odot = \odot(\text{accum})$ .

7182 Logically, this operation occurs in three steps:

7183       **Setup** The internal matrix used in the computation is formed and its domain is tested for  
 7184       compatibility.

7185       **Compute** The indicated computations are carried out.

7186       **Output** The result is written into the output scalar.

7187   One matrix argument is used in this GrB\_reduce operation:

7188       1.  $A = \langle \mathbf{D}(A), \mathbf{size}(A), \mathbf{L}(A) = \{(i, j, A_{i,j})\} \rangle$

7189   The output scalar, argument matrix, reduction operator and accumulation operator (if provided)  
 7190   are tested for domain compatibility as follows:

7191       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  
 7192        $\mathbf{D}_{in_1}(\text{op})$  and  $\mathbf{D}_{in_2}(\text{op})$  from  $F_b$ ).

7193       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  
 7194        $\mathbf{D}_{out}(\text{accum})$  of the accumulation operator, and  $\mathbf{D}(\text{op})$  from  $M$  (or  $\mathbf{D}_{out}(\text{op})$  from  $F_b$ ) must  
 7195       be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of the accumulation operator.

7196       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  $F_b$ ).

7197   Two domains are compatible with each other if values from one domain can be cast to values in  
 7198   the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7199   compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7200   any compatibility rule above is violated, execution of GrB\_reduce ends and the domain mismatch  
 7201   error listed above is returned.

7202   The number of values stored in the input,  $A$ , is checked. If there are no stored values in  $A$ , then  
 7203   one of the following occurs depending on the output variant:

$$7204 \quad \mathbf{L}(\text{s}) = \begin{cases} \{\}, & \text{(cleared) if accum = GrB\_NULL,} \\ \mathbf{L}(\text{s}), & \text{(unchanged) otherwise,} \end{cases}$$

7205   or

$$7206 \quad \text{val} = \begin{cases} \mathbf{0}(\text{op}), & \text{(cleared) if accum = GrB\_NULL,} \\ \text{val} \odot \mathbf{0}(\text{op}), & \text{otherwise,} \end{cases}$$

7207   where  $\mathbf{0}(\text{op})$  is the identity of the monoid. The operation returns immediately with GrB\_SUCCESS.

7208   For all other cases, the internal matrix and scalar used in the computation is formed ( $\leftarrow$  denotes  
 7209   copy):

7210       1. Matrix  $\tilde{A} \leftarrow A$ .

7211       2. Scalar  $\tilde{s} \leftarrow s$  (GraphBLAS scalar case).

7212 We are now ready to carry out the reduce and any additional associated operations. An intermediate  
 7213 scalar result  $t$  is computed as follows:

$$7214 \quad t = \bigoplus_{(i,j) \in \text{ind}(\tilde{\mathbf{A}})} \tilde{\mathbf{A}}(i,j),$$

7215 where  $\oplus = \odot(\text{op})$ .

7216 The final reduction value is computed as follows:

$$7217 \quad \mathbf{L}(\mathbf{s}) \leftarrow \begin{cases} \{t\}, & \text{when accum} = \text{GrB\_NULL} \text{ or } \tilde{s} \text{ is empty, or} \\ \{\mathbf{val}(\tilde{s}) \odot t\}, & \text{otherwise;} \end{cases}$$

7218 or

$$7219 \quad \mathbf{val} \leftarrow \begin{cases} t, & \text{when accum} = \text{GrB\_NULL, or} \\ \mathbf{val} \odot t, & \text{otherwise;} \end{cases}$$

7220 In both GrB\_BLOCKING and GrB\_NONBLOCKING modes, the method exits with return value  
 7221 GrB\_SUCCESS and the new contents of the output scalar is as defined above.

#### 7222 4.3.11 transpose: Transpose rows and columns of a matrix

7223 This version computes a new matrix that is the transpose of the source matrix.

#### 7224 C Syntax

```
7225      GrB_Info GrB_transpose(GrB_Matrix      C,
7226                           const GrB_Matrix Mask,
7227                           const GrB_BinaryOp accum,
7228                           const GrB_Matrix A,
7229                           const GrB_Descriptor desc);
```

#### 7230 Parameters

7231 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
 7232 that may be accumulated with the result of the transpose operation. On output,  
 7233 the matrix holds the results of the operation.

7234 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
 7235 stored into the output matrix C. The mask dimensions must match those of the  
 7236 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
 7237 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
 7238 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
 7239 dimensions of C), GrB\_NULL should be specified.

7240 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7241 entries. If assignment rather than accumulation is desired, **GrB\_NULL** should be  
7242 specified.

7243 **A** (IN) The GraphBLAS matrix on which transposition will be performed.

7244 **desc** (IN) An optional operation descriptor. If a *default* descriptor is desired, **GrB\_NULL**  
7245 should be specified. Non-default field/value pairs are listed as follows:

7246

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.

7247

## 7248 Return Values

7249 **GrB\_SUCCESS** In blocking mode, the operation completed successfully. In non-  
7250 blocking mode, this indicates that the compatibility tests on di-  
7251 mensions and domains for the input arguments passed successfully.  
7252 Either way, output matrix C is ready to be used in the next method  
7253 of the sequence.

7254 **GrB\_PANIC** Unknown internal error.

7255 **GrB\_INVALID\_OBJECT** This is returned in any execution mode whenever one of the opaque  
7256 GraphBLAS objects (input or output) is in an invalid state caused  
7257 by a previous execution error. Call **GrB\_error()** to access any error  
7258 messages generated by the implementation.

7259 **GrB\_OUT\_OF\_MEMORY** Not enough memory available for the operation.

7260 **GrB\_UNINITIALIZED\_OBJECT** One or more of the GraphBLAS objects has not been initialized by  
7261 a call to **new** (or **Matrix\_dup** for matrix parameters).

7262 **GrB\_DIMENSION\_MISMATCH** mask, C and/or A dimensions are incompatible.

7263 **GrB\_DOMAIN\_MISMATCH** The domains of the various matrices are incompatible with the cor-  
7264 responding domains of the accumulation operator, or the mask's do-  
7265 main is not compatible with **bool** (in the case where **desc[GrB\_MASK].GrB\_STRUCTURE**  
7266 is not set).

## 7267 Description

7268 GrB\_transpose computes the result of performing a transpose of the input matrix:  $C = A^T$ ; or, if an  
 7269 optional binary accumulation operator ( $\odot$ ) is provided,  $C = C \odot A^T$ . We note that the input matrix  
 7270 A can itself be optionally transposed before the operation, which would cause either an assignment  
 7271 from A to C or an accumulation of A into C.

7272 Logically, this operation occurs in three steps:

7273     **Setup** The internal matrix and mask used in the computation are formed and their domains  
 7274             and dimensions are tested for compatibility.

7275     **Compute** The indicated computations are carried out.

7276     **Output** The result is written into the output matrix, possibly under control of a mask.

7277 Up to three matrix arguments are used in this GrB\_transpose operation:

- 7278     1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7279     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)
- 7280     3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$

7281 The argument matrices and accumulation operator (if provided) are tested for domain compatibility  
 7282 as follows:

- 7283     1. If Mask is not GrB\_NULL, and desc[GrB\_MASK].GrB\_STRUCTURE is not set, then  $\mathbf{D}(\text{Mask})$   
 7284         must be from one of the pre-defined types of Table 3.2.
- 7285     2.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}(A)$  of the input matrix.
- 7286     3. If accum is not GrB\_NULL, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
 7287         of the accumulation operator and  $\mathbf{D}(A)$  of the input matrix must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$   
 7288         of the accumulation operator.

7289 Two domains are compatible with each other if values from one domain can be cast to values in  
 7290 the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all  
 7291 compatible with each other. A domain from a user-defined type is only compatible with itself. If  
 7292 any compatibility rule above is violated, execution of GrB\_transpose ends and the domain mismatch  
 7293 error listed above is returned.

7294 From the argument matrices, the internal matrices and mask used in the computation are formed  
 7295 ( $\leftarrow$  denotes copy):

- 7296     1. Matrix  $\tilde{C} \leftarrow C$ .
- 7297     2. Two-dimensional mask,  $\tilde{M}$ , is computed from argument Mask as follows:

- 7298 (a) If  $\text{Mask} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\mathbf{C}), \mathbf{ncols}(\mathbf{C}), \{(i, j), \forall i, j : 0 \leq i < \mathbf{nrows}(\mathbf{C}), 0 \leq$   
7299  $j < \mathbf{ncols}(\mathbf{C})\} \rangle$ .
- 7300 (b) If  $\text{Mask} \neq \text{GrB\_NULL}$ ,
- 7301 i. If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_STRUCTURE}$  is set, then  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}), \{(i, j) :$   
7302  $(i, j) \in \mathbf{ind}(\text{Mask})\} \rangle$ ,
- 7303 ii. Otherwise,  $\widetilde{\mathbf{M}} = \langle \mathbf{nrows}(\text{Mask}), \mathbf{ncols}(\text{Mask}),$   
7304  $\{(i, j) : (i, j) \in \mathbf{ind}(\text{Mask}) \wedge (\text{bool})\text{Mask}(i, j) = \text{true}\} \rangle$ .
- 7305 (c) If  $\text{desc}[\text{GrB\_MASK}].\text{GrB\_COMP}$  is set, then  $\widetilde{\mathbf{M}} \leftarrow \neg \widetilde{\mathbf{M}}$ .
- 7306 3. Matrix  $\widetilde{\mathbf{A}} \leftarrow \text{desc}[\text{GrB\_INP0}].\text{GrB\_TRAN} ? \mathbf{A}^T : \mathbf{A}$ .

7307 The internal matrices and masks are checked for dimension compatibility. The following conditions  
7308 must hold:

- 7309 1.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{M}})$ .
- 7310 2.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{M}})$ .
- 7311 3.  $\mathbf{nrows}(\widetilde{\mathbf{C}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$ .
- 7312 4.  $\mathbf{ncols}(\widetilde{\mathbf{C}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$ .

7313 If any compatibility rule above is violated, execution of `GrB_transpose` ends and the dimension  
7314 mismatch error listed above is returned.

7315 From this point forward, in `GrB_NONBLOCKING` mode, the method can optionally exit with  
7316 `GrB_SUCCESS` return code and defer any computation and/or execution error codes.

7317 We are now ready to carry out the matrix transposition and any additional associated operations.  
7318 We describe this in terms of two intermediate matrices:

- 7319 •  $\widetilde{\mathbf{T}}$ : The matrix holding the transpose of  $\widetilde{\mathbf{A}}$ .
- 7320 •  $\widetilde{\mathbf{Z}}$ : The matrix holding the result after application of the (optional) accumulation operator.

7321 The intermediate matrix

$$7322 \quad \widetilde{\mathbf{T}} = \langle \mathbf{D}(\mathbf{A}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \{(j, i, A_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{A}})\} \rangle$$

7323 is created.

7324 The intermediate matrix  $\widetilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7325 • If  $\text{accum} = \text{GrB\_NULL}$ , then  $\widetilde{\mathbf{Z}} = \widetilde{\mathbf{T}}$ .
- 7326 • If  $\text{accum}$  is a binary operator, then  $\widetilde{\mathbf{Z}}$  is defined as

$$7327 \quad \widetilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\widetilde{\mathbf{C}}), \mathbf{ncols}(\widetilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid (i, j) \in \mathbf{ind}(\widetilde{\mathbf{C}}) \cup \mathbf{ind}(\widetilde{\mathbf{T}})\} \rangle.$$

7328 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
7329 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$\begin{aligned}
7330 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})), \\
7331 \\
7332 \quad Z_{ij} &= \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))), \\
7333 \\
7334 \quad Z_{ij} &= \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),
\end{aligned}$$

7335 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7336 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
7337 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
7338 mask which acts as a “write mask”.

- 7339 • If desc[GrB\_OUTP].GrB\_REPLACE is set, then any values in  $\mathbf{C}$  on input to this operation are  
7340 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7341 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7342 • If desc[GrB\_OUTP].GrB\_REPLACE is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
7343 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
7344 mask are unchanged:

$$7345 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg\tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7346 In GrB\_BLOCKING mode, the method exits with return value GrB\_SUCCESS and the new content  
7347 of matrix  $\mathbf{C}$  is as defined above and fully computed. In GrB\_NONBLOCKING mode, the method  
7348 exits with return value GrB\_SUCCESS and the new content of matrix  $\mathbf{C}$  is as defined above but  
7349 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7350 sequence.

#### 7351 4.3.12 kronecker: Kronecker product of two matrices

7352 Computes the Kronecker product of two matrices. The result is a matrix.

#### 7353 C Syntax

```

7354      GrB_Info GrB_kronecker(GrB_Matrix      C,
7355                             const GrB_Matrix  Mask,
7356                             const GrB_BinaryOp accum,
7357                             const GrB_Semiring op,
7358                             const GrB_Matrix  A,
7359                             const GrB_Matrix  B,
7360                             const GrB_Descriptor desc);
7361
```

```

7362     GrB_Info GrB_kronecker(GrB_Matrix      C,
7363                           const GrB_Matrix Mask,
7364                           const GrB_BinaryOp accum,
7365                           const GrB_Monoid  op,
7366                           const GrB_Matrix A,
7367                           const GrB_Matrix B,
7368                           const GrB_Descriptor desc);
7369
7370     GrB_Info GrB_kronecker(GrB_Matrix      C,
7371                           const GrB_Matrix Mask,
7372                           const GrB_BinaryOp accum,
7373                           const GrB_BinaryOp op,
7374                           const GrB_Matrix A,
7375                           const GrB_Matrix B,
7376                           const GrB_Descriptor desc);

```

## 7377 Parameters

7378 **C** (INOUT) An existing GraphBLAS matrix. On input, the matrix provides values  
7379 that may be accumulated with the result of the Kronecker product. On output,  
7380 the matrix holds the results of the operation.

7381 **Mask** (IN) An optional “write” mask that controls which results from this operation are  
7382 stored into the output matrix C. The mask dimensions must match those of the  
7383 matrix C. If the GrB\_STRUCTURE descriptor is *not* set for the mask, the domain  
7384 of the Mask matrix must be of type bool or any of the predefined “built-in” types  
7385 in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the  
7386 dimensions of C), GrB\_NULL should be specified.

7387 **accum** (IN) An optional binary operator used for accumulating entries into existing C  
7388 entries. If assignment rather than accumulation is desired, GrB\_NULL should be  
7389 specified.

7390 **op** (IN) The semiring, monoid, or binary operator used in the element-wise “product”  
7391 operation. Depending on which type is passed, the following defines the binary  
7392 operator,  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \otimes \rangle$ , used:

7393 BinaryOp:  $F_b = \langle \mathbf{D}_{out}(\text{op}), \mathbf{D}_{in_1}(\text{op}), \mathbf{D}_{in_2}(\text{op}), \odot(\text{op}) \rangle$ .

7394 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-  
7395 nored.

7396 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  
7397 is ignored.

7398 **A** (IN) The GraphBLAS matrix holding the values for the left-hand matrix in the  
7399 product.



7400 B (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the  
7401 product.

7402 desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB\_NULL  
7403 should be specified. Non-default field/value pairs are listed as follows:  
7404

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.

## 7406 Return Values

7407 GrB\_SUCCESS In blocking mode, the operation completed successfully. In non-  
7408 blocking mode, this indicates that the compatibility tests on di-  
7409 mensions and domains for the input arguments passed successfully.  
7410 Either way, output matrix C is ready to be used in the next method  
7411 of the sequence.

7412 GrB\_PANIC Unknown internal error.

7413 GrB\_INVALID\_OBJECT This is returned in any execution mode whenever one of the opaque  
7414 GraphBLAS objects (input or output) is in an invalid state caused  
7415 by a previous execution error. Call GrB\_error() to access any error  
7416 messages generated by the implementation.

7417 GrB\_OUT\_OF\_MEMORY Not enough memory available for the operation.

7418 GrB\_UNINITIALIZED\_OBJECT One or more of the GraphBLAS objects has not been initialized by  
7419 a call to new (or Matrix\_dup for matrix parameters).

7420 GrB\_DIMENSION\_MISMATCH Mask and/or matrix dimensions are incompatible.

7421 GrB\_DOMAIN\_MISMATCH The domains of the various matrices are incompatible with the  
7422 corresponding domains of the binary operator (op) or accumulation  
7423 operator, or the mask's domain is not compatible with bool (in the  
7424 case where desc[GrB\_MASK].GrB\_STRUCTURE is not set).

## 7425 Description

7426 GrB\_kronecker computes the Kronecker product  $C = A \otimes B$  or, if an optional binary accumulation  
7427 operator ( $\odot$ ) is provided,  $C = C \odot (A \otimes B)$  (where matrices A and B can be optionally transposed).

7428 The Kronecker product is defined as follows:

7429

$$7430 \quad C = A \otimes B = \begin{bmatrix} A_{0,0} \otimes B & A_{0,1} \otimes B & \dots & A_{0,n_A-1} \otimes B \\ A_{1,0} \otimes B & A_{1,1} \otimes B & \dots & A_{1,n_A-1} \otimes B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_A-1,0} \otimes B & A_{m_A-1,1} \otimes B & \dots & A_{m_A-1,n_A-1} \otimes B \end{bmatrix}$$

7431 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  
7432 Kronecker product are defined as

$$7433 \quad C(i_A m_B + i_B, j_A n_B + j_B) = A_{i_A, j_A} \otimes B_{i_B, j_B},$$

7434 where  $\otimes$  is the multiplicative operator specified by the **op** parameter.

7435 Logically, this operation occurs in three steps:

7436 **Setup** The internal matrices and mask used in the computation are formed and their domains  
7437 and dimensions are tested for compatibility.

7438 **Compute** The indicated computations are carried out.

7439 **Output** The result is written into the output matrix, possibly under control of a mask.

7440 Up to four argument matrices are used in the **GrB\_kronecker** operation:

- 7441 1.  $C = \langle \mathbf{D}(C), \mathbf{nrows}(C), \mathbf{ncols}(C), \mathbf{L}(C) = \{(i, j, C_{ij})\} \rangle$
- 7442 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)
- 7443 3.  $A = \langle \mathbf{D}(A), \mathbf{nrows}(A), \mathbf{ncols}(A), \mathbf{L}(A) = \{(i, j, A_{ij})\} \rangle$
- 7444 4.  $B = \langle \mathbf{D}(B), \mathbf{nrows}(B), \mathbf{ncols}(B), \mathbf{L}(B) = \{(i, j, B_{ij})\} \rangle$

7445 The argument matrices, the "product" operator (**op**), and the accumulation operator (if provided)  
7446 are tested for domain compatibility as follows:

- 7447 1. If **Mask** is not **GrB\_NULL**, and **desc[GrB\_MASK].GrB\_STRUCTURE** is not set, then  $\mathbf{D}(\text{Mask})$   
7448 must be from one of the pre-defined types of Table 3.2.
- 7449 2.  $\mathbf{D}(A)$  must be compatible with  $\mathbf{D}_{in_1}(\text{op})$ .
- 7450 3.  $\mathbf{D}(B)$  must be compatible with  $\mathbf{D}_{in_2}(\text{op})$ .
- 7451 4.  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{out}(\text{op})$ .
- 7452 5. If **accum** is not **GrB\_NULL**, then  $\mathbf{D}(C)$  must be compatible with  $\mathbf{D}_{in_1}(\text{accum})$  and  $\mathbf{D}_{out}(\text{accum})$   
7453 of the accumulation operator and  $\mathbf{D}_{out}(\text{op})$  of **op** must be compatible with  $\mathbf{D}_{in_2}(\text{accum})$  of  
7454 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.

7488 The intermediate matrix  $\tilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\text{op}), \mathbf{nrows}(\tilde{\mathbf{A}}) \times \mathbf{nrows}(\tilde{\mathbf{B}}), \mathbf{ncols}(\tilde{\mathbf{A}}) \times \mathbf{ncols}(\tilde{\mathbf{B}}), \{(i, j, T_{ij}) \text{ where } i =$   
7489  $i_A \cdot m_B + i_B, j = j_A \cdot n_B + j_B, \forall (i_A, j_A) = \mathbf{ind}(\tilde{\mathbf{A}}), (i_B, j_B) = \mathbf{ind}(\tilde{\mathbf{B}})\}$  is created. The value of  
7490 each of its elements is computed by

$$7491 \quad T_{i_A \cdot m_B + i_B, j_A \cdot n_B + j_B} = \tilde{\mathbf{A}}(i_A, j_A) \otimes \tilde{\mathbf{B}}(i_B, j_B),$$

7492 where  $\otimes$  is the multiplicative operator specified by the `op` parameter.

7493 The intermediate matrix  $\tilde{\mathbf{Z}}$  is created as follows, using what is called a *standard matrix accumulate*:

- 7494 • If `accum = GrB_NULL`, then  $\tilde{\mathbf{Z}} = \tilde{\mathbf{T}}$ .
- 7495 • If `accum` is a binary operator, then  $\tilde{\mathbf{Z}}$  is defined as

$$7496 \quad \tilde{\mathbf{Z}} = \langle \mathbf{D}_{out}(\text{accum}), \mathbf{nrows}(\tilde{\mathbf{C}}), \mathbf{ncols}(\tilde{\mathbf{C}}), \{(i, j, Z_{ij}) \mid \forall (i, j) \in \mathbf{ind}(\tilde{\mathbf{C}}) \cup \mathbf{ind}(\tilde{\mathbf{T}})\} \rangle.$$

7497 The values of the elements of  $\tilde{\mathbf{Z}}$  are computed based on the relationships between the sets of  
7498 indices in  $\tilde{\mathbf{C}}$  and  $\tilde{\mathbf{T}}$ .

$$7499 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j) \odot \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}})),$$

$$7500 \quad Z_{ij} = \tilde{\mathbf{C}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{C}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

$$7501 \quad Z_{ij} = \tilde{\mathbf{T}}(i, j), \text{ if } (i, j) \in (\mathbf{ind}(\tilde{\mathbf{T}}) - (\mathbf{ind}(\tilde{\mathbf{T}}) \cap \mathbf{ind}(\tilde{\mathbf{C}}))),$$

7502 where  $\odot = \odot(\text{accum})$ , and the difference operator refers to set difference.

7503 Finally, the set of output values that make up matrix  $\tilde{\mathbf{Z}}$  are written into the final result matrix  $\mathbf{C}$ ,  
7504 using what is called a *standard matrix mask and replace*. This is carried out under control of the  
7505 mask which acts as a “write mask”.

- 7506 • If `desc[GrB_OUTP].GrB_REPLACE` is set, then any values in  $\mathbf{C}$  on input to this operation are  
7507 deleted and the content of the new output matrix,  $\mathbf{C}$ , is defined as,

$$7510 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

- 7511 • If `desc[GrB_OUTP].GrB_REPLACE` is not set, the elements of  $\tilde{\mathbf{Z}}$  indicated by the mask are  
7512 copied into the result matrix,  $\mathbf{C}$ , and elements of  $\mathbf{C}$  that fall outside the set indicated by the  
7513 mask are unchanged:

$$7514 \quad \mathbf{L}(\mathbf{C}) = \{(i, j, C_{ij}) : (i, j) \in (\mathbf{ind}(\mathbf{C}) \cap \mathbf{ind}(\neg \tilde{\mathbf{M}}))\} \cup \{(i, j, Z_{ij}) : (i, j) \in (\mathbf{ind}(\tilde{\mathbf{Z}}) \cap \mathbf{ind}(\tilde{\mathbf{M}}))\}.$$

7515 In `GrB_BLOCKING` mode, the method exits with return value `GrB_SUCCESS` and the new content  
7516 of matrix  $\mathbf{C}$  is as defined above and fully computed. In `GrB_NONBLOCKING` mode, the method  
7517 exits with return value `GrB_SUCCESS` and the new content of matrix  $\mathbf{C}$  is as defined above but  
7518 may not be fully computed. However, it can be used in the next GraphBLAS method call in a  
7519 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 2.0.1 (Released: ## Xxxxx 2022:

- (Issue GH-69) Fix error in description of contents of matrix constructed from `GrB_Matrix_diag`.

Changes in 2.0.0 (Released: 15 November 2021:

- Reorganized Chapters 2 and 3: Chapter 2 contains prose regarding the basic concepts captured in the API; Chapter 3 presents all of the enumerations, literals, data types, and predefined objects required by the API. Made short captions for the List of Tables.
- (Issue BB-49, BB-50) Updated and corrected language regarding multithreading and completion, and requirements regarding acquire-release memory orders. Methods that used to force complete no longer do.
- (Issue BB-74, BB-9) Assigned integer values to all return codes as well as all enumerations in the API to ensure run-time compatibility between libraries.
- (Issues BB-70, BB-67) Changed semantics and signature of `GrB_wait(obj, mode)`. Added wait modes for 'complete' or 'materialize' and removed `GrB_wait(void)`. **This breaks backward compatibility.**
- (Issue GH-51) Removed deprecated `GrB_SCMP` literal from descriptor values. **This breaks backward compatibility.**
- (Issues BB-8, BB-36) Added sparse `GrB_Scalar` object and its use in additional variants of `extract/setElement` methods, and `reduce`, `apply`, `assign` and `select` operations.
- (Issues BB-34, GH-33, GH-45) Added new `select` operation that uses an index unary operator. Added new variants of `apply` that take an index unary operator (matrix and vector variants).
- (Issues BB-68, BB-51) Added `serialize` and `deserialize` methods for matrices to/from implementation defined formats.

- 7549 • (Issues BB-25, GH-42) Added import and export methods for matrices to/from API specified  
7550 formats. Three formats have been specified: CSC, CSR, COO. Dense row and column formats  
7551 have been deferred.
- 7552 • (Issue BB-75) Added matrix constructor to build a diagonal `GrB_Matrix` from a `GrB_Vector`.
- 7553 • (Issue BB-73) Allow `GrB_NULL` for dup operator in matrix and vector `build` methods. Return  
7554 error if duplicate locations encountered.
- 7555 • (Issue BB-58) Added matrix and vector methods to remove (annihilate) elements.
- 7556 • (Issue BB-17) Added `GrB_ABS_T` (absolute value) unary operator.
- 7557 • (Issue GH-46) Adding `GrB_ONEB_T` binary operator that returns 1 cast to type T (not to  
7558 be confused with the proposed unary operator).
- 7559 • (Issue GH-53) Added language about what constitutes a “conformant” implementation. Added  
7560 `GrB_NOT_IMPLEMENTED` return value (API error) for API any combinations of inputs to  
7561 a method that is not supported by the implementation.
- 7562 • Added `GrB_EMPTY_OBJECT` return value (execution error) that is used when an opaque  
7563 object (currently only `GrB_Scalar`) is passed as an input that cannot be empty.
- 7564 • (Issue BB-45) Removed language about annihilators.
- 7565 • (Issue BB-69) Made names/symbols containing underscores searchable in PDF.
- 7566 • Updated a number algorithms in the appendix to use new operations and methods.
- 7567 • Numerous additions (some changes) to the non-polymorphic interface to track changes to the  
7568 specification.
- 7569 • Typographical error in version macros was corrected. They are all caps: `GRB_VERSION` and  
7570 `GRB_SUBVERSION`.
- 7571 • Typographical change to `eWiseAdd` Description to be consistent in order of set intersections.
- 7572 • Typographical errors in `eWiseAdd`: cut-and-paste errors from `eWiseMult`/set intersection  
7573 fixed to read `eWiseAdd`/set union.
- 7574 • Typographical error (`NEQ`  $\rightarrow$  `NE`) in Description of Table 3.8.

7575 Changes in 1.3.0 (Released: 25 September 2019):

- 7576 • (Issue BB-50) Changed definition of completion and added `GrB_wait()` that takes an opaque  
7577 GraphBLAS object as an argument.
- 7578 • (Issue BB-39) Added `GrB_kronecker` operation.
- 7579 • (Issue BB-40) Added variants of the `GrB_apply` operation that take a binary function and a  
7580 scalar.

7581  
7582  
  
7583  
  
7584  
7585  
  
7586  
7587  
  
7588  
  
7589  
7590  
  
7591  
7592  
  
7593  
7594  
  
7595  
  
7596  
7597  
  
7598  
  
7599  
7600  
  
7601  
7602  
  
7603  
  
7604  
7605  
  
7606  
  
7607  
7608  
  
7609  
7610  
  
7611  
  
7612  
  
7613

- (Issue BB-59) Changed specification about how reductions to scalar (`GrB_reduce`) are to be performed (to minimize dependence on monoid identity).
- (Issue BB-24) Added methods to resize matrices and vectors (`GrB_Matrix_resize` and `GrB_Vector_resize`).
- (Issue BB-47) Added methods to remove single elements from matrices and vectors (`GrB_Matrix_removeElement` and `GrB_Vector_removeElement`).
- (Issue BB-41) Added `GrB_STRUCTURE` descriptor flag for masks (consider only the structure of the mask and not the values).
- (Issue BB-64) Deprecated `GrB_SCMP` in favor of new `GrB_COMP` for descriptor values.
- (Issue BB-46) Added predefined descriptors covering all possible combinations of field, value pairs.
- Added unary operators: absolute value (`GrB_ABS_T`) and bitwise complement of integers (`GrB_BNOT_I`).
- (Issues BB-42, BB-62) Added binary operators: Added boolean exclusive-nor (`GrB_LXNOR`) and bitwise logical operators on integers (`GrB_BOR_I`, `GrB_BAND_I`, `GrB_BXOR_I`, `GrB_BXNOR_I`).
- (Issue BB-11) Added a set of predefined monoids and semirings.
- (Issue BB-57) Updated all examples in the appendix to take advantage of new capabilities and predefined objects.
- (Issue BB-43) Added parent-BFS example.
- (Issue BB-1) Fixed bug in the non-batch betweenness centrality algorithm in Appendix C.4 where source nodes were incorrectly assigned path counts.
- (Issue BB-3) Added compile-time preprocessor defines and runtime method for querying the GraphBLAS API version being used.
- (Issue BB-10) Clarified `GrB_init()` and `GrB_finalize()` errors.
- (Issue BB-16) Clarified behavior of boolean and integer division. **Note that `GrB_MINV` for integer and boolean types was removed from this version of the spec.**
- (Issue BB-19) Clarified aliasing in user-defined operators.
- (Issue BB-20) Clarified language about behavior of `GrB_free()` with predefined objects (implementation defined)
- (Issue BB-55) Clarified that multiplication does not have to distribute over addition in a GraphBLAS semiring.
- (Issue BB-45) Removed unnecessary language about annihilators.
- (Issue BB-61) Removed unnecessary language about implied zeros.
- (Issue BB-60) Added disclaimer against overspecification.

- 7614 • Fixed miscellaneous typographical errors (such as  $\otimes$ ,  $\oplus$ ).
- 7615 Changes in 1.2.0:
- 7616 • Removed "provisional" clause.
- 7617 Changes in 1.1.0:
- 7618 • Removed unnecessary `const` from `nindices`, `nrows`, and `ncols` parameters of both `extract` and
  - 7619 `assign` operations.
  - 7620 • Signature of `GrB_UnaryOp_new` changed: order of input parameters changed.
  - 7621 • Signature of `GrB_BinaryOp_new` changed: order of input parameters changed.
  - 7622 • Signature of `GrB_Monoid_new` changed: removal of domain argument which is now inferred
  - 7623 from the domains of the binary operator provided.
  - 7624 • Signature of `GrB_Vector_extractTuples` and `GrB_Matrix_extractTuples` to add an in/out ar-
  - 7625 gument, `n`, which indicates the size of the output arrays provided (in terms of number of
  - 7626 elements, not number of bytes). Added new execution error, `GrB_INSUFFICIENT_SPACE`
  - 7627 which is returned when the capacities of the output arrays are insufficient to hold all of the
  - 7628 tuples.
  - 7629 • Changed `GrB_Column_assign` to `GrB_Col_assign` for consistency in non-polymorphic inter-
  - 7630 face.
  - 7631 • Added replace flag (`z`) notation to Table 4.1.
  - 7632 • Updated the “Mathematical Description” of the `assign` operation in Table 4.1.
  - 7633 • Added triangle counting example.
  - 7634 • Added subsection headers for `accumulate` and `mask/replace` discussions in the Description
  - 7635 sections of GraphBLAS operations when the respective text was the “standard” text (i.e.,
  - 7636 identical in a majority of the operations).
  - 7637 • Fixed typographical errors.
- 7638 Changes in 1.0.2:
- 7639 • Expanded the definitions of `Vector_build` and `Matrix_build` to conceptually use intermediate
  - 7640 matrices and avoid casting issues in certain implementations.
  - 7641 • Fixed the bug in the `GrB_assign` definition. Elements of the output object are no longer being
  - 7642 erased outside the assigned area.
  - 7643 • Changes non-polymorphic interface:
    - 7644 – Renamed `GrB_Row_extract` to `GrB_Col_extract`.

- 7645           – Renamed GrB\_Vector\_reduce\_BinaryOp to GrB\_Matrix\_reduce\_BinaryOp.
- 7646           – Renamed GrB\_Vector\_reduce\_Monoid to GrB\_Matrix\_reduce\_Monoid.
- 7647       • Fixed the bugs with respect to isolated vertices in the Maximal Independent Set example.
- 7648       • 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.

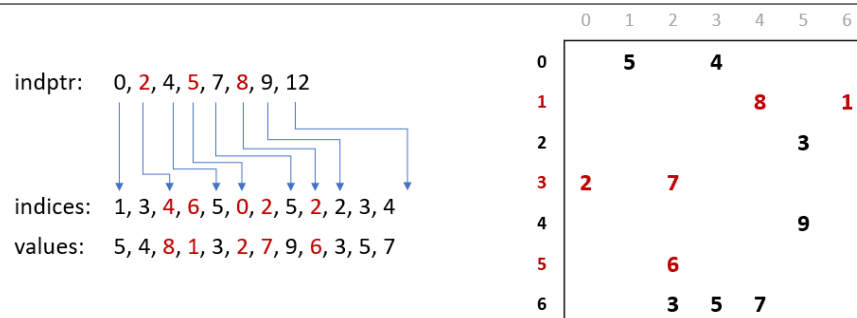


Figure B.1: Data layout for CSR format.

### 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.

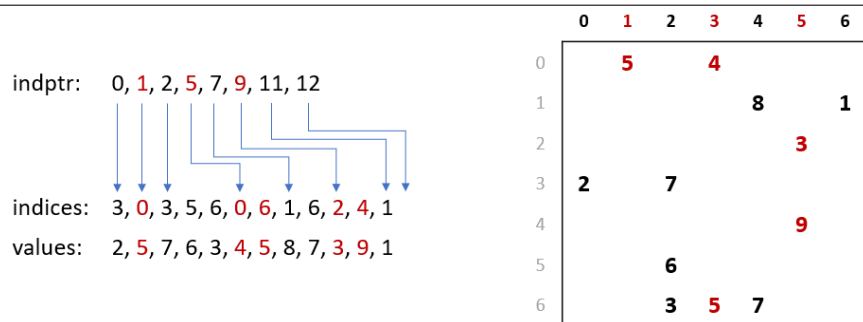


Figure B.2: Data layout for CSC format.

### 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.

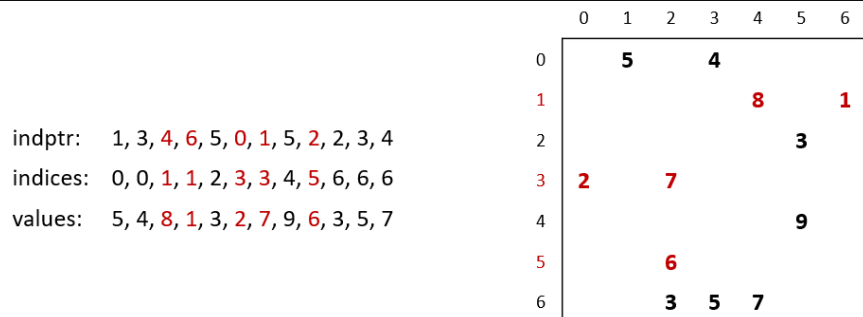


Figure B.3: Data layout for COO format.

7681 **Appendix C**

7682 **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);      //  $q[!v] = q \ || \ \&\& \ A$ ; 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                                              //  $q[s] = 1$ 
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
37      GrB_Index d = 0;                          // BFS level number
38      int32_t sum = 0;                          // sum == 0 when BFS phase is complete
39
40      do {
41          GrB_assign(sigma, GrB_NULL, GrB_NULL, q, d, GrB_ALL, n, GrB_NULL); //  $\text{sigma}[d, :] = q$ 
42          GrB_eWiseAdd(p, GrB_NULL, GrB_NULL, GrB_PLUS_INT32, p, q, GrB_NULL); // accum path counts on this level
43          GrB_vxm(q, p, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_INT32,
44                  q, A, GrB_DESC_RC);          //  $q = \#$  paths to nodes reachable
45                                              // from current level
46          GrB_reduce(&sum, GrB_NULL, GrB_PLUS_MONOID_INT32, q, GrB_NULL); // sum path counts at this level
47          ++d;
48      } while (sum);
49
50  /*
51   * BC computation phase
52   * ( $t_1, t_2, t_3, t_4$ ) are temporary vectors
53   */
54      GrB_Vector t1; GrB_Vector_new(&t1, GrB_FP32, n);
55      GrB_Vector t2; GrB_Vector_new(&t2, GrB_FP32, n);
56      GrB_Vector t3; GrB_Vector_new(&t3, GrB_FP32, n);
57      GrB_Vector t4; GrB_Vector_new(&t4, GrB_FP32, n);
58
59      for (int i=d-1; i>0; i--)
60      {
61          GrB_assign(t1, GrB_NULL, GrB_NULL, 1.0f, GrB_ALL, n, GrB_NULL); //  $t_1 = 1 + \delta$ 
62          GrB_eWiseAdd(t1, GrB_NULL, GrB_NULL, GrB_PLUS_FP32, t1, *delta, GrB_NULL);
63          GrB_extract(t2, GrB_NULL, GrB_NULL, sigma, GrB_ALL, n, i, GrB_DESC_T0); //  $t_2 = \text{sigma}[i, :]$ 
64          GrB_eWiseMult(t2, GrB_NULL, GrB_NULL, GrB_DIV_FP32, t1, t2, GrB_NULL); //  $t_2 = (1 + \delta) / \text{sigma}[i, :]$ 
65          GrB_mxv(t3, GrB_NULL, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_FP32,
66                  t3, t2, A, GrB_DESC_T0); // 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_mnv(neighbor_max, candidates, GrB_NULL, GrB_MAX_SECOND_SEMIRING_FP32, A, prob, GrB_DESC_R);
65
66 // select vertex if its probability is larger than all its active neighbors,
67 // and apply a "masked no-op" to remove stored falses
68 GrB_eWiseAdd(new_members, GrB_NULL, GrB_NULL, GrB_GT_FP64, prob, neighbor_max, GrB_NULL);
69 GrB_apply(new_members, new_members, GrB_NULL, GrB_IDENTITY_BOOL, new_members, GrB_DESC_R);
70
71 // add new members to independent set.
72 GrB_eWiseAdd(*iset, GrB_NULL, GrB_NULL, GrB_LOR, *iset, new_members, GrB_NULL);
73
74 // remove new members from set of candidates  $c = c \ominus !new$ 
75 GrB_eWiseMult(candidates, new_members, GrB_NULL,
76               GrB_LAND, candidates, candidates, GrB_DESC_RC);
77
78 GrB_Vector_nvals(&nvals, candidates);
79 if (nvals == 0) { break; } // early exit condition
80
81 // Neighbors of new members can also be removed from candidates
82 GrB_mnv(new_neighbors, candidates, GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL,
83         A, new_members, GrB_NULL);
84 GrB_eWiseMult(candidates, new_neighbors, GrB_NULL, GrB_LAND,
85               candidates, candidates, GrB_DESC_RC);
86
87 GrB_Vector_nvals(&nvals, candidates);
88 }
89
90 GrB_free(&neighbor_max); // free all objects "new'ed"
91 GrB_free(&new_members);
92 GrB_free(&new_neighbors);
93 GrB_free(&prob);
94 GrB_free(&candidates);
95 GrB_free(&set_random);
96 GrB_free(&degrees);
97
98 return GrB_SUCCESS;
99 }

```

## C.7 Example: Counting triangles in GraphBLAS

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <stdint.h>
4 #include <stdbool.h>
5 #include "GraphBLAS.h"
6
7 /*
8  * Given an  $n \times n$  boolean adjacency matrix,  $A$ , of an undirected graph, computes
9  * the number of triangles in the graph.
10 */
11 uint64_t triangle_count(GrB_Matrix A)
12 {
13     GrB_Index n;
14     GrB_Matrix_nrows(&n, A);           //  $n = \#$  of vertices
15
16     //  $L$ :  $N \times N$ , lower-triangular, bool
17     GrB_Matrix L;
18     GrB_Matrix_new(&L, GrB_BOOL, n, n);
19     GrB_select(L, GrB_NULL, GrB_NULL, GrB_TRIL, A, 0UL, GrB_NULL);
20
21     GrB_Matrix C;
22     GrB_Matrix_new(&C, GrB_UINT64, n, n);
23
24     GrB_mxm(C, L, GrB_NULL, GrB_PLUS_TIMES_SEMIRING_UINT64, L, L, GrB_NULL); //  $C \langle L \rangle = L +.* L$ 
25
26     uint64_t count;
27     GrB_reduce(&count, GrB_NULL, GrB_PLUS_MONOID_UINT64, C, GrB_NULL); // 1-norm of  $C$ 
28
29     GrB_free(&C);
30     GrB_free(&L);
31
32     return count;
33 }
```